インテル® MKL

クックブック

インテル® MKL

資料番号: 330244-006JA

目次

著作権と商標について

本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスも許諾するものではありません。

インテルは、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、知的財産権の非侵害性への保証、およびインテル製品の性能、取引、使用から生じるいかなる保証を含みますが、これらに限定されるものではありません。

本資料には、開発中の製品、サービスおよびプロセスについての情報が含まれています。 本資料に含まれる情報は予告なく変更されることがあります。 最新の予測、スケジュール、仕様、ロードマップについては、インテルの担当者までお問い合わせください。

本資料で説明されている製品およびサービスには、不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。

Intel、インテル、Intel ロゴ、Xeon、Intel Xeon Phi、VTune は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

Microsoft、Visual Studio、および Windows は、米国 Microsoft Corporation の、米国およびその他の国における登録商標または商標です。

Java は、Oracle および / または関連会社の登録商標です。

サードパーティー・コンテンツ

インテル® マス・カーネル・ライブラリー (インテル® MKL) には、いくつかのサードパーティー提供のコンテンツが含まれており、それぞれ以下のライセンス規約が適用されます (敬称略)。

- Portions© Copyright 2001 Hewlett-Packard Development Company, L.P.
- Linear Algebra PACKage (LAPACK) ルーチンのセクションには、著作権で保護された派生物の一部が含まれています。

  © 1991, 1992, and 1998 by The Numerical Algorithms Group, Ltd.
- インテル® MKL は、以下のライセンスの下で LAPACK 3.5 の計算、ドライバー、および補助ルーチンをサポートしています。

  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.

インテル® MKL の一部の基となった LAPACK の原版は http://www.netlib.org/lapack/index.html (英語) から入手できます。LAPACK の開発は、E. Anderson、Z. Bai、C. Bischof、S. Blackford、J. Demmel、J. Dongarra、J. Du Croz、A. Greenbaum、S. Hammarling、A. McKenney、D. Sorensen らによって行われました。

- インテル® MKL の一部の基となった BLAS の原版は http://www.netlib.org/blas/index.html (英語) から入手できます。
- XBLAS は、以下の著作権の下で配布されています。

  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.
- インテル® MKL の一部の基となった BLACS の原版は http://www.netlib.org/blacs/index.html (英語) から入手できます。BLACS の開発は、Jack Dongarra と R. Clint Whaley によって行われました。
- インテル® MKL の一部の基となった ScaLAPACK の原版は http://www.netlib.org/scalapack/index.html (英語) から入手できます。ScaLAPACK の開発は、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、R. C. Whaley らによって行われました。

- インテル® MKL の一部の基となった PBLAS の原版は http://www.netlib.org/scalapack/html/pblas_qref.html (英語) から入手できます。
- インテル® MKL の PARDISO* (PARallel DIrect SOlver) の開発は、バーゼル大学のコンピューター・サイエンス学部 (http://www.unibas.ch (英語)) によって行われました。http://www.pardiso-project.org (英語) から入手できます。
- 拡張固有値ソルバーは、Feast ソルバーパッケージを基にしており、以下のライセンスの下で配布されています。

  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.
- 本リリースのインテル® MKL の一部の高速フーリエ変換 (FFT) 関数は、カーネギーメロン大学からライセンスを受けて、SPIRAL ソフトウェア生成システム (http://www.spiral.net/ (英語)) によって生成されました。SPIRAL の開発は、Markus Püschel、José Moura、Jeremy Johnson、David Padua、Manuela Veloso、Bryan Singer、Jianxin Xiong、Franz Franchetti、Aca Gacic、Yevgen Voronenko、Kang Chen、Robert W. Johnson、Nick Rizzolo らによって行われました。
- Open MPI は、以下の New BSD ライセンスの下で配布されています。

  このリリースの多くのファイルには、ファイルを編集した組織の著作権が適用されます。 以下の著作権は順不同で、一般に、このリリースに貢献したコードを所有する Open MPI コアチームのメンバーを反映しています。ほかの組織の許可の下で使用しているコードの著作権は、対応するファイルに含まれています。

  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.

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.

- Safe C Library は、以下の著作権の下で配布されています。

  Copyright ©

  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 の著作権情報とライセンス使用許諾条件

  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.

ヘルプとサポートについて

インテル® MKL 製品の Web サイトでは、製品機能、ホワイトペーパー、技術資料など、製品に関する最新かつ全般的な情報を提供しています。最新情報は、http://www.intel.com/software/products/support (英語) を参照してください。

インテルでは、使い方のヒント、既知の問題点、製品のエラッタ、ライセンス情報、ユーザーフォーラムなどの多くのセルフヘルプ情報を含むサポート Web サイトを提供しています。詳しくは、http://www.intel.com/software/products/ (英語) を参照してください。

登録を行うことで、サポートサービス期間中 (通常は 1 年間)、製品アップデートとテクニカルサポートが提供されます。インテル® プレミアサポート Web サイトでは、次のサービスを利用できます。

- 問題の送信とステータスの確認
- 製品アップデートのダウンロード

製品の登録、製品サポートの利用、インテルへの問い合わせは、次のサイトにアクセスしてください: http://www.intel.com/software/products/support (英語)。

新機能

このバージョンのクックブックには、以下のレシピが追加されました。

- 「Python* 科学計算の高速化」では、インテル® MKL を使用して NumPy* および SciPy* ソースをビルドし、インテル® MKL の自動オフロードを有効にして Python* コードのパフォーマンスを高速化します。
- 「ヒストスプライン手法を使用した画像スケーリング」では、ヒストスプライン計算の画像スケーリングおよびスプライン補間にインテル® MKL のデータ・フィッティング関数を使用します。

また、方程式のグラフィックも改良されました。

表記規則

このマニュアルでは、以下のように対象となるオペレーティング・システムを指しています。

Windows®	サポートしているすべての Windows® オペレーティング・システムで有効な情報を指します。
Linux*	サポートしているすべての Linux* オペレーティング・システムで有効な情報を指します。
OS X*	OS X* オペレーティング・システムを実行しているインテル® プロセッサー・ベースのシステムで有効な情報を指します。

このマニュアルでは、次の表記規則を使用しています。

- ルーチン名の省略形 (例えば、`cungqr`/`zungqr` の代わりに `?ungqr`)
- テキストとコードを区別するためのフォントの表記規則

ルーチン名の省略形

省略形 では、疑問符 "`?`" を含む名前は同様の機能を備えたグループのルーチンを表します。各グループは通常、使用されるルーチンと 4 つの基本的なデータ型 (単精度実数、倍精度実数、単精度複素数、倍精度複素数) から成ります。疑問符は関数の任意またはすべての種類を示します。次に例を示します。

`?swap`	ベクトル-ベクトル `?swap` ルーチンの 4 つのデータ型すべて (`sswap`、`dswap`、`cswap`、`zswap`) を指します。

フォントの表記規則

以下の フォントの表記規則が使用されています。

大文字の `COURIER`	Fortran インターフェイスの入出力パラメーターの記述で使用されるデータ型。例: `CHARACTER*1`。
小文字の `courier`	コードサンプル: `a(k+i,j) = matrix(i,j)` および C インターフェイスのデータ型。例: `const float*`。
小文字と大文字の `courier`	C インターフェイスの関数名。例: `vmlSetMode`。
小文字斜体の `courier`	引数およびパラメーター記述の変数。例: *incx*。

*	コードサンプルや方程式の乗算記号として、またプログラミング言語の構文で必要な個所に使用されます。

関連情報

アプリケーションでライブラリーを使用する方法については、このドキュメントのほか、次のドキュメントも併せて参照してください。

- インテル® MKL リファレンス・マニュアル - ルーチンの機能、パラメーターの説明、インターフェイス、呼び出し構文と戻り値についてのリファレンス情報を提供します。
- インテル® MKL デベロッパー・ガイド。

これらのドキュメントの Web バージョンは、インテル® ソフトウェア・ドキュメント・ライブラリー (英語) から入手できます。

インテル® MKLのレシピ

インテル® マス・カーネル・ライブラリー (インテル® MKL) には、行列を乗算する、連立方程式を解く、フーリエ変換を行うなど、さまざまな数値問題を解く際に役立つ多くのルーチンが含まれています。専用のインテル® MKL ルーチンが用意されていない問題については、インテル® MKL で提供されているビルディング・ブロックを組み合わせることにより問題を解くことができます。

インテル® MKL クックブックには、より複雑な問題を解くためにインテル® MKL ルーチンを組み合わせる際に役立つ手法が含まれています。

- 行列のレシピ (インテル® MKL PARDISO、BLAS、スパース BLAS、LAPACK ルーチンを使用)
  - 「定常非線形熱伝導方程式の近似解を求める」では、インテル® MKL PARDISO、BLAS、スパース BLAS ルーチンを使用して非線形方程式の解を求める手法を紹介します。
  - 「ブロック三重対角行列の因数分解」では、BLAS と LAPACK ルーチンのインテル® MKL 実装を使用します。
  - 「LU 因数分解されたブロック三重対角係数行列を含む連立線形方程式を解く」では、因数分解レシピを拡張して連立方程式を解きます。
  - 「ブロック三重対称正定値行列の因数分解」では、BLAS と LAPACK ルーチンを使用した対称正定値ブロック三重行列のコレスキー因数分解を説明します。
  - 「ブロック三重対称正定値係数行列を含む連立線形方程式を解く」では、BLAS と LAPACK ルーチンを使用して連立方程式を解く因数分解レシピを拡張します。
  - 「2 つの部分空間の間の主角度の計算」では、LAPACK SVD を使用して主角度を計算します。
  - 「ブロック三角行列の不変部分空間の間の主角度の計算」では、LAPACK SVD の使用を部分空間がブロック三角行列の不変部分空間で互いに補完する場合に拡張します。
- 高速フーリエ変換のレシピ
  - 「フーリエ積分の評価」では、インテル® MKL 高速フーリエ変換 (FFT) インターフェイスを使用して連続するフーリエ変換積分を評価します。
  - 「高速フーリエ変換を使用したコンピューター・トモグラフィー・イメージの復元」では、インテル® MKL FFT インターフェイスを使用してコンピューター・トモグラフィー・データからイメージを復元します。
- 数値演算のレシピ
  - 「金融市場のデータストリームにおけるノイズ・フィルタリング」では、インテル® MKL サマリー統計ルーチンを使用してストリーミング・データの相関行列を計算します。
  - 「モンテカルロ法を使用したヨーロピアン・オプションの価格計算」では、インテル® MKL の基本乱数ジェネレーター (BRNG) を利用してヨーロピアン・オプション (コールおよびプット) の価格を計算します。
  - 「ブラックショールズ方程式を使用したヨーロピアン・オプションの価格計算」では、インテル® MKL ベクトルマス関数を利用してブラックショールズ方程式を使用したヨーロピアン・オプションの価格計算をスピードアップします。
  - 「置換のない複数の単純なランダム・サンプリング」では、大きな K についてサイズ N の母集団から置換なしで K の単純なランダム長 M のサンプルを生成します。
  - 「ヒストスプライン手法を使用した画像スケーリング」では、ヒストスプライン計算の画像スケーリングおよびスプライン補間にインテル® MKL のデータ・フィッティング関数を使用します。
- 異なるプログラミング環境でインテル® MKL を使用するためのレシピ

- 「Python* 科学計算の高速化」では、インテル® MKL を使用して NumPy* および SciPy* ソースをビルドし、インテル® MKL の自動オフロードを有効にして Python* コードのパフォーマンスを高速化します。

注

クックブックのコードサンプルは、Fortran または C で提供されています。

最適化に関する注意事項
インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。 注意事項の改訂 #20110804

定常非線形熱伝導方程式の近似解を求める

1

目的

熱方程式の境界値問題の解と、その解に依存する熱係数を得る。

ソリューション

固定小数点の反復アプローチ [Amos10] を使用し、インテル® MKL PARDISO によって各外部反復の線形問題を解きます。

1. CSR (Compressed Sparse Rows: 圧縮スパース行) 形式で行列構造を設定します。
2. 残差のノルムが許容差未満になるまで固定小数点の反復を行います。
   - **a.** `pardiso` ルーチンを使用して現在の反復の線形化されたシステムを解きます。
   - **b.** `dcopy` ルーチンを使用して、システムの解を主方程式の次の近似に設定します。
   - **c.** 新しい近似に基づいて、行列の新しい要素を計算します。
   - **d.** `mkl_cspblas_dcsrgemv` ルーチンを使用して現在の解の残差を計算します。
   - **e.** `dnrm2` ルーチンを使用して残差のノルムを計算し、許容差と比較します。
3. ソルバーの内部メモリーを解放します。

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `sparse` フォルダーを参照してください。

インテル® MKL PARDISO、スパース BLAS、BLAS を使用して近似解を求める

```
CONSTRUCT_MATRIX_STRUCTURE (nx, ny, nz, &ia, &ja, &a, &error);
CONSTRUCT_MATRIX_VALUES (nx, ny, nz, us, ia, ja, a, &error);
DO WHILE res > tolerance
  phase = 13;
  PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, &idum,
    &nrhs, iparm, &msglvl, f, u, &error );
  DCOPY (&n, u, &one, u_next, &one);
  CONSTRUCT_MATRIX_VALUES (nx, ny, nz, u_next, ia, ja, a, &error);
  MKL_CSPBLAS_DCSRGEMV (uplo, &n, a, ia, ja, u, temp );
  DAXPY (&n, &minus_one, f, &one, temp, &one);
  res = DNRM2 (&n, temp, &one);
END DO
phase = -1;
PARDISO ( pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, &idum,
  &nrhs, iparm, &msglvl, f, u, &error );
```

使用するルーチン

タスク	ルーチン	説明
現在の反復の線形化されたシステムを解き、ソルバーの内部メモリーを解放する	PARDISO	複数の右辺を備えた 1 セットのスパース線形方程式の解を計算します。
見つかった解を主方程式の次の近似として設定する	DCOPY	ベクトルを別のベクトルにコピーします。
現在の非線形反復の残差を計算する	MKL_CSPBLAS_DCSRGEMV	ゼロベースでインデックスされ、CSR 形式で格納されたスパース一般行列の行列-ベクトル積を計算します。
	DAXPY	ベクトル-スカラー積を計算して結果をベクトルに追加します。
残差のノルムを計算し、停止条件と比較する	DNRM2	ベクトルのユークリッド・ノルムを計算します。

説明

定常非線形熱伝導方程式は、非線形偏微分方程式の境界値問題として説明できます。

$$-\frac{\partial}{\partial x}\left(\mu(v)\frac{\partial v}{\partial x}\right)-\frac{\partial}{\partial y}\left(\mu(v)\frac{\partial v}{\partial y}\right)-\frac{\partial}{\partial z}\left(\mu(v)\frac{\partial v}{\partial z}\right)=1,\quad (x,y,z)\in D$$

$$v|_{\partial D}=0$$

ここで、領域 *D* は立方体であると仮定します: $D=(0,1)^3$。$v(x,y,z)$ は温度の未知関数です。

デモ目的のため、問題は解の熱係数の線形従属性に制限されています。

$$\mu(v)=1+10v$$

数の解を得るため、領域 *D* でグリッドステップ *h* の等距離のグリッドが選択され、偏微分方程式は差分を使用して近似されます。 このプロシージャーから [Smith86] 非線形代数方程式のシステムが得られます。

$$-u_{i,j,k-1} * \frac{m_{i,j,k-1}+m_{i,j,k}}{2} + u_{i,j,k} * \frac{m_{i,j,k-1}+2m_{i,j,k}+m_{i,j,k+1}}{2} - u_{i,j,k+1} * \frac{m_{i,j,k}+m_{i,j,k+1}}{2}$$
$$-u_{i,j-1,k} * \frac{m_{i,j-1,k}+m_{i,j,k}}{2} + u_{i,j,k} * \frac{m_{i,j-1,k}+2m_{i,j,k}+m_{i,j+1,k}}{2} - u_{i,j+1,k} * \frac{m_{i,j,k}+m_{i,j+1,k}}{2}$$
$$-u_{i-1,j,k} * \frac{m_{i-1,j,k}+m_{i,j,k}}{2} + u_{i,j,k} * \frac{m_{i-1,j,k}+2m_{i,j,k}+m_{i+1,j,k}}{2} - u_{i+1,j,k} * \frac{m_{i,j,k}+m_{i+1,j,k}}{2}$$
$$= h^2$$

$$u_{0,j,k} = u_{n,j,k} = u_{i,0,k} = u_{i,n,k} = u_{i,j,0} = u_{i,j,n} = 0$$

$$m_{i,j,k} = 1 + 10 * u_{i,j,k}$$

$$i = \overline{1,n}; j = \overline{1,n}; k = \overline{1,n}; n = \frac{1}{h}$$

各方程式は、7 つのグリッドポイントで、未知グリッド関数 *u* の値とそれぞれの右辺の値を関連付けます。方程式の左辺は、解に依存するグリッド関数値と係数の線形の組み合わせとして表せます。これらの係数から構成される行列を利用すると、方程式をベクトル-行列形式で書き直すことができます。

$$A(\tilde{u})\tilde{u} = g$$

係数行列 *A* は疎である (各行に非ゼロ要素が 7 つしかない) ため、この反復アルゴリズムで解くために、行列を CSR 形式の配列に格納して (『インテル® MKL デベロッパー・リファレンス』の「スパース行列格納形式」を参照)、PARDISO* ソルバーを使用することは適切です。

1. u を初期値 u^0 に設定します。
2. 残差 $r = A(u)u - g$ を計算します。
3. $||r||$ < 許容差の場合:
   - **a.** 方程式 $A(u)w = g$ を w について解きます。
   - **b.** $u = w$ を設定します。
   - **c.** 残差 $r = A(u)u - g$ を計算します。

一般的なブロック三重対角行列の因数分解

2

目的

一般的なブロック三重対角行列の LU 因数分解を行う。

ソリューション

インテル® MKL LAPACK は、密行列、帯行列、三重対角行列を含む、一般的な行列の LU 因数分解用のさまざまなサブルーチンを提供しています。この手法は、すべてのブロックが正方形で同位という条件を前提として、一般的なブロック三重対角行列の機能の範囲を拡張します。

サイズ *NB* x *NB* の正方形ブロックでブロック三重対角行列の LU 因数分解を行うには:

1. 部分 LU 因数分解を、行列の最初の 2 つのブロック行と最初の 3 つのブロック列 ($M = 2NB$、$N = 3NB$、$K = NB$) で構成されたサイズ M x N の長方形ブロックにシーケンシャルに適用し、最後の 1 つのブロック列が処理されるまで対角線に沿って下に移動します。

   部分 LU 因数分解: 一般的なブロック三重対角行列の LU 因数分解では、長方形の M x N 行列の部分 LU 因数分解用に別の機能が用意されていると便利です。パラメーター K (ここで、$K \leq \min(M, N)$) の部分 LU 因数分解アルゴリズムは、次のステップからなります。

   a. M x K の部分行列の LU 因数分解を実行します。

   b. 三角係数行列を含む方程式を解きます。

   c. 右下の (M - K) x (N - K) ブロックを更新します。

   生成される行列は $A = P(LU + A1)$ です。ここで、L は下台形 M x K 行列、U は上台形行列、P は置換 (ピボット) 行列、$A1$ は最後の M - K 行と N - K 列の交差領域のみの非ゼロ要素行列です。

2. 一般的な LU 因数分解を最後の (2*NB*) x (2*NB*) ブロックに適用します。

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `BlockTDS_GE/source/dgeblttrf.f` ファイルを参照してください。

部分 LU 因数分解の実行

```
      SUBROUTINE PTLDGETRF(M, N, K, A, LDA, IPIV, INFO)
      …
          CALL DGETRF( M, K, A, LDA, IPIV, INFO )
          …
          DO I=1,K
              IF(IPIV(I).NE.I)THEN
                  CALL DSWAP(N-K, A(I,K+1), LDA, A(IPIV(I), K+1), LDA)
              END IF
          END DO
          CALL DTRSM('L','L','N','U',K,N-K,1D0, A, LDA, A(1,K+1), LDA)
          CALL DGEMM('N', 'N', M-K, N-K, K, -1D0, A(K+1,1), LDA,
     &            A(1,K+1), LDA, 1D0, A(K+1,K+1), LDA)
          …
```

ブロック三重対角行列の因数分解

```
      …
      DO K=1,N-2
C ブロック構造の 2*NB x 3*NB の部分行列 A を構成
C     (D_K   C_K 0    )
C     (B_K D_K+1 C_K+1)
      …
C 部分行列を部分的に因数分解
          CALL PTLDGETRF(2*NB, 3*NB, NB, A, 2*NB, IPIV(1,K), INFO)
C 因数分解の結果を三重対角行列のブロックを格納する配列に戻す
      …
      END DO
C ループを抜けて最後の 2*NB x 2*NB の部分行列を因数分解
      CALL DGETRF(2*NB, 2*NB, A, 2*NB, IPIV(1,N-1), INFO)
C 最後の結果を三重対角行列のブロックを格納する配列に戻す
      …
```

使用するルーチン

タスク	ルーチン	説明
M x *K* の部分行列の LU 因数分解	DGETRF	一般的な m x n 行列の LU 因数分解を計算します。
行列の行の置換	DSWAP	2 つのベクトルをスワップします。
三角係数行列を含む方程式を解く	DTRSM	三角行列を含む方程式を解きます。
右下の (*M* - *K*) x (*N* - *K*) ブロックを更新する	DGEMM	一般的な行列を含む行列-行列の積を計算します。

説明

部分 LU 因数分解では、A を長方形の *m* x *n* 行列にします。

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,k-1} & a_{1,k} & a_{1,k+1} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,k-1} & a_{2,k} & a_{2,k+1} & \cdots & a_{2,n-1} & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,k-1} & a_{3,k} & a_{3,k+1} & \cdots & a_{3,n-1} & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \iddots & \vdots & \vdots \\ a_{k-1,1} & a_{k-1,2} & a_{k-1,3} & \cdots & a_{k-1,k-1} & a_{k-1,k} & a_{k-1,k+1} & \cdots & a_{k-1,n-1} & a_{k-1,n} \\ a_{k,1} & a_{k,2} & a_{k,3} & \cdots & a_{k,k-1} & a_{k,k} & a_{k,k+1} & \cdots & a_{k,n-1} & a_{k,n} \\ a_{k+1,1} & a_{k+1,2} & a_{k+1,3} & \cdots & a_{k+1,k-1} & a_{k+1,k} & a_{k+1,k+1} & \cdots & a_{k+1,n-1} & a_{k+1,n} \\ \vdots & \vdots & \vdots & \iddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,1} & a_{m-1,2} & a_{m-1,3} & \cdots & a_{m-1,k-1} & a_{m-1,k} & a_{m-1,k-1} & \cdots & a_{m-1,n-1} & a_{m-1,n} \\ a_{m,1} & a_{m,2} & a_{m,3} & \cdots & a_{m,k-1} & a_{m,k} & a_{m,k+1} & \cdots & a_{m,n-1} & a_{m,n} \end{pmatrix}$$

注

読みやすいように、ここでは *m*、*n*、*k*、*nb* のように小文字のインデックスを使用しています。これらのインデックスは、Fortran ソリューションおよびコードサンプルで使用されている大文字のインデックスに相当します。

行列は、*m* x *k* (ここで、0 < *k*≤*n*) の部分行列の LU 因数分解を使用して分解できます。このアプリケーションでは、*m*≤*k*≤*n* または *n* = *k*≤*m* の場合は行列の因数分解に `?getrf` を直接使用できるため、*k*< min(*m*, *n*) です。

A はブロック行列として表現できます。

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

ここで、A_{11} は *k* x *k* の部分行列、A_{12} は *k* x (*n* - *k*) の部分行列、A_{21} は (*m* - *k*) x *k* の部分行列、A_{22} は (*m* - *k*) x (*n* - *k*) の部分行列です。

m x *k* のパネル A_1 は次のように定義できます。

$$A_1 = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$$

A_1 は (`?getrf` を使用して) $A_1 = PLU$ として LU 因数分解できます。ここで、*P* は置換 (ピボット) 行列、*L* は対角要素を含む下台形行列、*U* は上三角行列です。

$$L = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ l_{2,1} & 1 & 0 & \cdots & 0 & 0 \\ l_{3,1} & l_{3,2} & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{k-1,1} & l_{k-1,2} & l_{i-1,3} & \cdots & 1 & 0 \\ l_{k,1} & l_{k,2} & l_{k,3} & \cdots & l_{k,k-1} & 1 \\ l_{k+1,1} & l_{k+1,2} & l_{k+1,3} & \cdots & l_{k+1,k-1} & l_{k+1,k} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{m-1,1} & l_{m-1,2} & l_{m-1,3} & \cdots & l_{m-1,k-1} & l_{m-1,k} \\ l_{m,1} & l_{m,2} & l_{m,3} & \cdots & l_{m,k-1} & l_{m,k} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix}$$

$$U = \begin{pmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,k-1} & u_{1,k} \\ 0 & u_{2,2} & u_{2,3} & \cdots & u_{2,k-1} & u_{2,k} \\ 0 & 0 & u_{3,3} & \cdots & u_{3,k-1} & u_{3,k} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & u_{k-1,k-1} & u_{k-1,k} \\ 0 & 0 & 0 & \cdots & 0 & u_{k,k} \end{pmatrix}$$

注

L の対角要素は格納する必要がないため、A_1 を格納するために使用する配列を、L と U の要素を格納するために使用できます。

P^{T} を A の 2 つ目のパネルに適用すると次のようになります。

$$P^{\mathrm{T}}A_2 = P^{\mathrm{T}}\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} = \begin{pmatrix} A'_{12} \\ A'_{22} \end{pmatrix}$$

次の方程式が得られます。

$$P^{\mathrm{T}}A = \begin{pmatrix} L_{11}U & A'_{12} \\ L_{21}U & A'_{22} \end{pmatrix}$$

A''_{12} を次のように定義します。

$$A''_{12} = L_{11}^{-1}A'_{12}$$

$P^{\mathrm{T}}A$ の方程式を変更します。

$$\begin{aligned} P^{\mathrm{T}}A &= \begin{pmatrix} L_{11}U & L_{11}A''_{12} \\ L_{21}U & A'_{22} \end{pmatrix} \\ \square &= \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix}\begin{pmatrix} U & A''_{12} \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & A'_{22} - L_{21}A''_{12} \end{pmatrix} \end{aligned}$$

前の方程式と P を乗算すると次のようになります。

$$A = P\left(\begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix}\begin{pmatrix} U & L_{11}^{-1}A'_{12} \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & A'_{22} - L_{21}L_{11}^{-1}A'_{12} \end{pmatrix}\right)$$

この方程式は最初の行列の部分 LU 因数分解と考えることができます。

注

- 積 $L^{-1}{}_{11}A'_{12}$ は `?trsm` を呼び出して計算し、A_{12} に使用される配列の代わりに格納できます。 更新 A'_{22} - $L_{21}(L^{-1}{}_{11}A'_{12})$ は `?gemm` を呼び出して計算し、A_{22} に使用される配列の代わりに格納できます。
- 部分行列にすべてのランクが含まれない場合、LU 因数分解に失敗するため、このメソッドは適用できません。
- 一般的な行列の LU 因数分解とは異なり、下記に示す一般的なブロック三重対角行列の因数分解 $A = LU$ は、$A = PLU$ (P は置換行列) 形式で記述できません。ピボットのため、左の係数 L は置換を含みます。また、右の係数 U も複雑になります (2 つの対角線ではなく 3 つの対角線を含む)。

ロック三重対角行列の LU 因数分解では、A を、すべてのブロックが正方形で同位のブロック三重対角行列 n_b にします。

$$A = \begin{pmatrix} D_1 & C_1 & 0 & \cdots & 0 & 0 & 0 \\ B_1 & D_2 & C_2 & \cdots & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N \end{pmatrix}$$

行列は $A = LU$ として因数分解されます。

最初に、2 x 3 ブロックの部分行列を考えます。

$$\begin{pmatrix} D_1 & C_1 & 0 \\ B_1 & D_2 & C_2 \end{pmatrix}$$

この部分行列は次のように分解できます。

$$P_1^{\mathrm{T}}\begin{pmatrix} D_1 & C_1 & 0 \\ B_1 & D_2 & C_2 \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} & U_{13} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & D'_2 & C'_2 \end{pmatrix}$$

この分解は前述の部分 LU 因数分解を適用して取得できます。 ここで、$P^{\mathrm{T}}{}_1$ は n_b 要素の順列の積であり、$2n_b$ x $2n_b$ 行列として表現できます。

$$P_1^{\mathrm{T}} = \begin{pmatrix} P_{11}^1 & P_{12}^1 \\ P_{21}^1 & P_{22}^1 \end{pmatrix}$$

すべてのブロックがサイズ n_b x n_b の N x N ブロック行列を適用します。

$$\begin{pmatrix} P_{11}^1 & P_{12}^1 & 0 & \cdots & 0 \\ P_{21}^1 & P_{22}^1 & 0 & \cdots & 0 \\ 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & I \end{pmatrix}$$

分解は次のように表現できます。

$$\begin{pmatrix} P_{11}^1 & P_{12}^1 & 0 & \cdots & 0 \\ P_{21}^1 & P_{22}^1 & 0 & \cdots & 0 \\ 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & I \end{pmatrix} \begin{pmatrix} D_1 & C_1 & 0 & \cdots & 0 & 0 & 0 \\ B_1 & D_2 & C_2 & \cdots & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N \end{pmatrix}$$

$$= \begin{pmatrix} L_{11} \\ L_{21} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} & 0 & \cdots & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & D'_2 & C'_2 & \cdots & 0 & 0 & 0 \\ 0 & B_2 & D_3 & C_3 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N \end{pmatrix}$$

次に、方程式の右辺の行列の、2 番目と 3 番目の行の 2 x 3 ブロック行列を因数分解します。

$$P_2^{\mathrm{T}} \begin{pmatrix} D'_2 & C'_2 & 0 \\ B_2 & D_3 & C_3 \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} \begin{pmatrix} U_{22} & U_{23} & U_{24} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & D'_3 & C'_3 \end{pmatrix}$$

ここで、$P^{\mathrm{T}}{}_2$ は次のように定義されます。

$$P_2^{\mathrm{T}} = \begin{pmatrix} P_{22}^2 & P_{23}^2 \\ P_{32}^2 & P_{33}^2 \end{pmatrix}$$

分解は次のようになります。

$$\begin{pmatrix} I & 0 & 0 & 0 & \cdots & 0 \\ 0 & P_{11}^1 & P_{12}^1 & 0 & \cdots & 0 \\ 0 & P_{21}^1 & P_{22}^1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & I \end{pmatrix} \begin{pmatrix} P_{11}^1 & P_{12}^1 & 0 & 0 & \cdots & 0 \\ P_{21}^1 & P_{22}^1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & I & 0 & \cdots & 0 \\ 0 & 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & I \end{pmatrix} \begin{pmatrix} D_1 & C_1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ B_1 & D_2 & C_2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & B_2 & D_3 & C_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N \end{pmatrix}$$

$$= \begin{pmatrix} I & 0 & 0 & 0 & \cdots & 0 \\ 0 & P_{11}^1 & P_{12}^1 & 0 & \cdots & 0 \\ 0 & P_{21}^1 & P_{22}^1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & I \end{pmatrix} \begin{pmatrix} L_{11} \\ L_{21} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} & 0 & \cdots & 0 \end{pmatrix}$$

$$+ \begin{pmatrix} I & 0 & 0 & 0 & \cdots & 0 \\ 0 & P_{11}^1 & P_{12}^1 & 0 & \cdots & 0 \\ 0 & P_{21}^1 & P_{22}^1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & I \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & D'_2 & C'_2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & B_2 & D_3 & C_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N \end{pmatrix}$$

$$= \begin{pmatrix} I & 0 & 0 & 0 & \cdots & 0 \\ 0 & P_{11}^1 & P_{12}^1 & 0 & \cdots & 0 \\ 0 & P_{21}^1 & P_{22}^1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & I \end{pmatrix} \begin{pmatrix} L_{11} \\ L_{21} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} & 0 & \cdots & 0 \end{pmatrix} + \begin{pmatrix} 0 \\ L_{22} \\ L_{32} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} 0 & U_{22} & U_{23} & U_{24} & 0 & \cdots & 0 \end{pmatrix}$$

$$+ \begin{pmatrix} 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & D'_3 & C'_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N \end{pmatrix}$$

この表記をピボット行列に適用して方程式を単純化します。

$$\Pi_j^{\mathrm{T}} = \begin{pmatrix} I & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \iddots & \vdots \\ 0 & \cdots & I & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & P^j_{j,j} & P^j_{j,j+1} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & P^j_{j+1,j} & P^j_{j+1,j+1} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & I & \cdots & 0 \\ \vdots & \iddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & I \end{pmatrix}, \quad j = 1, 2, \ldots, N-1$$

$$P_j^{\mathrm{T}} = \begin{pmatrix} P^j_{j,j} & P^j_{j,j+1} \\ P^j_{j+1,j} & P^j_{j+1,j+1} \end{pmatrix}$$

ここで、P^{T}_j は $2n_b$ x $2n_b$ で j 番目と (j+1) 番目の行と列の交差領域にあります。分解は次のように簡潔に表記されます。

$$\Pi_2^{\mathrm{T}}\Pi_1^{\mathrm{T}}\begin{pmatrix} D_1 & C_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ B_1 & D_2 & C_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & B_2 & D_3 & C_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & 0 & 0 & 0 & B_{N-1} & D_N \end{pmatrix}$$

$$= \Pi_2^{\mathrm{T}}\begin{pmatrix} L_{11} \\ L_{21} \\ 0 \\ \vdots \\ 0 \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} & U_{13} & 0 & \cdots & 0 \end{pmatrix} + \begin{pmatrix} 0 \\ L_{22} \\ L_{32} \\ 0 \\ \vdots \\ 0 \end{pmatrix}\begin{pmatrix} 0 & U_{22} & U_{23} & U_{24} & 0 & \cdots & 0 \end{pmatrix}$$

$$+\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & D'_3 & C'_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & 0 & 0 & B_{N-1} & D_N \end{pmatrix}$$

ステップ *N* - 2 のローカル因数分解は次のようになります。

$$P_{N-2}^{\mathrm{T}}\begin{pmatrix} D'_{N-2} & C'_{N-2} & 0 \\ B_{N-2} & D_{N-1} & C_{N-1} \end{pmatrix} = \begin{pmatrix} L_{N-2,N-2} \\ L_{N-1,N-2} \end{pmatrix}\begin{pmatrix} U_{N-2,N-2} & U_{N-2,N-1} & U_{N-2,N} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & D'_{N-1} & C'_{N-1} \end{pmatrix}$$

このステップの後、ピボット行列で乗算します。

$$\Pi_j^{\mathrm{T}} \quad j = 3, 4, \ldots, N-2$$

分解は次のようになります。

$$\Pi_{N-2}^{\mathrm{T}} \dots \Pi_2^{\mathrm{T}} \Pi_1^{\mathrm{T}} \begin{pmatrix} D_1 & C_1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ B_1 & D_2 & C_2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & B_2 & D_3 & C_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N \end{pmatrix}$$

$$= \Pi_{N-2}^{\mathrm{T}} \dots \Pi_3^{\mathrm{T}} \Pi_2^{\mathrm{T}} \begin{pmatrix} L_{1,1} \\ L_{2,1} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} & 0 & \cdots & 0 \end{pmatrix} + \Pi_{N-2}^{\mathrm{T}} \dots \Pi_3^{\mathrm{T}} \begin{pmatrix} 0 \\ L_{2,2} \\ L_{3,2} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} 0 & U_{22} & U_{23} & U_{24} & 0 & \cdots & 0 \end{pmatrix}$$

$$+ \dots + \Pi_{N-2}^{\mathrm{T}} \begin{pmatrix} 0 \\ \vdots \\ L_{N-3,N-3} \\ L_{N-2,N-3} \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & \cdots & 0 & U_{N-3,N-3} & U_{N-3,N-2} & U_{N-3,N-1} & 0 \end{pmatrix}$$

$$+ \begin{pmatrix} 0 \\ \vdots \\ 0 \\ L_{N-2,N-2} \\ L_{N-1,N-2} \\ 0 \end{pmatrix} \begin{pmatrix} 0 & \cdots & 0 & U_{N-2,N-2} & U_{N-2,N-1} & U_{N-2,N} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & D'_{N-1} & C'_{N-1} \\ 0 & 0 & 0 & 0 & \cdots & B_{N-1} & D_N \end{pmatrix}$$

最後の (N - 1) 番目のステップでは、行列は正方形で因数分解は完了です。

$$P_{N-1}^{\mathrm{T}} \begin{pmatrix} D'_{N-1} & C'_{N-1} \\ B_{N-1} & D_N \end{pmatrix} = \begin{pmatrix} L_{N-1,N-1} & 0 \\ L_{N,N-1} & L_{N,N} \end{pmatrix} \begin{pmatrix} U_{N-1,N-1} & U_{N-1,N} \\ 0 & U_{N,N} \end{pmatrix}$$

最後のステップはそれまでのステップとピボットの構造が異なります。前の P^{T}_j (j = 1, 2, ..., N - 2) はすべて n_b 置換の積 (n_b 整数パラメーターに依存) でしたが、$P^{\mathrm{T}}_{N\text{-}1}$ は次数 $2n_b$ の正方行列 ($2n_b$ パラメーターに依存) に適用されます。そのため、ピボット・インデックスをすべて格納するには、長さ (N - 2) n_b + $2n_b$ = Nn_b の整数配列が必要です。

左から前の分解と $\Pi^{\mathrm{T}}_{N\text{-}1}$ を乗算すると、最終的な分解が得られます。

この分解と $\Pi_1\Pi_2\ldots\Pi_{N\text{-}1}$ を乗算すると、LU 因数分解形式で記述できます。

$$
\begin{pmatrix}
D_1 & C_1 & 0 & 0 & \cdots & 0 & 0 & 0 \\
B_1 & D_2 & C_2 & 0 & \cdots & 0 & 0 & 0 \\
0 & B_2 & D_3 & C_3 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\
0 & 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N
\end{pmatrix} = LU
$$

$$
= \left(\Pi_1 \begin{pmatrix} L_{11} \\ L_{21} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \Pi_1\Pi_2 \begin{pmatrix} 0 \\ L_{22} \\ L_{32} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \cdot \ldots \cdot \Pi_1\Pi_2 \cdots \Pi_{N-2} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ L_{N-2,N-2} \\ L_{N-1,N-2} \\ 0 \end{pmatrix} \Pi_1\Pi_2 \cdots \Pi_{N-2}\Pi_{N-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ L_{N-1,N-1} \\ L_{N,N-1} \end{pmatrix} \Pi_1\Pi_2 \cdots \Pi_{N-2}\Pi_{N-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ L_{N,N} \end{pmatrix} \right)
$$

$$
\cdot \begin{pmatrix}
U_{1,1} & U_{1,2} & U_{1,3} & 0 & \cdots & 0 & 0 & 0 \\
0 & U_{2,2} & U_{2,3} & U_{2,4} & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & 0 & 0 & 0 & \cdots & U_{N-2,N-2} & U_{N-2,N-1} & U_{N-2,N} \\
0 & 0 & 0 & 0 & \cdots & 0 & U_{N-1,N-1} & U_{N-1,N} \\
0 & 0 & 0 & 0 & \cdots & 0 & 0 & U_{N,N}
\end{pmatrix}
$$

この式を適用するときは、Π_j (j = 1, 2, …, N-2) はインデックス j と j+1 のブロック行に適用される n_b 転置の積ですが、$\Pi_{N\text{-}1}$ は最後の 2 つのブロック行 N-1 と N に適用される $2n_b$ 転置の積であることに注意してください。

LU 因数分解されたブロック三重対角係数行列を含む連立線形方程式を解く

3

目的

インテル® MKL LAPACK のルーチンを使用してブロック三重対角係数行列を含む連立方程式の解を求める (LAPACK にはブロック三重対角係数行列を含む式を直接解くルーチンがないため)。

ソリューション

インテル® MKL LAPACK は、LU 因数分解された係数行列を含む連立方程式を解くためのさまざまなサブルーチンを提供しています (密行列、帯行列、三重対角行列など)。 この手法は、すべてのブロックが正方形で同位という条件を前提として、このセットをブロック三重対角行列に拡張します。ブロック三重対角行列 *A* の形式は次のとおりです。

$$A = \begin{pmatrix} D_1 & C_1 & 0 & \cdots & 0 & 0 & 0 \\ B_1 & D_2 & C_2 & \cdots & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N \end{pmatrix}$$

LU 因数分解された行列 *A*=*LU* と複数の右辺 (RHS) を含む式 *AX*=*F* は、2 段階で解きます (LU 因数分解の詳細は、「一般的なブロック三重対角行列の因数分解」を参照)。

1. 前方置換。ピボットで連立方程式 *LY*=*F* (*L* は下三角係数行列) を解きます。因数分解されたブロック三重対角行列では、最後のブロックを除く *Y* のブロックはすべて、次の方法によりループ内で見つかります。
   - **a.** 右辺にピボット置換を適用します。
   - **b.** 下三角係数行列の *NB* 線形方程式を解きます (*NB* はブロックの次数)。
   - **c.** 次のステップのために右辺を更新します。

   最後のピボットの構造 (2 つのブロック置換を連続して適用する必要がある) と係数行列の構造により、最後の 2 つのブロック・コンポーネントはループの外で見つかります。
2. 後方置換。式 *UX*=*Y* を解きます。 ピボットを含まないため、このステップはより単純です。 プロシージャーは、最初のステップに似ています。
   - **a.** 三角係数行列を含む方程式を解きます。
   - **b.** 右辺のブロックを更新します。

   前のステップとの違いは、係数行列が下三角ではなく上三角で、ループの向きが逆になっていることです。

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の BlockTDS_GE/source/dgeblttrs.f ファイルを参照してください。

前方置換

```
! 前方置換
! ループ内で配列 F に格納されているコンポーネント Y_K を計算
      DO K = 1, N-2
          DO I = 1, NB
              IF (IPIV(I,K) .NE. I) THEN
                  CALL DSWAP(NRHS, F((K-1)*NB+I,1), LDF, F((K-1)*NB+IPIV(I,K),1), LDF)
              END IF
          END DO
          CALL DTRSM('L', 'L', 'N', 'U', NB, NRHS, 1D0, D(1,(K-1)*NB+1), NB, F((K-1)*NB+1,1), LDF)
          CALL DGEMM('N', 'N', NB, NRHS, NB, -1D0, DL(1,(K-1)*NB+1), NB, F((K-1)*NB+1,1), LDF, 1D0,
     +        F(K*NB+1,1), LDF)
      END DO

! 最後の 2 つのピボットを適用
      DO I = 1, NB
           IF (IPIV(I,N-1) .NE. I) THEN
               CALL DSWAP(NRHS, F((N-2)*NB+I,1), LDF, F((N-2)*NB+IPIV(I,N-1),1), LDF)
           END IF
      END DO

      DO I = 1, NB
           IF(IPIV(I,N)-NB.NE.I)THEN
               CALL DSWAP(NRHS, F((N-1)*NB+I,1), LDF, F((N-2)*NB+IPIV(I,N),1), LDF)
           END IF
      END DO
! ループの外で Y_N-1 と Y_N を計算して配列 F に格納
      CALL DTRSM('L', 'L', 'N', 'U', NB, NRHS, 1D0, D(1,(N-2)*NB+1), NB, F((N-2)*NB+1,1), LDF)
      CALL DGEMM('N', 'N', NB, NRHS, NB, -1D0, DL(1,(N-2)*NB +1), NB, F((N-2)*NB+1,1), LDF, 1D0,
     +     F((N-1)*NB+1,1), LDF)
```

後方置換

```
      …
! 後方置換
! ループの外で X_N を計算して配列 F に格納
      CALL DTRSM('L', 'U', 'N', 'N', NB, NRHS, 1D0, D(1,(N-1)*NB+1), NB, F((N-1)*NB+1,1), LDF)
! ループの外で X_N-1 を計算して配列 F に格納
      CALL DGEMM('N', 'N', NB, NRHS, NB, -1D0, DU1(1,(N-2)*NB +1), NB, F((N-1)*NB+1,1), LDF, 1D0,
     +    F((N-2)*NB+1,1), LDF)
      CALL DTRSM('L', 'U', 'N', 'N', NB, NRHS, 1D0, D(1,(N-2)*NB+1), NB, F((N-2)*NB+1,1), LDF)
! ループ内で配列 F に格納されているコンポーネント X_K を計算
      DO K = N-2, 1, -1
          CALL DGEMM('N','N',NB, NRHS, NB, -1D0, DU1(1,(K-1)*NB +1), NB, F(K*NB+1,1), LDF, 1D0,
     +        F((K-1)*NB+1,1), LDF)
          CALL DGEMM('N','N',NB, NRHS, NB, -1D0, DU2(1,(K-1)*NB +1), NB, F((K+1)*NB+1,1), LDF, 1D0,
     +        F((K-1)*NB+1,1), LDF)
```

```
        CALL DTRSM('L', 'U', 'N', 'N', NB, NRHS, 1D0, D(1,(K-1)*NB+1), NB, F((K-1)*NB+1,1), LDF)
    END DO
    …
```

使用するルーチン

タスク	ルーチン	説明
ピボット置換を適用する	dswap	ベクトルを別のベクトルとスワップします。
下三角係数行列と上三角係数行列を用いて連立線形方程式を解く	dtrsm	三角行列を含む方程式を解きます。
右辺のブロックを更新する	dgemm	一般的な行列を含む行列-行列の積を計算します。

説明

注

サイズ *NB* x *NB* のブロックの一般的なブロック三重対角行列は、帯域幅 4**NB*-1 の帯行列として扱い、インテル® MKL LAPACK の帯行列を因数分解するサブルーチン (?gbtrf) と、帯行列を解くサブルーチン (?gbtrs) を呼び出して解くことができます。 しかし、ブロック行列を帯行列として格納すると、帯の多くのゼロ要素が非ゼロとして扱われて計算中に処理されるため、この手法で説明したアプローチで必要な浮動小数点計算は少なくなります。 より大きな *NB* では影響も大きくなります。帯行列をブロック三重対角行列として扱うこともできますが、 ブロックに非ゼロとして扱われる多くのゼロが含まれるため、この格納手法は効率的ではありません。 このため、帯格納手法とブロック三重対角格納手法、およびそれらのソルバーは、互いに補完的な手法として考えるべきです。

次の連立線形方程式について考えます。

$$AX = \begin{pmatrix} D_1 & C_1 & 0 & \cdots & 0 & 0 & 0 \\ B_1 & D_2 & C_2 & \cdots & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_{N-1} \\ X_N \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_{N-1} \\ F_N \end{pmatrix}$$

ブロック三重対角係数行列 *A* は次のように因数分解されると仮定します。

$$
\begin{aligned}
A &= L \cdot U \\
&= \left(\Pi_1 \begin{pmatrix} L_{11} \\ L_{21} \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \Pi_1\Pi_2 \begin{pmatrix} 0 \\ L_{22} \\ L_{32} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \cdots \Pi_1\Pi_2 \cdots \Pi_{N-2} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ L_{N-2,N-2} \\ L_{N-1,N-2} \\ 0 \end{pmatrix} \Pi_1\Pi_2 \cdots \Pi_{N-2}\Pi_{N-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ L_{N-1,N-1} \\ L_{N,N-1} \end{pmatrix} \Pi_1\Pi_2 \cdots \Pi_{N-2}\Pi_{N-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ L_{NN} \end{pmatrix} \right) \\
&\cdot \begin{pmatrix} U_{11} & U_{12} & U_{13} & 0 & \cdots & 0 & 0 & 0 \\ 0 & U_{22} & U_{23} & U_{24} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & U_{N-2,N-2} & U_{N-2,N-1} & U_{N-2,N} \\ 0 & 0 & 0 & 0 & \cdots & 0 & U_{N-1,N-1} & U_{N-1,N} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & U_{NN} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{N-1} \\ X_N \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_{N-1} \\ F_N \end{pmatrix}
\end{aligned}
$$

使用している用語の定義は、「一般的なブロック三重対角行列の因数分解」を参照してください。

式は 2 つの連立線形方程式に分解されます。

$$UX = Y, LY = F$$

2 つ目の方程式を拡張します。

$$
\begin{aligned}
LY &= \Pi_1 \begin{pmatrix} L_{11} \\ L_{21} \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} Y_1 + \Pi_1\Pi_2 \begin{pmatrix} 0 \\ L_{22} \\ L_{32} \\ 0 \\ \vdots \\ 0 \end{pmatrix} Y_2 + \cdots + \Pi_1\Pi_2 \cdots \Pi_{N-2} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ L_{N-2,N-2} \\ L_{N-1,N-2} \\ 0 \end{pmatrix} Y_{N-2} \\
&+ \Pi_1\Pi_2 \cdots \Pi_{N-2}\Pi_{N-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ L_{N-1,N-1} \\ L_{N,N-1} \end{pmatrix} Y_{N-1} + \Pi_1\Pi_2 \cdots \Pi_{N-2}\Pi_{N-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ L_{NN} \end{pmatrix} Y_N = \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_{N-1} \\ F_N \end{pmatrix}
\end{aligned}
$$

Y_1 を見つけるには、最初に置換 Π_1^T を適用する必要があります。 この置換は、右辺の最初の 2 つのブロックのみ変更します。

$$\begin{pmatrix} F_1' \\ F_2' \\ F_3 \\ \vdots \\ F_{N-1} \\ F_N \end{pmatrix} = \Pi_1^{\mathrm{T}} \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_{N-1} \\ F_N \end{pmatrix}$$

置換をローカルに適用します。

$$\begin{pmatrix} F_1' \\ F_2' \end{pmatrix} = P_1^{\mathrm{T}} \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$$

Y_1 が見つかりました。

$$Y_1 = L_{11}^{-1} F_1'$$

Y_1 を見つけた後、同様の計算を繰り返してほかの値 (Y_2, Y_3, ..., Y_{N-2}) を見つけます。

注

Π_{N-1} の異なる構造 (「一般的なブロック三重対角行列の因数分解」を参照) は、同じ方程式を Y_{N-1} と Y_N の計算に使用できず、ループの外で計算する必要があることを意味します。

Y を見つける方程式を用いるアルゴリズムは次のとおりです。

do for k = 1 to N - 2

$$\begin{pmatrix} F_k \\ F_{k+1} \end{pmatrix} := P_k^{\mathrm{T}} \begin{pmatrix} F_k \\ F_{k+1} \end{pmatrix}$$

$$Y_k = L_{kk}^{-1} F_k$$ //?trsm を使用

$$F_k := F_k - L_{k,k-1} Y_{k-1}$$ //?gemm を使用して右辺を更新

end do

$$\begin{pmatrix} F_{N-1} \\ F_N \end{pmatrix} := P_N^{\mathrm{T}} \begin{pmatrix} F_{N-1} \\ F_N \end{pmatrix}$$

$$Y_{N-1} = L_{N-1,N-1}^{-1} F_{N-1}$$ //?trsm

$$F_N := F_N - L_{N,N-1} Y_{N-1}$$ //?gemm を使用して右辺を更新

$$Y_N = L_{NN}^{-1} F_N$$ //?trsm

UX = *Y* 方程式は次のように表現できます。

$$\begin{pmatrix} U_{11} & U_{12} & U_{13} & 0 & \cdots & 0 & 0 & 0 \\ 0 & U_{22} & U_{23} & U_{24} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & \cdots & U_{N-2,N-2} & U_{N-2,N-1} & U_{N-2,N} \\ 0 & 0 & 0 & 0 & \cdots & 0 & U_{N-1,N-1} & U_{N-1,N} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & U_{NN} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{N-1} \\ X_N \end{pmatrix} = \begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_{N-1} \\ Y_N \end{pmatrix}$$

これらの方程式を解くアルゴリズムは次のとおりです。

$$X_N = U_{NN}^{-1} Y_N$$ //?trsm

$$X_{N-1} = U_{N-1,N-1}^{-1} \left(Y_{N-1} - U_{N-1,N-1} X_N \right)$$ //?gemm の後 ?trsm

do for *k* = *N* - 1 to 1 in steps of -1

$$X_k := Y_k - U_{k,\,k+1} X_{k+1}$$ //?gemm

$$X_k := X_k - U_{k,k+2} X_{k+2}$$ //?gemm

$$X_k := U_{k,k}^{-1} X_k$$ //?trsm

end do

ブロック三重対称正定値行列の因数分解

4

目的

対称正定値ブロック三重対角行列のコレスキー因数分解を行う。

ソリューション

対称正定値ブロック三重対角行列 (サイズ *NB* x *NB* の *N* の正方形ブロック) のコレスキー因数分解を行います。

1. 最初の対角ブロックのコレスキー因数分解を行います。
2. *N* - 1 回対角線に沿って下に移動します。
   - **a.** 三角係数の非対角ブロックを計算します。
   - **b.** 新しく計算した非対角ブロックで対角ブロックを更新します。
   - **c.** 対角ブロックのコレスキー因数分解を行います。

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `BlockTDS_SPD/source/dpbltrf.f` ファイルを参照してください。

対称正定値ブロック三重対角行列のコレスキー因数分解

```
…
CALL DPOTRF('L', NB, D, LDD, INFO)
…
DO K=1,N-1
    CALL DTRSM('R', 'L', 'T', 'N', NB, NB, 1D0, D(1,(K-1)*NB+1), LDD, B(1,(K-1)*NB+1), LDB)
    CALL DSYRK('L', 'N', NB, NB, -1D0, B(1,(K-1)*NB+1), LDB, 1D0, D(1,K*NB+1), LDD)
    CALL DPOTRF('L', NB, D(1,K*NB+1), LDD, INFO)
    …
END DO
```

使用するルーチン

タスク	ルーチン	説明
対角ブロックのコレスキー因数分解を行う	`DPOTRF`	対称 (エルミート) 正定値行列のコレスキー因数分解を計算します。
三角係数の非対角ブロックを計算する	`DTRSM`	三角行列を含む方程式を解きます。
対角ブロックを更新する	`DSYRK`	対称ランク-k 更新を行います。

説明

対称正定値ブロック三重対角行列 (サイズ *NB* x *NB* の *N* の対角ブロック D_i および *N* - 1 の 1 つ下の対角ブロック B_i) は、次のように因数分解されます。

$$\begin{pmatrix} D_1 & B_1^{\mathrm{T}} & & & \\ B_1 & D_2 & B_2^{\mathrm{T}} & & \\ & \ddots & \ddots & \ddots & \\ & & B_{N-2} & D_{N-1} & B_{N-1}^{\mathrm{T}} \\ & & & B_{N-1} & D_N \end{pmatrix} = \begin{pmatrix} L_1 & & & & \\ C_1 & L_2 & & & \\ & \ddots & \ddots & \ddots & \\ & & C_{N-2} & L_{N-1} & \\ & & & C_{N-1} & L_N \end{pmatrix} \cdot \begin{pmatrix} L_1^{\mathrm{T}} & C_1^{\mathrm{T}} & & & \\ & L_2^{\mathrm{T}} & C_2^{\mathrm{T}} & & \\ & \ddots & \ddots & \ddots & \\ & & & L_{N-1}^{\mathrm{T}} & C_{N-1}^{\mathrm{T}} \\ & & & & L_N^{\mathrm{T}} \end{pmatrix}$$

右の行列のブロックを乗算します。

$$\begin{pmatrix} L_1 & & & & \\ C_1 & L_2 & & & \\ & \ddots & \ddots & \ddots & \\ & & C_{N-2} & L_{N-1} & \\ & & & C_{N-1} & L_N \end{pmatrix} \cdot \begin{pmatrix} L_1^{\mathrm{T}} & C_1^{\mathrm{T}} & & & \\ & L_2^{\mathrm{T}} & C_2^{\mathrm{T}} & & \\ & \ddots & \ddots & \ddots & \\ & & & L_{N-1}^{\mathrm{T}} & C_{N-1}^{\mathrm{T}} \\ & & & & L_N^{\mathrm{T}} \end{pmatrix}$$

$$= \begin{pmatrix} L_1 L_1^{\mathrm{T}} & L_1 C_1^{\mathrm{T}} & & & \\ C_1 L_1^{\mathrm{T}} & C_1 C_1^{\mathrm{T}} + L_2 L_2^{\mathrm{T}} & L_2 C_2^{\mathrm{T}} & & \\ & \ddots & \ddots & \ddots & \\ & & C_{N-2} L_{N-2}^{\mathrm{T}} & C_{N-2} C_{N-2}^{\mathrm{T}} + L_{N-1} L_{N-1}^{\mathrm{T}} & L_{N-1} C_{N-1}^{\mathrm{T}} \\ & & & C_{N-1} L_{N-1}^{\mathrm{T}} & C_{N-1} C_{N-1}^{\mathrm{T}} + L_N L_N^{\mathrm{T}} \end{pmatrix}$$

オリジナルブロック三重対角行列の要素と乗算した係数の要素を等式にします。

$$
\begin{aligned}
L_1 L_1^{\mathrm{T}} &= D_1 \\
C_1 L_1^{\mathrm{T}} &= B_1 \\
C_1 C_1^{\mathrm{T}} + L_2 L_2^{\mathrm{T}} &= D_2 \\
C_2 L_2^{\mathrm{T}} &= B_2 \\
&\vdots \\
C_{N-2} C_{N-2}^{\mathrm{T}} + L_{N-1} L_{N-1}^{\mathrm{T}} &= D_{N-1} \\
C_{N-1} L_{N-1}^{\mathrm{T}} &= B_{N-1} \\
C_{N-1} C_{N-1}^{\mathrm{T}} + L_N L_N^{\mathrm{T}} &= D_N
\end{aligned}
$$

C_i および $L_i L_i^{\mathrm{T}}$ を解きます。

$$
\begin{aligned}
L_1 L_1^{\mathrm{T}} &= D_1 \\
C_1 &= B_1 L_1^{-\mathrm{T}} \\
L_2 L_2^{\mathrm{T}} &= D_2 - C_1 C_1^{\mathrm{T}} \\
C_2 &= B_2 L_2^{-\mathrm{T}} \\
&\vdots \\
L_{N-1} L_{N-1}^{\mathrm{T}} &= D_{N-1} - C_{N-2} C_{N-2}^{\mathrm{T}} \\
C_{N-1} &= B_{N-1} L_{N-1}^{-\mathrm{T}} \\
L_N L_N^{\mathrm{T}} &= D_N - C_{N-1} C_{N-1}^{\mathrm{T}}
\end{aligned}
$$

$L_i L_i^{\mathrm{T}}$ の方程式の右辺はコレスキー因数分解であることに注意してください。このため、次のようなコードを使用して、コレスキー因数分解を行うルーチン `chol()` をこの問題に利用できます。

```
L1=chol(D1)
do i=1,N-1
     Ci=Bi·Li^-T //trsm()
     Di+1:=Di+1 - Ci·Ci^T //syrk()
     Li+1=chol(Di+1)
end do
```

ブロック三重対称正定値係数行列を含む連立線形方程式を解く

5

目的

コレスキー因数分解された対称正定値ブロック三重対角係数行列を含む連立線形方程式を解く。

ソリューション

係数対称正定値ブロック三重対角行列 (それぞれ同じサイズ*NB* x *NB* の正方形ブロック) を LLT 因数分解する場合、次の 2 段階で解きます。

1. *N* x *N* ブロック (サイズ *NB* x *NB*) で、対角ブロックが下三角行列の下二重対角係数行列を含む連立線形方程式を解きます。
   - **a.** 右辺ベクトルの係数行列として使用されるサイズ *NB* x *NB* の下三角対角ブロックを含む *N* 連立方程式を解きます。
   - **b.** 右辺を更新します。
2. *N* x *N* ブロック (サイズ *NB* x *NB*) で、対角ブロックが上三角行列の上二重対角係数行列を含む連立線形方程式を解きます。
   - **a.** 連立方程式を解きます。
   - **b.** 右辺を更新します。

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `BlockTDS_SPD/source/dpbltrs.f` ファイルを参照してください。

対称正定値ブロック三重対角行列のコレスキー因数分解

```
      …
      …
      CALL DTRSM('L', 'L', 'N', 'N', NB, NRHS, 1D0, D, LDD, F, LDF)
      DO K = 2, N
          CALL DGEMM('N', 'N', NB, NRHS, NB, -1D0, B(1,(K-2)*NB+1), LDB, F((K-2)*NB+1,1), LDF, 1D0,
F((K-1)*NB+1,1), LDF)
          CALL DTRSM('L','L', 'N', 'N', NB, NRHS, 1D0, D(1,(K-1)*NB+1), LDD, F((K-1)*NB+1,1), LDF)
      END DO

      CALL DTRSM('L', 'L', 'T', 'N', NB, NRHS, 1D0, D(1,(N-1)*NB+1), LDD, F((N-1)*NB+1,1), LDF)
      DO K = N-1, 1, -1
          CALL DGEMM('T', 'N', NB, NRHS, NB, -1D0, B(1,(K-1)*NB+1), LDB, F(K*NB+1,1), LDF, 1D0,
F((K-1)*NB+1,1), LDF)
          CALL DTRSM('L','L', 'T', 'N', NB, NRHS, 1D0, D(1,(K-1)*NB+1), LDD, F((K-1)*NB+1,1), LDF)
      END DO
```

```
...
```

使用するルーチン

タスク	ルーチン	説明
連立線形方程式を解く	DTRSM	三角行列を含む方程式を解きます。
右辺を更新する	DGEMM	一般的な行列を含む行列-行列の積を計算します。

説明

次の連立線形方程式について考えます。

$$AX = \begin{pmatrix} D_1 & B_1^{\mathrm{T}} & & & \\ B_1 & D_2 & B_2^{\mathrm{T}} & & \\ & \ddots & \ddots & \ddots & \\ & & B_{N-2} & D_{N-1} & B_{N-1}^{\mathrm{T}} \\ & & & B_{N-1} & D_N \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_{N-1} \\ X_N \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_{N-1} \\ F_N \end{pmatrix}$$

行列 *A* は、すべてのブロックがサイズ *NB* x *NB* の対称正定値ブロック三重対角係数行列であると仮定します。*A* は、「ブロック三重対称正定値行列の因数分解」で説明しているように、次のように因数分解できます。

$$A = \begin{pmatrix} L_1 & & & & \\ C_1 & L_2 & & & \\ & \ddots & \ddots & \ddots & \\ & & C_{N-2} & L_{N-1} & \\ & & & C_{N-1} & L_N \end{pmatrix} \cdot \begin{pmatrix} L_1^{\mathrm{T}} & C_1^{\mathrm{T}} & & & \\ & L_2^{\mathrm{T}} & C_2^{\mathrm{T}} & & \\ & \ddots & \ddots & \ddots & \\ & & & L_{N-1}^{\mathrm{T}} & C_{N-1}^{\mathrm{T}} \\ & & & & L_N^{\mathrm{T}} \end{pmatrix}$$

連立方程式を解くアルゴリズムは次のとおりです。

1. 対角ブロックが下三角行列の下二重対角係数行列を含む連立線形方程式を解きます。

```
Y1=L1^-1 F1 //trsm()
do i=2,N
     Gi=Fi - Ci - 1 Yi - 1 //gemm()
     Yi=Li^-1 Gi //trsm()
end do
```

2. 対角ブロックが上三角行列の上二重対角係数行列を含む連立線形方程式を解きます。

```
X_N=L_N^-T Y_N //trsm()
do i=N-1,1,-1
     Z_i=F_i-C_i^T X_i + 1 //gemm()
     X_i=L_i^-T Z_i //trsm()
end do
```

2 つの部分空間の間の主角度の計算

6

目的

内積空間の 2 つの部分空間の相対的な位置に関する情報を得る。

ソリューション

部分空間はいくつかのベクトルのスパンとして表現され、部分空間の相対的な位置は部分空間の間の*主角度*のセットを計算して得ることができると仮定します。角度を計算するには、次の操作を行います。

1. 各部分空間に正規直交基底を構築して、部分空間の次元を決定します。
   - **a.** 適切なサブルーチンを呼び出して (列が複数の部分空間をスパンする) 行列のピボットで QR 因数分解を行います。
   - **b.** しきい値を使用して、部分空間の次元を決定します。
   - **c.** 正規直交基底を形成します。
2. 1 つの部分空間の基底ベクトルと別の部分空間の基底ベクトルの内積の行列を形成する
3. 行列の特異値分解を計算します。

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `ANGLES/definition/main.f` ファイルを参照してください。

正規直交基底を構築して部分空間の次元を決定

```
…
REAL*8 Y(LDY,K),TAU(N),WORK(3*N)
      …
!
! ピボットで QR 因数分解を適用
      CALL DGEQPF(N, K, Y, LDY, JPVT, TAU, WORK, INFO)
! DGEQPF により返された INFO を処理
      …
! ランクを計算
      K1=0
      DO WHILE((K1 .LT. K) .AND. (ABS(Y(K1+1,K1+1)) .GT. THRESH))
          K1 = K1 + 1
      END DO
!
! DORGQR を呼び出して K1 直交ベクトルを形成
      CALL DORGQR(N, K1, K1, Y, LDY, TAU, WORK, LWORK, INFO)
! DORGQR により返された INFO を処理
      …
```

内積の行列を形成して SVD を計算

```
REAL*8 U(N,KU),V(N,KV),W(KU,KV),VECL(KU,KMIN)
REAL*8 VECRT(KMIN,KV),S(KMIN),WORK(5*KU)
…
! Form W=U^t*V
      CALL DGEMM('T', 'N', KU, KV, N, 1D0, U, N, V, N, 0D0, W, KU1)
…
! SVD of W=U^t*V
      CALL DGESVD('S', 'S', KU, KV, W, KU, S, VECL, KU, VECRT, KMIN, WORK, LWORK, INFO)
! DGESVD により返された INFO を処理
      …
```

説明

使用するルーチン

タスク	ルーチン	説明
行列のピボットを使用した QR 因数分解	dgeqpf	ピボットで一般的な m x n 行列の QR 因数分解を計算します。
正規直交基底を形成する	dorgqr	?geqpf または ?geqp3 で形成される QR 因数分解の実直交行列 Q を生成します。
1 つの部分空間の基底ベクトルと別の部分空間の基底ベクトルの内積の行列を形成する	dgemm	一般的な行列を含む行列-行列の積を計算します。
行列の特異値分解を計算する	dgesvd	一般的な長方行列の特異値分解を計算します。

最初に、各部分空間に正規直交基底を構築して、部分空間の次元を決定します。

U をいくつかの内積線形空間のベクトルを表現する列を含む N x k 行列 ($N \geq k$) とします。この空間の正規直交基底を構築するには、行列 U の QR 因数分解を使用します。ピボットは $UP = QR$ として表現できます。空間の次元が l ($l \leq k$) で、丸め誤差が発生しない場合、直交 (複素数値行列のユニタリ) N x N 行列 Q および上三角 N x k 行列 R が得られます。

$$R = \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,l-1} & r_{1,l} & r_{1,l+1} & r_{1,l+2} & \cdots & r_{1,k} \\ & r_{2,2} & \cdots & r_{2,l-1} & r_{2,l} & r_{2,l+1} & r_{2,l+2} & \cdots & r_{2,k} \\ & & \ddots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ & & & r_{l-1,l-1} & r_{l-1,l} & r_{l-1,l+1} & r_{l-1,l+2} & \cdots & r_{l-1,k} \\ & & & & r_{l,l} & r_{l,l+1} & r_{l,l+2} & \cdots & r_{l,k} \\ & & & & & 0 & 0 & \cdots & 0 \\ & & & & & & \ddots & \vdots & \vdots \\ & & & & & & & 0 & 0 \\ & & & & & & & & 0 \end{pmatrix}$$

方程式 $UP = QR$ は、U のすべての列が Q の最初の l 列の線形の組み合わせであることを意味します。ピボットにより、R の対角要素 $r_{j,j}$ は絶対値の降順になります。事実、ピボットにより、より強い不等式になります。

$$\left|r_{j,j}\right| \geq \sqrt{\sum_{i=j}^{m} \left|r_{im}\right|^2}$$

$j \leq m \leq k$。

丸め誤差を含む実際の計算では、R の右下 $(k - l) \times (k - l)$ 三角の要素は小さいがゼロではないため、しきい値 threshold を使用してランク $|r_{l,l}| > threshold > |r_{l+1,l+1}|$ を決定します。

これで部分空間の間の角度を決定できるようになりました。

$\mathcal{U}$ および $\mathcal{W}$ を、$\dim(\mathcal{U})=k$、$\dim(\mathcal{W})=l$、$k \leq l$ の同じ N 次元ユークリッド空間の 2 つの部分空間とします。 これらの部分空間の相対的な位置を見つけるには、主角度 $\theta_1 \geq \theta_2 \geq \ldots \geq \theta_k \geq 0$ を使用します。次のように定義されます。

最初の角度は次のように定義されます。

$$\theta_1 = \min\{\arccos(u, w) | u \in \mathcal{U}, w \in \mathcal{W}, |u| = |w| = 1\} = \angle(u_1, w_1)$$

ベクトル u_1 および w_1 は*主ベクトル*と呼ばれます。 ほかの主角度およびベクトルは再帰的に定義されます。

$$\theta_i = \min\left\{\arccos(u, w) | u \in \mathcal{U}, w \in \mathcal{W}, |u| = |w| = 1, u \perp u_j, w \perp w_j \quad \forall j \in \{1, \ldots, i-1\}\right\}$$

同じ部分空間の主ベクトルはペアで直交です。

$(u_i, u_j) = (w_i, w_j) = \delta_{ij}$

主角度の計算には、行列の特異値分解を使用します。 U および W をそれぞれサイズ $N \times k$ および $N \times l$ の行列、列をそれぞれ $\mathcal{U}$ および $\mathcal{W}$ の正規直交基底とします。 $k \times l$ 行列 $U^T W$ の SVD を計算します。

$U^T W = P \Sigma Q^T$, $P^T P = I_k$, $Q Q^T = I_l$

これで、Σ の対角要素が主角度の余弦であることを証明できます。

$$\Sigma = \begin{bmatrix} \cos\theta_1 & & & & 0 & 0 & \cdots & 0 \\ & \cos\theta_2 & & & 0 & 0 & \cdots & 0 \\ & & \ddots & & \vdots & \vdots & \ddots & \vdots \\ & & & \cos\theta_k & 0 & 0 & \cdots & 0 \end{bmatrix}$$

各ペアの主ベクトルは Up^i および Wq^i です。ここで、p^i および q^i は、P および Q の i 番目の列です。

ブロック三角行列の不変部分空間の間の主角度の計算

7

目的

ブロック三角行列の 2 つの不変部分空間の相対的な位置に関する情報を得る。

ソリューション

部分空間はいくつかのベクトルのスパンとして表現され、部分空間の相対的な位置は部分空間の間の主角度のセットを計算して得ることができると仮定します (主角度を参照)。さらに、ブロック三角行列の不変部分空間はシルベスター行列方程式を使用して解くものとします。 使用するソルバーは、行列の特性に依存します。

- 三角行列の対角ブロックがどちらも上三角の場合、LAPACK `?trsyl` ルーチンを使用します。
- 三角行列の対角ブロックがどちらも大きくなく、上三角でない場合、LAPACK 線形ソルバーを使用します。
- 三角行列の対角ブロックがどちらも大きく、上三角で、疎の場合、インテル® MKL PARDISO ソルバーを使用します。

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) のファイルを参照してください。

- `ANGLES/uep_subspace1/main.f`
- `ANGLES/uep_subspace2/main.f`
- `ANGLES/uep_subspace3/main.f`

LAPACK ?trsyl を使用してシルベスター行列方程式を解く

```
      CALL DTRSYL('N', 'N', -1, K, N-K, AA, N, AA(K+1,K+1), N,
     &             AA(1,K+1), N, ALPHA, INFO)
      IF(INFO.EQ.0) THEN
          PRINT *,"DTRSYL completed, SCALE=",ALPHA
      ELSE IF(INFO.EQ.1) THEN
          PRINT *,"DTRSYL solved perturbed equations"
      ELSE
          PRINT *,"DTRSYL failed. INFO=",INFO
          STOP
      END IF
```

LAPACK 線形ソルバーを使用してシルベスター行列方程式を解く

```
      REAL*8 AA(N,N), FF(K*(N-K)), AAA(K*(N-K),K*(N-K))
…
! シルベスター方程式の密係数行列を形成
      CALL SYLMAT(K, AA, N, N-K, AA(K+1,K+1), N, -1D0, 1D0, AAA, NK,
     &        INFO)
! SYLMAT により返された INFO を処理
…
! シルベスター方程式に相当する連立線形方程式の
! 右辺を形成
      DO I = 1, K
          DO J = 1, N-K
```

```
            FF((J-1)*K+I) = AA(I,J+K)
        END DO
    END DO
! 連立線形方程式を解く
    CALL DGESV(NK, 1, AAA, NK, IPIV, FF, NK, INFO )
! DGESV により返された INFO を処理
```

インテル® MKL PARDISO を使用してシルベスター行列方程式を解く

```
    REAL*8 AA(N,N), FF(K*(N-K)), VAL(K*(N-K)*(N-1))
    INTEGER ROWINDEX(K*(N-K)+1), COLS(K*(N-K)*(N-1))
…
! シルベスター方程式の疎係数行列を形成
    CALL FSYLVOP(K, AA, N, N-K, AA(K+1,K+1), N, -1D0, 1D0, COLS,
   &            ROWINDEX, VAL, INFO)
! FSYLVOP により返された INFO を処理
…
! シルベスター方程式の右辺を形成
    DO I=1,K
        DO J=1,N-K
            FF((J-1)*K+I) = AA(I,J+K)
        END DO
    END DO

    CALL PARDISOINIT (PT, 1, IPARM)
    CALL PARDISO (PT, 1, 1, 11, 13, NK, VAL, ROWINDEX,
   &                COLS, PERM, 1, IPARM, 1, FF, X, IERR)
! PARDISO により返された IERR を処理
…
```

説明

使用するルーチン

タスク	ルーチン	説明
上三角対角ブロックを含む行列についてシルベスター行列方程式を解く	dtrsyl	実数擬似三角または複素数三角行列についてシルベスター方程式を解きます。
小さく上三角でない行列についてシルベスター方程式を解く	dgesv	正方行列 *A* および複数の右辺を含む連立線形方程式の解を計算します。
小さくなく、上三角でない行列についてシルベスター方程式を解く	pardiso	単一または複数の右辺を含む 1 セットの疎線形方程式の解を計算します。

行列の不変部分空間の間の主角度を決定するには、最初に *N* x *N* 行列をブロック三角形式で表します。

$$A = \begin{pmatrix} A & F \\ 0 & B \end{pmatrix}$$

置換式

$$A = \begin{pmatrix} A & F \\ 0 & B \end{pmatrix}$$

ここで、対角ブロック *A* および *B* はそれぞれ、次数 *k* および *N-k* の正方行列です。I_k が *k* の単位行列を示す場合、次の等式

$$\begin{pmatrix} A & F \\ 0 & B \end{pmatrix}\begin{pmatrix} I_k \\ 0 \end{pmatrix} = \begin{pmatrix} A \\ 0 \end{pmatrix}$$

は、標準基底の最初の *k* ベクトルのスパンが行列 *A* の変換に対して不変であることを意味します。

別の不変部分空間は合成行列 $\begin{pmatrix} X \\ I_{N-k} \end{pmatrix}$ の列のスパンとして得ることができます。

ここで、*X* は長方形の *k* x (*N* - *k*) 行列です。 積を計算します。

$$\begin{pmatrix} A & F \\ 0 & B \end{pmatrix}\begin{pmatrix} X \\ I_{N-k} \end{pmatrix} = \begin{pmatrix} AX + F \\ B \end{pmatrix}$$

X がシルベスター方程式 *XB* - *AX* = *F* の解の場合、最後の方程式の結果は次のようになります。

$$\begin{pmatrix} AX + F \\ B \end{pmatrix} = \begin{pmatrix} XB \\ B \end{pmatrix} = \begin{pmatrix} X \\ I_{N-k} \end{pmatrix} B$$

これは、$\begin{pmatrix} X \\ I_{N-k} \end{pmatrix}$ の列でスパンされた部分空間の不変性を表しています。

2 つ目の不変部分空間の基底を直交させるには、QR 因数分解を使用します。

$$\begin{pmatrix} X \\ I_{N-k} \end{pmatrix} = \begin{pmatrix} C \\ S \end{pmatrix} R$$

ここで、*C* は *k* x (*N*-*k*) 行列で、*S* は (*N*-*k*) x (*N*-*k*) 行列です。*C* および *S* は方程式 $C^{\mathrm{T}}C + S^{\mathrm{T}}S = I_{N-k}$ を満たします。ここで、*R* は次数 *N*-*k* の上三角正方行列です。*C* の SVD を使用してこれらの 2 つの不変部分空間の間の主角度を計算します。

$$C = \begin{pmatrix} I_k \\ 0 \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} C \\ S \end{pmatrix} = V\Sigma U^{\mathrm{T}}, V^{\mathrm{T}}V = I_k, U^{\mathrm{T}}U = I_{N-k}$$

Σ の対角要素は主角度の余弦です。

シルベスター方程式の行列

シルベスター方程式 $\alpha AX + \beta XB = F$ を考えます。

ここで、正方行列 *A* および *B* の次数はそれぞれ *M* と *N* で、α と β はスカラーです。*F* は指定される *M* x *N* 行列です。

$$F = \begin{pmatrix} f_{11} & f_{12} & \cdots & f_{1N} \\ f_{21} & f_{12} & \cdots & f_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ f_{M1} & f_{M2} & \cdots & f_{MN} \end{pmatrix}$$

X は求める *M* x *N* 行列です。

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{12} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{pmatrix}$$

この行列方程式は、ベクトル *x* と右辺ベクトル *f* の *MN* 成分が不明な、$\mathcal{A}x = f(*)$ 系の *MN* 線型方程式と見なすことができます。

$x = (x_{11}, x_{21}, \ldots, x_{M1}, x_{12}, x_{22}, \ldots, x_{M2}, \ldots, x_{1N}, x_{2N}, \ldots, x_{MN})^T$

$f = (f_{11}, f_{21}, \ldots, f_{M1}, f_{12}, f_{22}, \ldots, f_{M2}, \ldots, f_{1N}, f_{2N}, \ldots, f_{MN})^T$

次数 *MN* の行列 $\mathcal{A}$ は、2 つの行列の合計として表すことができます。1 つの行列は、行列 *X* (左から) と行列 *A* の乗算に相当し、サイズ *M* x *M* のブロック形式で表現できます。 この行列は、対角線上で *N* ブロックのブロック対角行列を形成します。

$$\begin{pmatrix} \alpha A & & & & \\ & \alpha A & & & \\ & & \alpha A & & \\ & & & \ddots & \\ & & & & \alpha A \end{pmatrix}$$

合計の別の行列は、行列 *X* (右から) と行列 *B* の乗算に相当します。同じブロック形式を使用して行列は次のように表現できます。

$$\begin{pmatrix} \beta b_{11} I_M & \beta b_{21} I_M & \beta b_{31} I_M & \cdots & \beta b_{N1} I_M \\ \beta b_{12} I_M & \beta b_{22} I_M & \beta b_{32} I_M & \cdots & \beta b_{N2} I_M \\ \beta b_{13} I_M & \beta b_{23} I_M & \beta b_{33} I_M & \cdots & \beta b_{N3} I_M \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \beta b_{1N} I_M & \beta b_{2N} I_M & \beta b_{3N} I_M & \cdots & \beta b_{NN} I_M \end{pmatrix}$$

ここで、I_M は次数 *M* の単位行列を表します。 つまり、係数行列は次のようになります。

$$
\mathcal{A} = \begin{pmatrix} \beta b_{11} I_M + \alpha A & \beta b_{21} I_M & \beta b_{31} I_M & \cdots & \beta b_{N1} I_M \\ \beta b_{12} I_M & \beta b_{22} I_M + \alpha A & \beta b_{32} I_M & \cdots & \beta b_{N2} I_M \\ \beta b_{13} I_M & \beta b_{23} I_M & \beta b_{33} I_M + \alpha A & \cdots & \beta b_{N3} I_M \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \beta b_{1N} I_M & \beta b_{2N} I_M & \beta b_{3N} I_M & \cdots & \beta b_{NN} I_M + \alpha A \end{pmatrix}
$$

この行列は、*MN* 要素の行に *M* + *N* - 1 の非ゼロ要素を含む疎行列です。 このため、インテル® MKL PARDISO スパースソルバーを効率的に使用できます (インテル® MKL PARDISO 向けに CSR 形式で係数行列を形成するコード `ANGLES/source/fsylvop.f` を参照)。 しかし、*M* および *N* が小さい場合は、インテル® MKL LAPACK 線形ソルバーがより効率的です (`dgesv` を使用する密行列として係数行列を形成するコード `ANGLES/source/sylmat.f` を参照)。

フーリエ積分の評価

8

目的

高速フーリエ変換 (FFT) を使用して連続するフーリエ変換積分を数値的に評価する。

$$F(\xi) = \int_{-\infty}^{+\infty} f(x)\exp(-i\xi x)dx.$$

ソリューション

実数値関数 $f(x)$ が範囲 $[a, b]$ 外のゼロで、N 個の等距離のポイント $x_n = a + nT/N$ (ここで、$T = |b - a|$ および $n = 0, 1, \ldots, N-1$) でサンプリングされると仮定します。FFT を使用して、ポイント $\xi_k = k2\pi/T$ (ここで、$k = 0, 1, \ldots, N/2$) の積分を評価します。

インテル® MKL の FFT インターフェイスの使用 (C/C++)

```
float *f;   // input: f[n] = f(a + n*T/N), n=0...N-1
complex *F; // output: F[k] = F(2*k*PI/T), k=0...N/2
DFTI_DESCRIPTOR_HANDLE h = NULL;
DftiCreateDescriptor(&h,DFTI_SINGLE,DFTI_REAL,1,(MKL_LONG)N);
DftiSetValue(h,DFTI_CONJUGATE_EVEN_STORAGE,DFTI_COMPLEX_COMPLEX);
DftiSetValue(h,DFTI_PLACEMENT,DFTI_NOT_INPLACE);
DftiCommitDescriptor(h);
DftiComputeForward(h,f,F);
for (int k = 0; k <= N/2; ++k)
{
  F[k] *= (T/N)*complex( cos(2*PI*a*k/T), -sin(2*PI*a*k/T) );
}
```

説明

評価は積分の階段関数近似に基づき、この派生に従います。

$$F_k = F(\xi_k) = \int_a^b f(x)\exp(-i2\pi xk/T)dx$$

$$\approx (T/N)\sum_{n=0}^{N-1} f(x_n)\exp(-i2\pi(a + nT/N)k/T)$$

$$= (T/N)\exp(-i2\pi ak/T)\sum_{n=0}^{N-1} f_n \exp(-i2\pi kn/N).$$

最後の行の合計は定義による FFT です。関数 *f* のサポートがゼロ付近で対称的に拡張される、つまり [*a*, *b*] = [-*T*/2, *T*/2] の場合、合計の前の係数は $(T/N)(-1)^k$ になります。

関数 *f* が実数値の場合、$F(\xi_k) = \mathrm{conj}(F(\xi_{N-k}))$ になります。実数から複素数 FFT の最初の *N*/2 + 1 複素数値は、実数入力とほぼ同じメモリーを占めます。共役により全体の結果を計算するのに十分です。FFT 計算が実数から複素数への変換を行うように構成されている場合、さらに複素数から複素数 FFT の約半分の時間がかかります。

高速フーリエ変換を使用したコンピューター・トモグラフィー・イメージの復元

9

目的

高速フーリエ変換 (FFT) 関数を使用してコンピューター・トモグラフィー (CT) データから元のイメージを復元する。

ソリューション

表記規則:

- インデックス範囲の表記には MATLAB* で使われている表記規則を採用しています。

  例えば、k=-q : q は、k=-q, -q+1, -q+2,..., q-1, q を意味します。
- $f(x)$ は関数 f のポイント x の値を意味し、$f[n]$ は離散データセット f の n 番目の要素の値を意味します。

仮定:

- 2 次元 (2D) イメージの密度 $f(x, y)$ は単位円の外部で消えます。

  $x^2 + y^2 > 1$ の場合 $f = 0$
- CT データは、角度 $\theta_j = j\pi/p$ で取得したイメージの p の投影からなります。ここで、$j = 0 : p - 1$ です。
- 各投影には、次の線に沿ったイメージの積分に近似する $2q + 1$ 密度値 $g[j, l] = g(\theta_j , s_l)$ が含まれます。

  $(x, y) = (-t \sin\theta_j + s_l \cos\theta_j , t \cos\theta_j + s_l \sin\theta_j)$

  ここで、$l = -q : q$、$s_l = l /q$、t は積分パラメーターです。

離散イメージ復元アルゴリズムは次のステップからなります。

1. p 1 次元 (1D) フーリエ変換 ($j = 0 : p - 1$ および $r = -q : q$) を評価します。

$$g_1(\theta_j, \pi r / q) = (2 / \sqrt{2\pi}) \sum_{l=-q}^{q} g[j, l]\, e^{-i\pi r l/q}.$$

2. $g_1[j, r]$ をラジアルグリッド $(\pi r/q)(\cos\theta_j , \sin\theta_j)$ からデカルトグリッド $(\xi, \eta) = (-q : q, -q : q)$ に補間し、$f_2(\pi\xi/q, \pi\eta/q)$ を取得します。

3. 逆 2 次元複素数-複素数 FFT を評価して、複素数値の復元イメージ f_1 を取得します。

$$f_1[m, n] = (2\pi)^{-1} \sum_{\xi=-q}^{q} \sum_{\eta=-q}^{q} f_2[\xi, \eta]\, e^{i\pi m\xi/q} e^{i\pi n\eta/q},$$

ここで、$f(m/q, n/q) \approx f_1[m, n]$ ($m = -q : q$ および $n = -q : q$) です。

ステップ 1 と 3 の計算は、インテル® MKL FFT インターフェイスを呼び出します。ステップ 2 の計算は、インテル® MKL FFT で使用されるデータレイアウトに合わせた、単純なバージョンの補間を実装します。

元の CT イメージの復元 (C/C++)

```
// 宣言
int Nq = 2*(q+1); // インプレース r2c FFT 用の空間
void    *gmem = mkl_malloc( sizeof(float)*p*Nq, 64 );
float    *g = gmem; // g[j*Nq + ell+q]
complex *g1 = gmem; // g1[j*Nq/2 + r+q]

// g を CT データで初期化
for (int j = 0; j < p; ++j)
for (int ell = 0; ell < 2*q+1; ++ell) {
  g[j*Nq + ell+q] = get_g(theta_j,s_ell);
}

// ステップ 1: 1D 実数-複素数 FFT を構成して計算
DFTI_DESCRIPTOR_HANDLE h1 = NULL;
DftiCreateDescriptor(&h1,DFTI_SINGLE,DFTI_REAL,1,(MKL_LONG)2*q);
DftiSetValue(h1,DFTI_CONJUGATE_EVEN_STORAGE,DFTI_COMPLEX_COMPLEX);
DftiSetValue(h1,DFTI_NUMBER_OF_TRANSFORMS,(MKL_LONG)p);
DftiSetValue(h1,DFTI_INPUT_DISTANCE,(MKL_LONG)Nq);
DftiSetValue(h1,DFTI_OUTPUT_DISTANCE,(MKL_LONG)Nq/2);
DftiSetValue(h1,DFTI_FORWARD_SCALE,fscale);
DftiCommitDescriptor(h1);
DftiComputeForward(h1,g); // gmem に g1 が含まれる

// ステップ 2: g1 を f2 に補間 - ここでは省略
complex *f = mkl_malloc( sizeof(complex) * 2*q * 2*q, 64 );

// ステップ 3: 2D 複素数-複素数 FFT を構成して計算
DFTI_DESCRIPTOR_HANDLE h3 = NULL;
MKL_LONG sizes[2] = {2*q, 2*q};
DftiCreateDescriptor(&h3,DFTI_SINGLE,DFTI_COMPLEX,2,sizes);
DftiCommitDescriptor(h3);
DftiComputeBackward(h3,f); // f が複素数値で復元される
```

ソースコード、イメージファイル、メイクファイル: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `fft-ct` フォルダーを参照してください。

説明

このコードは、最初に `DftiComputeForward` 関数の単一の呼び出しで 1 次元フーリエ変換のバッチを計算するインテル® MKL FFT ディスクリプターを構成した後、バッチ変換を計算します。 複数の変換の距離は、対応する領域 (入力は実数、出力は複素数) の要素で設定されます。 デフォルトでは、変換はインプレースです。

メモリーのフットプリントを小さくするため、FFT は「インプレース」で計算されます。つまり、計算の結果は入力データに上書きされます。インプレースの実数-複素数 FFT では、FFT の結果を格納するのに必要なメモリーが入力よりもやや多くなるため、入力配列は余分な空間を予約します。

ステップ 1 の入力で、配列 `g` には $p \times (2q+1)$ 実数値のデータ要素 $g(\theta_j, s_l)$ が含まれます。 このステップの出力の同じメモリーには、$p \times (q + 1)$ 複素数値の出力要素 $g_1(\theta_j, \pi r/q)$ が含まれます。 結果の複素共役部分は格納されません。そのため、配列 `g1` は、r の $q + 1$ 値のみを指します。

g_1 から f_2 に補間するため、ステップ 3 の逆 FFT の複素数値のデータ $f_2(\xi, \eta)$ と複素数値の出力 $f_1(x, y)$ を格納する追加の配列 `f` が割り当てられます。 補間ステップはインテル® MKL 関数を呼び出しません。C++ 実装は、この手法のソースコード (`main.cpp`) の `step2_interpolation` 関数を参照してください。 補間の最も単純な実装は次のとおりです。

- 単位円の内部のすべての (ξ, η) で、最も近い $(\theta_j, \pi r/q)$ を見つけて、$g_1(\theta_j, \pi r/q)$ の値を f_2 に使用します。
- 単位円の外部のすべての (ξ, η) で、f_2 を 0 に設定します。
- 区間 $-\pi < \theta_j < 0$ に対応する (ξ, η) の場合は、実数-複素数変換の結果の共役偶数プロパティー $g_1(\theta, \omega)=\mathrm{conj}(g(-\theta, -\omega))$ を使用します。

ステップ 1 の FFT は、計算された出力が $e^{i(\pi r/q)q} = (-1)^r$ で乗算される、表現区間の半分でオフセットされたデータに適用されることに注意してください。 個別のパスで修正する代わりに、補間は乗数を考慮しています。

同様に、ステップ 3 の 2D FFT はイメージの中心を隅にシフトする出力を生成し、ステップ 2 はステップ 3 に入力をシフトする過程でこれを防いでいます。

ステップ 3 は、配列 `f` に含まれる補間されたデータについて 2 次元 $(2q) \times (2q)$ 複素数-複素数 FFT を計算します。 この計算の後、複素数値のイメージ f_1 からビジュアルピクチャーへの変換が行われます。CT イメージ復元を実装する完全な C++ プログラムは、この手法のソースコード (`main.cpp`) を参照してください。

金融市場のデータストリームにおけるノイズ・フィルタリング

10

目的

一部の株式の価格変動が大規模な株式ポートフォリオのほかの株式の価格変動にどのように影響を及ぼすかを特定する。

ソリューション

データ全体の依存関係を表す相関行列を、信号行列とノイズ行列の 2 つの成分に分割します。 信号行列から、株式間の依存関係を正確に評価することができます。 アルゴリズム ([Zhang12]、[Kargupta02]) は、累積データの相関行列の固有値を調べることにより、有用な情報からノイズ成分を分割する固有状態ベースのアプローチを取ります。

インテル® MKL サマリー統計は、ストリーミング・データの相関行列を計算する機能を提供します。 インテル® MKL LAPACK には、さまざまな特性や格納形式の対称行列の固有値と固有ベクトルを計算する計算ルーチン群が含まれています。

オンライン・ノイズ・フィルタリング・アルゴリズムの手順を次に示します。

1. λ_{min} と λ_{max} (ノイズ固有状態の間隔の境界) を計算します。
2. データストリームから新しいデータブロックを取得します。
3. 最新のデータブロックを使用して相関行列を更新します。
4. 間隔 [λ_{min}, λ_{max}] に属する相関行列の固有値を検索して、ノイズ成分を定義する固有値と固有ベクトルを計算します。
5. ステップ 4 で計算した固有値と固有ベクトルを組み合わせてノイズ成分の相関行列を計算します。
6. 相関行列全体からノイズ成分を取り除いて信号成分の相関行列を計算します。 さらにデータがある場合は、ステップ 2 に戻ります。

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `nf` フォルダーを参照してください。

初期化

相関解析タスクとパラメーターを初期化します。

```
VSLSSTaskPtr task;
double *x, *mean, *cor;
double W[2];
MKL_INT x_storage, cor_storage;

...

scanf("%d", &m);            // ブロックの観測数
scanf("%d", &n);            // 株式の数 (タスクの次元)

...

/* メモリーを割り当て */
nfAllocate(m, n, &x, &mean, &cor, ...);
```

```
/* サマリー統計タスク構造を初期化 */
nfInitSSTask(&m, &n, &task, x, &x_storage, mean, cor, &cor_storage, W);
...

/* メモリーを割り当て */
void nfAllocate(MKL_INT m, MKL_INT n, double **x, double **mean, double **cor,
                ...)
{
    *x = (double *)mkl_malloc(m*n*sizeof(double), ALIGN);
    CheckMalloc(*x);

    *mean = (double *)mkl_malloc(n*sizeof(double), ALIGN);
    CheckMalloc(*mean);

    *cor = (double *)mkl_malloc(n*n*sizeof(double), ALIGN);
    CheckMalloc(*cor);
...
}
/* サマリー統計タスク構造を初期化 */
void nfInitSSTask(MKL_INT *m, MKL_INT *n, VSLSSTaskPtr *task, double *x,
                  MKL_INT *x_storage, double *mean, double *cor,
                  MKL_INT *cor_storage, double *W)
{
    int status;

    /* VSL サマリー統計タスクを作成 */
    *x_storage = VSL_SS_MATRIX_STORAGE_COLS;
    status = vsldSSNewTask(task, n, m, x_storage, x, 0, 0);
    CheckSSError(status);

    /* タスクの重みのレジスター配列 */
    W[0] = 0.0;
    W[1] = 0.0;
    status = vsldSSEditTask(*task, VSL_SS_ED_ACCUM_WEIGHT, W);
    CheckSSError(status);

    /* 相関行列計算にフルストレージを使用する
       タスク・パラメーターの初期化 */
    *cor_storage = VSL_SS_MATRIX_STORAGE_FULL;
    status = vsldSSEditCovCor(*task, mean, 0, 0, cor, cor_storage);
    CheckSSError(status);
}
```

計算

データの各ブロックについてノイズ・フィルタリングのステップを実行します。

```
/* ノイズ成分を定義するしきい値を設定 */
sqrt_n_m = sqrt((double)n / (double)m);
lambda_min = (1.0 - sqrt_n_m) * (1.0 - sqrt_n_m);
lambda_max = (1.0 + sqrt_n_m) * (1.0 + sqrt_n_m);

...
/* データブロックをループ */
for (i = 0; i < n_block; i++)
{
    /* データの次の部分を読む */
```

```
    nfReadDataBlock(m, n, x, fh);

    /* "信号" と "ノイズ" 共分散の評価を更新 */
    nfKernel(m, n, lambda_min, lambda_max, x, cor, cor_copy,
             task, eval, evect, work, iwork, isuppz,
             cov_signal, cov_noise);
}
...

void nfKernel(…)
{
...

    /* FAST メソッドを使用して相関行列の評価を更新 */
    errcode = vsldSSCompute(task, VSL_SS_COR, VSL_SS_METHOD_FAST);
    CheckSSError(errcode);

...
    /* 相関行列の固有値と固有ベクトルを計算 */
    dsyevr(&jobz, &range, &uplo, &n, cor_copy, &n, &lmin, &lmax,
           &imin, &imax, &abstol, &n_noise, eval, evect, &n, isuppz,
           work, &lwork, iwork, &liwork, &info);

    /* 共分散行列の "信号" と "ノイズ" 部分を計算 */
    nfCalculateSignalNoiseCov(n, n_signal, n_noise,
        eval, evect, cor, cov_signal, cov_noise);
}

...
static int nfCalculateSignalNoiseCov(int n, int n_signal, int n_noise,
        double *eval, double *evect, double *cor, double *cov_signal,
        double *cov_noise)
{
    int i, j, nn;

    /* SYRK パラメーター */
    char uplo, trans;
    double alpha, beta;

    /* 共分散行列の "ノイズ" 部分を計算 */
    for (j = 0; j < n_noise; j++) eval[j] = sqrt(eval[j]);

    for (i = 0; i < n_noise; i++)
        for (j = 0; j < n; j++)
            evect[i*n + j] *= lambda[i];

    uplo  = 'U';
    trans = 'N';
    alpha = 1.0;
    beta  = 0.0;
    nn = n;

    if (n_noise > 0)
    {
        dsyrk(&uplo, &trans, &nn, &n_noise,  &alpha, evect, &nn,
              &beta, cov_noise, &nn);
    }
```

```
    else
    {
        for (i = 0; i < n*n; i++) cov_noise[i] = 0.0;
    }

    /* 共分散行列の "信号" 部分を計算 */
    if (n_signal > 0)
    {
        for (i = 0; i < n; i++)
            for (j = 0; j <= i; j++)
                cov_signal[i*n + j] = cor[i*n + j] - cov_noise[i*n + j];
    }
    else
    {
        for (i = 0; i < n*n; i++) cov_signal[i] = 0.0;
    }

    return 0;
}
```

リソースの解放

タスクを削除し、関連付けられたリソースを解放します。

```
errcode = vslSSDeleteTask(task);
CheckSSError(errcode);
MKL_Free_Buffers();
```

使用するルーチン

タスク	ルーチン	説明
サマリー統計タスクを初期化して解析用のオブジェクト (データセット、サイズ (変数の数と観測の数)、格納形式)) を定義する	vsldSSNewTask	新しいサマリー統計タスク・ディスクリプターを作成して初期化します。
相関行列を保持するメモリーを指定する	vsldSSEditCovCor	共分散/相関/クロス積パラメーターのポインターを変更します。
処理した観測の重みの合計を保持する 2 要素の配列を指定する (データストリームの評価の正確な計算に必要)	vsldSSEditTask	タスク・ディスクリプターの入出力パラメーターのアドレスを変更します。
計算タイプ (VSL_SS_COR) と計算メソッド (VSL_SS_METHOD_FAST) を指定して主計算ドライバーを呼び出す	vsldSSCompute	サマリー統計評価を計算します。
タスクに関連付けられているリソースの割り当てを解除する	vslSSDeleteTask	タスク・オブジェクトを破棄してメモリーを解放します。

タスク	ルーチン	説明
相関行列の固有値と固有ベクトルを計算する	dsyevr	選択した固有値を (オプションで、Relatively Robust Representation アルゴリズムを使用して実対称行列の固有値を) 計算します。
対称ランク-k 更新を行う	dsyrk	対称ランク-k 更新を行います。

説明

アルゴリズムのステップ 4 では、対称行列の固有値問題を解きます。 オンライン・ノイズ・フィルタリング・アルゴリズムを使用するには、データのノイズを定義する、事前定義間隔 [λ_{min}, λ_{max}] に属する固有値を計算する必要があります。LAPACK ドライバールーチン ?syevr は、この種の問題を解くデフォルトのルーチンです。?syevr インターフェイスで、呼び出し元は、固有値を検索する範囲の下限と表現のペア (このケースでは λ_{min} と λ_{max}) を指定できます。

固有ベクトルは直交行列 *A* を含む配列の列として返され、固有値は対角行列 *Diag* の要素を含む配列で返されます。ノイズ成分の相関行列は、$A*Diag*A^T$ を計算して取得できます。しかし、ここでは、2 つの一般的な行列乗算によってノイズ相関行列を構築する代わりに、1 つの対角行列乗算と 1 つのランク n 更新操作を使用してより効率良く計算しています。

$$Cor_{noise} = ADiagA^{\mathrm{T}} = A\sqrt{Diag}\,\sqrt{Diag}^{\mathrm{T}}A^{\mathrm{T}} = \left(A\sqrt{Diag}\right)\left(A\sqrt{Diag}\right)^{\mathrm{T}}$$

ランク n 更新操作用に、インテル® MKL は BLAS 関数 ?syrk を提供しています。

モンテカルロ法を使用したヨーロピアン・オプションの価格計算

11

目的

nsamp 個のサンプルを基にヨーロピアン・オプション (コールおよびプット) の価格 (*nopt*) を計算します。

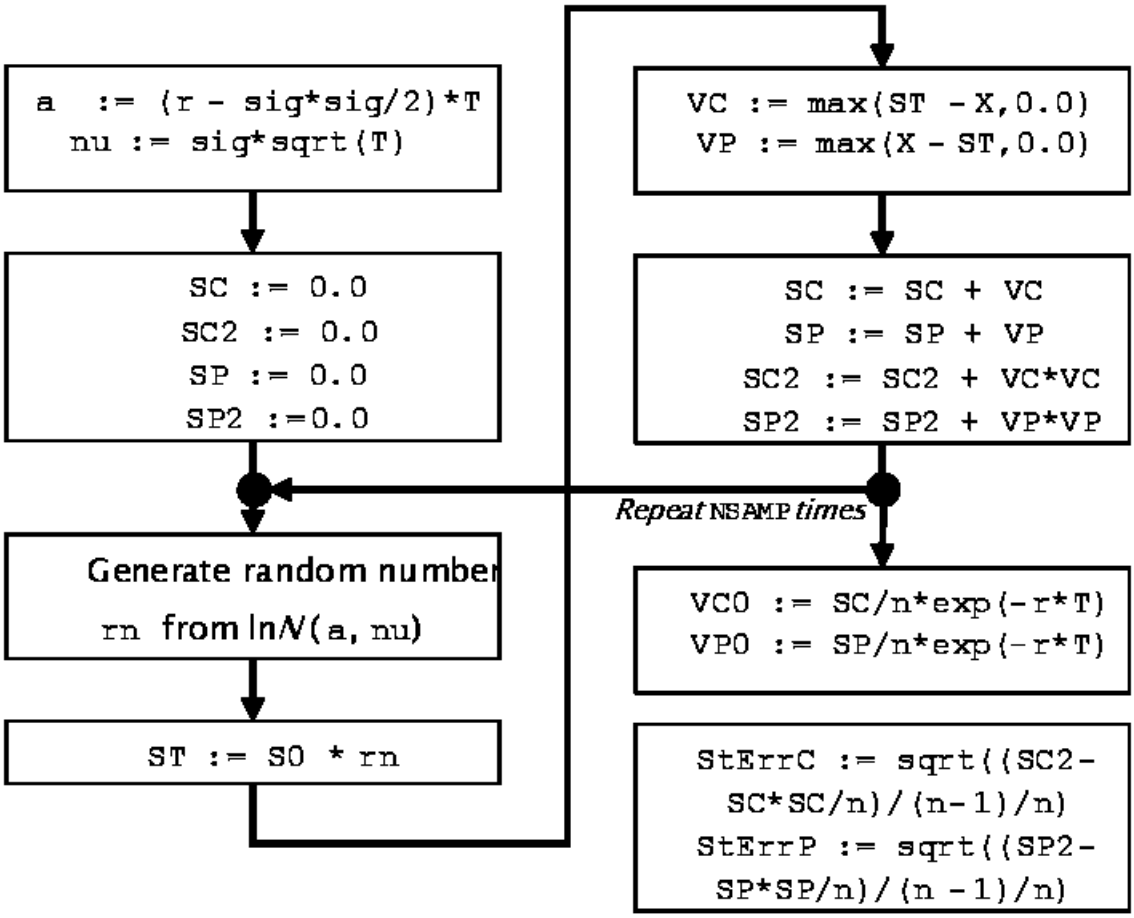
ソリューション

モンテカルロ・シミュレーションを使用してヨーロピアン・オプションの価格を計算します。コールオプションとプットオプションのペアは、次のように計算します。

1. 初期化します。
2. オプションの価格を並列に計算します。
3. コール価格とプット価格のペアの計算をブロックに分割します。
4. ブロックの計算を実行します。
5. リソースを解放します。

注

OS X* でこのソリューションを利用するには、インテル® MKL 11.2 Update 3 以上が必要です。



ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `mc` フォルダーを参照してください。

OpenMP* セクションの初期化

OpenMP* 並列セクションを作成し、MT2203 乱数ジェネレーターを初期化します。

```
#pragma omp parallel
  {
      ...
      VSLStreamStatePtr stream;

      j = omp_get_thread_num();

      /* RNG を初期化*/
      vslNewStream( &stream, VSL_BRNG_MT2203 + j, SEED );
      ...
  }
```

この初期化モデルは、各スレッドで個別の乱数ストリームを生成します。

オプション価格の並列計算

利用可能なスレッドにオプションを分配します。

```
#pragma omp parallel
  {
      ...

      /* オプションの価格付け */
    #pragma omp for
      for(i=0;i<nopt;i++)
      {
          MonteCarloEuroOptKernel( ... );
      }

      ...
  }
```

コール価格とプット価格のペアの計算をブロックに分割

パスの生成をブロックに分割し、データの局所性を保持してパフォーマンスを最適化します。

```
  const int nbuf = 1024;
  nblocks = nsamp/nbuf;
  ...
  /* ブロックの計算 */
  for ( i = 0; i < nblocks; i++ )
  {
      /* 末尾が正しく計算されることを保証 */
      int block_size = (i != nblocks-1)?(nbuf):(nsamp - (nblocks-1)*nbuf);
      ...
  }
```

ブロックの計算の実行

メインの計算で、乱数を生成し、リダクションを実行します。

```
  /* ブロックの計算 */
  for ( i = 0; i < nblocks; i++ )
  {
      ...

      /* 乱数のブロックを生成 */
      vdRngLognormal( VSL_RNG_METHOD_LOGNORMAL_ICDF, stream,
                      block_size, rn, a, nu, 0.0, 1.0 );

      /* リダクション */
    #pragma vector aligned
    #pragma simd
      for ( j=0; j<block_size; j++ )
      {
          st = s0*rn[j];
          vc = MAX( st-x, 0.0 );
          vp = MAX( x-st, 0.0 );
          sc += vc;
          sp += vp;
      }
  }
```

```
*vcall = sc/nsamp * exp(-r*t);
*vput  = sp/nsamp * exp(-r*t);
```

リソースの解放

RNG ストリームを削除します。

```
#pragma omp parallel
  {
      ...
      VSLStreamStatePtr stream;
      ...
      /* RNG を削除 */
      vslDeleteStream( &stream );
  }
```

使用するルーチン

タスク	ルーチン
ランダムストリームを作成して初期化する	vslNewStream
対数正規型に分布している乱数を生成する	vdRngLogNormal
ランダムストリームを削除する	vslDeleteStream

説明

モンテカルロ・シミュレーションは、ランダム・サンプリングを繰り返すことでモデルの特性を決定する、広く使用されている手法です。 モンテカルロ・シミュレーションによるヨーロピアン・オプションの価格付けは、簡単な金融ベンチマークであり、モンテカルロ・シミュレーションを使用する実際のアプリケーションの取り掛かりとして利用できます。

S_t は、次の確率過程が適用される t 時点の株価とします。

$dS_t = \mu S_t dt + \sigma S_t dW_t$, S_0

ここで、μ は *ドリフト*、σ は *ボラティリティー* (定数と仮定)、$W = (W_t)_{t\geq 0}$ は *Wiener 過程*、dt は時間ステップで、S_0 ($t = 0$ の株価) は X に依存しません。

予想値は、$E(S_t) = S_0 \exp(rt)$ で定義され、r はリスク中立レートです。 前述の S_t の定義から $E(S_t) = S_0 \exp((\mu + \sigma^2/2)t)$、$\mu = r - \sigma^2/2$ となります。

$0 \leq t \leq T$ におけるヨーロピアン・オプションの値 $V(t, S_t)$ は、株価 S_t に依存します。 オプションは $t = 0$ に発行され、$t = T$ (*満了日*) に行使されます。ヨーロピアン・オプション (*コール*および*プット*) の場合、満了日のオプション価格 $V(T, S_T)$ は次のように定義されます。

- コールオプション: $V(T, S_T) = \max(S_T - X, 0)$
- プットオプション: $V(T, S_T) = \max(X - S_T, 0)$

X は *権利行使価格*です。ここでは、$V(0, S_0)$ を評価します。

モンテカルロ法を使用するこの問題の解は、S_T の n 個の可能性をシミュレーションし、$V(T, S_T)$ の平均を係数 $\exp(-rt)$ で割り引いて、オプションの現在の価格 $V(0, S_0)$ を取得します。最初の式から S_T は対数正規型に分布しています。

$$S_T = S_0 \exp\left((r - \sigma^2 / 2)T + \sigma\sqrt{T}\xi\right)$$

ξ は標準正規分布の確率変数です。

インテル® MKL の基本乱数ジェネレーターは (BRNG) は、さまざまなパラメーター・セット、ブロック分割、リープフロッギングの使用など、多様な並列計算モデルをサポートします。

この例は、6024 種類のパラメーター・セットをサポートする MT2203 BRNG を使用します。 ストリームの初期化関数で、*j* を BRNG 識別子 `VSL_BRNG_MT2203` に加えて、パラメーター・セットを選択します。

```
vslNewStream( &stream, VSL_BRNG_MT2203 + j, SEED );
```

インテル® MKL の VS BRNG 実装でサポートされる並列モデルについては、「Intel® MKL Vector Statistics Notes」を参照してください。

計算ブロックのサイズ (この例では 1024) は、ブロック内のメモリーアクセス量に依存します。通常、ブロック内のすべてのメモリーアクセスがターゲット・プロセッサーのキャッシュに収まるように計算ブロックのサイズを選択します。

ブラックショールズ方程式を使用したヨーロピアン・オプションの価格計算

12

目的

ブラックショールズ方程式を使用したヨーロピアン・オプションの価格計算をスピードアップする。

ソリューション

インテル® MKL ベクトルマス関数を使用して計算をスピードアップします。

ブラックショールズ・モデルは、市場の行動を連立確率微分方程式で表します [Black73]。 この市場で発行されるヨーロピアン・オプション (コールおよびプット) は、ブラックショールズ方程式に従って価格付けされます。

$$V_{\text{call}} = S_0 \cdot \text{CDF}(d_1) - e^{-rT} \cdot X \cdot \text{CDF}(d_2)$$
$$V_{\text{put}} = e^{-rT} \cdot X \cdot \text{CDF}(-d_2) - S_0 \cdot \text{CDF}(-d_1)$$

ここで、

$$d_1 = \frac{\ln\left({}^{S_0}/_{X}\right) + \left(r + {}^{\sigma^2}/_{2}\right)T}{\sigma\sqrt{T}}$$
$$d_2 = \frac{\ln\left({}^{S_0}/_{X}\right) + \left(r - {}^{\sigma^2}/_{2}\right)T}{\sigma\sqrt{T}}$$

V_{call} /V_{put} はコール/プットオプションの現在の値、S_0 は現在の株価、X は権利行使価格、r はリスク中立レート、σ はボラティリティー、T は満期日、CDF は標準正規分布の累積分布関数です。

累積正規分布関数との関係が単純な誤差関数 ERF を使用することもできます。

$$\text{CDF}(x) = {}^{1}/_{2} + {}^{1}/_{2} \cdot \text{ERF}\left({}^{x}/_{\sqrt{2}}\right)$$

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `black-scholes` フォルダーを参照してください。

ブラックショールズの単純実装

このコードは、コールおよびプットオプションの価格計算にクローズド形式のソリューションを実装します。

```
void BlackScholesFormula( int nopt,
    tfloat r, tfloat sig, const tfloat s0[], const tfloat x[],
    const tfloat t[], tfloat vcall[], tfloat vput[] )
{
    tfloat d1, d2;
    int i;

    for ( i=0; i<nopt; i++ )
    {
        d1 = ( LOG(s0[i]/x[i]) + (r + HALF*sig*sig)*t[i] ) /
             ( sig*SQRT(t[i]) );
        d2 = ( LOG(s0[i]/x[i]) + (r - HALF*sig*sig)*t[i] ) /
             ( sig*SQRT(t[i]) );

        vcall[i] = s0[i]*CDFNORM(d1) - EXP(-r*t[i])*x[i]*CDFNORM(d2);
        vput[i] = EXP(-r*t[i])*x[i]*CDFNORM(-d2) - s0[i]*CDFNORM(-d1);
    }
}
```

オプションの数は *nopt* パラメーターで指定します。 `tfloat` の型は、使用する精度に応じて `float` または `double` のいずれかです。同様に、`LOG`、`EXP`、`SQRT`、`CDFNORM` を数学関数の単精度または倍精度バージョンにマップします。定数 `HALF` は 0.5f (単精度) または 0.5 (倍精度) のいずれかです。

nopt に加えて、アルゴリズムの入力パラメーターは、*s0* (現在の株価)、*r* (リスク中立レート)、*sig* (ボラティリティー)、*t* (満期日)、*x* (権利行使価格) です。結果は *vcall* および *vput* (コールオプションとプットオプションの現在の値) で返されます。

r および *sig* は価格付けを行うすべてのオプションの定数であり、ほかのパラメーターは浮動小数点値の配列であると仮定しています。*vcall* および *vput* パラメーターは出力配列です。

説明

超越関数は、ブラックショールズ方程式ベンチマークの中心となるものです。しかし、各オプションの値は 5 つのパラメーターに依存しており、計算が速くなるとメモリー効率がより影響するようになります。そのため、配列パラメーターの数は計算による制限からメモリー帯域幅による制限に計算を変更する重要な要因です。さらに、価格計算のオプションが膨大な場合、メモリーサイズの制約を考慮すべきです。

インテル® C++ コンパイラーは、ブラックショールズ方程式に関してインテル® アーキテクチャーの SIMD およびマルチコア性能を引き出す、ベクトル化と並列化機能を提供します。最適化されベクトル化された数学関数は、SVML (Short Vector Mathematical Library) ランタイム・ライブラリーで利用可能です。

インテル® MKL ベクトルマス (VM) 関数は、式の計算をさらにスピードアップするために使用できる、高度にチューニングされた超越数学関数を提供します。

コードを最適化する機会は、共通の部分式のループ外への移動、`CDFNORM` 関数から `ERF` 関数 (通常はより高速) への置換、関係 `ERF(-x) = -ERF(x)` の活用、`SQRT` による除算から平方根の逆数 `INVSQRT` による乗算 (通常はより高速) への置換、などいくつかあります。

ブラックショールズの最適化実装

```
void BlackScholesFormula( int nopt,
    tfloat r, tfloat sig, tfloat s0[], tfloat x[],
    tfloat t[], tfloat vcall[], tfloat vput[] )
{
    int i;
```

```
    tfloat a, b, c, y, z, e;
    tfloat d1, d2, w1, w2;
    tfloat mr = -r;
    tfloat sig_sig_two = sig * sig * TWO;

    for ( i = 0; i < nopt; i++ )
    {
        a = LOG( s0[i] / x[i] );
        b = t[i] * mr;
        z = t[i] * sig_sig_two;

        c = QUARTER * z;
        e = EXP ( b );
        y = INVSQRT( z );

        w1 = ( a - b + c ) * y;
        w2 = ( a - b - c ) * y;
        d1 = ERF( w1 );
        d2 = ERF( w2 );
        d1 = HALF + HALF*d1;
        d2 = HALF + HALF*d2;

        vcall[i] = s0[i]*d1 - x[i]*e*d2;
        vput[i]  = vcall[i] - s0[i] + x[i]*e;
    }
}
```

このコードで、`INVSQRT(x)` は精度に応じて `1.0/sqrt(x)` または `1.0f/sqrtf(x)` のいずれかです。`TWO` および `QUARTER` はそれぞれ、浮動小数点定数 2 と 0.25 です。

説明

いくつかの最適化はインテル® C/C++ コンパイラーを使用した効率的なコードの生成に役立ちます。このセクションで推奨しているコンパイラーのプラグマおよびスイッチに関する詳細は、『インテル® C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』を参照してください。

`#pragma simd` は、ループをベクトル化するようにコンパイラーに伝えます。また、`#pragma vector aligned` は、配列がアライメントされていて (メモリー割り当て段階でベクトルを適切にアライメントする必要があります) アライメント済みロード命令およびストア命令が安全であることをコンパイラーに伝えます。SVML で利用可能な、効率的なベクトル化を行うことにより、スカラーコードの数倍のスピードアップを達成できます。

```
…
#pragma simd
#pragma vector aligned
for ( i = 0; i < nopt; i++ )
…
```

これらの変更により、すべての利用可能な CPU コアを活用できます。 最も簡単な方法は、コンパイラーがコードを自動的に並列化するように、コンパイル行に `-autopar` スイッチを追加することです。別のオプションは、標準 OpenMP* プラグマを使用することです。

```
…
#pragma omp parallel for
for ( i = 0; i < nopt; i++ )
…
```

`-fp-model fast`、`-no-prec-div`、`-ftz`、`-fimf-precision`、`-fimf-max-error`、`-fimf-domain-exclusion` などのインテル® C++ コンパイラー・オプションを使用して数学関数の精度を緩和できる場合、さらにパフォーマンスを向上することが可能です。

注

Linux* 固有のコンパイラー・スイッチも用意されています。詳細は、『インテル® C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』を参照してください。

並列度が非常に高い場合、数学関数の計算時間がメモリー帯域幅よりも低くなり、ループのパフォーマンスを制限することがあります。この制限は、並列化による直線的なスピードアップの障害となります。その場合は、メモリー帯域幅の処理に優れた非テンポラルなロード/ストア命令を使用します。

```
…
#pragma vector nontemporal
for ( i = 0; i < nopt; i++ )
…
```

インテル® MKL VM コンポーネントは、さらなるパフォーマンス向上に役立つ高度にチューニングされた超越数学関数を提供します。しかし、これらの関数を使用するには、VM API のベクトル機能に対応するコードのリファクタリングが必要になります。次のコードサンプルでは、自明でない数学関数は VM の関数を、残りの基本的な演算はコンパイラーの関数を利用しています。

ベクトルマス計算の中間結果を保持するため、一時バッファーが関数のスタックに割り当てられます。バッファーを適用可能な最大の SIMD レジスターサイズでアライメントすることが重要です。バッファーサイズは、VM 関数が (ベクトル関数のスタートアップにかかるコストを補い) 最適なパフォーマンスを達成でき、データがキャッシュに収まるように選択されます。バッファーサイズは実験して決定できます。推奨する開始サイズは `NBUF=1024` です。

ブラックショールズのインテル® MKL VM 実装

```
void BlackScholesFormula_MKL( int nopt,
    tfloat r, tfloat sig, tfloat * s0, tfloat * x,
    tfloat * t, tfloat * vcall, tfloat * vput )
{
    int i;
    tfloat mr = -r;
    tfloat sig_sig_two = sig * sig * TWO;

    #pragma omp parallel for                                  \
        shared(s0, x, t, vcall, vput, mr, sig_sig_two, nopt) \
        default(none)
    for ( i = 0; i < nopt; i+= NBUF )
    {
        int j;
        tfloat *a, *b, *c, *y, *z, *e;
        tfloat *d1, *d2, *w1, *w2;
        __declspec(align(ALIGN_FACTOR)) tfloat Buffer[NBUF*4];
        // nopt が NBUF の倍数でなかった場合
        // ループの最後の反復のベクトル長を計算
        #define MY_MIN(x, y) ((x) < (y)) ? (x) : (y)
        int nbuf = MY_MIN(NBUF, nopt - i);

        a      = Buffer + NBUF*0;          w1 = a; d1 = w1;
        c      = Buffer + NBUF*1;          w2 = c; d2 = w2;
        b      = Buffer + NBUF*2; e = b;
        z      = Buffer + NBUF*3; y = z;


        // 各スレッドに VM の精度を設定
        vmlSetMode( VML_ACC );
```

```
        VDIV(nbuf, s0 + i, x + i, a);
        VLOG(nbuf, a, a);

        #pragma simd
        for ( j = 0; j < nbuf; j++ )
        {
            b[j] = t[i + j] * mr;
            a[j] = a[j] - b[j];
            z[j] = t[i + j] * sig_sig_two;
            c[j] = QUARTER * z[j];
        }

        VINVSQRT(nbuf, z, y);
        VEXP(nbuf, b, e);

        #pragma simd
        for ( j = 0; j < nbuf; j++ )
        {
            tfloat aj = a[j];
            tfloat cj = c[j];
            w1[j] = ( aj + cj ) * y[j];
            w2[j] = ( aj - cj ) * y[j];
        }

        VERF(nbuf, w1, d1);
        VERF(nbuf, w2, d2);

        #pragma simd
        for ( j = 0; j < nbuf; j++ )
        {
            d1[j] = HALF + HALF*d1[j];
            d2[j] = HALF + HALF*d2[j];
            vcall[i+j] = s0[i+j]*d1[j] - x[i+j]*e[j]*d2[j];
            vput[i+j]  = vcall[i+j] - s0[i+j] + x[i+j]*e[j];
        }
    }
}
```

同等の精度では、計算により制限される問題 (データが L2 キャッシュに収まる) の場合、インテル® MKL VM はインテル® コンパイラーと SVML ベースのソリューションよりも 30-50% 優れたパフォーマンスを実現します。この場合、キャッシュ読み取り/書き込み操作のレイテンシーは計算によりマスクされます。問題サイズが大きくなりメモリー帯域幅により制限されるようになると、メモリー使用量の最適化はさらに重要となり、中間バッファーを利用するインテルの VM ソリューションは SVML を利用したバッファリングなしの 1 パス・ソリューションに対する優位性を失います。

使用するルーチン

タスク	ルーチン
mode パラメーターに従って VM 関数に新しい精度モードを設定し、以前の VM モードを oldmode に格納する	`vmlSetMode`
ベクトル a とベクトル b の要素単位の除算を実行する	`vsdiv/vddiv`
ベクトル要素の自然対数を計算する	`vsln/vdln`

タスク	ルーチン
ベクトル要素の逆平方根を計算する	vsinvsqrt/vdinvsqrt
ベクトル要素の指数を計算する	vsexp/vdexp
ベクトル要素の誤差関数の値を計算する	vserf/vderf

置換のない複数の単純なランダム・サンプリング

13

目的

サイズ *N* (1 ≤*M*≤*N*) の母集団から置換なしで *K*>>1 の単純なランダム長 *M* のサンプルを生成する。

ソリューション

問題の定義と詳細は、[SRSWOR] を参照してください。

部分 Fisher-Yates シャッフル・アルゴリズム [KnuthV2] の次の実装とインテル® MKL 乱数ジェネレーター (RNG) を使用して各サンプルを生成します。

部分 Fisher-Yates シャッフル・アルゴリズム

```
A2.1: (初期化ステップ) PERMUT_BUF は自然数 1, 2, ..., N を含むものとする

 A2.2: for i from 1 to M do:

     A2.3: {i,...,N} で一様なランダムな整数を生成

     A2.4: PERMUT_BUF[i] と PERMUT_BUF[X] を交換

 A2.5: (コピーステップ) for i from 1 to M do: RESULTS_ARRAY[i]=PERMUT_BUF[i]

 End.
```

アルゴリズムを実装するプログラムは 11 969 664 の実験を行います。 各実験 (1 から *N* までの *M* の一様なランダム自然数のシーケンスを生成) は、実際には *N* 要素の母集団から部分長 *M* をランダムにシャッフルします。アルゴリズムのメインループは実際に抽選を行うため、プログラムで各実験は「*N* から *M* の抽選」と呼ばれています。

プログラムは *M*=6 および *N*=49 を使用して、結果のサンプル (長さ *M* のシーケンス) を単一配列 `RESULTS_ARRAY` に格納し、利用可能な並列スレッドをすべて使用します。

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `lottery6of49` フォルダーを参照してください。

並列化

```
#pragma omp parallel
{
    thr = omp_get_thread_num(); /* スレッドのインデックス */
    VSLStreamStatePtr stream;
    /* このスレッドの RNG ストリームを初期化 */
    vslNewStream( &stream, VSL_BRNG_MT2203+thr, seed );
```

```
    ... /* (スレッド番号 thr で) 実験サンプルを生成 */
    vslDeleteStream( &stream );
}
```

コードは、OpenMP* #pragma parallel プラグマを使用して、すべての CPU のすべての利用可能なプロセッサー・コアを利用します。実験結果の配列 RESULTS_ARRAY は THREADS_NUM の部分に分割されます。ここで、THREADS_NUM は利用可能なスレッド数で、各スレッド (並列領域) は配列の個別の部分を処理します。

インテル® MKL 基本乱数ジェネレーターは、VSL_BRNG_MT2203 パラメーターを指定することにより、各スレッドで並列の独立したストリームをサポートします。

実験サンプルの生成

```
/* A2.1: 初期化ステップ */
/*  PERMUT_BUF は自然数 1, 2, ..., N を含むものとする */
 for( i=0; i<N; i++ ) PERMUT_BUF[i]=i+1; /* セット {1,...,N} を使用 */
 for( sample_num=0; sample_num<EXPERIM_NUM/THREADS_NUM; sample_num++ ){
     /* 次の抽選サンプルを生成 (ステップ A2.2、A2.3、A2.4): */
     Fisher_Yates_shuffle(...);
     /*  A2.5: コピーステップ */
     for(i=0; i<M; i++)
         RESULTS_ARRAY[thr*ONE_THR_PORTION_SIZE + sample_num*M + i] = PERMUT_BUF[i];
 }
```

このコードは各スレッドで部分 Fisher-Yates シャッフル・アルゴリズムを実装します。

多くの実験をシミュレーションする場合、初期化ステップは各実験の最初に 1 回のみ必要です。(実際の抽選のように) PERMUT_BUF 配列の自然数 1...*N* の順序は重要ではありません。

Fisher_Yates_shuffle 関数

```
/* A2.2: for i from 0 to M-1 do */
Fisher_Yates_shuffle (...)
{
    for(i=0; i<M; i++) {
        /* A2.3: {i,...,N-1} からランダム自然数 X を生成 */
        j = Next_Uniform_Int(...);
        /* A2.4: PERMUT_BUF[i] と PERMUT_BUF[X] を交換 */
        tmp = PERMUT_BUF[i];
        PERMUT_BUF[i] = PERMUT_BUF[j];
        PERMUT_BUF[j] = tmp;
    }
}
```

ループ A2.2 の各反復は、実際に抽選を行います。残りのアイテム PERMUT_BUF[i], ..., PERMUT_BUF[N] を含む箱からランダムなアイテム X を取り出し、アイテム X を結果行 PERMUT_BUF[1],...,PERMUT_BUF[i] の最後に追加します。長さ *N* の完全な置換ではなく、一部の長さ *M* のみを生成するため、アルゴリズムは部分的です。

注

アルゴリズムで説明した擬似コードと異なり、プログラムはゼロベースの配列を使用します。

説明

ステップ A2.3 で、プログラムは Next_Uniform_Int 関数を呼び出して次のランダム整数 X を生成し、{*i*, ..., *N*-1} で一様にします (詳細はソースコードを参照)。インテル® MKL のベクトル化された RNG の能力を引き出しつつ、ベクトル化のオーバーヘッドを最小限に抑えるため、ジェネレーターは L1 キャッシュに収まるサイズ RNGBUFSIZE のベクトル D_UNIFORM01_BUF を生成する必要があります。 各スレッドは、独自のバッファー D_UNIFORM01_BUF およびそのバッファーで最後に使用された乱数の後を指すインデックス D_UNIFORM01_IDX を使用します。Next_Uniform_Int 関数の最初の呼び出しでは (あるいはバッファーの乱数がすべて使用された場合は)、長さ RNGBUFSIZE とインデックス D_UNIFORM01_IDX をゼロにセットして vdRngUniform 関数を呼び出し、乱数バッファーを生成し直します。

```
vdRngUniform( ... RNGBUFSIZE, D_UNIFORM01_BUF ... );
```

インテル® MKL は同じ分布の乱数値ジェネレーターのみ提供しますが、ステップ A2.3 では異なる範囲のランダム整数が必要なため、[0;1) で一様に分布された倍精度乱数をバッファーに格納した後、「整数のスケーリング」ステップで、これらの倍精度値を必要な整数範囲に収まるように変換します。

```
number 0   distributed on {0,...,N-1}   = 0   + {0,...,N-1}
number 1   distributed on {1,...,N-1}   = 1   + {0,...,N-2}

...

number M-1 distributed on {M-1,...,N-1} = M-1 + {0,...,N-M}

(前の M ステップを繰り返す)

number M     distributed on: see (0)
number M+1   distributed on: see (1)

...

number 2*M-1 distributed on: see (M-1)
(前の M ステップを再度繰り返す)
...

続く
```

整数スケーリング

```
/* 整数スケーリング・ステップ */
for(i=0;i<RNGBUFSIZE/M;i++)
    for(k=0;k<M;k++)
        I_RNG_BUF[i*M+k] =
            k + (unsigned int)(D_UNIFORM01_BUF[i*M+k] * (double)(N-k));
```

ここで RNGBUFSIZE は *M* の倍数。

このコードに関するパフォーマンスの注意点は [SRSWOR] を参照してください。

使用するルーチン

タスク	ルーチン
RNG ストリームを作成して初期化する	vslNewStream

タスク	ルーチン
区間 [0;1) に一様に分散された倍精度数を生成する	vdRngUniform
RNG ストリームを削除する	vslDeleteStream
64 ビット境界でアライメントされたメモリーバッファーを割り当てる	mkl_malloc
mkl_malloc で割り当てられたメモリーを解放する	mkl_free

最適化に関する注意事項

インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。

注意事項の改訂 #20110804

ヒストスプライン手法を使用した画像スケーリング

14

目的

ヒストスプライン手法を使用してカラーまたはグレースケール画像を再スケーリングする。

ソリューション

ヒストスプライン計算の画像スケーリングおよびスプライン補間にインテル® MKL のデータ・フィッティング関数を使用する。

[Bosner14] に記述されているように、1 次元ヒストスプラインは 2 次元画像スケーリングの問題に適用できます。2 次元（サーフェス）補間および近似に対する 1 次元補間プリミティブのアプリケーションは、[Zavyalov80] に記述されています。

多くの 1 次元ヒストポレーション問題に縮小できる 2 次元問題として、サイズ *n1***m1* ピクセル (ここで、*n1* は行数で *m1* は列数) の入力画像を *n2***m2* ピクセルの画像にスケーリングする問題について考えます。

入力画像の各色平面 *c* で、以下の操作を行います。

1. 入力画像の各行で、以下の操作を行います。
   - **a.** 現在の行の各ピクセルで、累計のセル平均 (または色の値) を計算して、配列 `VX` の現在の行に格納します。
   - **b.** 配列を補間された関数値、[0; *m1*] に一様に分布された (*m1*+1) ブレークポイント `x_breaks[]` として考えます。 自然 3 次スプラインを使用してヒストスプラインを構築し、その微分を計算して、配列 `VXR` の現在の行、[0; *m1*] に一様に分布された (*m2*+1) サイト `x_sites[]` の *i*=0..*m*2 に格納します。
2. 配列 `VXR` を配列 `VXRT` に転置します。
3. `VX` と同様に、`VY` の列ごとに同じ操作を行って配列 `VYR` の結果を取得します。
   - **a.** ステップ 1.a と同様に、計算した `VXRT` の値の累計として配列 `VY` を計算します。

     注

     このステップは、`VXR` の転置と同時に実行できるため、結果を `VXRT` に格納する必要はありません。

   - **b.** `VYR` 配列に微分を格納します。
4. `VYR` から `VR` に転置して整数結果を取得し、出力画像の色平面 *c* に格納します。

   注

   `VR` を格納する必要はありません。また、最後の行および列は破棄されます。

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の image_scaling フォルダーを参照してください。

インテル® MKL のデータ・フィッティング関数を使用した画像スケーリング

```
for( c=0; c<nc; c++ ) {
    /* 1) ピクセル行列 n1*m1 --> VX 行列 n1*(m1+1) */
    for( y1=0; y1<n1; y1++ ) {
        /* 1.a) ピクセル強度の累計として VX を取得 */
        for( x1=0, s=0; x1<=m1; x1++ ) {
            VX[y1*(m1+1)+x1]=(fptype)s;
            s+=row_pointers1[y1][x1*nc+c];
        }
        VX[y1*(m1+1)+x1]=(fptype)s;
    }
    for( y1=0; y1<n1; y1++ ) {
        /* 1.b) 微分のみ取得 */
        interpolate_der(m1+1,x_breaks, 1,&VX[y1*(m1+1)+0], m2+1,x_sites, &VXR[y1*(m2+1)+0]);
    }
    /* 2) VXR から VXRT の転置は必要なし (省略可能) */
    /* 3) */
    for( x2=0; x2<=m2; x2++ ) {
        /* 3.a) VXR の転置; 計算した値の累計として VY を取得 */
        for( y1=0,fs=0.0; y1<=n1; y1++ ) {
            VY[x2*(n1+1)+y1]=fs; fs+=VXR[y1*(m2+1)+x2];
        }
        VY[x2*(n1+1)+y1]=fs;
    }
    /* 3.b) 微分のみ取得 */
    for( x2=0; x2<=m2; x2++ ) {
        interpolate_der(n1+1,y_breaks, 1,&VY[x2*(n1+1)+0], n2+1,y_sites, &VYR[x2*(n2+1)+0]);
    }
    /* 4) VYR を VR に転置して整数結果を取得 (VR を格納する必要なし) */
    for( y2=0; y2<n2; y2++ ) { /* 最後の列を使用しない */
       for( x2=0; x2<m2; x2++ ) { /* 最後の行を使用しない */
           fptype v;
           int    i;
           v=VYR[x2*(n2+1)+y2];
           /* 次の整数への変換中に最も近くの整数に丸めるため 0.5 を追加 */
           v = v + 0.5f;
           i = (int)v;
           /* 飽和 */
           if(i<0) i=0;
           if(i>255)i=255;
           /* 整数に変換して出力画像ピクセルの色平面 c に保存 */
           row_pointers2[y2][x2*nc+c]=(png_byte)i;
       }
    }
}
```

インテル® MKL のデータ・フィッティング関数を使用したスプライン計算

interpolate_der 関数はスプライン補間を使用してヒストスプラインを計算します。

インテル® MKL のデータ・フィッティング・ルーチンを使用して *nx* ブレークポイント `x[]` で関数値 `f[]` の指定された *ny* = 1 行について自由端境界条件で自然 3 次スプラインを計算した後、`nsite` サイト `xx[]` で微分を計算し、関数微分の `nsite` 値の行を `r[]` に出力します。

```
void interpolate_der(int nx, fptype* x,    int ny, fptype* f,    int nsite, fptype* xx,    fptype* r)
{
    /* ... */
    scoeff=(fptype*)mkl_malloc(sizeof(fptype)*SPLORDER*ny*(nx-1),64);
    if(scoeff==NULL) {
        printf("Error: not enough memory for scoeff.\n");
        exit(-1);
    }

    errorCode = NewTask1D(&interpTask, nx,x,xhint, ny,f, yhint);
    if(errorCode)printf("NewTask1D errorCode=%d\n",errorCode);

    errorCode = EditPPSpline1D(interpTask,
        SPLORDER,
        SPLTYPE,
        DF_BC_FREE_END, NULL, /* 自由端境界条件。*/
        DF_NO_IC, NULL,       /* 内部条件なし。*/
        scoeff, DF_NO_HINT);
    if(errorCode)printf("EditPPSpline1D errorCode=%d\n",errorCode);

    errorCode = Construct1D(interpTask, 0, 0);
    if(errorCode)printf("Construct1D errorCode=%d\n",errorCode);

    errorCode = Interpolate1D(interpTask, DF_INTERP, ny, nsite,xx, siteHint, der_orders_sz,der_orders,
 datahint,r,rhint, nxx_cell_indexes);
    if(errorCode)printf("Interpolate1D errorCode=%d\n",errorCode);

    errorCode = dfDeleteTask(&interpTask);
    if(errorCode)printf("dfDeleteTask errorCode=%d\n",errorCode);

    mkl_free(scoeff);
}
```

説明

サンプルは入力画像の各行 (および中間画像の各行) についてループの `interpolate_der` 関数を呼び出すため、`interpolate_der` の *ny* パラメーターは 1 で、単一の補間関数 (または画像行) を表します。 しかし、データ・フィッティング・ルーチンは *ny* > 1 をサポートしています。 これは、`interpolate_der` の呼び出しのループを、処理する画像の行の総数を設定する *ny* パラメーターを含む 1 つの呼び出しに置換して、サンプルを書き直すことができることを意味します。

注

画像が十分大きくなると、データ・フィッティング構築および補間ルーチンの内部で自動並列化が行われます。

ny = 1 の場合、`#pragma omp parallel for` を使用して `interpolate_der` の呼び出しのループを並列化してもかまいませんが、データ・フィッティング・ルーチン内部の自動並列化を利用するほうが良いでしょう。 理論上は、各色平面のアルゴリズムはほかの色平面から独立しているため、このプラグマを *c* ループに追加することもできます。 しかし、異なるスレッドがメモリーの同じライン内の 1 バイトのフラグメント (同じピクセルの RGB コンポーネント) にアクセスできるため、パフォーマンスが低下する可能性があります。また、小さな画像では相対的な並列化オーバーヘッドが大きくなります。パフォーマンスの低下を回避する 1 つの方法は、画像データをブロックに分割することです。どんな場合でも、並列化は大量のデータを扱う場合にのみ効果があります。

また、ベクトル化に `#pragma simd` を使用する場合は注意が必要です。

注

外部ライブラリーは画像ファイルのロードと保存用のプリミティブを供給できます。 サンプルコードは、Linux* および OS X* では libpng を、Windows® では Microsoft® Foundation Class (MFC) ライブラリーの CImage クラスを使用します。

サンプル入力画像は `image_scaling/png_input` フォルダーに、出力画像は `image_scaling/png_output` フォルダーにあります。

結果

サンプル入力

ヒストスプライン手法を使用したスケーリング後の出力

注

画像スケーリングの結果を評価するには、*image_scaling/png_output* にある実際の出力イメージを参照してください。

使用するルーチン

タスク	ルーチン
1 次元データ・フィッティング・タスクの新しいタスク・ディスクリプターを作成して初期化する	dfsNewTask1D
スプラインの順序、種類、データ・フィッティング・タスク・ディスクリプターのスプラインを表す境界条件パラメーターを変更する	dfsEditPPSpline1D
自然 3 次スプラインを構築する	dfsConstruct1D
指定された場所でデータ・フィッティング補間およびスプライン微分の計算を実行する	dfsInterpolate1D
データ・フィッティング・タスク・オブジェクトを破棄してメモリーを解放する	dfDeleteTask
64 ビット境界でアライメントされたメモリーバッファーを割り当てる	mkl_malloc
mkl_malloc で割り当てられたメモリーを解放する	mkl_free

Python* 科学計算の高速化

15

目的

インテル® マス・カーネル・ライブラリー (インテル® MKL) を使用して大量の数学計算を実行する Python* アプリケーションを高速化する。

ソリューション

大量の数学計算を実行する Python* アプリケーションで以下のパッケージを使用します。

NumPy*	*N* 次元配列オブジェクト、汎用データの多次元コンテナーからなります。
SciPy*	線形代数、統計、積分、フーリエ変換、常微分方程式ソルバー、その他のモジュールを含みます。 高速 *N* 次元配列操作については NumPy* に依存します。

NumPy*/SciPy* 計算を高速化するには、インテル® MKL を使用してこれらのパッケージのソースをビルドし、パフォーマンスを測定するサンプルを実行します。 インテル® Xeon Phi™ コプロセッサーが利用可能なシステムでさらにパフォーマンスを向上するには、自動オフロードを有効にします。

インテル® MKL を使用した NumPy* および SciPy* のビルド

これらのステップは、Linux* または Windows® オペレーティング・システム、インテル® 64 アーキテクチャー、ILP64 インターフェイスを想定しています。

1. http://www.scipy.org/Download (英語) から最新の NumPy* および SciPy* パッケージを取得して展開します。
2. 最新バージョンのインテル® MKL、インテル® C++ コンパイラー、インテル® Fortran コンパイラーをインストールします。
3. インテル® C++ コンパイラーおよびインテル® Fortran コンパイラーの環境変数を設定します。
   - **Linux*:**

     次のコマンドを実行します。

     ```
     $source <intel tools installation dir>/bin/compilervars.sh intel64
     ```

   - **Windows®:**

     環境設定を起動して Intel64 ビルドバイナリーの Visual Studio® モードを指定します。

     i. (Windows® 8:) マウスポインターを画面の左下隅に移動し、マウスの右ボタンをクリックして、**[検索]** を選択し、画面の白い部分をクリックします。

     ii. **[Intel Parallel Studio 2016 (インテル® Parallel Studio 2016)]** セクションに移動して、**[Intel64 Visual Studio 20*XX* mode (インテル® 64 VS 20XX モード)]** を選択します。

4. ディレクトリーを *<NumPy dir>* に変更します。
5. 既存の `site.cfg.example` をコピーし、`site.cfg` として保存します。

6. site.cfg を開き、[mkl] セクションのコメントを解除して、次のように変更します。

- **Linux*:**

```
[mkl]
library_dirs = /opt/intel/compilers_and_libraries_2016/linux/mkl/lib/intel64
include_dirs = /opt/intel/compilers_and_libraries_2016/linux/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

- **Windows®:**

```
[mkl]
library_dirs = C:\Program Files
(x86)\IntelSWTools\compilers_and_libraries_2016\windows\mkl\lib\intel64;
C:\Program Files (x86)\Intel\Composer XE 2015.x.yyy\compiler\lib\intel64
include_dirs = C:\Program Files (x86)\IntelSWTools\compilers_and_libraries_2016\windows\mkl\include
mkl_libs = mkl_lapack95_lp64,mkl_blas95_lp64,mkl_intel_lp64,mkl_intel_thread,mkl_core,libiomp5md
lapack_libs = mkl_lapack95_lp64,mkl_blas95_lp64,mkl_intel_lp64,mkl_intel_thread,mkl_core,libiomp5md
```

7. インテル® C++ コンパイラーの最適化オプションを渡すように、*<NumPy dir>*/distutils の intelccompiler.py を変更します。

- **Linux*:**

```
self.cc_exe = 'icc -O3 -g -xhost -fPIC
-fomit-frame-pointer -openmp -DMKL_ILP64'
```

- **Windows®:**

```
self.compile_options = [ '/nologo', '/O3', '/MD', '/W3', '/Qstd=c99',
'/QxHost', '/fp:strict', '/Qopenmp']
```

8. インテル® Fortran コンパイラーの最適化オプションを渡すように、*<NumPy dir>*/distutils/fcompiler フォルダーの intel.py を変更します。

- **Linux*:**

```
ifort -xhost -openmp -i8 -fPIC
```

- **Windows®:**

```
def get_flags(self):
    opt = ['/nologo', '/MD', '/nbs','/names:lowercase', '/assume:underscore']
```

9. ディレクトリーを *<NumPy dir>* に変更し、NumPy* をビルドしてインストールします。

- **Linux*:**

```
$python setup.py config --compiler=intelem build_clib
--compiler=intelem
build_ext --compiler=intelem install
```

- **Windows®:**

```
python setup.py config --compiler=intelemw build_clib
--compiler=intelemw build_ext --compiler=intelemw install
```

10. ディレクトリーを *<SciPy dir>* に変更し、SciPy* をビルドしてインストールします。

- **Linux*:**

```
$python setup.py config --compiler=intelem --fcompiler=intelem build_clib
--compiler=intelem --fcompiler=intelem build_ext --compiler=intelem
--fcompiler=intelem install
```

- **Windows®:**

```
python setup.py config --compiler=intelemw --fcompiler=intelvem build_clib
--compiler=intelemw --fcompiler=intelvem build_ext --compiler=intelemw
--fcompiler=intelvem install
```

コードサンプル

```
import NumPy as np
import scipy.linalg.blas as slb
import time

M = 10000
N = 6000
k_list = [64, 128, 256, 512, 1024, 2048, 4096, 8192]

np.show_config()

for K in k_list:
        a = np.array(np.random.random((M, N)), dtype=np.double, order='C', copy=False)
        b = np.array(np.random.random((N, K)), dtype=np.double, order='C', copy=False)
        A = np.matrix(a, dtype=np.double, copy=False)
        B = np.matrix(b, dtype=np.double, copy=False)

        start = time.time()
        C = slb.dgemm(1.0, a=A, b=B)
        end = time.time()

        tm = start - end
        print ('{0:4}, {1:9.7}'.format(K, tm))
```

ソースコード: サンプル (http://software.intel.com/en-us/mkl_cookbook_samples (英語)) の `dgemm_python` フォルダーを参照してください。

自動オフロードの有効化

インテル® Xeon Phi™ コプロセッサーがシステムで利用可能な場合、コプロセッサーへの計算の自動オフロードを有効にするには、`MKL_MIC_ENABLE` 環境変数を 1 に設定します。

説明

ビルドステップはデフォルトの Python* パスに NumPy* および SciPy* をインストールします。NumPy* および SciPy* をホームまたは別のフォルダーにインストールするには、ステップ 9 および 10 のコマンドで `-prefix=$HOME` またはフォルダーのパスを指定します。Python* を `$HOME` にインストールした場合、NumPy* をビルドした後、SciPy* をビルドする前に、`PYTHONPATH` 環境変数を `$HOME/lib/pythonY.Z/site-packages` に設定します。ここで、*Y.Z* は Python* のバージョンです。

Intel64 ビルドバイナリーの Visual Studio® モードを選択するステップ 3 の命令は、Windows® のバージョンに依存します。 次に例を示します。

Windows® 7 では、**[すべてのプログラム]** > **[Intel Parallel Studio XE 20*XX* (インテル® Parallel Studio XE 20XX)]** > **[Compiler and Performance Libraries (コンパイラーおよびライブラリー)]** > **[Intel Compiler <version> Command Prompt (インテル(R) コンパイラー <バージョン> コマンドプロンプト)]** に移動して、**[Intel64 Visual Studio 20*XX* mode (インテル® 64 VS 20XX モード)]** を選択します。ここで、**20*XX*** は Visual Studio® のバージョン (2014 など) です。

コードサンプルは最も一般的な行列乗算ルーチン `dgemm` を SciPy* および NumPy* の配列から使用して、入力行列を作成、初期化します。 インテル® MKL を使用して NumPy* および SciPy* をビルドした場合、このコードはインテル® MKL BLAS `dgemm` ルーチンを呼び出します。

インテル® Xeon Phi™ コプロセッサーがシステムで利用可能な場合、一部のインテル® MKL ルーチンはコプロセッサーを活用することができます (自動オフロードが有効なインテル® MKL 関数のリストは、[AO] を参照してください)。自動オフロードが有効な場合、これらのルーチンはホスト CPU とコプロセッサー間で計算を分割します。

文献目録 (英語)

スパースソルバー

[Amos10] Ron Amos. *Lecture 3: Solving Equations Using Fixed Point Iterations,* University of Wisconsin CS412: Introduction to Numerical Analysis, 2010 (http://pages.cs.wisc.edu/~holzer/cs412/lecture03.pdf).

[Smith86] G.D. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods,* Oxford Applied Mathematics & Computing Science Series, Oxford University Press, USA, 1986.

数値演算

[Black73] Fischer Black and Myron S. Scholes. *The Pricing of Options and Corporate Liabilities.* Journal of Political Economy, v81 issue 3, 637-654, 1973.

[Kargupta02] Hillol Kargupta, Krishnamoorthy Sivakumar, and Samiran Ghost. *A Random Matrix-Based Approach for Dependency Detection from Data Streams,* Proceedings of Principles of Data Mining and Knowledge Discovery, PKDD 2002: 250-262. Springer, New York, 2002.

[KnuthV2] D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, section 3.4.2: Random Sampling and Shuffling. Algorithm P*, 3rd Edition, Addison-Wesley Professional, 2014.

[SRSWOR] *Using Intel® C++ Composer XE for Multiple Simple Random Sampling without Replacement* (https://software.intel.com/en-us/articles/using-intel-c-composer-xe-for-multiple-simple-random-sampling-without-replacement).

[Zhang12] Zhang Zhang, Andrey Nikolaev, and Victoriya Kardakova. *Optimizing Correlation Analysis of Financial Market Data Streams Using Intel® Math Kernel Library,* Intel Parallel Universe Magazine, 12: 42-48, 2012.

[Bosner14] Tina Bosner, Bojan Crnković, and Jerko Škifić. *Tension splines with application on image resampling*, Mathematical Communications 19 (3), 2014, 517-529.

[Zavyalov80] Yu. S. Zavyalov, B.I. Kvasov, and V. L. Miroshnichenko. *Methods of Spline Functions*, Nauka (in Russian), 1980.

異なるプログラミング環境でのインテル® MKL の使用

[AO] *Intel MKL Automatic Offload enabled functions for Intel® Xeon Phi™ coprocessors* (https://software.intel.com/en-us/articles/intel-mkl-automatic-offload-enabled-functions-for-intel-xeon-phi-coprocessors).

索引