Intel® Math Kernel Library

Cookbook

Intel® MKL

Document Number: 330244-007US

Legal Information

Contents

Legal Information

Third Party Content

Getting Help and Support

Intel MKL provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, check: http://www.intel.com/software/products/support.

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more (visit http://www.intel.com/software/products/).

Registering your product entitles you to one year of technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing these services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, contact Intel, or seek product support, please visit http://www.intel.com/software/products/support.

What's New

The following improvements have been made in this version of the cookbook:

- The Speeding up Python* scientific computations recipe explains how to benefit from NumPy* and SciPy* prebuilt with Intel MKL by using the Intel® Distribution for Python*.

Notational Conventions

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

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

This manual uses the following notational conventions:

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

Routine Name Shorthand

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

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

Font Conventions

The following font conventions are used:

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

Related Information

To reference how to use the library in your application, use this guide in conjunction with the following documents:

- The *Intel® Math Kernel Library Reference Manual*, which provides *reference* information on routine functionalities, parameter descriptions, interfaces, calling syntaxes, and return values.
- The *Intel® Math Kernel Library User's Guide*.

Web versions of these documents are available in the Intel® Software Documentation Library at http://software.intel.com/en-us/intel-software-technical-documentation.

Intel® Math Kernel Library Recipes

The Intel® Math Kernel Library (Intel® MKL) contains many routines to help you solve various numerical problems, such as multiplying matrices, solving a system of equations, and performing a Fourier transform. While many problems do not have dedicated Intel MKL routines, you can solve them by assembling the building blocks provided by Intel MKL.

The Intel Math Kernel Library Cookbook includes these recipes to help you to assemble Intel MKL routines for solving some more complex problems:

- Matrix recipes using Intel MKL PARDISO, BLAS, Sparse BLAS, and LAPACK routines
  - Finding an approximate solution to a nonlinear equation demonstrates a method of finding a solution to a nonlinear equation using Intel MKL PARDISO, BLAS, and Sparse BLAS routines.
  - Factoring a block tridiagonal matrix uses Intel MKL implementations of BLAS and LAPACK routines.
  - Solving a system of linear equations with an LU-factored block tridiagonal coefficient matrix extends the factoring recipe to solving a system of equations.
  - Factoring block tridiagonal symmetric positive definite matrices using BLAS and LAPACK routines demonstrates Cholesky factorization of a symmetric positive definite block tridiagonal matrix using BLAS and LAPACK routines.
  - Solving a system of linear equations with block tridiagonal symmetric positive definite coefficient matrix extends the factoring recipe to solving a system of equations using BLAS and LAPACK routines.
  - Computing principal angles between two subspaces uses LAPACK SVD to calculate the principal angles.
  - Computing principal angles between invariant subspaces of block triangular matrices extends the use of LAPACK SVD to the case where the subspaces are invariant subspaces of a block triangular matrix and are complementary to each other.
- Fast Fourier Transform recipes
  - Evaluating a Fourier Integral uses Intel MKL Fast Fourier Transform (FFT) interface to evaluate a continuous Fourier transform integral.
  - Using Fast Fourier Transforms for computer tomography image reconstruction uses Intel MKL FFT interface to reconstruct an image from computer tomography data.
- Numerics recipes
  - Noise filtering in financial market data streams uses Intel MKL summary statistics routines for computing a correlation matrix for streaming data.
  - Using the Monte Carlo method for simulating European options pricing computes call and put European option prices with an Intel MKL basic random number generator (BRNG).
  - Using the Black-Scholes formula for European options pricing speeds up Black-Scholes computation of European options pricing with Intel MKL vector math functions.
  - Multiple simple random sampling without replacement generates *K* simple random length-*M* samples without replacement from a population of size *N* for a large *K*.
  - Using a histospline technique to scale images uses Intel MKL data fitting functions for image scaling and spline interpolation for histospline computation.
- Recipes for using Intel MKL in different programming environments
  - Speeding up Python* scientific computations demonstrates a performance boost of Python code by building NumPy* and SciPy* sources with Intel MKL and enabling Intel MKL Automatic Offload.

NOTE
Code examples in the cookbook are provided in Fortran for some recipes and in C for other recipes.

Finding an approximate solution to a stationary nonlinear heat equation

1

Goal

Obtain a solution to a boundary value problem for the thermal equation, with thermal coefficients that depend on the solution.

Solution

Use a fixed-point iteration approach [Amos10], utilizing Intel MKL PARDISO for solving linear problems on each external iteration.

1. Set up the matrix structure in CSR format.
2. Perform fixed-point iteration until the residual norm becomes lower than the tolerance.
   a. Use the `pardiso` routine to solve the linearized system for the current iteration.
   b. Set the solution of the system to the next approximation of the main equation using the `dcopy` routine.
   c. Based on the new approximation, calculate the new elements of the matrix.
   d. Calculate the residual of the current solution using the `mkl_cspblas_dcsrgemv` routine.
   e. Calculate the norm of the residual using the `dnrm2` routine and compare it with the tolerance.
3. Free the internal memory of the solver.

Source code: see the `sparse` folder in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Finding an approximate solution using Intel MKL PARDISO, Sparse BLAS, and BLAS

```
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 );
```

Routines Used

Task	Routine	Description
Solve the linearized system on the current iteration; free internal memory of the solver.	`PARDISO`	Calculates the solution of a set of sparse linear equations with multiple right-hand sides.

Task	Routine	Description
Set the solution found as the next approximation of the main equation.	DCOPY	Copies vector to another vector.
Calculate the residual of the current nonlinear iteration.	MKL_CSPBLAS_DCSRGEMV	Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with zero-based indexing.
	DAXPY	Computes a vector-scalar product and adds the result to a vector.
Calculate the norm of the residual to compare it with stopping criteria.	DNRM2	Computes the Euclidean norm of a vector.

Discussion

The stationary nonlinear heat equation can be described as a boundary value problem for a nonlinear partial differential equation:

$$-\frac{\partial}{\partial x}\left(\mu(v)\frac{\partial v}{\partial x}\right)-\frac{\partial}{\partial y}\left(\mu(v)\frac{\partial v}{\partial y}\right)-\frac{\partial}{\partial z}\left(\mu(v)\frac{\partial v}{\partial z}\right)=1, \qquad (x,y,z)\in D$$
$$v|_{\partial D}=0$$

Where the domain *D* is assumed to be a cube: $D=(0,1)^3$, and $v(x,y,z)$ is an unknown function of temperature.

For the purpose of demonstration, the problem is restricted to linear dependence of the thermal coefficient on the solution:

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

To obtain a numerical solution, an equidistant grid with grid step *h* in the domain *D* is chosen, and the partial differential equation is approximated using finite differences. This procedure [Smith86] yields a system of nonlinear algebraic equations:

$$\begin{aligned}
&-u_{i,j,k-1}*\frac{m_{i,j,k-1}+m_{i,j,k}}{2}+u_{i,j,k}*\frac{m_{i,j,k-1}+2m_{i,j,k}+m_{i,j,k+1}}{2}-\quad u_{i,j,k+1}*\frac{m_{i,j,k}+m_{i,j,k+1}}{2}\\
&-u_{i,j-1,k}*\frac{m_{i,j-1,k}+m_{i,j,k}}{2}+u_{i,j,k}*\frac{m_{i,j-1,k}+2m_{i,j,k}+m_{i,j+1,k}}{2}-\quad u_{i,j+1,k}*\frac{m_{i,j,k}+m_{i,j+1,k}}{2}\\
&-u_{i-1,j,k}*\frac{m_{i-1,j,k}+m_{i,j,k}}{2}+u_{i,j,k}*\frac{m_{i-1,j,k}+2m_{i,j,k}+m_{i+1,j,k}}{2}-\quad u_{i+1,j,k}*\frac{m_{i,j,k}+m_{i+1,j,k}}{2}\\
&=h^2
\end{aligned}$$

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

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

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

Each equation ties together the value of the unknown grid function *u* and the value of the respective right hand side at seven grid points. The left hand sides of the equations can be represented as linear combinations of the grid function values with coefficients which depend on the solution itself. Introducing a matrix composed of these coefficients, the equations can be rewritten in vector-matrix form:

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

Since the coefficient matrix *A* is sparse (it has only seven nonzero elements in each row), it is suitable to store it in a CSR-format array (see *Sparse Matrix Storage Formats* in the *Intel Math Kernel Library Reference Manual*), and use the PARDISO* solver for solving it using this iterative algorithm:

1. Set u to initial value u^0.
2. Calculate residual $r = A(u)u - g$.
3. Do while $||r|| <$ tolerance:
    a. Solve system $A(u)w = g$ for w.
    b. Set $u = w$.
    c. Calculate residual $r = A(u)u - g$.

Factoring general block tridiagonal matrices

2

Goal

Perform LU factorization of a general block tridiagonal matrix.

Solution

Intel MKL LAPACK provides a wide range of subroutines for LU factorization of general matrices, including dense matrices, band matrices, and tridiagonal matrices. This recipe extends the range of functionality to general block tridiagonal matrices subject to condition all the blocks are square and have the same order.

To perform LU factorization of a block tridiagonal matrix with square blocks of size *NB* by *NB*:

1. Sequentially apply partial LU factorization to rectangular blocks of size *M* by *N* formed by the first two block rows and first three block columns of the matrix (where *M* = 2*NB*, *N* = 3*NB*, and *K* = *NB*), and moving down along the diagonal until the last but one block row is processed.

   Partial LU factorization: for LU factorization of a general block tridiagonal matrix it is useful to have separate functionality for partial LU factorization of a rectangular *M*-by-*N* matrix. The partial LU factorization algorithm with parameter *K*, where $K \le \min(M, N)$, consists of

   a. Perform LU factorization of the *M*-by-*K* submatrix.
   b. Solve the system with triangular coefficient matrix.
   c. Update the lower right (*M* - *K*)-by-(*N* - *K*) block.

   The resulting matrix is *A* = *P*(*LU* + *A1*) where *L* is a lower trapezoidal *M*-by-*K* matrix, *U* is an upper trapezoidal matrix, *P* is permutation (pivoting) matrix, and *A1* is a matrix with nonzero elements only in the intersection of the last *M* - *K* rows and *N* - *K* columns.
2. Apply general LU factorization to the last (2*NB*) by (2*NB*) block.

Source code: see the `BlockTDS_GE/source/dgeblttrf.f` file in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Performing partial LU factorization

```
      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)
          …
```

Factoring a block tridiagonal matrix

```
      …
      DO K=1,N-2
C Form a 2*NB by 3*NB submatrix A with block structure
C     (D_K   C_K 0    )
```

```
C     (B_K D_K+1 C_K+1)
      …
C Partial factorization of the submatrix
         CALL PTLDGETRF(2*NB, 3*NB, NB, A, 2*NB, IPIV(1,K), INFO)
C Factorization results to be copied back to arrays storing blocks of the tridiagonal matrix
      …
      END DO
C Out of loop factorization of the last 2*NB by 2*NB submatrix
      CALL DGETRF(2*NB, 2*NB, A, 2*NB, IPIV(1,N-1), INFO)
C Copy the last result back to arrays storing blocks of the tridiagonal matrix
      …
```

Routines Used

Task	Routine	Description
LU factorization of the *M*-by-*K* submatrix	`DGETRF`	Compute the LU factorization of a general m-by-n matrix
Permute rows of the matrix	`DSWAP`	Swap two vectors
Solving a system with triangular coefficient matrix	`DTRSM`	Solve a triangular matrix equation
Update lower-right (*M* - *K*)-by-(*N* - *K*) block	`DGEMM`	Compute a matrix-matrix product with general matrices.

Discussion

For partial LU factorization, let A be a rectangular *m*-by-*n* matrix:

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

NOTE
For ease of reading, lower-case indexes such as *m*, *n*, *k*, and *nb* are used in this discussion. These correspond to the upper-case indexes used in the Fortran solution and code samples.

The matrix can be decomposed using LU factorization of the *m*-by-*k* submatrix, where 0 < *k*≤*n*. For this application, *k*< min(*m*, *n*), because `?getrf` can be used directly to factor the matrix if *m*≤*k*≤*n* or *n* = *k*≤*m*.

A can be represented as a block matrix:

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

where A_{11} is a *k*-by-*k* submatrix, A_{12} is a *k*-by-(*n* - *k*) submatrix, A_{21} is an (*m* - *k*)-by-*k* submatrix, and A_{22} is an (*m* - *k*)-by-(*n* - *k*) submatrix.

The *m*-by-*k* panel A_1 can be defined as

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

A_1 can be LU factored (using `?getrf`) as $A_1 = PLU$, where *P* is a permutation (pivoting) matrix, *L* is lower trapezoidal with unit elements on the diagonal, and *U* is upper triangular:

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

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

NOTE
Since the diagonal elements of *L* do not need to be stored, the array used to store A_1 can be used to store the elements of *L* and *U*.

Applying P^T to the second panel of *A* gives:

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

This yields the equation:

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

Introducing the term A''_{12} defined as

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

and substituting it into the equation for $P^T A$ yields:

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

Multiplying the previous equation by P gives:

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

This can be considered a partial LU factorization of the initial matrix.

- The product $L^{-1}{}_{11}A'_{12}$ can be computed by calling `?trsm` and can be stored in place of the array used for A_{12}. The update A'_{22} - $L_{21}(L^{-1}{}_{11}A'_{12})$ can be computed by calling `?gemm` and can be stored in place of the array used for A_{22}.
- If the submatrices do not have full rank, this method cannot be applied because LU factorization would fail.
- Unlike LU factorization of general matrices, for general block tridiagonal matrices the factorization $A = LU$ described below cannot be written in the form $A = PLU$ (where P is a permutation matrix). Because of pivoting, the structure of the left factor, L, includes permutations. Pivoting also complicates the right factor, U, which has three diagonals instead of two.

For LU factorization of a block tridiagonal matrix, let A be a block tridiagonal matrix where all blocks are square and of the same order n_b:

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

The matrix is to be factored as $A = LU$.

First, consider 2-by-3 block submatrix

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

which can be decomposed as

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

This decomposition can be obtained by applying the partial LU factorization described previously. Here P^{T}_{1} is a product of n_b elementary permutations which can be represented as a $2n_b$-by-$2n_b$ matrix:

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

Introducing an *N*-by-*N* block matrix where all blocks are size n_b-by-n_b:

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

This allows the previous decomposition to be rewritten as:

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

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

Next, factor the 2-by-3 block matrix of the second and third rows of the matrix on the right-hand side of that equation:

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

where P^{T}_{2} is defined as:

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

The previous decomposition can be continued as:

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

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

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

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

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

Introducing this notation for the pivoting matrix simplifies the equations:

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

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

where $P^{\mathrm{T}}{}_j$ is $2n_b$ by $2n_b$, and is located at the intersection of the *j*-th and (*j*+1)-st rows and columns. This allows the decomposition above to be written more compactly as

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

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

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

At step *N* - 2 the local factorization is:

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

After this step, multiplying by the pivoting matrix:

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

gives:

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

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

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

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

At the last (N - 1)-st step the matrix is square and factorization is complete:

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

The last step differs from previous ones in the structure of the pivoting as well: all previous P^{T}_j for j = 1, 2, ..., N - 2 were products of n_b permutations (they depend on n_b integer parameters), whereas P^{T}_{N-1} is applied to a square matrix of order $2n_b$ (it depends on $2n_b$ parameters). So in order to store all of the pivoting indices an integer array of length $(N - 2)\, n_b + 2n_b = Nn_b$ is necessary.

Multiplying the previous decomposition from the left by Π^{T}_{N-1} gives the final decomposition

$$\varPi_{N-1}^{\mathrm{T}}\varPi_{N-2}^{\mathrm{T}}\ldots\varPi_2^{\mathrm{T}}\varPi_1^{\mathrm{T}}\begin{pmatrix} D_1 & C_1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ B_1 & D_2 & C_2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & B_2 & D_3 & C_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & B_{N-2} & D_{N-1} & C_{N-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & B_{N-1} & D_N \end{pmatrix}$$

$$= \varPi_{N-1}^{\mathrm{T}}\varPi_{N-2}^{\mathrm{T}}\ldots\varPi_3^{\mathrm{T}}\varPi_2^{\mathrm{T}}\begin{pmatrix} L_{1,1} \\ L_{2,1} \\ 0 \\ \vdots \\ 0 \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} & U_{13} & 0 & \cdots & 0 \end{pmatrix}$$

$$+\varPi_{N-1}^{\mathrm{T}}\varPi_{N-2}^{\mathrm{T}}\ldots\varPi_3^{\mathrm{T}}\begin{pmatrix} 0 \\ L_{2,2} \\ L_{3,2} \\ 0 \\ \vdots \\ 0 \end{pmatrix}\begin{pmatrix} 0 & U_{22} & U_{23} & U_{24} & 0 & \cdots & 0 \end{pmatrix}$$

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

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

$$+\begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ L_{N-1,N-1} \\ L_{N,N-1} \end{pmatrix}\begin{pmatrix} 0 & \cdots & 0 & 0 & U_{N-1,N-1} & U_{N-1,N} \end{pmatrix}+\begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ L_{N,N} \end{pmatrix}\begin{pmatrix} 0 & \cdots & 0 & 0 & 0 & U_{N,N} \end{pmatrix}$$

Multiplying this decomposition by $\Pi_1\Pi_2\ldots\Pi_{N-1}$allows it to be written in LU factorization form:

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

$$= \left(\mathit{\Pi}_1 \begin{pmatrix} L_{11} \\ L_{21} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \mathit{\Pi}_1 \mathit{\Pi}_2 \begin{pmatrix} 0 \\ L_{22} \\ L_{32} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \cdot \ldots \cdot \mathit{\Pi}_1 \mathit{\Pi}_2 \cdots \mathit{\Pi}_{N-2} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ L_{N-2,N-2} \\ L_{N-1,N-2} \\ 0 \end{pmatrix} \mathit{\Pi}_1 \mathit{\Pi}_2 \cdots \mathit{\Pi}_{N-2} \mathit{\Pi}_{N-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ L_{N-1,N-1} \\ L_{N,N-1} \end{pmatrix} \mathit{\Pi}_1 \mathit{\Pi}_2 \cdots \mathit{\Pi}_{N-2} \mathit{\Pi}_{N-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ L_{N,N} \end{pmatrix} \right)$$

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

While applying this formula it should be taken into account that Π_j for $j = 1, 2, \ldots, N-2$ are products of n_b elementary transpositions applied to block rows with indices j and $j+1$, but Π_{N-1} is the product of $2n_b$ transpositions applied to the last two block rows N-1 and N.

Solving a system of linear equations with an LU-factored block tridiagonal coefficient matrix

3

Goal

Use Intel MKL LAPACK routines to craft a solution to a system of equations involving a block tridiagonal matrix, since LAPACK does not have routines that directly solve systems with block tridiagonal matrices.

Solution

Intel MKL LAPACK provides a wide range of subroutines for solving systems of linear equations with an LU-factored coefficient matrix. It covers dense matrices, band matrices and tridiagonal matrices. This recipe extends this set to block tridiagonal matrices subject to condition all the blocks are square and have the same order. A block triangular matrix *A* has the form

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

Solving a system *AX*=*F* with an LU-factored matrix *A*=*LU* and multiple right hand sides (RHS) consists of two stages (see Factoring Block Tridiagonal Matrices for LU factorization).

1. Forward substitution, which consists of solving a system of equations *LY*=*F* with pivoting, where *L* is a lower triangular coefficient matrix. For factored block tridiagonal matrices, all blocks of *Y* except the last one can be found in a loop which consists of

   a. Applying pivoting permutations locally to the right hand side.
   b. Solving the local system of *NB* linear equations with a lower triangular coefficient matrix, where *NB* is the order of the blocks.
   c. Updating the right hand side for the next step.

   The last two block components are found outside of the loop because of the structure of the final pivoting (two block permutations need to be applied consecutively) and the structure of the coefficient matrix.
2. Backward substitution, which consists of solving the system *UX*=*Y*. This step is simpler because it does not involve pivoting. The procedure is similar to the first step:

   a. Solving systems with triangular coefficient matrices.
   b. Updating right hand side blocks.

   The difference from the previous step is that the coefficient matrix is upper, not lower, triangular, and the direction of the loop is reversed.

Source code: see the `BlockTDS_GE/source/dgeblttrs.f` file in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Forward Substitution

```
! Forward substitution
! In the loop compute components Y_K stored in array F
      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

! Apply two last pivots
      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
! Computing Y_N-1 and Y_N out of loop and store in array F
      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)
```

Backward Substitution

```
      …
! Backward substitution
! Computing X_N out of loop and store in array F
      CALL DTRSM('L', 'U', 'N', 'N', NB, NRHS, 1D0, D(1,(N-1)*NB+1), NB, F((N-1)*NB+1,1), LDF)
! Computing X_N-1 out of loop and store in array F
      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)
! In the loop computing components X_K stored in array F
      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
      …
```

Routines Used

Task	Routine	Description
Apply pivoting permutations	`dswap`	Swap a vector with another vector
Solve a system of linear equations with lower and upper triangular coefficient matrices	`dtrsm`	Solve a triangular matrix equation
Update the right hand side blocks	`dgemm`	Compute a matrix-matrix product with general matrices.

Discussion

NOTE
A general block tridiagonal matrix with blocks of size *NB* by *NB* can be treated as a band matrix with bandwidth 4**NB*-1 and solved by calling Intel MKL LAPACK subroutines for factoring and solving band matrices (`?gbtrf` and `?gbtrs`). But using the approach described in this recipe requires fewer floating point computations because if the block matrix is stored as a band matrix, many zero elements would be treated as nonzeros in the band and would be processed during computations. The effect increases for bigger *NB*. Analogously, band matrices can also be treated as block tridiagonal matrices. But this storage scheme is also not very efficient because the blocks would contain many zeros treated as nonzeros. So band storage schemes and block tridiagonal storage schemes and their respective solvers should be considered as complementary to each other.

Given a system of linear equations:

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

The block tridiagonal coefficient matrix *A* is assumed to be factored as shown:

$$A = L \cdot U$$

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

$$\cdot \begin{pmatrix} U_{11} & U_{12} & U_{13} & 0 & \cdots & 0 & 0 & 0 \\ 0 & U_{22} & U_{23} & U_{24} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & U_{N-2,N-2} & U_{N-2,N-1} & U_{N-2,N} \\ 0 & 0 & 0 & 0 & \cdots & 0 & U_{N-1,N-1} & U_{N-1,N} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & U_{NN} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{N-1} \\ X_N \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_{N-1} \\ F_N \end{pmatrix}$$

See Factoring Block Tridiagonal Matrices for a definition of the terms used.

The system is decomposed into two systems of linear equations:

$$UX = Y, LY = F$$

The second equation can be expanded:

$$LY = \Pi_1 \begin{pmatrix} L_{11} \\ L_{21} \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} Y_1 + \Pi_1\Pi_2 \begin{pmatrix} 0 \\ L_{22} \\ L_{32} \\ 0 \\ \vdots \\ 0 \end{pmatrix} Y_2 + \cdots + \Pi_1\Pi_2\cdots\Pi_{N-2} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ L_{N-2,N-2} \\ L_{N-1,N-2} \\ 0 \end{pmatrix} Y_{N-2}$$

$$+\Pi_1\Pi_2\cdots\Pi_{N-2}\Pi_{N-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ L_{N-1,N-1} \\ L_{N,N-1} \end{pmatrix} Y_{N-1} + \Pi_1\Pi_2\cdots\Pi_{N-2}\Pi_{N-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ L_{NN} \end{pmatrix} Y_N = \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_{N-1} \\ F_N \end{pmatrix}$$

In order to find Y_1, first the permutation Π_1^T must be applied. This permutation only changes the first two blocks of the right hand side:

$$\begin{pmatrix} F'_1 \\ F'_2 \\ F_3 \\ \vdots \\ F_{N-1} \\ F_N \end{pmatrix} = \Pi_1^T \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_{N-1} \\ F_N \end{pmatrix}$$

Applying the permutation locally gives

$$\begin{pmatrix} F'_1 \\ F'_2 \end{pmatrix} = P_1^T \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$$

Now Y_1 can be found:

$$Y_1 = L_{11}^{-1} F'_1$$

After finding Y_1, similar computations can be repeated to find Y_2, Y_3, ..., and Y_{N-2}.

NOTE
The different structure of Π_{N-1} (see Factoring Block Tridiagonal) means that the same equations cannot be used to compute Y_{N-1} and Y_N and that they must be computed outside of the loop.

The algorithm to use the equations to find *Y* is:

```
do for $k$ = 1 to $N$ - 2
    $\begin{pmatrix} F_k \\ F_{k+1} \end{pmatrix} := P_k^{\mathrm{T}} \begin{pmatrix} F_k \\ F_{k+1} \end{pmatrix} Y_k = L_{kk}^{-1} F_k$                //using ?trsm
    $F_k := F_k - L_{k,k-1} Y_{k-1}$     //update right hand side using ?gemm
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
$F_N := F_N - L_{N,N-1} Y_{N-1}$     //update right hand side using ?gemm
$Y_N = L_{NN}^{-1} F_N$                     //?trsm
```

The *UX* = *Y* equations can be represented as

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

The algorithm for solving these equations is:

```
$X_N = U_{NN}^{-1} Y_N$                                  //?trsm
$X_{N-1} = U_{N-1,N-1}^{-1} \left( Y_{N-1} - U_{N-1,N-1} X_N \right)$    //?gemm followed by ?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
```

Factoring block tridiagonal symmetric positive definite matrices

4

Goal

Perform Cholesky factorization of a symmetric positive definite block tridiagonal matrix.

Solution

To perform Cholesky factorization of a symmetric positive definite block tridiagonal matrix, with *N* square blocks of size *NB* by *NB*:

1. Perform Cholesky factorization of the first diagonal block.
2. Repeat *N* - 1 times moving down along the diagonal:
   a. Compute the off-diagonal block of the triangular factor.
   b. Update the diagonal block with newly computed off-diagonal block.
   c. Perform Cholesky factorization of a diagonal block.

Source code: see the `BlockTDS_SPD/source/dpbltrf.f` file in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Cholesky factorization of a symmetric positive definite block tridiagonal matrix

```
      …
      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
```

Routines Used

Task	Routine	Description
Perform Cholesky factorization of diagonal blocks	`DPOTRF`	Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.
Compute off-diagonal blocks of the triangular factor	`DTRSM`	Solves a triangular matrix equation.
Update the diagonal blocks	`DSYRK`	Performs a symmetric rank-k update.

Discussion

A symmetric positive definite block tridiagonal matrix, with *N* diagonal blocks D_i and *N* - 1 sub-diagonal blocks B_i of size *NB* by *NB* is factored as:

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

Multiplying the blocks of the matrices on the right gives:

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

$$= \begin{pmatrix} L_1L_1^{\mathrm{T}} & L_1C_1^{\mathrm{T}} & & & \\ C_1L_1^{\mathrm{T}} & C_1C_1^{\mathrm{T}} + L_2L_2^{\mathrm{T}} & L_2C_2^{\mathrm{T}} & & \\ & \ddots & \ddots & \ddots & \\ & & C_{N-2}L_{N-2}^{\mathrm{T}} & C_{N-2}C_{N-2}^{\mathrm{T}} + L_{N-1}L_{N-1}^{\mathrm{T}} & L_{N-1}C_{N-1}^{\mathrm{T}} \\ & & & C_{N-1}L_{N-1}^{\mathrm{T}} & C_{N-1}C_{N-1}^{\mathrm{T}} + L_NL_N^{\mathrm{T}} \end{pmatrix}$$

Equating the elements of the original block tridiagonal matrix to the elements of the multiplied factors yields:

$$\begin{aligned} L_1L_1^{\mathrm{T}} &= D_1 \\ C_1L_1^{\mathrm{T}} &= B_1 \\ C_1C_1^{\mathrm{T}} + L_2L_2^{\mathrm{T}} &= D_2 \\ C_2L_2^{\mathrm{T}} &= B_2 \\ &\vdots \\ C_{N-2}C_{N-2}^{\mathrm{T}} + L_{N-1}L_{N-1}^{\mathrm{T}} &= D_{N-1} \\ C_{N-1}L_{N-1}^{\mathrm{T}} &= B_{N-1} \\ C_{N-1}C_{N-1}^{\mathrm{T}} + L_NL_N^{\mathrm{T}} &= D_N \end{aligned}$$

Solving for C_i and $L_iL_i^{\mathrm{T}}$:

$$\begin{aligned} L_1L_1^{\mathrm{T}} &= D_1 \\ C_1 &= B_1L_1^{-\mathrm{T}} \\ L_2L_2^{\mathrm{T}} &= D_2 - C_1C_1^{\mathrm{T}} \\ C_2 &= B_2L_2^{-\mathrm{T}} \\ &\vdots \\ L_{N-1}L_{N-1}^{\mathrm{T}} &= D_{N-1} - C_{N-2}C_{N-2}^{\mathrm{T}} \\ C_{N-1} &= B_{N-1}L_{N-1}^{-\mathrm{T}} \\ L_NL_N^{\mathrm{T}} &= D_N - C_{N-1}C_{N-1}^{\mathrm{T}} \end{aligned}$$

Note that the right-hand sides of the equations for $L_iL_i^T$ is a Cholesky factorization. Therefore a routine `chol()` for performing Cholesky factorization can be applied to this problem using code such as:

```
$L_1$=chol($D_1$)
do i=1,N-1
     $C_i=B_i\cdot L_i^{-T}$ //trsm()
     $D_{i+1}:=D_{i+1} - C_i\cdot C_i^T$ //syrk()
     $L_{i+1}$=chol($D_{i+1}$)
end do
```

Solving a system of linear equations with a block tridiagonal symmetric positive definite coefficient matrix

5

Goal

Solve a system of linear equations with a Cholesky-factored symmetric positive definite block tridiagonal coefficient matrix.

Solution

Given a coefficient symmetric positive definite block tridiagonal matrix (with square blocks each of the same *NB*-by-*NB* size) is LLT factored, the solving stage consists of:

1. Solve the system of linear equations with a lower bidiagonal coefficient matrix which is composed of *N* by *N* blocks of size *NB* by *NB* and with diagonal blocks which are lower triangular matrices:
   a. Solve the *N* local systems of equations with lower triangular diagonal blocks of size *NB* by *NB* which are used as coefficient matrices and respective parts of the right hand side vectors.
   b. Update the local right hand sides.
2. Solve the system of linear equations with an upper bidiagonal coefficient matrix which is composted of block size *N* by *N* blocks of size *NB* by *NB* and with diagonal blocks which are upper triangular matrices.
   a. Solve the local systems of equations.
   b. Update the local right hand sides.

Source code: see the `BlockTDS_SPD/source/dpbltrs.f` file in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Cholesky factorization of a symmetric positive definite block tridiagonal matrix

```
      …
      …
      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
      …
```

Routines Used

Task	Routine	Description
Solve a local system of linear equations	`DTRSM`	Solves a triangular matrix equation.
Update the local right hand sides	`DGEMM`	Computes a matrix-matrix product with general matrices.

Discussion

Consider a system of linear equations described by:

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

Assume that matrix *A* is a symmetric positive definite block tridiagonal coefficient matrix with all blocks of size *NB* by *NB*. *A* can be factored as described in Factorizing block tridiagonal symmetric positive definite matrices uisng BLAS and LAPACK routines to give:

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

Then the algorithm to solve the system of equations is:

1. Solve the system of linear equations with a lower bidiagonal coefficient matrix in which the diagonal blocks are lower triangular matrices:

```
$Y_1=L_1^{-1} F_1$ //trsm()
do i=2,N
    $G_i=F_i - C_{i-1} Y_{i-1}$ //gemm()
    $Y_i=L_i^{-1} G_i$ //trsm()
end do
```

2. Solve the system of linear equations with an upper bidiagonal coefficient matrix in which the diagonal blocks are upper triangular matrices:

```
$X_N=L_N^{-T} Y_N$ //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
```

6 Computing principal angles between two subspaces

Goal

Get information about the relative position of two subspaces in an inner product space.

Solution

Assuming the subspaces are represented as spans of some vectors, the relative position of the subspaces can be obtained by calculating the set of *Principal Angles* between the subspaces. To calculate the angles:

1. Build orthonormal bases in each subspace and determine the dimensions of the subspaces.
    - **a.** Call an appropriate subroutine to perform a QR factorization with pivoting of matrices, the columns of which span the subspaces.
    - **b.** Using the threshold, determine the dimensions of the subspaces.
    - **c.** Form the orthonormal bases.
2. Form a matrix of inner products of the basis vectors from the one subspace and the basis vectors of another subspace.
3. Compute the Singular Value Decomposition of the matrix.

Source code: see the `ANGLES/definition/main.f` file in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Building orthonormal bases and determining subspace dimensions

```
…
REAL*8 Y(LDY,K),TAU(N),WORK(3*N)
      …
!
! Apply QR factorization with pivoting
      CALL DGEQPF(N, K, Y, LDY, JPVT, TAU, WORK, INFO)
! Process info returned by DGEQPF
      …
! Compute the rank
      K1=0
      DO WHILE((K1 .LT. K) .AND. (ABS(Y(K1+1,K1+1)) .GT. THRESH))
          K1 = K1 + 1
      END DO
!
! Form K1 orthonormal vectors via call DORGQR
      CALL DORGQR(N, K1, K1, Y, LDY, TAU, WORK, LWORK, INFO)
! Process info returned by DORGQR
      …
```

Forming matrix of inner products and computing SVD

```
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)
! Process info returned by DGESVD
      …
```

Discussion

Routines Used

Task	Routine	Description
QR factorization with pivoting of matrices	`dgeqpf`	Compute the QR factorization of a general m-by-n matrix with pivoting
Form orthonormal bases	`dorgqr`	Generates the real orthogonal matrix Q of the QR factorization formed by `?geqpf` or `?geqp3`
Form a matrix of inner products of the basis vectors from the one subspace and the basis vectors of another subspace.	`dgemm`	Compute a matrix-matrix product with general matrices
Compute the Singular Value Decomposition of the matrix	`dgesvd`	Compute the singular value decomposition of a general rectangular matrix

The first step is to build orthonormal bases in each subspace and determine the dimensions of the subspaces.

Let *U* be an *N*-by-*k* matrix (*N*≥*k*) with columns representing vectors in some inner product linear space. To construct an orthonormal basis in this space you can use QR factorization of the matrix *U*, which with pivoting can be represented as *UP* = *QR*. If the dimension of the space is *l* (*l*≤*k*), in the absence of rounding errors occur this yields an orthogonal (unitary for complex-valued matrices) *N*-by-*N* matrix *Q* and upper triangular *N*-by-*k* matrix *R*:

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

The equation *UP* = *QR* means that all columns of *U* are linear combinations of the first *l* columns of *Q*. Due to pivoting, the diagonal elements $r_{j,j}$ of *R* are ordered in non-increasing order of absolute values. In fact, pivoting provides even stronger inequalities:

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

for $j \le m \le k$.

In actual computations with rounding errors, the elements of the lower right (k - l)-by-(k - l) triangle of R are small but non-zero, so a threshold is used to determine the rank $|r_{l,l}| > threshold > |r_{l+1,l+1}|$.

Now you can determine the set of angles between subspaces.

Let $\mathcal{U}$ and $\mathcal{W}$ be two subspaces in the same N-dimensional Euclidean space with $\dim(\mathcal{U})=k$, $\dim(\mathcal{W})=l$, and $k \le l$. To find out the relative position of these subspaces you can use principal angles $\theta_1 \ge \theta_2 \ge \ldots \ge \theta_k \ge 0$, which are defined as follows.

The first angle is defined as:

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

The vectors u_1 and w_1 are called *principal vectors*. The other principal angles and vectors are defined recursively:

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

The principal vectors from the same subspace are pairwise orthogonal:

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

To compute the principal angles you can use Singular Value Decomposition of matrices. Let U and W be matrices of sizes N-by-k and N-by-l respectively, with columns being orthonormal bases in $\mathcal{U}$ and $\mathcal{W}$ respectively. Compute the SVD of the k-by-l matrix $U^T W$:

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

It can be proven that the diagonal elements of Σ are the cosines of the principal angles:

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

The respective pairs of principal vectors are Up^i and Wq^i where p^i and q^i are the i-th columns of P and Q.

Computing principal angles between invariant subspaces of block triangular matrices

7

Goal

Get information about the relative position of two invariant subspaces of a block triangular matrix.

Solution

Assuming the subspaces are represented as spans of some vectors, the relative position of the subspaces can be obtained via calculating the set of principal angles between the subspaces (see Computing principal angles between two subspaces). Additionally, for invariant subspaces of block triangular matrices the Sylvester matrix equation must be solved. The solver used depends on the matrix characteristics:

- If both diagonal blocks of the triangular matrix are upper triangular, use the LAPACK `?trsyl` routine.
- If both diagonal blocks of the triangular matrix are not large and not upper triangular, use LAPACK linear solvers.
- If both diagonal blocks of the triangular matrix are large, upper triangular, and sparse, use the Intel MKL PARDISO solver.

Source code: see these files in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples:

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

Solving Sylvester matrix equation using LAPACK ?trsyl

```
      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
```

Solving Sylvester matrix equation using LAPACK linear solvers

```
      REAL*8 AA(N,N), FF(K*(N-K)), AAA(K*(N-K),K*(N-K))
…
! Forming dense coefficient matrix for Sylvester equation
      CALL SYLMAT(K, AA, N, N-K, AA(K+1,K+1), N, -1D0, 1D0, AAA, NK,
     &        INFO)
! Processing INFO returned by SYLMAT
…
! Forming the right hand side for the system of linear equations that
! correspond to Sylvester equation.
      DO I = 1, K
```

```
          DO J = 1, N-K
              FF((J-1)*K+I) = AA(I,J+K)
          END DO
      END DO
! Solve the system of linear equations
      CALL DGESV(NK, 1, AAA, NK, IPIV, FF, NK, INFO )
! Processing INFO returned by DGESV
```

Solving Sylvester matrix equation using Intel MKL PARDISO

```
      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))
…
! Forming sparse coefficient matrix for Sylvester equation
      CALL FSYLVOP(K, AA, N, N-K, AA(K+1,K+1), N, -1D0, 1D0, COLS,
     &            ROWINDEX, VAL, INFO)
! Processing INFO returned by FSYLVOP
…
! Form the right hand side of the Sylvester equation
      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)
! Processing IERR returned by PARDISO
…
```

Discussion

Routines Used

Task	Routine	Description
Solve Sylvester matrix equation for matrix with upper triangular diagonal blocks	`dtrsyl`	Solve Sylvester equation for real quasi-triangular or complex triangular matrices.
Solve Sylvester matrix equation for matrix which is small and not upper triangular	`dgesv`	Computes the solution to the system of linear equations with a square matrix *A* and multiple right-hand sides.
Solve Sylvester matrix equation for matrix which is not small and not upper triangular	`pardiso`	Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

In order to determine the principal angles between invariant subspaces of the matrix, first let an *N*-by-*N* matrix be represented in a block triangular form:

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

Here diagonal blocks *A* and *B* are square matrices of order *k* and *N*-*k*, respectively. If I_k denotes the unit matrix of order *k*, the equality

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

means the span of first *k* vectors of the standard basis is invariant with respect to transformations on matrix *A*.

Another invariant subspace can be found as a span of columns of the compound matrix $\begin{pmatrix} X \\ I_{N-k} \end{pmatrix}$

Here *X* is some rectangular *k*-by-(*N* - *k*) matrix which should be found. Compute the product

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

If *X* is a solution of the Sylvester equation *XB* - *AX* = *F* the result in the last equation is

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

This demonstrates invariance of the subspace spanned by columns of $\begin{pmatrix} X \\ I_{N-k} \end{pmatrix}$

QR factorization can be used to orthogonalize the basis in the second invariant subspace:

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

Here *C* is a *k*-by-(*N*-*k*) matrix and *S* is an (*N*-*k*)-by-(*N*-*k*) matrix. *C* and *S* satisfy the equation $C^T C + S^T S = I_{N-k}$, where *R* is an upper triangular square matrix of order *N*-*k*. Compute principal angles between these two invariant subspaces using the SVD of *C*:

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

Diagonal elements of Σ are the cosines of the principal angles.

Matrix of Sylvester equations

Consider the Sylvester equation $\alpha AX + \beta XB = F$.

Here square matrices *A* and *B* have orders *M* and *N*, respectively, and α and β are scalars. *F* is a given *M*-by-*N* matrix:

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

X is the *M*-by-*N* matrix to be found:

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

This matrix equation can be considered as a system $\mathcal{A}x = f\,(*)$ of *MN* linear equations for *MN* unknown components of the vector *x* and right hand side vector *f*:

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

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

Matrix $\mathcal{A}$ of order *MN* can be represented as a sum of two matrices. One corresponds to multiplication of matrix *X* from the left by matrix *A* which can be represented in block form with blocks of size *M* by *M*. The matrix forms a block diagonal matrix with *N* blocks on the diagonal:

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

The other matrix in the sum corresponds to multiplication of matrix *X* from the right by matrix *B*. Using the same block form representation yields

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

Here I_M represents the unit matrix of order *M*. Thus the coefficient matrix is

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

This matrix is sparse, with *M* + *N* - 1 nonzero elements in each *MN*-element row. Therefore the Intel MKL PARDISO sparse solver can be used effectively (see the code `ANGLES/source/fsylvop.f`, which forms the coefficient matrix in CSR format for Intel MKL PARDISO). However, for comparatively small *M* and *N* the Intel MKL LAPACK linear solvers are more efficient (see the code `ANGLES/source/sylmat.f`, which forms the coefficient matrix as a dense matrix for use with `dgesv`).

Evaluating a Fourier integral 8

Goal

Use a fast Fourier transform (FFT) to numerically evaluate the continuous Fourier transform integral

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

Solution

Let's assume that the real-valued function *f*(*x*) is zero outside the interval [*a*, *b*] and is sampled at *N* equidistant points $x_n = a + nT/N$, where *T* = |b - a| and *n* = 0, 1, ... , *N*-1. An FFT will be used to evaluate the integral at points $\xi_k = k2\pi/T$, where *k* = 0, 1, ... , *N*/2.

Using Intel® Math Kernel Library FFT Interface in 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) );
}
```

Discussion

The evaluation follows this derivation, based on step-function approximation of the integral:

$$\begin{aligned} F_k = F(\xi_k) &= \int_a^b f(x)\exp(-i2\pi xk/T)\,dx \\ &\approx (T/N)\sum_{n=0}^{N-1} f(x_n)\exp(-i2\pi(a+nT/N)k/T) \\ &= (T/N)\exp(-i2\pi ak/T)\sum_{n=0}^{N-1} f_n\exp(-i2\pi kn/N). \end{aligned}$$

The sum in the last line is an FFT by definition. When the support of the function *f* extends symmetrically around zero, that is, [*a*, *b*] = [-*T*/2, *T*/2], the factor before the sum turns into $(T/N)(-1)^k$.

When the function *f* is real-valued, $F(\xi_k) = \text{conj}(F(\xi_{N-k}))$. The first *N*/2 + 1 complex values of the real-to-complex FFT occupy approximately the same memory as the real input, and they suffice to compute the whole result by conjugation. If the FFT computation is configured to perform a real-to-complex transform, it also takes approximately half as much time as a complex-to-complex FFT.

Using Fast Fourier Transforms for computer tomography image reconstruction 9

Goal

Reconstruct the original image from the Computer Tomography (CT) data using fast Fourier transform (FFT) functions.

Solution

Notation:

- Specification of index ranges adopts the notation used in MATLAB*.

  For example: $k=-q : q$ means $k=-q, -q+1, -q+2, \ldots, q-1, q$.
- While $f(x)$ means the value of the function f at point x, $f[n]$ means the value of nth element of the discrete data set f.

Assumptions:

- The density $f(x, y)$ of a two-dimensional (2D) image vanishes outside the unit circle:

  $f = 0$ when $x^2 + y^2 > 1$.
- The CT data consists of p projections of the image taken at angles $\theta_j = j\pi/p$, where $j = 0 : p - 1$.
- Each projection contains $2q + 1$ density values $g[j, l] = g(\theta_j, s_l)$ approximating the integral of the image along the line

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

  where $l = -q : q$, $s_l = l/q$, and t is the integration parameter.

The discrete image reconstruction algorithm consists of the following steps:

1. Evaluate p one-dimensional (1D) Fourier transforms (for $j = 0 : p - 1$ and $r = -q : q$):

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

2. Interpolate $g_1[j, r]$ from radial grid $(\pi r/q)(\cos\theta_j , \sin\theta_j)$ onto Cartesian grid $(\xi, \eta) = (-q : q, -q : q)$, obtaining $f_2(\pi\xi/q, \pi\eta/q)$.
3. Evaluate one inverse two-dimensional complex-to-complex FFT to obtain a complex-valued reconstruction f_1 of the image:

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

   where $f(m/q, n/q) \approx f_1[m, n]$ for $m = -q : q$ and $n = -q : q$.

Computations in steps 1 and 3 call Intel MKL FFT interfaces. Computations in step 2 implement a simple version of interpolation tailored to the data layout used by Intel MKL FFT.

Reconstructing the original CT image in C/C++

```
// Declarations
int Nq = 2*(q+1); // space for in-place r2c FFT
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]

// Initialize g with the CT data
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);
}

// Step 1: Configure and compute 1D real-to-complex FFTs
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); // now gmem contains g1

// Step 2: Interpolate g1 to f2 - omitted here
complex *f = mkl_malloc( sizeof(complex) * 2*q * 2*q, 64 );

// Step 3: Configure and compute 2D complex-to-complex FFT
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); // now f is complex-valued reconstruction
```

Source code, image file, and makefiles: see the `fft-ct` folder in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Discussion

The code first configures the Intel MKL FFT descriptor for computing a batch of the one-dimensional Fourier transforms in a single call to the `DftiComputeForward` function and then computes the batch transform. The distance for the multiple transforms is set in terms of elements of the corresponding domain (real on input and complex on output). The transforms are in-place by default.

To have a smaller memory footprint, the FFT is computed *in place*, that is, the result of the computation overwrites the input data. With an in-place real-to-complex FFT the input array reserves extra space because the result of the FFT takes slightly more memory than the input.

On input to step 1, array `g` contains p x $(2q+1)$ real-valued data elements $g(\theta_j, s_l)$. The same memory on output of this step contains p x $(q + 1)$ complex-valued output elements $g_1(\theta_j, \pi r/q)$. The complex-conjugate part of the result is not stored, and therefore array `g1` refers to only $q + 1$ values of r.

To interpolate from g_1 to f_2, an additional array `f` is allocated to store complex-valued data $f_2(\xi, \eta)$ and complex-valued output $f_1(x, y)$ of inverse FFT in step 3. The interpolation step does not call Intel MKL functions, but you can find its C++ implementation in the function `step2_interpolation` of the source code for this recipe (file `main.cpp`). The simplest implementation of interpolation is:

- For every (ξ, η) inside the unit circle, find the closest $(\theta_j , \pi r/q)$ and use the value of $g_1(\theta_j , \pi r/q)$ for f_2.
- For every (ξ, η) outside the unit circle, set f_2 to 0.
- In the case of (ξ, η) corresponding to the interval $-\pi < \theta_j < 0$, use conjugate even property of the result of a real-to-complex transform: $g_1(\theta, \omega)=\text{conj}(g(-\theta, -\omega))$.

Notice that the FFT in step 1 is applied to the data offset by half the representation interval, which causes the computed output be multiplied by $e^{i(\pi r/q)q}= (-1)^r$. Instead of correcting this in a separate pass, the interpolation takes the multiplier into account.

Similarly, the 2D FFT in step 3 produces an output that shifts the center of the image to the corner, and step 2 prevents this by phase shifting the input to step 3.

Step 3 computes the two-dimensional (2*q*) x (2*q*) complex-to-complex FFT on the interpolated data contained in array `f`. This computation is followed by a conversion of the complex-valued image f_1 to a visual picture. You can find a complete C++ program that implements the CT image reconstruction in the source code for this recipe (file `main.cpp`).

Noise filtering in financial market data streams 10

Goal

Detect how the price movements of some stocks influences the price movements of others in a large stock portfolio.

Solution

Split a correlation matrix representing the overall dependencies in data into two components, a signal matrix and a noise matrix. The signal matrix gives an accurate estimate of dependencies between stocks. The algorithm ([Zhang12],[Kargupta02]) relies on an eigenstate-based approach that separates noise from useful information by considering the eigenvalues of the correlation matrix for the accumulated data.

Intel MKL Summary Statistics provides functions to calculate correlation matrix for streaming data. Intel MKL LAPACK contains a set of computational routines to compute eigenvalues and eigenvectors for symmetric matrices of various properties and storage formats.

The online noise filtering algorithm is:

1. Compute λ_{min} and λ_{max}, the boundaries of the interval of the noise eigenstates.
2. Get a new block of data from the data stream.
3. Update the correlation matrix using the latest data block.
4. Compute the eigenvalues and eigenvectors that define the noise component, by searching the eigenvalues of the correlation matrix belonging to the interval [λ_{min}, λ_{max}].
5. Compute the correlation matrix of the noise component by combining the eigenvalues and eigenvectors computed in Step 4.
6. Compute the correlation matrix of the signal component by subtracting the noise component from the overall correlation matrix. If there is more data, go back to Step 2.

Source code: see the `nf` folder in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Initialization

Initialize a correlation analysis task and its parameters.

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

...

scanf("%d", &m);            // number of observations in block
scanf("%d", &n);            // number of stocks (task dimension)

...

/* Allocate memory */
nfAllocate(m, n, &x, &mean, &cor, ...);


/* Initialize Summary Statistics task structure */
nfInitSSTask(&m, &n, &task, x, &x_storage, mean, cor, &cor_storage, W);
```

```
...

/* Allocate memory */
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);
...
}
/* Initialize Summary Statistics task structure */
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;

    /* Create VS Summary Statistics task */
    *x_storage = VSL_SS_MATRIX_STORAGE_COLS;
    status = vsldSSNewTask(task, n, m, x_storage, x, 0, 0);
    CheckSSError(status);

    /* Register array of weights in the task */
    W[0] = 0.0;
    W[1] = 0.0;
    status = vsldSSEditTask(*task, VSL_SS_ED_ACCUM_WEIGHT, W);
    CheckSSError(status);

    /* Initialization of the task parameters using full storage
       for correlation matrix computation */
    *cor_storage = VSL_SS_MATRIX_STORAGE_FULL;
    status = vsldSSEditCovCor(*task, mean, 0, 0, cor, cor_storage);
    CheckSSError(status);
}
```

Computation

Perform noise filtering steps for each block of data.

```
/* Set threshold that define noise component */
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);

...
/* Loop over data blocks */
for (i = 0; i < n_block; i++)
{
    /* Read next portion of data */
    nfReadDataBlock(m, n, x, fh);

    /* Update "signal" and "noise" covariance estimates */
    nfKernel(m, n, lambda_min, lambda_max, x, cor, cor_copy,
             task, eval, evect, work, iwork, isuppz,
             cov_signal, cov_noise);
}
...

void nfKernel(…)
{
...

    /* Update correlation matrix estimate using FAST method */
    errcode = vsldSSCompute(task, VSL_SS_COR, VSL_SS_METHOD_FAST);
    CheckSSError(errcode);

...
    /* Compute eigenvalues and eigenvectors of the correlation matrix */
    dsyevr(&jobz, &range, &uplo, &n, cor_copy, &n, &lmin, &lmax,
           &imin, &imax, &abstol, &n_noise, eval, evect, &n, isuppz,
           work, &lwork, iwork, &liwork, &info);

    /* Calculate "signal" and "noise" part of covariance matrix */
    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 parameters */
    char uplo, trans;
    double alpha, beta;

    /* Calculate "noise" part of covariance matrix. */
    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;
    }

    /* Calculate "signal" part of covariance matrix. */
    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;
}
```

Deinitialization

Delete the task and release associated resources.

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

Routines Used

Task	Routine	Description
Initialize a summary statistics task and define the objects for analysis: dataset, its sizes (number of variables and number of observations), and the storage format.	`vsldSSNewTask`	Creates and initializes a new summary statistics task descriptor.
Specify the memory to hold the correlation matrix.	`vsldSSEditCovCor`	Modifies the pointers to covariance/ correlation/cross-product parameters.
Specify the two-element array intended to hold accumulated weights of observations processed so far (necessary for correct computation of estimates for data streams)	`vsldSSEditTask`	Modifies address of an input/output parameter in the task descriptor.
Call the major compute driver by specifying computation type `VSL_SS_COR`, and computation method, `VSL_SS_METHOD_FAST`.	`vsldSSCompute`	Computes Summary Statistics estimates.
De-allocate resources associated with the task.	`vslSSDeleteTask`	Destroys the task object and releases the memory.
Compute eigenvalues and eigenvectors of the correlation matrix.	`dsyevr`	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using the Relatively Robust Representations.

Task	Routine	Description
Perform a symmetric rank-k update.	`dsyrk`	Performs a symmetric rank-k update.

Discussion

Step 4 of the algorithm involves solving an eigenvalue problem for a symmetric matrix. The online noise filtration algorithm requires computation of eigenvalues that belong to the predefined interval [λ_{min}, λ_{max}], which define noise in the data. The LAPACK driver routine `?syevr` is the default routine for solving this type of problem. The `?syevr` interface allows the caller to specify a pair of values, in this case corresponding to λ_{min} and λ_{max}, as the lower and upper bounds of the interval to be searched for eigenvalues.

The eigenvectors found are returned as columns of an array containing an orthogonal matrix *A*, and eigenvalues are returned in an array containing elements of the diagonal matrix *Diag*. The correlation matrix for the noise component can be obtained by computing $A*Diag*A^{T}$. However, instead of constructing a noise correlation matrix using two general matrix multiplications, this can be more efficiently computed with one diagonal matrix multiplication and one rank-n update operation:

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

For the rank-n update operation, Intel MKL provides the BLAS function `?syrk`.

Using the Monte Carlo method for simulating European options pricing

11

Goal

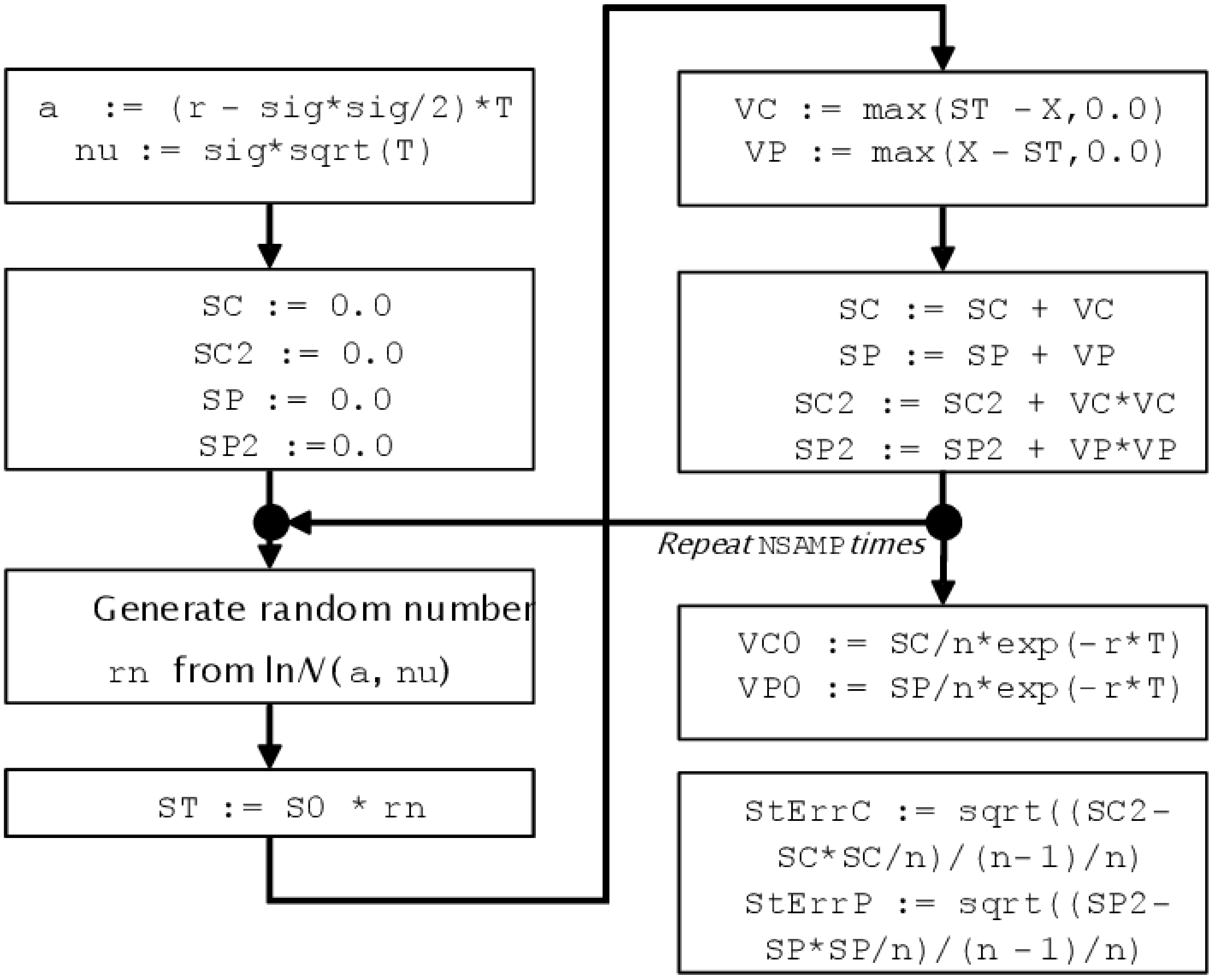
Compute *nopt* call and put European option prices based on *nsamp* independent samples.

Solution

Use Monte Carlo simulation to compute European option pricing. The computation for a pair of call and put options can be described as:

1. Initialize.
2. Compute option prices in parallel.
3. Divide computation of call and put prices pair into blocks.
4. Perform block computation.
5. Deinitialize.

NOTE
On OS X*, this solution requires Intel MKL version 11.2 update 3 or higher.



Source code: see the mc folder in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Initialize in OpenMP section

Create an OpenMP parallel section and initialize the MT2203 random number generator.

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

      j = omp_get_thread_num();

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

This initialization model ensures independent random numbers streams in each thread.

Compute option prices in parallel

Distribute options across available threads.

```
 #pragma omp parallel
   {
       ...

       /* Price options */
     #pragma omp for
       for(i=0;i<nopt;i++)
       {
           MonteCarloEuroOptKernel( ... );
       }

       ...
   }
```

Divide computation of call and put prices pair into blocks

Divide generation of paths into blocks to maintain data locality for optimal performance.

```
   const int nbuf = 1024;
   nblocks = nsamp/nbuf;
   ...
   /* Blocked computations */
   for ( i = 0; i < nblocks; i++ )
   {
       /* Make sure that tail is correctly computed */
       int block_size = (i != nblocks-1)?(nbuf):(nsamp - (nblocks-1)*nbuf);
       ...
   }
```

Perform block computation

In the main computation, generate random numbers and perform reduction.

```
   /* Blocked computations */
   for ( i = 0; i < nblocks; i++ )
   {
       ...

       /* Generating block of random numbers */
       vdRngLognormal( VSL_RNG_METHOD_LOGNORMAL_ICDF, stream,
                       block_size, rn, a, nu, 0.0, 1.0 );

       /* Reduction */
     #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);
```

Deinitialize

Delete the RNG stream.

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

Routines Used

Task	Routine
Creates and initializes a random stream.	`vslNewStream`
Generates lognormally distributed random numbers.	`vdRngLogNormal`
Deletes a random stream.	`vslDeleteStream`

Discussion

Monte Carlo simulation is a widely used technique based on repeated random sampling to determine the properties of some model. The Monte Carlo simulation of European options pricing is a simple financial benchmark which can be used as a starting point for real-life Monte Carlo applications.

Let S_t represent the stock price at a given moment t that follows the stochastic process described by:

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

whereμ is the *drift* and σ is the *volatility*, which are assumed to be constants, $W = (W_t)_{t\geq 0}$ is the *Wiener process*, dt is a time step, and S_0 (the stock price at $t = 0$) does not depend on X.

By definition the expected value is $E(S_t) = S_0 \exp(rt)$, where r is the risk-neutral rate. The previous definition of S_t gives $E(S_t) = S_0 \exp((\mu + \sigma^2/2)t)$, and combining them yields $\mu = r - \sigma^2/2$.

The value of a European option $V(t, S_t)$ defined for $0 \leq t \leq T$ depends on the price S_t of the underlying stock. The option is issued at $t = 0$ and exercised at a time $t = T$ is called the *maturity*. For European *call* and *put* options the value of the option at maturity $V(T, S_T)$ is defined as:

- Call option: $V(T, S_T) = \max(S_T - X, 0)$
- Put option: $V(T, S_T) = \max(X - S_T, 0)$

where X is the *strike price*. The problem is to estimate $V(0, S_0)$.

The Monte Carlo approach to the solution of this problem is a simulation of n possible realizations of S_T followed by averaging $V(T, S_T)$, and then discounting the average by factor $\exp(-rt)$ to get present value of option $V(0, S_0)$. From the first equation S_T follows the log-normal distribution:

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

where ξ is a random variable of the standard normal distribution.

Intel MKL provides a set of basic random number generators (BRNGs) which support different models for parallel computations such as using different parameter sets, block-splitting, and leapfrogging.

This example illustrates MT2203 BRNG which supports 6024 independent parameter sets. In the stream initialization function, a set is selected by adding *j* to the BRNG identifier`VSL_BRNG_MT2203`:

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

See Intel® MKL Vector Statistics Notes for a list of the parallelization models supported by Intel MKL VS BRNG implementations.

The choice of size for computational blocks, which is 1024 in this example, depends on the amount of memory accessed within a block and is typically chosen so that all the memory accessed within a block fits in the target processor cache.

Using the Black-Scholes formula for European options pricing 12

Goal

Speed up Black-Scholes computation of European options pricing.

Solution

Use Intel MKL vector math functions to speed up computation.

The Black-Scholes model describes the market behavior as a system of stochastic differential equations [Black73]. Call and put European options issued in this market are then priced according to the Black-Scholes formulae:

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

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

where

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

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

V_{call} /V_{put} are the present values of the call/put options,S_0 is the present price of the stock , X is the strike price, r is the risk-neutral rate, σ is the volatility, T is the maturity and CDF is the cumulative distribution function of the standard normal distribution.

Alternatively, you can use the error function ERF which has a simple relationship with the cumulative normal distribution function:

$$\mathrm{CDF}(x) = 1/2 + 1/2 \cdot \mathrm{ERF}(x/\sqrt{2})$$

Source code: see the `black-scholes` folder in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Straightforward implementation of Black-Scholes

This code implements the closed form solution for pricing call and put options.

```
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);
    }
}
```

The number of options is specified as the *nopt* parameter. The `tfloat` type is either `float` or `double` depending on the precision you want to use. Similarly `LOG` , `EXP` , `SQRT`, and `CDFNORM` map to single or double precision versions of the respective math functions. The constant `HALF` is either 0.5f or 0.5 for single and double precision, respectively.

In addition to *nopt*, the input parameters for the algorithm are *s0* (present stock price), *r* (risk-neutral rate), *sig* (volatility), *t* (maturity), and *x* (strike price). The result is returned in *vcall* and *vput* (the present value of the call and put options, respectively).

It is assumed that *r* and *sig* are constant for all options being priced; the other parameters are arrays of floating-point values. The *vcall* and *vput* parameters are output arrays.

Discussion

Transcendental functions are at the core of the Black-Scholes formula benchmark. However, each option value depends on five parameters and as the math is computed faster, the memory effects become more pronounced. Thus the number of array parameters is an important factor that can change the computations from being compute-limited to memory bandwidth limited. Additionally, memory size constraints should be considered when pricing hundred millions of options.

The Intel C++ Compiler provides vectorization and parallelization controls that might help uncover the SIMD and multi-core potential of Intel Architecture with respect to the Black-Scholes formula. Optimized vectorized math functionality is available with the Short Vector Math Library (SVML) runtime library.

The Intel MKL Vector Mathematical (VM) functions provide highly tuned transcendental math functionality that can be used to further speed up formula computations.

There are several opportunities for optimization of the straightforward code: hoisting common sub-expressions out of loops, replacing the `CDFNORM` function with the `ERF` function (which is usually faster), exploiting the relationship `ERF(-x) = -ERF(x)`, and replacing the division by `SQRT` with multiplication by the reciprocal square root `INVSQRT` (which is usually faster).

Optimized implementation of Black-Scholes

```
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;
    }
}
```

In this code `INVSQRT(x)` is either `1.0/sqrt(x)` or `1.0f/sqrtf(x)` depending on precision; `TWO` and `QUARTER` are the floating-point constants 2 and 0.25 respectively.

Discussion

A few optimizations help generating effective code using the Intel® C/C++ compiler. See the *User and Reference Guide for the Intel® C++ Compiler* for more details about the compiler pragmas and switches suggested in this section.

Apply `#pragma simd` to tell the compiler to vectorize the loop and `#pragma vector aligned` to notify the compiler that the arrays are aligned (you need to properly align vectors at the memory allocation stage) and that it is safe to rely on aligned load and store instructions. Efficient vectorization, such as that available with SVML, can achieve a speedup of several times versus scalar code.

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

With these changes the code can take advantage of all available CPU cores. The simplest way is to add the –`autopar` switch to the compilation line so that the compiler attempts to parallelize the code automatically. Another option is to use the standard OpenMP* pragma:

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

Further performance improvements are possible if you can relax the accuracy of math functions using Intel C ++ Compiler options such as `-fp-model fast`, `-no-prec-div`, `-ftz`, `-fimf-precision`, `-fimf-max-error`, and `-fimf-domain-exclusion`.

NOTE
Linux* OS specific syntax is given for the compiler switches. See the *User and Reference Guide for the Intel® C++ Compiler* for more detail.

In massively parallel cases, the compute time of math functions can be low enough for memory bandwidth to emerge as the limiting factor for loop performance. This can impair the otherwise linear speedup from parallelism. In such cases memory bandwidth friendly non-temporal load/store instructions can help:

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

The Intel MKL VM component provides highly tuned transcendental math functions that can help further improving performance. However, using them requires refactoring of the code to accommodate for the vector nature of the VM APIs. In the following code example, non-trivial math functions are taken from VM, while remaining basic arithmetic is left to the compiler.

A temporary buffer is allocated on the stack of the function to hold intermediate results of vector math computations. It is important for the buffer to be aligned on the maximum applicable SIMD register size. The buffer size is chosen to be large enough for the VM functions to achieve their best performance (compensating for vector function startup cost), yet small enough to maximize cache residence of the data. You can experiment with buffer size; a suggested starting point is `NBUF=1024`.

Intel MKL VM implementation of Black-Scholes

```
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];
        // This computes vector length for the last iteration of the loop
        // in case nopt is not exact multiple of NBUF
        #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;


        // Must set VM accuracy in each thread
        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];
        }
    }
}
```

For comparable precisions, Intel MKL VM can deliver 30-50% better performance versus Intel Compiler and SVML-based solutions if the problem is compute bound (the data fits in the L2 cache). In this case the latency of cache read/write operations is masked by computations. Once the memory bandwidth emerges as a factor with the growth of the problem size, it becomes more important to optimize the memory usage, and the Intel VM solution based on intermediate buffers can lose its advantage to the no-buffering one-pass solution with SVML.

Routines Used

Task	Routine
Sets a new accuracy mode for VM functions according to the mode parameter and stores the previous VM mode to oldmode.	`vmlSetMode`
Performs element by element division of vector a by vector b	`vsdiv`/`vddiv`
Computes natural logarithm of vector elements.	`vsln`/`vdln`
Computes an inverse square root of vector elements.	`vsinvsqrt`/`vdinvsqrt`
Computes an exponential of vector elements.	`vsexp`/`vdexp`
Computes the error function value of vector elements.	`vserf`/`vderf`

Multiple simple random sampling without replacement

13

Goal

Generate *K*>>1 simple random length-*M* samples without replacement from a population of size *N* (1 ≤*M*≤*N*).

Solution

For exact definitions and more details of the problem, see [SRSWOR].

Use the following implementation of a partial Fisher-Yates Shuffle algorithm [KnuthV2] and Intel MKL random number generators (RNG) to generate each sample:

Partial Fisher-Yates Shuffle algorithm

```
A2.1: (Initialization step) let PERMUT_BUF contain natural numbers 1, 2, ..., N

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

     A2.3: generate random integer X uniform on {i,...,N}

     A2.4: interchange PERMUT_BUF[i] and PERMUT_BUF[X]

 A2.5: (Copy step) for i from 1 to M do: RESULTS_ARRAY[i]=PERMUT_BUF[i]

 End.
```

The program that implements the algorithm conducts 11 969 664 experiments. Each experiment, which generates a sequence of *M* unique random natural numbers from 1 to *N*, is actually a partial length-*M* random shuffle of the whole population of *N* elements. Because the main loop of the algorithm works as a real lottery, each experiment is called "lottery *M* of *N*" in the program.

The program uses *M*=6 and *N*=49, stores result samples (sequences of length *M*) in a single array `RESULTS_ARRAY`, and uses all available parallel threads.

Source code: see the `lottery6of49` folder in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Parallelization

```
#pragma omp parallel
{
    thr = omp_get_thread_num(); /* the thread index */
    VSLStreamStatePtr stream;
    /* RNG stream initialization in this thread */
    vslNewStream( &stream, VSL_BRNG_MT2203+thr, seed );
    ... /* Generation of experiment samples (in thread number thr) */
    vslDeleteStream( &stream );
}
```

The code exploits all CPUs with all available processor cores by using the OpenMP* `#pragma parallel` directive. The array of experiment results `RESULTS_ARRAY` is broken down into `THREADS_NUM` portions, where `THREADS_NUM` is the number of available CPU threads, and each thread (parallel region) processes its own portion of the array.

Intel MKL basic random number generators with the `VSL_BRNG_MT2203` parameter easily support a parallel independent stream in each thread.

Generation of experiment samples

```
/* A2.1: Initialization step */
/*  Let PERMUT_BUF contain natural numbers 1, 2, ..., N */
 for( i=0; i<N; i++ ) PERMUT_BUF[i]=i+1; /* using the set {1,...,N} */
 for( sample_num=0; sample_num<EXPERIM_NUM/THREADS_NUM; sample_num++ ){
     /* Generate next lottery sample (steps A2.2, A2.3, and A2.4): */
     Fisher_Yates_shuffle(...);
     /*  A2.5: Copy step */
     for(i=0; i<M; i++)
         RESULTS_ARRAY[thr*ONE_THR_PORTION_SIZE + sample_num*M + i] = PERMUT_BUF[i];
 }
```

This code implements the partial Fisher-Yates Shuffle algorithm in each thread.

In the case of simulating many experiments, the `Initialization step` is only needed once because at the beginning of each experiment, the order of natural numbers 1...*N* in the `PERMUT_BUF` array does not matter (like in a real lottery).

Fisher_Yates_shuffle function

```
/* A2.2: for i from 0 to M-1 do */
Fisher_Yates_shuffle (...)
{
    for(i=0; i<M; i++) {
        /* A2.3: generate random natural number X from {i,...,N-1} */
        j = Next_Uniform_Int(...);
        /* A2.4: interchange PERMUT_BUF[i] and PERMUT_BUF[X] */
        tmp = PERMUT_BUF[i];
        PERMUT_BUF[i] = PERMUT_BUF[j];
        PERMUT_BUF[j] = tmp;
    }
}
```

Each iteration of the loop `A2.2` works as a real lottery step: it extracts a random item `X` from the bin with remaining items `PERMUT_BUF[i], ..., PERMUT_BUF[N]` and puts the item `X` at the end of the results row `PERMUT_BUF[1],...,PERMUT_BUF[i]`. The algorithm is partial because it does not generate the full permutation of length *N*, but only a part of length *M*.

NOTE
Unlike the pseudocode that describes the algorithm, the program uses zero-based arrays.

Discussion

In step `A2.3`, the program calls the `Next_Uniform_Int` function to generate the next random integer `X`, uniform on {*i*, ..., *N*-1} (see the source code for details). To exploit the full power of vectorized RNGs from Intel MKL, but to minimize vectorization overheads, the generator must generate a sufficiently large vector `D_UNIFORM01_BUF` of size `RNGBUFSIZE` that fits the L1 cache. Each thread uses its own buffer `D_UNIFORM01_BUF` and the index `D_UNIFORM01_IDX` pointing to after the last used random number from that buffer. In the first call to `Next_Uniform_Int` function (or in the case all random numbers from the buffer have been used), the full buffer of random numbers is regenerated again by calling the `vdRngUniform` function with the length `RNGBUFSIZE` and the index `D_UNIFORM01_IDX` set to zero (earlier in the program):

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

Because Intel MKL only provides generators of random values with the same distribution, but step A2.3 requires random integers on different intervals, the buffer is filled with double-precision random numbers uniformly distributed on [0;1) and then, in the Integer scaling step, these double-precision values are converted to fit the needed integer intervals:

```
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}

(then repeat previous M steps)

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

...

number 2*M-1 distributed on: see (M-1)
(then again repeat previous M steps)
...

and so on
```

Integer scaling

```
/* Integer scaling step */
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));
```

Here RNGBUFSIZE is a multiple of *M*.

See [SRSWOR] for performance notes related to this code.

Routines Used

Task	Routine
Creates and initializes an RNG stream.	vslNewStream
Generates double-precision numbers uniformly distributed over the interval [0;1).	vdRngUniform
Deletes an RNG stream.	vslDeleteStream
Allocates memory buffers aligned on 64-byte boundaries for the results and population.	mkl_malloc
Frees memory allocated by mkl_malloc.	mkl_free

Using a histospline technique to scale images 14

Goal

Rescale color or grayscale images using a histospline technique.

Solution

Use Intel MKL data fitting functions for image scaling and spline interpolation for the computation of the histospline.

As described in [Bosner14], a one-dimensional histospline can be applied to the problem of two-dimensional image scaling. The application of one-dimensional interpolation primitives to two-dimensional (surface) interpolation and approximation is described in [Zavyalov80].

Consider the problem of scaling an input image of size *n1*m1* pixels, where *n1* is the number of rows and *m1* is the number of columns, to an image of *n2*m2* pixels as a two-dimensional problem that can be reduced to a number of one-dimensional histopolation problems as follows.

For each color plane *c* of the input image:

1. For each row of the input image:

   a. For each pixel in the current row, compute accumulated cell averages (or color values) and store them in the current row in array *VX*.

   b. Consider the array as interpolated function values, in (*m1*+1) breakpoints `x_breaks[]` uniformly distributed over [0; *m1*]. Construct a histospline using a natural cubic spline, compute its derivatives, and store them in the current row of array *VXR*, *i*=0..*m2* in (*m2*+1) sites `x_sites[]` uniformly distributed over [0; *m1*]:

2. Transpose array *VXR* to array *VXRT*.
3. Similar to *VX*, do the same operations row-by-row for *VY* to get results in array *VYR*:

   a. Similar to step 1.a., compute array *VY* as accumulated sums of the values from *VXRT* just computed.

   NOTE
   This step can be performed simultaneously with the transposition of *VXR*, so it is not necessary to store *VXRT*.

   b. Store derivatives in *VYR* array.

4. Transpose *VYR* to *VR* to get integer results and store it to output image color plane *c*.

   NOTE
   It is not necessary to store *VR*, and the extra last row and column are discarded.

Source code: see the `image_scaling` folder in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Image Scaling Using Intel MKL Data Fitting Functions

```
    for( c=0; c<nc; c++ ) {
        /* 1) PIXELS matrix n1*m1 --> to VX matrix n1*(m1+1) */
        for( y1=0; y1<n1; y1++ ) {
            /* 1.a) get VX as accumulated sums of pixel intencities */
            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) get only derivatives */
            interpolate_der(m1+1,x_breaks, 1,&VX[y1*(m1+1)+0], m2+1,x_sites, &VXR[y1*(m2+1)+0]);
        }
        /* 2) transpose VXR to VXRT not needed (can be skipped) */
        /* 3) */
        for( x2=0; x2<=m2; x2++ ) {
            /* 3.a) transpose VXR; and get VY as accumulated sums of just computed values */
            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) get only derivatives */
        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) transpose VYR to VR and get integer result (VR not needed to store) */
        for( y2=0; y2<n2; y2++ ) { /* not using last row */
           for( x2=0; x2<m2; x2++ ) { /* not using last column */
               fptype v;
               int    i;
               v=VYR[x2*(n2+1)+y2];
               /* add 0.5 for rounding to nearest during next conversion to integer */
               v = v + 0.5f;
               i = (int)v;
               /* saturation */
               if(i<0) i=0;
               if(i>255)i=255;
               /* convert to integer and save to color plane c of output image pixel */
               row_pointers2[y2][x2*nc+c]=(png_byte)i;
           }
       }
    }
```

Spline Computation Using Intel MKL Data Fitting Functions

The `interpolate_der` function uses spline interpolation to compute the histospline.

It uses Intel MKL data fitting routines to compute natural cubic spline with free-end boundary conditions for a given *ny* = 1 rows of function values `f[]` in *nx* breakpoints `x[]`, then computes its derivatives in `nsite` sites `xx[]` and outputs this resulting row of `nsite` values of function derivatives to `r[]`.

```
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, /* Free-end boundary condition. */
        DF_NO_IC, NULL,       /* No internal conditions. */
        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);
}
```

Discussion

The example calls the `interpolate_der` function in a loop for each row of input image (and for each row of intermediate image), so the *ny* parameter of `interpolate_der` is 1, representing a single interpolated function (or image row). But data fitting routines support *ny* > 1. This means that the example can be rewritten to replace the loop of calls to `interpolate_der` by one call with the *ny* parameter set to the total number of image rows to process.

NOTE
When the image is large enough, automatic parallelization is done inside the data fitting construction and interpolation routines.

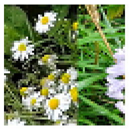
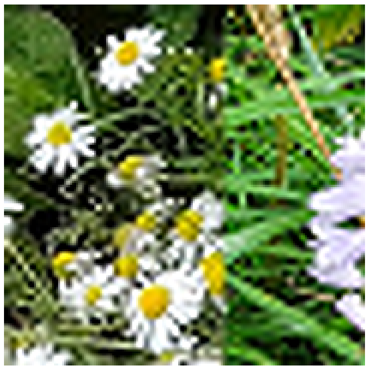
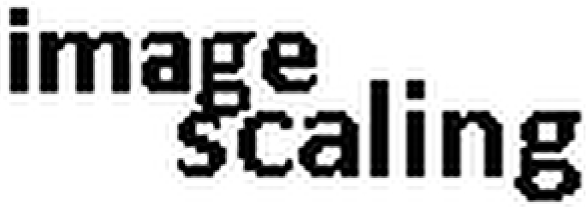
When *ny* = 1, it can make sense to parallelize the loop of calls to `interpolate_der` using `#pragma omp parallel for`, but it is better to rely on automatic parallelization inside the data fitting routines. Theoretically, this pragma could also be added over the *c* loop, since the algorithm for each color plane is independent from other color planes. However, this can cause even degradations because different threads can access one-byte sized fragments (the RGB components of same pixel) within same line of memory. Additionally, relative parallelization overhead can be too large for images which are too small. One way to avoid degradations is to break the image data into blocks. In any case, parallelization only makes sense for large enough amounts of data.

Additionally, take care using `#pragma simd` for vectorization.

NOTE
External libraries can supply primitives for loading and saving image files. The sample code uses libpng for Linux* and OS X*, and the Microsoft Foundation Class Library CImage class for Windows*.

Sample input images are located in `image_scaling/png_input` folder, and resulting output images are located in `image_scaling/png_output`.

Results

Sample input		image scaling
Scaled using histospline technique **NOTE** To evaluate the result of scaling the images, refer to the actual output images in `image_scaling/png_output`.		

Routines Used

Task	Routine
Creates and initializes a new task descriptor for a one-dimensional Data Fitting task.	`dfsNewTask1D`
Modifies spline order, type, and boundary conditions parameters representing the spline in the Data Fitting task descriptor.	`dfsEditPPSpline1D`
Constructs natural cubic spline.	`dfsConstruct1D`
Performs data fitting interpolation and computation of spline derivatives at given sites.	`dfsInterpolate1D`
Destroys a Data Fitting task object and frees the memory.	`dfDeleteTask`
Allocates memory buffers aligned on 64-byte boundaries.	`mkl_malloc`
Frees memory allocated by `mkl_malloc`.	`mkl_free`

Speeding up Python* scientific computations 15

Goal

Use Intel® Math Kernel Library (Intel® MKL) to boost Python* applications that perform heavy mathematical computations.

Solution

Python applications with a high amount of mathematical computations use these packages:

NumPy*	Consists of an *N*-dimensional array object, a multi-dimensional container of generic data.
SciPy*	Includes modules for linear algebra, statistics, integration, Fourier transforms, ordinary differential equations solvers, and more. Depends on NumPy for fast *N*-dimensional array manipulation.

To speed up NumPy/SciPy computations, build the sources of these packages with Intel MKL and run an example to measure the performance. To get further performance boost on systems with Intel® Xeon Phi™ coprocessors available, enable Automatic Offload.

Building NumPy and SciPy with Intel MKL

Important
To benefit from NumPy and SciPy prebuilt with Intel MKL, download Intel® Distribution for Python* from https://software.intel.com/en-us/intel-distribution-for-python.

These steps assume a Linux* or Windows* operating system, Intel® 64 architecture, and ILP64 interface.

1. Get the *latest* NumPy and SciPy packages from http://www.scipy.org/Download and unpack them
2. Install the latest versions Intel MKL and Intel® C++ and Intel® Fortran Compilers
3. Set the environment variables for Intel C++ and Fortran compilers:

   - **Linux*:**

     Execute the command:

     ```
     $source <intel tools installation dir>/bin/compilervars.sh intel64
     ```
   - **Windows*:**

     Launch environment setters to specify the Visual Studio* mode for your Intel64 build binaries:

     i. (Windows 8:) Place the mouse pointer in the bottom-left corner of the screen, click the right mouse button, select **Search**, and click anywhere in the screen white space.
     ii. Navigate to the **Intel Parallel Studio 2016** section and select **Intel64 Visual Studio 20*XX* mode**.
4. Change directory to `<numpy dir>`
5. Make a copy of the existing `site.cfg.example` and save it as `site.cfg`
6. Open `site.cfg`, uncomment the `[mkl]` section, and modify it to look as follows:

- **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. Modify `intelccompiler.py` in *<numpy dir>*`/distutils` to pass optimization options to Intel C++ Compiler:

- **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. Modify `intel.py` in the *<numpy dir>*`/distutils/fcompiler` folder to pass optimization options to Intel Fortran Compiler:

- **Linux:**

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

- **Windows:**

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

9. Change directory to *<numpy dir>* and build and install NumPy:

- **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. Change directory to *<scipy dir>* and build and install SciPy:

- **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
```

Code Example

```
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))
```

Source code: see the `dgemm_python` folder in the samples archive available at http://software.intel.com/en-us/mkl_cookbook_samples.

Enabling Automatic Offload

If Intel® Xeon Phi™ coprocessors are available on your system, to enable Automatic Offload of computations to coprocessors, set the environment variable `MKL_MIC_ENABLE` to 1.

Discussion

The build steps install NumPy and SciPy in the default Python path. To install them in your home directory or another specific folder, pass `–prefix=$HOME` or the folder path to the commands in steps 9 or 10. IF you install Python into `$HOME`, after building NumPy and before building SciPy, set the `PYTHONPATH` environment variable to `$HOME/lib/python`*`Y.Z`*`/site-packages`, where *`Y.Z`* is the Python version.

Specific instructions in step 3 for selecting the Visual Studio* mode for your Intel64 build binaries depend on the Windows version. For example:

On Windows 7, go to **All Programs** -> **Intel Parallel Studio XE 20*XX*** -> **Command Prompt** and select **Intel64 Visual Studio 20*XX* mode**, where **20*XX*** is the version of Visual Studio, such as 2014.

The code example uses the most common matrix-matrix multiplication routine `dgemm` from SciPy and NumPy arrays to create and initialize the input matrices. If NumPy and SciPy are built with Intel MKL, this code actually calls Intel MKL BLAS `dgemm` routine.

If Intel Xeon Phi coprocessors are available on your system, some Intel MKL routines can take advantage of the coprocessors (for the list of Automatic Offload enabled Intel MKL functions, see [AO]). If Automatic Offload is enabled, these routines split the computations between the host CPU(s) and coprocessor(s).

Bibliography

Sparse Solvers

[Amos10] Ron Amos. *Lecture 3: Solving Equations Using Fixed Point Iterations,* University of Wisconsin CS412: Introduction to Numerical Analysis, 2010 (available at 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.

Numerics

[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*, available at 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.

Using Intel® Math Kernel Library (Intel® MKL) in different programming environments

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

Index