PERFORMANCE OPTIMIZATION OF SYMMETRIC FACTORIZATION ALGORITHMS

SILVIO TARCA

Contents

Date: August 31, 2010.

Appendix A. Source Code Listings

A.1. **timing.c – timing functions.**

```c
/*
 * Functions that interface with the C time.h library, and perform date/ time
 * manipulations.
 */

#include <time.h>

#include "timing.h"

#if defined(CLOCK_HIGHRES)
    #define CLOCK CLOCK_HIGHRES
#elif defined(CLOCK_REALTIME)
    #define CLOCK CLOCK_REALTIME
#else
    #error No suitable clock found.  Check docs for clock_gettime.
#endif

/*
 * Converts the clock resolultion stored in the timespec structure into a long
 * double (seconds).
 */
long double timespec_to_ldbl( struct timespec ts )
{
    return ts.tv_sec + 1.0E-9 * ts.tv_nsec;
}

/*
 * Calculates the difference between start and end times, measured in seconds.
 */
long double timespec_diff( struct timespec sta, struct timespec end )
{
    long double delta;

    delta = (end.tv_nsec - (double)sta.tv_nsec) * 1.0E-9;
    delta += end.tv_sec - (double)sta.tv_sec;

    return delta;
}

/*
 * Gets resolution for the defined CLOCK.  The clock resolution, which is stored
 * in the timespec structure, is converted to a long double (seconds) before it
 * is returned by the function.
 */
long double timer_resolution( void )
{
    struct timespec ts;
    clock_getres( CLOCK, &ts );

    return timespec_to_ldbl( ts );
}

/*
 * Gets time for the defined CLOCK.
 */
void get_time( struct timespec *ts )
{
    clock_gettime( CLOCK, ts );
}
```

A.2. matcom.c – common matrix operations.

```
/*
 * Common functions used in matrix computations.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

#include "matcom.h"

/*
 * Generates a random m-by-n matrix with leading dimension m.  The uniform
 * randomly generated elements are scaled by factor alpha.
 */
void create_random_matrix( double alpha, int m, int n, double *E )
{
    const int ldim = m;

    for ( int j = 0; j < n; j++ ) {
        for ( int i = 0; i < m; i++ ) {
            E[j*ldim + i] = alpha * drand48() - (0.50 * alpha);
        }
    }
}

/*
 * Generates a random n-by-n unit lower triangular matrix with leading dimension
 * n.  Matrix elements are stored in column-major order. The uniform randomly
 * generated elements are scaled by factor alpha.
 */
void create_random_unit_lower( double alpha, int n, double *E )
{
    const int ldim = n;

    for ( int j = 0; j < n; j++ ) {
        double *E_j = E + j*ldim;
        memset( E_j, 0, (j-1)*sizeof(double) );
        *(E_j + j) = 1.0;
        for ( int i = j+1; i < n; i++ ) {
            *(E_j + i) = alpha * drand48() - (0.50 * alpha);
        }
    }
}

/*
 * Generates a random n-by-n lower triangular matrix with leading dimension n.
 * Matrix elements are stored in column-major order.  The uniform randomly
 * generated elements are scaled by factor alpha.
 */
void create_random_lower( double alpha, int n, double *E )
{
    const int ldim = n;

    for ( int j = 0; j < n; j++ ) {
        double *E_j = E + j*ldim;
        memset( E_j, 0, (j-1)*sizeof(double) );
        for ( int i = j; i < n; i++ ) {
            *(E_j + i) = alpha * drand48() - (0.50 * alpha);
        }
    }
}


/*
 * Generates a random n-by-n upper triangular matrix with leading dimension n.
 * Matrix elements are stored in column-major order.  The uniform randomly
 * generated elements are scaled by factor alpha.
 */
void create_random_upper( double alpha, int n, double *E )
{
    const int ldim = n;

    for ( int j = 0; j < n; j++ ) {
        double *E_j = E + j*ldim;
        for ( int i = 0; i <= j; i++ ) {
            *(E_j + i) = alpha * drand48() - (0.50 * alpha);
        }
        memset( E_j+j+1, 0, (n-j-1)*sizeof(double) );
    }
}

/*
 * Generates a random n-by-n nonsingular (invertible) matrix with leading
 * dimension n.  Matrix elements are stored in column-major order.  A
 * nonsingular matrix has a unique LU factorization, where L is a unit lower
 * triangular matrix and U is an upper triangular matrix.  The uniform randomly
 * generated elements are scaled by factor alpha.
 */
void create_random_nonsingular( double alpha, int n, double *E )
{
    const int ldim = n;
    double *L, *U;

    L = (double *) malloc( n*n*sizeof(double) );
    U = (double *) malloc( n*n*sizeof(double) );

    create_random_unit_lower( alpha, n, L );
    create_random_upper( alpha, n, U );
    // Compute E = L*U
    clear_matrix( n, n, E );
    multiply_matrix( n, n, n, ldim, L, ldim, U, ldim, E );

    free( L );
    free( U );
}

/*
 * Generates a random n-by-n symmetric positive definite (SPD) matrix with
 * leading dimension n.  Matrix elements are stored in column-major order.
 * x'Ex = x'(M'M)x = (Mx)'(Mx) = ||Mx||^2 >= 0
 * The uniform randomly generated elements are scaled by factor alpha.
 */
void create_random_spd( double alpha, int n, double *E )
{
    const int ldim = n;
    double *M, *T;

    M = (double *) malloc( n*n*sizeof(double) );
    T = (double *) malloc( n*n*sizeof(double) );

    create_random_matrix( alpha, n, n, M );
    transpose_matrix( n, n, M, T );
    // Compute E = M'*M = T*M
    clear_matrix( n, n, E );
    multiply_matrix( n, n, n, ldim, T, ldim, M, ldim, E );

    free( M );
    free( T );
}

/*
 * Generates a random n-by-n symmetric matrix with leading dimension n.  Matrix
 * elements are stored in column-major order. The uniform randomly generated
 * elements are scaled by factor alpha.
 */
void create_random_symmetric( double alpha, int n, double *E )
{
    const int ldim = n;

    // Generate random lower triangular matrix
    for ( int j = 0; j < n; j++ ) {
        for ( int i = j; i < n; i++ ) {
            E[j*ldim + i] = alpha * drand48();
        }
    }
    // Transpose element below the diagonal to create symmetric matrix
    for ( int j = 0; j < n; j++ ) {
        for ( int i = j+1; i < n; i++ ) {
            *(E + j + i*ldim) = *(E + i + j*ldim);
        }
    }
}

/*
 * Sets elements of m-by-n matrix with leading dimension m to zero.
 */
void clear_matrix( int m, int n, double *E )
{
    const int ldim = m;

    for ( int j =  0; j < n; j++ ) {
        memset( E + j*ldim, 0, m*sizeof(double) );
    }

}

/*
 * Copies the elements of an m-by-n matrix E to matrix F.  For both matrices
 * the leading dimension is m, and elements are stored in column-major order.
 */
void copy_matrix( int m, int n, const double *E, double *F )
{
    const int ldim = m;

    for ( int j = 0; j < n; j++ ) {
        const double *E_j = E + j*ldim;
        double *F_j = F + j*ldim;
        memcpy( F_j, E_j, m*sizeof(double) );
    }
}


/*
 * Transposes the elements of an m-by-n matrix E and stores them in matrix F.
 * The leading dimension of matrix E is m, while that of matrix F is n.  Both
 * matrices are stored in column-major order.
 */
void transpose_matrix( int m, int n, const double *E, double *F )
{
    const int ldimE = m;
    const int ldimF = n;

    for ( int j = 0; j < n; j++ ) {
        for ( int i = 0; i < m; i++ ) {
            *(F + j + i*ldimF) = *(E + i + j*ldimE);
        }
    }
}
```

```
/*
 * Copies elements of an m-by-n matrix E to mm-by-nn matrix F with leading
 * dimensions ldimE and ldimF, respectively. Elements of matrix E are stored in
 * column-major order. Array F stores bdim-by-bdim matrix blocks contiguously,
 * and within each block stores elements in column-major order. Also, elements
 * of fringe blocks that do not belong to m-by-n matrix E are set to zero in
 * array F. (Note that contiguous blocks of matrix F are stored in column-major
 * order.)
 */
void form_contig_blocks( int m, int n, int ldimE, const double *E,
    int mm, int nn, int bdim, int ldimF, double *F )
{

    for ( int j = 0; j < nn; j += bdim ) {
        int s = (j + bdim > n) ? (n - j) : bdim;
        int q = (j + bdim > nn) ? (nn - j) : bdim;
        for ( int i = 0; i < mm; i += bdim ) {
            int r = (i + bdim > m) ? (m - i) : bdim;
            int p = (i + bdim > mm) ? (mm - i) : bdim;
            // Clear fringe blocks by setting elements to zero
            if ( s != q || r != p ) {
                double *Fij = F + j*ldimF + i*q;
                memset( Fij, 0, p*q*sizeof(double) );
            }
            for ( int k = 0; k < s; k++ ) {
                const double *Eij = E + j*ldimE + i + k*ldimE;
                double *Fij = F + j*ldimF + i*q + k*p;
                memcpy( Fij, Eij, r*sizeof(double) );
            }
        }
    }
}

/*
 * Copies elements of an m-by-n matrix E to mm-by-nn matrix F with leading
 * dimensions ldimE and ldimF, respectively. Elements of matrix E are stored in
 * column-major order. First, matrix E is copied to a temporary array, where
 * bdim-by-bdim blocks are stored contiguously. Then, these contiguous blocks
 * are copied to array E, such that kdim-by-kdim sub-blocks of each block are
 * stored contiguously. That is, matrix F employs recursive contiguous block
 * storage. (Note that contiguous blocks of matrix F are stored in column-
 * major order, and contiguous sub-blocks within each block are stored in
 * column-major order.)
 */
void form_recur_blocks( int m, int n, int ldimE, const double *E,
    int mm, int nn, int kdim, int bdim, int ldimF, double *F )
{
    double *W;
    W = (double *) malloc( mm*nn*sizeof(double) );
    // Form contiguous matrix blocks
    form_contig_blocks( m, n, ldimE, E, mm, nn, bdim, ldimF, W );

    // Within each matrix block, form contiguous matrix sub-blocks
    for ( int j = 0; j < nn; j += bdim ) {
        int s = (j + bdim > n) ? (n - j) : bdim;
        int q = (j + bdim > nn) ? (nn - j) : bdim;
        for ( int i = 0; i < mm; i += bdim ) {
            int r = (i + bdim > m) ? (m - i) : bdim;
            int p = (i + bdim > mm) ? (mm - i) : bdim;
            double *Wij = W + j*ldimF + i*q;
            double *Fij = F + j*ldimF + i*q;
            form_contig_blocks( r, s, p, Wij, p, q, kdim, p, Fij );
        }
    }
    free( W );
}

/*
 * Copies elements of an mm-by-nn matrix E to m-by-n matrix F with leading
 * dimensions ldimE and ldimF, respectively.  Array E stores bdim-by-bdim matrix
 * blocks contiguously, and within each block stores elements in column-major
 * order.  As matrix E is copied to array F, elements are unpacked and stored
 * in conventional column-major order.
 */
void unpack_contig_blocks( int mm, int nn, int bdim, int ldimE,
    const double *E, int m, int n, int ldimF, double *F )
{
    for ( int j = 0; j < nn; j += bdim ) {
        int s = (j + bdim > n) ? (n - j) : bdim;
        int q = (j + bdim > nn) ? (nn - j) : bdim;
        for ( int i = 0; i < mm; i += bdim ) {
            int r = (i + bdim > m) ? (m - i) : bdim;
            int p = (i + bdim > mm) ? (mm - i) : bdim;
            for ( int k = 0; k < s; k++ ) {
                const double *Eij = E + j*ldimE + i*q + k*p;
                double *Fij = F + j*ldimF + i + k*ldimF;
                memcpy( Fij, Eij, r*sizeof(double) );
            }
        }
    }
}

/*
 * Copies elements of an mm-by-nn matrix E to m-by-n matrix F with leading
 * dimensions ldimE and ldimF, respectively.  Array E employs recursive
 * contiguous block storage, i.e., matrix blocks of size bdim-by-bdim are stored
 * contiguously, and within each block, sub-blocks of size kdim-by-kdim are
 * stored contiguously.  First, matrix E is copied to a temporary array, where
 * elements of each bdim-by-bdim block are unpacked and stored in column-major
 * order.  Then the temporary array is copied to matrix F where the elements of
 * matrix E are unpacked and stored in conventional column-major order.
 */
void unpack_recur_blocks( int mm, int nn, int kdim, int bdim, int ldimE,
    const double *E, int m, int n, int ldimF, double *F )
{
    double *W;

    W = (double *) malloc( mm*nn*sizeof(double) );
    // Within each matrix block, unpack contiguous sub-blocks
    for ( int j = 0; j < nn; j += bdim ) {
        int s = (j + bdim > n) ? (n - j) : bdim;
        int q = (j + bdim > nn) ? (nn - j) : bdim;
        for ( int i = 0; i < mm; i += bdim ) {
            int r = (i + bdim > m) ? (m - i) : bdim;
            int p = (i + bdim > mm) ? (mm - i) : bdim;
            const double *Eij = E + j*ldimE + i*q;
            double *Wij = W + j*ldimE + i*q;
            unpack_contig_blocks( p, q, kdim, p, Eij, r, s, p, Wij );
        }
    }

    // Unpack contiguous matrix blocks
    unpack_contig_blocks( mm, nn, bdim, ldimE, W, m, n, ldimF, F );
    free( W );
}

/*
 * Computes the relative and absolute errors in a matrix computation using the
 * Frobenius norm ||F - E||, where F is the result of the floating point matrix
 * computation and E is the exact solution.  Both matrices are stored in
 * column-major order with leading dimension m.
 */
void error_matrix_comp_frob( double *eps, double *err, int m, int n,
    const double *E, const double *F )
{
    int     ldim = m;
    double  ssq_delta = 0.0;
    double  ssq_eij = 0.0;

    for ( int j = 0; j < n; j++ ) {
        const double *E_j = E + j*ldim;
        const double *F_j = F + j*ldim;
        for ( int i = 0; i < m; i++ ) {
            double delta = *(E_j + i) - *(F_j + i);
            ssq_delta +=  delta * delta;
            ssq_eij += *(E_j + i) * *(E_j + i);
        }
    }
    *err = sqrt( ssq_delta );
    *eps = *err / sqrt( ssq_eij );
}

/*
 * Computes the relative and absolute errors in a matrix computation using the
 * l1-norm ||F - E||, where F is the result of the floating point matrix
 * computation and E is the exact solution.  Both matrices are stored in
 * column-major order with leading dimension m.
 */
void error_matrix_comp_l1( double *eps, double *err, int m, int n,
    const double *E, const double *F )
{
    int     ldim = m;
    double  sum_abs_delta = 0.0;
    double  sum_abs_eij = 0.0;

    *err = 0.0;
    *eps = 0.0;

    for ( int j = 0; j < n; j++ ) {
        const double *E_j = E + j*ldim;
        const double *F_j = F + j*ldim;
        for ( int i = 0; i < m; i++ ) {
            double delta = *(E_j + i) - *(F_j + i);
            sum_abs_delta +=  fabs( delta );
            sum_abs_eij += fabs(*(E_j + i));
        }
        if ( sum_abs_delta > *err ) {
            *err = sum_abs_delta;
            *eps = *err / sum_abs_eij;
        }
    }
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using the SAXPY
 * operation -- jki indexing.  The inner-most loop adds a scalar multiple of
 * column vector x to column vector y.  A (m-by-p), B (p-by-n) and C (m-by-n)
 * are rectangular matrices stored in column-major order with leading dimensions
 * ldimA, ldimB and ldimC, respectively.
 */
void multiply_matrix( int m, int n, int p, int ldimA, const double *A,
    int ldimB, const double *B, int ldimC, double *C )
{
    for ( int j = 0; j < n; j++ ) {
        const double *B_j = B + j*ldimB;            // Points to element B(0,j)
        double *C_j = C + j*ldimC;                  // Points to element C(0,j)
        for ( int k = 0; k < p; k++ ) {
            const double *A_k = A + k*ldimA;        // Points to element A(0,k)
            double bkj = *(B_j + k);                    // Element B(k,j)
            for ( int i = 0; i < m; i++ ) {
                *(C_j + i) += *(A_k + i) * bkj;
            }                                           // C(i,j) += A(i,k) * B(k,j)
        }
    }
}
```

A.3. lufact.c – Gaussian elimination (LU factorization).

```
/*
 * Algorithms implementing unblocked and blocked LU factorization (Gaussian
 * elimination) of nonsingular matrices representing linear systems.  Unblocked
 * algorithms include the outer product method and SAXPY operation, while
 * blocked algorithms include simple blocking and recursive contiguous blocking.
 * LU factorization algorithms are implemented with and without partial
 * pivoting.  (If a nonsingular matrix exhibits certain properties, such a
 * diagonal dominance, then Gaussian elimination without pivoting is numerically
 * stable.)  Also, a function wrapper facilitates calling LAPACK Gaussian
 * elimination routine DGETRF.
 */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

#include "lufact.h"
#include "lapack.h"
#include "matcom.h"

static void eval_pivot_gauss( int n, int k, const double *vec,
    int *piv, int *ord );
static void tri_solve_l1xb( int m, int n, int ldim, double *L, double *B );
static void tri_solve_l1xb_pivot( int m, int n, int ldim,
    int *piv, double *L, double *B );
static void tri_solve_l1xb_kernel( const double *L, double *B );
static void tri_solve_l1xb_blk_ker( int m, int n, int ldimL, const double *L,
    int ldimB, double *B );
static void tri_solve_xub_kernel( const double *U, double *B );
static void tri_solve_xub_blk_ker( int m, int n, int ldimU, const double *U,
    int ldimB, double *B );
static void reduce_matrix( int m, int n, int p, int ldim,
    const double *L, const double *U, double *A );
static void reduce_mat_blk( int m, int n, int p, int ldim, int bdim,
    const double *L, const double *U, double *A);
static void reduce_kernel( const double *L, const double *U, double *A );
static void reduce_blk_ker( int m, int n, int p, int ldimL, const double *L,
    int ldimU, const double *U, int ldimA, double *A );
static void lu_kernel( const int n, double *A );
static void lu_blk_ker( int n, int ldim, double *A );
static void lu_factor( int m, int n, int ldim, double *A );
static void lu_pivot( char pivot, int m, int n, int ldim,
    int *piv, int *ord, double *A );

/***************************************************************************/

/*
 * Determines optimal block dimension for the local environment given a routine
 * and matrix leading dimension.  The function returns the block dimension
 * chosen by the LAPACK LU factorization routine, or a block dimension for
 * testing (debugging).  If the leading dimension is less than the optimal block
 * dimension, the block dimension is set to the leading dimension, and the
 * matrix computation becomes an unblocked algorithm.
 */
int get_block_dim_lu( int ldim )
{
    const int   optm_bdim = 1;
    const int   no_dim = -1;
    const char  *parm_str = " ";
    const char  *func_name = "DGETRF";

    int bdim;

#if defined(DEBUG)
    bdim = BDIM;
#else
    bdim = ilaenv_( &optm_bdim, func_name, parm_str,
        &ldim, &ldim, &no_dim, &no_dim );
#endif
    if ( bdim <= 1 || bdim > ldim ) {
        bdim = ldim;
    }
    return bdim;
}

/*
 * Performs partial pivot selection on an n-by-1 vector representing elements
 * of a column of an n-by-n matrix.  The maximum magnitude element is chosen as
 * the pivot.  A single pivot adjusted by row offset k, and its order (=1)
 * are stored in vectors piv[] and ord[], respectively.  piv[k] specifies the
 * row permutation applied to row k when performing Gaussian elimination.
 */
void eval_pivot_gauss( int n, int k, const double *vec, int *piv, int *ord )
{
    int     p = k;
    double  lambda = -1.0;

    for (int i = 0; i < n; i++) {
        double x = fabs( vec[i] );
        if ( x > lambda ) {
            lambda = x;
            p = i + k;
        }
    }
    piv[k] = p;
    ord[k] = 1;
}

/*
```

```
 * Uses forward substitution to solve the triangular system of linear equations
 * L*X = B, where L is an m-by-m unit lower triangular matrix, and X and B are
 * m-by-n matrices.  Matrices L, X and B are stored in column-major order with
 * leading dimension ldim.  The solution X overwrites matrix B.
 */
void tri_solve_l1xb( int m, int n, int ldim, double *L, double *B )
{
    for ( int k = 0; k < n; k++ ) {
        double *B_k = B + k*ldim;
        for ( int j = 0; j < m-1; j++ ) {
            double bjk = *(B_k + j);
            double *L_j = L + j*ldim;
            for ( int i = j+1; i < m; i++ ) {
                *(B_k+i) -= *(L_j+i) * bjk;
            }
        }
    }
}

/*
 * Uses forward substitution to solve the triangular system of linear equations
 * L*X = B, where L is an m-by-m unit lower triangular matrix, and X and B are
 * m-by-n matrices.  The permutation matrix encoded in the pivot vector piv[]
 * is first applied to matrix B before solving for X.  Matrices L, X and B are
 * stored in column-major order with leading dimension ldim.  The solution X
 * overwrites matrix B.
 */
void tri_solve_l1xb_pivot( int m, int n, int ldim,
    int *piv, double *L, double *B )
{
    for (int k = 0; k < n; k++) {
        double *B_k = B + k*ldim;
        // Apply permutation matrix to column k of matrix B
        for (int i = 0; i < m; i++) {
            double bik = *(B_k + i);
            *(B_k + i) = *(B_k + piv[i]);
            *(B_k + piv[i]) = bik;
        }
        for (int j = 0; j < m-1; j++) {
            double bjk = *(B_k + j);
            double *L_j = L + j*ldim;
            for (int i = j+1; i < m; i++) {
                *(B_k+i) -= *(L_j+i) * bjk;
            }
        }
    }
}

/*
 * Uses forward substitution to solve the triangular system of linear equations
 * L*X = B, where L, X and B are contiguous KDIM-by-KDIM matrix sub-blocks and
 * L is unit lower triangular.  Looping is controlled by a symbolic constant
 * (KDIM), which is evaluated during compilation.  The solution X overwrites B.
 */
void tri_solve_l1xb_kernel( const double *L, double *B )
{
    for ( int j = 0; j < KDIM; j++ ) {
        double *B_j = B + j*KDIM;
        for ( int k = 0; k < KDIM-1; k++ ) {
            double bkj = *(B_j + k);
            const double *L_k = L + k*KDIM;
            for ( int i = k+1; i < KDIM; i++ ) {
                *(B_j + i) -= *(L_k + i) * bkj;
            }
        }
    }
}

/*
 * Uses forward substitution to solve the triangular system of linear equations
 * L*X = B, where L is an m-by-m unit lower triangular matrix block, and X and B
 * are m-by-n matrix blocks.  Matrix blocks L and B are stored contiguously
 * with leading dimensions ldimL and ldimB, respectively.  Within blocks of
 * L and B, sub-blocks of size KDIM*KDIM are stored contiguously.  Suppose that
 * L is decomposed into sub-blocks [L_00, 0; L_10, L_11].  Then
 * L_00*X_00 = B_00; L_00*X_01 = B_01;
 * L_10*X_00 + L_11*X_10 = B_10 --> L_11*X_10 = B_10 - L_10*X_00; and
 * L_10*X_01 + L_11*X_11 = B_11 --> L_11*X_11 = B_11 - L_10*X_01.
 */
void tri_solve_l1xb_blk_ker( int m, int n, int ldimL, const double *L,
    int ldimB, double *B )
{
    for ( int j = 0; j < n; j += KDIM ) {
            double *B_j = B + j*ldimB;

        for ( int i = 0; i < m; i += KDIM ) {
            const double *Li_ = L + i*KDIM;
            const double *Lii = Li_ + i*ldimL;
            double *Bij = B_j + i*KDIM;

            for ( int k = 0; k < i; k += KDIM ) {
                const double *Lik = Li_ + k*ldimL;
                double *Bkj = B_j + k*KDIM;
                reduce_kernel( Lik, Bkj, Bij );
            }
            tri_solve_l1xb_kernel( Lii, Bij );
        }
    }
}
```

```
/*
 * Uses forward substitution to solve the triangular system of linear equations
 * X*U = B, where X, U and B are contiguous KDIM-by-KDIM matrix sub-blocks and
 * U is upper triangular.  Looping is controlled by a symbolic constant (KDIM),
 * which is evaluated during compilation.  The solution X overwrites B.
 */
void tri_solve_xub_kernel( const double *U, double *B )
{
    for ( int k = 0; k < KDIM; k++ ) {
        double ukk = *(U + k + k*KDIM);
        double *B_k = B + k*KDIM;
        for ( int i = 0; i < KDIM; i++ ) {
            *(B_k + i) /= ukk;
        }
        for ( int j = k+1; j < KDIM; j++ ) {
            double ukj = *(U + k + j*KDIM);
            double *B_j = B + j*KDIM;
            for (int i = 0; i < KDIM; i++) {
                *(B_j + i) -= *(B_k + i) * ukj;
            }
        }
    }
}

/*
 * Uses forward substitution to solve the triangular system of linear equations
 * X*U = B, where U is an n-by-n upper triangular matrix block, and X and B are
 * m-by-n matrix blocks.  Matrix blocks U and B are stored contiguously
 * with leading dimensions ldimU and ldimB, respectively.  Within blocks of
 * U and B, sub-blocks of size KDIM*KDIM are stored contiguously.  Suppose that
 * U is decomposed into sub-blocks [U_00, U_01; 0, U_11].  Then
 * X_00*U_00 = B_00; X_10*U_00 = B_10;
 * X_00*U_01 + X_01*U_11 = B_01 --> X_01*U_11 = B_01 - X_00*U_01; and
 * X_10*U_01 + X_11*U_11 = B_11 --> X_11*U_11 = B_11 - X_10*U_01.
 */
void tri_solve_xub_blk_ker( int m, int n, int ldimU, const double *U,
    int ldimB, double *B )
{
    for ( int j = 0; j < n; j += KDIM ) {
        const double *U_j = U + j*ldimU;
        const double *Ujj = U_j + j*KDIM;
        double *B_j = B + j*ldimB;

        for ( int k = 0; k < j; k += KDIM ) {
            const double *Ukj = U_j + k*KDIM;
            double *B_k = B + k*ldimB;

            for ( int i = 0; i < m; i += KDIM ) {
                double *Bik = B_k + i*KDIM;
                double *Bij = B_j + i*KDIM;
                reduce_kernel( Bik, Ukj, Bij );
            }
        }

        for ( int i = 0; i < m; i += KDIM ) {
            double *Bij = B_j + i*KDIM;
            tri_solve_xub_kernel( Ujj, Bij );
        }
    }
}

/*
 * Matrix factorization reduces trailing sub-matrix A by computing A = A - L*U,
 * where A is an m-by-n sub-matrix, L is an m-by-p block of a unit lower
 * triangular matrix and U is a p-by-n block of an upper triangular matrix.
 * Matrices A, L and U are stored in column-major order with leading dimension
 * ldim.  The trailing sub-matrix update is an implementation of the SAXPY
 * operation.
 */
void reduce_matrix( int m, int n, int p, int ldim,
    const double *L, const double *U, double *A )
{
    for ( int j = 0; j < n; j++ ) {
        const double *U_j = U + j*ldim;             // Points to element U(0,j)
        double *A_j = A + j*ldim;                   // Points to element A(0,j)
        for ( int k = 0; k < p; k++ ) {
            const double *L_k = L + k*ldim;         // Points to element L(0,k)
            double ukj = *(U_j + k);                // Element U(k,j)
            for ( int i = 0; i < m; i++ ) {
                *(A_j + i) -= *(L_k + i) * ukj;
            }                                       // A(i,j) -= L(i,k) * U(k,j)
        }
    }
}

/*
 * Matrix factorization reduces trailing sub-matrix A by computing A = A - L*U,
 * where A is an m-by-n sub-matrix, L is an m-by-p column block of a unit lower
 * triangular matrix and U is a p-by-n row block of an upper triangular matrix.
 * Matrices A, L and U are stored in column-major order with leading dimension
 * ldim. Blocking is used to optimize memory access for the trailing sub-matrix
 * reduction, and bdim is the blocking parameter.
 */
void reduce_mat_blk( int m, int n, int p, int ldim, int bdim,
    const double *L, const double *U, double *A )
{
    for ( int j = 0; j < n; j += bdim ) {
        // Determine number of columns in (i,j)th block of A
        const int s = (j + bdim > n) ? (n - j) : bdim;
        for ( int k = 0; k < p; k += bdim ) {
            // Determine number of columns of Lik and rows of Ukj
            const int t = (k + bdim > p) ? (p − k) : bdim;
            // Set pointer to block matrix Ukj
            const double *Ukj = U + k + j*ldim;

            for ( int i = 0; i < m; i += bdim ) {
                // Determine number of rows in (i,j)th block of A
                const int r = (i + bdim > m) ? (m − i) : bdim;
                // Set pointers to block matrices Lik and Aij
                const double *Lik = L + i + k*ldim;
                double *Aij = A + i + j*ldim;
                // Reduce trailing block matrix
                reduce_matrix( r, s, t, ldim, Lik, Ukj, Aij );
            }
        }
    }
}

/*
 * Matrix factorization reduces the trailing sub−matrix by computing A = A − L*U,
 * where A, L and U are contiguous KDIM−by−KDIM sub−blocks of the trailing
 * sub−matrix, a unit lower triangular matrix and an upper triangular matrix,
 * respectively. Looping is controlled by a symbolic constant (KDIM), which is
 * evaluated during compilation. The trailing sub−matrix update is an
 * implementation of the SAXPY operation.
 */
void reduce_kernel( const double *L, const double *U, double *A )
{
    for ( int j = 0; j < KDIM; j++ ) {
        const double *U_j = U + j*KDIM;             // Points to element U(0,j)
        double *A_j = A + j*KDIM;                   // Points to element A(0,j)
        for ( int k = 0; k < KDIM; k++ ) {
            const double *L_k = L + k*KDIM;         // Points to element L(0,k)
            double ukj = *(U_j + k);                // Element U(k,j)
            for ( int i = 0; i < KDIM; i++ ) {
                *(A_j + i) −= *(L_k + i) * ukj;     // A(i,j) −= L(i,k) * U(k,j)
            }
        }
    }
}

/*
 * Matrix factorization reduces the trailing sub−matrix by computing A = A − L*U,
 * where A is an m−by−n block of the trailing sub−matrix, L is an m−by−p block
 * of a unit lower triangular matrix and U is a p−by−n block of an upper
 * triangular matrix. Matrix blocks A, L and U are stored contiguously with
 * leading dimension ldimA, ldimL and ldimU, respectively. Within blocks of
 * A, L and U, sub−blocks of size KDIM*KDIM are stored contiguously.
 */
```

```
void reduce_blk_ker( int m, int n, int p, int ldimL, const double *L,
    int ldimU, const double *U, int ldimA, double *A )
{
    for ( int j = 0; j < n; j += KDIM ) {

        for ( int k = 0; k < p; k += KDIM ) {
            // Set pointer to sub-block Ukj
            const double *Ukj = U + k*KDIM + j*ldimU;

            for ( int i = 0; i < m; i += KDIM ) {
                // Set pointers to sub-blocks Lik and Aij
                const double *Lik = L + i*KDIM + k*ldimL;
                double *Aij = A + i*KDIM + j*ldimA;
                // Perform matrix reduction on sub-blocks
                reduce_kernel( Lik, Ukj, Aij );
            }
        }
    }
}

/*
 * Factorizes an n-by-n matrix sub-block A into a unit lower triangular sub-
 * block L and an upper triangular sub-block U, such that A = L*U.  KDIM-by-KDIM
 * matrix sub-block A is stored contiguously.  The LU factorization algorithm is
 * an implementation of the SAXPY operation.  Looping is controlled by a
 * symbolic constant (KDIM), which is evaluated during compilation.  The factors
 * L and U overwrite A.
 */
void lu_kernel( const int n, double *A )
{
    for (int j= 0; j < n; j++) {

        // Perform cumulative trailing sub-matrix updates on elements of
        // column j above the diagonal
        double *A_j = A + j*KDIM;
        for (int k = 0; k < j; k++) {
            double *A_k = A + k*KDIM;
            double akj = *(A_j + k);
            for (int i = k+1; i < j; i++) {
                *(A_j+i) -= *(A_k+i) * akj;
            }
        }
        // Perform cumulative trailing sub-matrix updates on diagonal element
        // and elements below the diagonal of column j
        for (int k = 0; k < j; k++) {
            double *A_k = A + k*KDIM;
            double akj = *(A_j + k);
            for (int i = j; i < KDIM; i++) {
                *(A_j+i) -= *(A_k+i) * akj;
            }
        }
        // Divide elements in column j below the diagonal by the diagonal element
        double ajj = *(A_j + j);
        for (int i = j+1; i < KDIM; i++) {
            *(A_j+i) /= ajj;
        }
    }
}

/*
 * Factorizes an n-by-n matrix block A into a unit lower triangular block L and
 * an upper triangular block U, such that A = L*U.  Matrix block A is stored
 * contiguously with leading dimension ldim, and within the matrix block, sub-
 * blocks of size KDIM*KDIM are stored contiguously.
 */
void lu_blk_ker( int n, int ldim, double *A )
{
    for ( int j = 0; j < n; j += KDIM ) {
        const int s = (j + KDIM > n) ? (n - j) : KDIM;
        double *A_j = A + j*ldim;
        const double *U_j = A_j;

        // Solve for L*X = A using forward substitution, and perform cumulative
        // trailing sub-matrix updates on matrix sub-blocks above the diagonal
        for ( int k = 0; k < j; k += KDIM ) {
            const double *L_k = A + k*ldim;
            const double *Lkk = L_k + k*KDIM;
            const double *Ukj = U_j + k*KDIM;
            double *Akj = A_j + k*KDIM;
            tri_solve_l1xb_kernel( Lkk, Akj );
            for (int i = k+KDIM; i < j; i += KDIM) {
                const double *Lik = L_k + i*KDIM;
                double *Aij = A_j + i*KDIM;
                reduce_kernel( Lik, Ukj, Aij );
            }
        }

        // Perform cumulative trailing sub-matrix updates on diagonal sub-block
        // and sub-blocks below the diagonal
        for (int k = 0; k < j; k += KDIM) {
            const double *L_k = A + k*ldim;
            const double *Ukj = U_j + k*KDIM;
            for ( int i = j; i < n; i += KDIM ) {
                const double *Lik = L_k + i*KDIM;
                double *Aij = A_j + i*KDIM;
                reduce_kernel( Lik, Ukj, Aij );
            }
        }

        // Factorize diagonal sub-block, and solve X*U = A using forward
        // substitution on sub-blocks below the diagonal
        double *Ajj = A_j + j*KDIM;
        const double *Ujj = Ajj;
        lu_kernel( s, Ajj );
        for ( int i = j+KDIM; i < n; i += KDIM ) {
            double *Aij = A_j + i*KDIM;
            tri_solve_xub_kernel( Ujj, Aij );
        }
    }
}

/*
 * Implements a rectangular version of SAXPY operation (jki indexing) for
 * Gaussian elimination.  Nonsingular m-by-n matrix A with leading dimension
 * ldim is factored into a unit lower triangular matrix L and upper triangular
 * matrix U, such that A = L*U.  It is assumed that properties of matrix A,
 * e.g., diagonally dominant, obviate the need for pivoting.  Elements of L are
 * stored in A(k+1:n-1,k), while elements of U are stored in A(0:k,k), assuming
 * base 0 indexing.  The inner-most loop subtracts a scalar multiple of a vector
 * from another vector.
 */
void lu_factor( const int m, const int n, int ldim, double *A )
{
    for ( int j= 0; j < n; j++ ) {

        // Perform cumulative trailing sub-matrix updates on elements of
        // column j above the diagonal
        double *A_j = A + j*ldim;
        for ( int k = 0; k < j; k++ ) {
            double *A_k = A + k*ldim;
            double akj = *(A_j + k);
            for (int i = k+1; i < j; i++) {
                *(A_j+i) -= *(A_k+i) * akj;
            }
        }
        // Perform cumulative trailing sub-matrix updates on diagonal element
        // and elements below the diagonal of column j
        for (int k = 0; k < j; k++) {
            double *A_k = A + k*ldim;
            double akj = *(A_j + k);
            for (int i = j; i < m; i++) {
                *(A_j+i) -= *(A_k+i) * akj;
            }
        }
        // Divide elements in column j below the diagonal by the diagonal element
        double ajj = *(A_j + j);
        for (int i = j+1; i < m; i++) {
            *(A_j+i) /= ajj;
        }
    }
```

```
}

/*
 * Implements a rectangular version of the SAXPY operation (jki indexing) for
 * Guassian elimination with partial pivoting.  Nonsingular m-by-n matrix A
 * with leading dimension ldim is factored into a unit lower triangular matrix L
 * and upper triangular matrix U.  Row permuted matrix A^ = P*A = L*U.
 * Permutation matrix P is encoded in vectors piv[] and ord[], such that
 * row k is interchanged with row piv[k], and ord[k] = 1 is the diagonal block
 * order.  Elements of L are stored in A(k+1:n-1,k), while elements of U are
 * stored in A(0:k,k), assuming base 0 indexing.  The inner-most loop subtracts
 * a scalar multiple of a vector from another vector.
 */
void lu_pivot( char pivot, int m, int n, int ldim,
    int *piv, int *ord, double *A )
{
    for ( int j= 0; j < n; j++ ) {

        // Apply permutation matrix encoded in pivot vector to column j
        double *A_j = A + j*ldim;
        double *Ajj = A_j +j;
        for ( int k = 0; k < j; k++ ) {
            double akj = *(A_j + k);
            *(A_j + k) = *(A_j + piv[k]);
            *(A_j + piv[k]) = akj;
        }
        // Perform cumulative trailing sub-matrix updates on elements of
        // column j above the diagonal
        for ( int k = 0; k < j; k++ ) {
            double *A_k = A + k*ldim;
            double akj = *(A_j + k);
            for ( int i = k+1; i < j; i++ ) {
                *(A_j+i) -= *(A_k+i) * akj;
            }
        }
        // Perform cumulative trailing sub-matrix updates on diagonal element
        // and elements below the diagonal of column j
        for ( int k = 0; k < j; k++ ) {
            double *A_k = A + k*ldim;
            double akj = *(A_j + k);
            for ( int i = j; i < m; i++ ) {
                *(A_j+i) -= *(A_k+i) * akj;
            }
        }
        // Determine pivot for column j and interchange elements in the pivot
        // row from columns 0 to j with elements in row j
        switch ( pivot ) {
        case 'G':
            eval_pivot_gauss( m-j, j, Ajj, piv, ord );
            break;
        default:
            eval_pivot_gauss( m-j, j, Ajj, piv, ord );
            break;
        }
        if ( j != piv[j] ) {
            for ( int k = 0; k <= j; k++ ) {
                double ajk = *(A + j + k*ldim);
                *(A + j + k*ldim) = *(A + piv[j] + k*ldim);
                *(A + piv[j] + k*ldim) = ajk;
            }
        }
        // Divide elements in column j below the diagonal by the diagonal element
        double ajj = *Ajj;
        for ( int i = j+1; i < m; i++ ) {
            *(A_j+i) /= ajj;
        }
    }
}

/****************************************************************************/

/*
 * If a nonsingular matrix exhibits certain properties, such a diagonal
 * dominance, then Gaussian elimination without pivoting is numerically stable.
 */

/*
 * Implements the outer product method (kji indexing) to factorize nonsingular
 * n-by-n matrix A into a unit lower triangular matrix L and upper triangular
 * matrix U, such that A = L*U.  Elements of L are stored in A(k+1:n-1,k), while
 * elements of U are stored in A(0:k,k), assuming base 0 indexing.  Each pass
 * through the k-loop performs an outer product operation.
 */
void lu_outer_product( int n, double *A )
{
    const int ldim = n;

    for ( int k = 0; k < n-1; k++ ) {

        // Divide elements of column k below the diagonal by the diagonal element
        double *A_k = A + k*ldim;
        double akk = *(A_k + k);
        for ( int i = k+1; i < n; i++ ) {
            *(A_k + i)  /= akk;
        }
        // Update trailing sub-matrix by subtracting the outer product
        for ( int j = k+1; j < n; j++ ) {
            double *A_j = A + j*ldim;
            double akj = *(A_j + k);
            for ( int i = k+1; i < n; i++ ) {
                *(A_j+i) -= *(A_k+i) * akj;
            }
        }
    }
}

/*
 * Implements the SAXPY operation using jki indexing to factorize nonsingular
 * n-by-n matrix A into a unit lower triangular matrix L and upper triangular
 * matrix U, such that A = L*U.  Elements of L are stored in A(k+1:n-1,k), while
 * elements of U are stored in A(0:k,k), assuming base 0 indexing.  The inner-
 * most loop subtracts a scalar multiple of a vector from another vector.
 */
void lu_saxpy( int n, double *A )
{
    const int ldim = n;

    lu_factor( n, n, ldim, A );
}

/*
 * Implements simple blocking to factorize nonsingular n-by-n matrix A into
 * a unit lower triangular matrix L and an upper triangular matrix U, such that
 * A = L*U.  Suppose A is decomposed into blocks [A_00, A_01; A_10, A_11],
 * where A_00 is an r-by-r matrix block.  First, a rectangular unblocked version
 * of the SAXPY operation for LU factorization computes [L_00; L_10] and U_00.
 * Given that A_01 = L_00 * U_01, we can solve for U_01 using forward
 * substitution.  Then the trailing sub-matrix is updated,
 * A_11 = A_11 - L_10 * U_01.  This procedure is repeated iteratively on the
 * trailing sub-matrix until the last diagonal block (dimension <= r) is reached.
 * Simple blocking is also used to optimize memory access when updating the
 * trailing sub-matrix.
 */
void lu_block( int n, double *A )
{
    const int ldim = n;
    const int bdim = get_block_dim_lu( ldim );

    int        r, t;
    double  *Ajj, *L, *U;

    Ajj = A;
    r = (bdim > n) ? n : bdim;
    lu_factor(n, r, ldim, Ajj);

    t = n - bdim;
    for ( int j = bdim; j < n; j += bdim, t -= bdim ) {
        U = Ajj + bdim*ldim;
        tri_solve_l1xb( bdim, t, ldim, Ajj, U );
        L = Ajj + bdim;
        Ajj = A + j*ldim + j;
        reduce_mat_blk( t, t, bdim, ldim, bdim, L, U, Ajj );
        r = (t < bdim) ? t : bdim;
        lu_factor( t, r, ldim, Ajj );
    }
}

/*
 * Implements recursive contiguous blocking to factorize nonsingular n-by-n
 * matrix A into a unit lower triangular matrix L and an upper triangular
 * matrix U, such that A = L*U.  Matrix A, which is stored in column-major
 * order is first copied to array AA, which stores recursive contiguous blocks.
 * That is, matrix blocks are stored contiguously, and within each block, sub-
 * blocks of size KDIM*KDIM are stored contiguously.  Gaussian elimination
 * yields factors L and U stored in recursive contiguous blocks in array AA,
 * which is then copied to array A, where matrix elements are stored in
 * conventional column-major order.
 */
void lu_recur_block( int n, double *A )
{
    const int   nn = (n / KDIM) * KDIM + ((n % KDIM) ? KDIM : 0);
    const int   ldim = nn;
    const int   bdim = get_block_dim_lu( ldim );

    double  *AA;

    AA = (double *) malloc( ldim*ldim*sizeof(double) );
    form_recur_blocks( n, n, n, A, nn, nn, KDIM, bdim, ldim, AA );

    for ( int j = 0; j < nn; j += bdim ) {
        int s = (j + bdim > n) ? (n - j) : bdim;
        int q = (j + bdim > nn) ? (nn - j) : bdim;
        double *A_j = AA + j*ldim;
        const double *U_j = A_j;

        // Solve for L*X = A using forward substitution, and perform cumulative
        // trailing sub-matrix updates on matrix blocks above the diagonal
        for ( int k = 0; k < j; k += bdim ) {
            const double *L_k = AA + k*ldim;
            const double *Lkk = L_k + k*bdim;
            const double *Ukj = U_j + k*q;
            double *Akj = A_j + k*q;
            tri_solve_l1xb_blk_ker(bdim, s, bdim, Lkk, bdim, Akj);
            for ( int i = k+bdim; i < j; i += bdim ) {
                int r = (i + bdim > n) ? (n - i) : bdim;
                int p = (i + bdim > nn) ? (nn - i) : bdim;
                const double *Lik = L_k + i*bdim;
                double *Aij = A_j + i*q;
                reduce_blk_ker( r, s, bdim, p, Lik, bdim, Ukj, p, Aij );
            }
        }

        // Perform cumulative trailing sub-matrix updates on diagonal block
        // and matrix blocks below the diagonal
        for ( int k = 0; k < j; k += bdim ) {
            const double *L_k = AA + k*ldim;
            const double *Ukj = U_j + k*q;
            for ( int i = j; i < nn; i += bdim ) {
                int r = (i + bdim > n) ? (n - i) : bdim;
                int p = (i + bdim > nn) ? (nn - i) : bdim;
                const double *Lik = L_k + i*bdim;
                double *Aij = A_j + i*q;
                reduce_blk_ker(r, s, bdim, p, Lik, bdim, Ukj, p, Aij);
            }
        }

        // Factorize diagonal block, and solve X*U = A using forward
        // substitution on blocks below the diagonal
        double *Ajj = A_j + j*q;
        const double *Ujj = Ajj;
        lu_blk_ker(s, q, Ajj);
        for ( int i = j+BDIM; i < nn; i += bdim ) {
            int r = (i + bdim > n) ? (n - i) : bdim;
            int p = (i + bdim > nn) ? (nn - i) : bdim;
            double *Aij = A_j + i*bdim;
            tri_solve_xub_blk_ker(r, bdim, bdim, Ujj, p, Aij);
        }
    }
    unpack_recur_blocks( nn, nn, KDIM, bdim, ldim, AA, n, n, n, A );
    free( AA );
}

/***************************************************************************/

/*
 * In general, Gaussian elimination requires pivoting to ensure numerical
 * stability.  We implement partial pivoting to compute the LU factorization of
 * row permuted matrix P*A = L*U, where the permutation matrix is encoded in a
 * pivot vector.  The factorization overwrites matrix A with unit lower
 * triangular matrix L and upper triangular matrix U.  Although, one only needs
 * the pivot vector and the unit lower and upper triangular factors to solve the
 * the corresponding linear system, the function prototypes have additional
 * arguments to be consistent with matrix factorizations that implement a
 * variety of more complicated pivoting strategies.  This enables the use of
 * function pointers to invoke matrix factorizations, which specify a pivoting
 * strategy in the argument list and return all neccessary information to solve
 * the corresponding linear system.
 */

/*
```

```
 * Employs the outer product method (kji indexing) with partial pivoting to
 * factorize nonsingular n-by-n matrix A into a unit lower triangular matrix L
 * and upper triangular matrix U.  Row permuted matrix A^ = P*A = L*U.
 * Permutation matrix P is encoded in vectors piv[] and ord[], such that row k
 * is interchanged with row piv[k] and ord[k] = 1 is the diagonal block order.
 * Elements of L are stored in A(k+1:n-1,k), while elements of U are stored in
 * A(0:k,k), assuming base 0 indexing.  Each pass through the k-loop performs
 * an outer product operation.
 */
void lu_pivot_outer_product( char pivot, int n, int *piv, int *ord, double *A )
{
    const int ldim = n;

    for ( int k = 0; k < n-1; k++ ) {

        double *A_k = A + k*ldim;
        double *Akk = A_k + k;
        // Determine pivot for column k and interchange elements in the pivot
        // row with elements in row k
        switch ( pivot ) {
        case 'G':
            eval_pivot_gauss( n-k, k, Akk, piv, ord );
            break;
        default:
            eval_pivot_gauss( n-k, k, Akk, piv, ord );
            break;
        }
        if ( k != piv[k] ) {
            for ( int j = 0; j < n; j++ ) {
                double akj = *(A + k + j*ldim);
                *(A + k + j*ldim) = *(A + piv[k] + j*ldim);
                *(A + piv[k] + j*ldim) = akj;
            }
        }
        // Divide elements of column k below the diagonal by the diagonal element
        double akk = *Akk;
        for ( int i = k+1; i < n; i++ ) {
            *(A_k + i)  /= akk;
        }
        // Update trailing sub-matrix by subtracting the outer product
        for ( int j = k+1; j < n; j++ ) {
            double *A_j = A + j*ldim;
            double akj = *(A_j + k);
            for ( int i = k+1; i < n; i++ ) {
                *(A_j+i) -= *(A_k+i) * akj;
            }
        }
    }
    piv[n-1] = n-1;
}
```

```
/*
 * Employs the SAXPY operation (jki indexing) with partial pivoting to factorize
 * nonsingular n-by-n matrix A into a unit lower triangular matrix L and upper
 * triangular matrix U.  Row permuted matrix A^ = P*A = L*U.  Permutation matrix
 * P is encoded in vectors piv[] and ord[], such that row k is interchanged with
 * row piv[k] and ord[k] = 1 is the diagonal block order.  Elements of L are
 * stored in A(k+1:n-1,k), while elements of U are stored in A(0:k,k), assuming
 * base 0 indexing.  The inner-most loop subtracts a scalar multiple of vector
 * from another vector.
 */
void lu_pivot_saxpy( char pivot, int n, int *piv, int *ord, double *A )
{
    const int ldim = n;

    lu_pivot( pivot, n, n, ldim, piv, ord, A );
}

/*
 * Implements simple blocking with partial pivoting to factorize nonsingular
 * n-by-n matrix A into a unit lower triangular matrix L and an upper triangular
 * matrix U.  Row permuted matrix A^ = P*A = L*U.  Permutation matrix P is
 * encoded in vectors piv[] and ord[], such that row k is interchanged with
 * row piv[k], and ord[k] = 1 is the diagonal block order. Suppose A is
 * decomposed into blocks [A_00, A_01; A_10, A_11], where A_00 is an r-by-r
 * matrix block.  First, a rectangular version of the SAXPY operation for LU
 * factorization with partial pivoting computes
 * P*[A_00; A_10] = [L_00; L_10]*U_00.  Let [A_01^; A_11^] = P*[A_01; A_11].
 * Given that A_01^ = L_00*U_01, solve for U_01.  Then the trailing sub-matrix
 * is updated, A_11^ = A_11^ - L_10*U_01. This procedure is repeated iteratively
 * on the trailing sub-matrix until the last diagonal block (dimension <= r) is
 * reached. Simple blocking is also used to optimize memory access when updating
 * the trailing sub-matrix.
 */
void lu_pivot_block( char pivot, int n, int *piv, int *ord, double *A )
{
    const int ldim = n;
    const int bdim = get_block_dim_lu( ldim );

    int       d, j, r, t;
    double  *Ajj, *L, *U;

    j = 0;
    Ajj = A;
    r = (bdim > n) ? n : bdim;
    // Perform rectangular factorization on first column block A(0:n-1,0:r)
    lu_pivot( pivot, n, r, ldim, &piv[j], &ord[j], Ajj );

    d = 0;
    j = bdim;
    t = n - bdim;
    for ( ; j < n; j += bdim, d += bdim, t -= bdim ) {
        U = Ajj + bdim*ldim;
        // Solve for U(j-BDIM:j,j:n-1) where
        // P * A(j-BDIM:j,j:n-1) = L(j-BDIM:j,j-BDIM:j) * U(j-BDIM:j,j:n-1)
        tri_solve_l1xb_pivot( bdim, t, ldim, &piv[d], Ajj, U );

        // Adjust pivot vector of previous block for diagonal offset
        for ( int i = d; i < j; i++ ) {
            piv[i] += d;
        }
        L = Ajj + bdim;
        Ajj = A + j + j*ldim;
        // Reduce trailing sub-matrix
        // P * A(j:n-1,j:n-1) = L(j:n-1,j-BDIM:j-1) * U(j-BDIM:j-1,j:n-1)
        reduce_mat_blk( t, t, bdim, ldim, bdim, L, U, Ajj );
        r = t < bdim ? t : bdim;
        // Perform rectangular factorization on column block A(j:n-1,j:j+r-1)
        lu_pivot( pivot, t, r, ldim, &piv[j], &ord[j], Ajj );

        // Apply permutation matrix for current block, encoded in piv(j:j+r-1),
        // to columns to the left of current block A(:,0:j-1)
        for ( int i = j; i < j+r; i++ ) {
            if ( i != piv[i] + j ) {
                for ( int k = 0; k < j; k++ ) {
                    double aik = *(A + i + k*ldim);
                    *(A + i + k*ldim) = *(A + piv[i] + j + k*ldim);
                    *(A + piv[i] + j + k*ldim) = aik;
                }
            }
        }
    }
    // Adjust pivot vector of last block for diagonal offset
    for ( int i = d; i < n; i++ ) {
        piv[i] += d;
    }
}

/*
 * Wrapper for calling LAPACK routine DGETRF which computes an LU factorization
 * of a nonsingular matrix using partial pivoting with row interchanges.
 */
void lu_pivot_lapack( char pivot, int n, int *piv, int *ord, double *A )
{
    const int   ldim = n;
    int         info = 0;

    dgetrf_(&n, &n, A, &ldim, piv, &info);
}
```

A.4. cholfact.c – Cholesky factorization.

```
/*
 * Algorithms implementing unblocked and blocked Cholesky factorizaton of
 * symmetric positive definite matrices representing linear systems.  Unblocked
 * algorithms include the outer product method and SAXPY operation, while
 * blocked algorithms include simple blocking, contiguous blocking and recursive
 * contiguous blocking.  One implementation of a blocked algorithm uses tuned
 * BLAS (Basic Linear Algebra Subroutines).  Also, function wrappers facilitate
 * calling unblocked and blocked LAPACK Cholesky factorization routines DPOTF2
 * and DPOTRF, respectively.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

#include "cholfact.h"
#include "lapack.h"
#include "matcom.h"
#include "timing.h"

static void reduce_sym_matrix( int diag, int m, int n, int p, int ldimL,
    const double *L, int ldimT, const double *T, int ldimA, double *A );
static void reduce_sym_mat_blk( int m, int n, int p, int ldim, int bdim,
    const double *L, const double *T, double *A );
static void reduce_sym_kernel( const int diag,
    const double *L, const double *T, double *A );
static void reduce_sym_blk_ker( int diag, int m, int n, int p, int ldimL,
    const double *L, int ldimT, const double *T, int ldimA, double *A );
static void tri_solve_xltb_matrix( int m, int n, int ldimL, const double *L,
    int ldimB, double *B );
static void tri_solve_xltb_mat_blk( int m, int n, int ldim, int bdim,
    const double *L, double *B );
static void tri_solve_xltb_kernel( const double *L, double *B );
static void tri_solve_xltb_blk_ker( int m, int n, int ldimL, const double *L,
    int ldimB, double *B );
static void chol_kernel( const int n, double *A );
static void chol_blk_ker( int n, int ldim, double *A );
static void chol_factor( int m, int n, int ldim, double *A );

/***************************************************************************/

/*
 * Determines optimal block dimension for the local environment given a routine
 * and matrix leading dimension.  The function returns the block dimension
 * chosen by the LAPACK Cholesky factorization routine, or a block dimension for
 * testing (debugging).  If the leading dimension is less than the optimal block
 * dimension, the block dimension is set to the leading dimension, and the
 * matrix computation becomes an unblocked algorithm.
 */
int get_block_dim_chol( int ldim )
{
    const int   optm_bdim = 1;
    const int   no_dim = -1;
    const char  *parm_str = "L";
    const char  *func_name = "DPOTRF";

    int bdim;

#if defined(DEBUG)
    bdim = BDIM;
#else
    bdim = ilaenv_( &optm_bdim, func_name, parm_str,
        &ldim, &no_dim, &no_dim, &no_dim );
#endif
    if ( bdim <= 1 || bdim > ldim ) {
        bdim = ldim;
    }
    return bdim;
}

/*
 * Matrix factorization reduces symmetric trailing sub-matrix A by computing
 * A = A - L*T', where A is an m-by-n sub-matrix, and L and T are m-by-p and
 * n-by-p blocks, respectively, of a lower triangular matrix.  A, L, and T are
 * stored in column-major order with leading dimensions ldimL, ldimT and ldimA,
 * respectively.  The trailing sub-matrix update is an implementation of the
 * SAXPY operation.  Because of symmetry, the trailing sub-matrix update need
 * only be performed on elements on or below the diagonal.
 */
void reduce_sym_matrix( int diag, int m, int n, int p, int ldimL,
    const double *L, int ldimT, const double *T, int ldimA, double *A )
{
    for ( int j = 0; j < n; j++ ) {
        const double *Tj_ = T + j;                 // Points to element T(j,0)
        double *A_j = A + j*ldimA;                 // Points to element A(0,j)
        for ( int k = 0; k < p; k++ ) {
            const double *L_k = L + k*ldimL;       // Points to element L(0,k)
            double tjk = *(Tj_ + k*ldimT);         // Element T(j,k) = T'(k,j)
            for ( int i = diag ? j : 0; i < m; i++ ) {
                *(A_j + i) -= *(L_k + i) * tjk;
            }                                      // A(i,j) -= L(i,k) * T'(k,j)
        }
    }
}

/*
 * Matrix factorization reduces symmetric trailing sub-matrix A by computing
 * A = A - L*T', where A is an m-by-n sub-matrix, and L and T are m-by-p and
 * n-by-p column blocks, respectively, of a lower triangular matrix.  A, L and T
 * are stored in column-major order with leading dimension ldim.  Blocking is
 * used to optimize memory access for the trailing sub-matrix update, and bdim
 * is the blocking parameter.  Because of symmetry, the trailing sub-matrix
 * update need only be performed on diagonal blocks and blocks below the
 * diagonal.
 */
void reduce_sym_mat_blk( int m, int n, int p, int ldim, int bdim,
    const double *L, const double *T, double *A )
{
    for ( int j = 0; j < n; j += bdim ) {
        // Determine number of columns in (i,j)th block of A
        int s = (j + bdim > n) ? (n - j) : bdim;

        for ( int k = 0; k < p; k += bdim ) {
            int diag = 1;        // Diagonal block = TRUE
            // Determine number of columns of Lik and Tjk (rows of T'kj)
            int t = (k + bdim > p) ? (p - k) : bdim;
            // Set pointer to block matrix Tjk
            const double *Tjk = T + j + k*ldim;

            for ( int i = j; i < m; i += bdim ) {
                // Determine number of rows in (i,j)th block of A
                int r = (i + bdim > m) ? (m - i) : bdim;
                // Set pointers to block matrices Lik and Aij
                const double *Lik = L + i + k*ldim;
                double *Aij = A + i + j*ldim;
                // Reduce trailing block matrix
                reduce_sym_matrix( diag, r, s, t,
                    ldim, Lik, ldim, Tjk, ldim, Aij );
                diag = 0;        // Diagonal block = FALSE
            }
        }
    }
}

/*
 * Matrix factorization reduces symmetric trailing sub-matrix by computing
 * A = A - L*T', where A is a contiguous KDIM-by-KDIM sub-block of the trailing
 * sub-matrix, and L and T are contiguous KDIM-by-KDIM sub-blocks of a lower
 * triangular matrix. Looping is controlled by a symbolic constant (KDIM), which
 * is evaluated during compilation.  The trailing sub-matrix update is an
 * implementation of the SAXPY operation.  Because of symmetry, the trailing
 * sub-matrix update need only be performed on elements on and below the diagonal.
 */
void reduce_sym_kernel( const int diag,
    const double *L, const double *T, double *A )
{
    for ( int j = 0; j < KDIM; j++ ) {
        const double *Tj_ = T + j;                  // Points to element T(j,0)
        double *A_j = A + j*KDIM;                    // Points to element A(0,j)
        for ( int k = 0; k < KDIM; k++ ) {
            const double *L_k = L + k*KDIM;          // Points to element L(0,k)
            double tjk = *(Tj_ + k*KDIM);            // Element T(j,k) = T'(k,j)
            for ( int i = diag ? j : 0; i < KDIM; i++ ) {
                *(A_j + i) -= *(L_k + i) * tjk;      // A(i,j) -= L(i,k) * T'(k,j)
            }
        }
    }
}

/*
 * Matrix factorization reduces symmetric trailing sub-matrix by computing
 * A = A - L*T', where A is an m-by-n block of the trailing sub-matrix,
 * and L and T are m-by-p and n-by-p blocks of a lower triangular matrix.
 * Matrix blocks A, L and T are stored contiguously with leading dimension
 * ldimA, ldimL and ldimT, respectively.  Within blocks of A, L and T, sub-
 * blocks of size KDIM*KDIM are stored contiguously.  Because of symmetry, the
 * trailing sub-matrix update need only be performed on diagonal sub-blocks and
 * sub-blocks below the diagonal.
 */
void reduce_sym_blk_ker( int diag, int m, int n, int p, int ldimL,
    const double *L, int ldimT, const double *T, int ldimA, double *A )
{
    for ( int j = 0; j < n; j += KDIM ) {

        for ( int k = 0; k < p; k += KDIM ) {
            int diag_blk = diag;    // Diagonal block --> diagonal sub-block
            // Set pointer to sub-block Tjk (T'kj)
            const double *Tjk = T + j*KDIM + k*ldimT;

            for ( int i = diag_blk ? j : 0; i < m; i += KDIM ) {
                // Set pointers to sub-blocks Lik and Aij
                const double *Lik = L + i*KDIM + k*ldimL;
                double *Aij = A + i*KDIM + j*ldimA;
                // Perform matrix reduction on sub-blocks
                reduce_sym_kernel( diag_blk, Lik, Tjk, Aij );
                diag_blk = 0;   // Diagonal sub-block = FALSE
            }
        }
    }
}

/*
 * Uses forward substitution to solve the triangular system of linear equations
 * X*L' = B, where L is an n-by-n lower triangular matrix and L' its transpose,
 * and X and B are m-by-n matrices.  L and B are stored in column-major order
 * with leading dimensions ldimL and ldimB, respectively.  The solution X
 * overwrites B.
 */
```

```
void tri_solve_xltb_matrix( int m, int n, int ldimL, const double *L,
    int ldimB, double *B )
{
    for ( int k = 0; k < n; k++ ) {
        double lkk = *(L + k + k*ldimL);
        double *B_k = B + k*ldimB;
        for (int i = 0; i < m; i++) {
            *(B_k + i) /= lkk;
        }
        for ( int j = k+1; j < n; j++ ) {
            double ljk = *(L + j + k*ldimL);             // Element L(j,k) = L'(k,j)
            double *B_j = B + j*ldimB;
            for (int i = 0; i < m; i++) {
                *(B_j+i) -= *(B_k+i) * ljk;
            }
        }
    }
}

/*
 * Uses forward substitution to solve the triangular system of linear equations
 * X*L' = B, where L is an n-by-n lower triangular matrix and L' its transpose,
 * and X and B are m-by-n matrices.  X, L and B are stored in column-major order
 * with leading dimension ldim.  Blocking is used to optimize memory access for
 * the triangular solve operation, and bdim is the blocking parameter.
 */
void tri_solve_xltb_mat_blk( int m, int n, int ldim, int bdim,
    const double *L, double *B )
{
    for ( int i = 0; i < m; i += bdim ) {
        int r = (i + bdim > m) ? (m - i) : bdim;
        tri_solve_xltb_matrix( r, n, ldim, L, ldim, B+i );
    }
}

/*
 * Uses forward substitution to solve the triangular system of linear equations
 * X*L' = B, where X, L and B are contiguous KDIM-by-KDIM matrix sub-blocks, and
 * L is lower triangular and L' its transpose.  Looping is controlled by a
 * symbolic constant (KDIM), which is evaluated during compilation.  The
 * solution X overwrites B.
 */
void tri_solve_xltb_kernel( const double *L, double *B )
{
    for ( int k = 0; k < KDIM; k++ ) {
        double lkk = *(L + k + k*KDIM);
        double *B_k = B + k*KDIM;
        for ( int i = 0; i < KDIM; i++ ) {
            *(B_k + i) /= lkk;
        }
        for ( int j = k+1; j < KDIM; j++ ) {
            double ljk = *(L + j + k*KDIM);             // Element L(j,k) = L'(k,j)
            double *B_j = B + j*KDIM;
            for ( int i = 0; i < KDIM; i++ ) {
                *(B_j+i) -= *(B_k+i) * ljk;
            }
        }
    }
}

/*
 * Uses forward substitution to solve the triangular system of linear equations
 * X*L' = B, where L is an n-by-n lower triangular matrix block and L' its
 * transpose, and X and B are m-by-n matrix blocks.  Matrix blocks L and B are
 * stored contiguously with leading dimension ldimL and ldimB, respectively.
 * Within blocks of L and B, sub-blocks of size KDIM*KDIM are stored
 * contiguously.  Suppose that L' is decomposed into sub-blocks
 * [L_00', L_10'; 0, L_11'].  Then X_00*L_00' = B_00; X_10*L_00' = B_10;
 * X_00*L_10' + X_01*L_11' = B_01 --> X_01*L_11' = B_01 - X_00*L_10'; and
 * X_10*L_10' + X_11*L_11' = B_11 --> X_11*L_11' = B_11 - X_10*L_10'.
 */
void tri_solve_xltb_blk_ker( int m, int n, int ldimL, const double *L,
    int ldimB, double *B )
{
    for ( int j = 0; j < n; j += KDIM ) {
        const double *Lj_ = L + j*KDIM;
        const double *Ljj = Lj_ + j*ldimL;
        double *B_j = B + j*ldimB;

        for ( int k = 0; k < j; k += KDIM ) {
            const double *Ljk = Lj_ + k*ldimL;
            double *B_k = B + k*ldimB;

            for ( int i = 0; i < m; i += KDIM ) {
                double *Bik = B_k + i*KDIM;
                double *Bij = B_j + i*KDIM;
                reduce_sym_kernel( 0, Bik, Ljk, Bij );
            }

        }
        for (int i = 0; i < m; i += KDIM) {
            double *Bij = B_j + i*KDIM;
            tri_solve_xltb_kernel( Ljj, Bij );
        }
    }
}

/*
 * Factorizes an n-by-n symmetric matrix sub-block A into a lower triangular
 * sub-block L, such that A = L*L'.  KDIM-by-KDIM sub-block A is stored
 * contiguously.  The Cholesky factorization algorithm is an implementation of
 * the SAXPY operation.  Looping is controlled by a symbolic constant (KDIM),
 * which is evaluated during compilation.  The factor L overwrites elements of A
 * on and below the diagonal.
 */
void chol_kernel( const int n, double *A )
{
    // Divide elements of the first column by square root of element in first row
    double ajj = sqrt(*A);
    for ( int i = 0; i < KDIM; i++ ) {
        *(A + i) /= ajj;
    }

    for ( int j = 1; j < n; j++ ) {
        // Perform cumulative trailing sub-matrix updates on diagonal element
        // and elements below the diagonal of column j
        double *A_j = A + j*KDIM;
        for ( int k = 0; k < j; k++ ) {
            double *L_k = A + k*KDIM;                  // Points to  L(0,k) = A(0,k)
            double ljk = *(L_k + j);                   // Element L(j,k) = L'(k,j)
            for ( int i = j; i < KDIM; i++ ) {
                *(A_j + i) -= *(L_k + i) * ljk;        // A(i,j) -= L(i,k) * L'(k,j)
            }
        }

        // Divide elements of column j by square root of the diagonal element
        ajj = sqrt( *(A_j + j) );
        for ( int i = j; i < KDIM; i++ ) {
            *(A_j+i) /= ajj;
        }
    }
}

/*
 * Factorizes an n-by-n symmetric matrix block A into a lower triangular block L,
 * such that A = L*L'.  Matrix block A is stored contiguously with leading
 * dimension ldim, and within the matrix block, sub-blocks of size KDIM*KDIM are
 * stored contiguously.  Because of symmetry, the Cholesky factorization need
 * only be performed on diagonal sub-blocks and sub-blocks below the diagonal.
 */
void chol_blk_ker( int n, int ldim, double *A )
{
    const double *L, *T;

    for ( int j = 0; j < n; j += KDIM ) {
        const int s = (j + KDIM > n) ? (n - j) : KDIM;
        double *A_j = A + j*ldim;
        double *Ajj = A_j + j*KDIM;
        T = A + j*KDIM;
        // Perform cumulative trailing sub-matrix updates on diagonal sub-block
        // and sub-blocks below the diagonal
        for ( int k = 0; k < j; k += KDIM ) {
            int diag = 1;          // Diagonal block = TRUE
            L = T;

            for ( int i = j; i < n; i += KDIM ) {
                double *Aij = A_j + i*KDIM;
                reduce_sym_kernel( diag, L, T, Aij );
                L = L + KDIM*KDIM;
                diag = 0;          // Diagonal block = FALSE
            }
            T = T + KDIM*ldim;
        }

        // Factorize diagonal sub-block, and solve X*L' = A using forward
        // substitution on sub-blocks below the diagonal
        chol_kernel( s, Ajj );
        T = Ajj;
        for ( int i = j+KDIM; i < n; i += KDIM ) {
            double *Aij = A_j + i*KDIM;
            tri_solve_xltb_kernel( T, Aij );
        }
    }
}

/*
 * Implements a rectangular version the SAXPY operation (jki indexing) for
 * Cholesky factorization.  Symmetric positive definite m-by-n matrix A with
 * leading dimension ldim is factored into a lower triangular matrix L, such
 * that A = L*L', where L' is the transpose of L.  Elements of L are stored in
 * A(k:n-1,k), base 0 indexing i.e., on and below the diagonal.  The inner-most
 * loop subtracts a scalar multiple of a vector from another vector.
 */
void chol_factor( int m, int n, int ldim, double *A )
{
    for ( int j = 0; j < n; j++ ) {
        // Perform cumulative trailing sub-matrix updates on diagonal element
        // and elements below the diagonal of column j
        double *A_j = A + j*ldim;
        for ( int k = 0; k < j; k++ ) {
            double *L_k = A + k*ldim;                // Element L(0,k) = A(0,k)
            double ljk = *(L_k + j);                 // Element L(j,k) = L'(k,j)
            for ( int i = j; i < m; i++ ) {
                *(A_j + i) -= *(L_k + i) * ljk;      // A(i,j) -= L(i,k) * L'(k,j)
            }
        }

        // Divide elements of column j by square root of the diagonal element
        double ajj = sqrt(*(A_j + j));
        for ( int i = j; i < m; i++ ) {
            *(A_j+i) /= ajj;
        }
    }
}

/***************************************************************************/

/*
 * Implements the outer product method (kji indexing) to factorize symmetric
 * positive definite n-by-n matrix A into a lower triangular matrix L, such that
 * A = L*L', where L' is the transpose of L.  Symmetric postive definite
 * matrices have weighty diagonals, which precludes the need for pivoting.
 * Elements of L are stored in A(k:n-1,k), base 0 indexing i.e., on and below
 * the diagonal.  Each pass through the k-loop performs an outer product
 * operation.
 */
void chol_outer_product( int n, double *A )
{
    const int ldim = n;

    for ( int k = 0; k < n; k++ ) {

        // Divide elements of column k on and below the diagonal by the
        // square root of the diagonal element
        double *A_k = A + k*ldim;
        double akk = sqrt( *(A_k + k) );
        *(A_k + k) = akk;
        for ( int i = k+1; i < n; i++ ) {
            *(A_k + i)  /= akk;
        }
        // Update trailing sub-matrix by subtracting the outer product
        for ( int j = k+1; j < n; j++ ) {
            double *A_j = A + j*ldim;
            double ajk = *(A_k + j);
            for ( int i = j; i < n; i++ ) {
                *(A_j+i) -= *(A_k+i) * ajk;
            }
        }
    }
}

/*
 * Implements the SAXPY operation (jki indexing) to factorize symmetric positive
 * definite n-by-n matrix A into a lower triangular matrix L, such that A = L*L',
 * where L' is the transpose of L.   Symmetric postive definite matrices have
 * weighty diagonals, which precludes the need for pivoting. Elements of L are
 * stored in A(k:n-1,k), base 0 indexing i.e., on and below the diagonal.  The
 * inner-most loop subtracts a scalar multiple of a vector from another vector.
 */
```

```
void chol_saxpy( int n, double *A )
{
    const int ldim = n;

    chol_factor( n, n, ldim, A );
}

/*
 * Implements simple blocking to factorize symmetric positive definite n-by-n
 * matrix A into a lower triangular matrix L, such that A = L*L', where L' is the
 * transpose of L.  Suppose A is decomposed into blocks [A_00, A_01; A_10, A_11],
 * where A_00 is an r-by-r block matrix.  First, an implementation of the SAXPY
 * operation computes the Cholesky factorization of r-by-r diagonal block,
 * A_00 = L_00*L_00'.  Then, solve for L_10 in the triangular system of linear
 * equations L_00 * L_10' = A_10', and update the trailing sub-matrix,
 * A_11 = A_11 - L_10 * L_10'.  This procedure is repeated iteratively on the
 * trailing sub-matrix until the last diagonal block (dimension less than or
 * equal to r) is reached.  Simple blocking is also used to optimize memory
 * access when updating the trailing sub-matrix.
 */
void chol_block( int n, double *A )
{
    const int ldim = n;
    const int bdim = get_block_dim_chol( ldim );

    int     r, t;
    double  *Ajj, *L;

#if defined(CHOLFACT) && defined(PROFILE)
    struct      timespec sta_chol, sta_factor, sta_tri_solve, sta_reduce,
                end_chol, end_factor, end_tri_solve, end_reduce;
    double tm_chol = 0.0;
    double tm_factor = 0.0;
    double tm_tri_solve = 0.0;
    double tm_reduce = 0.0;

    get_time( &sta_chol );
#endif

    Ajj = A;
    r = (bdim > n) ? n : bdim;
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &sta_factor );
#endif
    chol_factor( r, r, ldim, Ajj );
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &end_factor );
    tm_factor += timespec_diff( sta_factor, end_factor );
#endif
```

```
    for ( int j = bdim; j < n; j += bdim ) {
        t = n - j;
        L = Ajj + bdim;
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &sta_tri_solve );
#endif
        tri_solve_xltb_mat_blk( t, r, ldim, bdim, Ajj, L );
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &end_tri_solve );
    tm_tri_solve += timespec_diff( sta_tri_solve, end_tri_solve );
#endif
        Ajj = A + j*ldim + j;
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &sta_reduce );
#endif
        reduce_sym_mat_blk( t, t, bdim, ldim, bdim, L, L, Ajj );
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &end_reduce );
    tm_reduce += timespec_diff( sta_reduce, end_reduce );
#endif
        r = (j + bdim > n) ? t : bdim;
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &sta_factor );
#endif
        chol_factor( r, r, ldim, Ajj );
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &end_factor );
    tm_factor += timespec_diff( sta_factor, end_factor );
#endif
    }

#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &end_chol );
    tm_chol += timespec_diff( sta_chol, end_chol );
    fprintf( stdout, "%.3f\t%.3f\t\t%.3f\t\t%.3f\t\t%.1f\t\t%.1f\t\t%.1f\n",
    tm_chol, tm_factor, tm_tri_solve, tm_reduce,
    tm_factor/tm_chol*100, tm_tri_solve/tm_chol*100, tm_reduce/tm_chol*100 );
#endif
}

/*
 * Implements simple blocking to factorize symmetric positive definite n-by-n
 * matrix A into a lower triangular matrix L, such that A = L*L', where L' is the
 * transpose of L.  Suppose A is decomposed into blocks [A_00, A_01; A_10, A_11],
 * where A_00 is an r-by-r block matrix.  First, a rectangular version of the
 * SAXPY operation computes the Cholesky factorization of n-by-r column block
 * [A_00; A_10] = [L_00; L10]*L_00'.  Then, update the trailing sub-matrix,
 * A_11 = A_11 - L_10 * L_10'.  This procedure is repeated iteratively on the
 * trailing sub-matrix until the last diagonal block (dimension less than or
 * equal to r) is reached.  Simple blocking is also used to optimize memory
 * access when updating the trailing sub-matrix.
 */
void chol_rect_block( int n, double *A )
{
    const int ldim = n;
    const int bdim = get_block_dim_chol( ldim );

    int      r, t;
    double  *Ajj, *L;

    Ajj = A;
    r = (bdim > n) ? n : bdim;
    chol_factor( n, r, ldim, Ajj );

    for ( int j = bdim; j < n; j += bdim ) {
        t = n - j;
        L = Ajj + bdim;
        Ajj = A + j*ldim + j;
        reduce_sym_mat_blk( t, t, bdim, ldim, bdim, L, L, Ajj );
        r = (j + bdim > n) ? t : bdim;
        chol_factor( t, r, ldim, Ajj );
    }
}

/*
 * Implements contiguous blocking to factorize symmetric positive definite
 * n-by-n matrix A into a lower triangular matrix L such that A = L*L', where
 * L' is the transpose of L. Matrix A, which is stored in column-major order
 * is first copied to array AA, which stores contiguous blocks. Cholesky
 * factorization yields lower triangular matrix L stored in contiguous blocks
 * in array AA, which is then copied to array A, where matrix elements are
 * stored in conventional column-major order.
 */
void chol_contig_block( int n, double *A )
{
    const int   ldim = n;
    const int   bdim = get_block_dim_chol( ldim );

    double  *AA, *L, *T;

    AA = (double *) malloc( ldim*ldim*sizeof(double) );
    form_contig_blocks( n, n, ldim, A, n, n, bdim, ldim, AA );
    for ( int j = 0; j < n; j += bdim ) {
        int s = (j + bdim > n) ? (n - j) : bdim;
        double *A_j = AA + j*ldim;
        double *Ajj = A_j + j*s;
        T = AA + j*bdim;

        // Perform cumulative trailing sub-matrix updates on diagonal block
        // and matrix blocks below the diagonal
        for ( int k = 0; k < j; k += bdim ) {
            int diag = 1;        // Diagonal block = TRUE
            L = T;

            for ( int i = j; i < n; i += bdim ) {
                int r = (i + bdim > n) ? (n - i) : bdim;
                double *Aij = A_j + i*s;
                reduce_sym_matrix( diag, r, s, bdim, r, L, s, T, r, Aij );
                L = L + bdim*bdim;
                diag = 0;        // Diagonal block = FALSE
            }
            T = T + bdim*ldim;
        }

        // Factorize diagonal block, and solve X*L' = A using forward
        // substitution on blocks below the diagonal
        chol_factor( s, s, s, Ajj );
        T = Ajj;
        for ( int i = j+bdim; i < n; i += bdim ) {
            int r = (i + bdim > n) ? (n - i) : bdim;
            double *Aij = A_j + i*bdim;
            tri_solve_xltb_matrix( r, bdim, bdim, T, r, Aij );
        }
    }
    unpack_contig_blocks( n, n, bdim, ldim, AA, n, n, ldim, A );
    free( AA );
}

/*
 * Implements recursive contiguous blocking to factorize symmetric positive
 * definite n-by-n matrix A into a lower triangular matrix L such that A = L*L',
 * where L' is the transpose of L.  Matrix A, which is stored in column-major
 * order is first copied to array AA, which stores recursive contiguous blocks.
 * That is, matrix blocks are stored contiguously, and within each block, sub-
 * blocks of size KDIM*KDIM are stored contiguously.  Cholesky factorization
 * yields lower triangular matrix L stored in recursive contiguous blocks in
 * array AA, which is then copied to array A, where matrix elements are stored
 * in conventional column-major order.
 */
void chol_recur_block( int n, double *A )
{
    const int    nn = (n / KDIM) * KDIM + ((n % KDIM) ? KDIM : 0);
    const int    ldim = nn;

    int      bdim, bdim_low, bdim_high;
    double   *AA, *L, *T;

    bdim = get_block_dim_chol( ldim );
    bdim_low = (bdim / KDIM) * KDIM;
    bdim_high = (bdim / KDIM) * KDIM + ((bdim % KDIM) ? KDIM : 0);
    if ( bdim_low == 0 ) {
        bdim = bdim_high;
    } else {
        if ( (bdim - bdim_low) > (bdim_high - bdim) ) {
            bdim = bdim_high;
        } else {
            bdim = bdim_low;
        }
    }

    AA = (double *) malloc( ldim*ldim*sizeof(double) );
    form_recur_blocks( n, n, n, A, nn, nn, KDIM, bdim, ldim, AA);
    for ( int j = 0; j < nn; j += bdim ) {
        int s = (j + bdim > n) ? (n - j) : bdim;
        int q = (j + bdim > nn) ? (nn - j) : bdim;
        double *A_j = AA + j*ldim;
        double *Ajj = A_j + j*q;
        T = AA + j*bdim;

        // Perform cumulative trailing sub-matrix updates on diagonal block
        // and matrix blocks below the diagonal
        for ( int k = 0; k < j; k += bdim ) {
            int diag = 1;        // Diagonal block = TRUE
            L = T;

            for ( int i = j; i < nn; i += bdim ) {
                int r = (i + bdim > n) ? (n - i) : bdim;
                int p = (i + bdim > nn) ? (nn - i) : bdim;
                double *Aij = A_j + i*q;
                reduce_sym_blk_ker( diag, r, s, bdim, p, L, q, T, p, Aij );
                L = L + bdim*bdim;
                diag = 0;        // Diagonal block = FALSE
            }
            T = T + bdim*ldim;
        }

        // Factorize diagonal block, and solve X*L' = A using forward
        // substitution on blocks below the diagonal
        chol_blk_ker( s, q, Ajj );
        T = Ajj;
        for ( int i = j+bdim; i < nn; i += bdim ) {
            int r = (i + bdim > n) ? (n - i) : bdim;
            int p = (i + bdim > nn) ? (nn - i) : bdim;
            double *Aij = A_j + i*bdim;
            tri_solve_xltb_blk_ker( r, bdim, bdim, T, p, Aij );
        }
    }
    unpack_recur_blocks( nn, nn, KDIM, bdim, ldim, AA, n, n, n, A );
    free( AA );
}
```

```
/*
 * Implements simple blocking to factorize symmetric positive definite n-by-n
 * matrix A into a lower triangular matrix L, such that A = L*L', where L' is the
 * transpose of L.  Suppose A is decomposed into blocks [A_00, A_01; A_10, A_11],
 * where A_00 is an r-by-r block matrix.  First, an implementation of the SAXPY
 * operation computes the Cholesky factorization of r-by-r diagonal block,
 * A_00 = L_00*L_00'.  BLAS routine DTRSM solves for L_10 in the triangular
 * system of linear equations L_00 * L_10' = A_10'.  Then, BLAS routine DSYRK is
 * invoked to reduce the trailing sub-matrix, A_11 = A_11 - L_10 * L_10'.  This
 * procedure is repeated iteratively on the trailing sub-matrix until the last
 * diagonal block (dimension less than or equal to r) is reached.
 */
void chol_block_blas( int n, double *A )
{
    const char   lower = 'L';
    const char   trans = 'T';
    const char   no_trans = 'N';
    const char   rhs = 'R';
    const char   not_unit = 'N';
    const int    ldim = n;
    const int    bdim = get_block_dim_chol( ldim );
    const double         _one = -1.0;
    const double         one = 1.0;

    int          r, t;
    double   *Ajj, *L;

#if defined(CHOLFACT) && defined(PROFILE)
    struct       timespec sta_chol, sta_factor, sta_tri_solve, sta_reduce,
                 end_chol, end_factor, end_tri_solve, end_reduce;
    double tm_chol = 0.0;
    double tm_factor = 0.0;
    double tm_tri_solve = 0.0;
    double tm_reduce = 0.0;

    get_time( &sta_chol );
#endif

    Ajj = A;
    r = (bdim > n) ? n : bdim;
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &sta_factor );
#endif
    chol_factor( r, r, ldim, Ajj );
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &end_factor );
    tm_factor += timespec_diff( sta_factor, end_factor );
#endif
```

```
    for ( int j = bdim; j < n; j += bdim ) {
        t = n - j;
        L = Ajj + bdim;
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &sta_tri_solve );
#endif
        dtrsm_( &rhs, &lower, &trans, &not_unit, &t, &bdim,
            &one, Ajj, &ldim, L, &ldim );
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &end_tri_solve );
    tm_tri_solve += timespec_diff( sta_tri_solve, end_tri_solve );
#endif
        Ajj = A + j*ldim + j;
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &sta_reduce );
#endif
        dsyrk_( &lower, &no_trans, &t, &bdim, &_one, L, &ldim,
            &one, Ajj, &ldim );
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &end_reduce );
    tm_reduce += timespec_diff( sta_reduce, end_reduce );
#endif
        r = (j + bdim > n) ? t : bdim;
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &sta_factor );
#endif
        chol_factor( r, r, ldim, Ajj );
#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &end_factor );
    tm_factor += timespec_diff( sta_factor, end_factor );
#endif
    }

#if defined(CHOLFACT) && defined(PROFILE)
    get_time( &end_chol );
    tm_chol += timespec_diff( sta_chol, end_chol );
    fprintf( stdout, "%.3f\t%.3f\t\t%.3f\t\t%.3f\t\t%.1f\t\t%.1f\t\t%.1f\n",
    tm_chol, tm_factor, tm_tri_solve, tm_reduce,
    tm_factor/tm_chol*100, tm_tri_solve/tm_chol*100, tm_reduce/tm_chol*100 );
#endif
}

/*
 * Implements contiguous blocking to factorize symmetric positive definite
 * n-by-n matrix A into a lower triangular matrix L such that A = L*L', where
 * L' is the transpose of L.  Matrix A, which is stored in column-major order
 * is first copied to array AA, which stores contiguous blocks.  Cholesky
 * factorization yields lower triangular matrix L stored in contiguous blocks
 * in array AA, which is then copied to array A, where matrix elements are
 * stored in conventional column-major order.  LAPACK unblocked routine DPOTF2
 * computes the Cholesky factorization of a diagonal block; BLAS routine DTRSM
 * solves for blocks of the lower triangular matrix; and BLAS routines DSYRK
 * and DGEMM update the trailing sub-matrix.
 */
void chol_contig_block_blas( int n, double *A )
{
    const char  lower = 'L';
    const char  trans = 'T';
    const char  no_trans = 'N';
    const char  rhs = 'R';
    const char  not_unit = 'N';
    const int   ldim = n;
    const int   bdim = get_block_dim_chol( ldim );
    const double        _one = -1.0;
    const double        one = 1.0;

    int      info = 0;
    double  *AA, *L, *T;

    AA = (double *) malloc( ldim*ldim*sizeof(double) );
    form_contig_blocks( n, n, ldim, A, n, n, bdim, ldim, AA );

    for ( int j = 0; j < n; j += bdim ) {
        int s = (j + bdim > n) ? (n - j) : bdim;
        double *A_j = AA + j*ldim;
        double *Ajj = A_j + j*s;
        T = AA + j*bdim;

        // Perform cumulative trailing sub-matrix updates on diagonal block
        // and matrix blocks below the diagonal
        for ( int k = 0; k < j; k += bdim ) {
            int diag = 1;        // Diagonal block = TRUE
            L = T;

            for ( int i = j; i < n; i += bdim ) {
                int r = (i + bdim > n) ? (n - i) : bdim;
                double *Aij = A_j + i*s;
                if ( diag == 0 ) {
                    dgemm_( &no_trans, &trans, &r, &s, &bdim, &_one, L, &r,
                        T, &s, &one, Aij, &r );
                } else {
                    dsyrk_( &lower, &no_trans, &s, &bdim, &_one, L, &r,
                        &one, Aij, &r );
                }
                L = L + bdim*bdim;
                diag = 0;        // Diagonal block = FALSE
            }
            T = T + bdim*ldim;
        }
        // Factorize diagonal block, and solve X*L' = A using forward
        // substitution on blocks below the diagonal
        dpotf2_( &lower, &s, Ajj, &s, &info );
        T = Ajj;
        for ( int i = j+bdim; i < n; i += bdim ) {
            int r = (i + bdim > n) ? (n - i) : bdim;
            double *Aij = A_j + i*bdim;
        dtrsm_( &rhs, &lower, &trans, &not_unit, &r, &bdim,
            &one, T, &bdim, Aij, &r );
        }
    }
    unpack_contig_blocks( n, n, bdim, ldim, AA, n, n, ldim, A );
    free( AA );
}

/*
 * Wrapper for calling LAPACK routine DPOTF2 which computes the Cholesky
 * factorization of a real symmetric positive definite matrix. DPOTF2 is
 * LAPACK's unblocked version of Cholesky factorization.
 */
void chol_lapack_unblocked( int n, double *A )
{
    const char  lower = 'L';
    const int   ldim = n;
    int         info = 0;

    dpotf2_( &lower, &n, A, &ldim, &info );
}

/*
 * Wrapper for calling LAPACK routine DPOTRF, which computes the Cholesky
 * factorization of a real symmetric positive definite matrix.
 */
void chol_lapack( int n, double *A )
{
    const char  lower = 'L';
    const int   ldim = n;
    int         info = 0;

    dpotrf_( &lower, &n, A, &ldim, &info );
}
```

A.5. **ldltfact.c – symmetric indefinite factorization.**

```
/*
 * Algorithms implementing unblocked and blocked symmetric indefinite
 * factorizaton of matrices representing linear systems.  Unblocked algorithms
 * include the outer product method and SAXPY operation, while blocked
 * algorithms include simple blocking and an implementation that uses tuned
 * BLAS (Basic Linear Algebra Subroutines).  Also, function wrappers facilitate
 * calling unblocked and blocked LAPACK symmetric indefinite factorization
 * routines DSYTF2 and DSYTRF, respectively.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <float.h>

#include "ldltfact.h"
#include "lapack.h"
#include "matcom.h"
#include "timing.h"

static void eval_pivot_bk( int n, int d, int ldim, const double *A,
    int *piv, int *ord );
static void eval_pivot_bbk( int n, int d, int ldim, const double *A,
    int *piv, int *ord );
static void eval_pivot_bp( int n, int d, int ldim, const double *A,
    int *piv, int *ord );
static void eval_pivot_reduce_bk( int n, int d, int ldim, const double *L,
    const double *D, const double *A, double *Q, int *piv, int *ord );
static void eval_pivot_reduce_bbk( int n, int d, int ldim, const double *L,
    const double *D, const double *A, double *Q, int *piv, int *ord );
static void eval_pivot_blas_bk( int n, int d, int ldim, const double *L,
    const double *M, const double *A, double *Q, int *piv, int *ord );
static void eval_pivot_blas_bbk( int n, int d, int ldim, const double *L,
    const double *M, const double *A, double *Q, int *piv, int *ord );
static void pivot_sym_reduce( int n, int k, int r, double *vec, int ldim,
    double *A );
static void pivot_sym_blas( int n, int k, int r, int ldim, double *W,
    double *A );
static void reduce_ldlt_vector( int m, int n, int r, int *ord, int ldim,
    const double *L, const double *D, double *vec );
static void reduce_ldlt_matrix( int m, int n, int p, int diag, const int *ord,
    int ldim, const double *L, const double *D, const double *M,
    const double *T, double *A );
static void reduce_ldlt_matrix_blas( int m, int n, int p, int diag,
    const int *ord, int ldim, const double *L, const double *D, const double *M,
    const double *T, double *A );
static void ldlt_factor( char pivot, int m, int *n, int *piv, int *ord,
    int ldim, double *A, double *W );
```

```
static void ldlt_factor_blas( char pivot, int m, int *n, int *piv, int *ord,
    int ldim, double *A, double *W );
static void ldlt_block_rook_pivot( int blas, char pivot, int n,
    int *piv, int *ord, int ldim, double *A );
static void ldlt_block_comp_pivot( int blas, char pivot, int n,
    int *piv, int *ord, int ldim, double *A );

// Parameter for bounding element growth in trailing sub-matrix
static const double alpha = (1.0 + sqrt(17.0)) / 8.0;

#if defined(LDLTFACT) && defined(PROFILE)
    static int xtra_work = 0;
    static double tm_ldlt = 0.0;
    static double tm_factor = 0.0;
    static double tm_pivot = 0.0;
    static double tm_reduce = 0.0;
    static double tm_fact_piv = 0.0;
    static double tm_red_vec = 0.0;
#endif

/***************************************************************************/

/*
 * Determines optimal block dimension for the local environment given a routine
 * and matrix leading dimension.  The function returns different block
 * dimensions for simple blocking, blocking using BLAS, and the LAPACK symmetric
 * indefinite factorization routine.  Also, it facilitates the use of a
 * different block dimension for testing (debugging).  If the leading dimension
 * is less than the optimal block dimension, the block dimension is set to the
 * leading dimension, and the matrix computation becomes an unblocked algorithm.
 */
int get_block_dim_ldlt( int lapack, int blas, int ldim )
{
    const int   optm_bdim = 1;
    const int   no_dim = -1;
    const char  *parm_str = "L";
    const char  *func_name = "DSYTRF";

    int bdim;

#if defined(DEBUG)
    bdim = BDIM;
#else
    if ( lapack ) {
        bdim = ilaenv_( &optm_bdim, func_name, parm_str,
            &ldim, &no_dim, &no_dim, &no_dim );
    } else {
        if ( blas ) {
            bdim = 64;
        } else {
            bdim = 128;
        }
    }
#endif
    if ( bdim <= 1 || bdim > ldim ) {
        bdim = ldim;
    }
    return bdim;
}

/*
 * Counts the number of pivots performed -- row and column interchanges --
 * during matrix factorization.  piv_ord passed in the argument list determines
 * whether 1-by-1 pivots, 2-by-2 pivots, or both are counted.  piv[k] specifies
 * the permutation applied to row/ column k, so if piv[k] != k then a row/
 * column interchange is performed.
 */
int count_pivot( int piv_ord, int n, const int *piv, const int *ord )
{
    int count = 0;

    if ( piv_ord == 0 ) {                    // Count 1-by-1 and 2-by-2 pivots
        for (int k = 0; k < n; k++) {
            if ( k != piv[k] ) {
                count++;
            }
        }
    } else if ( piv_ord == 1 ) {             // Count 1-by-1 pivots
        for (int k = 0; k < n; k++) {
            if ( ord[k] == 1 && piv[k] != k ) {
                count++;
            }
        }
    } else {
        for (int k = 0; k < n; k++) {
            if ( ord[k] == 2 ) {             // Count 2-by-2 pivots
                count++;
            }
        }
    }
    return count;
}

/*
 * Performs diagonal pivot selection on an n-by-1 vector representing the
 * diagonal elements of an n-by-n matrix.  The maximum magnitude diagonal
 * element is chosen as the pivot.  A single pivot adjusted by diagonal offset
 * d, and its order (=1) are stored in vectors piv[] and ord[], respectively.
 * piv[k] specifies the permutation applied to row/ column k when performing
 * matrix factorization.
 */
void eval_pivot_diag( int n, int d, const double *diag, int *piv, int *ord )
{
    int     p = d;
    double  lambda = -1.0;

    for ( int i = 0; i < n; i++ ) {
        double x = fabs( diag[i] );
        if ( x > lambda ) {
            lambda = x;
            p = i + d;
        }
    }
    piv[d] = p;
    ord[d] = 1;
}

/*
 * Performs Bunch-Kaufman (partial) pivot selection on an n-by-n trailing
 * sub-matrix A with diagonal offset d.  The Bunch-Kaufman algorithm selects a
 * 1-by-1 or 2-by-2 pivot for symmetric indefinite factorization, L*D*L', where
 * D is block diagonal with block order 1 or 2.  The selected 1-by-1 or 2-by-2
 * pivot and its order are stored in vectors piv[] and ord[], respectively.
 * piv[k] (and piv[k+1]) specifies the permutation(s) applied to row(s)/
 * column(s) k (and k+1).  It is assumed that the matrix factorization is
 * implemented using the outer product method, such that trailing sub-matrix A
 * has been reduced (updated) prior to pivot selection.  Matrix A is stored in
 * column-major order with leading dimension ldim.
 */
void eval_pivot_bk( int n, int d, int ldim, const double *A,
    int *piv, int *ord )
{
    int r = 0;
    double lambda = -1.0;
    double sigma = -1.0;
    double a, arr;
    const double *A_r, *Ar_;

    // Determine largest magnitude off-diagonal entry in 1st row/ column
    for ( int i = 1; i < n; i++ ) {
        double x = fabs( *(A+i) );
        if ( x  > lambda ) {
            lambda = x;
            r = i;
        }
    }

    if ( lambda > 0.0 ) {
        a = fabs(*A);
        if ( a >= alpha * lambda ) {
            // Use 1st row/ column as 1-by-1 pivot
            piv[d] = d;
            ord[d] = 1;
        } else {
            A_r = A + r*ldim;
            Ar_ = A + r;
            arr = fabs ( *(A_r+r) );
            // Determine maximum magnitude off-diagonal entry in row/ column r.
            // Because of symmetry only entries on and below the diagonal have
            // been updated, so check entries in row r corresponding to entries
            // above the diagonal in column r and entries below the diagonal
            // in column r
            for ( int j = 0; j < r; j++ ) {
                double x = fabs( *(Ar_+j*ldim) );
                if ( x > sigma ) {
                    sigma = x;
                }
            }
            for ( int i = r+1; i < n; i++ ) {
                double x = fabs( *(A_r+i) );
                if ( x > sigma ) {
                    sigma = x;
                }
            }

            if ( a * sigma >= alpha * lambda * lambda ) {
                // Use 1st row/ column as 1-by-1 pivot
                piv[d] = d;
                ord[d] = 1;
            } else if ( arr >= alpha * sigma ) {
                // Use rth row/ column as 1-by-1 pivot
                piv[d] = d + r;
                ord[d] = 1;
            } else {
                // Use 1st and rth rows/ columns as 2-by-2 pivot
                piv[d] = d;
                piv[d+1] = d + r;
                ord[d] = 2;
                ord[d+1] = 0;
            }
        }
    } else {
        // Use 1st row/ column as 1-by-1 pivot
        piv[d] = d;
        ord[d] = 1;
    }
}

/*
 * Performs bounded Bunch-Kaufman (rook) pivot selection on an n-by-n trailing
 * sub-matrix A with diagonal offset d.  The bounded Bunch-Kaufman algorithm
 * selects a 1-by-1 or 2-by-2 pivot for symmetric indefinite factorization,
 * L*D*L', where D is block diagonal with block order 1 or 2.  The selected
 * 1-by-1 or 2-by-2 pivot and its order are stored in vectors piv[] and ord[],
 * respectively.  piv[k] (and piv[k+1]) specifies the permutation(s) applied to
 * row(s)/column(s) k (and k+1).  It is assumed that the matrix factorization
 * is implemented using the outer product method, such that trailing sub-matrix
 * A has been reduced (updated) prior to pivot selection.  Matrix A is stored in
 * column-major order with leading dimension ldim.
 */
void eval_pivot_bbk( int n, int d, int ldim, const double *A,
    int *piv, int *ord )
{
    const double tol = 100.0*DBL_EPSILON;

    int k = 0;
    int p = 0;
    int r = 0;
    double lambda = -1.0;
    double sigma = -1.0;
    double a, arr, eps;
    const double *A_r, *Ar_;

    // Determine largest magnitude off-diagonal entry in 1st row/column
    for ( int i = 1; i < n; i++ ) {
        double x = fabs( *(A+i) );
        if ( x  > lambda ) {
            lambda = x;
            r = i;
        }
    }

    if ( lambda > 0.0 ) {
        a = fabs(*A);
        if ( a >= alpha * lambda ) {
            // Use 1st row/column as 1-by-1 pivot
            piv[d] = d;
            ord[d] = 1;
        } else {
            int piv_slct = 0;            // Pivot selected = FALSE
            while ( piv_slct == 0 ) {    // Until pivot selected perform ...
                A_r = A + r*ldim;
                Ar_ = A + r;
                arr = fabs ( *(A_r+r) );
                // Determine maximum magnitude off-diagonal entry in row/
                // column r.  Because of symmetry only entries on and below the
                // diagonal have been updated, so check entries in row r
                // corresponding to entries above the diagonal in column r and
                // entries below the diagonal in column r
                for ( int j = 0; j < r; j++ ) {
                    double x = fabs( *(Ar_+j*ldim) );
                    if ( x > sigma ) {
                        sigma = x;
                        p = j;
                    }
                }
                for ( int i = r+1; i < n; i++ ) {
                    double x = fabs( *(A_r+i) );
                    if ( x > sigma ) {
                        sigma = x;
                        p = i;
                    }
                }

                // Calculate relative difference between lambda and sigma to
                // check whether they are equal within rounding error tolerance
                eps = fabs( lambda - sigma ) / sigma;
                if ( arr >= alpha * sigma ) {
                    // Use rth row/ column as 1-by-1 pivot
                    piv[d] = d + r;
                    ord[d] = 1;
                    piv_slct = 1;
                } else if ( eps < tol ) {
                    // Use kth and rth rows/ columns as 2-by-2 pivot
                    piv[d] = d + k;
                    piv[d+1] = d + r;
                    ord[d] = 2;
                    ord[d+1] = 0;
                    piv_slct = 1;
                } else {
                    // Continue search for pivot
                    k = r;
                    lambda = sigma;
                    r = p;
                    sigma = -1.0;
                }
            }
        }
    } else {
        // Use 1st row/ column as 1-by-1 pivot
        piv[d] = d;
        ord[d] = 1;
    }
}

/*
 * Performs Bunch-Parlett (complete) pivot selection on an n-by-n trailing
 * sub-matrix A with diagonal offset d. The Bunch-Parlett algorithm selects a
 * 1-by-1 or 2-by-2 pivot for symmetric indefinite factorization, L*D*L', where
 * D is block diagonal with block order 1 or 2. The selected 1-by-1 or 2-by-2
 * pivot and its order are stored in vectors piv[] and ord[], respectively.
 * piv[k] (and piv[k+1]) specifies the permutation(s) applied to row(s)/
 * column(s) k (and k+1).  It is assumed that the matrix factorization is
 * implemented using the outer product method, such that trailing sub-matrix A
 * has been reduced (updated) prior to pivot selection.  Matrix A is stored in
 * column-major order with leading dimension ldim.
 */
void eval_pivot_bp( int n, int d, int ldim, const double *A,
    int *piv, int *ord )
{
    int r = 0;
    int s = 0;
    int t = 0;
    double mu = -1.0;
    double nu = -1.0;
    const double *A_j;

    // Determine largest magnitude diagonal (nu) and off-diagonal (mu) entries
    // of trailing sub-matrix A, and their respective indexes.
    for ( int j = 0; j < n; j++ ) {
        A_j = A + j*ldim;
        double ajj = fabs( *(A_j+j) );
        if ( ajj > nu ) {
            nu = ajj;
            t = j;
        }
        for ( int i = j+1; i < n; i++ ) {
            double aij = fabs( *(A_j+i) );
            if ( aij > mu ) {
                mu = aij;
                r = i;
                s = j;
            }
        }
    }

    if ( mu > 0.0 || nu > 0.0 ) {
        if ( nu >= alpha * mu ) {
            // Use row/ column corresponding to maximum magnitude diagonal entry
            // as 1-x-1 pivot
            piv[d] = d + t;
            ord[d] = 1;
        } else {
            // Use rows/ columns corresponding to maximum magnitude off-diagonal
            // entry as 2-x-2 pivot
            piv[d] = d + s;
            piv[d+1] = d + r;
            ord[d] = 2;
            ord[d+1] = 0;
        }
    } else {
        // Use 1st row/ column as 1-by-1 pivot
        piv[d] = d;
        ord[d] = 1;
    }
}

/*
 * Performs Bunch-Kaufman (partial) pivot selection on an n-by-n trailing
 * sub-matrix A with diagonal offset d.  The Bunch-Kaufman algorithm selects a
 * 1-by-1 or 2-by-2 pivot for symmetric indefinite factorization, L*D*L', where
 * D is block diagonal with block order 1 or 2.  The selected 1-by-1 or 2-by-2
 * pivot and its order are stored in vectors piv[] and ord[], respectively.
 * piv[k] (and piv[k+1]) specifies the permutation(s) applied to row(s)/
 * column(s) k (and k+1).  It is assumed that the symmetric indefinite
 * factorization is an implementation of the SAXPY operation, such that trailing
 * sub-matrix A has yet to be reduced (updated) when pivot selection is
 * performed. Unit lower triangular matrix L and block diagonal matrix D are
 * used to reduce the rows/ columns of trailing sub-matrix A, which are
 * evaluated during pivot selection.  The reduced rows/ columns are stored in
 * matrix Q for reuse in the matrix factorization.  Matrices A, L and D are
 * stored in column-major order with leading dimension ldim.
 */
void eval_pivot_reduce_bk( int n, int d, int ldim, const double *L,
    const double *D, const double *A, double *Q, int *piv, int *ord )
{
    const double    *A_r, *Ar_;

    int          r = 0;
    double  lambda = -1.0;
    double  sigma = -1.0;
    double  q, qrr;
    double  *Q_r;

    // Copy 1st row/ column of A to Q and perform cumulative trailing
    // sub-matrix reduction on this vector
    for ( int i = 0; i < n; i++ ) {
        *(Q + i) = *(A + i);
    }
    reduce_ldlt_vector( n, d, r, ord, ldim, L, D, Q );

    // Determine largest magnitude off-diagonal entry in reduced 1st row/ column
    for ( int i = 1; i < n; i++ ) {
        double x = fabs( *(Q+i) );
        if ( x  > lambda ) {
            lambda = x;
            r = i;
        }
    }
    if ( lambda > 0.0 ) {
        q = fabs( *Q );
        if ( q >= alpha * lambda ) {
            // Use 1st row/ column as 1-by-1 pivot
            piv[d] = d;
            ord[d] = 1;
        } else {
            A_r = A + r*ldim;
            Ar_ = A + r;
            Q_r = Q + ldim;
            // Determine maximum magnitude off-diagonal entry in reduced row/
            // column r.  Because of symmetry only entries on and below the
            // diagonal have been updated/ pivoted, so check entries in row r
            // corresponding to entries above the diagonal in column r and
            // entries below the diagonal in column r
            for ( int j = 0; j < r; j++ ) {
                *(Q_r + j) = *(Ar_ + j*ldim);
            }
            memcpy( Q_r + r, A_r + r, (n-r)*sizeof(double) );
            reduce_ldlt_vector( n, d, r, ord, ldim, L, D, Q_r );
            qrr = fabs ( *(Q_r+r) );

            for ( int i = 0; i < n; i++ ) {
                double x = fabs( *(Q_r+i) );
                if ( x > sigma && i != r ) {
                    sigma = x;
                }
            }

            if ( q * sigma >= alpha * lambda * lambda ) {
                // Use 1st row/ column as 1-by-1 pivot
                piv[d] = d;
                ord[d] = 1;
#if defined(LDLTFACT) && defined(PROFILE)
    xtra_work++;
#endif
            } else if ( qrr >= alpha * sigma ) {
                // Use rth row/ column as 1-by-1 pivot, and copy reduced rth
                // row/ column to 1st column of Q
                piv[d] = d + r;
                ord[d] = 1;
                memcpy( Q, Q_r, n*sizeof(double) );
#if defined(LDLTFACT) && defined(PROFILE)
    xtra_work++;
#endif
            } else {
                // Use 1st and rth rows/ columns as 2-by-2 pivot
                piv[d] = d;
                piv[d+1] = d + r;
                ord[d] = 2;
                ord[d+1] = 0;
            }
        }
    } else {
        // Use 1st row/ column as 1-by-1 pivot
        piv[d] = d;
        ord[d] = 1;
    }
}

/*
 * Performs bounded Bunch-Kaufman (rook) pivot selection on an n-by-n trailing
 * sub-matrix A with diagonal offset d. The bounded Bunch-Kaufman algorithm
 * selects a 1-by-1 or 2-by-2 pivot for symmetric indefinite factorization,
 * L*D*L', where D is block diagonal with block order 1 or 2. The selected
 * 1-by-1 or 2-by-2 pivot and its order are stored in vectors piv[] and ord[],
 * respectively. piv[k] (and piv[k+1]) specifies the permutation(s) applied to
 * row(s)/ column(s) k (and k+1). It is assumed that the symmetric indefinite
 * factorization is an implementation of the SAXPY operation, such that trailing
 * sub-matrix A has yet to be reduced (updated) when pivot selection is
 * performed. Unit lower triangular matrix L and block diagonal matrix D are
 * used to reduce the rows/ columns of trailing sub-matrix A, which are
 * evaluated during pivot selection. The reduced rows/ columns are stored in
 * matrix Q for reuse in the matrix factorization. Matrices A, L and D are
 * stored in column-major order with leading dimension ldim.
 */
void eval_pivot_reduce_bbk( int n, int d, int ldim, const double *L,
    const double *D, const double *A, double *Q, int *piv, int *ord )
{
    const double    tol = 100.0*DBL_EPSILON;
    const double    *A_r, *Ar_;

    int         k = 0;
    int         p = 0;
    int         r = 0;
    double  lambda = -1.0;
    double  sigma = -1.0;
    double  eps, q, qrr;
    double  *Q_r;

    // Copy 1st row/ column of A to Q and perform cumulative trailing
    // sub-matrix reduction on this vector
    for ( int i = 0; i < n; i++ ) {
        *(Q + i) = *(A + i);
    }
    reduce_ldlt_vector( n, d, r, ord, ldim, L, D, Q );

    // Determine largest magnitude off-diagonal entry in 1st row/ column
    for ( int i = 1; i < n; i++ ) {
        double x = fabs( *(Q+i) );
        if ( x  > lambda ) {
            lambda = x;
            r = i;
        }
    }

    if ( lambda > 0.0 ) {
        q = fabs( *Q );
        if ( q >= alpha * lambda ) {
            // Use 1st row/ column as 1-by-1 pivot
            piv[d] = d;
            ord[d] = 1;
        } else {
            Q_r = Q + ldim;
            int piv_slct = 0;           // Pivot selected = FALSE
            while ( piv_slct == 0 ) {   // Until pivot selected perform ...
                A_r = A + r*ldim;
                Ar_ = A + r;
                // Determine maximum magnitude off-diagonal entry in reduced
                // row/ column r.  Because of symmetry only entries on and below
                // the diagonal have been updated/ pivoted, so check entries in
                // row r corresponding to entries above the diagonal in column r
                // and entries below the diagonal in column r
                for ( int j = 0; j < r; j++ ) {
                    *(Q_r + j) = *(Ar_ + j*ldim);
                }
                memcpy( Q_r + r, A_r + r, (n-r)*sizeof(double) );
                reduce_ldlt_vector( n, d, r, ord, ldim, L, D, Q_r );
                qrr = fabs( *(Q_r+r) );
                for ( int i = 0; i < n; i++ ) {
                    double x = fabs( *(Q_r+i) );
                    if ( x > sigma && i != r ) {
                        sigma = x;
                        p = i;
                    }
                }

                // Calculate relative difference between lambda and sigma to
                // check whether they are equal within rounding error tolerance
                eps = fabs( lambda - sigma ) / sigma;
                if ( qrr >= alpha * sigma ) {
                    // Use rth row/ column as 1-by-1 pivot
                    piv[d] = d + r;
                    ord[d] = 1;
                    memcpy( Q, Q_r, n*sizeof(double) );
                    piv_slct = 1;
#if defined(LDLTFACT) && defined(PROFILE)
    xtra_work++;
#endif
                } else if ( eps < tol ) {
                    // Use kth and rth rows/ columns as 2-by-2 pivot
                    piv[d] = d + k;
                    piv[d+1] = d + r;
                    ord[d] = 2;
                    ord[d+1] = 0;
                    piv_slct = 1;
                } else {
                    // Continue search for pivot
                    k = r;
                    lambda = sigma;
                    memcpy( Q, Q_r, n*sizeof(double) );
                    r = p;
                    sigma = -1.0;
#if defined(LDLTFACT) && defined(PROFILE)
    xtra_work++;
#endif
                }
            }
        }
    } else {
        // Use 1st row/ column as 1-by-1 pivot
        piv[d] = d;
        ord[d] = 1;
    }
}

/*
 * Performs Bunch-Kaufman (partial) pivot selection on an n-by-n trailing
 * sub-matrix A with diagonal offset d.  The Bunch-Kaufman algorithm selects a
 * 1-by-1 or 2-by-2 pivot for symmetric indefinite factorization, L*D*L', where
 * D is block diagonal with block order 1 or 2.  The selected 1-by-1 or 2-by-2
 * pivot and its order are stored in vectors piv[] and ord[], respectively.
 * piv[k] (and piv[k+1]) specifies the permutation(s) applied to row(s)/
 * column(s) k (and k+1).  It is assumed that the symmetric indefinite
 * factorization is an implementation of the SAXPY operation, such that trailing
 * sub-matrix A has yet to be reduced (updated) when pivot selection is
 * performed. BLAS routines are invoked to reduce the rows/ columns of trailing
 * sub-matrix A, which are evaluated during pivot selection.  The reduced rows/
 * columns are stored in matrix Q for reuse in the matrix factorization.
 * Matrices A, L and D are stored in column-major order with leading dimension
 * ldim.
 */
void eval_pivot_blas_bk( int n, int d, int ldim, const double *L,
    const double *M, const double *A, double *Q, int *piv, int *ord )
{
    const int        one = 1;
    const double         *A_r, *Ar_;

    int          r, t;
    double   lambda = -1.0;
    double  sigma = -1.0;
    double  q, qrr;
    double  *Q_r;


    // Copy 1st row/ column of A to Q and perform cumulative trailing
    // sub-matrix reduction on this vector
    for ( int i = 0; i < n; i++ ) {
        *(Q + i) = *(A + i);
    }
    r = 0;
    reduce_ldlt_vector_blas( n, d, r, ord, ldim, L, M, Q );

    // Determine largest magnitude off-diagonal entry in reduced 1st row/ column
    for ( int i = 1; i < n; i++ ) {
        double x = fabs( *(Q+i) );
        if ( x  > lambda ) {
            lambda = x;
            r = i;
        }
    }

    if ( lambda > 0.0 ) {
        q = fabs( *Q );
        if ( q >= alpha * lambda ) {
            // Use 1st row/ column as 1-by-1 pivot
            piv[d] = d;
            ord[d] = 1;
        } else {
            A_r = A + r*ldim;
            Ar_ = A + r;
            Q_r = Q + ldim;
            // Determine maximum magnitude off-diagonal entry in reduced row/
            // column r.  Because of symmetry only entries on and below the
            // diagonal have been updated/ pivoted, so check entries in row r
            // corresponding to entries above the diagonal in column r and
            // entries below the diagonal in column r
            const double     *Arr = A_r + r;
            double *Qrr = Q_r + r;
            t = n - r;
            dcopy_( &r, Ar_, &ldim, Q_r, &one );
            dcopy_( &t, Arr, &one, Qrr, &one );
            reduce_ldlt_vector_blas( n, d, r, ord, ldim, L, M, Q_r );
            qrr = fabs ( *(Q_r+r) );
            // Determine largest magnitude off-diagonal entry in reduced
            // rth row/ column
            for ( int i = 0; i < n; i++ ) {
                double x = fabs( *(Q_r+i) );
                if ( x > sigma && i != r ) {
                    sigma = x;
                }
            }

            if ( q * sigma >= alpha * lambda * lambda ) {
                // Use 1st row/ column as 1-by-1 pivot
                piv[d] = d;
                ord[d] = 1;
#if defined(LDLTFACT) && defined(PROFILE)
    xtra_work++;
#endif
            } else if ( qrr >= alpha * sigma ) {
                // Use rth row/ column as 1-by-1 pivot, and copy reduced rth
                // row/ column to 1st column of Q
                piv[d] = d + r;
                ord[d] = 1;
                dcopy_( &n, Q_r, &one, Q, &one );
#if defined(LDLTFACT) && defined(PROFILE)
    xtra_work++;
#endif
            } else {
                // Use 1st and rth rows/ columns as 2-by-2 pivot
                piv[d] = d;
                piv[d+1] = d + r;
                ord[d] = 2;
                ord[d+1] = 0;
            }
        }
    } else {
        // Use 1st row/ column as 1-by-1 pivot
        piv[d] = d;
        ord[d] = 1;
    }
}

/*
 * Performs bounded Bunch-Kaufman (rook) pivot selection on an n-by-n trailing
 * sub-matrix A with diagonal offset d.  The bounded Bunch-Kaufman algorithm
 * selects a 1-by-1 or 2-by-2 pivot for symmetric indefinite factorization,
 * L*D*L', where D is block diagonal with block order 1 or 2.  The selected
 * 1-by-1 or 2-by-2 pivot and its order are stored in vectors piv[] and ord[],
 * respectively.  piv[k] (and piv[k+1]) specifies the permutation(s) applied to
 * row(s)/ column(s) k (and k+1).  It is assumed that the symmetric indefinite
 * factorization is implemented using the SAXPY method, such that trailing
 * sub-matrix A has yet to be reduced (updated) when pivot selection is
 * performed. BLAS routines are invoked to reduce the rows/ columns of trailing
 * sub-matrix A, which are evaluated during pivot selection.  The reduced rows/
 * columns are stored in matrix Q for reuse in the matrix factorization.
 * Matrices A, L and D are stored in column-major order with leading dimension
 * ldim.
 */
```

```
void eval_pivot_blas_bbk( int n, int d, int ldim, const double *L,
    const double *M, const double *A, double *Q, int *piv, int *ord )
{
    const int         one = 1;
    const double          tol = 100.0*DBL_EPSILON;
    const double          *A_r, *Ar_;

    int       k, p, r, t;
    double  lambda = -1.0;
    double  sigma = -1.0;
    double  eps, q, qrr;
    double  *Q_r;

    // Copy 1st row/ column of A to Q and perform cumulative trailing
    // sub-matrix reduction on this vector
    for ( int i = 0; i < n; i++ ) {
        *(Q + i) = *(A + i);
    }
    r = 0;
    reduce_ldlt_vector_blas( n, d, r, ord, ldim, L, M, Q );

    // Determine largest magnitude off-diagonal entry in 1st row/ column
    for ( int i = 1; i < n; i++ ) {
        double x = fabs( *(Q+i) );
        if ( x  > lambda ) {
            lambda = x;
            r = i;
        }
    }

    k = 0;
    p = 0;
    if ( lambda > 0.0 ) {
        q = fabs( *Q );
        if ( q >= alpha * lambda ) {
            // Use 1st row/ column as 1-by-1 pivot
            piv[d] = d;
            ord[d] = 1;
        } else {
            Q_r = Q + ldim;
            int piv_slct = 0;           // Pivot selected = FALSE
            while ( piv_slct == 0 ) {   // Until pivot selected perform ...
                A_r = A + r*ldim;
                Ar_ = A + r;
                // Determine maximum magnitude off-diagonal entry in reduced
                // row/ column r.  Because of symmetry only entries on and below
                // the diagonal have been updated/ pivoted, so check entries in
                // row r corresponding to entries above the diagonal in column r
                // and entries below the diagonal in column r
                const double      *Arr = A_r + r;
                double *Qrr = Q_r + r;
                t = n - r;
                dcopy_( &r, Ar_, &ldim, Q_r, &one );
                dcopy_( &t, Arr, &one, Qrr, &one );
                reduce_ldlt_vector_blas( n, d, r, ord, ldim, L, M, Q_r );
                qrr = fabs( *(Q_r+r) );
                for ( int i = 0; i < n; i++ ) {
                    double x = fabs( *(Q_r+i) );
                    if ( x > sigma && i != r ) {
                        sigma = x;
                        p = i;
                    }
                }

                // Calculate relative difference between lambda and sigma to
                // check whether they are equal within rounding error tolerance
                eps = fabs( lambda - sigma ) / sigma;
                if ( qrr >= alpha * sigma ) {
                    // Use rth row/ column as 1-by-1 pivot
                    piv[d] = d + r;
                    ord[d] = 1;
                    dcopy_( &n, Q_r, &one, Q, &one );
                    piv_slct = 1;
#if defined(LDLTFACT) && defined(PROFILE)
    xtra_work++;
#endif
                } else if ( eps < tol ) {
                    // Use kth and rth rows/ columns as 2-by-2 pivot
                    piv[d] = d + k;
                    piv[d+1] = d + r;
                    ord[d] = 2;
                    ord[d+1] = 0;
                    piv_slct = 1;
                } else {
                    // Continue search for pivot
                    k = r;
                    lambda = sigma;
                    dcopy_( &n, Q_r, &one, Q, &one );
                    r = p;
                    sigma = -1.0;
#if defined(LDLTFACT) && defined(PROFILE)
    xtra_work++;
#endif
                }
            }
        }
    } else {
        // Use 1st row/ column as 1-by-1 pivot
        piv[d] = d;
        ord[d] = 1;
    }
}

/*
 * Performs pivoting of an n-by-n symmetric matrix A stored in column-major
 * with leading dimension ldim.  To preserve the symmetry of matrix A both row
 * and column k are interchanged with row and column r.  A^ = P*A*P', where P is
 * the permutation matrix and P' its transpose, is called a symmetric permutation
 * of A.  Because of symmetry only elements on and below the diagonal need be
 * interchanged.
 */
void pivot_sym( int n, int k, int r, int ldim, double *A )
{
    double *A_k = A + k*ldim;
    double *A_r = A + r*ldim;
    double *Ak_ = A + k;
    double *Ar_ = A + r;

    // Interchange elements A(k,0:k-1) with A(r,0:k-1)
    // i.e., elements of rows k and r to the left of column k
    for ( int j = 0; j < k; j++ ) {
        double akj = *(Ak_ + j*ldim);
        *(Ak_ + j*ldim) = *(Ar_ + j*ldim);
        *(Ar_ + j*ldim) = akj;
    }

    // Interchange diagonal elements of rows/ columns k and r
    double akk = *(A_k + k);
    *(A_k + k) = *(A_r + r);
    *(A_r + r) = akk;

    // Interchange elements A(k+1:r-1,k) with A(r,k+1:r-1)
    for ( int i = k+1; i < r; i++ ) {
        double aik = *(A_k + i);
        *(A_k + i) = *(Ar_ + i*ldim);
        *(Ar_ + i*ldim) = aik;
    }

    // Interchange elements A(r+1:n-1,k) with A(r+1:n-1,r)
    // i.e., elements of columns k and r below row r
    for ( int i = r+1; i < n; i++ ) {
        double aik = *(A_k + i);
        *(A_k + i) = *(A_r + i);
        *(A_r + i) = aik;
    }
}

/*
 * Performs pivoting of an n-by-n symmetric matrix A stored in column-major
 * order with leading dimension ldim.  To preserve the symmetry of matrix A both
 * row and column k are interchanged with row and column r.  A^ = P*A*P', where
 * P is the permutation matrix and P' its transpose, is called a symmetric
 * permutation of A.  The pivoting algorithm populates column k on and below the
 * diagonal of matrix A with row/ column r of the associated reduced trailing
 * sub-matrix stored in vector vec[].  Because of symmetry only elements on and
 * below the diagonal need be interchanged.
 */
void pivot_sym_reduce( int n, int k, int r, double *vec, int ldim, double *A )
{
    double *A_k = A + k*ldim;
    double *A_r = A + r*ldim;
    double *Ak_ = A + k;
    double *Ar_ = A + r;

    if ( k != r ) {
        // Interchange elements A(k,0:k-1) with A(r,0:k-1)
        // i.e., elements of rows k and r to the left of column k
        for ( int j = 0; j < k; j++ ) {
            double akj = *(Ak_ + j*ldim);
            *(Ak_ + j*ldim) = *(Ar_ + j*ldim);
            *(Ar_ + j*ldim) = akj;
        }

        // Interchange reduced diagonal elements vec[k] and vec[r], and
        // overwrite diagonal element A(r,r) with A(k,k)
        double vk = vec[k];
        vec[k] = vec[r];
        vec[r] = vk;
        *(A_r + r) = *(A_k + k);

        // Replace elements A(r,k+1:r-1) with A(k+1:r-1,k)
        for ( int i = k+1; i < r; i++ ) {
            *(Ar_ + i*ldim) = *(A_k + i);
        }

        // Replace elements A(r+1:n-1,r) with A(r+1:n-1,k)
        // i.e., elements below row r
        memcpy( A_r+r+1, A_k+r+1, (n-r-1)*sizeof(double) );
    }

    // Replace elements A(k:n-1,k) with reduced vector vec[k:n-1]
    memcpy( A_k + k, &vec[k], (n-k)*sizeof(double) );
}

/*
 * Performs pivoting of an n-by-n symmetric matrix A, and working array W, which
 * stores trailing sub-matrix updates applied to columns of A.  Arrays A and W
 * are stored in column-major order with leading dimension ldim.  To preserve
 * the symmetry of matrix A both row and column k are interchanged with row and
 * column r.  A^ = P*A*P', where P is the permutation matrix and P' its
 * transpose, is called a symmetric permutation of A. BLAS routines are invoked
 * to perform copy and swap operations that constitute symmetric pivoting.
 * Because of symmetry only elements on and below the diagonal need be
 * interchanged.
 */
void pivot_sym_blas( int n, int k, int r, int ldim, double *W, double *A )
{
    const int one = 1;

    int     t;
    double  *A_k = A + k*ldim;
    double  *A_r = A + r*ldim;
    double  *W_k = W + k*ldim;
    double  *Ak_ = A + k;
    double  *Ar_ = A + r;
    double  *Wk_ = W + k;
    double  *Wr_ = W + r;

    if ( k != r ) {
        // Interchange elements A(k,0:k-1) with A(r,0:k-1) and W(k,0:k-1) with
        // W(r,0:k-1), i.e., elements of rows k and r to the left of column k
        dswap_( &k, Ak_, &ldim, Ar_, &ldim );
        dswap_( &k, Wk_, &ldim, Wr_, &ldim );

        // Interchange reduced diagonal elements W(k,k) and W(r,k), and
        // overwrite diagonal element A(r,r) with A(k,k)
        double wkk = *(W_k + k);
        *(W_k + k) = *(W_k + r);
        *(W_k + r) = wkk;
        *(A_r + r) = *(A_k + k);

        // Replace elements A(r,k+1:r-1) with A(k+1:r-1,k)
        double  *Arl = Ar_ + k*ldim + ldim;
        double  *Alk = A_k + k + 1;
        t = r - k - 1;
        dcopy_( &t, Alk, &one, Arl, &ldim );

        // Replace elements A(r+1:n-1,r) with A(r+1:n-1,k)
        // i.e., elements below row r
        double  *Asr = A_r + r + 1;
        double  *Ask = A_k + r + 1;
        t = n - r - 1;
        dcopy_( &t, Ask, &one, Asr, &one );
    }
}

/*
 * Performs cumulative trailing sub-matrix updates (reduction) on row/ column r
 * of a symmetric indefinite matrix, which is stored in vector vec[] during the
 * factorization procedure. Suppose that P*A*P' = [A_00^, A_01^; A_10^, A_11^]
 * = [L_00, 0; L_10, L_11] * [D_00, 0; 0; D_11] * [L_00', L_10'; 0, L_11'].
 * Then the trailing sub-matrix A_11^^ = L_11*D_11*L_11' =
 * A_11^ - L_10*D_00*L_10', where vec[] is row/ column r of A_11^.
 * L is a pointer to m-by-n block L_10 of a unit lower triangular matrix, and
 * D is a pointer to n-by-n block D_00 of a block diagonal matrix with block
 * order 1 or 2 defined in ord[].  Matrices L and D are stored in column-major
 * order with leading dimension ldim.
 */
void reduce_ldlt_vector( int m, int n, int r, int *ord, int ldim,
    const double *L, const double *D, double *vec )
{
#if defined(LDLTFACT) && defined(PROFILE)
    struct  timespec sta_red_vec, end_red_vec;

    get_time( &sta_red_vec );
#endif

    const double *Lr_ = L + r;                          // Points to element L'(0,r)

    for ( int k = 0; k < n; ) {
        const double *L_k = L + k*ldim;                 // Points to element L(0,k)

        if ( ord[k] == 1 ) {                            // 1-by-1 pivot
            double dkk = *(D + k + k*ldim);             // Element D(k,k)
            double lrk = *(Lr_ + k*ldim);               // Element L'(k,r) = L(r,k)
            for ( int i = 0; i < m; i++ ) {
                vec[i] -= *(L_k + i) * dkk * lrk;
            }
            k++;
        } else {                                        // 2-by-2 pivot
            double d00 = *(D + k + k*ldim);             // Element D(k,k)
            double d10 = *(D + k + 1 + k*ldim);         // Element D(k+1,k) = D(k,k+1)
            double d11 = *(D + k + 1 + k*ldim + ldim);   // Element D(k+1,k+1)
            double lr0 = *(Lr_ + k*ldim);               // Element L'(k,r) = L(r,k)
            double lr1 = *(Lr_ + k*ldim + ldim);        // Element L'(k+1,r) = L(r,k+1)
            for ( int i = 0; i < m; i++ ) {
                vec[i] -=   *(L_k + i) * (d00*lr0 + d10*lr1) +
                            *(L_k + i + ldim) * (d10*lr0 + d11*lr1);
            }
            k += 2;
        }
    }
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_red_vec );
    tm_red_vec += timespec_diff( sta_red_vec, end_red_vec );
#endif
}

/*
 * Performs cumulative trailing sub-matrix updates (reduction) on row/ column r
 * of a symmetric indefinite matrix, which is stored in vector vec[] during the
 * factorization procedure.  Suppose that P*A*P' = [A_00^, A_01^; A_10^, A_11^]
 * = [L_00, 0; L_10, L_11] * [D_00, 0; 0; D_11] * [L_00', L_10'; 0, L_11'].
 * Then the trailing sub-matrix A_11^^ = L_11*D_11*L_11' =
 * A_11^ - L_10*D_00*L_10', where vec[] is row/ column r of A_11^.
 * L is a pointer to m-by-n block L_10 of a unit lower triangular matrix, and
 * M is a pointer to m-by-n working array that stores the product L_10*D_00.
 * Matrices L and M are stored in column-major order with leading dimension ldim.
 * BLAS routine DGMEMV performs the matrix-vector multiplication operation that
 * constitutes the trailing sub-matrix update of column r.
 */
void reduce_ldlt_vector_blas( int m, int n, int r, int *ord, int ldim,
    const double *L, const double *M, double *vec )
{
#if defined(LDLTFACT) && defined(PROFILE)
    struct  timespec sta_red_vec, end_red_vec;

    get_time( &sta_red_vec );
#endif

    const char      no_trans = 'N';
    const int       incx = ldim;
    const int       incy = 1;
    const double    _one = -1.0;
    const double    one = 1.0;

    const double *Lr_ = L + r;       // Points to element L(r,0) = L'(0,r)

    dgemv_( &no_trans, &m, &n, &_one, M, &ldim, Lr_, &incx, &one, vec, &incy );

#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_red_vec );
    tm_red_vec += timespec_diff( sta_red_vec, end_red_vec );
#endif
}

/*
 * Matrix factorization reduces trailing sub-matrix A by computing
 * A = A - L*D*T', where A is an m-by-n sub-matrix, L and T are m-by-p and
 * n-by-p blocks of unit lower triangular matrices, and D is a p-by-p block of a
 * block diagonal matrix with block order 1 or 2 -- vector ord[k] specifies the
 * diagonal block order.  A, L, D and T are stored in column-major order with
 * leading dimension ldim.  The trailing sub-matrix update is an implementation
 * of the SAXPY operation.  Because of symmetry, the trailing sub-matrix update
 * need only be performed on elements on or below the diagonal.
 */
void reduce_ldlt_matrix( int m, int n, int p, int diag, const int *ord,
    int ldim, const double *L, const double *D, const double *M,
    const double *T, double *A )
{
    for ( int j = 0; j < n; j++ ) {
        const double *Tj_ = T + j;                  // Points to element T'(0,j)
        double *A_j = A + j*ldim;                   // Points to element A(0,j)
        for ( int k = 0; k < p; ) {
            const double *L_k = L + k*ldim;         // Points to element L(0,k)

            if ( ord[k] == 1 ) {                    // 1-by-1 pivot
                double dkk = *(D + k + k*ldim);     // Element D(k,k)
                double tjk = *(Tj_ + k*ldim);       // Element T'(k,j) = T(j,k)
                for (int i = diag ? j : 0; i < m; i++) {
                    *(A_j + i) -= *(L_k + i) * dkk * tjk;
                }
                k++;
            } else {                                // 2-by-2 pivot
                double d00 = *(D + k + k*ldim);     // Element D(k,k)
                double d10 = *(D + k + 1 + k*ldim); // Element D(k+1,k) = D(k,k+1)
                double d11 = *(D + k + 1 + k*ldim + ldim);  // Element D(k+1,k+1)
                double tj0 = *(Tj_ + k*ldim);       // Element T'(k,j) = T(j,k)
                double tj1 = *(Tj_ + k*ldim + ldim);// Element T'(k+1,j) = T(j,k+1)
                for ( int i = diag ? j : 0; i < m; i++ ) {
                    *(A_j + i) -=   *(L_k + i) * (d00*tj0 + d10*tj1) +
                                    *(L_k + i + ldim) * (d10*tj0 + d11*tj1);
                }
                k += 2;
            }
        }
    }
}

/*
 * Matrix factorization reduces trailing sub-matrix A by computing
 * A = A - L*D*T', where A is an m-by-n sub-matrix, L and T are m-by-p and
 * n-by-p blocks of unit lower triangular matrices, and D is a p-by-p block of a
 * block diagonal matrix with block order 1 or 2 -- vector ord[k] specifies the
 * diagonal block order.  Matrix M stores the product L*D.  A, M and T are
 * stored in column-major order with leading dimension ldim.  BLAS routines
 * DGEMM and DGEMV perform the trailing sub-matrix update, which need only be
 * applied to elements on or below the diagonal because of symmetry.
 */
void reduce_ldlt_matrix_blas( int m, int n, int p, int diag, const int *ord,
    int ldim, const double *L, const double *D, const double *M,
    const double *T, double *A )
{
    const char      trans = 'T';
    const char      no_trans = 'N';
    const int       incx = ldim;
    const int       incy = 1;
    const double    _one = -1.0;
    const double    one = 1.0;
    // Compute A = A - L*D*T' = A - M*T'
    if ( diag ) {                                    // diagonal block, m = n
        for ( int j = 0; j < n; j++ ) {
            int r = n - j;
            const double *Mj_ = M + j;
            const double *Tj_ = T + j;
            double *Ajj = A + j + j*ldim;
            dgemv_( &no_trans, &r, &p, &_one, Mj_, &ldim, Tj_, &incx,
                &one, Ajj, &incy );
        }
    } else {                                         // rectangular block
        dgemm_( &no_trans, &trans, &m, &n, &p, &_one, M, &ldim, T, &ldim,
            &one, A, &ldim );
    }
}

/*
 * Matrix factorization reduces trailing sub-matrix A by computing
 * A = A - L*D*L', where A is an m-by-m sub-matrix, L is an m-by-n column block
 * of a unit lower triangular matrix and D is an n-by-n block of a block
 * diagonal matrix with block order 1 or 2 -- vector ord[k] specifies the
 * diagonal block order. For the implementation that uses the BLAS library,
 * matrix M stores the product L*D. Matrices A, L, D, M and T are stored in
 * column-major order with leading dimension ldim. Blocking is used to optimize
 * memory access for the trailing sub-matrix update, and bdim is the blocking
 * parameter. Because of symmetry, the trailing sub-matrix update need only be
 * performed on diagonal blocks and blocks below the diagonal.
 */
void reduce_ldlt_mat_blk( int blas, int m, int n, const int *ord, int bdim,
    int ldim, const double *L, const double *D, const double *M, double *A )
{
    void    (*reduce_matrix)( int m, int n, int p, int diag, const int *ord,
                int ldim, const double *L, const double *D, const double *M,
                const double *T, double *A );

    if ( blas ) {
        reduce_matrix = reduce_ldlt_matrix_blas;
    } else {
        reduce_matrix = reduce_ldlt_matrix;
    }

    for ( int j = 0; j < m; j += bdim ) {
        // Determine number of columns in (i,j)th block of A
        const int s = (j + bdim > m) ? (m - j) : bdim;

        for ( int k = 0; k < n; k += bdim ) {
            int diag = 1;         // Diagonal block = TRUE
            // Determine number of columns of Lik, rows of L'kj,
            // and dimension of square matrix block Dkk
            const int t = (k + bdim > n) ? (n - k) : bdim;
            // Set pointer to matrix blocks Dkk and L'kj.  Pointer to L'kj also
            // points to Ljk -- referred to as Tjk to differentiate from Lik
            const double *Dkk = D + k + k*ldim;
            const double *Tjk = L + j + k*ldim;

            for ( int i = j; i < m; i += bdim ) {
                // Determine number of rows in (i,j)th block of A
                const int r = (i + bdim > m) ? (m - i) : bdim;
                // Set pointers to block matrices Lik, Mik = Lik*Dkk, and Aij
                const double *Mik = M + i + k*ldim;
                const double *Lik = L + i + k*ldim;
                double *Aij = A + i + j*ldim;
                // Reduce trailing block matrix
                reduce_matrix( r, s, t, diag, &ord[k],
                    ldim, Lik, Dkk, Mik, Tjk, Aij );
                diag = 0;       // Diagonal block = FALSE
            }
        }
    }
}

/*
 * Implements a rectangular version of the SAXPY operation (jki indexing) for
 * symmetric indefinite factorization.  Symmetric indefinite m-by-n matrix A
 * with leading dimension ldim is factored into a unit lower triangular matrix L
 * and block diagonal matrix D with block order 1 or 2.  Symmetrically permuted
 * matrix A^ = P*A*P' = L*D*L', where P is the permutation matrix, and L' and P'
 * are the transpose of L and P, respectively.  Permutation matrix P is encoded
 * in vectors piv[] and ord[], such that row/ column k is interchanged with
 * row/ column piv[k], and ord[k] specifies the diagonal block order.  Entries
 * of L and D are stored on and below the diagonal of matrix A, i.e., L and D
 * overwrite A.
 */
void ldlt_factor( char pivot, int m, int *n, int *piv, int *ord, int ldim,
    double *A, double *W )
{
#if defined(LDLTFACT) && defined(PROFILE)
    struct  timespec sta_fact_piv, end_fact_piv;
#endif

    int     j = 0;
    double  *D = A;

    for ( ; j < *n; ) {
        double *L = A + j;
        double *A_j = A + j*ldim;
        double *Ajj = A_j + j;
        // Evaluate pivot using to method specified in argument list
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_fact_piv );
```

```
#endif
        switch ( pivot ) {
        case 'B':
            eval_pivot_reduce_bbk( m-j, j, ldim, L, D, Ajj, W+j, piv, ord );
            break;
        case 'K':
            eval_pivot_reduce_bk( m-j, j, ldim, L, D, Ajj, W+j, piv, ord );
            break;
        default:
            eval_pivot_reduce_bk( m-j, j, ldim, L, D, Ajj, W+j, piv, ord );
            break;
        }
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_fact_piv );
    tm_fact_piv += timespec_diff( sta_fact_piv, end_fact_piv );
#endif
        // Perform symmetric pivoting using reduced trailing sub-matrix row(s)/
        // column(s) returned by pivot selection algorithm, and compute column(s)
        // of unit lower triangular matrix L -- because of symmetry need only
        // update elements on and below the diagonal.  Details of these
        // computations differ depending on whether the diagonal block (pivot)
        // is 1-by-1 or 2-by-2
        if ( ord[j] == 1 ) {                    // 1-x-1 pivot
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_fact_piv );
#endif
            pivot_sym_reduce( m, j, piv[j], W, ldim, A );
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_fact_piv );
    tm_fact_piv += timespec_diff( sta_fact_piv, end_fact_piv );
#endif
            double ajj = *Ajj;
            for ( int i = j+1; i < m; i++ ) {
                *(A_j + i) /= ajj;
            }
            j++;

        } else {                                // 2-x-2 pivot, ord[k] == 2
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_fact_piv );
#endif
            // Apply first of two pivots to matrix A using first column of W
            pivot_sym_reduce( m, j, piv[j], W, ldim, A );
            // Apply first of two pivots to second column of W
            *(W + ldim + piv[j]) = *(W + ldim + j);
            // Apply second of two pivots to matrix A using second column of W
            pivot_sym_reduce( m, j+1, piv[j+1], W+ldim, ldim, A );
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_fact_piv );
    tm_fact_piv += timespec_diff( sta_fact_piv, end_fact_piv );
#endif
            // Let A(k:n-1,k:k+1) = [D, C'; C, A^] =
// [I, 0; C*inv(D), I] * [D, 0; 0, A^ - C*inv(D)*C'] * [I, 0; C*inv(D), I]
            // where D is 2-by-2 symmetric diagonal block and inv(D) its inverse.
            // First solve for (n-k-2)-by-2 unit lower triangular block, then
            // reduce trailing sub-matrix by computing A^ - C*inv(D)*C'.  Once
            // computed, L and D overwrite A
            double d00 = *Ajj;                  // Element D(j,j)
            double d10 = *(Ajj + 1);            // Element D(j+1,j) = D(j,j+1)
            double d11 = *(Ajj + 1 + ldim);     // Element D(j+1,j+1)
            double denom = d00 * d11 - d10 * d10;
            for ( int i = j+2; i < m; i++ ) {
                double aij = *(A_j + i);
                double aik = *(A_j + i + ldim);
                *(A_j+i) = ( aij * d11 - aik * d10 ) / denom;
                *(A_j+i+ldim) = ( aik * d00 - aij * d10 ) / denom;
            }
            j += 2;
        }
    }
    *n = j;
}

/*
 * Implements a rectangular version of the SAXPY operation (jki indexing) for
 * symmetric indefinite factorization.  Symmetric indefinite m-by-n matrix A
 * with leading dimension ldim is factored into a unit lower triangular matrix L
 * and block diagonal matrix D with block order 1 or 2.  Symmetrically permuted
 * matrix A^ = P*A*P' = L*D*L', where P is the permutation matrix, and L' and P'
 * are the transpose of L and P, respectively.  Permutation matrix P is encoded
 * in vectors piv[] and ord[], such that row/ column k is interchanged with
 * row/ column piv[k], and ord[k] specifies the diagonal block order.  To the
 * extent possible, this implementation of the SAXPY operation uses the BLAS
 * library to perform matrix operations.  Entries of L and D are stored on and
 * below the diagonal of matrix A, i.e., L and D overwrite A.
 */
void ldlt_factor_blas( char pivot, int m, int *n, int *piv, int *ord, int ldim,
    double *A, double *W )
{
#if defined(LDLTFACT) && defined(PROFILE)
    struct  timespec sta_fact_piv, end_fact_piv;
#endif

    int     j = 0;

    for ( ; j < *n; ) {
        double *L = A + j;          // A stores unit lower triangular matrix L
        double *M = W + j;          // W stores L*D
        double *W_j = W + j*ldim;
        double *Wjj = W_j + j;
        double *A_j = A + j*ldim;
        double *Ajj = A_j + j;

        // Evaluate pivot using to method specified in argument list
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_fact_piv );
#endif
        switch ( pivot ) {
        case 'B':
            eval_pivot_blas_bbk( m-j, j, ldim, L, M, Ajj, Wjj, piv, ord );
            break;
        case 'K':
            eval_pivot_blas_bk( m-j, j, ldim, L, M, Ajj, Wjj, piv, ord );
            break;
        default:
            eval_pivot_blas_bk( m-j, j, ldim, L, M, Ajj, Wjj, piv, ord );
            break;
        }
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_fact_piv );
    tm_fact_piv += timespec_diff( sta_fact_piv, end_fact_piv );
#endif
        // Perform symmetric pivoting using reduced trailing sub-matrix row(s)/
        // column(s) returned by pivot selection algorithm, and compute column(s)
        // of unit lower triangular matrix L -- because of symmetry need only
        // update elements on and below the diagonal.  Details of these
        // computations differ depending on whether the diagonal block (pivot)
        // is 1-by-1 or 2-by-2
        if ( ord[j] == 1 ) {          // 1-x-1 pivot
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_fact_piv );
#endif
            pivot_sym_blas( m, j, piv[j], ldim, W, A );
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_fact_piv );
    tm_fact_piv += timespec_diff( sta_fact_piv, end_fact_piv );
#endif
            double ajj = *Wjj;
            *Ajj = ajj;
            for ( int i = j+1; i < m; i++ ) {
                *(A_j + i)  = *(W_j + i) / ajj;
            }
            j++;

        } else {                      // 2-x-2 pivot, ord[k] == 2
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_fact_piv );
#endif
            // Apply first of two pivots to matrix A using first column of W
            pivot_sym_blas( m, j, piv[j], ldim, W, A );
```

```
          // Apply first of two pivots to second column of W
          *(W_j + ldim + piv[j]) = *(W_j + ldim + j);
          // Apply second of two pivots to matrix A using second column of W
          pivot_sym_blas( m, j+1, piv[j+1], ldim, W, A );
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_fact_piv );
    tm_fact_piv += timespec_diff( sta_fact_piv, end_fact_piv );
#endif
          // Solve for L(j+2:n-1,j:j+1) * D(j:j+1,j:j+1) = A(j+2:n-1,j:j+1)
          // where columns of updated trailing sub-matrix A and diagonal block
          // are stored in W.  Once computed, L and D overwrite A
          double d00 = *Wjj;                    // Element D(j,j)
          double d10 = *(Wjj + 1);              // Element D(j+1,j) = D(j,j+1)
          double d11 = *(Wjj + 1 + ldim);       // Element D(j+1,j+1)
          double denom = d00 * d11 - d10 * d10;
          *Ajj = d00;
          *(Ajj + 1) = d10;
          *(Ajj + 1 + ldim) = d11;
          for ( int i = j+2; i < m; i++ ) {
              *(A_j+i) = ( *(W_j+i) * d11 - *(W_j+i+ldim) * d10 ) / denom;
              *(A_j+i+ldim) = ( *(W_j+i+ldim) * d00 - *(W_j+i) * d10 ) / denom;
          }
          j += 2;
      }
    }
    *n = j;
}

/*
 * Implements simple blocking with partial or rook pivoting to factorize
 * symmetric indefinite n-by-n matrix A (with leading dimension ldim) into a unit
 * lower triangular matrix L and block diagonal matrix D with block order 1 or 2.
 * Symmetrically permuted matrix A^ = P*A*P' = L*D*L', where P is the
 * permutation matrix, and L' and P' are the transpose of L and P, respectively.
 * Permutation matrix P is encoded in vectors piv[] and ord[], such that row/
 * column k is interchanged with row/ column piv[k], and ord[k] specifies the
 * diagonal block order.  Suppose that A is decomposed into blocks
 * [A_00, A_01; A_10, A_11], where A_00 is an r-by-r block matrix.  First, a
 * rectangular version of the SAXPY operation for symmetric indefinite
 * factorization computes
 * P * [A_00, A_01; A_10, A_11] * P' = [A_00^, A_01^; A_10^, A_11^]
 * = [L_00, 0; L_10, L_11] * [D_00, 0; 0; D_11] * [L_00', L_10'; 0, L_11']
 * = [L_00*D_00*L_00', (L_10*D_00*L00')';
 *     L_10*D_00*L00', L_10*D_00*L_10' + L_11*D_11*L11'].
 * This computation yields the LDL' factorization for the first n-by-r column
 * block of A.  Then the trailing submatrix is updated to give
 * A_11^^ = L_11*D_11*L11' = A_11^ - L_10*D_00*L_10'.
 * This procedure is repeated iteratively on the trailing sub-matrix until the
 * last diagonal block (dimension <= r) is reached.  Simple blocking is also
 * used to optimize memory access when updating the trailing sub-matrix.
 */
void ldlt_block_rook_pivot( int blas, char pivot, int n, int *piv, int *ord,
    int ldim, double *A )
{
    const int lapack = 0;
    const int bdim = get_block_dim_ldlt( lapack, blas, ldim );

    int     d, j, r, t;
    double  *Ajj, *L, *D, *W;
    void    (*ldlt)( char pivot, int m, int *n, int *piv, int *ord, int ldim,
                double *A, double *W );

#if defined(LDLTFACT) && defined(PROFILE)
    struct  timespec sta_ldlt, sta_factor, sta_pivot, sta_reduce,
            end_ldlt, end_factor, end_pivot, end_reduce;
    xtra_work = 0;
    tm_ldlt = 0.0;
    tm_factor = 0.0;
    tm_pivot = 0.0;
    tm_reduce = 0.0;
    tm_fact_piv = 0.0;
    tm_red_vec = 0.0;

    get_time(&sta_ldlt);
#endif

    if ( blas ) {
        ldlt = ldlt_factor_blas;
        W = (double *) malloc( ldim*(bdim+1)*sizeof(double) );
    } else {
        ldlt = ldlt_factor;
        W = (double *) malloc( ldim*2*sizeof(double) );
    }

    j = 0;
    r = (bdim > n) ? n : bdim;
    // Perform rectangular factorization on first column block A(0:n-1,0:r)
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_factor );
#endif
    ldlt( pivot, n, &r, &piv[j], &ord[j], ldim, A, W );
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_factor );
    tm_factor += timespec_diff( sta_factor, end_factor );
#endif

    d = 0;
    j = r;
    t = n - r;
    Ajj = A;
    for ( ; j < n; j += r, t -= r ) {
        // Adjust pivot vector of previous block for diagonal offset
        for (int i = d; i < j; i++) {
            piv[i] += d;
        }
        L = Ajj + r;
        D = Ajj;
        Ajj = A + j + j*ldim;
        // Reduce trailing sub-matrix, P * A(j:n-1,j:n-1) * P'
        // -= L(j:n-1,j-r:j-1) * D(j-r:j-1,j-r:j-1) * L'(j-r:j-1,j:n-1)
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_reduce );
#endif
        reduce_ldlt_mat_blk( blas, t, r, &ord[d], r, ldim, L, D, W+j, Ajj );
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_reduce );
    tm_reduce += timespec_diff( sta_reduce, end_reduce );
#endif
        d += r;
        r = t < bdim ? t : bdim;
        // Perform rectangular factorization on column block A(j:n-1,j:j+r-1)
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_factor );
#endif
        ldlt( pivot, t, &r, &piv[j], &ord[j], ldim, Ajj, W+j );
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_factor );
    tm_factor += timespec_diff( sta_factor, end_factor );
#endif
        // Apply permutation matrix for current block, encoded in piv(j:j+r-1),
        // to columns to the left of current block A(:,0:j-1)
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_pivot );
#endif
        for ( int i = j; i < j+r; i++ ) {
            if ( i != piv[i] + j ) {
                if ( blas ) {
                    double *Ai_ = A + i;
                    double *Ar_ = A + piv[i] + j;
                    dswap_( &j, Ai_, &ldim, Ar_, &ldim );
                } else {
                    for ( int k = 0; k < j; k++ ) {
                        double aik = *(A + i + k*ldim);
                        *(A + i + k*ldim) = *(A + piv[i] + j + k*ldim);
                        *(A + piv[i] + j + k*ldim) = aik;
                    }
                }
            }
        }
#if defined(LDLTFACT) && defined(PROFILE)
```

```
    get_time( &end_pivot );
    tm_pivot += timespec_diff( sta_pivot, end_pivot );
#endif
    }
    // Adjust pivot vector of last block for diagonal offset
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &sta_pivot );
#endif
    for ( int i = d; i < n; i++ ) {
        piv[i] += d;
    }
#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_pivot );
    tm_pivot += timespec_diff( sta_pivot, end_pivot );
#endif
    free( W );

#if defined(LDLTFACT) && defined(PROFILE)
    get_time( &end_ldlt );
    tm_ldlt += timespec_diff( sta_ldlt, end_ldlt );

    int num_piv = count_pivot( 0, n, piv, ord );
    double frac_fact = num_piv / (double) (num_piv + xtra_work);
    tm_pivot = tm_pivot + tm_fact_piv - frac_fact * tm_red_vec;
    tm_factor = tm_factor - tm_fact_piv + frac_fact * tm_red_vec;
    fprintf( stdout, "%.3f\t%.3f\t\t%.3f\t\t%.3f\t\t%.1f\t\t%.1f\t\t%.1f\n",
        tm_ldlt, tm_factor, tm_pivot, tm_reduce,
        tm_factor/tm_ldlt*100, tm_pivot/tm_ldlt*100, tm_reduce/tm_ldlt*100 );
#endif
}

/*
 * Implements simple blocking with complete pivoting to factorize symmetric
 * indefinite n-by-n matrix A (with leading dimension ldim) into a unit lower
 * triangular matrix L and block diagonal matrix D with block order 1 or 2.
 * Symmetrically permuted matrix A^ = P*A*P' = L*D*L', where P is the
 * permutation matrix, and L' and P' are the transpose of L and P, respectively.
 * Permutation matrix P is encoded in vectors piv[] and ord[], such that row/
 * column k is interchanged with row/ column piv[k], and ord[k] specifies the
 * diagonal block order.  Each pass through the k-loop selects a 1-by-1 or
 * 2-by-2 pivot, computes column(s) of the unit lower triangular matrix below
 * the pivot, and performs an outer product operation to reduce the trailing
 * sub-matrix.  In order to implement complete pivoting the entire trailing
 * sub-matrix must be updated prior to each pivot selection, i.e., prior to each
 * pass through the k-loop.  Hence, the blocking procedure is applied to the
 * trailing sub-matrix reduction alone.
 */
void ldlt_block_comp_pivot( int blas, char pivot, int n, int *piv, int *ord,
    int ldim, double *A )
{
```

```
const int   lapack = 0;
const int   bdim = get_block_dim_ldlt( lapack, blas, ldim );

double  *A_k, *Akk, *L, *D, *M, *W;

W = (double *) malloc( ldim*2*sizeof(double) );
A_k = A;
Akk = A;
for ( int k = 0; k < n-1; ) {

    // Evaluate pivot using to method specified in argument list
    switch ( pivot ) {
    case 'P':
        eval_pivot_bp( n-k, k, ldim, Akk, piv, ord );
        break;
    default:
        eval_pivot_bp( n-k, k, ldim, Akk, piv, ord );
        break;
    }

    // Perform symmetric pivoting, compute column(s) of unit lower
    // triangular matrix L, and update trailing sub-matrix -- because of
    // symmetry need only update elements on and below the diagonal.
    // Details of these computations differ depending on whether the
    // diagonal block (pivot) is 1-by-1 or 2-by-2
    if ( ord[k] == 1 ) {          // 1-x-1 pivot
        if ( k != piv[k] ) {
            pivot_sym( n, k, piv[k], ldim, A );
        }
        // Store A(k+1:n-1,k), a column vector of LD, in working array W
        memcpy( W+k+1, Akk+1, (n-k-1)*sizeof(double) );
        double akk = *Akk;
        for ( int i = 1; i < n-k; i++ ) {
            *(Akk + i)  /= akk;
        }
    } else {                      // 2-x-2 pivot, ord[k] == 2
        if ( k != piv[k] ) {
            pivot_sym( n, k, piv[k], ldim, A );
        }
        if ( k+1 != piv[k+1] ) {
            pivot_sym( n, k+1, piv[k+1], ldim, A );
        }
        // Store A(k+2:n-1,k:k+1), a column block of LD, in working array W
        memcpy( W+k+2, Akk+2, (n-k-2)*sizeof(double) );
        memcpy( W+k+2+ldim, Akk+2+ldim, (n-k-2)*sizeof(double) );
        // Solve for L(k+2:n-1,k:k+1) * D(k:k+1,k:k+1) = A(k+2:n-1,k:k+1)
        // Once computed, L and D overwrite A
        double d00 = *Akk;                // Element D(k,k)
        double d10 = *(Akk + 1);          // Element D(k+1,k) = D(k,k+1)
        double d11 = *(Akk + 1 + ldim);   // Element D(k+1,k+1)
```

```
            double denom = d00 * d11 - d10 * d10;
            for ( int i = k+2; i < n; i++ ) {
                *(A_k+i) = ( *(W+i) * d11 - *(W+i+ldim) * d10 ) / denom;
                *(A_k+i+ldim) = ( *(W+i+ldim) * d00 - *(W+i) * d10 ) / denom;
            }
        }
        L = Akk + ord[k];
        D = Akk;
        M = W + k + ord[k];
        A_k = A_k + ord[k]*ldim;
        Akk = A_k + k + ord[k];
        // Reduce trailing sub-matrix, P * A(k+ord[k]:n-1,k+ord[k]:n-1) * P' -=
        // L(k+ord[k]:n-1,k:k+ord[k]) * D(k:k+ord[k],k:k+ord[k])
        // * L'(k:k+ord[k]:k+ord[k]:n-1)
        reduce_ldlt_mat_blk( blas, n-k-ord[k], ord[k], &ord[k], bdim,
            ldim, L, D, M, Akk );
        k += ord[k];
    }
    if ( ord[n-2] != 2 ) {          // Last pivot is 1-by-1
        piv[n-1] = n-1;
        ord[n-1] = 1;
    }
    free( W );
}

/****************************************************************************/

/*
 * Implements the outer product method (kji indexing) to factorize symmetric
 * indefinite n-by-n matrix A into a unit lower triangular matrix L and block
 * diagonal matrix D with block order 1 or 2.  Symmetrically permuted matrix
 * A^ = P*A*P' = L*D*L', where P is the permutation matrix, and L' and P' are
 * the transpose of L and P, respectively.  The permutation matrix P is encoded
 * in vectors piv[] and ord[], such that row/ column k is interchanged with
 * row/ column piv[k], and ord[k] specifies the diagonal block order.  Entries
 * of L and D are stored on and below the diagonal of matrix A, i.e., L and D
 * overwrite A.  Each pass through the k-loop performs an outer product
 * operation.
 */
void ldlt_outer_product( char pivot, int n, int *piv, int *ord, double *A )
{
    const int ldim = n;

    double *W;

    W = (double *) malloc( ldim*2*sizeof(double) );

    for (int k = 0; k < n-1; ) {

        double *A_k = A + k*ldim;
        double *Akk = A_k + k;
        // Evaluate pivot using to method specified in argument list
        switch ( pivot ) {
        case 'B':
            eval_pivot_bbk( n-k, k, ldim, Akk, piv, ord );
            break;
        case 'K':
            eval_pivot_bk( n-k, k, ldim, Akk, piv, ord );
            break;
        case 'P':
            eval_pivot_bp( n-k, k, ldim, Akk, piv, ord );
            break;
        default:
            eval_pivot_bbk( n-k, k, ldim, Akk, piv, ord );
            break;
        }

        // Perform symmetric pivoting, compute column(s) of unit lower
        // triangular matrix L, and update trailing sub-matrix -- because of
        // symmetry need only update elements on and below the diagonal.
        // Details of these computations differ depending on whether the
        // diagonal block (pivot) is 1-by-1 or 2-by-2
        if ( ord[k] == 1 ) {          // 1-x-1 pivot
            if ( k != piv[k] ) {
                pivot_sym( n, k, piv[k], ldim, A );
            }
            double akk = *Akk;
            for ( int i = k+1; i < n; i++ ) {
                *(A_k + i)  /= akk;
            }
            for ( int j = k+1; j < n; j++ ) {
                double *A_j = A + j*ldim;
                double ajk = *(A_k + j);
                for ( int i = j; i < n; i++ ) {
                    *(A_j+i) -= *(A_k+i) * ajk * akk;
                }
            }
            k++;

        } else {                       // 2-x-2 pivot, ord[k] == 2
            if ( k != piv[k] ) {
                pivot_sym( n, k, piv[k], ldim, A );
            }
            if ( k+1 != piv[k+1] ) {
                pivot_sym( n, k+1, piv[k+1], ldim, A );
            }
            // Let A(k:n-1,k:k+1) = [D, C'; C, A^] =
// [I, 0; C*inv(D), I] * [D, 0; 0, A^ - C*inv(D)*C'] * [I, 0; C*inv(D), I]
            // where D is 2-by-2 symmetric diagonal block and inv(D) its inverse.
            // First solve for (n-k-2)-by-2 unit lower triangular block, then
            // reduce trailing sub-matrix by computing A^ - C*inv(D)*C'. Once
            // computed, L and D overwrite A.  Store A(k+2:n-1,k:k+1) in W
            memcpy( W+k+2, Akk+2, (n-k-2)*sizeof(double) );
            memcpy( W+k+2+ldim, Akk+2+ldim, (n-k-2)*sizeof(double) );
            // Solve for L(k+2:n-1,k:k+1) * D(k:k+1,k:k+1) = A(k+2:n-1,k:k+1)
            // Once computed, L and D overwrite A
            double d00 = *Akk;                    // Element D(k,k)
            double d10 = *(Akk + 1);              // Element D(k+1,k) = D(k,k+1)
            double d11 = *(Akk + 1 + ldim);       // Element D(k+1,k+1)
            double denom = d00 * d11 - d10 * d10;
            for ( int i = k+2; i < n; i++ ) {
                *(A_k+i) = ( *(W+i) * d11 - *(W+i+ldim) * d10 ) / denom;
                *(A_k+i+ldim) = ( *(W+i+ldim) * d00 - *(W+i) * d10 ) / denom;
            }
            // Reduce trailing sub-matrix, A(k+2:n-1,k+2,n-1) =
            // A(k+2:n-1,k+2,n-1) - L(k+2:n-1,k:k+1) * W(k+2:n-1,k:k+1)'
            double *L;
            double wjk;
            for ( int j = k+2; j < n; j++ ) {
                double *A_j = A + j*ldim;
                L = A + k*ldim;
                wjk = *(W+j);
                for (int i = j; i < n; i++) {
                    *(A_j + i) -= *(L + i) * wjk;
                }
                L = A + k*ldim + ldim;
                wjk = *(W+j+ldim);
                for (int i = j; i < n; i++) {
                    *(A_j + i) -= *(L + i) * wjk;
                }
            }
            k += 2;
        }
    }
    if ( ord[n-2] != 2 ) {            // Last pivot is 1-by-1
        piv[n-1] = n-1;
        ord[n-1] = 1;
    }
    free( W );
}

/*
 * Implements the SAXPY operation (jki indexing) to factorize symmetric
 * indefinite n-by-n matrix A into a unit lower triangular matrix L and block
 * diagonal matrix D with block order 1 or 2.  Symmetrically permuted matrix
 * A^ = P*A*P' = L*D*L', where P is the permutation matrix, and L' and P' are
 * the transpose of L and P, respectively.  Permutation matrix P is encoded in
 * vectors piv[] and ord[], such that row/ column k is interchanged with row/
 * column piv[k], and ord[k] specifies the diagonal block order.  Entries of L
 * and D are stored on and below the  diagonal of matrix A, i.e., L and D
 * overwrite A.
 */
void ldlt_saxpy( char pivot, int n, int *piv, int *ord, double *A )
{
    const int ldim = n;

    double  *W;

    W = (double *) malloc( ldim*2*sizeof(double) );

    ldlt_factor( pivot, n, &n, piv, ord, ldim, A, W );

    free( W );
}

/*
 * Implements simple blocking to factorize symmetric indefinite n-by-n matrix A
 * into a unit lower triangular matrix L and block diagonal matrix D with block
 * order 1 or 2.  Symmetrically permuted matrix A^ = P*A*P' = L*D*L', where P
 * is the permutation matrix, and L' and P' are the transpose of L and P,
 * respectively.  Permutation matrix P is encoded in vectors piv[] and ord[],
 * such that row/ column k is interchanged with row/ column piv[k], and ord[k]
 * specifies the diagonal block order.  In the case where a partial or rook
 * pivoting strategy is specified the blocked algorithm employs the SAXPY
 * operation to perform symmetric indefinite factorization -- cumulative
 * trailing sub-matrix updates are applied to candidate pivot row(s)/ column(s)
 * as the factorization proceeds.  When a complete pivoting strategy is
 * specified the entire trailing sub-matrix must be updated prior to each pivot
 * selection, so the blocked algorithm employs the outer product method.
 */
void ldlt_block( char pivot, int n, int *piv, int *ord, double *A )
{
    const int   ldim = n;
    const int   blas = 0;

    // Choose blocked algorithm based on pivot strategy
    switch ( pivot ) {
    case 'B':
        ldlt_block_rook_pivot( blas, pivot, n, piv, ord, ldim, A );
        break;
    case 'K':
        ldlt_block_rook_pivot( blas, pivot, n, piv, ord, ldim, A );
        break;
    case 'P':
        ldlt_block_comp_pivot( blas, pivot, n, piv, ord, ldim, A );
        break;
    default:
        ldlt_block_rook_pivot( blas, pivot, n, piv, ord, ldim, A );
        break;
    }
```

```
}

/*
 * Implements simple blocking to factorize symmetric indefinite n-by-n matrix A
 * into a unit lower triangular matrix L and block diagonal matrix D with block
 * order 1 or 2.  Symmetrically permuted matrix A^ = P*A*P' = L*D*L', where P
 * is the permutation matrix, and L' and P' are the transpose of L and P,
 * respectively.  Permutation matrix P is encoded in vectors piv[] and ord[],
 * such that row/ column k is interchanged with row/ column piv[k], and ord[k]
 * specifies the diagonal block order.  To the extent possible, this
 * implementation of a blocked algorithm for symmetric indefinite factorization
 * uses the BLAS library to perform matrix operations.  In the case where a
 * partial or rook pivoting strategy is specified the blocked algorithm employs
 * the SAXPY operation to perform symmetric indefinite factorization --
 * cumulative trailing sub-matrix updates are applied to candidate pivot row(s)/
 * column(s) as the factorization proceeds.  When a complete pivoting strategy
 * is specified the entire trailing sub-matrix must be updated prior to each
 * pivot selection, so the blocked algorithm employs the outer product method.
 */
void ldlt_block_blas( char pivot, int n, int *piv, int *ord, double *A )
{
    const int    blas = 1;
    const int    ldim = n;

    // Choose blocked algorithm based on pivot strategy
    switch ( pivot ) {
    case 'B':
        ldlt_block_rook_pivot( blas, pivot, n, piv, ord, ldim, A );
        break;
    case 'K':
        ldlt_block_rook_pivot( blas, pivot, n, piv, ord, ldim, A );
        break;
    case 'P':
        ldlt_block_comp_pivot( blas, pivot, n, piv, ord, ldim, A );
        break;
    default:
        ldlt_block_rook_pivot( blas, pivot, n, piv, ord, ldim, A );
        break;
    }
}

/*
 * Wrapper for calling LAPACK routine DPOTF2 which computes the Cholesky
 * factorization of a real symmetric positive definite matrix.  DPOTF2 is
 * LAPACK's unblocked version of Cholesky factorization.
 */
void ldlt_lapack_unblocked( char pivot, int n, int *piv, int *ord, double *A )
{
    const char   lower = 'L';
    const int    ldim = n;
    int          info = 0;

    dsytf2_( &lower, &n, A, &ldim, piv, &info );
}

/*
 * Wrapper for calling LAPACK routine DSYTRF which computes the factorization
 * of a real symmetric indefinite matrix.
 */
void ldlt_lapack( char pivot, int n, int *piv, int *ord, double *A )
{
    const char   lower = 'L';
    const int    lapack = 1;
    const int    blas = 0;
    const int    ldim = n;
    const int    bdim = get_block_dim_ldlt( lapack, blas, ldim );

    int          lwork = ldim*bdim;
    int          info = 0;
    double       *work;

    work = (double *) malloc( lwork*sizeof(double) );
    dsytrf_( &lower, &n, A, &ldim, piv, work, &lwork, &info );
    free( work );
}
```

A.6. modchol.c – modified Cholesky algorithms.

```
/*
 * Gill-Murray-Wright and Cheng-Higham modified Cholesky algorithms.  Unblocked
 * versions of these algorithms include the outer product method and SAXPY
 * operation, while blocked versions include simple blocking and an
 * implementation that uses tuned BLAS (Basic Linear Algebra Subroutines).
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <float.h>

#include "modchol.h"
#include "ldltfact.h"
#include "lapack.h"
#include "matcom.h"
#include "timing.h"

static double calc_beta_sqr_gmw( int n, int ldim, const double *A );
static double calc_delta_ch( int n, int ldim, const double *A );
static void pivot_sym_blas( int n, int k, int r, int ldim, double *W,
    double *A );
static double modify_pivot_gmw( double delta, double beta_sqr, int n,
    double *vec );
static void ldlt_spectral_decomp( int ldim, const double *A, double *U,
    double *lambda );
static void modify_blk_diag( int type, double delta, int n, const int *ord,
    int ldim, double *A );
static void chol_gmw_factor( char pivot, double delta, double beta_sqr,
    int m, int n, int *piv, int *ord, double *diag, int ldim, double *A,
    double *W );
static void chol_gmw_factor_blas( char pivot, double delta, double beta_sqr,
    int m, int n, int *piv, int *ord, double *diag, int ldim, double *A,
    double *W );
static void chol_gmw_block_handler( int blas, char pivot, double delta,
    double beta_sqr, int n, int *piv, int *ord, int ldim, double *A );

#if defined(MODCHOL) && defined(PROFILE)
    static double tm_mod_chol = 0.0;
    static double tm_mod_fact = 0.0;
#endif

/*
 * Computes and returns a parameter, beta_sqr, that bounds the magnitude of the
 * off-diagonal elements in the unit lower triangular matrix produced by
 * symmetric indefinite factorization, P*A*P' = L*D*L'.  The function value is
 * used in the modified Cholesky algorithm proposed by Gill, Murray and Wright
 * to determine the perturbation applied to symmetric indefinite n-by-n matrix A
 * to make it positive definite.
 */
double calc_beta_sqr_gmw( int n, int ldim, const double *A )
{
    double mu = -1.0;
    double nu = -1.0;
    double beta_sqr;

    for ( int j = 0; j < n; j++ ) {
        const double *A_j = A + j*ldim;
        // Maximum magnitude of diagonal elements
        double ajj = fabs( *(A_j+j) );
        if ( ajj > nu ) {
            nu = ajj;
        }
        // Maximum magnitude of off-diagonal elements -- exploit symmetry
        for ( int i = j+1; i < n; i++ ) {
            double aij = fabs( *(A_j+i) );
            if ( aij > mu ) {
                mu = aij;
            }
        }
    }

    beta_sqr = mu / sqrt( n*n - 1 );
    if ( nu > beta_sqr ) {
        beta_sqr = nu;
    }
    if ( DBL_EPSILON > beta_sqr ) {
        beta_sqr = DBL_EPSILON;
    }

    return beta_sqr;
}

/*
 * Computes and returns a preset modification tolerance, delta, which is used to
 * determine the perturbation applied to symmetric indefinite n-by-n matrix A
 * to make it positive definite.  The function value is used in the modified
 * Cholesky algorithm proposed by Cheng and Higham.
 */
double calc_delta_ch( int n, int ldim, const double *A )
{
    double delta = sqrt( DBL_EPSILON / 2.0 );
    double sigma = -1.0;

    // Compute sigma = infinity norm of symmetric matrix A
    // = maximum absolute row sum = maximum absolute column sum
    for ( int j = 0; j < n; j++ ) {
        double sum = 0.0;
        const double *A_j = A + j*ldim;
        for ( int i = 0; i < n; i++ ) {
            sum += fabs( *(A_j+i) );
        }
        if ( sum > sigma ) {
            sigma = sum;
        }
    }
    delta *= sigma;

    return delta;
}

/*
 * Performs pivoting of an n-by-n symmetric matrix A, and working array W, which
 * stores trailing sub-matrix updates applied to columns of A.  Arrays A and W
 * are stored in column-major order with leading dimension ldim.  To preserve
 * the symmetry of matrix A both row and column k are interchanged with row and
 * column r.  A^ = P*A*P', where P is the permutation matrix and P' its
 * transpose, is called a symmetric permutation of A.  BLAS routine DSWAP is
 * invoked to perform symmetric pivoting.  Because of symmetry only elements on
 * and below the diagonal need be interchanged.
 */
void pivot_sym_blas( int n, int k, int r, int ldim, double *W, double *A )
{
    const int one = 1;

    int     t;
    double  *A_k = A + k*ldim;
    double  *A_r = A + r*ldim;
    double  *Ak_ = A + k;
    double  *Ar_ = A + r;
    double  *Wk_ = W + k;
    double  *Wr_ = W + r;

    if ( k != r ) {
        // Interchange elements A(k,0:k-1) with A(r,0:k-1) and W(k,0:k-1) with
        // W(r,0:k-1), i.e., elements of rows k and r to the left of column k
        dswap_( &k, Ak_, &ldim, Ar_, &ldim );
        dswap_( &k, Wk_, &ldim, Wr_, &ldim );

        // Interchange diagonal elements of rows/ columns k and r
        double akk = *(A_k + k);
        *(A_k + k) = *(A_r + r);
        *(A_r + r) = akk;

        // Interchange elements A(k+1:r-1,k) with A(r,k+1:r-1)
        double  *Arl = Ar_ + k*ldim + ldim;
        double  *Alk = A_k + k + 1;
        t = r - k - 1;
        dswap_( &t, Alk, &one, Arl, &ldim );

        // Interchange elements A(r+1:n-1,k) with A(r+1:n-1,r)
        // i.e., elements of columns k and r below row r
        double  *Asr = A_r + r + 1;
        double  *Ask = A_k + r + 1;
        t = n - r - 1;
        dswap_( &t, Ask, &one, Asr, &one );
    }
}

/*
 * Modifies the selected pivot -- element vec[0] equal to akk, the kth diagaonal
 * element of some matrix A -- such that matrix (A + dA) is sufficiently
 * positive definite and reasonably well-conditioned.  The Gill-Murray-Wright
 * algorithm performs a Type-I modification,
 * akk = max{|akk|, delta, c_sqr/beta_sqr}, where c_sqr is the square of the
 * infinity norm of vec[1:n-1] = A(k+1:k+n-1,k).
 */
double modify_pivot_gmw( double delta, double beta_sqr, int n, double *vec )
{
#if defined(MODCHOL) && defined(PROFILE)
    struct  timespec sta_mod_fact, end_mod_fact;

    get_time( &sta_mod_fact );
#endif
    double c_sqr = -1.0;
    double akk = fabs( vec[0] );
    for ( int i = 1; i < n; i++ ) {
        double x = vec[i] * vec[i];
        if ( x > c_sqr ) {
            c_sqr = x;
        }
    }
    if ( (c_sqr / beta_sqr) > akk ) {
        akk = c_sqr / beta_sqr;
    }
    if ( delta > akk ) {
        akk = delta;
    }
#if defined(MODCHOL) && defined(PROFILE)
    get_time( &end_mod_fact );
    tm_mod_fact += timespec_diff( sta_mod_fact, end_mod_fact );
#endif
    return akk;
}

/*
 * Computes the spectral decomposition of 2-by-2 symmetric matrix A,
 * A = U * [lambda[0], 0; 0, lambda[1]] * U', where lambda[0] and lambda[1]
 * are eigenvalues of A, and eigenvectors belonging to lambda[0] and lambda[1]
 * form the columns of orthogonal matrix U.
 */
void ldlt_spectral_decomp( int ldim, const double *A, double *U, double *lambda )
{
    // Trace of symmetric 2-by-2 matrix A
    double trA = *A + *(A + 1 + ldim);
    // Determinant of symmetric 2-by-2 matrix A
    double detA = *A * *(A + 1 + ldim) - *(A + 1) * *(A + 1);
    // The characteristic polynomial of 2-by-2 matrix A is:
    // p(t) = t*t - trA*t + detA
    // Calculate roots of the characteristic polynomial, i.e., eigenvalues.
    // Note that if A is a real symmetric 2-by-2 matrix, with non-zero
    // off-diagonal entries, then A has two distinct real eigenvalues
    lambda[0] = ( trA + sqrt(trA*trA - 4.0*detA) ) / 2.0;
    lambda[1] = ( trA - sqrt(trA*trA - 4.0*detA) ) / 2.0;

    if ( lambda[0] == lambda[1] ) {
        // A is 2-by-2 diagonal matrix with one distinct diagonal element.
        // Single distinct eigenvalue is equal to diagonal element,
        // and U is the identity matrix
        U[0] = 1;
        U[1] = 0;
        U[2] = 0;
        U[3] = 1;
    } else {
        // Compute eigenvectors belonging to distinct real eigenvalues.
        // y = a*x --> u = alpha * (1; a), alpha any real number
        double a = -(*A - lambda[0]) / *(A + 1);
        double norm_u = sqrt( 1.0 + a*a );
        U[0] = 1 / norm_u;
        U[1] = a / norm_u;
        // y = b*x --> v = beta * (1; b), beta any real number
        double b = -(*A - lambda[1]) / *(A + 1);
        double norm_v = sqrt( 1.0 + b*b );
        U[2] = 1 / norm_v;
        U[3] = b / norm_v;
    }
}

/*
 * Modifies the diagonal block matrix computed by symmetric indefinite
 * factorization, P*A*P' = L*D*L', where A is an n-by-n symmetric matrix, P is a
 * permutation matrix, L is unit lower triangular and D is block diagonal with
 * block order 1 or 2.  Block diagonal matrix D is modified to make matrix
 * (A + dA) positive definite.  Each 1-by-1 diagonal block is modified so that
 * either dk = max{delta, |dk|} (Type-I modification) or dk = max{delta, dk}
 * (Type-II modification), where dk is the kth diagonal element and delta is a
 * preset modification tolerance.  The modification type is specified in the
 * argument list.  For each 2-by-2 diagonal block, first compute its spectral
 * decompostion, D = U * [lambda[0], 0; 0, lambda[1]] * U', where lambda[0] and
 * lambda[1] are eigenvalues of D and U is orthogonal.  Then apply Type-I or
 * Type-II modification to the eigenvalues and calculate the modified block
 * diagonal D.
 */
void modify_blk_diag( int type, double delta, int n, const int *ord,
    int ldim, double *A )
{
    double U[2*2];
    double lambda[2*1];

    for ( int k = 0; k < n; ) {
        double *Akk = A + k + k*ldim;

        if ( ord[k] == 1 ) {         // 1-by-1 pivot
            if ( type == 1 ) {       // Type-I modification
                *Akk = fabs( *Akk );
            }
            if ( delta > *Akk ) {
                *Akk = delta;
            }
            k++;
        } else {                     // 2-by-2 pivot
            ldlt_spectral_decomp( ldim, Akk, U, lambda );
            if ( type == 1 ) {       // Type-I modification
                lambda[0] = fabs( lambda[0] );
                lambda[1] = fabs( lambda[1] );
            }
            if ( delta > lambda[0] || delta > lambda[1] ) {
                for ( int i = 0; i < 2; i++ ) {
                    if ( delta > lambda[i] ) {
                        lambda[i] = delta;
                    }
                }
                // D^ = U * [lambda[0]^, 0; 0, lambda[1]^] * U'
                *Akk = U[0] * lambda[0] * U[0] + U[1] * lambda[1] * U[1];
                *(Akk + 1) = U[2] * lambda[0] * U[0] + U[3] * lambda[1] * U[1];
                *(Akk + 1 + ldim) = U[2] * lambda[0] * U[2] +
                    U[3] * lambda[1] * U[3];
            }
            k += 2;
        }
    }
}

/*
 * Implements a rectangular version of the SAXPY operation (jki indexing) for
 * the modified Cholesky algorithm proposed by Gill, Murray and Wright.  If the
 * n-by-n symmetric principal minor of A is not positive definite, it is
 * perturbed such that (A + dA) is sufficiently positive definite and reasonably
 * well-conditioned while preserving as much as possible the information
 * contained in the Hessian.  Factorization yields P*(A+dA)*P' = L*D*L', where
 * L is unit lower triangular and D is diagonal.  The permutation matrix P is
 * encoded in vectors piv[] and ord[], which contain the pivot index and its
 * order (=1).  The pivoting strategy is passed in the argument list.  The
 * Gill-Murray-Wright algorithm modifies diagonal matrix D, computed by
 * symmetric indefinite factorization, as the decomposition proceeds to make
 * matrix (A + dA) positive definite.  Factors L and D overwrite matrix A on
 * and below the diagonal.
 */
void chol_gmw_factor( char pivot, double delta, double beta_sqr, int m, int n,
    int *piv, int *ord, double *diag, int ldim, double *A, double *W )
{
    for ( int j = 0; j < n; j++ ) {
        double *A_j = A + j*ldim;
        double *Ajj = A_j + j;

        // Determine pivot using method specified in the argument list
        switch ( pivot ) {
        case 'D':
            eval_pivot_diag( m-j, j, &diag[j], piv, ord );
            break;
        default:
            eval_pivot_diag( m-j, j, &diag[j], piv, ord );
            break;
        }
        // Perform symmetric pivoting
        if ( j != piv[j] ) {
            pivot_sym( m, j, piv[j], ldim, A );
            double dj = diag[j];
            diag[j] = diag[ piv[j] ];
            diag[ piv[j] ] = dj;
        }
        // Perform cumulative trailing sub-matrix updates on diagonal element
        // and elements below the diagonal of column j
        *Ajj = diag[j];
        for ( int k = 0; k < j; k++ ) {
            const double *A_k = A + k*ldim;
            double akk = *(A_k + k);
            double ajk = *(A_k + j);
            for ( int i = j+1; i < m; i++ ) {
                *(A_j + i) -= *(A_k + i) * ajk * akk;
            }
        }

        // Modify diagonal element (pivot) so that matrix (A + dA) is
        // sufficiently positive definite and reasonably well-conditioned
        double ajj = modify_pivot_gmw( delta, beta_sqr, m-j, Ajj );
        *Ajj = ajj;
        // Divide elements of column k of matrix A below the diagonal by the
        // diagonal element, and perform trailing sub-matrix update on the
        // vector diagonal elements used in pivot selection
        for ( int i = j+1; i < m; i++ ) {
            *(A_j + i)  /= ajj;
            diag[i] -= *(A_j + i) * ajj *  *(A_j + i);
        }
    }
}

/*
 * Implements a rectangular version of the SAXPY operation (jki indexing) for
 * the modified Cholesky algorithm proposed by Gill, Murray and Wright.  If the
 * n-by-n symmetric principal minor of A is not positive definite, it is
 * perturbed such that (A + dA) is sufficiently positive definite and reasonably
 * well-conditioned while preserving as much as possible the information
 * contained in the Hessian.  Factorization yields P*(A+dA)*P' = L*D*L', where
 * L is unit lower triangular and D is diagonal.  The permutation matrix P is
 * encoded in vectors piv[] and ord[], which contain the pivot index and its
 * order (=1).  The pivoting strategy is passed in the argument list.  The
 * Gill-Murray-Wright algorithm modifies diagonal matrix D, computed by
 * symmetric indefinite factorization, as the decomposition proceeds to make
 * matrix (A + dA) positive definite.  To the extent possible, this
 * implementation of the SAXPY operation uses the BLAS library to perform
 * matrix operations.  Factors L and D overwrite matrix A on and below the
 * diagonal.
 */
void chol_gmw_factor_blas( char pivot, double delta, double beta_sqr,
    int m, int n, int *piv, int *ord, double *diag, int ldim, double *A,
    double *W )
{
    for ( int j = 0; j < n; j++ ) {
        double *L = A + j;
        double *M = W + j;
        double *W_j = W + j*ldim;
        double *A_j = A + j*ldim;
        double *Ajj = A_j + j;

        // Determine pivot using method specified in the argument list
        switch ( pivot ) {
        case 'D':
            eval_pivot_diag( m-j, j, &diag[j], piv, ord );
            break;
        default:
            eval_pivot_diag( m-j, j, &diag[j], piv, ord );
            break;
        }
        // Perform symmetric pivoting
        if ( j != piv[j] ) {
            pivot_sym_blas( m, j, piv[j], ldim, W, A );
            double dj = diag[j];
            diag[j] = diag[ piv[j] ];
            diag[ piv[j] ] = dj;
        }
        // Perform cumulative trailing sub-matrix updates on diagonal element
        // and elements below the diagonal of column j
        // A = A = ML'
        *Ajj = diag[j];
        reduce_ldlt_vector_blas( m-j-1, j, 0, ord, ldim, L, M+1, Ajj+1 );

        // Modify diagonal element (pivot) so that matrix (A + dA) is
        // sufficiently positive definite and reasonably well-conditioned
        double ajj = modify_pivot_gmw( delta, beta_sqr, m-j, Ajj );
        *Ajj = ajj;
        // Elements of column k of matrix A on and below the diagonal are equal
        // to elements of column k of L*D.  Store column k of L*D in W before
        // solving for L by dividing column k of L*D by diagonal element dkk.
        // Then perform trailing sub-matrix updates on the vector of diagonal
        // elements used in pivot selection
        for ( int i = j+1; i < m; i++ ) {
            *(W_j + i) = *(A_j + i);
            *(A_j + i)  /= ajj;
            diag[i] -= *(A_j + i) * ajj *  *(A_j + i);
        }
    }
}

/*
 * Implements the modified Cholesky algorithm proposed by Gill, Murray and
 * Wright using simple blocking to optimize memory access.  If n-by-n symmetric
 * matrix A with leading dimension ldim is not positive definite, it is
 * perturbed such that (A + dA) is sufficiently positive definite and reasonably
 * well-conditioned while preserving as much as possible the information
 * contained in the Hessian.  Factorization yields P*(A+dA)*P' = L*D*L', where L
 * is unit lower triangular and D is diagonal.  The permutation matrix P is
 * encoded in vectors piv[] and ord[], which contain the pivot and its order.
 * The pivoting strategy is passed in the argument list.  The Gill-Murray-Wright
 * algorithm modifies diagonal matrix D, computed by symmetric indefinite
 * factorization, as the decomposition proceeds to make matrix (A + dA) positive
 * definite. The blocked algorithm handler determines which implementation of
 * SAXPY operation -- native or using BLAS -- to invoke to factor a column block
 * of matrix A.  Factors L and D overwrite matrix A on and below the diagonal.
 */
void chol_gmw_block_handler( int blas, char pivot, double delta, double beta_sqr,
    int n, int *piv, int *ord, int ldim, double *A )
{
    const int   lapack = 0;
    const int   bdim = get_block_dim_ldlt( lapack, blas, ldim );

    int      d, j, r, t;
    double   *Ajj, *L, *D, *W, *diag;
```

```
void    (*chol_gmw)( char pivot, double delta, double beta_sqr,
             int m, int n, int *piv, int *ord, double *diag, int ldim,
             double *M, double *A  );

if ( blas ) {
    chol_gmw = chol_gmw_factor_blas;
    W = (double *) malloc( ldim*bdim*sizeof(double) );
} else {
    chol_gmw = chol_gmw_factor;
    W = (double *) malloc( ldim*2*sizeof(double) );
}

// For efficient memory access during pivot selection, copy diagonal
// elements of matrix A into vector diag[]
diag = (double *) malloc( ldim*sizeof(double) );
for ( int k = 0; k < n; k++ ) {
    diag[k] = *(A + k + k*ldim);
}

j = 0;
r = (bdim > n) ? n : bdim;
// Perform rectangular factorization on first column block A(0:n-1,0:r)
chol_gmw( pivot, delta, beta_sqr, n, r, &piv[j], &ord[j], &diag[j],
    ldim, A, W );

d = 0;
j = bdim;
t = n - bdim;
Ajj = A;
for ( ; j < n; j += bdim, d += bdim, t -= bdim ) {

    // Adjust pivot vector of previous block for diagonal offset
    for ( int i = d; i < j; i++ ) {
        piv[i] += d;
    }
    L = Ajj + bdim;
    D = Ajj;
    Ajj = A + j + j*ldim;
    // Reduce trailing sub-matrix, P * A(j:n-1,j:n-1) * P' =
    // L(j:n-1,j-BDIM:j-1) * D(j-BDIM:j-1,j-BDIM:j-1) * L'(j-BDIM:j-1,j:n-1)
    reduce_ldlt_mat_blk( blas, t, r, &ord[d], bdim, ldim, L, D, W+j, Ajj );

    r = t < bdim ? t : bdim;
    // Perform rectangular factorization on column block A(j:n-1,j:j+r-1)
    chol_gmw( pivot, delta, beta_sqr, t, r, &piv[j], &ord[j], &diag[j],
        ldim, Ajj, W+j );

    // Apply permutation matrix for current block, encoded in piv(j:j+r-1),
    // to columns to the left of current block A(:,0:j-1)
    for ( int i = j; i < j+r; i++ ) {
```

```
                if ( i != piv[i] + j ) {
                    if ( blas ) {
                        double *Ai_ = A + i;
                        double *Ar_ = A + piv[i] + j;
                        dswap_( &j, Ai_, &ldim, Ar_, &ldim );
                    } else {
                        for ( int k = 0; k < j; k++ ) {
                            double aik = *(A + i + k*ldim);
                            *(A + i + k*ldim) = *(A + piv[i] + j + k*ldim);
                            *(A + piv[i] + j + k*ldim) = aik;
                        }
                    }
                }
            }
        }
    // Adjust pivot vector of last block for diagonal offset
    for ( int i = d; i < n; i++ ) {
        piv[i] += d;
    }
    free( diag );
    free( W );
}

/***************************************************************************/

/*
 * Implements the modified Cholesky algorithm proposed by Gill, Murray and
 * Wright using the outer product method (kji indexing).  If n-by-n symmetric
 * matrix A is not positive definite, it is perturbed such that (A + dA) is
 * sufficiently positive definite and reasonably well-conditioned while
 * preserving as much as possible the information contained in the Hessian.
 * Factorization yields P*(A+dA)*P' = L*D*L', where L is unit lower triangular
 * and D is diagonal.  The permutation matrix P is encoded in vectors piv[] and
 * ord[], which contain the pivot and its order (=1).  The pivoting strategy is
 * passed in the argument list.  The Gill-Murray-Wright algorithm modifies
 * diagonal matrix D, computed by symmetric indefinite factorization, as the
 * proceeds to make matrix (A + dA) positive definite.  Factors L and D
 * overwrite matrix A on and below the diagonal.
 */
void chol_gmw_outer_product( char pivot, int n, int *piv, int *ord, double *A )
{
    const int ldim = n;

    double *diag;
    double delta = DBL_EPSILON;
    double beta_sqr = calc_beta_sqr_gmw( n, ldim, A );

    diag = (double *) malloc( ldim*sizeof(double) );

    for ( int k = 0; k < n; k++ ) {
        double *A_k = A + k*ldim;
        double *Akk = A_k + k;
        // For efficient memory access during pivot selection, copy diagonal
        // elements of trailing sub-matrix A into vector diag[]
        for ( int j = k; j < n; j++ ) {
            diag[j] = *(A + j + j*ldim);
        }
        // Determine pivot using method specified in the argument list
        switch ( pivot ) {
        case 'D':
            eval_pivot_diag( n-k, k, &diag[k], piv, ord );
            break;
        default:
            eval_pivot_diag( n-k, k, &diag[k], piv, ord );
            break;
        }
        // Perform symmetric pivoting
        if ( k != piv[k] ) {
            pivot_sym( n, k, piv[k], ldim, A );
        }

        // Modify diagonal element (pivot) so that matrix (A + dA) is
        // sufficiently positive definite and reasonably well-conditioned
        double akk = modify_pivot_gmw( delta, beta_sqr, n-k, Akk );
        *Akk = akk;
        // Divide elements of column k below the diagonal by the diagonal element
        for ( int i = k+1; i < n; i++ ) {
            *(A_k + i)  /= akk;
        }
        // Update trailing sub-matrix by subtracting the outer product.
        // Because of symmetry need only update elements on and below diagonal
        for ( int j = k+1; j < n; j++ ) {
            double *A_j = A + j*ldim;
            double ajk = *(A_k + j);
            for ( int i = j; i < n; i++ ) {
                *(A_j+i) -= *(A_k+i) * ajk * akk;
            }
        }
    }
}

/*
 * Implements the modified Cholesky algorithm proposed by Gill, Murray and
 * Wright using the SAXPY operation (jki indexing).  If n-by-n symmetric
 * matrix A is not positive definite, it is perturbed such that (A + dA) is
 * sufficiently positive definite and reasonably well-conditioned while
 * preserving as much as possible the information contained in the Hessian.
 * Factorization yields P*(A+dA)*P' = L*D*L', where L is unit lower triangular
 * and D is diagonal.  The permutation matrix P is encoded in vectors piv[] and
 * ord[], which contain the pivot and its order (=1).  The pivoting strategy is
 * passed in the argument list.  The Gill-Murray-Wright algorithm modifies
 * diagonal matrix D, computed by symmetric indefinite factorization, as the
 * proceeds to make matrix (A + dA) positive definite.  Factors L and D
 * overwrite matrix A on and below the diagonal.
 */
void chol_gmw_saxpy( const char pivot, const int n, int *piv, int *ord,
    double *A )
{
    const int ldim = n;

    double *W, *diag;
    double delta = DBL_EPSILON;
    double beta_sqr = calc_beta_sqr_gmw( n, ldim, A );

    W = (double *) malloc( sizeof(double) );
    // For efficient memory access during pivot selection, copy diagonal
    // elements of matrix A into vector diag[]
    diag = (double *) malloc( ldim*sizeof(double) );
    for (int k = 0; k < n; k++) {
        diag[k] = *(A + k + k*ldim);
    }

    chol_gmw_factor(pivot, delta, beta_sqr, n, n, piv, ord, diag, ldim, A, W);

    free( diag );
    free( W );
}

/*
 * Implements simple blocking for the modified Cholesky algorithm proposed by
 * Gill, Murray and Wright.  A is an n-by-n symmetric, possibly indefinite,
 * matrix.
 */
void chol_gmw_block( char pivot, int n, int *piv, int *ord, double *A )
{
    const int ldim = n;
    const int blas = 0;
#if defined(MODCHOL) && defined(PROFILE)
    struct  timespec sta_mod_chol, sta_mod_fact, end_mod_chol, end_mod_fact;

    tm_mod_chol = 0.0;
    tm_mod_fact = 0.0;
    get_time( &sta_mod_chol );

    get_time( &sta_mod_fact );
#endif
    double delta = DBL_EPSILON;
    double beta_sqr = calc_beta_sqr_gmw( n, ldim, A );
#if defined(MODCHOL) && defined(PROFILE)
```

```
    get_time( &end_mod_fact );
    tm_mod_fact += timespec_diff( sta_mod_fact, end_mod_fact );
#endif

    chol_gmw_block_handler( blas, pivot, delta, beta_sqr, n, piv, ord, ldim, A );

#if defined(MODCHOL) && defined(PROFILE)
    get_time( &end_mod_chol );
    tm_mod_chol += timespec_diff( sta_mod_chol, end_mod_chol );
    fprintf( stdout, "%.4f\t\t%.4f\t\t%.2f\n", tm_mod_chol, tm_mod_fact,
        tm_mod_fact/tm_mod_chol*100 );
#endif
}

/*
 * Implements simple blocking using the BLAS library for the modified Cholesky
 * algorithm proposed by Gill, Murray and Wright.  A is an n-by-n symmetric,
 * possibly indefinite, matrix.
 */
void chol_gmw_block_blas( char pivot, int n, int *piv, int *ord, double *A )
{
    const int ldim = n;
    const int blas = 1;
#if defined(MODCHOL) && defined(PROFILE)
    struct  timespec sta_mod_chol, sta_mod_fact, end_mod_chol, end_mod_fact;

    tm_mod_chol = 0.0;
    tm_mod_fact = 0.0;
    get_time( &sta_mod_chol );

    get_time( &sta_mod_fact );
#endif
    double delta = DBL_EPSILON;
    double beta_sqr = calc_beta_sqr_gmw( n, ldim, A );
#if defined(MODCHOL) && defined(PROFILE)
    get_time( &end_mod_fact );
    tm_mod_fact += timespec_diff( sta_mod_fact, end_mod_fact );
#endif

    chol_gmw_block_handler( blas, pivot, delta, beta_sqr, n, piv, ord, ldim, A );

#if defined(MODCHOL) && defined(PROFILE)
    get_time( &end_mod_chol );
    tm_mod_chol += timespec_diff( sta_mod_chol, end_mod_chol );
    fprintf( stdout, "%.4f\t\t%.4f\t\t%.2f\n", tm_mod_chol, tm_mod_fact,
        tm_mod_fact/tm_mod_chol*100 );
#endif
}
```

```
/*
 * Implements the modified Cholesky algorithm proposed by Cheng and Higham using
 * the outer product method (kji indexing). . If n-by-n symmetric matrix A is
 * not positive definite, it is perturbed such that (A + dA) is sufficiently
 * positive definite and reasonably well-conditioned while preserving as much as
 * possible the information contained in the Hessian. Factorization yields
 * P*(A+dA)*P' = L*D*L', where L is unit lower triangular and D is block
 * diagonal with block order 1 or 2. The permutation matrix P is encoded in
 * vectors piv[] and ord[], which contain the pivot and its order. The pivoting
 * strategy (Bunch-Kaufman, bounded Bunch-Kaufman or Bunch-Parlett) is passed in
 * the argument list. Once the symmetric indefinite factorization has been
 * computed for matrix A, block diagonal matrix D is modified to make matrix
 * (A + dA) positive definite. The Cheng-Higham algorithm performs a Type-II
 * modification of the block diagonal matrix. Factors L and D overwrite matrix
 * A on and below the diagonal.
 */
void chol_ch_outer_product( const char pivot, const int n, int *piv, int *ord,
    double *A )
{
    const int   ldim = n;
    const int   type = 2;

    double delta = calc_delta_ch( n, ldim, A );

    ldlt_outer_product( pivot, n, piv, ord, A );
    modify_blk_diag( type, delta, n, ord, ldim, A );
}

/*
 * Implements the modified Cholesky algorithm proposed by Cheng and Higham using
 * the SAXPY operation (jki indexing). If n-by-n symmetric matrix A is not
 * positive definite, it is perturbed such that (A + dA) is sufficiently
 * positive definite and reasonably well-conditioned while preserving as much as
 * possible the information contained in the Hessian. Factorization yields
 * P*(A+dA)*P' = L*D*L', where L is unit lower triangular and D is block
 * diagonal with block order 1 or 2. The permutation matrix P is encoded in
 * vectors piv[] and ord[], which contain the pivot and its order. The pivoting
 * strategy (Bunch-Kaufman, bounded Bunch-Kaufman or Bunch-Parlett) is passed in
 * the argument list. Once the symmetric indefinite factorization has been
 * computed for matrix A, block diagonal matrix D is modified to make matrix
 * (A + dA) positive definite. The Cheng-Higham algorithm performs a Type-II
 * modification of the block diagonal matrix. Factors L and D overwrite matrix
 * A on and below the diagonal.
 */
void chol_ch_saxpy( const char pivot, const int n, int *piv, int *ord,
    double *A )
{
    const int   ldim = n;
    const int   type = 2;
    double delta = calc_delta_ch( n, ldim, A );

    ldlt_saxpy( pivot, n, piv, ord, A );
    modify_blk_diag( type, delta, n, ord, ldim, A );
}

/*
 * Implements the modified Cholesky algorithm proposed by Cheng and Higham using
 * simple blocking to optimize memory access.  If n-by-n symmetric matrix A
 * is not positive definite, it is perturbed such that (A + dA) is sufficiently
 * positive definite and reasonably well-conditioned while preserving as much as
 * possible the information contained in the Hessian.  Factorization yields
 * P*(A+dA)*P' = L*D*L', where L is unit lower triangular and D is block
 * diagonal with block order 1 or 2.  The permutation matrix P is encoded in
 * vectors piv[] and ord[], which contain the pivot and its order.  The pivoting
 * strategy (Bunch-Kaufman, bounded Bunch-Kaufman or Bunch-Parlett) is passed in
 * the argument list.  Once the symmetric indefinite factorization has been
 * computed for matrix A, block diagonal matrix D is modified to make matrix
 * (A + dA) positive definite.  The Cheng-Higham algorithm performs a Type-II
 * modification of the block diagonal matrix.  Factors L and D overwrite matrix
 * A on and below the diagonal.
 */
void chol_ch_block( const char pivot, const int n, int *piv, int *ord,
    double *A )
{
    const int    ldim = n;
    const int    type = 2;
#if defined(MODCHOL) && defined(PROFILE)
    struct  timespec sta_mod_chol, sta_mod_fact, end_mod_chol, end_mod_fact;

    tm_mod_chol = 0.0;
    tm_mod_fact = 0.0;
    get_time( &sta_mod_chol );

    get_time( &sta_mod_fact );
#endif
    double delta = calc_delta_ch( n, ldim, A );
#if defined(MODCHOL) && defined(PROFILE)
    get_time( &end_mod_fact );
    tm_mod_fact += timespec_diff( sta_mod_fact, end_mod_fact );
#endif

    ldlt_block( pivot, n, piv, ord, A );
#if defined(MODCHOL) && defined(PROFILE)
    get_time( &sta_mod_fact );
#endif
    modify_blk_diag( type, delta, n, ord, ldim, A );
#if defined(MODCHOL) && defined(PROFILE)
    get_time( &end_mod_fact );
    tm_mod_fact += timespec_diff( sta_mod_fact, end_mod_fact );
    get_time( &end_mod_chol );
    tm_mod_chol += timespec_diff( sta_mod_chol, end_mod_chol );
    fprintf( stdout, "%.4f\t\t%.4f\t\t%.2f\n", tm_mod_chol, tm_mod_fact,
        tm_mod_fact/tm_mod_chol*100 );
#endif
}

/*
 * Implements simple blocking using BLAS for the modified Cholesky algorithm
 * proposed by Cheng and Higham.  If n-by-n symmetric matrix A is not positive
 * definite, it is perturbed such that (A + dA) is sufficiently positive
 * definite and reasonably well-conditioned while preserving as much as possible
 * the information contained in the Hessian.  Factorization yields
 * P*(A+dA)*P' = L*D*L', where L is unit lower triangular and D is block
 * diagonal with block order 1 or 2.  The permutation matrix P is encoded in
 * vectors piv[] and ord[], which contain the pivot and its order.  The pivoting
 * strategy (Bunch-Kaufman, bounded Bunch-Kaufman or Bunch-Parlett) is passed in
 * the argument list.  Once the symmetric indefinite factorization has been
 * computed for matrix A, block diagonal matrix D is modified to make matrix
 * (A + dA) positive definite.  The Cheng-Higham algorithm performs a Type-II
 * modification of the block diagonal matrix.  Factors L and D overwrite matrix
 * A on and below the diagonal.
 */
void chol_ch_block_blas( const char pivot, const int n, int *piv, int *ord,
    double *A )
{
    const int   ldim = n;
    const int   type = 2;
#if defined(MODCHOL) && defined(PROFILE)
    struct  timespec sta_mod_chol, sta_mod_fact, end_mod_chol, end_mod_fact;

    tm_mod_chol = 0.0;
    tm_mod_fact = 0.0;
    get_time( &sta_mod_chol );

    get_time( &sta_mod_fact );
#endif
    double delta = calc_delta_ch( n, ldim, A );
#if defined(MODCHOL) && defined(PROFILE)
    get_time( &end_mod_fact );
    tm_mod_fact += timespec_diff( sta_mod_fact, end_mod_fact );
#endif

    ldlt_block_blas( pivot, n, piv, ord, A );
#if defined(MODCHOL) && defined(PROFILE)
    get_time( &sta_mod_fact );
#endif
    modify_blk_diag( type, delta, n, ord, ldim, A );
#if defined(MODCHOL) && defined(PROFILE)
```

```
    get_time( &end_mod_fact );
    tm_mod_fact += timespec_diff( sta_mod_fact, end_mod_fact );

    get_time( &end_mod_chol );
    tm_mod_chol += timespec_diff( sta_mod_chol, end_mod_chol );
    fprintf( stdout, "%.4f\t\t%.4f\t\t%.2f\n", tm_mod_chol, tm_mod_fact,
        tm_mod_fact/tm_mod_chol*100 );
#endif
}
```

A.7. **matmult.c – matrix multiplication.**

```
/*
 * Algorithms implementing unblocked and blocked matrix multiplication
 * (and addition), C = C + A*B.  Unblocked algorithms include: dot (inner)
 * product method, SAXPY operation, loop unrolling and software pipelining.
 * Blocked algorithms include: simple blocking, contiguous blocking, and
 * recursive contiguous blocking.  Also, a function wrapper facilitates calling
 * BLAS matrix multiplication routine DGEMM.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

#include "matmult.h"
#include "lapack.h"
#include "matcom.h"

static void multiply_kernel( const double *A, const double *B, double *C );
static void multiply_rect_kernel( int m, int n, int p,
    const double *A, const double *B, double *C );
static void multiply_blk_ker( int m, int n, int p, int ldimA, const double *A,
    int ldimB, const double *B, int ldimC, double *C );
static void multiply_rect_blk_ker( int m, int n, int p, int ldimA, const double *A,
    int ldimB, const double *B, int ldimC, double *C );

/***************************************************************************/

/*
 * Determines optimal block dimension for the local environment given the matrix
 * leading dimension.  Also, it facilitates the use of a different block
 * dimension for testing (debugging).  If the leading dimension is less than the
 * optimal block dimension, the block dimension is set to the leading dimension,
 * and the matrix computation becomes an unblocked algorithm.
 */
int get_block_dim_mmult( int ldim )
{
    int bdim;

#if defined(DEBUG)
    bdim = BDIM;
#else
    bdim = BDIM;
#endif
    if ( bdim <= 1 || bdim > ldim ) {
        bdim = ldim;
    }
    return bdim;
}
```

```
/*
 * Performs matrix multiplication and addition, C = C + A*B, using the SAXPY
 * operation -- jki indexing.  A, B and C are contiguous KDIM-by-KDIM sub-blocks
 * stored in column-major order.  Looping is controlled by a symbolic constant
 * (KDIM), which is evaluated during compilation.
 */
void multiply_kernel( const double *A, const double *B, double *C )
{
    for ( int j = 0; j < KDIM; j++ ) {
        const double *B_j = B + j*KDIM;             // Points to element B(0,j)
        double *C_j = C + j*KDIM;                   // Points to element C(0,j)
        for ( int k = 0; k < KDIM; k++ ) {
            const double *A_k = A + k*KDIM;         // Points to element A(0,k)
            double bkj = *(B_j + k);                // Element B(k,j)
            for ( int i = 0; i < KDIM; i++ ) {
                *(C_j + i) += *(A_k + i) * bkj;     // C(i,j) += A(i,k) * B(k,j)
            }
        }
    }
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using the SAXPY
 * operation -- jki indexing.  A (m-by-p), B (p-by-n) and C (m-by-n) are
 * rectangular contiguous sub-blocks stored in column-major order with leading
 * dimension KDIM.  Looping is controlled by variables, which are evaluated at
 * run time.
 */
void multiply_rect_kernel( int m, int n, int p,
    const double *A, const double *B, double *C )
{
    for ( int j = 0; j < n; j++ ) {
        const double *B_j = B + j*KDIM;             // Points to element B(0,j)
        double *C_j = C + j*KDIM;                   // Points to element C(0,j)
        for ( int k = 0; k < p; k++ ) {
            const double *A_k = A + k*KDIM;         // Points to element A(0,k)
            double bkj = *(B_j + k);                // Element B(k,j)
            for ( int i = 0; i < m; i++ ) {
                *(C_j + i) += *(A_k + i) * bkj;     // C(i,j) += A(i,k) * B(k,j)
            }
        }
    }
}

/*
 * Performs (block) matrix multiplication and addition, C = C + A*B, using the
 * SAXPY operation -- jki indexing.  A (m-by-p), B (p-by-n) and C (m-by-n) are
 * contiguous matrix blocks with leading dimensions ldimA, ldimB and ldimC,
 * respectively.  Within blocks of A, B and C, sub-blocks of size KDIM*KDIM are
 * stored contiguously.  Matrix multiplication and addition is ultimately
 * performed on KDIM-by-KDIM sub-blocks with looping controlled by a symbolic
 * constant.
 */
void multiply_blk_ker( int m, int n, int p, int ldimA, const double *A,
    int ldimB, const double *B, int ldimC, double *C )
{
    for ( int j = 0; j < n; j += KDIM ) {

        for ( int k = 0; k < p; k += KDIM ) {
            // Set pointer to kernel (sub-block) Bkj
            const double *Bkj = B + k*KDIM + j*ldimB;

            for ( int i = 0; i < m; i += KDIM ) {
                // Set pointers to kernels (sub-blocks) Aik and Cij
                const double *Aik = A + i*KDIM + k*ldimA;
                double *Cij = C + i*KDIM + j*ldimC;
                // Perform matrix multiplication on kernels (sub-blocks)
                multiply_kernel( Aik, Bkj, Cij );
            }
        }
    }
}

/*
 * Performs (block) matrix multiplication and addition, C = C + A*B, using the
 * SAXPY operation -- jki indexing.  A (m-by-p), B (p-by-n) and C (m-by-n) are
 * contiguous matrix blocks with leading dimensions ldimA, ldimB and ldimC,
 * respectively.  Within blocks of A, B and C, sub-blocks of size KDIM*KDIM are
 * stored contiguously.  Matrix multiplication and addition is ultimately
 * performed on r-by-s, r-by-t and t-by-s sub-blocks with looping controlled
 * by variables.
 */
void multiply_rect_blk_ker( int m, int n, int p, int ldimA, const double *A,
    int ldimB, const double *B, int ldimC, double *C )
{
    for ( int j = 0; j < n; j += KDIM ) {
        // Determine number of columns in (i,j)th sub-block of C
        const int s = (j + KDIM > n) ? (n - j) : KDIM;

        for ( int k = 0; k < p; k += KDIM ) {
            // Determine number of columns of Aik and rows of Bkj
            const int t = (k + KDIM > p) ? (p - k) : KDIM;
            // Set pointer to kernel (sub-block) Bkj
            const double *Bkj = B + k*KDIM + j*ldimB;

            for ( int i = 0; i < m; i += KDIM ) {
                // Determine number of rows in (i,j)th sub-block of C
                const int r = (i + KDIM > m) ? (m - i) : KDIM;
                // Set pointers to kernels (sub-blocks) Aik and Cij
                const double *Aik = A + i*KDIM + k*ldimA;
                double *Cij = C + i*KDIM + j*ldimC;
                // Perform matrix multiplication on kernels (sub-blocks)
                multiply_rect_kernel( r, s, t, Aik, Bkj, Cij );
            }
        }
    }
}

/*************************************************************************/

/*
 * Performs matrix multiplication and addition, C = C + A*B, using the dot
 * (inner) product method -- ijk indexing.  For each element C(i,j), the
 * inner-most loop computes the dot product of row i of A with column j of B,
 * and adds the result to C(i,j).  A, B and C are n-by-n matrices stored in
 * column-major order with leading dimension n.
 */
void mmult_dot_product( int n, const double *A, const double *B, double *C )
{
    const int ldim = n;

    for ( int i = 0; i < n; i++ ) {
        const double *Ai_ = A + i;              // Points to element A(i,0)
        for ( int j = 0; j < n; j++ ) {
            const double *B_j = B + j*ldim;     // Points to element B(0,j)
            double cij = *(C + j*ldim + i);     // Element C(i,j)
            for ( int k = 0; k < n; k++ ) {
                cij += *(Ai_ + k*ldim) * *(B_j + k);
            }                                   // C(i,j) += A(i,k) * B(k,j)
            *(C + j*ldim + i) = cij;
        }
    }
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using the SAXPY
 * operation -- jki indexing.  The inner-most loop adds a scalar multiple of a
 * column to another column.  A, B and C are n-by-n matrices stored in
 * column-major order with leading dimension n.
 */
void mmult_saxpy( int n, const double *A, const double *B, double *C )
{
    const int ldim = n;

    multiply_matrix( n, n, n, ldim, A, ldim, B, ldim, C );
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using the dot
 * (inner) product method -- ijk indexing.  Optimizes floating point
 * operations by unrolling the inner-most loop to a depth of 8.  A, B and C are
 * n-by-n matrices stored in column-major order with leading dimension n.
 */
void mmult_unroll( int n, const double *A, const double *B, double *C )
{
    const int ldim = n;

    for ( int i = 0; i < n; i++ ) {
        const double *Ai_ = A + i;              // Points to element A(i,0)
        for ( int j = 0; j < n; j++ ) {
            const double *B_j = B + j*ldim;     // Points to element B(0,j)
            double  cij = *(C + j*ldim + i),
                    c0 = 0, c1 = 0, c2 = 0, c3 = 0,
                    c4 = 0, c5 = 0, c6 = 0, c7 = 0;
            int k = 0;
            for ( ; k < n-7; k += 8 ) {
                c0 += *(Ai_ + (k+0)*ldim) * *(B_j + (k+0));
                c1 += *(Ai_ + (k+1)*ldim) * *(B_j + (k+1));
                c2 += *(Ai_ + (k+2)*ldim) * *(B_j + (k+2));
                c3 += *(Ai_ + (k+3)*ldim) * *(B_j + (k+3));
                c4 += *(Ai_ + (k+4)*ldim) * *(B_j + (k+4));
                c5 += *(Ai_ + (k+5)*ldim) * *(B_j + (k+5));
                c6 += *(Ai_ + (k+6)*ldim) * *(B_j + (k+6));
                c7 += *(Ai_ + (k+7)*ldim) * *(B_j + (k+7));
            }
            cij += c0 + c1 + c2 + c3 + c4 + c5 + c6 + c7;
            // Finish up dot product computation for cases where matrix
            // dimension n is not a multiple of the depth of loop unrolling
            for ( ; k < n; k++ ) {
                cij += *(Ai_ + k*ldim) * *(B_j + k);
            }
            *(C + j*ldim + i) = cij;            // Element C(i,j)
        }
    }
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using the SAXPY
 * operation -- jki indexing.  Optimizes floating point operations through
 * software pipelining with the inner-most loop unrolled to a depth of 8.
 * A, B and C are n-by-n matrices stored in column-major order with leading
 * dimension n.
 */
void mmult_pipeline( int n, const double *A, const double *B, double *C )
{
    const int ldim = n;

    for ( int j = 0; j < n; j++ ) {
        const double *B_j = B + j*ldim;         // Points to element B(0,j)
```

```
double *C_j = C + j*ldim;                  // Points to element C(0,j)
for ( int k = 0; k < n; k++ ) {
    const double *A_k = A + k*ldim;        // Points to element A(0,k)
    double  bkj,
            a0, a1, a2, a3, a4, a5, a6, a7,
            c0, c1, c2, c3, c4, c5, c6, c7;
    int i = 0;
    bkj = *(B_j + k);                      // Element B(k,j)
    if ( n > 7+8 ) {                       // Proceed with software pipelining
        a0 = *(A_k + 0);
        a1 = *(A_k + 1);
        a2 = *(A_k + 2);
        a3 = *(A_k + 3);
        a4 = *(A_k + 4);
        a5 = *(A_k + 5);
        a6 = *(A_k + 6);
        a7 = *(A_k + 7);
        c0 = *(C_j + 0) + a0 * bkj;
        c1 = *(C_j + 1) + a1 * bkj;
        c2 = *(C_j + 2) + a2 * bkj;
        c3 = *(C_j + 3) + a3 * bkj;
        c4 = *(C_j + 4);
        c5 = *(C_j + 5);
        c6 = *(C_j + 6);
        c7 = *(C_j + 7);

        for (; i < n-7-8; i += 8) {
            *(C_j + i + 0) = c0;
            a4 *= bkj;
            a0 = *(A_k + i + 8);
            c4 += a4;
            c0 = *(C_j + i + 8);

            *(C_j + i + 1) = c1;
            a5 *= bkj;
            a1 = *(A_k + i + 9);
            c5 += a5;
            c1 = *(C_j + i + 9);

            *(C_j + i + 2) = c2;
            a6 *= bkj;
            a2 = *(A_k + i + 10);
            c6 += a6;
            c2 = *(C_j + i + 10);

            *(C_j + i + 3) = c3;
            a7 *= bkj;
            a3 = *(A_k + i + 11);
            c7 += a7;
            c3 = *(C_j + i + 11);
```

```
                    *(C_j + i + 4) = c4;
                    a0 *= bkj;
                    a4 = *(A_k + i + 12);
                    c0 += a0;
                    c4 = *(C_j + i + 12);

                    *(C_j + i + 5) = c5;
                    a1 *= bkj;
                    a5 = *(A_k + i + 13);
                    c1 += a1;
                    c5 = *(C_j + i + 13);

                    *(C_j + i + 6) = c6;
                    a2 *= bkj;
                    a6 = *(A_k + i + 14);
                    c2 += a2;
                    c6 = *(C_j + i + 14);

                    *(C_j + i + 7) = c7;
                    a3 *= bkj;
                    a7 = *(A_k + i + 15);
                    c3 += a3;
                    c7 = *(C_j + i + 15);
                }
                *(C_j + i + 0) = c0;
                *(C_j + i + 1) = c1;
                *(C_j + i + 2) = c2;
                *(C_j + i + 3) = c3;
                *(C_j + i + 4) = c4 + a4 * bkj;
                *(C_j + i + 5) = c5 + a5 * bkj;
                *(C_j + i + 6) = c6 + a6 * bkj;
                *(C_j + i + 7) = c7 + a7 * bkj;
                i += 8;
            }
            // Finish up combined scalar multiplication and vector addition for
            // cases where matrix dimension n is not a multiple of the depth of
            // loop unrolling
            for ( ; i < n; i++ ) {
                *(C_j + i) += *(A_k + i) * bkj;
            }
        }
    }
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using simple
 * blocking to optimize memory access.  The underlying unblocked matrix
 * multiplication algorithm is the SAXPY operation.  A, B and C are n-by-n
 * matrices stored in column-major order with leading dimension n.
 */
void mmult_block( int n, const double *A, const double *B, double *C )
{
    const int ldim = n;
    const int bdim = get_block_dim_mmult( ldim );

    for ( int j = 0; j < n; j += bdim ) {
        // Determine number of columns in (i,j)th block of C
        int s = (j + bdim > n) ? (n - j) : bdim;

        for ( int k = 0; k < n; k += bdim ) {
            // Determine number of columns of Aik and rows of Bkj
            int t = (k + bdim > n) ? (n - k) : bdim;
            // Set pointer to block matrix Bkj
            const double *Bkj = B + k + j*ldim;

            for ( int i = 0; i < n; i += bdim ) {
                // Determine number of rows in (i,j)th block of C
                int r = (i + bdim > n) ? (n - i) : bdim;
                // Set pointers to block matrices Aik and Cij
                const double *Aik = A + i + k*ldim;
                double *Cij = C + i + j*ldim;
                // Perform multiplication on block matrices
                multiply_matrix( r, s, t, ldim, Aik, ldim, Bkj, ldim, Cij );
            }
        }
    }
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using contiguous
 * blocking to optimize memory access. The underlying unblocked matrix
 * multiplication algorithm is the SAXPY operation. A, B and C are n-by-n
 * matrices stored in column-major order with leading dimension n. A, B, and C
 * are first copied to arrays AA, BB and CC, respectively, where bdim-by-bdim
 * matrix blocks are stored contiguously, and within each block, elements are
 * stored in column-major order. Matrix multiplication and addition on
 * contiguous blocks yields CC = CC + AA*BB, and the result is copied from array
 * CC to C, where matrix elements are stored in conventional column-major order.
 */
void mmult_contig_block( int n, const double *A, const double *B, double *C )
{
    const int   ldim = n;
    const int   bdim = get_block_dim_mmult( ldim );

    double  *AA, *BB, *CC;

    AA = (double *) malloc( ldim*ldim*sizeof(double) );
    BB = (double *) malloc( ldim*ldim*sizeof(double) );
    CC = (double *) malloc( ldim*ldim*sizeof(double) );
    form_contig_blocks( n, n, ldim, A, n, n, bdim, ldim, AA );
    form_contig_blocks( n, n, ldim, B, n, n, bdim, ldim, BB );
    form_contig_blocks( n, n, ldim, C, n, n, bdim, ldim, CC );

    for ( int j = 0; j < n; j += bdim ) {
        // Determine number of columns in (i,j)th block of CC
        int s = (j + bdim > n) ? (n - j) : bdim;

        for ( int k = 0; k < n; k += bdim ) {
            // Determine number of columns of AAik and rows of BBkj
            int t = (k + bdim > n) ? (n - k) : bdim;
            // Set pointer to block matrix BBkj
            double *BBkj = BB + k*s + j*ldim;

            for ( int i = 0; i < n; i += bdim ) {
                // Determine number of rows in (i,j)th block of CC
                int r = (i + bdim > n) ? (n - i) : bdim;
                // Set pointers to block matrices AAik and CCij
                double *AAik = AA + i*t + k*ldim;
                double *CCij = CC + i*s + j*ldim;
                // Perform multiplication on block matrices
                multiply_matrix( r, s, t, r, AAik, t, BBkj, r, CCij );
            }
        }
    }
    // Extract matrix C from contiguous blocks
    unpack_contig_blocks( n, n, bdim, ldim, CC, n, n, ldim, C );
    free( AA );
    free( BB );
    free( CC );
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using recursive
 * contiguous blocking to optimize memory access.  The underlying unblocked
 * matrix multiplication algorithm is the SAXPY operation.  A, B and C are
 * n-by-n matrices stored in column-major order with leading dimension n.
 * A, B and C are first copied to arrays AA, BB and CC, respectively, where
 * bdim-by-bdim matrix blocks are stored contiguously, and within each block,
 * sub-blocks of size KDIM*KDIM are stored contiguously.  Matrix multiplication
 * and addition on recursive contiguous blocks yields CC = CC + AA*BB, and the
 * result is copied from array CC to C, where matrix elements are stored in
 * conventional column-major order.  Ultimately, a symbolic constant (KDIM)
 * controls looping in the matrix multiplication kernel.
 */
void mmult_recur_block( int n, const double *A, const double *B, double *C )
{
    const int   nn = (n / KDIM) * KDIM + ((n % KDIM) ? KDIM : 0);
    const int   ldim = nn;
    const int   bdim = get_block_dim_mmult( ldim );
    double  *AA, *BB, *CC;

    AA = (double *) malloc( ldim*ldim*sizeof(double) );
    BB = (double *) malloc( ldim*ldim*sizeof(double) );
    CC = (double *) malloc( ldim*ldim*sizeof(double) );
    form_recur_blocks( n, n, n, A, nn, nn, KDIM, bdim, ldim, AA );
    form_recur_blocks( n, n, n, B, nn, nn, KDIM, bdim, ldim, BB );
    form_recur_blocks( n, n, n, C, nn, nn, KDIM, bdim, ldim, CC );

    for ( int j = 0; j < nn; j += bdim ) {
        // Determine number of columns in (i,j)th block of CC
        int s = (j + bdim > n) ? (n - j) : bdim;
        int v = (j + bdim > nn) ? (nn - j) : bdim;

        for ( int k = 0; k < nn; k += bdim ) {
            // Determine number of columns of AAik and rows of BBkj
            int t = (k + bdim > n) ? (n - k) : bdim;
            int w = (k + bdim > nn) ? (nn - k) : bdim;
            // Set pointer to block matrix BBkj
            double *BBkj = BB + k*v + j*ldim;

            for ( int i = 0; i < nn; i += bdim ) {
                // Determine number of rows in (i,j)th block of CC
                int r = (i + bdim > n) ? (n - i) : bdim;
                int u = (i + bdim > nn) ? (nn - i) : bdim;
                // Set pointers to block matrices AAik and CCij
                double *AAik = AA + i*w + k*ldim;
                double *CCij = CC + i*v + j*ldim;
                // Perform multiplication on recursive block matrices
                multiply_blk_ker( r, s, t, u, AAik, w, BBkj, u, CCij );
            }
        }
    }
    // Extract matrix C from recursive contiguous blocks
    unpack_recur_blocks( nn, nn, KDIM, bdim, ldim, CC, n, n, n, C );
    free( AA );
    free( BB );
    free( CC );
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using recursive
 * contiguous blocking to optimize memory access.  The underlying unblocked
 * matrix multiplication algorithm is the SAXPY operation.  A, B and C are
 * n-by-n matrices stored in column-major order with leading dimension n.
 * A, B and C are first copied to arrays AA, BB and CC, respectively, where
 * bdim-by-bdim matrix blocks are stored contiguously, and within each block,
 * sub-blocks of size KDIM*KDIM are stored contiguously.  Matrix multiplication
 * and addition on recursive contiguous blocks yields CC = CC + AA*BB, and the
 * result is copied from array CC to C, where matrix elements are stored in
 * conventional column-major order.  Ultimately, variables control looping in
 * the matrix multiplication kernel.
 */
void mmult_rect_recur_block( int n, const double *A, const double *B, double *C )
{
    const int   nn = (n / KDIM) * KDIM + ((n % KDIM) ? KDIM : 0);
    const int   ldim = nn;
    const int   bdim = get_block_dim_mmult( ldim );

    double  *AA, *BB, *CC;

    AA = (double *) malloc( ldim*ldim*sizeof(double) );
    BB = (double *) malloc( ldim*ldim*sizeof(double) );
    CC = (double *) malloc( ldim*ldim*sizeof(double) );
    form_recur_blocks( n, n, n, A, nn, nn, KDIM, bdim, ldim, AA );
    form_recur_blocks( n, n, n, B, nn, nn, KDIM, bdim, ldim, BB );
    form_recur_blocks( n, n, n, C, nn, nn, KDIM, bdim, ldim, CC );

    for ( int j = 0; j < nn; j += bdim ) {
        // Determine number of columns in (i,j)th block of CC
        int s = (j + bdim > n) ? (n - j) : bdim;
        int v = (j + bdim > nn) ? (nn - j) : bdim;

        for ( int k = 0; k < nn; k += bdim ) {
            // Determine number of columns of AAik and rows of BBkj
            int t = (k + bdim > n) ? (n - k) : bdim;
            int w = (k + bdim > nn) ? (nn - k) : bdim;
            // Set pointer to block matrix BBkj
            double *BBkj = BB + k*v + j*ldim;

            for ( int i = 0; i < nn; i += bdim ) {
                // Determine number of rows in (i,j)th block of CC
                int r = (i + bdim > n) ? (n - i) : bdim;
                int u = (i + bdim > nn) ? (nn - i) : bdim;
                // Set pointers to block matrices AAik and CCij
                double *AAik = AA + i*w + k*ldim;
                double *CCij = CC + i*v + j*ldim;
                // Perform multiplication on recursive block matrices
                multiply_rect_blk_ker( r, s, t, u, AAik, w, BBkj, u, CCij );
            }
        }
    }
    // Extract matrix C from recursive contiguous blocks
    unpack_recur_blocks( nn, nn, KDIM, bdim, ldim, CC, n, n, n, C );
    free( AA );
    free( BB );
    free( CC );
}
```

```
/*
 * Wrapper for calling BLAS routine DGEMM, which performs matrix multiplication.
 */
void mmult_blas( int n, const double *A, const double *B, double *C )
{
    const char      no_trans = 'N';
    const double    one = 1.0;

    dgemm_(&no_trans, &no_trans, &n, &n, &n, &one, A, &n, B, &n, &one, C, &n);
}
```

A.8. **matmultp.c – parallel matrix multiplication.**

```
/*
 * Algorithms implementing parallel matrix multiplication (and addition),
 * C = C + A*B, using the MPI (Message-Passing Interface) library.  Includes
 * functions that facilitate interprocess communication, and Fox's algorithm
 * for parallel matrix multiplication.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <mpi.h>

#include "matmultp.h"

static void form_contig_blocks( int m, int n, int ldimE, const double *E,
    int mm, int nn, int bdim, int ldimF, double *F );
static void unpack_contig_blocks( int mm, int nn, int bdim, int ldimE,
    const double *E, int m, int n, int ldimF, double *F  );

static MPI_Datatype contig;

static double   *AA, *BB, *CC;
static double   *X, *Y, *Z;

/****************************************************************************/

/*
 * Copy elements of an m-by-n matrix E to mm-by-nn matrix F.  Leading dimension
 * ldimE is the number of rows of E, while ldimF is the number of columns of F.
 * Elements of E are stored in column-major order.  Array F stores bdim-by-bdim
 * matrix blocks contiguously, with blocks stored in row-major order and
 * elements of blocks stored in column-major order.
 */
void form_contig_blocks( int m, int n, int ldimE, const double *E,
    int mm, int nn, int bdim, int ldimF, double *F )
{
    for ( int i = 0; i < mm; i += bdim ) {
        int r = (i + bdim > m) ? (m - i) : bdim;
        int p = (i + bdim > mm) ? (mm - i) : bdim;
        for ( int j = 0; j < nn; j += bdim ) {
            int s = (j + bdim > n) ? (n - j) : bdim;
            int q = (j + bdim > nn) ? (nn - j) : bdim;
            // Clear fringe blocks by setting elements to zero
            if ( s != q || r != p ) {
                double *Fij = F + i*ldimF + j*p;
                memset( Fij, 0, p*q*sizeof(double) );
            }
            for ( int k = 0; k < s; k++ ) {
                const double *Eij = E + j*ldimE + i + k*ldimE;
                double *Fij = F + i*ldimF + j*p + k*p;
                memcpy( Fij, Eij, r*sizeof(double) );
            }
        }
    }
}

/*
 * Copies elements of an mm-by-nn matrix E to m-by-n matrix F. Leading dimension
 * ldimE is the number of columns of E, while ldimF is the number of rows of F.
 * Array E stores bdim-by-bdim matrix blocks contiguously, with blocks stored in
 * row-major order and elements of blocks stored in column-major order.  As
 * matrix E is copied to array F, elements are unpacked and stored in
 * conventional column-major order.
 */
void unpack_contig_blocks( int mm, int nn, int bdim, int ldimE,
    const double *E, int m, int n, int ldimF, double *F  )
{
    for ( int i = 0; i < mm; i += bdim ) {
        int r = (i + bdim > m) ? (m - i) : bdim;
        int p = (i + bdim > mm) ? (mm - i) : bdim;
        for ( int j = 0; j < nn; j += bdim ) {
            int s = (j + bdim > n) ? (n - j) : bdim;
            for ( int k = 0; k < s; k++ ) {
                const double *Eij = E + i*ldimE + j*p + k*p;
                double *Fij = F + j*ldimF + i + k*ldimF;
                memcpy( Fij, Eij, r*sizeof(double) );
            }
        }
    }
}

/*****************************************************************************/

/*
 * Generates a random m-by-n matrix with leading dimension m.  The uniform
 * randomly generated elements are scaled by factor alpha.
 */
void create_random_matrix( double alpha, int m, int n, double *E )
{
    const int ldim = m;

    for ( int j = 0; j < n; j++ ) {
        for ( int i = 0; i < m; i++ ) {
            E[j*ldim + i] = alpha * drand48() - (0.50 * alpha);
        }
    }
}
```

```
/*
 * Sets elements of m-by-n matrix with leading dimension m to zero.
 */
void clear_matrix( int m, int n, double *E )
{
    const int ldim = m;

    for ( int j =  0; j < n; j++ ) {
        memset( E + j*ldim, 0, m*sizeof(double) );
    }

}

/*
 * Copies the elements of an m-by-n matrix E to matrix F.  For both matrices
 * the leading dimension is m, and elements are stored in column-major order.
 */
void copy_matrix( int m, int n, const double *E, double *F )
{
    const int ldim = m;

    for ( int j = 0; j < n; j++ ) {
        const double *E_j = E + j*ldim;
        double *F_j = F + j*ldim;
        memcpy( F_j, E_j, m*sizeof(double) );
    }
}

/*
 * Establish the Cartesian (square grid) topology that facilitates collective
 * communication between processes.  It is assumed that the number of
 * processes is a perfect square, so that the row dimension equals the column
 * dimension of the grid.
 */
void setup_mpi_grid( struct mpi_grid *grid )
{
    int dims, reord, world_rank;
    int dim[2],
        wrap[2],
        coords[2],
        free_coords[2];

    MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    dims = 2;                          // Number of grid dimensions
    grid->q = (int) sqrt(grid->p);     // Assume p is a perfect square
    dim[0] = grid->q;                  // Number of row blocks
    dim[1] = grid->q;                  // Number of column blocks
    wrap[0] = 0;
    wrap[1] = 1;                    // Wrap around or circular shift for columns
    reord = 1;                      // Permit reordering of processes

    // Create communicator with Cartesian topology
    MPI_Cart_create( MPI_COMM_WORLD, dims, dim, wrap, reord, &(grid->comm) );
    // Get process rank in Cartesian communicator
    MPI_Comm_rank( grid->comm, &(grid->rank) );
    // Get process coordinates in Cartesian communicator
    MPI_Cart_coords( grid->comm, grid->rank, dims, coords );
    grid->row = coords[0];
    grid->col = coords[1];

    // Setup row communicators
    free_coords[0] = 0;             // Fix row coordinates
    free_coords[1] = 1;             // Vary column coordinates
    MPI_Cart_sub( grid->comm, free_coords, &(grid->row_comm) );
    // Setup column communicators
    free_coords[0] = 1;             // Vary row coordinates
    free_coords[1] = 0;             // Fix column coordinates
    MPI_Cart_sub( grid->comm, free_coords, &(grid->col_comm) );
}

/*
 * The full matrices stored on the root processor are partitioned into square
 * blocks corresponding to the processes in the Cartesian grid.  The root
 * processor sends (scatters) the blocks to their respective processors
 * where (block) matrix multiplication is performed.
 */
void scatter_blocks( int bdim, int n,
        const double *A, const double *B, double *C, struct mpi_grid *grid )
{
    const int    nn = bdim * grid->q;
    const int    ldim = nn;
    const long   blk_sz = bdim * bdim;

    // On root processor copy matrix blocks into array so that elements are
    // stored contiguously to facilitate collective communication
    if ( grid->rank == 0 ) {
        AA = (double *) malloc( grid->p*blk_sz*sizeof(double) );
        form_contig_blocks( n, n, n, A, nn, nn, bdim, ldim, AA );
        BB = (double *) malloc( grid->p*blk_sz*sizeof(double) );
        form_contig_blocks( n, n, n, B, nn, nn, bdim, ldim, BB );
        CC = (double *) malloc( grid->p*blk_sz*sizeof(double) );
        form_contig_blocks( n, n, n, C, nn, nn, bdim, ldim, CC );
    }

    // Allocate memory for matrix blocks on processors
    X = (double *) malloc( blk_sz*sizeof(double) );
    Y = (double *) malloc( blk_sz*sizeof(double) );
    Z = (double *) malloc( blk_sz*sizeof(double) );
    // Distribute matrix blocks across processors in communicator
    MPI_Scatter( AA, bdim, contig, X, bdim, contig, 0, grid->comm );
    MPI_Scatter( BB, bdim, contig, Y, bdim, contig, 0, grid->comm );
    MPI_Scatter( CC, bdim, contig, Z, bdim, contig, 0, grid->comm );
}

/*
 * Once the processes have completed (block) matrix multiplication, the root
 * processor receives (gathers) the resulting block matrices from their
 * respective processors within the Cartesian grid, and assembles the blocks
 * into matrix C, which is the result of the matrix multiplication operation
 * C = C + A*B.
 */
void gather_blocks( int bdim, int n, double *C, struct mpi_grid *grid )
{
    const int   nn = bdim * grid->q;
    const int   ldim = nn;

    // Collect matrix blocks from processors in communicator
    MPI_Gather( Z, bdim, contig, CC, bdim, contig, 0, grid->comm );
    free( X );
    free( Y );
    free( Z );

    // On root processor copy contiguous matrix blocks into array where elements
    // are stored in column-major order
    if (grid->rank == 0) {
        unpack_contig_blocks( nn, nn, bdim, ldim, CC, n, n, n, C );
        free( AA );
        free( BB );
        free( CC );
    }
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using the SAXPY
 * operation -- jki indexing.  The inner-most loop adds a scalar multiple of
 * column vector x to column vector y.  A (m-by-p), B (p-by-n) and C (m-by-n)
 * are rectangular matrices stored in column-major order with leading dimensions
 * ldimA, ldimB and ldimC, respectively.
 */
void multiply_matrix( int m, int n, int p, int ldimA, const double *A,
    int ldimB, const double *B, int ldimC, double *C )
{
    for ( int j = 0; j < n; j++ ) {
        const double *B_j = B + j*ldimB;            // Points to element B(0,j)
        double *C_j = C + j*ldimC;                  // Points to element C(0,j)
        for ( int k = 0; k < p; k++ ) {
            const double *A_k = A + k*ldimA;        // Points to element A(0,k)
            double bkj = *(B_j + k);                    // Element B(k,j)
            for ( int i = 0; i < m; i++ ) {
                *(C_j + i) += *(A_k + i) * bkj;
            }                                           // C(i,j) += A(i,k) * B(k,j)
        }
    }
}

/*
 * Performs matrix multiplication and addition, C = C + A*B, using simple
 * blocking to optimize memory access.  The underlying unblocked matrix
 * multiplication algorithm is the SAXPY operation.  A (m-by-p), B (p-by-n)
 * and C (m-by-n) are rectangular matrices stored in column-major order with
 * leading dimensions ldimA, ldimB and ldimC, respectively.
 */
void blocked_matrix_multiply ( int m, int n, int p, int ldimA, const double *A,
    int ldimB, const double *B, int ldimC, double *C )
{
    const int bdim = BDIM;

    for ( int j = 0; j < n; j += bdim ) {
        // Determine number of columns in (i,j)th block of C
        int s = (j + bdim > n) ? (n - j) : bdim;

        for ( int k = 0; k < p; k += bdim ) {
            // Determine number of columns of Aik and rows of Bkj
            int t = (k + bdim > p) ? (p - k) : bdim;
            // Set pointer to block matrix Bkj
            const double *Bkj = B + k + j*ldimB;

            for ( int i = 0; i < m; i += bdim ) {
                // Determine number of rows in (i,j)th block of C
                int r = (i + bdim > m) ? (m - i) : bdim;
                // Set pointers to block matrices Aik and Cij
                const double *Aik = A + i + k*ldimA;
                double *Cij = C + i + j*ldimC;
                // Perform multiplication on block matrices
                multiply_matrix( r, s, t, ldimA, Aik, ldimB, Bkj, ldimC, Cij );
            }
        }
    }
}

/*
 * Performs serial matrix multiplication and addition, C = C + A*B, using
 * simple blocking to optimize memory access.  A, B and C are n-by-n matrices
 * stored in column-major order with leading dimension n.
 */
void serial_matrix_multiply( int n, const double *A, const double *B, double *C )
{
    const int ldim = n;

    blocked_matrix_multiply( n, n, n, ldim, A, ldim, B, ldim, C );
}

/*
 * Implements the memory efficient Fox algorithm for parallel matrix
 * multiplication.  The number of stages is equal to the square root of the
 * number of processes i.e., the number of row/ column blocks.  For each stage
 * a block matrix Aik is broadcast across processes in each row communicator,
 * block matrix multiplication, Cij = Cij + Aik*Bkj, is performed, and block
 * matrices Bkj are rolled bewteen processes within each column communicator.
 */
void fox_matrix_multiply( const int n,
        double *A, double *B, double *C, struct mpi_grid *grid )
{
    const int       ldim = n;
    const long      blk_sz = n * n;

    int             src, dest;
    double          *T;
    MPI_Status      stat;

    // Allocate memory for temporary matrix block
    T = (double *) malloc( blk_sz*sizeof(double) );
    // Determine row index of source processor from which block matrix is
    // received, and row index of destination processor to which block matrix
    // is sent, for next block matrix multiplication operation
    src = (grid->row + 1) % grid->q;
    dest = (grid->row + grid->q - 1) % grid->q;

    // Number of stages = number row blocks = number of column blocks.
    // For each stage broadcast block matrix Aik across processors in row
    // communicator, perform block matrix multiplication, Cij = Cij + Aik*Bkj,
    // and then roll block matrices Bkj between processors within column
    // communicator.
    for ( int stage = 0; stage < grid->q; stage++ ) {
        int bcast_root = (grid->row + stage) % grid->q;
        if ( bcast_root == grid->col ) {
            MPI_Bcast( A, n, contig, bcast_root, grid->row_comm );
            blocked_matrix_multiply( n, n, n, ldim, A, ldim, B, ldim, C );
        } else {
            MPI_Bcast( T, n, contig, bcast_root, grid->row_comm );
            blocked_matrix_multiply( n, n, n, ldim, T, ldim, B, ldim, C );
        }
        MPI_Sendrecv_replace( B, n, contig, dest, 0, src, 0,
            grid->col_comm, &stat );
    }
}
```

```
/*
 * Perform matrix multiplication (and addition), C = C + A*B, using parallel
 * programming with MPI.  First, matrices on the root processor are partitioned
 * into blocks and sent to processes in the Cartesian grid topology.  Then, the
 * Fox algorithm performs (block) matrix multiplication.  Finally, the root
 * processor receives the resulting block matrices from the processes and
 * assembles the resulting matrix C.
 */
void parallel_matrix_multiply( int n,
        const double *A, const double *B, double *C, struct mpi_grid *grid )
{
    const int    bdim = (n / grid->q) + ((n % grid->q ? 1 : 0));
                                        // Dimension of matrix blocks
    //Create and commit MPI derived datatype constructor
    MPI_Type_contiguous( bdim, MPI_DOUBLE, &contig );
    MPI_Type_commit( &contig );

    scatter_blocks( bdim, n, A, B, C, grid );

    fox_matrix_multiply( bdim, X, Y, Z, grid );

    gather_blocks( bdim, n, C, grid );

}
```

A.9. **mfactime.c – timing harness for matrix factorization.**

```
/*
 * Timing harness for measuring the performance of basic and "optimized"
 * algorithms implementing matrix factorizations on square matrices over a range
 * of dimensions.  Matrix factorizations include LU (Gaussian elimination),
 * standard Cholesky, symmetric indefinite (LDL'), and modified Cholesky
 * (Gill-Murray-Wright and Cheng-Higham algorithms).  Performance data are
 * written to an output file destination.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "lufact.h"
#include "cholfact.h"
#include "ldltfact.h"
#include "modchol.h"
#include "matcom.h"
#include "timing.h"

#if !defined(PROC)
#   define PROC "unknown"
#endif
#if !defined(CORES)
#   define CORES "unknown"
#endif
#if !defined(CLKSPEED)
#   define CLKSPEED "unknown"
#endif
#if !defined(CACHE)
#   define CACHE "unknown"
#endif
#if !defined(COMPILER)
#   define COMPILER "unknown"
#endif
#if !defined(LANGUAGE)
#   define LANGUAGE "default"
#endif
#if !defined(OPTM)
#   define OPTM "default"
#endif
#if !defined(DATADIR)
#   define DATADIR "."       // Current directory ./
#endif

#if defined(DEBUG)
#   define MIN_ITER 4        // Minimum number of iterations of algorithm
#   define MIN_SECS 1.0      // Minimum elapsed time for execution of algorithm
    // Define sizes (dimensions) of square matrices used to measure performance
    const int mat_size[] = { 65, 130, 195, 254 };
#else
#   define MIN_ITER 8
#   define MIN_SECS 2.0
    const int mat_size[] = { 65, 130, 195, 254, 258, 321, 387, 450, 508, 516,
        579, 642, 707, 764, 772, 833, 899, 963, 1021, 1027, 1155, 1278, 1282,
        1411, 1532, 1540, 1666, 1789, 1795, 1921, 2046, 2050 };
#endif
#define SIZES (sizeof(mat_size) / sizeof(int))

static void read_matrix( const char *file, int m, int n, double *A );
static void write_data_file( const char *file, const char *hdr_text,
    const int rows, const int cols, const double *data );
static double time_mfact( void (*mfact)(int n, double *A), int n, double *A );
static double time_mfact_pivot( void (*mfact_piv)(char pivot, int n, int *piv,
    int *ord, double *A), char pivot, int n, int *piv, int *ord, double *A );
static void time_lu( void );
static void time_lu_pivot( void );
static void time_chol( void );
static void profile_chol( void );
static void time_ldlt( void );
static void time_ldlt_indef( void );
static void profile_ldlt( void );
static void time_chol_gmw( void );
static void time_chol_ch( void );
static void time_mod_chol_indef( void );
static void profile_mod_chol( void );

static char     *file_path;

int main()
{
    // Specify file path for output data files
    file_path = (char *) calloc( strlen(DATADIR) + 2, sizeof(char) );
    strcpy( file_path, DATADIR );
    strcat( file_path, "/" );

#if defined(LUFACT)
    time_lu();
#endif

#if defined(LUPIVOT)
    time_lu_pivot();
#endif

#if defined(CHOLFACT)
#if defined(PROFILE)
    profile_chol();
#else
    time_chol();
#endif
#endif

#if defined(LDLTFACT)
#if defined(PROFILE)
    profile_ldlt();
#else
    time_ldlt();
    time_ldlt_indef();
#endif
#endif

#if defined(MODCHOL)
#if defined(PROFILE)
    profile_mod_chol();
#else
    time_chol_gmw();
    time_chol_ch();
    time_mod_chol_indef();
#endif
#endif

    return 0;
}

/**************************************************************************/

/*
 * Reads matrix data in specified file into array A passed in argument list.
 * Matrix A is stored in column-major order.
 */
void read_matrix( const char *file, int m, int n, double *A )
{
    const int ldim = m;

    FILE *fp;

    if ( (fp = fopen(file, "r")) == NULL ) {
        fprintf( stderr, "Error opening file %s.", file );
        exit(-1);
    }
    // Read matrix data from file
    for ( int i = 0; i < m; i++ ) {
        for ( int j = 0; j < n; j++ ) {
            fscanf( fp, "%lg", (A + i + j*ldim) );
        }
    }
    fclose( fp );
}
```

```
/*
 * Writes header text and experimental data to the file specified in the
 * argument list.  Experimental data is enumerated in a matrix stored in
 * column-major order.
 */
void write_data_file( const char *file, const char *hdr_text,
    int rows, int cols, const double *data )
{
    FILE *fp;

    if ( (fp = fopen( file, "w" )) == NULL ) {
        fprintf( stderr, "Error opening file %s.", file );
        exit( -1 );
    }
    // Write header text
    fprintf( fp, "# Processor:\t%s\n", PROC);
    fprintf( fp, "# Cores:\t%s\n", CORES);
    fprintf( fp, "# Clock speed:\t%s\n", CLKSPEED);
    fprintf( fp, "# Cache:\t%s\n", CACHE);
    fprintf( fp, "# \n" );
    fprintf( fp, "# C compiler:\t%s\n", COMPILER );
    fprintf( fp, "# C language standard:\t%s\n", LANGUAGE );
    fprintf( fp, "# Optimization level and options:\t%s\n", OPTM );
    fprintf( fp, "# Clock resolution:\t%Lg\n", timer_resolution() );
    fprintf( fp, "# \n" );
    fprintf( fp, "%s\n", hdr_text );
    // Write experimental data
    for ( int i = 0; i < rows; i++ ) {
        for ( int j = 0; j < cols; j++ ) {
            fprintf( fp, "%g\t", *(data+j*rows+i) );
        }
        fprintf( fp, "\n" );
    }
    fclose( fp );
}

/*
 * Measures the average time (number of seconds) to factor an n-by-n matrix.
 * Assumes that pivoting is not required to ensure numerical stability of the
 * factorization procedure.  Matrix factorization is performed iteratively for
 * at least the minimum number of iterations, and until the minimumn time
 * (in seconds) has elapsed.
 */
double time_mfact( void (*mfact)(int n, double *A), int n, double *A )
{
    struct      timespec sta, end;
    long int    num_iter = MIN_ITER;
    double      secs = -1.0;
    double      *M;
    // Save copy of matrix A before performing matrix factorization
    M = (double *) malloc( n*n*sizeof(double) );
    copy_matrix( n, n, A, M );

    while ( secs < MIN_SECS ) {
        get_time( &sta );
        for ( int i = 0; i < num_iter; i++ ) {
            mfact( n, A );
            copy_matrix( n, n, M, A );      // Reset matrix A to initial value
        }
        get_time( &end );
        secs = timespec_diff( sta, end );
        num_iter *= 2;
    }
    free( M );

    // On exiting the while loop, the number of iterations (num_iter) has been
    // doubled in the event that secs < MIN_SECS, so num_iter must be halved
    return secs / (num_iter/2.0);
}

/*
 * Measures the average time (number of seconds) to factor an n-by-n matrix.
 * Assumes that pivoting is required to ensure numerical stability of the
 * factorization procedure.  Matrix factorization is performed iteratively for
 * at least the minimum number of iterations, and until the minimum time
 * (in seconds) has elapsed.
 */
double time_mfact_pivot( void (*mfact_piv)(char pivot, int n, int *piv,
    int *ord, double *A), char pivot, int n, int *piv, int *ord, double *A )
{
    struct      timespec sta, end;
    long int    num_iter = MIN_ITER;
    double      secs = -1.0;
    double      *M;

    // Save copy of matrix A before performing matrix factorization
    M = (double *) malloc( n*n*sizeof(double) );
    copy_matrix( n, n, A, M );

    while ( secs < MIN_SECS ) {
        get_time( &sta );
        for ( int i = 0; i < num_iter; i++ ) {
            mfact_piv( pivot, n, piv, ord, A );
            copy_matrix( n, n, M, A );      // Reset matrix A to initial value
        }
        get_time( &end );
        secs = timespec_diff( sta, end );
        num_iter *= 2;
    }
    free( M );

    // On exiting the while loop, the number of iterations (num_iter) has been
    // doubled in the event that secs < MIN_SECS, so num_iter must be halved
    return secs / (num_iter/2.0);
}

/*
 * Measures the performance (Mflops/sec) of basic and optimized algorithms
 * implementing LU factorization on nonsingular matrices over a range of
 * dimensions.  The algorithms employ performance optimization techniques
 * including loop reordering and blocking.  It is assumed that LU factorization
 * is performed on matrices with properties -- for example, diagonally dominant
 * (alpha = 1.0) -- that do not require pivoting.
 */
void time_lu( void )
{
#define FIELDS 6            // Number of output data fields
    const char      *data_file_name = "lu.dat";
    const char      *hdr_text =
"# N:       Matrix dimension, N-by-N\n"
"# BDIM:    Block dimension for blocked algorithms\n"
"#          Mflop/sec for Guassian elimination (LU factorization) algorithms\n"
"# OUTPROD: Outer product method, kji indexing\n"
"# SAXPY:   SAXPY operation, jki indexing\n"
"# BLKSIMP: Simple blocking\n"
"# BLKRCR:  Recursive contiguous blocking\n"
"# \n"
"# N\tBDIM\tOUTPROD\tSAXPY\tBLKSIMP\tBLKRCR";
    const int       col_n = 0,
                    col_bdim = 1,
                    col_outprod = 2,
                    col_saxpy = 3,
                    col_blksimp = 4,
                    col_blkrcr = 5;
    const double    alpha = 1.0;    // Scaling factor for random matrix

    char    *data_file;
    int     n, bdim;
    double  mflops;
    double  perf_data[FIELDS*SIZES];
    double  *A;
    void    (*mfact)( int n, double *A );

    // Concatenate file path and name
    data_file = (char *) calloc( strlen(file_path) + strlen(data_file_name) + 1,
        sizeof(char) );
    strcpy( data_file, file_path );
    strcat( data_file, data_file_name );
    for ( int i = 0; i < SIZES; i++ ) {
        n = mat_size[i];
        bdim = get_block_dim_lu( n );
        fprintf( stdout, "n = %d, bdim = %d\n", n, bdim );
        // LU factorization takes (2/3)*n^3 floating point operations
        mflops = 1.0e-06 * (2.0/3.0) * n * n * n;
        // Create random n-by-n nonsingular matrix that is diagonally dominant
        A = (double *) malloc( n*n*sizeof(double) );
        create_random_nonsingular( alpha, n, A );

        // Performance data is stored in perf_data[] array in column-major order
        perf_data[i+col_n*SIZES] = (double) n;
        perf_data[i+col_bdim*SIZES] = (double) bdim;
printf("lu_outer_product\n");
        // Measure performance of LU factorization algorithms:
        // Outer product method (kji indexing)
        mfact = lu_outer_product;
        perf_data[i+col_outprod*SIZES] = mflops / time_mfact( mfact, n, A );
printf("lu_saxpy\n");
        // SAXPY operation (jki indexing)
        mfact = lu_saxpy;
        perf_data[i+col_saxpy*SIZES] = mflops / time_mfact( mfact, n, A );
printf("lu_block\n");
        // Simple blocking
        mfact = lu_block;
        perf_data[i+col_blksimp*SIZES] = mflops / time_mfact( mfact, n, A );
printf("lu_recur_block\n");
        // Recursive contiguous blocking
        mfact = lu_recur_block;
        perf_data[i+col_blkrcr*SIZES] = mflops / time_mfact( mfact, n, A );

        free( A );
    }
    write_data_file( data_file, hdr_text, SIZES, FIELDS, perf_data );

#undef FIELDS
}

/*
 * Measures the performance (Mflops/sec) of basic and optimized algorithms
 * implementing LU factorization with partial pivoting on nonsingular matrices
 * over a range of dimensions.  The algorithms employ performance optimization
 * techniques including loop reordering, blocking and the use of the LAPACK
 * library.
 */
void time_lu_pivot( void )
{
#define FIELDS 6            // Number of output data fields
    const char      *data_file_name = "lu_pivot.dat";
    const char      *hdr_text =
"# N:       Matrix dimension, N-by-N\n"
"# BDIM:    Block dimension for blocked algorithms\n"
"#          Mflop/sec for LU factorization with partial pivoting algorithms\n"
"# OUTPROD: Outer product method, kji indexing\n"
"# SAXPY:   SAXPY operation, jki indexing\n"
"# BLOCK:   Simple blocking\n"
"# LAPACK:  LAPACK routine DGETRF\n"
"# \n"
"# N\tBDIM\tOUTPROD\tSAXPY\tBLOCK\tLAPACK";
    const int       col_n = 0,
                    col_bdim = 1,
                    col_outprod = 2,
                    col_saxpy = 3,
                    col_block = 4,
                    col_lapack = 5;
    const double    alpha = 10.0;   // Scaling factor for random matrix

    char    *data_file;
    int     n, bdim;
    int     *piv, *ord;
    double  mflops;
    double  perf_data[FIELDS*SIZES];
    double  *A;
    void    (*mfact_piv)( char pivot, int n, int *piv, int *ord, double *A );

    // Concatenate file path and name
    data_file = (char *) calloc( strlen(file_path) + strlen(data_file_name) + 1,
        sizeof(char) );
    strcpy( data_file, file_path );
    strcat( data_file, data_file_name );

    for (int i = 0; i < SIZES; i++) {
        n = mat_size[i];
        bdim = get_block_dim_lu( n );
        fprintf( stdout, "n = %d, bdim = %d\n", n, bdim );
        // LU factorization takes (2/3)*n^3 floating point operations
        mflops = 1.0e-06 * (2.0/3.0) * n * n * n;
        // Create random n-by-n nonsingular matrix that is diagonally dominant
        A = (double *) malloc( n*n*sizeof(double) );
        create_random_nonsingular( alpha, n, A );
        // Declare pivot and pivot order vectors
        piv = (int *) malloc( n*sizeof(int) );
        ord = (int *) malloc( n*sizeof(int) );

        // Performance data is stored in perf_data[] array in column-major order
        perf_data[i+col_n*SIZES] = (double) n;
        perf_data[i+col_bdim*SIZES] = (double) bdim;
printf("lu_pivot_outer_product\n");
        // Measure performance of LU factorization with partial pivoting algorithms:
        // Outer product method (kji indexing)
        mfact_piv = lu_pivot_outer_product;
        perf_data[i+col_outprod*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'G', n, piv, ord, A );
printf("lu_pivot_saxpy\n");
        // SAXPY operation (jki indexing)
        mfact_piv = lu_pivot_saxpy;
        perf_data[i+col_saxpy*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'G', n, piv, ord, A );
printf("lu_pivot_block\n");
        // Simple blocking
        mfact_piv = lu_pivot_block;
        perf_data[i+col_block*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'G', n, piv, ord, A );
printf("lu_pivot_lapack\n");
        // LAPACK routine DGETRF
        mfact_piv = lu_pivot_lapack;
        perf_data[i+col_lapack*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'G', n, piv, ord, A );

        free( A );
        free( piv );
        free( ord );
    }
    write_data_file( data_file, hdr_text, SIZES, FIELDS, perf_data );

#undef FIELDS
}

/*
 * Measures the performance (Mflops/sec) of basic and optimized algorithms
 * implementing Cholesky factorization on symmetric positive definite matrices
 * over a range of dimensions.  The algorithms employ performance optimization
 * techniques including loop reordering, blocking and the use of BLAS and
 * LAPACK libraries.
 */
void time_chol( void )
{
#define FIELDS 12           // Number of output data fields
    const char      *data_file_name = "chol.dat";
    const char      *hdr_text =
"# N:       Matrix dimension, N-by-N\n"
"# BDIM:    Block dimension for blocked algorithms\n"
"#          Mflop/sec for Cholesky factorization algorithms\n"
"# OUTPROD: Outer product method, kji indexing\n"
"# SAXPY:   SAXPY operation, jki indexing\n"
"# LAPUNBK: LAPACK routine DPOTF2, unblocked version\n"
"# BLKSIMP: Simple blocking\n"
"# BLKRECT: Simple blocking, rectangular version of Cholesky factorization\n"
"# BLKCTG:  Contiguous blocking\n"
"# BLKRCR:  Recursive contiguous blocking\n"
"# BLAS:    Simple blocking using BLAS\n"
"# CTGBLAS: Contiguous blocking using BLAS\n"
"# LAPACK:  LAPACK routine DPOTRF\n"
"# \n"
"# N\tBDIM\tOUTPROD\tSAXPY\tLAPUNBK\tBLKSIMP\tBLKRECT\tBLKCTG\tBLKRCR"
"\tBLAS\tCTGBLAS\tLAPACK";
    const int        col_n = 0,
                     col_bdim = 1,
                     col_outprod = 2,
                     col_saxpy = 3,
                     col_lapunbk = 4,
                     col_blksimp = 5,
                     col_blkrect = 6,
                     col_blkctg = 7,
                     col_blkrcr = 8,
                     col_blas = 9,
                     col_ctgblas = 10,
                     col_lapack = 11;
    const double     alpha = 1.0;    // Scaling factor for random matrix

    char     *data_file;
    int      n, bdim;
    double   mflops;
    double   perf_data[FIELDS*SIZES];
    double   *A;
    void     (*mfact)( int n, double *A );

    // Concatenate file path and name
    data_file = (char *) calloc( strlen(file_path) + strlen(data_file_name) + 1,
        sizeof(char) );
    strcpy( data_file, file_path );
    strcat( data_file, data_file_name );

    for ( int i = 0; i < SIZES; i++ ) {
        n = mat_size[i];
        bdim = get_block_dim_chol( n );
        fprintf( stdout, "n = %d, bdim = %d\n", n, bdim );
        // Cholesky factorization takes (1/3)*n^3 floating point operations
        mflops = (1.0e-06 / 3.0) * n * n * n;
        // Create random n-by-n symmetric positive definite matrix
        A = (double *) malloc( n*n*sizeof(double) );
        create_random_spd( alpha, n, A );

        // Performance data is stored in perf_data[] array in column-major order
        perf_data[i+col_n*SIZES] = (double) n;
        perf_data[i+col_bdim*SIZES] = (double) bdim;

        // Measure performance of standard Cholesky algorithms:
printf("chol_outer_product\n");
        // Outer product method (kji indexing)
        mfact = chol_outer_product;
        perf_data[i+col_outprod*SIZES] = mflops / time_mfact( mfact, n, A );
printf("chol_saxpy\n");
        // SAXPY operation (jki indexing)
        mfact = chol_saxpy;
        perf_data[i+col_saxpy*SIZES] = mflops / time_mfact( mfact, n, A );
printf("chol_lapack_unblocked\n");
        // LAPACK routine DPOTF2, unblocked version
        mfact = chol_lapack_unblocked;
        perf_data[i+col_lapunbk*SIZES] = mflops / time_mfact( mfact, n, A );
printf("chol_block\n");
        // Simple blocking
        mfact = chol_block;
        perf_data[i+col_blksimp*SIZES] = mflops / time_mfact( mfact, n, A );
printf("chol_rect_block\n");
        // Simple blocking, rectangular version of Cholesky factorization
        mfact = chol_rect_block;
        perf_data[i+col_blkrect*SIZES] = mflops / time_mfact( mfact, n, A );
printf("chol_contig_block\n");
        // Contiguous blocking
        mfact = chol_contig_block;
        perf_data[i+col_blkctg*SIZES] = mflops / time_mfact( mfact, n, A );
printf("chol_recur_block\n");
        // Recursive contiguous blocking
        mfact = chol_recur_block;
        perf_data[i+col_blkrcr*SIZES] = mflops / time_mfact( mfact, n, A );
printf("chol_block_blas\n");
        // Simple blocking using the BLAS library
        mfact = chol_block_blas;
        perf_data[i+col_blas*SIZES] = mflops / time_mfact( mfact, n, A );
printf("chol_contig_block_blas\n");
        // Contiguous blocking using the BLAS library
        mfact = chol_contig_block_blas;
        perf_data[i+col_ctgblas*SIZES] = mflops / time_mfact( mfact, n, A );
printf("chol_lapack\n");
        // LAPACK routine DPOTRF
        mfact = chol_lapack;
        perf_data[i+col_lapack*SIZES] = mflops / time_mfact( mfact, n, A );

        free( A );
    }
    write_data_file( data_file, hdr_text, SIZES, FIELDS, perf_data );

#undef FIELDS
}

/*
 * Profiles blocked algorithms implementing Cholesky factorization on symmetric
 * positive definite matrices. Profile data estimate the time and proportion of
 * time spent factoring diagonal blocks, solving for lower triangular column
 * blocks and updating the trailing sub-matrix.
 */
void profile_chol( void )
{
    const int   n = 2000;
    const char  *mat_file_name = "mat_2000_spd.dat";

    char    *mat_file;
    double  *A, *W;

    A = (double *) malloc( n*n*sizeof(double) );
    W = (double *) malloc( n*n*sizeof(double) );

    // Concatenate file path and name
    mat_file = (char *) calloc( strlen(file_path) + strlen(mat_file_name) + 1,
        sizeof(char) );
    strcpy( mat_file, file_path );
    strcat( mat_file, mat_file_name );

    read_matrix( mat_file, n, n, A );
    copy_matrix( n, n, A, W );

    fprintf( stdout, "Profile of Cholesky factorization (seconds)\n" );
    fprintf( stdout, "Simple blocking\n" );
    fprintf( stdout, "%d-by-%d symmetric positive definite matrix\n", n, n );
    fprintf( stdout, "tm_chol\ttm_factor\ttm_tri_solve\ttm_reduce\t" );
    fprintf( stdout, "pct_factor\tpct_tri_solve\tpct_reduce\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        chol_block( n, A );
        copy_matrix( n, n, W, A );
    }

    fprintf( stdout, "\nProfile of Cholesky factorization (seconds)\n" );
    fprintf( stdout, "Blocked algorithm using BLAS\n" );
    fprintf( stdout, "%d-by-%d symmetric positive definite matrix\n", n, n );
    fprintf( stdout, "tm_chol\ttm_factor\ttm_tri_solve\ttm_reduce\t" );
    fprintf( stdout, "pct_factor\tpct_tri_solve\tpct_reduce\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        chol_block_blas( n, A );
        copy_matrix( n, n, W, A );
    }
}

/*
 * Measures the performance (Mflops/sec) of basic and optimized algorithms
 * implementing symmetric indefinite factorization (LDL') on matrices over a
 * range of dimensions.  Our implementation of symmetric indefinite
 * factorization uses Bunch-Kaufman (partial), bounded Bunch-Kaufman (rook) or
 * Bunch-Parlett (complete) pivoting.  The algorithms employ performance
 * optimization techniques including loop reordering, blocking and the use of
 * BLAS and LAPACK libraries.
 */
void time_ldlt( void )
{
#define FIELDS 20           // Number of output data fields
    const char      *data_file_name = "ldlt.dat";
    const char      *hdr_text =
"# N:       Matrix dimension, N-by-N\n"
"# BDIM:    Block dimension for simple blocking algorithm\n"
"# BDIMBLA: Block dimension for blocked algorithm using BLAS\n"
"# BDIMLAP: Block dimension for LAPACK routine DSYTRF\n"
"# NUMPIVK: Pivot count, Bunch-Kaufman pivoting\n"
"# NUMPIVB: Pivot count, bounded Bunch-Kaufman pivoting\n"
"# NUMPIVP: Pivot count, Bunch-Parlett pivoting\n"
"#          Mflop/sec for symmetric indefinite factorization algorithms\n"
"# OUTPRDK: Outer product method, Bunch-Kaufman pivoting\n"
"# SAXPYK:  SAXPY operation, Bunch-Kaufman pivoting\n"
"# LAPUNBK: LAPACK routine DSYTF2, unblocked version\n"
"# BLOCKK:  Simple blocking, Bunch-Kaufman pivoting\n"
"# BLASK:   Simple blocking, Bunch-Kaufman pivoting, BLAS routines \n"
"# LAPACK:  LAPACK routine DSYTRF, Bunch-Kaufman pivoting\n"
"# OUTPRDB: Outer product method, bounded Bunch-Kaufman pivoting\n"
"# SAXPYB:  SAXPY operation, bounded Bunch-Kaufman pivoting\n"
"# BLOCKB:  Simple blocking, bounded Bunch-Kaufman pivoting\n"
"# BLASB:   Simple blocking, bounded Bunch-Kaufman pivoting, BLAS routines \n"
"# OUTPRDP: Outer product method, Bunch-Parlett pivoting\n"
"# BLOCKP:  Simple blocking, Bunch-Parlett pivoting\n"
"# BLASP:   Simple blocking, Bunch-Parlett pivoting, BLAS routines \n"
"# \n"
"# N\tBDIM\tBDIMBLA\tBDIMLAP\tNUMPIVK\tNUMPIVB\tNUMPIVP\tOUTPRDK\tSAXPYK\tLAPUNBK"
"\tBLOCKK\tBLASK\tLAPACK\tOUTPRDB\tSAXPYB\tBLOCKB\tBLASB\tOUTPRDP\tBLOCKP\tBLASP";
    const int       col_n = 0,
                    col_bdim = 1,
                    col_bdimbla = 2,
                    col_bdimlap = 3,
                    col_numpivk = 4,
                    col_numpivb = 5,
                    col_numpivp = 6,
                    col_outprdk = 7,
                    col_saxpyk = 8,
                    col_lapunbk = 9,
                    col_blockk = 10,
                    col_blask = 11,
                    col_lapack = 12,
                    col_outprdb = 13,
                    col_saxpyb = 14,
                    col_blockb = 15,
                    col_blasb = 16,
                    col_outprdp = 17,
                    col_blockp = 18,
                    col_blasp = 19;
    const double    alpha = 10.0;   // Scaling factor for random matrix

    char     *data_file;
    int      n, bdim, bdim_blas, bdim_lapack;
    int      *piv, *ord;
    double   mflops;
    double   perf_data[FIELDS*SIZES];
    double   *A;
    void     (*mfact_piv)( char pivot, int n, int *piv, int *ord, double *A );

    // Concatenate file path and name
    data_file = (char *) calloc( strlen(file_path) + strlen(data_file_name) + 1,
        sizeof(char) );
    strcpy( data_file, file_path );
    strcat( data_file, data_file_name );

    for ( int i = 0; i < SIZES; i++ ) {
        n = mat_size[i];
        bdim = get_block_dim_ldlt( 0, 0, n );
        bdim_blas = get_block_dim_ldlt( 0, 1, n );
        bdim_lapack = get_block_dim_ldlt( 1, 0, n );
        fprintf( stdout, "n = %d\n", n );
        // (1/3)*n^3 floating point operations is a lower bound on symmetric
        // indefinite factorization
        mflops = 1.0e-06 * (1.0/3.0) * n * n * n;
        // Create random n-by-n symmetric matrix
        A = (double *) malloc( n*n*sizeof(double) );
        create_random_symmetric( alpha, n, A );
        // Declare pivot and pivot order vectors
        piv = (int *) malloc( n*sizeof(int) );
        ord = (int *) malloc( n*sizeof(int) );

        // Performance data is stored in perf_data[] array in column-major order
        perf_data[i+col_n*SIZES] = (double) n;
        perf_data[i+col_bdim*SIZES] = (double) bdim;
        perf_data[i+col_bdimbla*SIZES] = (double) bdim_blas;
        perf_data[i+col_bdimlap*SIZES] = (double) bdim_lapack;

        // Measure performance of LDL' algorithms
printf("ldlt_outer_product(Bunch-Kaufman)\n");
        // Outer product method (kji indexing), Bunch-Kaufman pivoting
        mfact_piv = ldlt_outer_product;
        perf_data[i+col_outprdk*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );

printf("ldlt_saxpy(Bunch-Kaufman)\n");
        // SAXPY operation (jki indexing), Bunch-Kaufman pivoting
        mfact_piv = ldlt_saxpy;
        perf_data[i+col_saxpyk*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );

printf("ldlt_lapack_unblocked(Bunch-Kaufman)\n");
        // LAPACK routine DSYTF2, unblocked version
        mfact_piv = ldlt_lapack_unblocked;
        perf_data[i+col_lapunbk*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );

printf("ldlt_block(Bunch-Kaufman)\n");
        // Simple blocking, Bunch-Kaufman pivoting
        mfact_piv = ldlt_block;
        perf_data[i+col_blockk*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );

printf("ldlt_block_blas(Bunch-Kaufman)\n");
        // Simple blocking, Bunch-Kaufman pivoting, BLAS routines
        mfact_piv = ldlt_block_blas;
        perf_data[i+col_blask*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );
        // Pivot count, Bunch-Kaufman
        perf_data[i+col_numpivk*SIZES] = (double) count_pivot( 0, n, piv, ord );

printf("ldlt_lapack\n");
        // LAPACK routine DSYTRF, Bunch-Kaufman pivoting
        mfact_piv = ldlt_lapack;
        perf_data[i+col_lapack*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );

printf("ldlt_outer_product(bounded Bunch-Kaufman)\n");
        // Outer product method (kji indexing), bounded Bunch-Kaufman pivoting
        mfact_piv = ldlt_outer_product;
        perf_data[i+col_outprdb*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'B', n, piv, ord, A );

printf("ldlt_saxpy(bounded Bunch-Kaufman)\n");
        // SAXPY operation (jki indexing), bounded Bunch-Kaufman pivoting
        mfact_piv = ldlt_saxpy;
        perf_data[i+col_saxpyb*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'B', n, piv, ord, A );

printf("ldlt_block(bounded Bunch-Kaufman)\n");
        // Simple blocking, bounded Bunch-Kaufman pivoting
        mfact_piv = ldlt_block;
        perf_data[i+col_blockb*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'B', n, piv, ord, A );

printf("ldlt_block_blas(bounded Bunch-Kaufman)\n");
        // Simple blocking, bounded Bunch-Kaufman pivoting, BLAS routines
        mfact_piv = ldlt_block_blas;
        perf_data[i+col_blasb*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'B', n, piv, ord, A );
        // Pivot count, bounded Bunch-Kaufman
        perf_data[i+col_numpivb*SIZES] = (double) count_pivot( 0, n, piv, ord );

printf("ldlt_outer_product(Bunch-Parlett)\n");
        // Outer product method (kji indexing), Bunch-Parlett pivoting
        mfact_piv = ldlt_outer_product;
        perf_data[i+col_outprdp*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'P', n, piv, ord, A );

printf("ldlt_block(Bunch-Parlett)\n");
        // Simple blocking, Bunch-Parlett pivoting
        mfact_piv = ldlt_block;
        perf_data[i+col_blockp*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'P', n, piv, ord, A );

printf("ldlt_block_blas(Bunch-Parlett)\n");
        // Simple blocking, Bunch-Parlett pivoting, BLAS routines
        mfact_piv = ldlt_block_blas;
        perf_data[i+col_blasp*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'P', n, piv, ord, A );
        // Pivot count, Bunch-Parlett
        perf_data[i+col_numpivp*SIZES] = (double) count_pivot( 0, n, piv, ord );

        free( A );
        free( piv );
        free( ord );
    }
    write_data_file( data_file, hdr_text, SIZES, FIELDS, perf_data );

#undef FIELDS
}

/*
 * Measures the time taken to perform symmetric indefinite (LDL') factorization
 * on matrices of varying degrees of indefiniteness as proxied by the number of
 * Bunch-Kaufman pivots.  Time measurements are made for blocked algorithms
 * employing Bunch-Kaufman (partial), bounded Bunch-Kaufman (rook) or
 * Bunch-Parlett (complete) pivoting.
 */
void time_ldlt_indef( void )
{
#if defined(DEBUG)
    #define MATS 4
#else
    #define MATS 14
#endif
#define FIELDS 13            // Number of output data fields
    const char      *data_file_name = "ldlt_indef.dat";
    const char      *hdr_text =
"# N:       Matrix dimension, N-by-N\n"
"# BDIM:    Block dimension for simple blocking algorithm\n"
"# BDIMBLA: Block dimension for blocked algorithm using BLAS\n"
"# BDIMLAP: Block dimension for LAPACK routine DSYTRF\n"
"# NUMPIVK: Pivot count, Bunch-Kaufman pivoting\n"
"# NUMPIVB: Pivot count, bounded Bunch-Kaufman pivoting\n"
"# NUMPIVP: Pivot count, Bunch-Parlett pivoting\n"
"#          Time (seconds) taken for LDL' factorization of symmetric matrices\n"
"# BLOCKK:  Simple blocking, Bunch-Kaufman pivoting\n"
"# BLASK:   Simple blocking, Bunch-Kaufman pivoting, BLAS routines \n"
"# LAPACK:  LAPACK routine DSYTRF, Bunch-Kaufman pivoting\n"
"# BLASB:   Simple blocking, bounded Bunch-Kaufman pivoting, BLAS routines\n"
"# BLASP:   Simple blocking, Bunch-Parlett pivoting, BLAS routines\n"
"# LAPCHOL: LAPACK routine DPOTRF, Cholesky factorization\n"
"# \n"
"# N\tBDIM\tBDIMBLA\tBDIMLAP\tNUMPIVK\tNUMPIVB\tNUMPIVP\tBLOCKK\tBLASK\tLAPACK"
"\tBLASB\tBLASP\tLAPCHOL";
    const int       col_n = 0,
                    col_bdim = 1,
                    col_bdimbla = 2,
                    col_bdimlap = 3,
                    col_numpivk = 4,
                    col_numpivb = 5,
                    col_numpivp = 6,
                    col_blockk = 7,
                    col_blask = 8,
                    col_lapack = 9,
                    col_blasb = 10,
                    col_blasp = 11,
                    col_lapchol = 12;
    const int       n = 2000;       // Matrix dimension
    const double    alpha = 10.0;   // Scaling factor for random matrix

    char    *data_file, *mat_file;
    char    *mat_file_name[MATS];
    int     bdim, bdim_blas, bdim_lapack;
    int     *piv, *ord;
    double  mflops;
    double  time[FIELDS*SIZES];
    double  *A;
    void    (*mfact_piv)( char pivot, int n, int *piv, int *ord, double *A );
    void    (*mfact)( int n, double *A );

    // Concatenate output data file path and name
    data_file = (char *) calloc( strlen(file_path) + strlen(data_file_name) + 1,
        sizeof(char) );
    strcpy( data_file, file_path );
    strcat( data_file, data_file_name );
    // Matrices of varying degrees of indefiniteness are stored in files
    mat_file_name[0] = "mat_2000_spd.dat";
    mat_file_name[1] = "mat_2000_bk24.dat";
    mat_file_name[2] = "mat_2000_bk49.dat";
    mat_file_name[3] = "mat_2000_bk98.dat";
#if !defined(DEBUG)
    mat_file_name[4] = "mat_2000_bk148.dat";
    mat_file_name[5] = "mat_2000_bk202.dat";
    mat_file_name[6] = "mat_2000_bk302.dat";
    mat_file_name[7] = "mat_2000_bk400.dat";
    mat_file_name[8] = "mat_2000_bk500.dat";
    mat_file_name[9] = "mat_2000_bk597.dat";
    mat_file_name[10] = "mat_2000_bk703.dat";
    mat_file_name[11] = "mat_2000_bk797.dat";
    mat_file_name[12] = "mat_2000_bk880.dat";
    mat_file_name[13] = "mat_2000_sym.dat";
#endif
    bdim = get_block_dim_ldlt( 0, 0, n );
    bdim_blas = get_block_dim_ldlt( 0, 1, n );
    bdim_lapack = get_block_dim_ldlt( 1, 0, n );
    // Declare matrix A, and pivot and pivot order vectors
    A = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );

    for ( int i = 0; i < MATS; i++ ) {
        // Concatentate matrix file path and name
        mat_file = (char *) calloc( strlen(file_path) +
            strlen(mat_file_name[i]) + 1, sizeof(char) );
        strcpy( mat_file, file_path );
        strcat( mat_file, mat_file_name[i] );
        // Read n-by-n symmetric matrix from file
        read_matrix( mat_file, n, n, A );
        // Time measurements are stored in time_data[] array in column-major order
        time[i+col_n*MATS] = (double) n;
        time[i+col_bdim*MATS] = (double) bdim;
        time[i+col_bdimbla*MATS] = (double) bdim_blas;
        time[i+col_bdimlap*MATS] = (double) bdim_lapack;

        // Time LDL' factorization
printf("ldlt_block(Bunch-Kaufman)\n");
        // Simple blocking, Bunch-Kaufman pivoting
        mfact_piv = ldlt_block;
        time[i+col_blockk*MATS] =
            time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );
        // Pivot count, Bunch-Kaufman
        time[i+col_numpivk*MATS] = (double) count_pivot( 0, n, piv, ord );
        fprintf( stdout, "n = %d, number of (Bunch-Kaufman) pivots = %.0f\n",
            n, time[i+col_numpivk*MATS] );

printf("ldlt_block_blas(Bunch-Kaufman)\n");
        // Simple blocking, Bunch-Kaufman pivoting, BLAS routines
        mfact_piv = ldlt_block_blas;
        time[i+col_blask*MATS] =
            time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );

printf("ldlt_lapack\n");
        // LAPACK routine DSYTRF, Bunch-Kaufman pivoting
        mfact_piv = ldlt_lapack;
        time[i+col_lapack*MATS] =
            time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );

printf("ldlt_block(bounded Bunch-Kaufman)\n");
        // Simple blocking, bounded Bunch-Kaufman pivoting
        mfact_piv = ldlt_block_blas;
        time[i+col_blasb*MATS] =
            time_mfact_pivot( mfact_piv, 'B', n, piv, ord, A );
        // Pivot count, bounded Bunch-Kaufman
        time[i+col_numpivb*MATS] = (double) count_pivot( 0, n, piv, ord );

printf("ldlt_block(Bunch-Parlett)\n");
        // Simple blocking, Bunch-Parlett pivoting
        mfact_piv = ldlt_block_blas;
        time[i+col_blasp*MATS] =
            time_mfact_pivot( mfact_piv, 'P', n, piv, ord, A );
        // Pivot count, Bunch-Parlett
        time[i+col_numpivp*MATS] = (double) count_pivot( 0, n, piv, ord );

        if ( i == 0 ) {     // Symmetric positive definite
printf("chol_lapack\n");
            // LAPACK routine DPOTRF
            mfact = chol_lapack;
            time[i+col_lapchol*MATS] = time_mfact( mfact, n, A );
        } else {
            time[i+col_lapchol*MATS] = -1.0;
        }
    }
    free( A );
    free( piv );
    free( ord );
    write_data_file( data_file, hdr_text, MATS, FIELDS, time );

#undef FIELDS
#undef MATS
}

/*
 * Profiles blocked algorithms implementing symmetric indefinite factorization
 * (LDL'). Profile data estimate the time and proportion of time spent
 * factoring column blocks, performing symmetric pivoting and updating the
 * trailing sub-matrix.
 */
```

```
void profile_ldlt( void )
{
    const int   n = 2000;
    const char  *mat_file_name = "mat_2000_bk500.dat";

    char     *mat_file;
    int      num_piv;
    int      *piv, *ord;
    double   *A, *W;

    A = (double *) malloc( n*n*sizeof(double) );
    W = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );

    // Concatenate file path and name
    mat_file = (char *) calloc( strlen(file_path) + strlen(mat_file_name) + 1,
        sizeof(char) );
    strcpy( mat_file, file_path );
    strcat( mat_file, mat_file_name );

    read_matrix( mat_file, n, n, A );
    copy_matrix( n, n, A, W );

    fprintf( stdout, "Profile of symmetric indefinite factorization (seconds)\n" );
    fprintf( stdout, "Simple blocking, Bunch-Kaufman pivoting\n" );
    fprintf( stdout, "%d-by-%d symmetric matrix: %s\n", n, n, mat_file_name );
    fprintf( stdout, "tm_ldlt\ttm_factor\ttm_pivot\ttm_reduce\t" );
    fprintf( stdout, "pct_factor\tpct_pivot\tpct_reduce\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        ldlt_block( 'K', n, piv, ord, A );
        copy_matrix( n, n, W, A );
    }

    fprintf( stdout, "\nProfile of symmetric indefinite factorization (seconds)\n" );
    fprintf( stdout, "Blocked algorithm using BLAS, Bunch-Kaufman pivoting\n" );
    fprintf( stdout, "%d-by-%d symmetric matrix: %s\n", n, n, mat_file_name );
    fprintf( stdout, "tm_ldlt\ttm_factor\ttm_pivot\ttm_reduce\t" );
    fprintf( stdout, "pct_factor\tpct_pivot\tpct_reduce\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        ldlt_block_blas( 'K', n, piv, ord, A );
        copy_matrix( n, n, W, A );
    }

    fprintf( stdout, "\nProfile of symmetric indefinite factorization (seconds)\n" );
    fprintf( stdout, "Simple blocking, bounded Bunch-Kaufman pivoting\n" );
    fprintf( stdout, "%d-by-%d symmetric matrix: %s\n", n, n, mat_file_name );
    fprintf( stdout, "tm_ldlt\ttm_factor\ttm_pivot\ttm_reduce\t" );
    fprintf( stdout, "pct_factor\tpct_pivot\tpct_reduce\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        ldlt_block( 'B', n, piv, ord, A );
        copy_matrix( n, n, W, A );
    }

    fprintf( stdout, "\nProfile of symmetric indefinite factorization (seconds)\n" );
    fprintf( stdout, "Blocked algorithm using BLAS, bounded Bunch-Kaufman pivoting\n" );
    fprintf( stdout, "%d-by-%d symmetric matrix: %s\n", n, n, mat_file_name );
    fprintf( stdout, "tm_ldlt\ttm_factor\ttm_pivot\ttm_reduce\t" );
    fprintf( stdout, "pct_factor\tpct_pivot\tpct_reduce\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        ldlt_block_blas( 'B', n, piv, ord, A );
        copy_matrix( n, n, W, A );
    }
}

/*
 * Measures the performance (Mflops/sec) of basic and optimized algorithms
 * implementing the modified Cholesky factorization proposed by Gill, Murray &
 * Wright (with partial pivoting) on symmetric matrices over a range of
 * dimensions.  The algorithms employ performance optimization techniques
 * including loop reordering, blocking and the use of the BLAS library.
 */
void time_chol_gmw( void )
{
#define FIELDS 8        // Number of output data fields
    const char      *data_file_name = "chol_gmw.dat";
    const char      *hdr_text =
"# N:       Matrix dimension, N-by-N\n"
"# BDIM:    Block dimension for simple blocking algorithm\n"
"# BDIMBLA: Block dimension for blocked algorithm using BLAS\n"
"#          Mflop/sec for modified Cholesky algorithms (Gill, Murray & Wright)\n"
"# NUMPIV:  Pivot count\n"
"# OUTPROD: Outer product method, kji indexing\n"
"# SAXPY:   SAXPY operation, jki indexing\n"
"# BLOCK:   Simple blocking\n"
"# BLAS:    Simple blocking, BLAS routines\n"
"# \n"
"# N\tBDIM\tBDIMBLA\tNUMPIV\tOUTPROD\tSAXPY\tBLOCK\tBLAS";
    const int       col_n = 0,
                    col_bdim = 1,
                    col_bdimbla = 2,
                    col_numpiv = 3,
                    col_outprod = 4,
                    col_saxpy = 5,
                    col_block = 6,
                    col_blas = 7;
    const double    alpha = 10.0;   // Scaling factor for random matrix

    char    *data_file;
    int     n, bdim, bdim_blas;
    int       *piv, *ord;
    double    mflops;
    double    perf_data[FIELDS*SIZES];
    double    *A;
    void      (*mfact_piv)( char pivot, int n, int *piv, int *ord, double *A );

    // Concatenate file path and name
    data_file = (char *) calloc( strlen(file_path) + strlen(data_file_name) + 1,
        sizeof(char) );
    strcpy( data_file, file_path );
    strcat( data_file, data_file_name );

    for ( int i = 0; i < SIZES; i++ ) {
        n = mat_size[i];
        bdim = get_block_dim_ldlt( 0, 0, n );
        bdim_blas = get_block_dim_ldlt( 0, 1, n );
        fprintf( stdout, "n = %d\n", n );
        // (1/3)*n^3 floating point operations is a lower bound on modified
        // Cholesky factorization
        mflops = 1.0e-06 * (1.0/3.0) * n * n * n;
        // Create random n-by-n symmetric matrix
        A = (double *) malloc( n*n*sizeof(double) );
        create_random_symmetric( alpha, n, A );
        // Declare pivot and pivot order vectors
        piv = (int *) malloc( n*sizeof(int) );
        ord = (int *) malloc( n*sizeof(int) );

        // Performance data is stored in perf_data[] array in column-major order
        perf_data[i+col_n*SIZES] = (double) n;
        perf_data[i+col_bdim*SIZES] = (double) bdim;
        perf_data[i+col_bdimbla*SIZES] = (double) bdim_blas;

        // Measure performance of modified Cholesky algorithms
        // Gill, Murray & Wright algorithm with Type-1 modification
printf("chol_gmw_outer_product\n");
        // Outer product method (kji indexing)
        mfact_piv = chol_gmw_outer_product;
        perf_data[i+col_outprod*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'D', n, piv, ord, A );

printf("chol_gmw_saxpy\n");
        // SAXPY operation (jki indexing), diagonal pivoting
        mfact_piv = chol_gmw_saxpy;
        perf_data[i+col_saxpy*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'D', n, piv, ord, A );

printf("chol_gmw_block\n");
        // Simple blocking, diagonal pivoting
        mfact_piv = chol_gmw_block;
        perf_data[i+col_block*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'D', n, piv, ord, A );
        // Pivot count, diagonal (Gill, Murray & Wright)
        perf_data[i+col_numpiv*SIZES] = (double) count_pivot( 0, n, piv, ord );

printf("chol_gmw_block_blas\n");
        // Simple blocking, diagonal pivoting, BLAS routines
        mfact_piv = chol_gmw_block_blas;
        perf_data[i+col_blas*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'D', n, piv, ord, A );

        free( A );
        free( piv );
        free( ord );
    }
    write_data_file( data_file, hdr_text, SIZES, FIELDS, perf_data );

#undef FIELDS
}

/*
 * Measures the performance (Mflops/sec) of basic and optimized algorithms
 * implementing the modified Cholesky factorization proposed by Cheng & Higham
 * on symmetric matrices over a range of dimensions.  Our implementation of the
 * modified Cholesky factorization proposed by Cheng & Higham uses either
 * Bunch-Kaufman (partial) or bounded Bunch-Kaufman (rook) pivoting.  The
 * algorithms employ performance optimization techniques including loop
 * reordering, blocking and the use of the BLAS library.
 */
void time_chol_ch( void )
{
#define FIELDS 9            // Number of output data fields
    const char      *data_file_name = "chol_ch.dat";
    const char      *hdr_text =
"# N:       Matrix dimension, N-by-N\n"
"# BDIM:    Block dimension for simple blocking algorithm\n"
"# BDIMBLA: Block dimension for blocked algorithm using BLAS\n"
"# NUMPIVK: Pivot count, Bunch-Kaufman pivoting\n"
"# NUMPIVB: Pivot count, bounded Bunch-Kaufman pivoting\n"
"#          Mflop/sec for modified Cholesky algorithms (Cheng & Higham)\n"
"# BLOCKK:  Simple blocking, Bunch-Kaufman pivoting\n"
"# BLASK:   Simple blocking, Bunch-Kaufman pivoting, BLAS routines \n"
"# BLOCKB:  Simple blocking, bounded Bunch-Kaufman pivoting\n"
"# BLASB:   Simple blocking, bounded Bunch-Kaufman pivoting, BLAS routines \n"
"# \n"
"# N\tBDIM\tBDIMBLA\tNUMPIVK\tNUMPIVB\tBLOCKK\tBLASK\tBLOCKB\tBLASB";
    const int       col_n = 0,
                    col_bdim = 1,
                    col_bdimbla = 2,
                    col_numpivk = 3,
                    col_numpivb = 4,
                    col_blockk = 5,
                    col_blask = 6,
                    col_blockb = 7,
                    col_blasb = 8;
    const double     alpha = 10.0;    // Scaling factor for random matrix

    char        *data_file;
    int      n, bdim, bdim_blas;
    int      *piv, *ord;
    double   mflops;
    double   perf_data[FIELDS*SIZES];
    double   *A;
    void     (*mfact_piv)( char pivot, int n, int *piv, int *ord, double *A );

    // Concatenate file path and name
    data_file = (char *) calloc( strlen(file_path) + strlen(data_file_name) + 1,
        sizeof(char) );
    strcpy( data_file, file_path );
    strcat( data_file, data_file_name );

    for ( int i = 0; i < SIZES; i++ ) {
        n = mat_size[i];
        bdim = get_block_dim_ldlt( 0, 0, n );
        bdim_blas = get_block_dim_ldlt( 0, 1, n );
        fprintf( stdout, "n = %d\n", n );
        // (1/3)*n^3 floating point operations is a lower bound on modified
        // Cholesky factorization
        mflops = 1.0e-06 * (1.0/3.0) * n * n * n;
        // Create random n-by-n symmetric matrix
        A = (double *) malloc( n*n*sizeof(double) );
        create_random_symmetric( alpha, n, A );
        // Declare pivot and pivot order vectors
        piv = (int *) malloc( n*sizeof(int) );
        ord = (int *) malloc( n*sizeof(int) );

        // Performance data is stored in perf_data[] array in column-major order
        perf_data[i+col_n*SIZES] = (double) n;
        perf_data[i+col_bdim*SIZES] = (double) bdim;
        perf_data[i+col_bdimbla*SIZES] = (double) bdim_blas;

        // Measure performance of modified Cholesky algorithms
        // Cheng & Higham algorithm with Type-II modification
printf("chol_ch_block(Bunch-Kaufman)\n");
        // Simple blocking, Bunch-Kaufman pivoting
        mfact_piv = chol_ch_block;
        perf_data[i+col_blockk*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );
        // Pivot count, Bunch-Kaufman
        perf_data[i+col_numpivk*SIZES] = (double) count_pivot( 0, n, piv, ord );
printf("chol_ch_block_blas(Bunch-Kaufman)\n");
        // Simple blocking, Bunch-Kaufman pivoting, BLAS routines
        mfact_piv = chol_ch_block_blas;
        perf_data[i+col_blask*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );

printf("chol_ch_block(bounded Bunch-Kaufman)\n");
        // Simple blocking, Bunch-Kaufman pivoting
        mfact_piv = chol_ch_block;
        perf_data[i+col_blockb*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'B', n, piv, ord, A );
        // Pivot count, Bunch-Kaufman
        perf_data[i+col_numpivb*SIZES] = (double) count_pivot( 0, n, piv, ord );

printf("chol_ch_block_blas(bounded Bunch-Kaufman)\n");
        // Simple blocking, Bunch-Kaufman pivoting, BLAS routines
        mfact_piv = chol_ch_block_blas;
        perf_data[i+col_blasb*SIZES] =
            mflops / time_mfact_pivot( mfact_piv, 'B', n, piv, ord, A );

        free( A );
        free( piv );
        free( ord );
    }
    write_data_file( data_file, hdr_text, SIZES, FIELDS, perf_data );

#undef FIELDS
}

/*
 * Measures the time taken to perform modified Cholesky factorization
 * (Gill-Murray-Wright and Cheng-Higham) on matrices of varying degrees of
 * indefiniteness as proxied by the number of Bunch-Kaufman pivots. Time
 * measurements are made for blocked algorithms employing partial and rook
 * pivoting strategies.
 */
void time_mod_chol_indef( void )
{
#if defined(DEBUG)
    #define MATS 4
#else
    #define MATS 14
#endif
#define FIELDS 12            // Number of output data fields
    const char      *data_file_name = "mod_chol_indef.dat";
    const char      *hdr_text =
"# N:       Matrix dimension, N-by-N\n"
"# BDIM:    Block dimension for simple blocking algorithm\n"
"# BDIMBLA: Block dimension for blocked algorithm using BLAS\n"
"# NUMPIVK: Pivot count, Bunch-Kaufman pivoting\n"
"# NUMPIVB: Pivot count, bounded Bunch-Kaufman pivoting\n"
"# NUMPIVD: Pivot count, Gill-Murray-Wright diagonal pivoting\n"
"#          Time (seconds) taken for modified Cholesky factorization\n"
"# CHBLKK:  Cheng-Higham, simple blocking, Bunch-Kaufman pivoting\n"
"# CHBLKB:  Cheng-Higham, simple blocking, bounded Bunch-Kaufman pivoting\n"
"# CHBLASK: Cheng-Higham, BLAS routines, Bunch-Kaufman pivoting \n"
"# CHBLASB: Cheng-Higham, BLAS routines, bounded Bunch-Kaufman pivoting\n"
"# GMWBLK:  Gill-Murray-Wright, simple blocking, partial pivoting\n"
"# GMWBLAS: Gill-Murray-Wright, BLAS routines, partial pivoting\n"
"# \n"
"# N\tBDIM\tBDIMBLA\tNUMPIVK\tNUMPIVB\tNUMPIVD\tCHBLKK\tCHBLKB\tCHBLASK"
"\tCHBLASB\tGMWBLK\tGMWBLAS";
    const int       col_n = 0,
                    col_bdim = 1,
                    col_bdimbla = 2,
                    col_numpivk = 3,
                    col_numpivb = 4,
                    col_numpivd = 5,
                    col_chblkk = 6,
                    col_chblkb = 7,
                    col_chblask = 8,
                    col_chblasb = 9,
                    col_gmwblk = 10,
                    col_gmwblas = 11;
    const int       n = 2000;       // Matrix dimension
    const double    alpha = 10.0;   // Scaling factor for random matrix

    char     *data_file, *mat_file;
    char     *mat_file_name[MATS];
    int      bdim, bdim_blas;
    int      *piv, *ord;
    double   mflops;
    double   time[FIELDS*SIZES];
    double   *A;
    void     (*mfact_piv)( char pivot, int n, int *piv, int *ord, double *A );
    void     (*mfact)( int n, double *A );

    // Concatenate output data file path and name
    data_file = (char *) calloc( strlen(file_path) + strlen(data_file_name) + 1,
        sizeof(char) );
    strcpy( data_file, file_path );
    strcat( data_file, data_file_name );
    // Matrices of varying degrees of indefiniteness are stored in files
    mat_file_name[0] = "mat_2000_spd.dat";
    mat_file_name[1] = "mat_2000_bk24.dat";
    mat_file_name[2] = "mat_2000_bk49.dat";
    mat_file_name[3] = "mat_2000_bk98.dat";
#if !defined(DEBUG)
    mat_file_name[4] = "mat_2000_bk148.dat";
    mat_file_name[5] = "mat_2000_bk202.dat";
    mat_file_name[6] = "mat_2000_bk302.dat";
    mat_file_name[7] = "mat_2000_bk400.dat";
    mat_file_name[8] = "mat_2000_bk500.dat";
    mat_file_name[9] = "mat_2000_bk597.dat";
    mat_file_name[10] = "mat_2000_bk703.dat";
    mat_file_name[11] = "mat_2000_bk797.dat";
    mat_file_name[12] = "mat_2000_bk880.dat";
    mat_file_name[13] = "mat_2000_sym.dat";
#endif
    bdim = get_block_dim_ldlt( 0, 0, n );
    bdim_blas = get_block_dim_ldlt( 0, 1, n );
    // Declare matrix A, and pivot and pivot order vectors
    A = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );

    for ( int i = 0; i < MATS; i++ ) {
        // Concatentate matrix file path and name
        mat_file = (char *) calloc( strlen(file_path) +
            strlen(mat_file_name[i]) + 1, sizeof(char) );
        strcpy( mat_file, file_path );
        strcat( mat_file, mat_file_name[i] );
        // Read n-by-n symmetric matrix from file
        read_matrix( mat_file, n, n, A );
        // Time measurements are stored in time_data[] array in column-major order
        time[i+col_n*MATS] = (double) n;
        time[i+col_bdim*MATS] = (double) bdim;
        time[i+col_bdimbla*MATS] = (double) bdim_blas;

        // Time LDL' factorization
printf("chol_ch_block(Bunch-Kaufman)\n");
        // Cheng-Higham, simple blocking, Bunch-Kaufman pivoting
        mfact_piv = chol_ch_block;
        time[i+col_chblkk*MATS] =
            time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );
        // Pivot count, Bunch-Kaufman
        time[i+col_numpivk*MATS] = (double) count_pivot( 0, n, piv, ord );
        fprintf( stdout, "n = %d, number of (Bunch-Kaufman) pivots = %.0f\n",
            n, time[i+col_numpivk*MATS] );

printf("chol_ch_block(bounded Bunch-Kaufman)\n");
        // Cheng-Higham, simple blocking, bounded Bunch-Kaufman pivoting
        mfact_piv = chol_ch_block;
        time[i+col_chblkb*MATS] =
            time_mfact_pivot( mfact_piv, 'B', n, piv, ord, A );
        // Pivot count, bounded Bunch-Kaufman
        time[i+col_numpivb*MATS] = (double) count_pivot( 0, n, piv, ord );

printf("chol_ch_block_blas(Bunch-Kaufman)\n");
        // Cheng-Higham, BLAS routines, Bunch-Kaufman pivoting
        mfact_piv = chol_ch_block_blas;
        time[i+col_chblask*MATS] =
            time_mfact_pivot( mfact_piv, 'K', n, piv, ord, A );

printf("chol_ch_block_blas(bounded Bunch-Kaufman)\n");
        // Cheng-Higham, BLAS routines, bounded Bunch-Kaufman pivoting
        mfact_piv = chol_ch_block_blas;
        time[i+col_chblasb*MATS] =
            time_mfact_pivot( mfact_piv, 'B', n, piv, ord, A );

printf("chol_gmw_block\n");
        // Gill-Murray-Wright, simple blocking, partial pivoting
        mfact_piv = chol_gmw_block;
        time[i+col_gmwblk*MATS] =
            time_mfact_pivot( mfact_piv, 'D', n, piv, ord, A );
        // Pivot count, bounded Bunch-Kaufman
        time[i+col_numpivd*MATS] = (double) count_pivot( 0, n, piv, ord );

printf("chol_gmw_block_blas\n");
        // Gill-Murray-Wright, BLAS routines, partial pivoting
        mfact_piv = chol_gmw_block_blas;
        time[i+col_gmwblas*MATS] =
            time_mfact_pivot( mfact_piv, 'D', n, piv, ord, A );
    }
    free( A );
    free( piv );
    free( ord );
    write_data_file( data_file, hdr_text, MATS, FIELDS, time );

#undef FIELDS
#undef MATS
}

/*
 * Profiles blocked algorithms implementing modified Cholesky factorization
 * (Gill-Murray-Wright and Cheng-Higham) on symmetric matrices.  Profile data
 * estimate the time and proportion of time spent modifying the symmetric
 * indefinite factorization.
 */
void profile_mod_chol( void )
{
    const int   n = 2000;
    const char  *mat_file_name = "mat_2000_bk500.dat";

    char     *mat_file;
    int      *piv, *ord;
    double   *A, *W;

    A = (double *) malloc( n*n*sizeof(double) );
    W = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );

    // Concatenate file path and name
    mat_file = (char *) calloc( strlen(file_path) + strlen(mat_file_name) + 1,
        sizeof(char) );
    strcpy( mat_file, file_path );
    strcat( mat_file, mat_file_name );

    read_matrix( mat_file, n, n, A );
    copy_matrix( n, n, A, W );

    fprintf( stdout, "Time profile of modified Cholesky (seconds)\n" );
    fprintf( stdout, "Gill-Murray-Wright, simple blocking, partial pivoting\n" );
    fprintf( stdout, "%d-by-%d symmetric matrix: %s\n", n, n, mat_file_name );
    fprintf( stdout, "tm_mod_chol\ttm_mod_fact\tpct_mod_fact\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        chol_gmw_block( 'K', n, piv, ord, A );
        copy_matrix( n, n, W, A );
    }

    fprintf( stdout, "\nTime profile of modified Cholesky (seconds)\n" );
    fprintf( stdout, "Gill-Murray-Wright, BLAS routines, partial pivoting\n" );
    fprintf( stdout, "%d-by-%d symmetric matrix: %s\n", n, n, mat_file_name );
    fprintf( stdout, "tm_mod_chol\ttm_mod_fact\tpct_mod_fact\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        chol_gmw_block_blas( 'K', n, piv, ord, A );
        copy_matrix( n, n, W, A );
    }

    fprintf( stdout, "\nTime profile of modified Cholesky (seconds)\n" );
    fprintf( stdout, "Cheng-Higham, simple blocking, Bunch-Kaufman pivoting\n" );
    fprintf( stdout, "%d-by-%d symmetric matrix: %s\n", n, n, mat_file_name );
    fprintf( stdout, "tm_mod_chol\ttm_mod_fact\tpct_mod_fact\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        chol_ch_block( 'K', n, piv, ord, A );
        copy_matrix( n, n, W, A );
    }

    fprintf( stdout, "\nTime profile of modified Cholesky (seconds)\n" );
    fprintf( stdout, "Cheng-Higham, BLAS routines, Bunch-Kaufman pivoting\n" );
    fprintf( stdout, "%d-by-%d symmetric matrix: %s\n", n, n, mat_file_name );
    fprintf( stdout, "tm_mod_chol\ttm_mod_fact\tpct_mod_fact\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        chol_ch_block_blas( 'K', n, piv, ord, A );
        copy_matrix( n, n, W, A );
    }

    fprintf( stdout, "\nTime profile of modified Cholesky (seconds)\n" );
    fprintf( stdout, "Cheng-Higham, simple blocking, bounded Bunch-Kaufman pivoting\n" );
    fprintf( stdout, "%d-by-%d symmetric matrix: %s\n", n, n, mat_file_name );
    fprintf( stdout, "tm_mod_chol\ttm_mod_fact\tpct_mod_fact\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        chol_ch_block( 'B', n, piv, ord, A );
        copy_matrix( n, n, W, A );
    }

    fprintf( stdout, "\nTime profile of modified Cholesky (seconds)\n" );
    fprintf( stdout, "Cheng-Higham, BLAS routines, bounded Bunch-Kaufman pivoting\n" );
    fprintf( stdout, "%d-by-%d symmetric matrix: %s\n", n, n, mat_file_name );
    fprintf( stdout, "tm_mod_chol\ttm_mod_fact\tpct_mod_fact\n" );
    for ( int i = 0; i < MIN_ITER; i++ ) {
        chol_ch_block_blas( 'B', n, piv, ord, A );
        copy_matrix( n, n, W, A );
    }
}
```

A.10. mmultime.c – timing harness for matrix multiplication.

```
/*
 * Timing harness for measuring the performance of basic and "optimized"
 * algorithms implementing matrix multiplication (and addition), C = C + A*B,
 * on square matrices over a range of dimensions.  Performance of matrix
 * multiplication algorithms is also measured for different compiler
 * optimization levels and options.  Performance data are written to an output
 * file destination.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

//#include <sys/types.h>
//#include <sys/resource.h>
//#include <unistd.h>

#include "matmult.h"
#include "matcom.h"
#include "timing.h"

#if !defined(PROC)
#   define PROC "unknown"
#endif
#if !defined(CORES)
#   define CORES "unknown"
#endif
#if !defined(CLKSPEED)
#   define CLKSPEED "unknown"
#endif
#if !defined(CACHE)
#   define CACHE "unknown"
#endif
#if !defined(COMPILER)
#   define COMPILER "unknown"
#endif
#if !defined(LANGUAGE)
#   define LANGUAGE "default"
#endif
#if !defined(OPTM)
#   define OPTM "default"
#endif
#if !defined(DATADIR)
#   define DATADIR "."  // Current directory ./
#endif

#if defined(DEBUG)
#   define MIN_ITER 4        // Minimum number of iterations of algorithm
#   define MIN_SECS 1.0      // Minimum elapsed time for execution of algorithm
```

```
    // Define sizes (dimensions) of square matrices used to measure performance
    const int mat_size[] = { 65, 130, 195, 254 };
#else
#   define MIN_ITER 8
#   define MIN_SECS 2.0
    const int mat_size[] = { 65, 130, 195, 254, 258, 321, 387, 450, 508, 516,
        579, 642, 707, 764, 772, 833, 899, 963, 1021, 1027 };
#endif
#define SIZES (sizeof(mat_size) / sizeof(int))

static void write_data_file( const char *file, const char *hdr_text,
    int rows, int cols, const double *data );
static double time_mmult( void (*mmult)(int n, const double *A, const double *B,
    double *C), int n, const double *A, const double *B, double *C );
static void time_mmult_algo( void );
static void time_compiler_optm( void );

static char     *file_path;

int main()
{
    // Specify file path for output data files
    file_path = (char *) calloc( strlen(DATADIR) + 2, sizeof(char) );
    strcpy( file_path, DATADIR );
    strcat( file_path, "/" );

#if defined(MULTALGO)
    time_mmult_algo();
#endif

#if defined(CCOPTMDP) || defined(CCOPTMSA)
    time_compiler_optm();
#endif

    return 0;
}

/**************************************************************************/

/*
 * Writes header text and experimental data to the file specified in the
 * argument list.  Experimental data is enumerated in a matrix stored in
 * column-major order.
 */
void write_data_file( const char *file, const char *hdr_text,
    int rows, int cols, const double *data )
{
    FILE *fp;

    if ( (fp = fopen( file, "w" )) == NULL ) {
        fprintf( stderr, "Error opening file %s.", file );
        exit( -1 );
    }
    // Write header text
    fprintf( fp, "# Processor:\t%s\n", PROC);
    fprintf( fp, "# Cores:\t%s\n", CORES);
    fprintf( fp, "# Clock speed:\t%s\n", CLKSPEED);
    fprintf( fp, "# Cache:\t%s\n", CACHE);
    fprintf( fp, "# \n" );
    fprintf( fp, "# C compiler:\t%s\n", COMPILER );
    fprintf( fp, "# C language standard:\t%s\n", LANGUAGE );
    fprintf( fp, "# Optimization level and options:\t%s\n", OPTM );
    fprintf( fp, "# Clock resolution:\t%Lg\n", timer_resolution() );
    fprintf( fp, "# \n" );
#if defined(MULTALGO)
    fprintf( fp, "# Sub-block dimension (kernel multiplication):\t%d\n", KDIM );
    fprintf( fp, "# Depth of loop unrolling:\t%d\n", UNROLL_DEPTH );
    fprintf( fp, "# Depth of software pipelining:\t%d\n", PIPE_DEPTH );
#elif defined(CCOPTMDP)
    fprintf( fp, "# Dot product (ijk indexing) algorithm\n" );
#elif defined(CCOPTMSA)
    fprintf( fp, "# Scalar alpha x plus y (jki indexing) algorithm\n" );
#endif
    fprintf( fp, "# \n" );
    fprintf( fp, "%s\n", hdr_text );
    // Write experimental data
    for ( int i = 0; i < rows; i++ ) {
        for ( int j = 0; j < cols; j++ ) {
            fprintf( fp, "%g\t", *(data+j*rows+i) );
        }
        fprintf( fp, "\n" );
    }
    fclose( fp );
}

/*
 * Measures the average time (number of seconds) to perform matrix
 * multiplication (and addition), C = C + A*B, on n-by-n matrices. Matrix
 * multiplication is performed iteratively for at least the minimum number of
 * iterations, and until the minimum time (in seconds) has elapsed.
 */
double time_mmult( void (*mmult)(int n, const double *A, const double *B,
    double *C), int n, const double *A, const double *B, double *C )
{
    struct      timespec sta, end;
    long int    num_iter = MIN_ITER;
    double      secs = -1.0;
    double      *M;

    // Save copy of matrix C before performing matrix multiplication
    M = (double *) malloc( n*n*sizeof(double) );
    copy_matrix( n, n, C, M );

    while ( secs < MIN_SECS ) {
        get_time( &sta );
        for ( int i = 0; i < num_iter; i++ ) {
            mmult( n, A, B, C );
            copy_matrix( n, n, M, C );  // Reset matrix C to initial value
        }
        get_time( &end );
        secs = timespec_diff( sta, end );
        num_iter *= 2;
    }
    free( M );
    // On exiting the while loop, the number of iterations (num_iter) has been
    // doubled in the event that secs < MIN_SECS, so num_iter must be halved
    return secs / (num_iter/2.0);
}

/*
 * Measures the performance (Mflops/sec) of unblocked and blocked algorithms
 * performing matrix multiplication (and addition), C = C + A*B, on square
 * matrices over a range of dimensions.
 */
void time_mmult_algo( void )
{
#define FIELDS 11   // Number of output data fields
    const char  *data_file_name = "mmult.dat";
    const char  *hdr_text =
"# N:       Matrix dimension, n-by-n\n"
"# BDIM:    Block dimension used by blocking algorithms\n"
"#          Mflop/sec for matrix multiplication algorithms\n"
"# DOTPROD: Dot product, ijk indexing\n"
"# SAXPY:   Scalar alpha x plus y, jki indexing\n"
"# UNROLL:  Loop unrolling, dot product\n"
"# PIPELN:  Software pipelining, SAXPY\n"
"# BLKSIMP: Simple blocking\n"
"# BLKCTG:  Contiguous blocking\n"
"# BLKRCR:  Recursive contiguous blocking\n"
"# RCRRECT: Recursive contiguous blocking, variable sub-block sizes\n"
"# BLAS:    BLAS routine DGEMM\n"
"# \n"
"# N\tBDIM\tDOTPROD\tSAXPY\tUNROLL\tPIPELN\tBLKSIMP\tBLKCTG\tBLKRCR"
"\tRCRRECT\tBLAS";
    const int       col_n = 0,
                    col_bdim = 1,
                    col_dotprod = 2,
                    col_saxpy = 3,
                    col_unroll = 4,
                    col_pipeln = 5,
                    col_blksimp = 6,
                    col_blkctg = 7,
                    col_blkrcr = 8,
                    col_rcrrect = 9,
                    col_blas = 10;
    const double    alpha = 10.0;   // Scaling factor for random matrix

    char     *data_file;
    int      n, bdim;
    double   mflops;
    double   perf_data[FIELDS*SIZES];
    double   *A, *B, *C;
    void     (*mmult)( int n, const double *A, const double *B, double *C );

    // Concatenate file path and name
    data_file = (char *) calloc( strlen(file_path) + strlen(data_file_name) + 1,
        sizeof(char) );
    strcpy( data_file, file_path );
    strcat( data_file, data_file_name );

    for ( int i = 0; i < SIZES; i++ ) {
        n = mat_size[i];
        bdim = get_block_dim_mmult( n );
        fprintf( stdout, "n = %d, bdim = %d\n", n, bdim );
        // Matrix multiplication takes 2*n^3 floating point operations
        mflops = 1.0e-06 * 2.0 * n * n * n;
        // Create random n-by-n matrices
        A = (double *) malloc( n*n*sizeof(double) );
        B = (double *) malloc( n*n*sizeof(double) );
        C = (double *) malloc( n*n*sizeof(double) );
        create_random_matrix( alpha, n, n, A );
        create_random_matrix( alpha, n, n, B );
        create_random_matrix( alpha, n, n, C );

        // Performance data is stored in perf_data[] array in column-major order
        perf_data[i+col_n*SIZES] = (double) n;
        perf_data[i+col_bdim*SIZES] = (double) bdim;
printf("mmult_outer_product\n");
        // Measure performance of matrix multiplication algorithms:
        // Dot product, ijk indexing
        mmult = mmult_dot_product;
        perf_data[i+col_dotprod*SIZES] = 0.0;//mflops / time_mmult( mmult, n, A, B, C );
printf("mmult_saxpy\n");
        // Scalar alpha x plus y, jki indexing
        mmult = mmult_saxpy;
        perf_data[i+col_saxpy*SIZES] = 0.0;//mflops / time_mmult( mmult, n, A, B, C );
printf("mmult_unroll\n");
        // Loop unrolling, dot product
        mmult = mmult_unroll;
        perf_data[i+col_unroll*SIZES] = 0.0;//mflops / time_mmult( mmult, n, A, B, C );
printf("mmult_pipeline\n");
        // Software pipelining, SAXPY
        mmult = mmult_pipeline;
        perf_data[i+col_pipeln*SIZES] = 0.0;//mflops / time_mmult( mmult, n, A, B, C );
printf("mmult_block\n");
        // Simple blocking
        mmult = mmult_block;
        perf_data[i+col_blksimp*SIZES] = mflops / time_mmult( mmult, n, A, B, C );
printf("mmult_contig_block\n");
        // Blocking, contiguous block storage
        mmult = mmult_contig_block;
        perf_data[i+col_blkctg*SIZES] = 0.0;//mflops / time_mmult( mmult, n, A, B, C );
printf("mmult_recur_block\n");
        // Blocking, recursive contiguous blocking
        mmult = mmult_recur_block;
        perf_data[i+col_blkrcr*SIZES] = mflops / time_mmult( mmult, n, A, B, C );
printf("mmult_rect_recur_block\n");
        // Blocking, recursive contiguous blocking, variable looping
        mmult = mmult_rect_recur_block;
        perf_data[i+col_rcrrect*SIZES] = 0.0;//mflops / time_mmult( mmult, n, A, B, C );
printf("mmult_blas\n");
        // BLAS routine DGEMM
        mmult = mmult_blas;
        perf_data[i+col_blas*SIZES] = mflops / time_mmult( mmult, n, A, B, C );

        free( A );
        free( B );
        free( C );
    }

    write_data_file( data_file, hdr_text, SIZES, FIELDS, perf_data );

#undef FIELDS
}

/*
 * Measures the performance (Mflops/sec) of matrix multiplication (and addition),
 * C = C + A*B, on square matrices over a range of dimensions.  Performance is
 * measured for different compiler optimization levels and options.
 */
void time_compiler_optm( void )
{
#define FIELDS 2    // Number of output data fields
    const char      *data_file_suffix = ".dat",
                    *delim = " -";
    const char      *hdr_text =
"# N:       Matrix dimension, N-by-N\n"
"# PERF:    Mflop/sec for matrix multiplication algorithm\n"
"# \n"
"# N\tPERF";
```

```
    const int       col_n = 0,
                    col_perf = 1;
    const double    alpha = 10.0;   // Scaling factor for random matrix

    char     *data_file, *optm_lvl, *optm_str, *token;
    int      n;
    double   mflops;
    double   perf_data[FIELDS*SIZES];
    double   *A, *B, *C;
    void     (*mmult)( int n, const double *A, const double *B, double *C );

#if defined(CCOPTMDP)
    const char *data_file_prefix = "mmult_dot_ccoptm_";
    mmult = mmult_dot_product;
#elif defined(CCOPTMSA)
    const char *data_file_prefix = "mmult_saxpy_ccoptm_";
    mmult = mmult_saxpy;
#else
    const char *data_file_prefix = "mmult_saxpy_ccoptm_";
    mmult = mmult_saxpy;
#endif

    // Format optimization level and options as a string
    optm_lvl = (char *) calloc( strlen(OPTM) + 1, sizeof(char) );
    strcpy( optm_lvl, OPTM );
    optm_str = (char *) calloc( strlen(optm_lvl) + 1, sizeof(char) );
    if ( (token = strtok( optm_lvl, delim )) != NULL ) {
        strcpy( optm_str, token );
        while ( (token = strtok( NULL, delim )) != NULL ) {
            strcat( optm_str, token );
        }
    }
    // Concatenate file path and names
    data_file = (char *) calloc( strlen(file_path) + strlen(data_file_prefix) +
        strlen(optm_str) + strlen(data_file_suffix) + 1, sizeof(char) );
    strcpy( data_file, file_path );
    strcat( data_file, data_file_prefix );
    strcat( data_file, optm_str );
    strcat( data_file, data_file_suffix );

    for ( int i = 0; i < SIZES; i++ ) {
        n = mat_size[i];
        fprintf( stdout, "n = %d\n", n );
        // Matrix multiplication takes 2*n^3 floating point operations
        mflops = 1.0e-06 * 2.0 * n * n * n;
        // Create random n-by-n matrices
        A = (double *) malloc( n*n*sizeof(double) );
        B = (double *) malloc( n*n*sizeof(double) );
        C = (double *) malloc( n*n*sizeof(double) );
        create_random_matrix( alpha, n, n, A );
        create_random_matrix( alpha, n, n, B );
        create_random_matrix( alpha, n, n, C );
        // Matrix multiplication takes 2*n^3 floating point operations
        mflops = 1.0e-06 * 2.0 * n * n * n;

        perf_data[i+col_n*SIZES] = n;
        perf_data[i+col_perf*SIZES] = mflops / time_mmult( mmult, n, A, B, C );

        free( A );
        free( B );
        free( C );
    }

    write_data_file( data_file, hdr_text, SIZES, FIELDS, perf_data );

#undef FIELDS
}
```

A.11. mmultmp.c – timing harness for parallel matrix multiplication.

```
/*
 * Timing hareness for measuring the performance of parallel algorithms
 * implementing matrix multiplication (and addition), C = C + A*B, on square
 * matrices over a range of dimensions.  Performance data are written to an
 * output file destination.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <mpi.h>

#include "timing.h"
#include "matmultp.h"

#if !defined(PROC)
#   define PROC "unknown"
#endif
#if !defined(CORES)
#   define CORES "unknown"
#endif
#if !defined(CLKSPEED)
#   define CLKSPEED "unknown"
#endif
#if !defined(CACHE)
#   define CACHE "unknown"
#endif
#if !defined(COMPILER)
#   define COMPILER "unknown"
#endif
#if !defined(LANGUAGE)
#   define LANGUAGE "default"
#endif
#if !defined(OPTM)
#   define OPTM "default"
#endif
#if !defined(DATADIR)
#   define DATADIR "."  // Current directory ./
#endif

#define MIN_ITER 8       // Minimum number of iterations of algorithm
#define MIN_SECS 2.0     // Minimum elapsed time for execution of algorithm

static void write_data_file( const char *file, const char *hdr_text,
    int rows, int cols, const double *data, struct mpi_grid *grid );
static void read_data_file( const char *file, int rows, int cols, double *data );
static double time_matmultp( void (*matmultp)(int n, const double *A, const double *B,
    double *C, struct mpi_grid *grid), int n, const double *A, const double *B,
    double *C, struct mpi_grid *grid );
static void time_parallel_matrix_multiply( struct mpi_grid *grid  );

static char             *file_path;
static struct mpi_grid  grid;

int main( int argc, char **argv )
{
    const double    alpha = 10.0;   // Scaling factor for random matrix

    MPI_Init( &argc, &argv );
    //Establish Cartesian topology for collective communication
    setup_mpi_grid( &grid );

    // Specify file path for input and output data files
    file_path = (char *) calloc( strlen(DATADIR) + 2, sizeof(char) );
    strcpy( file_path, DATADIR );
    strcat( file_path, "/" );

    time_parallel_matrix_multiply( &grid );

    MPI_Finalize();
    return 0;
}

/***************************************************************************/

/*
 * Writes header text and experimental data to the file specified in the
 * argument list.  Experimental data is enumerated in a matrix stored in
 * column-major order.
 */
void write_data_file( const char *file, const char *hdr_text,
    int rows, int cols, const double *data, struct mpi_grid *grid )
{
    FILE *fp;

    if ( (fp = fopen( file, "w" )) == NULL ) {
        fprintf( stderr, "Error opening file %s.", file );
        exit( -1 );
    }
    // Write header text
    fprintf( fp, "# Processor:\t%s\n", PROC);
    fprintf( fp, "# Cores:\t%s\n", CORES);
    fprintf( fp, "# Clock speed:\t%s\n", CLKSPEED);
    fprintf( fp, "# Cache:\t%s\n", CACHE);
    fprintf( fp, "# \n" );
    fprintf( fp, "# C compiler:\t%s\n", COMPILER );
    fprintf( fp, "# C language standard:\t%s\n", LANGUAGE );
    fprintf( fp, "# Optimization level and options:\t%s\n", OPTM );
    fprintf( fp, "# Clock resolution:\t%Lg\n", timer_resolution() );
    fprintf( fp, "# Number of processors:\t%d\n", grid->p);
    fprintf( fp, "# \n" );
    fprintf( fp, "%s\n", hdr_text );
    // Write experimental data
    for ( int i = 0; i < rows; i++ ) {
        for ( int j = 0; j < cols; j++ ) {
            fprintf( fp, "%g\t", *(data+j*rows+i) );
        }
        fprintf( fp, "\n" );
    }
    fclose( fp );
}

/*
 * Reads matrix data in specified file into an array passed in argument list.
 * Data read from the file is stored in the array in column-major order.
 */
void read_data_file( const char *file, int rows, int cols, double *data )
{
    FILE *fp;

    if ( (fp = fopen(file, "r")) == NULL ) {
        fprintf( stderr, "Error opening file %s.", file );
        exit(-1);
    }
    for ( int i = 0; i < rows; i++ ) {
        for ( int j = 0; j < cols; j++ ) {
            fscanf( fp, "%lg", (data+j*rows+i) );
        }
    }
    fclose(fp);
}

/*
 * Measures the average time (number of seconds) to perform parallel matrix
 * multiplication (and addition), C = C + A*B, on n-by-n matrices. Parallel
 * matrix  multiplication is performed iteratively for at least the minimum
 * number of iterations, and until the minimum time (in seconds) has elapsed.
 */
double time_matmultp( void (*matmultp)(int n, const double *A, const double *B,
    double *C, struct mpi_grid *grid), int n, const double *A, const double *B,
    double *C, struct mpi_grid *grid )
{
    struct      timespec sta, end;
    long int    num_iter = MIN_ITER;
    double      secs = -1.0;
    double      *M;

    // Save copy of matrix C before performing matrix multiplication
    if (grid->rank == 0) {
        M = (double *) malloc( n*n*sizeof(double) );
        copy_matrix( n, n, C, M );
    }

    while ( secs < MIN_SECS ) {
        get_time( &sta );
        for ( int i = 0; i < num_iter; i++ ) {
            parallel_matrix_multiply( n, A, B, C, grid );
        }
        get_time( &end );
        secs = timespec_diff( sta, end );
        num_iter *= 2;
        if ( grid->rank == 0 ) {
            copy_matrix( n, n, M, C );       // Reset matrix C to initial value
        }
    }
    if (grid->rank == 0) {
        free( M );
    }
    // On exiting the while loop, the number of iterations (num_iter) has been
    // doubled in the event that secs < MIN_SECS, so num_iter must be halved
    return secs / (num_iter/2.0);
}

/*
 * Measures the performance of parallel matrix multiplication (and addition),
 * C = C + A*B, on square matrices over a range of dimensions.
 */
void time_parallel_matrix_multiply( struct mpi_grid *grid  )
{
#define IN_FIELDS 11         // Number of input data fields
#define OUT_FIELDS 5         // Number of output data fields
    const char  *out_file_prefix = "mmult_",
                *out_file_ext = ".dat",
                *in_file_name = "mmult.dat";
    const char  *hdr_text =
"# N:       Matrix dimension, n-by-n\n"
"# SERIAL:  Mflop/sec for serial matrix multiplication, simple blocking\n"
"# PARA:    Mflop/sec for parallel matrix multiplication, simple blocking\n"
"# SPEEDUP: Speed-up\n"
"# EFFNCY:  Efficiency\n"
"# \n"
"# N\tSERIAL\tPARA\tSPEEDUP\tEFFNCY";
    const int       col_n = 0,
                    col_serial = 1,
                    col_para = 2,
                    col_speedup = 3,
                    col_effncy = 4,
                    col_in_serial = COLINSER;
```

```
const double    alpha = 10.0;   // Scaling factor for random matrix

char      procs[4];
char      *out_data_file, *in_data_file;
int       n;
double    mflops, mflop_sec;
double    perf_data[OUT_FIELDS*SIZES],
          perf_serial[IN_FIELDS*SIZES];
double    *A, *B, *C;
void      (*matmultp)( int n, const double *A, const double *B, double *C,
              struct mpi_grid *grid );

// Concatenate file path and names
sprintf(procs, "np%d", grid->p);
out_data_file = (char *) calloc( strlen(file_path) + strlen(out_file_prefix)
    + strlen(procs) + strlen(out_file_ext) + 1, sizeof(char) );
strcpy( out_data_file, file_path );
strcat( out_data_file, out_file_prefix );
strcat( out_data_file, procs );
strcat( out_data_file, out_file_ext );
in_data_file = (char *) calloc( strlen(file_path) +
    strlen(in_file_name) + 1, sizeof(char) );
strcpy( in_data_file, file_path );
strcat( in_data_file, in_file_name );
// Read input data file (performance of serial matrix multiplication)
read_data_file( in_data_file, SIZES, IN_FIELDS, perf_serial );

for ( int i = 0; i < SIZES; i++ ) {
    n = perf_serial[i+col_n*SIZES];
    // Matrix multiplication takes 2*n^3 floating point operations
    mflops = 1.0e-06 * 2.0 * n * n * n;
    if ( grid->rank == 0 ) {
        fprintf( stdout, "n = %d\n", n );
        // Allocate memory for matrices
        A = (double *) malloc( n*n*sizeof(double) );
        B = (double *) malloc( n*n*sizeof(double) );
        C = (double *) malloc( n*n*sizeof(double) );
        // Create random matrices A, B and C
        create_random_matrix( alpha, n, n, A );
        create_random_matrix( alpha, n, n, B );
        create_random_matrix( alpha, n, n, C );
    }
    matmultp = parallel_matrix_multiply;
    mflop_sec = mflops / time_matmultp( matmultp, n, A, B, C, grid );
    if ( grid->rank == 0 ) {
        perf_data[i+col_n*SIZES] = perf_serial[i+col_n*SIZES];
        perf_data[i+col_serial*SIZES] = perf_serial[i+col_in_serial*SIZES];
        perf_data[i+col_para*SIZES] = mflop_sec;
        perf_data[i+col_speedup*SIZES] =
            mflop_sec / perf_data[i+col_serial*SIZES];
            perf_data[i+col_effncy*SIZES] =
                perf_data[i+col_speedup*SIZES] / grid->p;
            free( A );
            free( B );
            free( C );
        }
    }

    if ( grid->rank == 0 ) {
        write_data_file( out_data_file, hdr_text,
                SIZES, OUT_FIELDS, perf_data, grid );
    }
#undef FIELDS
}
```

A.12. mfactest.c – testing harness for matrix factorization.

```
/*
 * Testing harness for unblocked and blocked algorithms implementing the
 * factorization of matrices representing linear systems. Matrix factorizations
 * include LU (Gaussian elimination), standard Cholesky, symmetric indefinite
 * (LDL'), and modified Choleksy (Gill-Murray-Wright and Cheng-Higham
 * algorithms). The number of tests and error count are accumulated through a
 * single execution of the mfactest program, and all test results are written to
 * an output file destination (terminal).
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <float.h>

#include "lufact.h"
#include "cholfact.h"
#include "ldltfact.h"
#include "modchol.h"
#include "matcom.h"
#include "lapack.h"

static void test_assert( double eps, double tol, const char *test_name );
static void test_lu_outer_product( void );
static void test_lu_saxpy( void );
static void test_lu_block( void );
static void test_lu_recur_block( void );
static void test_lu_pivot_outer_product( void );
static void test_lu_pivot_saxpy( void );
static void test_lu_pivot_block( void );
static void test_lu_pivot_lapack( void );
static void test_chol_outer_product( void );
static void test_chol_saxpy( void );
static void test_chol_block( void );
static void test_chol_rect_block( void );
static void test_chol_contig_block( void );
static void test_chol_recur_block( void );
static void test_chol_block_blas( void );
static void test_chol_contig_block_blas( void );
static void test_chol_lapack( void );
static void test_ldlt_outer_product (void );
static void test_ldlt_saxpy( void );
static void test_ldlt_block( void );
static void test_ldlt_block_blas( void );
static void test_ldlt_lapack( void );
static void test_chol_gmw_outer_product( void );
static void test_chol_gmw_saxpy( void );
static void test_chol_gmw_block( void );
static void test_chol_gmw_block_blas( void );
static void test_chol_ch_outer_product( void );
static void test_chol_ch_saxpy( void );
static void test_chol_ch_block( void );
static void test_chol_ch_block_blas( void );
static void print_matrix( int m, int n, int ldim, const double *E );

static int      tests = 0,          // Test count
                errs = 0;           // Error count
static FILE     *fp;

int main()
{
    fp = stdout;

    // Test LU factorization
#if defined(LUFACT)
    test_lu_outer_product();
    test_lu_saxpy();
    test_lu_block();
    test_lu_recur_block();
#endif

    // Test LU factorization with partial pivoting
#if defined(LUPIVOT)
    test_lu_pivot_outer_product();
    test_lu_pivot_saxpy();
    test_lu_pivot_block();
    test_lu_pivot_lapack();
#endif

    // Test Cholesky factorization
#if defined(CHOLFACT)
    test_chol_outer_product();
    test_chol_saxpy();
    test_chol_block();
    test_chol_rect_block();
    test_chol_contig_block();
    test_chol_recur_block();
    test_chol_block_blas();
    test_chol_contig_block_blas();
    test_chol_lapack();
#endif

    // Test LDL' factorization
#if defined(LDLTFACT)
    test_ldlt_outer_product();
    test_ldlt_saxpy();
    test_ldlt_block();
    test_ldlt_block_blas();
    test_ldlt_lapack();
#endif

    // Test modified Cholesky factorization
#if defined(MODCHOL)
    test_chol_gmw_outer_product();
    test_chol_gmw_saxpy();
    test_chol_gmw_block();
    test_chol_gmw_block_blas();
    test_chol_ch_outer_product();
    test_chol_ch_saxpy();
    test_chol_ch_block();
    test_chol_ch_block_blas();
#endif

    if ( errs == 0 ) {
        fprintf( fp, "Passed all %d tests.\n", tests );
    } else {
        fprintf( fp, "Total of %d error(s) encountered in %d tests.\n",
            errs, tests );
    }
    return 0;
}

/*
 * Verifies that test results are accurate within specified tolerance, and
 * prints message indicating whether the routine passed or failed the test.
 */
void test_assert( double eps, double tol, char *test_name )
{
    tests++;
    if ( eps <= tol ) {
        fprintf( fp, "PASSED: %s\n(eps=%e <= tol=%e)\n", test_name, eps, tol );
    } else {
        fprintf( fp, "FAILED: %s\n(eps=%e > tol=%e)\n", test_name, eps, tol );
        errs++;
    }
}

/**************************************************************************/

/*
 * Checks whether the outer product method (kji indexing) for LU factorization
 * is performed correctly on nonsingular n-by-n matrices.  Matrix A represents
 * an n-by-n linear system and matrix LU stores the correct unit lower and upper
 * triangular factors.  Randomly generated nonsingular matrices are diagonally
 * dominant, so pivoting in not required.
 */
void test_lu_outer_product( void )
{
```

```
const double     tol = 1e-12;         // Error tolerance

char     test_name[80];
int      n;
double   eps, err;
double   A[] =   { 2, 0.5, 0.25, 0.25, 4.0625, 2.03125, 0.5, 0.25, 3.125 },
         LUA[] = { 2, 0.25, 0.125, 0.25, 4, 0.5, 0.5, 0.125, 3 },
         B[] =   { 6, 18, 12, 24, 1, 7, 26, 12,
                   3, 11, 23, 41, 2, 6, 5, 16 },
         LUB[] = { 6, 3, 2, 4, 1, 4, 6, 2,
                   3, 2, 5, 5, 2, 0, 1, 3 };
double   C[] =   {  3,  6,  3, 15,  9,  0,  3,  9, 15, 15,
                    2,  5,  5, 12, 10,  1,  4, 11, 14, 15,
                    1,  7, 18, 25, 29, 13, 19, 32, 31, 32,
                    2,  9, 18, 26, 29, 13, 19, 35, 38, 36,
                    4, 11, 15, 39, 33, 29, 33, 46, 65, 49,
                    2,  4,  7, 40, 23, 46, 51, 36, 64, 25,
                    4, 10, 11, 31, 23, 15, 28, 50, 69, 37,
                    1,  4, 11, 29, 23, 19, 29, 38, 52, 29,
                    3,  7, 10, 38, 28, 28, 37, 46, 72, 51,
                    5, 14, 19, 45, 39, 26, 44, 68, 99, 82 },
         LUC[] = {  3, 2, 1, 5, 3, 0, 1, 3, 5, 5,
                    2, 1, 3, 2, 4, 1, 2, 5, 4, 5,
                    1, 5, 2, 5, 3, 4, 4, 2, 3, 1,
                    2, 5, 1, 1, 0, 4, 3, 2, 5, 0,
                    4, 3, 2, 3, 3, 2, 2, 3, 4, 4,
                    2, 0, 5, 5, 2, 2, 5, 2, 3, 1,
                    4, 2, 1, 2, 0, 1, 5, 4, 5, 1,
                    1, 2, 4, 0, 0, 1, 3, 3, 3, 2,
                    3, 1, 4, 1, 3, 1, 2, 3, 2, 3,
                    5, 4, 2, 2, 2, 2, 3, 3, 4, 4 };

n = 3;
sprintf( test_name, "LU factorization, outer product, %dx%d matrix", n, n );
// Perform LU factorization and compare result with correct answer
lu_outer_product( n, A );
error_matrix_comp_frob( &eps, &err, n, n, LUA, A );
test_assert( eps, tol, test_name );

n = 4;
sprintf( test_name, "LU factorization, outer product, %dx%d matrix", n, n );
// Perform LU factorization and compare result with correct answer
lu_outer_product( n, B );
error_matrix_comp_frob( &eps, &err, n, n, LUB, B );
test_assert( eps, tol, test_name );

n = 10;
sprintf( test_name, "LU factorization, outer product, %dx%d matrix", n, n );
// Perform LU factorization and compare result with correct answer
lu_outer_product( n, C );
```

```
    error_matrix_comp_frob( &eps, &err, n, n, LUC, C );
    test_assert( eps, tol, test_name );
}

/*
 * Checks whether an implementation of the SAXPY operation (jki indexing) for
 * LU factorization is performed correctly on an n-by-n nonsingular matrix.
 * The result is verified against that produced by the outer product method.
 * Randomly generated nonsingular matrices are diagonally dominant, so pivoting
 * in not required.
 */
void test_lu_saxpy( void )
{
    const int       n = 12;          // n-by-n matrix A
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 1.0;     // Scaling factor for random matrix

    char    test_name[80];
    double  eps, err;
    double  *A, *LU;

    sprintf( test_name, "LU factorization, SAXPY, %dx%d matrix", n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    LU = (double *) malloc( n*n*sizeof(double) );
    create_random_nonsingular( alpha, n, A );
    copy_matrix( n, n, A, LU );

    // Perform LU factorization and compare result with outer product solution
    lu_saxpy( n, A );
    lu_outer_product( n, LU );
    error_matrix_comp_frob( &eps, &err, n, n, LU, A );
    test_assert( eps, tol, test_name );
    free( A );
    free( LU );
}

/*
 * Checks whether simple blocking for LU factorization is performed correctly
 * on n-by-n nonsingular matrices.  The results are verified against those
 * produced by the outer product method. Randomly generated nonsingular matrices
 * are diagonally dominant, so pivoting in not required.
 */
void test_lu_block( void )
{
    const int       mat_size[] = { 12, 64, 82 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 1.0;     // Scaling factor for random matrix

    char    *test_name[SIZES];
    double  eps, err;
    double  *A, *LU;

    // Define test names
    test_name[0] = "LU factorization, simple blocking --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "LU factorization, simple blocking --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "LU factorization, simple blocking --\n"
        "matrix dimension not a multiple of block dimension";

    for ( int i = 0; i < SIZES; i++ ) {
        int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        LU = (double *) malloc( n*n*sizeof(double) );
        create_random_nonsingular( alpha, n, A );
        copy_matrix( n, n, A, LU );
        // Perform LU factorization, compare result with outer product solution
        lu_block( n, A );
        lu_outer_product( n, LU );
        error_matrix_comp_frob( &eps, &err, n, n, LU, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( LU );
    }
#undef SIZES
}

/*
 * Checks whether recursive contiguous blocking for LU factorization is
 * performed correctly on n-by-n nonsingular matrices.  The results are verified
 * against those produced by the outer product method.  Randomly generated
 * nonsingular matrices are diagonally dominant, so pivoting is not required.
 */
void test_lu_recur_block( void )
{
    const int       mat_size[] = { 22, 96, 111 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,    // Error tolerance
                    alpha = 1.0;    // Scaling factor for random matrix

    char    *test_name[SIZES];
    double  eps, err;
    double  *A, *LU;

    // Define test names
    test_name[0] = "LU factorization, recursive contiguous blocking --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "LU factorization, recursive contiguous blocking --\n"
            "matrix dimension a multiple of block dimension";
    test_name[2] = "LU factorization, recursive contiguous blocking --\n"
            "matrix dimension not a multiple of block dimension";

    for ( int i = 0; i < SIZES; i++ ) {
        int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        LU = (double *) malloc( n*n*sizeof(double) );
        create_random_nonsingular( alpha, n, A );
        copy_matrix( n, n, A, LU );
        // Perform LU factorization, compare result with outer product solution
        lu_recur_block( n, A );
        lu_outer_product( n, LU );
        error_matrix_comp_frob( &eps, &err, n, n, LU, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( LU );
    }
#undef SIZES
}

/*****************************************************************************/

/*
 * Checks whether the outer product method (kji indexing) for LU factorization
 * with partial pivoting is performed correctly on n-by-n nonsingular matrices.
 * Matrix A represents an n-by-n linear system and matrix LU stores the correct
 * unit lower and upper triangular factors.
 */
void test_lu_pivot_outer_product( void )
{
    const double    tol = 1e-12;        // Error tolerance

    char    test_name[80];
    int     n;
    int     *piv, *ord;
    double  eps, err;
    double  A[] =   { 3, 2, 6, 17, 4, 18, 10, -2, -12 },
            LUA[] = { 6, 0.5, 1/3.0, 18, 8, -0.25, -12, 16, 6 },
            B[] =   { 6, 18, 12, 24, 1, 7, 26, 12,
                      3, 11, 23, 41, 2, 6, 5, 16 },
            LUB[] = { 24, 0.50, 0.75, 0.25, 12, 20, -0.10, -0.10,
                      41, 2.50, -19.50, 14/39.0, 16, -3, -6.30, -1/26.0 };

    n = 3;
    sprintf( test_name,
"LU factorization with partial pivoting, outer product, %dx%d matrix", n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Perform LU factorization and compare result with correct answer
    lu_pivot_outer_product( 'G', n, piv, ord, A );
    error_matrix_comp_frob( &eps, &err, n, n, LUA, A );
    test_assert( eps, tol, test_name );
    free( piv );
    free( ord );

    n = 4;
    sprintf( test_name,
"LU factorization with partial pivoting, outer product, %dx%d matrix", n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Perform LU factorization and compare result with correct answer
    lu_pivot_outer_product( 'G', n, piv, ord, B );
    error_matrix_comp_frob( &eps, &err, n, n, LUB, B );
    test_assert( eps, tol, test_name );
    free( piv );
    free( ord );
}

/*
 * Checks whether an implementation of the SAXPY operation (jki indexing)
 * for LU factorization with partial pivoting is performed correctly on n-by-n
 * nonsingular matrices.  The result is verified against that produced by the
 * outer product method.
 */
void test_lu_pivot_saxpy( void )
{
    const int       n = 14;          // n-by-n matrix A
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 10.0;    // Scaling factor for random matrix

    char     test_name[80];
    int      *piv, *ord;
    double   eps, err;
    double   *A, *LU;

    sprintf( test_name,
        "LU factorization with partial pivoting, SAXPY, %dx%d matrix", n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    LU = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_nonsingular( alpha, n, A );
    copy_matrix( n, n, A, LU );

    // Perform LU factorization and compare result with outer product solution
    lu_pivot_saxpy( 'G', n, piv, ord, A );
    lu_pivot_outer_product( 'G', n, piv, ord, LU );
    error_matrix_comp_frob( &eps, &err, n, n, LU, A );
    test_assert( eps, tol, test_name );
    free( A );
    free( LU );
    free( piv );
    free( ord );
}

/*
 * Checks whether simple blocking for LU factorization with partial pivoting
 * is performed correctly on n-by-n nonsingular matrices.  The results are
 * verified against those produced by the outer product method.
 */
void test_lu_pivot_block( void )
{
    const int       mat_size[] = { 14, 48, 82 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,    // Error tolerance
                    alpha = 10.0;   // Scaling factor for random matrix

    int     *piv, *ord;
    char    *test_name[SIZES];
    double  eps, err;
    double  *A, *LU;

    // Define test names
    test_name[0] = "LU factorization with partial pivoting, simple blocking --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "LU factorization with partial pivoting, simple blocking --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "LU factorization with partial pivoting, simple blocking --\n"
        "matrix dimension not a multiple of block dimension";

    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        LU = (double *) malloc( n*n*sizeof(double) );
        piv = (int *) malloc( n*sizeof(int) );
        ord = (int *) malloc( n*sizeof(int) );
        create_random_nonsingular( alpha, n, A );
        copy_matrix( n, n, A, LU );

        // Perform LU factorization, compare result with outer product solution
        lu_pivot_block( 'G', n, piv, ord, A );
        lu_pivot_outer_product( 'G', n, piv, ord, LU );
        error_matrix_comp_frob( &eps, &err, n, n, LU, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( LU );
        free( piv );
        free( ord );
    }
#undef SIZES
```

```
}

/*
 * Checks whether the wrapper function properly invokes LAPACK routine DGETRF,
 * which computes an LU factorization of a nonsingular matrix using partial
 * pivoting with row interchanges.
 */
void test_lu_pivot_lapack( void )
{
    const int       n = 42;          // n-by-n matrix A
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 10.0;    // Scaling factor for random matrix

    char    test_name[80];
    int     *piv, *ord;
    double  eps, err, normA, normLU;
    double  *A, *LU;

    sprintf( test_name,
        "LU factorization, LAPACK routine DGETRF, %dx%d matrix", n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    LU = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_nonsingular( alpha, n, A );
    copy_matrix( n, n, A, LU );

    // Perform LU factorization and compare result with outer product solution
    lu_pivot_lapack( 'G', n, piv, ord, A );
    lu_pivot_outer_product( 'G', n, piv, ord, LU );
    // The factorization produced by LAPACK routine DGETRF takes the form
    // A = P*L*U, whereas the factorization produced by lu_pivot_outer_product()
    // takes the form P*A = L*U.  Therefore, verify that DGETRF is invoked
    // correctly by comparing the norms of factors (matrices) computed by DGETRF
    // and lu_pivot_outer_product().
    normA = 0.0;
    normLU = 0.0;
    for ( int j = 0; j < n; j++ ) {
        for ( int i = 0; i < n; i++ ) {
            double aij = fabs( *(A + i + j*n) );
            double lij = fabs( *(LU + i + j*n) );
            normA += aij;
            normLU += lij;
        }
    }
    err = abs( normA - normLU );
    eps = err / normA;
    test_assert( eps, tol, test_name );
    free( A );
    free( LU );
    free( piv );
    free( ord );
}

/****************************************************************************/

/*
 * Checks whether the outer product method (kji indexing) for Cholesky
 * factorization is performed correctly on n-by-n symmetric positive definite
 * matrices.  Cholesky factorization, A = L*L', computes a unique lower
 * triangular factor.  Matrix L stores the correct lower triangular factor.
 */
void test_chol_outer_product( void )
{
    const double    tol = 1e-12;        // Error tolerance

    char    test_name[80];
    int     n;
    double  eps, err;
    double  A[] =   { 4, -2, -6, -2, 10, 9, -6, 9, 14 },
            LA[] =  { 2, -1, -3, -2, 3, 2, -6, 9, 1 },
            B[] =   { 16, 8, 12, 8, 8, 29, 11, 24,
                      12, 11, 46, 22, 8, 24, 22, 33 },
            LB[] =  { 4, 2, 3, 2, 8, 5, 1, 4,
                      12, 11, 6, 2, 8, 24, 22, 3 };
    double  C[] =   { 25, 10, 15,  0, 20, 10, 25,  5,
                      10,  8, 14,  8, 14,  4, 18, 12,
                      15, 14, 29, 24, 30,  8, 33, 29,
                       0,  8, 24, 33, 28,  6, 23, 33,
                      20, 14, 30, 28, 66, 23, 47, 44,
                      10,  4,  8,  6, 23, 19, 32, 19,
                      25, 18, 33, 23, 47, 32, 92, 54,
                       5, 12, 29, 33, 44, 19, 54, 62 },
            LC[] =  {  5,  2,  3,  0,  4,  2,  5,  1,
                      10,  2,  4,  4,  3,  0,  4,  5,
                      15, 14,  2,  4,  3,  1,  1,  3,
                       0,  8, 24,  1,  4,  2 , 3,  1,
                      20, 14, 30, 28,  4,  1,  0,  3,
                      10,  4,  8,  6, 23,  3,  5,  3,
                      25, 18, 33, 23, 47, 32,  4,  2,
                       5, 12, 29, 33, 44, 19, 54,  2 };

    n = 3;
    sprintf( test_name,
        "Cholesky factorization, outer product, %dx%d matrix", n, n );
    // Perform Cholesky factorization and compare result with correct answer
    chol_outer_product( n, A );
    error_matrix_comp_frob( &eps, &err, n, n, LA, A );
    test_assert( eps, tol, test_name );
    n = 4;
    sprintf( test_name,
        "Cholesky factorization, outer product, %dx%d matrix", n, n );
    // Perform Cholesky factorization and compare result with correct answer
    chol_outer_product( n, B );
    error_matrix_comp_frob( &eps, &err, n, n, LB, B );
    test_assert( eps, tol, test_name );

    n = 8;
    sprintf( test_name,
        "Cholesky factorization, outer product, %dx%d matrix", n, n );
    // Perform Cholesky factorization and compare result with correct answer
    chol_outer_product( n, C );
    error_matrix_comp_frob( &eps, &err, n, n, LC, C );
    test_assert( eps, tol, test_name );
}

/*
 * Checks whether an implementation of the SAXPY operation (jki indexing) for
 * Cholesky factorization is performed correctly on an n-by-n symmetric positive
 * definite matrix A.  The result is verified against that produced by the
 * outer product method.
 */
void test_chol_saxpy( void )
{
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 1.0;     // Scaling factor for random matrix
    const int       n = 21;          // n-by-n matrix A
    char    test_name[80];
    double  eps, err;
    double  *A, *L;

    sprintf( test_name, "Cholesky factorization, SAXPY, %dx%d matrix", n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    L = (double *) malloc( n*n*sizeof(double) );
    create_random_spd( alpha, n, A );
    copy_matrix( n, n, A, L );

    // Perform Cholesky factorization, compare result with outer product solution
    chol_saxpy( n, A );
    chol_outer_product( n, L );
    error_matrix_comp_frob( &eps, &err, n, n, L, A );
    test_assert( eps, tol, test_name );
    free( A );
    free( L );
}

/*
 * Checks whether simple blocking for Cholesky factorization is performed
 * correctly on n-by-n symmetric positive definite matrices.  The results are
 * verified against those produced by the outer product method.
 */
void test_chol_block( void )
{
    const int        mat_size[] = { 11, 48, 77 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double     tol = 1e-12,     // Error tolerance
                     alpha = 1.0;     // Scaling factor for random matrix

    char    *test_name[SIZES];
    double  eps, err;
    double  *A, *L;

    // Define test names
    test_name[0] = "Cholesky factorization, simple blocking --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "Cholesky factorization, simple blocking --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "Cholesky factorization, simple blocking --\n"
        "matrix dimension not a multiple of block dimension";

    for ( int i = 0; i < SIZES; i++ ) {
        int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        L = (double *) malloc( n*n*sizeof(double) );
        create_random_spd( alpha, n, A );
        copy_matrix( n, n, A, L );
        // Perform Cholesky factorization, compare result with outer product solution
        chol_block( n, A );
        chol_outer_product( n, L );
        error_matrix_comp_frob( &eps, &err, n, n, L, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( L );
    }
#undef SIZES
}

/*
 * Checks whether an implementation of simple blocking using a rectangular
 * version of the SAXPY operation for Cholesky factorization is performed
 * correctly on n-by-n symmetric positive definite matrices.  The results are
 * verified against those produced by the outer product method.
 */
void test_chol_rect_block( void )
{
    const int        mat_size[] = { 12, 48, 82 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double     tol = 1e-12,     // Error tolerance
                     alpha = 1.0;     // Scaling factor for random matrix
    char    *test_name[SIZES];
    double  eps, err;
    double  *A, *L;

    // Define test names
    test_name[0] = "Cholesky factorization, simple blocking, rectangular --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "Cholesky factorization, simple blocking, rectangular --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "Cholesky factorization, simple blocking, rectangular --\n"
        "matrix dimension not a multiple of block dimension";

    for ( int i = 0; i < SIZES; i++ ) {
        int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        L = (double *) malloc( n*n*sizeof(double) );
        create_random_spd( alpha, n, A );
        copy_matrix( n, n, A, L );
        // Perform Cholesky factorization, compare result with outer product solution
        chol_rect_block( n, A );
        chol_outer_product( n, L );
        error_matrix_comp_frob( &eps, &err, n, n, L, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( L );
    }
#undef SIZES
}

/*
 * Checks whether contiguous blocking for Cholesky factorization is performed
 * correctly on n-by-n symmetric positive definite matrices.  The results are
 * verified against those produced by the outer product method.
 */
void test_chol_contig_block( void )
{
    const int       mat_size[] = { 12, 96, 123 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 1.0;     // Scaling factor for random matrix

    char    *test_name[SIZES];
    double  eps, err;
    double  *A, *L;

    // Define test names
    test_name[0] = "Cholesky factorization, contiguous blocking --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "Cholesky factorization, contiguous blocking --\n"
```

```
        "matrix dimension a multiple of block dimension";
    test_name[2] = "Cholesky factorization, contiguous blocking --\n"
        "matrix dimension not a multiple of block dimension";

    for ( int i = 0; i < SIZES; i++ ) {
        int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        L = (double *) malloc( n*n*sizeof(double) );
        create_random_spd( alpha, n, A );
        copy_matrix( n, n, A, L );
        // Perform Cholesky factorization, compare result with outer product solution
        chol_contig_block( n, A );
        chol_outer_product( n, L );
        error_matrix_comp_frob( &eps, &err, n, n, L, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( L );
    }
#undef SIZES
}

/*
 * Checks whether recursive contiguous blocking for Cholesky factorization is
 * performed correctly on n-by-n symmetric positive definite matrices.  The
 * results are verified against those produced by the outer product method.
 */
void test_chol_recur_block( void )
{
    const int       mat_size[] = { 18, 96, 107 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,    // Error tolerance
                    alpha = 1.0;    // Scaling factor for random matrix

    char    *test_name[SIZES];
    double  eps, err;
    double  *A, *L;

    // Define test names
    test_name[0] = "Cholesky factorization, recursive contiguous blocking --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "Cholesky factorization, recursive contiguous blocking --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "Cholesky factorization, recursive contiguous blocking --\n"
        "matrix dimension not a multiple of block dimension";

    for ( int i = 0; i < SIZES; i++ ) {
        int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        L = (double *) malloc( n*n*sizeof(double) );
        create_random_spd( alpha, n, A );
        copy_matrix( n, n, A, L );
        // Perform Cholesky factorization, compare result with outer product solution
        chol_recur_block( n, A );
        chol_outer_product( n, L );
        error_matrix_comp_frob( &eps, &err, n, n, L, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( L );
    }
#undef SIZES
}

/*
 * Checks whether an implementation of simple blocking using the BLAS library
 * for Cholesky factorization is performed correctly on n-by-n symmetric
 * positive definite matrices.  The results are verified against those produced
 * by the outer product method.
 */
void test_chol_block_blas( void )
{
    const int       mat_size[] = { 12, 64, 82 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,    // Error tolerance
                    alpha = 1.0;    // Scaling factor for random matrix

    char    *test_name[SIZES];
    double  eps, err;
    double  *A, *L;

    // Define test names
    test_name[0] = "Cholesky, simple blocking using the BLAS library --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "Cholesky, simple blocking using the BLAS library --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "Cholesky, simple blocking using the BLAS library --\n"
        "matrix dimension not a multiple of block dimension";

    for ( int i = 0; i < SIZES; i++ ) {
        int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        L = (double *) malloc( n*n*sizeof(double) );
        create_random_spd( alpha, n, A );
        copy_matrix( n, n, A, L );
        // Perform Cholesky factorization, compare result with outer product solution
        chol_block_blas( n, A );
        chol_outer_product( n, L );
        error_matrix_comp_frob( &eps, &err, n, n, L, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( L );
    }
#undef SIZES
}

/*
 * Checks whether an implementation of contiguous blocking using the BLAS and
 * LAPACK libraries for Cholesky factorization is performed correctly on n-by-n
 * symmetric positive definite matrices.  The results are verified against those
 * produced by the outer product algorithm.
 */
void test_chol_contig_block_blas( void )
{
    const int        mat_size[] = { 11, 64, 87 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double     tol = 1e-12,     // Error tolerance
                     alpha = 1.0;     // Scaling factor for random matrix

    char    *test_name[SIZES];
    double  eps, err;
    double  *A, *L;

    // Define test names
    test_name[0] =
        "Cholesky, contiguous blocking using BLAS and LAPACK libraries --\n"
        "matrix dimension less than block dimension";
    test_name[1] =
        "Cholesky, contiguous blocking using BLAS and LAPACK libraries --\n"
        "matrix dimension a multiple of block dimension";

    test_name[2] =
        "Cholesky, contiguous blocking using BLAS and LAPACK libraries --\n"
        "matrix dimension not a multiple of block dimension";

    for ( int i = 0; i < SIZES; i++ ) {
        int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        L = (double *) malloc( n*n*sizeof(double) );
        create_random_spd( alpha, n, A );
        copy_matrix( n, n, A, L );
        // Perform Cholesky factorization, compare result with outer product solution
        chol_contig_block_blas( n, A );
        chol_outer_product( n, L );
        error_matrix_comp_frob( &eps, &err, n, n, L, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( L );
    }
#undef SIZES
}
```

```
/*
 * Checks whether the wrapper function properly invokes LAPACK routine DPOTRF,
 * which computes the Cholesky factorization of a real symmetric positive
 * definite matrix.
 */
void test_chol_lapack( void )
{
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 1.0;     // Scaling factor for random matrix
    const int       n = 52;          // n-by-n matrix A

    char    test_name[80];
    double  eps, err;
    double  *A, *L;

    sprintf( test_name,
        "Cholesky factorization, LAPACK routine DPOTRF, %dx%d matrix", n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    L = (double *) malloc( n*n*sizeof(double) );
    create_random_spd(alpha, n, A);
    copy_matrix( n, n, A, L );
    // Perform Cholesky factorization, compare result with outer product solution
    chol_lapack( n, A );
    chol_outer_product( n, L );
    error_matrix_comp_frob( &eps, &err, n, n, L, A );
    test_assert( eps, tol, test_name );
    free( A );
    free( L );
}

/**********************************************************************/

/*
 * Checks whether the outer product method (kji indexing) for symmetric
 * indefinite factorization (LDL') is performed correctly on n-by-n matrices.
 * The factorization PAP = LDL', where A is an n-by-n symmetric matrix, L is
 * unit lower triangular, D is block diagonal with block order 1 or 2 and P is
 * the permutation matrix, is tested with Bunch-Kaufman (partial), bounded
 * Bunch-Kaufman (rook) and Bunch-Parlett (complete) pivoting.  Matrix LD stores
 * the correct unit lower triangular and diagonal factors.
 */
void test_ldlt_outer_product( void )
{
    const double    tol = 1e-12;        // Error tolerance

    char    test_name[80];
    int     n;
    int     *piv, *ord;
    double  eps, err;
    double  AA[16], BB[16], CC[16];
    double  A[] =   { 1, 5, 7, 8, 5, 4, 12, 3,
                      7, 12, 10, 9, 8, 3, 9, 6 },
            LKA[] = { 6, 0.5, 1.5, 4.0/3.0, 5, 2.5, 7.5, -34.0/65.0,
                      7, 12, -3.5, 4.0/13.0, 8, 3, 9, -1483.0/195.0 },
            LBA[] = { 6, 0.5, 1.5, 4.0/3.0, 5, 2.5, 7.5, -34.0/65.0,
                      7, 12, -3.5, 4.0/13.0, 8, 3, 9, -1483.0/195.0 },
            B[] =   { -4, 8, 2, -4, 8, 6, -12, 3,
                      2, -12, 4, 2, -4, 3, 2, 3 },
            LBB[] = { 6, -12, -7.0/15.0, -0.3, 8, 4, -0.9, -0.4,
                      2, -12, 23.0/15.0, -12.0/23.0, -4, 3, 2, 197.0/46.0 },
            LPB[] = { 6, -12, -0.3, -7.0/15.0, 8, 4, -0.4, -0.9,
                      2, -12, 4.7, -8.0/47.0, -4, 3, 2, 197.0/141.0 },
            C[] =   { 4, 6, 1, -4, 6, 8, -12, 8,
                      1, -12, 6, 10, -4, 8, 10, 4 },
            LKC[] = { 4, 1.5, -1, 0.25, 6, -1, 14, 11.0/14.0,
                      1, -12, 0, -89.0/98.0, -4, 8, 10, 1291.0/49.0 },
            LBC[] = { 4, 1.5, -1, 0.25, 6, -1, 14, 11.0/14.0,
                      1, -12, 0, -89.0/98.0, -4, 8, 10, 1291.0/49.0 },
            LPC[] = { 8, -1.5, 1, 0.75, 6, -12, 22, -45.0/109.0,
                      1, -12, -4, 25.0/109.0, -4, 8, 10, 1291.0/218.0 };

    n = 4;
    sprintf( test_name,
"LDL' factorization, Bunch-Kaufman pivoting, outer product, %dx%d matrix", n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Perform LDL' factorization with Bunch-Kaufman pivoting,
    // and compare result with correct answer
    copy_matrix( n, n, A, AA );
    ldlt_outer_product( 'K', n, piv, ord, AA );
    error_matrix_comp_frob( &eps, &err, n, n, LKA, AA );
    test_assert( eps, tol, test_name );
    free( piv );
    free( ord );

    sprintf( test_name,
"LDL' factorization, bounded Bunch-Kaufman pivoting, outer product, %dx%d matrix",
        n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Perform LDL' factorization with bounded Bunch-Kaufman pivoting,
    // and compare result with correct answer
    copy_matrix( n, n, A, AA );
    ldlt_outer_product( 'B', n, piv, ord, AA );
    error_matrix_comp_frob( &eps, &err, n, n, LBA, AA );
    test_assert( eps, tol, test_name );
    free( piv );
    free( ord );

    sprintf( test_name,
"LDL' factorization, bounded Bunch-Kaufman pivoting, outer product, %dx%d matrix",
        n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Perform LDL' factorization with bounded Bunch-Kaufman pivoting,
    // and compare result with correct answer
    copy_matrix( n, n, B, BB );
    ldlt_outer_product( 'B', n, piv, ord, BB );
    error_matrix_comp_frob( &eps, &err, n, n, LBB, BB );
    test_assert( eps, tol, test_name );
    free( piv );
    free( ord );

    n = 4;
    sprintf( test_name,
"LDL' factorization, Bunch-Parlett pivoting, outer product, %dx%d matrix", n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Perform LDL' factorization with bounded Bunch-Kaufman pivoting,
    // and compare result with correct answer
    copy_matrix( n, n, B, BB );
    ldlt_outer_product( 'P', n, piv, ord, BB );
    error_matrix_comp_frob( &eps, &err, n, n, LPB, BB );
    test_assert( eps, tol, test_name );
    free( piv );
    free( ord );

    sprintf( test_name,
"LDL' factorization, Bunch-Kaufman pivoting, outer product, %dx%d matrix", n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Perform LDL' factorization with bounded Bunch-Kaufman pivoting,
    // and compare result with correct answer
    copy_matrix( n, n, C, CC );
    ldlt_outer_product( 'K', n, piv, ord, CC );
    error_matrix_comp_frob( &eps, &err, n, n, LKC, CC );
    test_assert( eps, tol, test_name );
    free( piv );
    free( ord );

    sprintf( test_name,
"LDL' factorization, bounded Bunch-Kaufman pivoting, outer product, %dx%d matrix",
        n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Perform LDL' factorization with bounded Bunch-Kaufman pivoting,
    // and compare result with correct answer
    copy_matrix( n, n, C, CC );
    ldlt_outer_product( 'B', n, piv, ord, CC );
    error_matrix_comp_frob( &eps, &err, n, n, LBC, CC );
    test_assert( eps, tol, test_name );
    free( piv );
    free( ord );

    n = 4;
    sprintf( test_name,
"LDL' factorization, Bunch-Parlett pivoting, outer product, %dx%d matrix", n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Perform LDL' factorization with bounded Bunch-Kaufman pivoting,
    // and compare result with correct answer
    copy_matrix( n, n, C, CC );
    ldlt_outer_product( 'P', n, piv, ord, CC );
    error_matrix_comp_frob( &eps, &err, n, n, LPC, CC );
    test_assert( eps, tol, test_name );
    free( piv );
    free( ord );
}

/*
 * Checks whether an implementation of the SAXPY operation (jki indexing) for
 * symmetric indefinite factorization (LDL') is performed correctly on an n-by-n
 * matrices.  Symmetric indefinite factorization is tested with Bunch-Kaufman
 * (partial) and bounded Bunch-Kaufman (rook) pivoting.  The results are
 * verified against those produced by the outer product method.
 */
void test_ldlt_saxpy( void )
{
    const int       n = 14;         // n-by-n matrix A
    const double    tol = 1e-12,    // Error tolerance
                    alpha = 10.0;   // Scaling factor for random matrix

    char    test_name[80];
    int     *piv, *ord;
    double  eps, err;
    double  *A, *LD;

    // Bunch-Kaufman pivoting
    sprintf( test_name,
"LDL' factorization, Bunch-Kaufman pivoting, SAXPY, %dx%d matrix", n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    LD = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_symmetric( alpha, n, A );
    copy_matrix( n, n, A, LD );
    // Perform LDL' factorization with Bunch-Kaufman pivoting,
    // and compare result with outer product solution
    ldlt_saxpy( 'K', n, piv, ord, A );
    ldlt_outer_product( 'K', n, piv, ord, LD );
    error_matrix_comp_frob( &eps, &err, n, n, LD, A );
    test_assert( eps, tol, test_name );
    free( A );
    free( LD );
    free( piv );
    free( ord );

    // Bounded Bunch-Kaufman pivoting
    sprintf( test_name,
"LDL' factorization, bounded Bunch-Kaufman pivoting, SAXPY, %dx%d matrix", n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    LD = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_symmetric( alpha, n, A );
    copy_matrix( n, n, A, LD );
    // Perform LDL' factorization with bounded Bunch-Kaufman pivoting,
    // and compare result with outer product solution
    ldlt_saxpy( 'B', n, piv, ord, A );
    ldlt_outer_product( 'B', n, piv, ord, LD );
    error_matrix_comp_frob( &eps, &err, n, n, LD, A );
    test_assert( eps, tol, test_name );
    free( A );
    free( LD );
    free( piv );
    free( ord );
}

/*
 * Checks whether simple blocking for symmetric indefinite factorization is
 * performed correctly on n-by-n matrices.  Symmetric indefinite factorization
 * is tested with Bunch-Kaufman (partial), bounded Bunch-Kaufman (rook) and
 * Bunch-Parlett (complete) pivoting.  The results are verified against those
 * produced by the outer product method.
 */
void test_ldlt_block( void )
{
    const int       mat_size[] = { 14, 64, 114 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,    // Error tolerance
                    alpha = 10.0;   // Scaling factor for random matrix

    char    *test_name[SIZES];
    int     *piv, *ord;
    double  eps, err;
    double  *A, *LD;

    // Bunch-Kaufman pivoting
    test_name[0] = "LDL' factorization, Bunch-Kaufman pivoting, simple blocking --\n"
        "matrix dimension less than block dimension";
```

```
test_name[1] = "LDL' factorization, Bunch-Kaufman pivoting, simple blocking --\n"
    "matrix dimension a multiple of block dimension";
test_name[2] = "LDL' factorization, Bunch-Kaufman pivoting, simple blocking --\n"
    "matrix dimension not a multiple of block dimension";
for ( int i = 0; i < SIZES; i++ ) {
    const int n = mat_size[i];
    A = (double *) malloc( n*n*sizeof(double) );
    LD = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_symmetric( alpha, n, A );
    copy_matrix( n, n, A, LD );
    // Perform LDL' factorization and compare result with outer product solution
    ldlt_block( 'K', n, piv, ord, A );
    ldlt_outer_product( 'K', n, piv, ord, LD );
    error_matrix_comp_frob( &eps, &err, n, n, LD, A );
    test_assert( eps, tol, test_name[i] );
    free( A );
    free( LD );
    free( piv );
    free( ord );
}

// Bounded Bunch-Kaufman pivoting
test_name[0] =
    "LDL' factorization, bounded Bunch-Kaufman pivoting, simple blocking --\n"
    "matrix dimension less than block dimension";
test_name[1] =
    "LDL' factorization, bounded Bunch-Kaufman pivoting, simple blocking --\n"
    "matrix dimension a multiple of block dimension";
test_name[2] =
    "LDL' factorization, bounded Bunch-Kaufman pivoting, simple blocking --\n"
    "matrix dimension not a multiple of block dimension";
for ( int i = 0; i < SIZES; i++ ) {
    const int n = mat_size[i];
    A = (double *) malloc( n*n*sizeof(double) );
    LD = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_symmetric( alpha, n, A );
    copy_matrix( n, n, A, LD );
    // Perform LDL' factorization, compare result with outer product solution
    ldlt_block( 'B', n, piv, ord, A );
    ldlt_outer_product( 'B', n, piv, ord, LD );
    error_matrix_comp_frob( &eps, &err, n, n, LD, A );
    test_assert( eps, tol, test_name[i] );
    free( A );
    free( LD );
    free( piv );
    free( ord );
    }

    // Bunch-Parlett pivoting
    test_name[0] = "LDL' factorization, Bunch-Parlett pivoting, simple blocking --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "LDL' factorization, Bunch-Parlett pivoting, simple blocking --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "LDL' factorization, Bunch-Parlett pivoting, simple blocking --\n"
        "matrix dimension not a multiple of block dimension";
    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        LD = (double *) malloc( n*n*sizeof(double) );
        piv = (int *) malloc( n*sizeof(int) );
        ord = (int *) malloc( n*sizeof(int) );
        create_random_symmetric( alpha, n, A );
        copy_matrix( n, n, A, LD );
        // Perform LDL' factorization, compare result with outer product solution
        ldlt_block( 'P', n, piv, ord, A );
        ldlt_outer_product( 'P', n, piv, ord, LD );
        error_matrix_comp_frob( &eps, &err, n, n, LD, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( LD );
        free( piv );
        free( ord );
    }
#undef SIZES
}

/*
 * Checks whether an implementation of simple blocking using the BLAS library
 * for symmetric indefinite factorization is performed correctly on n-by-n
 * matrices.  Symmetric indefinite factorization is tested with Bunch-Kaufman
 * (partial) and bounded Bunch-Kaufman (rook) pivoting.  The results are
 * verified against those produced by the outer product method.
 */
void test_ldlt_block_blas( void )
{
    const int       mat_size[] = { 27, 96, 133 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 10.0;    // Scaling factor for random matrix

    char    *test_name[SIZES];
    int     *piv, *ord;
    double  eps, err;
    double  *A, *LD;

    // Bunch-Kaufman pivoting
```

```
test_name[0] = "LDL' factorization, Bunch-Kaufman pivoting, BLAS routines --\n"
    "matrix dimension less than block dimension";
test_name[1] = "LDL' factorization, Bunch-Kaufman pivoting, BLAS routines --\n"
    "matrix dimension a multiple of block dimension";
test_name[2] = "LDL' factorization, Bunch-Kaufman pivoting, BLAS routines --\n"
    "matrix dimension not a multiple of block dimension";
for ( int i = 0; i < SIZES; i++ ) {
    const int n = mat_size[i];
    A = (double *) malloc( n*n*sizeof(double) );
    LD = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_symmetric( alpha, n, A );
    copy_matrix( n, n, A, LD );
    // Perform LDL' factorization, compare result with outer product solution
    ldlt_block_blas( 'K', n, piv, ord, A );
    ldlt_outer_product( 'K', n, piv, ord, LD );
    error_matrix_comp_frob( &eps, &err, n, n, LD, A );
    test_assert( eps, tol, test_name[i] );
    free( A );
    free( LD );
    free( piv );
    free( ord );
}

// Bounded Bunch-Kaufman pivoting
test_name[0] =
    "LDL' factorization, bounded Bunch-Kaufman pivoting, BLAS routines --\n"
    "matrix dimension less than block dimension";
test_name[1] =
    "LDL' factorization, bounded Bunch-Kaufman pivoting, BLAS routines --\n"
    "matrix dimension a multiple of block dimension";
test_name[2] =
    "LDL' factorization, bounded Bunch-Kaufman pivoting, BLAS routines --\n"
    "matrix dimension not a multiple of block dimension";
for ( int i = 0; i < SIZES; i++ ) {
    const int n = mat_size[i];
    A = (double *) malloc( n*n*sizeof(double) );
    LD = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_symmetric( alpha, n, A );
    copy_matrix( n, n, A, LD );
    // Perform LDL' factorization, compare result with outer product solution
    ldlt_block_blas( 'B', n, piv, ord, A );
    ldlt_outer_product( 'B', n, piv, ord, LD );
    error_matrix_comp_frob( &eps, &err, n, n, LD, A );
    test_assert( eps, tol, test_name[i] );
    free( A );
    free( LD );
```

```
        free( piv );
        free( ord );
    }
#undef SIZES
}

/*
 * Checks whether the wrapper function properly invokes LAPACK routine DSYTRF,
 * which computes the factorization of a real symmetric indefinite matrix.
 */
void test_ldlt_lapack( void )
{
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 10.0;    // Scaling factor for random matrix
    const int       n = 18;          // n-by-n matrix A

    double  eps, err, normA, normLD;
    char    test_name[80];
    int     *piv, *ord;
    double  *A, *LD;

    sprintf( test_name,
"LDL' factorization, Bunch-Kaufman pivoting, LAPACK routine DSYTRF, %dx%d matrix",
        n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    LD = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_symmetric( alpha, n, A );
    copy_matrix( n, n, A, LD );
    // Perform LDL' factorization with LAPACK routine DSYTRF,
    // and compare result with outer product solution
    ldlt_lapack( 'K', n, piv, ord, A );
    ldlt_outer_product( 'K', n, piv, ord, LD );

    // The factorization produced by LAPACK routine DSYTRF takes the form
    // A = (P*L)*D*(P*L)', whereas the factorization produced by
    // ldlt_outer_product() takes the form P*A*P' = L*D*L'.  Therefore, verify
    // that DSYTRF is called correctly by comparing the norms of factors
    // (matrices) computed by DSYTRF and ldlt_outer_product().
    normA = 0.0;
    normLD = 0.0;
    for ( int j = 0; j < n; j++ ) {
        for ( int i = 0; i < n; i++ ) {
            double aij = fabs( *(A + i + j*n) );
            double lij = fabs( *(LD + i + j*n) );
            normA += aij;
            normLD += lij;
        }
    }
    err = abs( normA - normLD );
    eps = err / normA;
    test_assert( eps, tol, test_name );
    free( A );
    free( LD );
    free( piv );
    free( ord );
}

/*****************************************************************************/

/*
 * Checks whether the outer product method (kji indexing) for the modified
 * Cholesky algorithm proposed by Gill, Murray & Wright is performed correctly
 * on an n-by-n symmetric matrix.  Matrix A represents an n-by-n symmetric
 * linear system and matrix LD stores the correct unit lower triangular and
 * modified diagonal factors.
 */
void test_chol_gmw_outer_product( void )
{
    const double     tol = 1e-12;          // Error tolerance

    char    test_name[80];
    int     n;
    int     *piv, *ord;
    double  eps, err;
    double  A[] =   { -4, 3, 8, 2, 8, 5, 6, 1,
                      2, 6, -9, 12, 3, 1, 5, 8 },
            LDA[] = { 16, 0.5, 0.75, 0.375, 8, 8, -0.5, 0,
                      2, 6, 3, -7.0/6.0, 3, 1, 5, 4.0/3.0 };

    n = 4;
    sprintf( test_name,
"Modified Cholesky, Gill-Murray-Wright, outer product, %dx%d matrix", n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Perform modified Cholesky factorization, compare result with correct answer
    chol_gmw_outer_product( 'D', n, piv, ord, A );
    error_matrix_comp_frob( &eps, &err, n, n, LDA, A );
    test_assert( eps, tol, test_name );
    free( piv );
    free( ord );
}

/*
 * Checks whether an implementation of the SAXPY operation (jki indexing) for
 * the modified Cholesky algorithm proposed by Gill, Murray & Wright is
 * performed correctly on an n-by-n symmetric matrix.  The result is verified
 * against that produced by the outer product method.
 */
```

```
void test_chol_gmw_saxpy( void )
{
    const int       n = 22;          // n-by-n matrix A
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 10.0;    // Scaling factor for random matrix

    char    test_name[80];
    int     *piv, *ord;
    double  eps, err;
    double  *A, *LD;

    sprintf( test_name,
        "Modified Cholesky, Gill-Murray-Wright, SAXPY, %dx%d matrix", n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    LD = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_symmetric( alpha, n, A );
    copy_matrix( n, n, A, LD );
    // Perform modified Cholesky factorization,
    // and compare result with outer product solution
    chol_gmw_saxpy( 'D', n, piv, ord, A );
    chol_gmw_outer_product( 'D', n, piv, ord, LD );
    error_matrix_comp_frob( &eps, &err, n, n, LD, A );
    test_assert( eps, tol, test_name );
    free( A );
    free( LD );
    free( piv );
    free( ord );
}

/*
 * Checks whether simple blocking for the modified Cholesky algorithm proposed
 * by Gill, Murray & Wright is performed correctly on n-by-n symmetric matrices.
 * The results are verified against those produced by the outer product method.
 */
void test_chol_gmw_block( void )
{
    const int       mat_size[] = { 25, 96, 107 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 10.0;    // Scaling factor for random matrix

    char    *test_name[SIZES];
    int     *piv, *ord;
    double  eps, err;
    double  *A, *LD;

    test_name[0] = "Modified Cholesky, Gill-Murray-Wright, simple blocking --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "Modified Cholesky, Gill-Murray-Wright, simple blocking --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "Modified Cholesky, Gill-Murray-Wright, simple blocking --\n"
        "matrix dimension not a multiple of block dimension";
    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        LD = (double *) malloc( n*n*sizeof(double) );
        piv = (int *) malloc( n*sizeof(int) );
        ord = (int *) malloc( n*sizeof(int) );
        create_random_symmetric( alpha, n, A );
        copy_matrix( n, n, A, LD );
        // Perform modified Cholesky factorization,
        // and compare result with outer product solution
        chol_gmw_block( 'D', n, piv, ord, A );
        chol_gmw_outer_product( 'D', n, piv, ord, LD );
        error_matrix_comp_frob( &eps, &err, n, n, LD, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( LD );
        free( piv );
        free( ord );
    }
#undef SIZES
}

/*
 * Checks whether an implementation of simple blocking using the BLAS library
 * for the modified Cholesky algorithm proposed by Gill, Murray & Wright is
 * performed correctly on n-by-n symmetric matrices.  The results are verified
 * against those produced by the outer product method.
 */
void test_chol_gmw_block_blas( void )
{
    const int       mat_size[] = { 25, 96, 107 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 10.0;    // Scaling factor for random matrix

    char    *test_name[SIZES];
    int     *piv, *ord;
    double  eps, err;
    double  *A, *LD;

    test_name[0] = "Modified Cholesky, Gill-Murray-Wright, BLAS routines --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "Modified Cholesky, Gill-Murray-Wright, BLAS routines --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "Modified Cholesky, Gill-Murray-Wright, BLAS routines --\n"
        "matrix dimension not a multiple of block dimension";
    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        LD = (double *) malloc( n*n*sizeof(double) );
        piv = (int *) malloc( n*sizeof(int) );
        ord = (int *) malloc( n*sizeof(int) );
        create_random_symmetric( alpha, n, A );
        copy_matrix( n, n, A, LD );
        // Perform modified Cholesky factorization,
        // and compare result with outer product solution
        chol_gmw_block_blas( 'D', n, piv, ord, A );
        chol_gmw_outer_product( 'D', n, piv, ord, LD );
        error_matrix_comp_frob( &eps, &err, n, n, LD, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( LD );
        free( piv );
        free( ord );
    }
#undef SIZES
}

/*
 * Checks whether the outer product method (kji indexing) for the modified
 * Cholesky algorithm proposed by Cheng & Higham is performed correctly on an
 * n-by-n symmetric matrix.  Matrix A represents an n-by-n symmetric linear
 * system and matrix LD stores the correct unit lower triangular and modified
 * diagonal factors.
 */
void test_chol_ch_outer_product( void )
{
    const double    tol = 1e-06;        // Error tolerance

    char     test_name[80];
    int      n;
    int      *piv, *ord;
    double   eps, err;
    double   a, b, c, delta;
    double   A[] =   { 1, 5, 7, 8, 5, 4, 12, 3,
                       7, 12, 10, 9, 8, 3, 9, 6 },
             LDA[] = { 6, 0.5, 1.5, 4.0/3.0, 5, 2.5, 7.5, -34.0/65.0,
                       7, 12, -3.5, 4.0/13.0, 8, 3, 9, -1483.0/195.0 };

    n = 4;
    sprintf( test_name,
        "Modified Cholesky, Cheng-Higham, outer product, %dx%d matrix", n, n );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    // Matrix LDA has been initialized with results from symmetric indefinite
    // factorization, P*A*P = L*D*L'.  Update diagonal block D so that (A + dA)
    // is positive definite i.e., P*(A+dA)*P = L*D*L'.
    delta = 38.0 * sqrt(0.5*DBL_EPSILON);
    c = sqrt(29.0);
    a = 58.0 + 4.0 * c;
    b = 58.0 - 4.0 * c;
    LDA[5] = 12.5 * (3.0*c - 1.0) / b + 25.0 * delta / a;
    LDA[6] = 2.5 * (89.0 - 7.0*c) / b + 5.0 * delta * (2.0 + c) / a;
    LDA[10] = 0.5 * (103.0*c - 381.0) / b + delta * (33.0 + 4.0*c) / a;
    LDA[15] = delta;
    // Perform modified Cholesky factorization, compare result with correct answer
    chol_ch_outer_product( 'K', n, piv, ord, A );
    error_matrix_comp_frob( &eps, &err, n, n, LDA, A );
    test_assert( eps, tol, test_name );
    free(piv);
    free(ord);
}

/*
 * Checks whether an implementation of the SAXPY operation (jki indexing) for
 * the modified Cholesky algorithm proposed by Cheng & Higham is performed
 * correctly on an n-by-n symmetric matrix.  The result is verified against that
 * produced by the outer product method.
 */
void test_chol_ch_saxpy( void )
{
    const int       n = 20;         // n-by-n matrix A
    const double    tol = 1e-12,    // Error tolerance
                    alpha = 10.0;   // Scaling factor for random matrix

    int     *piv, *ord;
    char    test_name[80];
    double  eps, err;
    double  *A, *LD;

    sprintf(test_name,
        "Modified Cholesky, Cheng-Higham, SAXPY, %dx%d matrix", n, n);
    A = (double *) malloc( n*n*sizeof(double) );
    LD = (double *) malloc( n*n*sizeof(double) );
    piv = (int *) malloc( n*sizeof(int) );
    ord = (int *) malloc( n*sizeof(int) );
    create_random_symmetric( alpha, n, A );
    copy_matrix( n, n, A, LD );
    // Perform modified Cholesky factorization,
    // and compare result with outer product solution
    chol_ch_saxpy( 'K', n, piv, ord, A );
    chol_ch_outer_product( 'K', n, piv, ord, LD );
    error_matrix_comp_frob( &eps, &err, n, n, LD, A );
    test_assert( eps, tol, test_name );
    free( A );
    free( LD );
```

```c
    free( piv );
    free( ord );
}

/*
 * Checks whether simple blocking for the modified Cholesky algorithm proposed
 * by Cheng & Higham is performed correctly on n-by-n symmetric matrices.  The
 * results are verified against those produced by the outer product method.
 */
void test_chol_ch_block( void )
{
    const int       mat_size[] = { 14, 64, 87 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 10.0;    // Scaling factor for random matrix

    char    *test_name[SIZES];
    int     *piv, *ord;
    double  eps, err;
    double  *A, *LD;

    // Bunch Kaufman pivoting
    test_name[0] = "Modified Cholesky, Cheng-Higham, simple blocking --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "Modified Cholesky, Cheng-Higham, simple blocking --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "Modified Cholesky, Cheng-Higham, simple blocking --\n"
        "matrix dimension not a multiple of block dimension";
    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        LD = (double *) malloc( n*n*sizeof(double) );
        piv = (int *) malloc( n*sizeof(int) );
        ord = (int *) malloc( n*sizeof(int) );
        create_random_symmetric( alpha, n, A );
        copy_matrix( n, n, A, LD );
        // Perform modified Cholesky factorization,
        // and compare result with outer product solution
        chol_ch_block( 'K', n, piv, ord, A );
        chol_ch_outer_product( 'K', n, piv, ord, LD );
        error_matrix_comp_frob( &eps, &err, n, n, LD, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( LD );
        free( piv );
        free( ord );
    }
#undef SIZES
}
```

```
/*
 * Checks whether an implementation of simple blocking using the BLAS library
 * for the modified Cholesky algorithm proposed by Cheng & Higham is performed
 * correctly on n-by-n symmetric matrices.  The results are verified against
 * those produced by the outer product method.
 */
void test_chol_ch_block_blas( void )
{
    const int        mat_size[] = { 14, 64, 87 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double     tol = 1e-12,     // Error tolerance
                     alpha = 10.0;    // Scaling factor for random matrix

    char     *test_name[SIZES];
    int      *piv, *ord;
    double   eps, err;
    double   *A, *LD;

    // Bunch Kaufman pivoting
    test_name[0] = "Modified Cholesky, Cheng-Higham, BLAS routines --\n"
        "matrix dimension less than block dimension";
    test_name[1] = "Modified Cholesky, Cheng-Higham, BLAS routines --\n"
        "matrix dimension a multiple of block dimension";
    test_name[2] = "Modified Cholesky, Cheng-Higham, BLAS routines --\n"
        "matrix dimension not a multiple of block dimension";
    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        LD = (double *) malloc( n*n*sizeof(double) );
        piv = (int *) malloc( n*sizeof(int) );
        ord = (int *) malloc( n*sizeof(int) );
        create_random_symmetric( alpha, n, A );
        copy_matrix( n, n, A, LD );
        // Perform modified Cholesky factorization,
        // and compare result with outer product solution
        chol_ch_block_blas( 'K', n, piv, ord, A );
        chol_ch_outer_product( 'K', n, piv, ord, LD );
        error_matrix_comp_frob( &eps, &err, n, n, LD, A );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( LD );
        free( piv );
        free( ord );
    }
#undef SIZES
}
```

A.13. mmultest.c – testing harness for matrix multiplication.

```
/*
 * Testing harness for unblocked and blocked algorithms implementing matrix
 * multiplication (and addition), C = C + A*B, on square matrices.  The number
 * of tests and error count are accumulated through a single execution of the
 * mmultest program, and all test results are written to an output file
 * destination (terminal).
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#include "matmult.h"
#include "matcom.h"

static void test_assert( double eps, double tol, const char *test_name );
static void test_mmult_dot_product( void );
static void test_mmult_saxpy( void );
static void test_mmult_unroll( void );
static void test_mmult_pipeline( void );
static void test_mmult_block( void );
static void test_mmult_contig_block( void );
static void test_mmult_recur_block( void );
static void test_mmult_rect_recur_block( void );
static void test_mmult_blas( void );

static int      tests = 0,          // Test count
                errs = 0;           // Error count
static FILE     *fp;

int main()
{
    fp = stdout;

    test_mmult_dot_product();
    test_mmult_saxpy();
    test_mmult_unroll();
    test_mmult_pipeline();
    test_mmult_block();
    test_mmult_contig_block();
    test_mmult_recur_block();
    test_mmult_rect_recur_block();
    test_mmult_blas();

    if ( errs == 0 ) {
        fprintf( fp, "Passed all %d tests.\n", tests );
    } else {
        fprintf( fp, "Total of %d error(s) encountered in %d tests.\n",
            errs, tests );
    }
    return 0;
}

/*
 * Verifies that test results are accurate within specified tolerance, and
 * prints message indicating whether the routine passed or failed the test.
 */
void test_assert( double eps, double tol, const char *test_name )
{
    tests++;
    if ( eps <= tol ) {
        fprintf( fp, "PASSED: %s\n(eps=%e <= tol=%e)\n", test_name, eps, tol );
    } else {
        fprintf( fp, "FAILED: %s\n(eps=%e > tol=%e)\n", test_name, eps, tol );
        errs++;
    }
}

/**************************************************************************/

/*
 * Checks whether the dot (inner) product method with ijk indexing performs
 * matrix multiplication (and addition), C = C + A*B, correctly.  Verification
 * is done on pre-specified n-by-n matrices with leading dimension n stored in
 * column-major order.
 */
void test_mmult_dot_product( void )
{
    const int       n = 4;          // n-by-n matrix
    const double    tol = 1e-12;    // Error tolerance

    char    test_name[80];
    double  eps, err;
    double  A[] = { 4.2, 9.2, 7.9, 9.6, 6.6, 0.4, 8.5, 9.3,
                    6.8, 7.6, 7.4, 3.9, 6.6, 1.7, 7.1, 0.3 },
            B[] = { 2.8, 0.5, 1.0, 8.2, 6.9, 3.2, 9.5, 0.3,
                    4.4, 3.8, 7.7, 8.0, 1.9, 4.9, 4.5, 6.5 },
            C[] = { 8.9, 9.6, 5.5, 1.4, 1.5, 2.6, 8.4, 2.5,
                    8.1, 2.4, 9.3, 3.5, 2.0, 2.5, 6.2, 4.7 },
            C_ans[] = {  84.88,  57.10,  97.49,  39.29,
                        118.18, 140.07, 162.54, 135.64,
                        156.82, 116.52, 190.14, 113.51,
                        115.82,  67.19, 142.31,  88.01 };

    sprintf( test_name,
        "Matrix multiplication, dot product (ijk indexing), %dx%d matrix", n, n );
    // Perform matrix multiplication and compare result with correct answer
    mmult_dot_product( n, A, B, C );
    error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
    test_assert( eps, tol, test_name );
}

/*
 * Verifies that the SAXPY operation with jki indexing performs matrix
 * multiplication (and addition), C = C + A*B, correctly.  The result from the
 * SAXPY operation is compared with that produced by the dot product method.
 */
void test_mmult_saxpy( void )
{
    const int       n = 72;          // n-by-n matrix
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 10.0;    // Scaling factor for random matrix

    char    test_name[80];
    double  eps, err;
    double  *A, *B, *C, *C_ans;

    sprintf( test_name,
        "Matrix multiplication, SAXPY (kji indexing), %dx%d matrix", n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    B = (double *) malloc( n*n*sizeof(double) );
    C = (double *) malloc( n*n*sizeof(double) );
    C_ans = (double *) malloc( n*n*sizeof(double) );
    create_random_matrix( alpha, n, n, A );
    create_random_matrix( alpha, n, n, B );
    create_random_matrix( alpha, n, n, C );
    copy_matrix( n, n, C, C_ans );

    // Compute C_ans = C + A*B using the dot product method
    mmult_dot_product( n, A, B, C_ans );
    // Compute C = C + A*B using the SAXPY operation, compare with C_ans
    mmult_saxpy( n, A, B, C );
    error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
    test_assert( eps, tol, test_name );
    free( A );
    free( B );
    free( C );
    free( C_ans );
}

/*
 * Verifies that the dot product method with loop unrolling performs matrix
 * multiplication (and addition), C = C + A*B, correctly.  The results from the
 * optimized algorithm with loop unrolling are compared with those produced by
 * the basic dot product method.
 */
void test_mmult_unroll( void )
{
    const int        mat_size[] = { 6, 48, 66 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double     tol = 1e-12,     // Error tolerance
                     alpha = 10.0;    // Scaling factor for random matrix

    char    *ptr, *test_name[SIZES];
    double  eps, err;
    double  *A, *B, *C, *C_ans;

    // Define test names
    ptr =   "Matrix multiplication, dot product with loop unrolling --\n"
            "matrix dimension less than UNROLL_DEPTH";
    test_name[0] = ptr;
    ptr =   "Matrix multiplication, dot product with loop unrolling --\n"
            "matrix dimension a multiple of UNROLL_DEPTH";
    test_name[1] = ptr;
    ptr =   "Matrix multiplication, dot product with loop unrolling --\n"
            "matrix dimension not a multiple of UNROLL_DEPTH";
    test_name[2] = ptr;

    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        B = (double *) malloc( n*n*sizeof(double) );
        C = (double *) malloc( n*n*sizeof(double) );
        C_ans = (double *) malloc( n*n*sizeof(double) );
        create_random_matrix( alpha, n, n, A );
        create_random_matrix( alpha, n, n, B );
        create_random_matrix( alpha, n, n, C );
        copy_matrix( n, n, C, C_ans );

        // Compute C_ans = C + A*B using the dot product method
        mmult_dot_product( n, A, B, C_ans );
        // Compute C = C + A*B using loop unrolling, compare with C_ans
        mmult_unroll( n, A, B, C );
        error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( B );
        free( C );
        free( C_ans );
    }
#undef SIZES
}

/*
 * Verifies that the SAXPY method with software pipelining performs matrix
 * multiplication (and addition), C = C + A*B, correctly.  The results from the
 * optimized algorithm with software pipelining are compared with those
 * produced by the basic dot product method.
 */
void test_mmult_pipeline( void )
{
    const int        mat_size[] = { 3, 64, 77 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double     tol = 1e-12,     // Error tolerance
                     alpha = 10.0;    // Scaling factor for random matrix

    char    *ptr, *test_name[SIZES];
    double  eps, err;
    double  *A, *B, *C, *C_ans;


    // Define test names
    ptr =   "Matrix multiplication, SAXPY with software pipelining --\n"
            "matrix dimension less than PIPE_DEPTH";
    test_name[0] = ptr;
    ptr =   "Matrix multiplication, SAXPY with software pipelining --\n"
            "matrix dimension a multiple of PIPE_DEPTH";
    test_name[1] = ptr;
    ptr =   "Matrix multiplication, SAXPY with software pipelining --\n"
            "matrix dimension not a multiple of PIPE_DEPTH";
    test_name[2] = ptr;

    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        B = (double *) malloc( n*n*sizeof(double) );
        C = (double *) malloc( n*n*sizeof(double) );
        C_ans = (double *) malloc( n*n*sizeof(double) );
        create_random_matrix( alpha, n, n, A );
        create_random_matrix( alpha, n, n, B );
        create_random_matrix( alpha, n, n, C );
        copy_matrix( n, n, C, C_ans );

        // Compute C_ans = C + A*B using the dot product method
        mmult_dot_product( n, A, B, C_ans );
        // Compute C = C + A*B using software pipelining, compare with C_ans
        mmult_pipeline( n, A, B, C );
        error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( B );
        free( C );
        free( C_ans );
    }
#undef SIZES
}

/*
```

```
 * Verifies that the simple blocking algorithm performs matrix multiplication
 * (and addition), C = C + A*B, correctly.  The results from the blocked
 * algorithm are compared with those produced by the dot product method.
 */
void test_mmult_block( void )
{
    const int        mat_size[] = { 21, 96, 111 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double     tol = 1e-12,     // Error tolerance
                     alpha = 10.0;    // Scaling factor for random matrix

    char     *ptr, *test_name[SIZES];
    double   eps, err;
    double   *A, *B, *C, *C_ans;

    // Define test names
    ptr =   "Matrix multiplication, simple blocking --\n"
            "matrix dimension less than block dimension";
    test_name[0] = ptr;
    ptr =   "Matrix multiplication, simple blocking --\n"
            "matrix dimension a multiple of block dimension";
    test_name[1] = ptr;
    ptr =   "Matrix multiplication, simple blocking --\n"
            "matrix dimension not a multiple of block dimension";
    test_name[2] = ptr;

    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        B = (double *) malloc( n*n*sizeof(double) );
        C = (double *) malloc( n*n*sizeof(double) );
        C_ans = (double *) malloc( n*n*sizeof(double) );
        create_random_matrix( alpha, n, n, A );
        create_random_matrix( alpha, n, n, B );
        create_random_matrix( alpha, n, n, C );
        copy_matrix( n, n, C, C_ans );

        // Compute C_ans = C + A*B using the dot product method
        mmult_dot_product( n, A, B, C_ans );
        // Compute C = C + A*B using the blocked algorithm, compare with C_ans
        mmult_block( n, A, B, C );
        error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( B );
        free( C );
        free( C_ans );
    }
#undef SIZES
}
```

```
/*
 * Verifies that the contiguous blocking algorithm performs matrix
 * multiplication (and addition), C = C + A*B, correctly.  The results from the
 * blocked algorithm are compared with those produced by the dot product method.
 */
void test_mmult_contig_block( void )
{
    const int        mat_size[] = { 13, 96, 122 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double     tol = 1e-12,     // Error tolerance
                     alpha = 10.0;    // Scaling factor for random matrix

    char     *ptr, *test_name[SIZES];
    double   eps, err;
    double   *A, *B, *C, *C_ans;

    // Define test names
    ptr =   "Matrix multiplication, contiguous block storage --\n"
            "matrix dimension less than block dimension";
    test_name[0] = ptr;
    ptr =   "Matrix multiplication, contiguous block storage --\n"
            "matrix dimension a multiple of block dimension";
    test_name[1] = ptr;
    ptr =   "Matrix multiplication, contiguous block storage --\n"
            "matrix dimension not a multiple of block dimension";
    test_name[2] = ptr;

    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        B = (double *) malloc( n*n*sizeof(double) );
        C = (double *) malloc( n*n*sizeof(double) );
        C_ans = (double *) malloc( n*n*sizeof(double) );
        create_random_matrix( alpha, n, n, A );
        create_random_matrix( alpha, n, n, B );
        create_random_matrix( alpha, n, n, C );
        copy_matrix( n, n, C, C_ans );

        // Compute C_ans = C + A*B using the dot product method
        mmult_dot_product( n, A, B, C_ans );
        // Compute C = C + A*B using the blocked algorithm, compare with C_ans
        mmult_contig_block( n, A, B, C );
        error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( B );
        free( C );
        free( C_ans );
    }
```

```
#undef SIZES
}

/*
 * Verifies that the recursive contiguous blocking algorithm performs matrix
 * multiplication (and addition), C = C + A*B, correctly.  The matrix
 * multiplication kernel uses a symbolic constant to control looping.  The
 * results from the blocked algorithm are compared with those produced by the
 * dot product method.
 */
void test_mmult_recur_block( void )
{
    const int       mat_size[] = { 13, 96, 122 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double    tol = 1e-12,     // Error tolerance
                    alpha = 10.0;    // Scaling factor for random matrix

    char    *ptr, *test_name[SIZES];
    double  eps, err;
    double  *A, *B, *C, *C_ans;

    // Define test names
    ptr =   "Matrix multiplication, recursive contiguous blocking --\n"
            "matrix dimension less than block dimension";
    test_name[0] = ptr;
    ptr =   "Matrix multiplication, recursive contiguous blocking --\n"
            "matrix dimension a multiple of block dimension";
    test_name[1] = ptr;
    ptr =   "Matrix multiplication, recursive contiguous blocking --\n"
            "matrix dimension not a multiple of block dimension";
    test_name[2] = ptr;


    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        B = (double *) malloc( n*n*sizeof(double) );
        C = (double *) malloc( n*n*sizeof(double) );
        C_ans = (double *) malloc( n*n*sizeof(double) );
        create_random_matrix( alpha, n, n, A );
        create_random_matrix( alpha, n, n, B );
        create_random_matrix( alpha, n, n, C );
        copy_matrix( n, n, C, C_ans );

        // Compute C_ans = C + A*B using the basic dot product algorithm
        mmult_dot_product( n, A, B, C_ans );
        // Compute C = C + A*B using the optimized algorithm, compare with C_ans
        mmult_recur_block( n, A, B, C );
        error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( B );
        free( C );
        free( C_ans );
    }
#undef SIZES
}

/*
 * Verifies that the recursive contiguous blocking algorithm performs matrix
 * multiplication (and addition), C = C + A*B, correctly.  The matrix
 * multiplication kernel uses variables to control looping.  The results from
 * the blocked algorithm are compared with those produced by the dot product
 * method.
 */
void test_mmult_rect_recur_block( void )
{
    const int        mat_size[] = { 25, 128, 141 };
#define SIZES (sizeof(mat_size) / sizeof(int))
    const double     tol = 1e-12,     // Error tolerance
                     alpha = 10.0;    // Scaling factor for random matrix

    char    *ptr, *test_name[SIZES];
    double  eps, err;
    double  *A, *B, *C, *C_ans;

    // Define test names
    ptr =   "Multiplication, recursive contiguous blocking, variable looping --\n"
            "matrix dimension less than block dimension";
    test_name[0] = ptr;
    ptr =   "Multiplication, recursive contiguous blocking, variable looping --\n"
            "matrix dimension a multiple of block dimension";
    test_name[1] = ptr;
    ptr =   "Multiplication, recursive contiguous blocking, variable looping --\n"
            "matrix dimension not a multiple of block dimension";
    test_name[2] = ptr;


    for ( int i = 0; i < SIZES; i++ ) {
        const int n = mat_size[i];
        A = (double *) malloc( n*n*sizeof(double) );
        B = (double *) malloc( n*n*sizeof(double) );
        C = (double *) malloc( n*n*sizeof(double) );
        C_ans = (double *) malloc( n*n*sizeof(double) );
        create_random_matrix( alpha, n, n, A );
        create_random_matrix( alpha, n, n, B );
        create_random_matrix( alpha, n, n, C );
        copy_matrix( n, n, C, C_ans );

        // Compute C_ans = C + A*B using the dot product method
        mmult_dot_product( n, A, B, C_ans );
        // Compute C = C + A*B using the blocked algorithm, compare with C_ans
        mmult_rect_recur_block( n, A, B, C );
        error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
        test_assert( eps, tol, test_name[i] );
        free( A );
        free( B );
        free( C );
        free( C_ans );
    }
#undef SIZES
}

/*
 * Checks whether the wrapper function properly invokes BLAS routine DGEMM,
 * which performs matrix multiplication.
 */
void test_mmult_blas( void )
{
    const int       n = 48;         // n-by-n matrix
    const double    tol = 1e-12,    // Error tolerance
                    alpha = 10.0;   // Scaling factor for random matrix

    char    test_name[80];
    double  eps, err;
    double  *A, *B, *C, *C_ans;

    sprintf( test_name,
        "Matrix multiplication, BLAS routine DGEMM, %dx%d matrix", n, n );
    A = (double *) malloc( n*n*sizeof(double) );
    B = (double *) malloc( n*n*sizeof(double) );
    C = (double *) malloc( n*n*sizeof(double) );
    C_ans = (double *) malloc( n*n*sizeof(double) );
    create_random_matrix( alpha, n, n, A );
    create_random_matrix( alpha, n, n, B );
    create_random_matrix( alpha, n, n, C );
    copy_matrix( n, n, C, C_ans );

    // Compute C_ans = C + A*B using the dot product method
    mmult_dot_product( n, A, B, C_ans );
    // Compute C = C + A*B using DGEMM, compare with C_ans
    mmult_blas( n, A, B, C );
    error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
    test_assert( eps, tol, test_name );
    free( A );
    free( B );
    free( C );
    free( C_ans );
}
```

A.14. mmultstp.c – testing harness for parallel matrix multiplication.

```c
/*
 * Testing harness for parallel algorithms implementing matrix multiplication
 * (and addition), C = C + A*B.  The number of tests and error count are
 * accumulated through a single execution of the mmultstp program, and all test
 * results are written to an output file destination (terminal).
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <mpi.h>

#include "matmultp.h"

static void test_assert( double eps, double tol, const char *test_name );
static void error_matrix_comp_frob( double *eps, double *err, int m, int n,
    const double *E, const double *F );
static void error_matrix_comp_l1( double *eps, double *err, int m, int n,
    const double *E, const double *F );
static void init_test_matrices_6(
    double *A, double *B, double *C, double *C_ans );
static void init_test_matrices( int n,
    double *A, double *B, double *C, double *C_ans );
static void test_serial_matrix_multiply( int n,
    const double *A, const double *B, double *C, const double *C_ans,
    double tol, const char *test_name );
static void test_parallel_matrix_multiply( int n,
    const double *A, const double *B, double *C, const double *C_ans,
    struct mpi_grid *grid, double tol, const char *test_name );

static int              tests = 0,          // Test count
                        errs = 0;           // Error count
static FILE             *fp;
static struct mpi_grid  grid;

int main( int argc, char **argv )
{
    const double    tol = 1e-12;    // Error tolerance

    char    test_name[80];
    int     n;
    double  *A, *B, *C, *C_ans;

    MPI_Init( &argc, &argv );
    //Establish Cartesian topology for collective communication
    setup_mpi_grid( &grid );

    fp = stdout;
```

```
n = 6;
if ( grid.rank == 0 ) {
    // Allocate memory for matrices
    A = (double *) malloc( n*n*sizeof(double) );
    B = (double *) malloc( n*n*sizeof(double) );
    C = (double *) malloc( n*n*sizeof(double) );
    C_ans = (double *) malloc( n*n*sizeof(double) );
    // Test serial matrix multiply algorithm on 6x6 test matrices
    sprintf( test_name,
        "Serial matrix multiply algorithm, %dx%d matrix", n, n );
    init_test_matrices_6( A, B, C, C_ans );
    test_serial_matrix_multiply( n, A, B, C, C_ans, tol, test_name );

    // Re-initialize matrices for parallel matrix multiply test
    sprintf( test_name,
        "Fox algorithm on %d parallel processors, %dx%d matrix",
        grid.p, n, n );
    init_test_matrices_6( A, B, C, C_ans );
}
// Matrices are partitioned into p full blocks for parallel processing on
// p processors
test_parallel_matrix_multiply( n, A, B, C, C_ans, &grid, tol, test_name );
if ( grid.rank == 0 ) {
    free( A );
    free( B );
    free( C );
    free( C_ans );
}

// Matrices are partitioned into (q-1)^2 full blocks and (2q-1) fringe
// blocks for parallel processing on p processors, p = q*q
n = 2000;
if ( grid.rank == 0 ) {
    // Allocate memory for matrices
    A = (double *) malloc( n*n*sizeof(double) );
    B = (double *) malloc( n*n*sizeof(double) );
    C = (double *) malloc( n*n*sizeof(double) );
    C_ans = (double *) malloc( n*n*sizeof(double) );
    init_test_matrices( n, A, B, C, C_ans );
    sprintf( test_name,
        "Fox algorithm on %d parallel processors, %dx%d matrix",
        grid.p, n, n );
}
test_parallel_matrix_multiply(n, A, B, C, C_ans, &grid, tol, test_name);
if ( grid.rank == 0 ) {
    free( A );
    free( B );
    free( C );
    free( C_ans );
    }

    if ( grid.rank == 0 ) {
        if ( errs == 0 ) {
            fprintf( fp, "\nPassed all %d tests.\n", tests );
        } else {
            fprintf( fp, "\nTotal of %d error(s) encountered in %d tests.\n",
                errs, tests );
        }
    }
    MPI_Finalize();
    return 0;
}

/***************************************************************************/

/*
 * Verifies that test results are accurate within specified tolerance, and
 * prints message indicating whether the routine passed or failed the test.
 */
void test_assert( double eps, double tol, const char *test_name )
{
    tests++;
    if ( eps <= tol ) {
        fprintf( fp, "PASSED: %s\n(eps=%e <= tol=%e)\n", test_name, eps, tol );
    } else {
        fprintf( fp, "FAILED: %s\n(eps=%e > tol=%e)\n", test_name, eps, tol );
        errs++;
    }
}

/*
 * Computes the relative and absolute errors in a matrix computation using the
 * Frobenius norm ||F - E||, where F is the result of the floating point matrix
 * computation and E is the exact solution.  Both matrices are stored in
 * column-major order with leading dimension m.
 */
void error_matrix_comp_frob( double *eps, double *err, int m, int n,
    const double *E, const double *F )
{
    int     ldim = m;
    double  ssq_delta = 0.0;
    double  ssq_eij = 0.0;

    for ( int j = 0; j < n; j++ ) {
        const double *E_j = E + j*ldim;
        const double *F_j = F + j*ldim;
        for ( int i = 0; i < m; i++ ) {
            double delta = *(E_j + i) - *(F_j + i);
            ssq_delta +=  delta * delta;
            ssq_eij += *(E_j + i) * *(E_j + i);
        }
    }
    *err = sqrt( ssq_delta );
    *eps = *err / sqrt( ssq_eij );
}

/*
 * Computes the relative and absolute errors in a matrix computation using the
 * l1-norm ||F - E||, where F is the result of the floating point matrix
 * computation and E is the exact solution.  Both matrices are stored in
 * column-major order with leading dimension m.
 */
void error_matrix_comp_l1( double *eps, double *err, int m, int n,
    const double *E, const double *F )
{
    int     ldim = m;
    double  sum_abs_delta = 0.0;
    double  sum_abs_eij = 0.0;

    *err = 0.0;
    *eps = 0.0;

    for ( int j = 0; j < n; j++ ) {
        const double *E_j = E + j*ldim;
        const double *F_j = F + j*ldim;
        for ( int i = 0; i < m; i++ ) {
            double delta = *(E_j + i) - *(F_j + i);
            sum_abs_delta +=  fabs( delta );
            sum_abs_eij += fabs(*(E_j + i));
        }
        if ( sum_abs_delta > *err ) {
            *err = sum_abs_delta;
            *eps = *err / sum_abs_eij;
        }
    }
}

/*
 * Initializes 6x6 matrices used to test serial and parallel matrix
 * multiplication algorithms.  Matrices A, B and C are initialized with preset
 * values, and C_ans contains the result of the matrix multiplication (and
 * addition), C_ans = C + A*B.  Matrices are stored in column-major order.
 */
void init_test_matrices_6( double *A, double *B, double *C, double *C_ans )
{
    const int       n = 6;          // n-by-n matrices

    double  AA[] =      { -2,  1, -8, -8,  1,  5,  8, -7,  1, -1, -9, -3,
                          -6,  6, -4,  1, -6,  2, -5,  3,  4,  5, -1, -8,
                     -5,  8, -7,  6,  1,  9, -8, -1, -7,  9, -9,  5 },
        BB[] =     {  6,  7, -8, -2, -5,  6, -1,  8, -6, -4, -7, -7,
                      7,  2,  1, -7,  7,  2, -3,  0, -2, -8, -5, -7,
                     -6, -5, -2, -9,  8,  8,  0,  0, -3,  8, -2, -7 },
        CC[] =     {  5, -2, -5, -2, -8, -7,  8,  9,  1, -8, -5, -3,
                      6, -9, -9, -6,  3,  4,  3, -1,  1, -4,  5, -6,
                      4, -6, -2,  2,  5, -8,  8,  5,  0, -1, -1, -4 },
        CC_ans[] = {   84, -145,  -29,  -51,  -74,  -13,
                      221, -145,  123, -139,   18, -110,
                      -14,   23, -158,  -38,  -18,  164,
                      142,  -73,   85, -115,   80,  -41,
                      -71,   40,  -99,  128,    1,  157,
                       52,    2,  107,  -39,   70, -127 };

    // Copy given nxn matrices into corresponding randomly generated matrices
    copy_matrix( n, n, AA, A );
    copy_matrix( n, n, BB, B );
    copy_matrix( n, n, CC, C );
    copy_matrix( n, n, CC_ans, C_ans );
}

/*
 * Initializes nxn matrices used to test parallel matrix multiplication.
 * A, B and C are randomly generated matrices, and C_ans contains the result of
 * C_ans = C + A*B, computed by serial matrix multiplication. The matrices are
 * stored in column-major order with leading dimension n.
 */
void init_test_matrices( const int n, double *A, double *B, double *C,
    double *C_ans )
{
    const double    alpha = 10.0;   // Scaling factor for random matrix

    create_random_matrix( alpha, n, n, A );
    create_random_matrix( alpha, n, n, B );
    create_random_matrix( alpha, n, n, C );
    // Compute C_ans = C + A*B using serial matrix multiply
    copy_matrix( n, n, C, C_ans );
    serial_matrix_multiply( n, A, B, C_ans );
}

/***************************************************************************/

/*
 * Checks whether the serial matrix multiply (jki indexing) algorithm performs
 * matrix multiplication (and addition), C = C + A*B, correctly.  The
 * verification is done on the n-by-n matrices passed in the argument list,
 * which are stored in column-major order with leading dimension n.
 */
void test_serial_matrix_multiply( int n,
    const double *A, const double *B, double *C, const double *C_ans,
    double tol, const char *test_name )
{
    double  eps, err;

    // Perform serial matrix multiplication and compare result with correct answer
    serial_matrix_multiply( n, A, B, C );
    error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
    test_assert( eps, tol, test_name );
}

/*
 * Verifies that the Fox algorithm on parallel processors performs matrix
 * multiplication (and addition), C = C + A*B, correctly.  The verification is
 * done on the n-by-n matrices passed in the argument list, which are stored
 * in column-major order with leading dimension n.
 */
void test_parallel_matrix_multiply( int n,
    const double *A, const double *B, double *C, const double *C_ans,
    struct mpi_grid *grid, double tol, const char *test_name )
{
    double  eps, err;

    // Compute C = C + A*B using parallel algorithm and compare with correct answer
    parallel_matrix_multiply( n, A, B, C, grid );
    if ( grid->rank == 0 ) {
        error_matrix_comp_frob( &eps, &err, n, n, C_ans, C );
        test_assert( eps, tol, test_name );
    }
}
```

Appendix B. Header Files

B.1. lapack.h – LAPACK and BLAS routines.

```
#if !defined(LAPACK_H_)
    #define LAPACK_H_ 1

#if defined(__cplusplus)
    extern "C" {
#endif

    // Prototypes for LAPACK and BLAS routines:

    // ILAENV determines the optimal block size for the local environment.
    int ilaenv_( const int *ISPEC, const char *NAME, const char *OPTS,
        const int *N1, const int *N2, const int *N3, const int *N4 );

    // DCOPY copies vector X to vector Y, where N is the number of elements, and
    // INCX and INCY are increments of elements of X and Y, respectively.
    int dcopy_( const int *N, const double *X, const int *INCX,
        double *Y, const int *INCY );

    // DSWAP interchanges vectors SX and SY, where N is number of elements, and
    // INCX and INCY are increments of elements of SX and SY, respectively.
    int dswap_( const int *N, const double *SX, const int *INCX,
        double *SY, const int *INCY );

    // IDAMAX returns the index of the element of DX having the maximum absolute
    // value (maximum magnitude), where N is the number of elements and INCX
    // is the increment of elements of DX.
    int idamax_( const int *N, const double *DX, const int *INCX );

    // DGEMM performs matrix multiplication, C = beta*C + alpha * op(A) * op(B),
    // where alpha and beta are scalars,
    // C, A and B are M-by-N, M-by-K and K-by-N matrices, respectively,
    // and op(X) = X or op(X) = X'.
    int dgemm_( const char *TRANSA, const char *TRANSB,
        const int *M, const int *N, const int *K,
        const double *ALPHA, const double *A, const int *LDA,
        const double *B, const int *LDB,
        const double *BETA, double *C, const int *LDC );

    // DGEMV performs matrix-vector operation y = beta*y + alpha * op(A) * x,
    // where alpha and beta are scalars, x and y are vectors, A is an M-by-N
    // matrix, and op(A) = A or op(A) = A'.
    int dgemv_( const char *TRANS, const int *M, const int *N,
        const double *ALPHA, const double *A, const int *LDA,
        const double *X, const int *INCX,
        const double *BETA, double *Y, const int *INCY );

    // DSYRK performs symmetric rank k operation
    // C = alpha*op(A)*op(A)' + beta*C, where alpha and beta are scalars,
    // C is an N-by-N symmetric matrix and A is an N-by-K matrix, and
    // op(X) = X or op(X) = X'.
    int dsyrk_( const char *UPLO, const char *TRANS, const int *N,
        const int *K, const double *ALPHA, const double *A, const int *LDA,
        const double *BETA, double *C, const int *LDC );

    // DTRSM solves a triangular system of the form
    // op(A)*X = alpha*B or X*op(A) = alpha*B,
    // where alpha is a scalar, X and B are m-by-n matrices, A is a triangular
    // matrix, and op(A) = A or op(A) = A'.
    int dtrsm_( const char *SIDE, const char *UPLO, const char *TRANSA,
        const char *DIAG, const int *M, const int *N, const double *ALPHA,
        const double *A, const int *LDA, double *B, const int *LDB );

    // DTRTRS solves a triangular system of the form A*X = B or A'*X = B,
    // where A is a triangular matrix of order N, and B is an N-by-NRHS matrix.
    int dtrtrs_( const char *UPLO, const char *TRANS, const char *DIAG,
        const int *N, const int *NRHS, const double *A, const int *LDA,
        double *B, const int *LDB, int *INFO );

    // DGETRF computes an LU factorization of a general M-by-N matrix A using
    // partial pivoting with row interchanges. The factorization takes the form
    // A = P*L*U, where P is the permutation matrix encoded in the vector IPIV,
    // L is unit lower triangular and U is upper triangular. On exit, L and U
    // overwrite A.
    int dgetrf_( const int *M, const int *N, double *A, const int *LDA,
        int *IPIV, int *INFO );

    // DPOTF2 computes the Cholesky factorization of an N-by-N real symmetric
    // positive definite matrix A. The factorization takes the form A = L*L' or
    // A = U'*U, where L is lower triangular and U is upper triangular. On exit,
    // L or U overwrites A. This is LAPACK's unblocked version of Cholesky
    // factorization.
    int dpotf2_( const char *UPLO, const int *N, double *A, const int *LDA,
        int *INFO );

    // DPOTRF computes the Cholesky factorization of an N-by-N real symmetric
    // positive definite matrix A. The factorization takes the form A = L*L' or
    // A = U'*U, where L is lower triangular and U is upper triangular. On exit,
    // L or U overwrites A.
    int dpotrf_( const char *UPLO, const int *N, double *A, const int *LDA,
        int *INFO );

    // DSYTRF computes the factorization of a real symmetric matrix A using
    // the Bunch-Kaufman pivoting. The factorization takes the form
    // A = (P*L)*D*(P*L)' or A = (P*U)*D*(P*U)', where P is the permuation
    // matrix encoded in IPIV, L is unit lower triangular, U is unit upper
    // triangular and D block diagonal with block order 1 or 2. On exit,
    // (P*L) or (P*U) and D overwrite A. This is LAPACK's unblocked version
```

```
    // of symmetric indefinite factorization.
    int dsytf2_(const char *UPLO, const int *N, double *A, const int *LDA,
        int *IPIV, int *INFO);

    // DSYTRF computes the factorization of a real symmetric matrix A using
    // the Bunch-Kaufman pivoting.  The factorization takes the form
    // A = (P*L)*D*(P*L)' or A = (P*U)*D*(P*U)', where P is the permuation
    // matrix encoded in IPIV, L is unit lower triangular, U is unit upper
    // triangular and D block diagonal with block order 1 or 2.  On exit,
    // (P*L) or (P*U) and D overwrite A.
    int dsytrf_(const char *UPLO, const int *N, double *A, const int *LDA,
        int *IPIV, double *WORK, int *LWORK, int *INFO);

#if defined(__cplusplus)
    }
#endif

#endif
```

B.2. **timing.h – timing functions.**

```
#if !defined(TIMING_H_)
    #define TIMING_H_ 1

#if defined(__cplusplus)
    extern "C" {
#endif

    long double timespec_to_ldbl( struct timespec ts );

    long double timespec_diff( struct timespec sta, struct timespec end );

    long double timer_resolution( void );

    void get_time( struct timespec *ts );

#if defined(__cplusplus)
    }
#endif

#endif
```

B.3. **matcom.h – common matrix operations.**

```
#if !defined(MATCOM_H_)
    #define MATCOM_H_ 1

#if defined(__cplusplus)
    extern "C" {
#endif

// BDIM is the blocking parameter, i.e., block size = BDIM-by-BDIM.
// Implementations of blocked algorithms typically use the blocking parameter
// chosen by LAPACK for their routines.  The BDIM parameter facilitates
// overriding the LAPACK chosen blocking parameter.  It is also used during
// testing of blocked algorithms that perform matrix computations.
// For recursive contiguous block storage, matrix computation kernels act on
// sub-blocks of size KDIM-by-KDIM.
#if defined(DEBUG)
    #define BDIM 32
    #define KDIM 8
#else
    #define BDIM 96
    #define KDIM 8
#endif

    void create_random_matrix( double alpha, int m, int n, double *E );

    void create_random_unit_lower( double alpha, int n, double *E );

    void create_random_lower( double alpha, int n, double *E );

    void create_random_upper( double alpha, int n, double *E );

    void create_random_nonsingular( double alpha, int n, double *E );

    void create_random_spd( double alpha, int n, double *E );

    void create_random_symmetric( double alpha, int n, double *E );

    void clear_matrix( int m, int n, double *E );

    void copy_matrix( int m, int n, const double *E, double *F );

    void transpose_matrix( int m, int n, const double *E, double *F );

    void form_contig_blocks( int m, int n, int ldimE, const double *E,
        int mm, int nn, int bdim, int ldimF, double *F );

    void form_recur_blocks( int m, int n, int ldimE, const double *E,
        int mm, int nn, int kdim, int bdim, int ldimF, double *F );

    void unpack_contig_blocks( int mm, int nn, int bdim, int ldimE,
        const double *E, int m, int n, int ldimF, double *F  );

    void unpack_recur_blocks( int mm, int nn, int kdim, int bdim, int ldimE,
        const double *E, int m, int n, int ldimF, double *F );

    void error_matrix_comp_frob( double *eps, double *err, int m, int n,
        const double *E, const double *F );

    void error_matrix_comp_l1( double *eps, double *err, int m, int n,
        const double *E, const double *F );

    void multiply_matrix( int m, int n, int p, int ldimA, const double *A,
        int ldimB, const double *B, int ldimC, double *C );

#if defined(__cplusplus)
    }
#endif

#endif
```

B.4. **lufact.h – Gaussian elimination (LU factorization).**

```
#if !defined(LUFACT_H_)
   #define LUFACT_H_ 1

#if defined(__cplusplus)
   extern "C" {
#endif

   int get_block_dim_lu( int ldim );

   void lu_outer_product( const int n, double *A );

   void lu_saxpy( const int n, double *A );

   void lu_block( const int n, double *A );

   void lu_recur_block( const int n, double *A );

   void lu_pivot_outer_product( const char pivot, const int n,
       int *piv, int *ord, double *A );

   void lu_pivot_saxpy( const char pivot, const int n,
       int *piv, int *ord, double *A );

   void lu_pivot_block( const char pivot, const int n,
       int *piv, int *ord, double *A );

   void lu_pivot_lapack( const char pivot, const int n,
       int *piv, int *ord, double *A );

#if defined(__cplusplus)
   }
#endif

#endif
```

B.5. **cholfact.h – Cholesky factorization.**

```
#if !defined(CHOLFACT_H_)
    #define CHOLFACT_H_ 1

#if defined(__cplusplus)
    extern "C" {
#endif

    int get_block_dim_chol( int ldim );

    void chol_outer_product( int n, double *A );

    void chol_saxpy( int n, double *A );

    void chol_block( int n, double *A );

    void chol_rect_block( int n, double *A );

    void chol_contig_block( int n, double *A );

    void chol_recur_block( int n, double *A );

    void chol_block_blas( int n, double *A );

    void chol_contig_block_blas( int n, double *A );

    void chol_lapack_unblocked( int n, double *A );

    void chol_lapack( int n, double *A );

#if defined(__cplusplus)
    }
#endif

#endif
```

B.6. **ldltfact.h – symmetric indefinite factorization.**

```
#if !defined(LDLTFACT_H_)
    #define LDLTFACT_H_ 1

#if defined(__cplusplus)
    extern "C" {
#endif

    int count_pivot( int piv_ord, int n, const int *piv, const int *ord );

    int get_block_dim_ldlt( int lapack, int blas, int ldim );

    void eval_pivot_diag( int n, int d, const double *diag, int *piv, int *ord );

    void reduce_ldlt_vector_blas( int m, int n, int r, int *ord, int ldim,
        const double *L, const double *M, double *vec );

    void reduce_ldlt_mat_blk( int blas, int m, int n, const int *ord, int bdim,
        int ldim, const double *L, const double *D, const double *M, double *A );

    void pivot_sym( int n, int k, int r, int ldim, double *A );

    void ldlt_outer_product( char pivot, int n, int *piv, int *ord, double *A );

    void ldlt_saxpy( char pivot, int n, int *piv, int *ord, double *A );

    void ldlt_block( char pivot, int n, int *piv, int *ord, double *A );

    void ldlt_block_blas( char pivot, int n, int *piv, int *ord, double *A );

    void ldlt_lapack_unblocked( char pivot, int n, int *piv, int *ord,
        double *A );

    void ldlt_lapack( char pivot, int n, int *piv, int *ord, double *A );

#if defined(__cplusplus)
    }
#endif

#endif
```

B.7. modchol.h – modified Cholesky algorithms.

```
#if !defined(MODCHOL_H_)
    #define MODCHOL_H_ 1

#if defined(__cplusplus)
    extern "C" {
#endif

    void chol_gmw_outer_product( char pivot, int n, int *piv, int *ord,
        double *A );

    void chol_gmw_saxpy( char pivot, int n, int *piv, int *ord, double *A );

    void chol_gmw_block( char pivot, int n, int *piv, int *ord, double *A );

    void chol_gmw_block_blas( char pivot, int n, int *piv, int *ord, double *A );

    void chol_ch_outer_product( char pivot, int n, int *piv, int *ord,
        double *A );

    void chol_ch_saxpy( char pivot, int n, int *piv, int *ord, double *A );

    void chol_ch_block( char pivot, int n, int *piv, int *ord, double *A );

    void chol_ch_block_blas( char pivot, int n, int *piv, int *ord, double *A );

#if defined(__cplusplus)
    }
#endif

#endif
```

B.8. **matmult.h – matrix multiplication.**

```
#if !defined(MATMULT_H_)
    #define MATMULT_H_ 1

#if defined(__cplusplus)
    extern "C" {
#endif

#define UNROLL_DEPTH 8        // Depth of loop unrolling
#define PIPE_DEPTH 8          // Depth of software pipelining

    int get_block_dim_mmult( int ldim );

    void mmult_dot_product( int n, const double *A, const double *B, double *C );

    void mmult_saxpy( int n, const double *A, const double *B, double *C );

    void mmult_unroll( int n, const double *A, const double *B, double *C );

    void mmult_pipeline( int n, const double *A, const double *B, double *C );

    void mmult_block( int n, const double *A, const double *B, double *C );

    void mmult_contig_block( int n,
        const double *A, const double *B, double *C );

    void mmult_recur_block( int n,
        const double *A, const double *B, double *C );

    void mmult_rect_recur_block( int n,
        const double *A, const double *B, double *C );

    void mmult_blas( int n, const double *A, const double *B, double *C );

#if defined(__cplusplus)
    }
#endif

#endif
```

B.9. **matmultp.h – parallel matrix multiplication.**

```
#if !defined(MATMULTP_H_)
    #define MATMULTP_H_ 1

#if defined(__cplusplus)
    extern "C" {
#endif

// Number of matrices of varying sizes for which performance is measured
#define SIZES 11
// Column of input data file containing performance data for serial algorithm
#define COLINSER 6
// Blocking parameter, i.e., block size = BDIM-by-BDIM
#define BDIM 96

    struct mpi_grid {
        MPI_Comm    comm;
        MPI_Comm    row_comm;
        MPI_Comm    col_comm;
        int         p;
        int         q;
        int         row;
        int         col;
        int         rank;
    };

    void setup_mpi_grid( struct mpi_grid *grid );

    void scatter_blocks( int bdim, int n,
        const double *A, const double *B, double *C, struct mpi_grid *grid );

    void gather_blocks( int bdim, int n, double *C, struct mpi_grid *grid );

    void create_random_matrix( double alpha, int m, int n, double *E );

    void clear_matrix( int m, int n, double *E );

    void copy_matrix( int m, int n, const double *E, double *F );

    void multiply_matrix( int m, int n, int p, int ldimA, const double *A,
        int ldimB, const double *B, int ldimC, double *C );

    void blocked_matrix_multiply ( int m, int n, int p, int ldimA,
        const double *A, int ldimB, const double *B, int ldimC, double *C );

    void serial_matrix_multiply( int n,
        const double *A, const double *B, double *C );

    void fox_matrix_multiply( int n,
        double *A, double *B, double *C, struct mpi_grid *grid );
    void parallel_matrix_multiply( int n,
        const double *A, const double *B, double *C, struct mpi_grid *grid );

#if defined(__cplusplus)
    }
#endif

#endif
```

Appendix C. Makefiles

C.1. Makefile – serial programs.

```
# Makefile for timing and testing of matrix computations: matrix multiplication,
# LU factorization (Gaussian elimination), Cholesky factorization, symmetric
# indefinite factorization and modified Cholesky algorithms.
# Sets compiler and linker parameters.
# Compiles source code and links objects to generate executable files:
# mmultime, mmultest, mfactime and mfactest.
# Cleans object and executable files in current directory.

# Hardware specifications
PROC = Intel Xeon 5345
CORES = 4 x dual-core
CLKSPEED = 2.33 GHz
CACHE = 4096 KB per dual-core

# Compiler options
CC = icc
LANGUAGE = -x c++
OPTM = -O3

CFLAGS = -Wall $(LANGUAGE) $(OPTM)
CPPFLAGS = -I$(includedir) -DCHOLFACT -DLDLTFACT -DMODCHOL
CPPFLAGS += "-DCOMPILER=\"$(CC)\"" "-DLANGUAGE=\"$(LANGUAGE)\"" \
    "-DOPTM=\"$(OPTM)\"" "-DPROC=\"$(PROC)\"" "-DCORES=\"$(CORES)\"" \
    "-DCLKSPEED=\"$(CLKSPEED)\"" "-DCACHE=\"$(CACHE)\"" \
    "-DDATADIR=\"$(DATADIR)\""

LDFLAGS = -shared-intel
LIBS =  -L/share/apps/intel/Compiler/11.1/046/mkl/lib/em64t \
        -lmkl_intel_lp64 -lmkl_core -lmkl_intel_thread \
        -L/share/apps/intel/Compiler/11.1/046/lib/intel64 \
        -liomp5 -lpthread -lrt -lstdc++

prefix = /scratch/st1185
projectdir = $(prefix)
sourcedir = $(projectdir)/source
includedir = $(projectdir)/include
DATADIR = $(projectdir)/data
VPATH = $(sourcedir) $(includedir) $(datadir)

objects1 = mmultime.o matmult.o matcom.o timing.o
objects2 = mmultest.o matmult.o matcom.o
objects3 = mfactime.o lufact.o cholfact.o modchol.o ldltfact.o matcom.o timing.o
objects4 = mfactest.o lufact.o cholfact.o modchol.o ldltfact.o matcom.o timing.o
objects = $(objects1) $(objects2) $(objects3) $(objects4)
sources = $(objects:.o=.c)

.PHONY: all
```

```
all: mmultime mmultest mfactime mfactest

mmultime: $(objects1)
mmultime.o: matmult.h matcom.h timing.h
matmult.o: matmult.h lapack.h matcom.h
matcom.o: matcom.h
timing.o: timing.h

mmultest: $(objects2)
mmultest.o: matmult.h matcom.h
matmult.o: matmult.h lapack.h matcom.h
matcom.o: matcom.h

mfactime: $(objects3)
mfactime.o: lufact.h cholfact.h modchol.h ldltfact.h matcom.h timing.h
lufact.o: lufact.h lapack.h matcom.h
cholfact.o: cholfact.h lapack.h matcom.h timing.h
modchol.o: modchol.h ldltfact.h lapack.h matcom.h timing.h
ldltfact.o: ldltfact.h lapack.h matcom.h timing.h
matcom.o: matcom.h
timing.o: timing.h

mfactest: $(objects4)
mfactest.o: lufact.h cholfact.h modchol.h ldltfact.h matcom.h
lufact.o: lufact.h lapack.h matcom.h
cholfact.o: cholfact.h lapack.h matcom.h timing.h
modchol.o: modchol.h ldltfact.h lapack.h matcom.h timing.h
ldltfact.o: ldltfact.h lapack.h matcom.h timing.h
matcom.o: matcom.h
timing.o: timing.h

# Pattern rules
%: %.o
    $(CC) $^ $(LDFLAGS) $(LIBS) -o $@
%.o: %.c
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

.PHONY: cleanall cleanobj
cleanall: cleanobj
    rm -f mmultime mmultest mfactime mfactest
cleanobj:
    rm -f *.o
clean1:
    rm -f mmultime $(objects1)
clean2:
    rm -f mmultime $(objects2)
clean3:
    rm -f mfactime $(objects3)
clean4:
    rm -f mfactest $(objects4)
```

C.2. **Makefile – parallel programs.**

```
# Makefile for timing and testing of parallel matrix multiplication (and addition).
# Sets compiler and linker parameters.
# Compiles source code and links objects to generate executable files:
# mmultmp and mmultstp.
# Cleans object and executable files in current directory.

# Hardware specifications
PROC = Intel Xeon 5345
CORES = 4 x dual-core
CLKSPEED = 2.33 GHz
CACHE = 4096 KB per dual-core

# Compiler options
CC = /usr/mpi/intel/mvapich-1.1.0/bin/mpicc
LANGUAGE = -x c++
OPTM = -O3

CFLAGS = -Wall $(LANGUAGE) $(OPTM)
CPPFLAGS = -I$(includedir)
CPPFLAGS += "-DCOMPILER=\"$(CC)\"" "-DLANGUAGE=\"$(LANGUAGE)\"" \
     "-DOPTM=\"$(OPTM)\"" "-DPROC=\"$(PROC)\"" "-DCORES=\"$(CORES)\"" \
     "-DCLKSPEED=\"$(CLKSPEED)\"" "-DCACHE=\"$(CACHE)\"" \
     "-DDATADIR=\"$(DATADIR)\""

LDFLAGS = -shared-intel
LIBS = -lm -lrt -lstdc++

prefix = /scratch/st1185
projectdir = $(prefix)
sourcedir = $(projectdir)/source
includedir = $(projectdir)/include
DATADIR = $(projectdir)/data
VPATH = $(sourcedir) $(includedir) $(datadir)

objects1 = mmultmp.o matmultp.o timing.o
objects2 = mmultstp.o matmultp.o
objects = $(objects1) $(objects2)
sources = $(objects:.o=.c)

.PHONY: all
all: mmultmp mmultstp

mmultmp: $(objects1)
mmultmp.o: matmultp.h timing.h
matmultp.o: matmultp.h
timing.o: timing.h

mmultstp: $(objects2)
mmultstp.o: matmultp.h
```

```
matmultp.o: matmultp.h

# Pattern rules
%: %.o
    $(CC) $^ $(LDFLAGS) $(LIBS) -o $@
%.o: %.c
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

.PHONY: cleanall cleanobj
cleanall: cleanobj
    rm -f mmultmp mmultstp
cleanobj:
    rm -f *.o
```