User Guide for CHOLMOD: a sparse Cholesky factorization and modification package

Timothy A. Davis
DrTimothyAldenDavis@gmail.com, http://www.suitesparse.com

VERSION 5.0.1, Oct 31, 2023

Abstract

CHOLMOD[1] is a set of routines for factorizing sparse symmetric positive definite matrices of the form $\mathbf{A}$ or $\mathbf{A}\mathbf{A}^\mathsf{T}$, updating/downdating a sparse Cholesky factorization, solving linear systems, updating/downdating the solution to the triangular system $\mathbf{Lx} = \mathbf{b}$, and many other sparse matrix functions for both symmetric and unsymmetric matrices. Its supernodal Cholesky factorization relies on LAPACK and the Level-3 BLAS, and obtains a substantial fraction of the peak performance of the BLAS. Both real and complex matrices are supported. It also includes a non-supernodal $\mathbf{LDL}^T$ factorization method that can factorize symmetric indefinite matrices if all of their leading submatrices are well-conditioned ($\mathbf{D}$ is diagonal). CHOLMOD is written in ANSI/ISO C, with both C and MATLAB interfaces. This code works on Microsoft Windows and many versions of Unix and Linux.

[1]CHOLMOD is short for CHOLesky MODification, since a key feature of the package is its ability to update/downdate a sparse Cholesky factorization

Contents

1 Overview

CHOLMOD is a set of ANSI C routines for solving systems of linear equations, $\mathbf{Ax} = \mathbf{b}$, when $\mathbf{A}$ is sparse and symmetric positive definite, and $\mathbf{x}$ and $\mathbf{b}$ can be either sparse or dense.[2] Complex matrices are supported, in two different formats. CHOLMOD includes high-performance left-looking supernodal factorization and solve methods [21], based on LAPACK [3] and the BLAS [12]. After a matrix is factorized, its factors can be updated or downdated using the techniques described by Davis and Hager in [8, 9, 10]. Many additional sparse matrix operations are provided, for both symmetric and unsymmetric matrices (square or rectangular), including sparse matrix multiply, add, transpose, permutation, scaling, norm, concatenation, sub-matrix access, and converting to alternate data structures. Interfaces to many ordering methods are provided, including minimum degree (AMD [1, 2], COLAMD [6, 7]), constrained minimum degree (CSYMAMD, CCOLAMD, CAMD), and graph-partitioning-based nested dissection (METIS [18]). Most of its operations are available within MATLAB via mexFunction interfaces.

CHOLMOD also includes a non-supernodal $\mathbf{LDL}^T$ factorization method that can factorize symmetric indefinite matrices if all of their leading submatrices are well-conditioned ($\mathbf{D}$ is diagonal).

A pair of articles on CHOLMOD appears in the ACM Transactions on Mathematical Software: [4, 11].

CHOLMOD 1.0 replaces `chol` (the sparse case), `symbfact`, and `etree` in MATLAB 7.2 (R2006a), and is used for `x=A\b` when `A` is symmetric positive definite [14]. It will replace `sparse` in a future version of MATLAB.

The C-callable CHOLMOD library consists of 133 user-callable routines and one include file. Each routine comes in two versions, one for `int` integers and another for `long`. Many of the routines can support either real or complex matrices, simply by passing a matrix of the appropriate type.

Nick Gould, Yifan Hu, and Jennifer Scott have independently tested CHOLMOD's performance, comparing it with nearly a dozen or so other solvers [17, 16]. Its performance was quite competitive.

[2] Some support is provided for symmetric indefinite matrices.

2 Single-precision sparse matrix support

CHOLMOD v5.0.0: introduces the first part of support for single precision sparse matrices, with the introduction of the new CHOLMOD:Utility Module. The CHOLMOD:Utility Module replaces the CHOLMOD:Core Module that appeared in prior versions of CHOLMOD. Single precision is not yet incorporated into the remaining Modules, however. This feature will be implemented soon in a later version of CHOLMOD.

3 Primary routines and data structures

Five primary CHOLMOD routines are required to factorize $\mathbf{A}$ or $\mathbf{AA}^\mathsf{T}$ and solve the related system $\mathbf{Ax} = \mathbf{b}$ or $\mathbf{AA}^\mathsf{T}\mathbf{x} = \mathbf{b}$, for either the real or complex cases:

1. `cholmod_start`: This must be the first call to CHOLMOD.

2. `cholmod_analyze`: Finds a fill-reducing ordering, and performs the symbolic factorization, either simplicial (non-supernodal) or supernodal.

3. `cholmod_factorize`: Numerical factorization, either simplicial or supernodal, $\mathbf{LL}^\mathsf{T}$ or $\mathbf{LDL}^\mathsf{T}$ using either the symbolic factorization from `cholmod_analyze` or the numerical factorization from a prior call to `cholmod_factorize`.

4. `cholmod_solve`: Solves $\mathbf{Ax} = \mathbf{b}$, or many other related systems, where $\mathbf{x}$ and $\mathbf{b}$ are dense matrices. The `cholmod_spsolve` routine handles the sparse case. Any mixture of real and complex $\mathbf{A}$ and $\mathbf{b}$ are allowed.

5. `cholmod_finish`: This must be the last call to CHOLMOD.

Additional routines are also required to create and destroy the matrices $\mathbf{A}$, $\mathbf{x}$, $\mathbf{b}$, and the $\mathbf{LL}^\mathsf{T}$ or $\mathbf{LDL}^\mathsf{T}$ factorization. CHOLMOD has five kinds of data structures, referred to as objects and implemented as pointers to `struct`'s:

1. `cholmod_common`: parameter settings, statistics, and workspace used internally by CHOLMOD. See Section 13 for details.

2. `cholmod_sparse`: a sparse matrix in compressed-column form, either pattern-only, real, complex, or "zomplex." In its basic form, the matrix `A` contains:

   - `A->p`, an integer array of size `A->ncol+1`.
   - `A->i`, an integer array of size `A->nzmax`.
   - `A->x`, a `double` array of size `A->nzmax` or twice that for the complex case. This is compatible with the Fortran and ANSI C99 complex data type.
   - `A->z`, a `double` array of size `A->nzmax` if `A` is zomplex. A zomplex matrix has a `z` array, thus the name. This is compatible with the MATLAB representation of complex matrices.

   For all four types of matrices, the row indices of entries of column `j` are located in `A->i [A->p [j] ... A->p [j+1]-1]`. For a real matrix, the corresponding numerical values are in `A->x` at the same location. For a complex matrix, the entry whose row index is `A->i [p]` is contained in `A->x [2*p]` (the real part) and `A->x [2*p+1]` (the imaginary part). For a zomplex matrix, the real part is in `A->x [p]` and imaginary part is in `A->z [p]`. See Section 14 for more details.

3. `cholmod_factor`: A symbolic or numeric factorization, either real, complex, or zomplex. It can be either an $\mathbf{LL}^\mathsf{T}$ or $\mathbf{LDL}^\mathsf{T}$ factorization, and either simplicial or supernodal. You will normally not need to examine its contents. See Section 15 for more details.

4. `cholmod_dense`: A dense matrix, either real, complex or zomplex, in column-major order. This differs from the row-major convention used in C. A dense matrix `X` contains

   - `X->x`, a double array of size `X->nzmax` or twice that for the complex case.
   - `X->z`, a double array of size `X->nzmax` if `X` is zomplex.

   For a real dense matrix x_{ij} is `X->x [i+j*d]` where `d = X->d` is the leading dimension of `X`. For a complex dense matrix, the real part of x_{ij} is `X->x [2*(i+j*d)]` and the imaginary part is `X->x [2*(i+j*d)+1]`. For a zomplex dense matrix, the real part of x_{ij} is `X->x [i+j*d]` and the imaginary part is `X->z [i+j*d]`. Real and complex dense matrices can be passed to LAPACK and the BLAS. See Section 16 for more details.

5. `cholmod_triplet`: CHOLMOD's sparse matrix (`cholmod_sparse`) is the primary input for nearly all CHOLMOD routines, but it can be difficult for the user to construct. A simpler method of creating a sparse matrix is to first create a `cholmod_triplet` matrix, and then convert it to a `cholmod_sparse` matrix via the `cholmod_triplet_to_sparse` routine. In its basic form, the triplet matrix `T` contains

   - `T->i` and `T->j`, integer arrays of size `T->nzmax`.
   - `T->x`, a double array of size `T->nzmax` or twice that for the complex case.
   - `T->z`, a double array of size `T->nzmax` if `T` is zomplex.

   The `k`th entry in the data structure has row index `T->i [k]` and column index `T->j [k]`. For a real triplet matrix, its numerical value is `T->x [k]`. For a complex triplet matrix, its real part is `T->x [2*k]` and its imaginary part is `T->x [2*k+1]`. For a zomplex matrix, the real part is `T->x [k]` and imaginary part is `T->z [k]`. The entries can be in any order, and duplicates are permitted. See Section 17 for more details.

Each of the five objects has a routine in CHOLMOD to create and destroy it. CHOLMOD provides many other operations on these objects as well. A few of the most important ones are illustrated in the sample program in the next section.

4 Simple example program

```
#include "cholmod.h"
int main (void)
{
    cholmod_sparse *A ;
    cholmod_dense *x, *b, *r ;
    cholmod_factor *L ;
    double one [2] = {1,0}, m1 [2] = {-1,0} ;       /* basic scalars */
    cholmod_common c ;
    cholmod_start (&c) ;                            /* start CHOLMOD */
    A = cholmod_read_sparse (stdin, &c) ;           /* read in a matrix */
    cholmod_print_sparse (A, "A", &c) ;             /* print the matrix */
    if (A == NULL || A->stype == 0)                 /* A must be symmetric */
    {
        cholmod_free_sparse (&A, &c) ;
        cholmod_finish (&c) ;
        return (0) ;
    }
    b = cholmod_ones (A->nrow, 1, A->xtype, &c) ;   /* b = ones(n,1) */
    L = cholmod_analyze (A, &c) ;                   /* analyze */
    cholmod_factorize (A, L, &c) ;                  /* factorize */
    x = cholmod_solve (CHOLMOD_A, L, b, &c) ;       /* solve Ax=b */
    r = cholmod_copy_dense (b, &c) ;                /* r = b */
#ifndef NMATRIXOPS
    cholmod_sdmult (A, 0, m1, one, x, r, &c) ;      /* r = r-Ax */
    printf ("norm(b-Ax) %8.1e\n",
            cholmod_norm_dense (r, 0, &c)) ;        /* print norm(r) */
#else
    printf ("residual norm not computed (requires CHOLMOD/MatrixOps)\n") ;
#endif
    cholmod_free_factor (&L, &c) ;                  /* free matrices */
    cholmod_free_sparse (&A, &c) ;
    cholmod_free_dense (&r, &c) ;
    cholmod_free_dense (&x, &c) ;
    cholmod_free_dense (&b, &c) ;
    cholmod_finish (&c) ;                           /* finish CHOLMOD */
    return (0) ;
}
```

Purpose: The `Demo/cholmod_simple.c` program illustrates the basic usage of CHOLMOD. It reads a triplet matrix from a file (in Matrix Market format), converts it into a sparse matrix, creates a linear system, solves it, and prints the norm of the residual.

See the `CHOLMOD/Demo/cholmod_demo.c` program for a more elaborate example, and `CHOLMOD/Demo/cholmod_l_demo.c` for its `long` integer version.

5 Installation of the C-callable library

CHOLMOD requires a suite of external packages, many of which are distributed along with CHOLMOD, but three of which are not. Those included with CHOLMOD are:

- AMD: an approximate minimum degree ordering algorithm, by Tim Davis, Patrick Amestoy, and Iain Duff [1, 2].
- COLAMD: an approximate column minimum degree ordering algorithm, by Tim Davis, Stefan Larimore, John Gilbert, and Esmond Ng [6, 7].
- CCOLAMD: a constrained approximate column minimum degree ordering algorithm, by Tim Davis and Siva Rajamanickam, based directly on COLAMD. This package is not required if CHOLMOD is compiled with the `-DNCAMD` flag.
- CAMD: a constrained approximate minimum degree ordering algorithm, by Tim Davis and Yanqing Chen, based directly on AMD. This package is not required if CHOLMOD is compiled with the `-DNCAMD` flag.
- `SuiteSparse_config`: a single place where all sparse matrix packages authored or co-authored by Davis are configured.

Three other packages are required for optimal performance:

- `METIS 5.1.0`: a graph partitioning package by George Karypis, Univ. of Minnesota. Not needed if `-DNPARTITION` is used. See http://www-users.cs.umn.edu/∼karypis/metis.
- BLAS: the Basic Linear Algebra Subprograms. Not needed if `-DNSUPERNODAL` is used. See http://www.netlib.org for the reference BLAS (not meant for production use). For Kazushige Goto's optimized BLAS (highly recommended for CHOLMOD) see http://www.tacc.utexas.edu/∼kgoto/ or http://www.cs.utexas.edu/users/flame/goto/. I recommend that you avoid the Intel MKL BLAS; one recent version returns NaN's, where both the Goto BLAS and the standard Fortran reference BLAS return the correct answer. See `CHOLMOD/README` for more information.
- LAPACK: the Basic Linear Algebra Subprograms. Not needed if `-DNSUPERNODAL` is used. See http://www.netlib.org.
- CUDA BLAS: CHOLMOD can exploit an NVIDIA GPU by using the CUDA BLAS for large supernodes. This feature is new to CHOLMOD v2.0.0.

You must first obtain and install LAPACK, and the BLAS. METIS 5.1.0 is optional; a copy of it is in `SuiteSparse_metis`.
CHOLMOD's specific settings are given by the `CHOLMOD_CONFIG` string:

- `-DNCHECK`: do not include the Check module.
- `-DNCHOLESKY`: do not include the Cholesky module.
- `-DNPARTITION`: do not include the interface to METIS in the Partition module.

- `-DCAMD`: do not include the interfaces to CAMD, CCOLAMD, and CSYMAMD in the Partition module.
- `-DNMATRIXOPS`: do not include the MatrixOps module. Note that the Demo requires the MatrixOps module.
- `-DNMODIFY`: do not include the Modify module.
- `-DNSUPERNODAL`: do not include the Supernodal module.
- `-DNPRINT`: do not print anything.

SuiteSparse now has a complete `cmake`-based build system. Each package (SuiteSparse_congig, AMD, CAMD, CCOLAMD, CAMD, and CHOLMOD) has its own `CMakeLists.txt`. Use `cmake` to build each package in that order.

An optional `Makefile` is provided at the top-level of `SuiteSparse` Type `make` in that directory. The AMD, COLAMD, CAMD, CCOLAMD, and `CHOLMOD` libraries will be compiled. To compile and run demo programs for each package, type `make demos`. For CHOLMOD, the residuals should all be small.

CHOLMOD is now ready for use in your own applications. You must link your programs with the `libcholmod.*`, `libamd.*`, `libcolamd.*`, LAPACK, and BLAS libraries. Unless `-DNCAMD` is present at compile time, you must link with `CAMD/libcamd.*`, and `CCOLAMD/libccolamd.*`. Each library has its own `Find*.cmake` script to use in the `cmake find_library` command.

To install CHOLMOD in default locations use `make install`. To remove CHOLMOD, do `make uninstall`.

6 Using CHOLMOD in MATLAB

CHOLMOD includes a set of m-files and mexFunctions in the CHOLMOD/MATLAB directory. The following functions are provided:

`analyze`	order and analyze a matrix
`bisect`	find a node separator
`chol2`	same as `chol`
`cholmod2`	same as `x=A\b` if `A` is symmetric positive definite
`cholmod_demo`	a short demo program
`cholmod_make`	compiles CHOLMOD for use in MATLAB
`etree2`	same as `etree`
`graph_demo`	graph partitioning demo
`lchol`	`L*L'` factorization
`ldlchol`	`L*D*L'` factorization
`ldl_normest`	estimate `norm(A-L*D*L')`
`ldlsolve`	`x = L'\(D\(L\b))`
`ldlsplit`	split the output of `ldlchol` into `L` and `D`
`ldlupdate`	update/downdate an `L*D*L'` factorization
`ldlrowmod`	add/delete a row from an `L*D*L'` factorization
`metis`	interface to `METIS_NodeND` ordering
`mread`	read a sparse or dense Matrix Market file
`mwrite`	write a sparse or dense Matrix Market file
`nesdis`	CHOLMOD's nested dissection ordering
`resymbol`	recomputes the symbolic factorization
`sdmult`	`S*F` where `S` is sparse and `F` is dense
`spsym`	determine symmetry
`sparse2`	same as `sparse`
`symbfact2`	same as `symbfact`

Each function is described in the next sections.

6.1 analyze: order and analyze

```
ANALYZE order and analyze a matrix using CHOLMOD's best-effort ordering.

  Example:
  [p count] = analyze (A)         orders A, using just tril(A)
  [p count] = analyze (A,'sym')   orders A, using just tril(A)
  [p count] = analyze (A,'row')   orders A*A'
  [p count] = analyze (A,'col')   orders A'*A

  an optional 3rd parameter modifies the ordering strategy:

  [p count] = analyze (A,'sym',k) orders A, using just tril(A)
  [p count] = analyze (A,'row',k) orders A*A'
  [p count] = analyze (A,'col',k) orders A'*A

  Returns a permutation and the count of the number of nonzeros in each
  column of L for the permuted matrix A.  That is, count is returned as:

     count = symbfact2 (A (p,p))       if ordering A
     count = symbfact2 (A (p,:),'row') if ordering A*A'
     count = symbfact2 (A (:,p),'col') if ordering A'*A

  CHOLMOD uses the following ordering strategy:

      k = 0:  Try AMD.  If that ordering gives a flop count >= 500 * nnz(L)
         and a fill-in of nnz(L) >= 5*nnz(C), then try METIS_NodeND (where
         C = A, A*A', or A'*A is the matrix being ordered.  Selects the best
         ordering tried.  This is the default.

      if k > 0, then multiple orderings are attempted.

      k = 1 or 2: just try AMD
      k = 3: also try METIS_NodeND
      k = 4: also try NESDIS, CHOLMOD's nested dissection (NESDIS), with
           default parameters.  Uses METIS's node bisector and CCOLAMD.
      k = 5: also try the natural ordering (p = 1:n)
      k = 6: also try NESDIS with large leaves of the separator tree
      k = 7: also try NESDIS with tiny leaves and no CCOLAMD ordering
      k = 8: also try NESDIS with no dense-node removal
      k = 9: also try COLAMD if ordering A'*A or A*A', (AMD if ordering A).
      k > 9 is treated as k = 9

      k = -1: just use AMD
      k = -2: just use METIS
      k = -3: just use NESDIS

      The method returning the smallest nnz(L) is used for p and count.
      k = 4 takes much longer than (say) k = 0, but it can reduce nnz(L) by
      a typical 5% to 10%.  k = 5 to 9 is getting extreme, but if you have
      lots of time and want to find the best ordering possible, set k = 9.

  If METIS is not installed for use in CHOLMOD, then the strategy is
  different:
```

```
      k = 1 to 4: just try AMD
      k = 5 to 8: also try the natural ordering (p = 1:n)
      k = 9: also try COLAMD if ordering A'*A or A*A', (AMD if ordering A).
      k > 9 is treated as k = 9

   See also METIS, NESDIS, BISECT, SYMBFACT, AMD
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.2 bisect: find a node separator

```
BISECT computes a node separator based on METIS_ComputeVertexSeparator.

   Example:
   s = bisect(A)       bisects A. Uses tril(A) and assumes A is symmetric.
   s = bisect(A,'sym') the same as p=bisect(A).
   s = bisect(A,'col') bisects A'*A.
   s = bisect(A,'row') bisects A*A'.

   A must be square for p=bisect(A) and bisect(A,'sym').

   s is a vector of length equal to the dimension of A, A'*A, or A*A',
   depending on the matrix bisected.  s(i)=0 if node i is in the left subgraph,
   s(i)=1 if it is in the right subgraph, and s(i)=2 if node i is in the node
   separator.

   Requires METIS, authored by George Karypis, Univ. of Minnesota.  This
   MATLAB interface, via CHOLMOD, is by Tim Davis.

   See also METIS, NESDIS
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.3 chol2: same as chol

```
CHOL2 sparse Cholesky factorization, A=R'R.
   Note that A=L*L' (LCHOL) and A=L*D*L' (LDLCHOL) factorizations are faster
   than R'*R (CHOL2 and CHOL) and use less memory.  The LL' and LDL'
   factorization methods use tril(A).  This method uses triu(A), just like
   the built-in CHOL.

   Example:
   R = chol2 (A)                 same as R = chol (A), just faster
   [R,p] = chol2 (A)             same as [R,p] = chol(A), just faster
   [R,p,q] = chol2 (A)           factorizes A(q,q) into R'*R, where q is
                                 a fill-reducing ordering

   A must be sparse.

   See also LCHOL, LDLCHOL, CHOL, LDLUPDATE.
```

6.4 cholmod2: supernodal backslash

```
CHOLMOD2 supernodal sparse Cholesky backslash, x = A\b

   Example:
   x = cholmod2 (A,b)

   Computes the LL' factorization of A(p,p), where p is a fill-reducing
   ordering, then solves a sparse linear system Ax=b. A must be sparse,
   symmetric, and positive definite).  Uses only the upper triangular part
   of A.  A second output, [x,stats]=cholmod2(A,b), returns statistics:

       stats(1)    estimate of the reciprocal of the condition number
       stats(2)    ordering used:
                   0: natural, 1: given, 2:amd, 3:metis, 4:nesdis,
                   5:colamd, 6: natural but postordered.
       stats(3)    nnz(L)
       stats(4)    flop count in Cholesky factorization.  Excludes solution
                   of upper/lower triangular systems, which can be easily
                   computed from stats(3) (roughly 4*nnz(L)*size(b,2)).
       stats(5)    memory usage in MB.

   The 3rd argument select the ordering method to use.  If not present or -1,
   the default ordering strategy is used (AMD, and then try METIS if AMD finds
   an ordering with high fill-in, and use the best method tried).

   Other options for the ordering parameter:

       0   natural (no etree postordering)
       -1  use CHOLMOD's default ordering strategy (AMD, then try METIS)
       -2  AMD, and then try NESDIS (not METIS) if AMD has high fill-in
       -3  use AMD only
       -4  use METIS only
       -5  use NESDIS only
       -6  natural, but with etree postordering
       p   user permutation (vector of size n, with a permutation of 1:n)

   See also CHOL, MLDIVIDE.
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.5 cholmod_demo: a short demo program

```
CHOLMOD_DEMO a demo for CHOLMOD

 Tests CHOLMOD with various randomly-generated matrices, and the west0479
 matrix distributed with MATLAB.  Random matrices are not good test cases,
 but they are easily generated.  It also compares CHOLMOD and MATLAB on the
 sparse matrix problem used in the MATLAB BENCH command.

 See CHOLMOD/MATLAB/Test/cholmod_test.m for a lengthy test using matrices from
 the SuiteSparse Matrix Collection.

 Example:
   cholmod_demo

 See also BENCH
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
 L = full (L) ;
 L = full (L) ;
```

6.6 cholmod_make: compile CHOLMOD in MATLAB

```
CHOLMOD_MAKE compiles the CHOLMOD mexFunctions

 Example:
   cholmod_make

 CHOLMOD relies on AMD and COLAMD, and optionally CCOLAMD, CAMD, and METIS.
 You must type the cholmod_make command while in the CHOLMOD/MATLAB directory.

 See also analyze, bisect, chol2, cholmod2, etree2, lchol, ldlchol, ldlsolve,
   ldlupdate, metis, spsym, nesdis, septree, resymbol, sdmult, sparse2,
   symbfact2, mread, mwrite, ldlrowmod
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
 MATLAB 8.3.0 now has a -silent option to keep 'mex' from burbling too much
 Determine if METIS is available
```

6.7 etree2: same as etree

```
ETREE2  sparse elimination tree.
   Finds the elimination tree of A, A'*A, or A*A', and optionaly postorders
   the tree.  parent(j) is the parent of node j in the tree, or 0 if j is a
   root.  The symmetric case uses only the upper or lower triangular part of
   A (etree2(A) uses the upper part, and etree2(A,'lo') uses the lower part).

   Example:
   parent = etree2 (A)         finds the elimination tree of A, using triu(A)
   parent = etree2 (A,'sym')   same as etree2(A)
   parent = etree2 (A,'col')   finds the elimination tree of A'*A
   parent = etree2 (A,'row')   finds the elimination tree of A*A'
   parent = etree2 (A,'lo')    finds the elimination tree of A, using tril(A)

   [parent,post] = etree2 (...) also returns a post-ordering of the tree.

   If you have a fill-reducing permutation p, you can combine it with an
   elimination tree post-ordering using the following code.  Post-ordering has
   no effect on fill-in (except for lu), but it does improve the performance
   of the subsequent factorization.

   For the symmetric case, suitable for chol(A(p,p)):

       [parent post] = etree2 (A (p,p)) ;
       p = p (post) ;

   For the column case, suitable for qr(A(:,p)) or lu(A(:,p)):

       [parent post] = etree2 (A (:,p), 'col') ;
       p = p (post) ;

   For the row case, suitable for qr(A(p,:)') or chol(A(p,:)*A(p,:)'):

       [parent post] = etree2 (A (p,:), 'row') ;
       p = p (post) ;

   See also TREELAYOUT, TREEPLOT, ETREEPLOT, ETREE
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.8 graph_demo: graph partitioning demo

```
GRAPH_DEMO graph partitioning demo
   graph_demo(n) constructs an set of n-by-n 2D grids, partitions them, and
   plots them in one-second intervals.  n is optional; it defaults to 60.

   Example:
       graph_demo

   See also DELSQ, NUMGRID, GPLOT, TREEPLOT
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.9 `lchol`: $\mathbf{LL^T}$ factorization

```
LCHOL sparse A=L*L' factorization.
   Note that L*L' (LCHOL) and L*D*L' (LDLCHOL) factorizations are faster than
   R'*R (CHOL2 and CHOL) and use less memory.  The LL' and LDL' factorization
   methods use tril(A).  A must be sparse.

   Example:
   L = lchol (A)                 same as L = chol (A')', just faster
   [L,p] = lchol (A)             same as [R,p] = chol(A') ; L=R', just faster
   [L,p,q] = lchol (A)           factorizes A(q,q) into L*L', where q is a
                                 fill-reducing ordering

   See also CHOL2, LDLCHOL, CHOL.
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.10 `ldlchol`: $\mathbf{LDL^T}$ factorization

```
LDLCHOL sparse A=LDL' factorization
   Note that L*L' (LCHOL) and L*D*L' (LDLCHOL) factorizations are faster than
   R'*R (CHOL2 and CHOL) and use less memory.  The LL' and LDL' factorization
   methods use tril(A).  A must be sparse.

   Example:
   LD = ldlchol (A)             return the LDL' factorization of A
   [LD,p] = ldlchol (A)         similar [R,p] = chol(A), but for L*D*L'
   [LD,p,q] = ldlchol (A)       factorizes A(q,q) into L*D*L', where q is a
                                fill-reducing ordering
   LD = ldlchol (A,beta)        return the LDL' factorization of A*A'+beta*I
   [LD,p] = ldlchol (A,beta)    like [R,p] = chol(A*A'+beta+I)
   [LD,p,q] = ldlchol (A,beta) factorizes A(q,:)*A(q,:)'+beta*I into L*D*L'

   The output matrix LD contains both L and D.  D is on the diagonal of LD, and
   L is contained in the strictly lower triangular part of LD.  The unit-
   diagonal of L is not stored.  You can obtain the L and D matrices with
   [L,D] = ldlsplit (LD).  LD is in the form needed by ldlupdate.

   Explicit zeros may appear in the LD matrix.  The pattern of LD matches the
   pattern of L as computed by symbfact2, even if some entries in LD are
   explicitly zero.  This is to ensure that ldlupdate and ldlsolve work
   properly.  You must NOT modify LD in MATLAB itself and then use ldlupdate
   or ldlsolve if LD contains explicit zero entries; ldlupdate and ldlsolve
   will fail catastrophically in this case.

   You MAY modify LD in MATLAB if you do not pass it back to ldlupdate or
   ldlsolve.  Just be aware that LD contains explicit zero entries, contrary
   to the standard practice in MATLAB of removing those entries from all
   sparse matrices.  LD = sparse2 (LD) will remove any zero entries in LD.

   See also LDLUPDATE, LDLSOLVE, LDLSPLIT, CHOL2, LCHOL, CHOL, SPARSE2
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.11 ldlsolve: solve using an LDL^T factorization

```
LDLSOLVE solve LDL'x=b using a sparse LDL' factorization

   Example:
   x = ldlsolve (LD,b)

   solves the system L*D*L'*x=b for x.  This is equivalent to

   [L,D] = ldlsplit (LD) ;
   x = L' \ (D \ (L \ b)) ;

   LD is from ldlchol, or as updated by ldlupdate or ldlrowmod.  You must not
   modify LD as obtained from ldlchol, ldlupdate, or ldlrowmod prior to passing
   it to this function.  See ldlupdate for more details.

   See also LDLCHOL, LDLUPDATE, LDLSPLIT, LDLROWMOD
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.12 ldlsplit: split an LDL^T factorization

```
LDLSPLIT split an LDL' factorization into L and D.

   Example:
   [L,D] = ldlsplit (LD)

   LD contains an LDL' factorization, computed with LD = ldlchol(A),
   for example.  The diagonal of LD contains D, and the entries below
   the diagonal contain L (which has a unit diagonal).  This function
   splits LD into its two components L and D so that L*D*L' = A.

   See also LDLCHOL, LDLSOLVE, LDLUPDATE.
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.13 ldlupdate: update/downdate an LDL^T factorization

```
LDLUPDATE multiple-rank update or downdate of a sparse LDL' factorization.

   On input, LD contains the LDL' factorization of A (L*D*L'=A or A(q,q)).
   The unit-diagonal of L is not stored.  In its place is the diagonal matrix
   D.  LD can be computed using the CHOLMOD mexFunctions:

       LD = ldlchol (A) ;
   or
       [LD,p,q] = ldlchol (A) ;

   With this LD, either of the following MATLAB statements,

   Example:
       LD = ldlupdate (LD,C)
       LD = ldlupdate (LD,C,'+')

   return the LDL' factorization of A+C*C' or A(q,q)-C*C' if LD holds the LDL'
   factorization of A(q,q) on input.  For a downdate:

       LD = ldlupdate (LD,C,'-')

   returns the LDL' factorization of A-C*C' or A(q,q)-C*C'.

   LD and C must be sparse and real.  LD must be square, and C must have the
   same number of rows as LD.  You must not modify LD in MATLAB (see the
   WARNING below).

   Note that if C is sparse with few columns, most of the time spent in this
   routine is taken by copying the input LD to the output LD.  If MATLAB
   allowed mexFunctions to safely modify its inputs, this mexFunction would
   be much faster, since not all of LD changes.

   See also LDLCHOL, LDLSPLIT, LDLSOLVE, CHOLUPDATE

   ===========================================================================
   =============================== WARNING ===================================
   ===========================================================================
   MATLAB drops zero entries from its sparse matrices.  LD can contain
   numerically zero entries that are symbolically present in the sparse matrix
   data structure.  These are essential for ldlupdate and ldlsolve to work
   properly, since they exploit the graph-theoretic structure of a sparse
   Cholesky factorization. If you modify LD in MATLAB, those zero entries may
   get dropped and the required graph property will be destroyed.  In this
   case, ldlupdate and ldlsolve will fail catastrophically (possibly with a
   segmentation fault, terminating MATLAB).  It takes much more time to ensure
   this property holds than the time it takes to do the update/downdate or the
   solve, so ldlupdate and ldlsolve simply assume the propertly holds.
   ===========================================================================
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.14 `ldlrowmod`: add/delete a row from an LDL^T factorization

```
LDLROWMOD add/delete a row from a sparse LDL' factorization.

   On input, LD contains the LDL' factorization of A (L*D*L'=A or A(q,q)).
   The unit-diagonal of L is not stored.  In its place is the diagonal matrix
   D.  LD can be computed using the CHOLMOD mexFunctions:

       LD = ldlchol (A) ;
   or
       [LD,p,q] = ldlchol (A) ;

   With this LD, either of the following MATLAB statements,

   Example:
       LD = ldlrowmod (LD,k,C)         add row k to an LDL' factorization

   returns the LDL' factorization of S, where S = A except for S(:,k) = C
   and S (k,:) = C.  The kth row of A is assumed to initially be equal to
   the kth row of identity.  To delete a row:

       LD = ldlrowmod (LD,k)           delete row k from an LDL' factorization

   returns the LDL' factorization of S, where S = A except that S(:,k) and
   S (k,:) become the kth column/row of speye(n), repespectively.

   LD and C must be sparse and real.  LD must be square, and C must have the
   same number of rows as LD.  You must not modify LD in MATLAB (see the
   WARNING below).

   Note that if C is sparse with few columns, most of the time spent in this
   routine is taken by copying the input LD to the output LD.  If MATLAB
   allowed mexFunctions to safely modify its inputs, this mexFunction would
   be much faster, since not all of LD changes.

   See also LDLCHOL, LDLSPLIT, LDLSOLVE, CHOLUPDATE, LDLUPDATE

   ===========================================================================
   =============================== WARNING ===================================
   ===========================================================================
   MATLAB drops zero entries from its sparse matrices.  LD can contain
   numerically zero entries that are symbolically present in the sparse matrix
   data structure.  These are essential for ldlrowmod and ldlsolve to work
   properly, since they exploit the graph-theoretic structure of a sparse
   Cholesky factorization. If you modify LD in MATLAB, those zero entries may
   get dropped and the required graph property will be destroyed.  In this
   case, ldlrowmod and ldlsolve will fail catastrophically (possibly with a
   segmentation fault, terminating MATLAB).  It takes much more time to ensure
   this property holds than the time it takes to do the row add/delete or the
   solve, so ldlrowmod and ldlsolve simply assume the propertly holds.
   ===========================================================================
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.15 `mread`: read a sparse or dense matrix from a Matrix Market file

```
MREAD read a sparse matrix from a file in Matrix Market format.

  Example:
  A = mread (filename)
  [A Z] = mread (filename, prefer_binary)

  Unlike MMREAD, only the matrix is returned; the file format is not
  returned.  Explicit zero entries can be present in the file; these are not
  included in A.  They appear as the nonzero pattern of the binary matrix Z.

  If prefer_binary is not present, or zero, a symmetric pattern-only matrix
  is returned with A(i,i) = 1+length(find(A(:,i))) if it is present in the
  pattern, and A(i,j) = -1 for off-diagonal entries.  If you want the original
  Matrix Market matrix in this case, simply use A = mread (filename,1).

  Compare with mmread.m at http://math.nist.gov/MatrixMarket

  See also load
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.16 `mwrite`: write a sparse or dense matrix to a Matrix Market file

```
MWRITE write a matrix to a file in Matrix Market form.

  Example:
       mtype = mwrite (filename, A, Z, comments_filename)

 A can be sparse or full.

 If present and non-empty, A and Z must have the same dimension.  Z contains
 the explicit zero entries in the matrix (which MATLAB drops).  The entries
 of Z appear as explicit zeros in the output file.  Z is optional.  If it is
 an empty matrix it is ignored.  Z must be sparse or empty, if present.
 It is ignored if A is full.

 filename is the name of the output file.  comments_filename is the file
 whose contents are include after the Matrix Market header and before the
 first data line.  Ignored if an empty string or not present.

 See also mread.
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.17 `metis`: order with METIS

```
METIS nested dissection ordering via METIS_NodeND.

   Example:
   p = metis(A)        returns p such chol(A(p,p)) is typically sparser than
                       chol(A).  Uses tril(A) and assumes A is symmetric.
   p = metis(A,'sym')  the same as p=metis(A).
   p = metis(A,'col')  returns p so that chol(A(:,p)'*A(:,p)) is typically
                       sparser than chol(A'*A).
   p = metis(A,'row')  returns p so that chol(A(p,:)*A(p,:)') is typically
                       sparser than chol(A'*A).

   A must be square for p=metis(A) or metis(A,'sym')

   Requires METIS, authored by George Karypis, Univ. of Minnesota.  This
   MATLAB interface, via CHOLMOD, is by Tim Davis.

   See also NESDIS, BISECT
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.18 nesdis: order with CHOLMOD nested dissection

```
NESDIS nested dissection ordering via CHOLMOD's nested dissection.

   Example:
   p = nesdis(A)         returns p such chol(A(p,p)) is typically sparser than
                         chol(A).  Uses tril(A) and assumes A is symmetric.
   p = nesdis(A,'sym')   the same as p=nesdis(A).
   p = nesdis(A,'col')   returns p so that chol(A(:,p)'*A(:,p)) is typically
                         sparser than chol(A'*A).
   p = nesdis(A,'row')   returns p so that chol(A(p,:)*A(p,:)') is typically
                         sparser than chol(A'*A).

   A must be square for p=nesdis(A) or nesdis(A,'sym').

   With three output arguments, [p cp cmember] = nesdis(...), the separator
   tree and node-to-component mapping is returned.  cmember(i)=c means that
   node i is in component c, where c is in the range of 1 to the number of
   components.  length(cp) is the number of components found.  cp is the
   separator tree; cp(c) is the parent of component c, or 0 if c is a root.
   There can be anywhere from 1 to n components, where n is dimension of A,
   A*A', or A'*A.  cmember is a vector of length n.

   An optional 3rd input argument, nesdis (A,mode,opts), modifies the default
   parameters.  opts(1) specifies the smallest subgraph that should not be
   partitioned (default is 200).  opts(2) is 0 by default; if nonzero,
   connected components (formed after the node separator is removed) are
   partitioned independently.  The default value tends to lead to a more
   balanced separator tree, cp.  opts(3) defines when a separator is kept; it
   is kept if the separator size is < opts(3) times the number of nodes in the
   graph being cut (valid range is 0 to 1, default is 1).

   opts(4) specifies graph is to be ordered after it is dissected.  For the
   'sym' case: 0: natural ordering, 1: CAMD, 2: CSYMAMD.  For other cases:
   0: natural ordering, nonzero: CCOLAMD.  The default is 1, to use CAMD for
   the symmetric case and CCOLAMD for the other cases.

   If opts is shorter than length 4, defaults are used for entries
   that are not present.

   NESDIS uses METIS' node separator algorithm to recursively partition the
   graph.  This gives a set of constraints (cmember) that is then passed to
   CCOLAMD, CSYMAMD, or CAMD, constrained minimum degree ordering algorithms.
   NESDIS typically takes slightly more time than METIS (METIS_NodeND), but
   tends to produce better orderings.

   Requires METIS, authored by George Karypis, Univ. of Minnesota.  This
   MATLAB interface, via CHOLMOD, is by Tim Davis.

   See also METIS, BISECT, AMD
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.19 `resymbol`: re-do symbolic factorization

```
RESYMBOL recomputes the symbolic Cholesky factorization of the matrix A.

   Example:
   L = resymbol (L, A)

   Recompute the symbolic Cholesky factorization of the matrix A.  A must be
   symmetric.  Only tril(A) is used.  Entries in L that are not in the Cholesky
   factorization of A are removed from L.  L can be from an LL' or LDL'
   factorization (lchol or ldlchol).  resymbol is useful after a series of
   downdates via ldlupdate or ldlrowmod, since downdates do not remove any
   entries in L.  The numerical values of A are ignored; only its nonzero
   pattern is used.

 See also LCHOL, LDLUPDATE, LDLROWMOD
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.20 `sdmult`: sparse matrix times dense matrix

```
SDMULT sparse matrix times dense matrix
   Compute C = S*F or S'*F where S is sparse and F is full (C is also sparse).
   S and F must both be real or both be complex.  This function is
   substantially faster than the MATLAB expression C=S*F when F has many
   columns.

   Example:
       C = sdmult (S,F) ;       C = S*F
       C = sdmult (S,F,0) ;     C = S*F
       C = sdmult (S,F,1) ;     C = S'*F

   See also MTIMES
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.21 spsym: determine symmetry

```
SPSYM determine if a sparse matrix is symmetric, Hermitian, or skew-symmetric.
   If so, also determine if its diagonal has all positive real entries.
   A must be sparse.

   Example:
   result = spsym (A) ;
   result = spsym (A,quick) ;

   If quick = 0, or is not present, then this routine returns:

       1: if A is rectangular
       2: if A is unsymmetric
       3: if A is symmetric, but with one or more A(j,j) <= 0
       4: if A is Hermitian, but with one or more A(j,j) <= 0 or with
           nonzero imaginary part
       5: if A is skew symmetric (and thus the diagonal is all zero as well)
       6: if A is symmetric with real positive diagonal
       7: if A is Hermitian with real positive diagonal

   If quick is nonzero, then the function can return more quickly, as soon as
   it finds a diagonal entry that is <= 0 or with a nonzero imaginary part.
   In this case, it returns 2 for a square matrix, even if the matrix might
   otherwise be symmetric or Hermitian.

   Regardless of the value of "quick", this function returns 6 or 7 if A is
   a candidate for sparse Cholesky.

   For an MATLAB M-file function that computes the same thing as this
   mexFunction (but much slower), see the get_symmetry function by typing
   "type spsym".

   This spsym function does not compute the transpose of A, nor does it need
   to examine the entire matrix if it is unsymmetric.  It uses very little
   memory as well (just size-n workspace, where n = size (A,1)).

   Examples:
       load west0479
       A = west0479 ;
       spsym (A)
       spsym (A+A')
       spsym (A-A')
       spsym (A+A'+3*speye(size(A,1)))

   See also mldivide.
       function result = get_symmetry (A,quick)
       %GET_SYMMETRY: does the same thing as the spsym mexFunction.
       % It's just a lot slower and uses much more memory.  This function
       % is meant for testing and documentation only.
       [m n] = size (A) ;
       if (m ~= n)
           result = 1 ;            % rectangular
           return
       end
```

```
      if (nargin < 2)
          quick = 0 ;
      end
      d = diag (A) ;
      posdiag = all (real (d) > 0) & all (imag (d) == 0) ;
      if (quick & ~posdiag)
          result = 2 ;            % Not a candidate for sparse Cholesky.
      elseif (~isreal (A) & nnz (A-A') == 0)
          if (posdiag)
              result = 7 ;        % complex Hermitian, with positive diagonal
          else
              result = 4 ;        % complex Hermitian, nonpositive diagonal
          end
      elseif (nnz (A-A.') == 0)
          if (posdiag)
              result = 6 ;        % symmetric with positive diagonal
          else
              result = 3 ;        % symmetric, nonpositive diagonal
          end
      elseif (nnz (A+A.') == 0)
          result = 5 ;            % skew symmetric
      else
          result = 2 ;            % unsymmetric
      end
With additional outputs, spsym computes the following for square matrices:
(in this case "quick" is ignored, and set to zero):

[result xmatched pmatched nzoffdiag nnzdiag] = spsym(A)

  xmatched is the number of nonzero entries for which A(i,j) = conj(A(j,i)).
  pmatched is the number of entries (i,j) for which A(i,j) and A(j,i) are
  both in the pattern of A (the value doesn't matter).  nzoffdiag is the
  total number of off-diagonal entries in the pattern.  nzdiag is the number
  of diagonal entries in the pattern.  If the matrix is rectangular,
  xmatched, pmatched, nzoffdiag, and nzdiag are not computed (all of them are
  returned as zero).  Note that a matched pair, A(i,j) and A(j,i) for i != j,
  is counted twice (once per entry).
Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
SPDX-License-Identifier: GPL-2.0+
```

6.22 sparse2: same as sparse

```
SPARSE2 replacement for SPARSE

   Example:
   S = sparse2 (i,j,s,m,n,nzmax)

   Identical to the MATLAB sparse function (just faster).
   An additional feature is added that is not part of the MATLAB sparse
   function, the Z matrix.  With an extra output,

   [S Z] = sparse2 (i,j,s,m,n,nzmax)

   the matrix Z is a binary real matrix whose nonzero pattern contains the
   explicit zero entries that were dropped from S.  Z only contains entries
   for the sparse2(i,j,s,...) usage.  [S Z]=sparse2(X) where X is full always
   returns Z with nnz(Z) = 0, as does [S Z]=sparse2(m,n).  More precisely,
   Z is the following matrix (where ... means the optional m, n, and nzmax
   parameters).

       S = sparse (i,j,s, ...)
       Z = spones (sparse (i,j,1, ...)) - spones (S)

   See also sparse.
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

6.23 symbfact2: same as symbfact

```
SYMBFACT2  symbolic factorization

   Analyzes the Cholesky factorization of A, A'*A, or A*A'.

   Example:
   count = symbfact2 (A)               returns row counts of R=chol(A)
   count = symbfact2 (A,'col')         returns row counts of R=chol(A'*A)
   count = symbfact2 (A,'sym')         same as symbfact2(A)
   count = symbfact2 (A,'lo')          same as symbfact2(A'), uses tril(A)
   count = symbfact2 (A,'row')         returns row counts of R=chol(A*A')

   The flop count for a subsequent LL' factorization is sum(count.^2)

   [count, h, parent, post, R] = symbfact2 (...) returns:

       h: height of the elimination tree
       parent: the elimination tree itself
       post: postordering of the elimination tree
       R: a 0-1 matrix whose structure is that of chol(A) for the symmetric
           case, chol(A'*A) for the 'col' case, or chol(A*A') for the
           'row' case.

   symbfact2(A) and symbfact2(A,'sym') uses the upper triangular part of A
   (triu(A)) and assumes the lower triangular part is the transpose of
   the upper triangular part.  symbfact2(A,'lo') uses tril(A) instead.

   With one to four output arguments, symbfact2 takes time almost proportional
   to nnz(A)+n where n is the dimension of R, and memory proportional to
   nnz(A).  Computing the 5th argument takes more time and memory, both
   O(nnz(L)).  Internally, the pattern of L is computed and R=L' is returned.

   The following forms return L = R' instead of R.  They are faster and take
   less memory than the forms above.  They return the same count, h, parent,
   and post outputs.

   [count, h, parent, post, L] = symbfact2 (A,'col','L')
   [count, h, parent, post, L] = symbfact2 (A,'sym','L')
   [count, h, parent, post, L] = symbfact2 (A,'lo', 'L')
   [count, h, parent, post, L] = symbfact2 (A,'row','L')

   See also CHOL, ETREE, TREELAYOUT, SYMBFACT
 Copyright 2006-2022, Timothy A. Davis, All Rights Reserved.
 SPDX-License-Identifier: GPL-2.0+
```

7 Installation for use in MATLAB

7.1 cholmod_make: compiling CHOLMOD in MATLAB

This is the preferred method, since it allows METIS to be reconfigured to use the MATLAB memory-management functions instead of malloc and free; this avoids the issue of METIS terminating MATLAB if it runs out of memory.

Start MATLAB, cd to the CHOLMOD/MATLAB directory, and type cholmod_make in the MATLAB command window. This will compile the MATLAB interfaces for AMD, COLAMD, CAMD, CCOLAMD, METIS, and CHOLMOD.

8 Using CHOLMOD with OpenMP acceleration

CHOLMOD includes OpenMP acceleration for some operations. In CHOLMOD versions prior to v6.0.0, the number of threads to use was controlled by a compile time parameter. This is now replaced with run-time controls.

Common->nthreads_max defaults to omp_get_max_threads(), or 1 if OpenMP is not in use. This value controls the maximum number of threads that CHOLMOD will use. If zero or less, the default is used. The Common->chunk parameter controls how many threads are used when the work to do is low. If w is a count of operations to perform, c = Common->chunk, and m = Common->nthreads_max, then a parallel region will use $\max(1, \min(\lfloor w/c \rfloor, m))$ threads. These parameters can be revised by the user application at run time.

9 Using CHOLMOD with GPU acceleration

Starting with CHOLMOD v2.0.0, it is possible to accelerate the numerical factorization phase of CHOLMOD using NVIDIA GPUs. Due to the large computational capability of the GPUs, enabling this capability can result in significant performance improvements. Similar to CPU processing, the GPU is better able to accelerate the dense math associated with larger supernodes. Hence the GPU will provide more significant performance improvements for larger matrices that have more, larger supernodes.

In CHOLMOD v2.3.0 this GPU capability has been improved to provide a significant increase in performance and the interface has been expanded to make the use of GPUs more flexible. CHOLMOD can take advantage of a single NVIDIA GPU that supports CUDA and has at least 64MB of memory. (But substantially more memory, typically about 3 GB, is recommended for best performance.)

Only the long integer version of CHOLMOD can leverage GPU acceleration.

9.1 Compiling CHOLMOD with GPU support

In order to support GPU processing, CHOLMOD must be compiled with the preprocessor macro ENABLE_CUDA defined. It is enabled by default but can be disabled by setting this to false when using cmake.

9.2 Enabling GPU acceleration in CHOLMOD

Even if compiled with GPU support, in CHOLMOD v.2.3.0, GPU processing is not enabled by default and must be specifically requested. There are two ways to do this, either in the code calling CHOLMOD or using environment variables.

The code author can specify the use of GPU processing with the `Common->useGPU` variable. If this is set to `1`, CHOLMOD will attempt to use the GPU. If this is set to `0` the use of the GPU will be prohibited. If this is set to `-1`, which is the default case, then the environment variables (following paragraph) will be queried to determine if the GPU is to be used. Note that the default value of `-1` is set when `cholmod_start(Common)` is called, so the code author must set `Common->useGPU` after calling `cholmod_start`.

Alternatively, or if it is not possible to modify the code calling CHOLMOD, GPU processing can invoked using the `CHOLMOD_USE_GPU` environment variable. This makes it possible for any CHOLMOD user to invoke GPU processing even if the author of the calling program did not consider this. The interpretation of the environment variable `CHOLMOD_USE_GPU` is that if the string evaluates to an integer other than zero, GPU processing will be enabled. Note that the setting of `Common->useGPU` takes precedence and the environment variable `CHOLMOD_USE_GPU` will only be queried if `Common->useGPU = -1`.

Note that in either case, if GPU processing is requested, but there is no GPU present, CHOLMOD will continue using the CPU only. Consequently it is always safe to request GPU processing.

9.3 Adjustable parameters

There are a number of parameters that have been added to CHOLMOD to control GPU processing. All of these have appropriate defaults such that GPU processing can be used without any modification. However, for any particular combination of CPU/GPU, better performance might be obtained by adjusting these parameters.

From `t_cholmod_gpu.c`

`CHOLMOD_ND_ROW_LIMIT` : Minimum number of rows required in a descendant supernode to be eligible for GPU processing during supernode assembly

`CHOLMOD_ND_COL_LIMIT` : Minimum number of columns in a descendant supernode to be eligible for GPU processing during supernode assembly

`CHOLMOD_POTRF_LIMIT` : Minimum number of columns in a supernode to be eligible for `POTRF` and `TRSM` processing on the GPU

`CHOLMOD_GPU_SKIP` : Number of small descendant supernodes to assembled on the CPU before querying if the GPU is needs more descendant supernodes queued

From `cholmod_core.h`

`CHOLMOD_HOST_SUPERNODE_BUFFERS` : Number of buffers in which to queue descendant supernodes for GPU processing

Programatically

`Common->maxGpuMemBytes` : Specifies the maximum amount of memory, in bytes, that CHOLMOD can allocate on the GPU. If this parameter is not set, CHOLMOD will allocate as much GPU memory as possible. Hence, the purpose of this parameter is to restrict CHOLMOD's GPU memory use so that CHOLMOD can be used simultaneously with other codes that also use GPU acceleration and require some amount of GPU memory. If the specified amount of GPU memory is not allocatable, CHOLMOD will allocate the available memory and continue.

`Common->maxGpuMemFraction` : Entirely similar to `Common->maxGpuMemBytes` but with the memory specified as a fraction of total GPU memory. Note that if both `maxGpuMemBytes` and `maxGpuMemFraction` are specified, whichever results in the minimum amount of memory will be used.

Environment variables

`CHOLMOD_GPU_MEM_BYTES` : Environment variable with a meaning equivalent to `Common->maxGpuMemBytes`. This will only be queried if `Common->useGPU = -1`.

`CHOLMOD_GPU_MEM_FRACTION` : Environment variable with a meaning equivalent to `Common->maxGpuMemFra` This will only be queried if `Common->useGPU = -1`.

10 Integer and floating-point types, and notation used

CHOLMOD supports both int and long integers. CHOLMOD routines with the prefix cholmod_ use int integers, cholmod_l_ routines use long. All floating-point values are double.

The long integer is redefinable, via SuiteSparse_config.h. That file defines a C preprocessor token SuiteSparse_long which is long on all systems except for Windows-64, in which case it is defined as __int64. The intent is that with suitable compile-time switches, int is a 32-bit integer and SuiteSparse_long is a 64-bit integer. The term long is used to describe the latter integer throughout this document (except in the prototypes).

Two kinds of complex matrices are supported: complex and zomplex. A complex matrix is held in a manner that is compatible with the Fortran and ANSI C99 complex data type. A complex array of size n is a double array x of size 2*n, with the real and imaginary parts interleaved (the real part comes first, as a double, followed the imaginary part, also as a double. Thus, the real part of the kth entry is x[2*k] and the imaginary part is x[2*k+1].

A zomplex matrix of size n stores its real part in one double array of size n called x and its imaginary part in another double array of size n called z (thus the name "zomplex"). This also how MATLAB stores its complex matrices. The real part of the kth entry is x[k] and the imaginary part is z[k].

Unlike UMFPACK, the same routine name in CHOLMOD is used for pattern-only, real, complex, and zomplex matrices. For example, the statement

```
C = cholmod_copy_sparse (A, &Common) ;
```

creates a copy of a pattern, real, complex, or zomplex sparse matrix A. The xtype (pattern, real, complex, or zomplex) of the resulting sparse matrix C is the same as A (a pattern-only sparse matrix contains no floating-point values). In the above case, C and A use int integers. For long integers, the statement would become:

```
C = cholmod_l_copy_sparse (A, &Common) ;
```

The last parameter of all CHOLMOD routines is always &Common, a pointer to the cholmod_common object, which contains parameters, statistics, and workspace used throughout CHOLMOD.

The xtype of a CHOLMOD object (sparse matrix, triplet matrix, dense matrix, or factorization) determines whether it is pattern-only, real, complex, or zomplex.

The names of the int versions are primarily used in this document. To obtain the name of the long version of the same routine, simply replace cholmod_ with cholmod_l_.

MATLAB matrix notation is used throughout this document and in the comments in the CHOLMOD code itself. If you are not familiar with MATLAB, here is a short introduction to the notation, and a few minor variations used in CHOLMOD:

- C=A+B and C=A*B, respectively are a matrix add and multiply if both A and B are matrices of appropriate size. If A is a scalar, then it is added to or multiplied with every entry in B.
- a:b where a and b are integers refers to the sequence a, a+1, ... b.
- [A B] and [A,B] are the horizontal concatenation of A and B.
- [A;B] is the vertical concatenation of A and B.

- `A(i,j)` can refer either to a scalar or a submatrix. For example:

`A(1,1)`	a scalar.
`A(:,j)`	column `j` of `A`.
`A(i,:)`	row `i` of `A`.
`A([1 2], [1 2])`	a 2-by-2 matrix containing the 2-by-2 leading minor of `A`.

  If `p` is a permutation of `1:n`, and `A` is `n`-by-`n`, then `A(p,p)` corresponds to the permuted matrix $\mathbf{PAP}^\mathsf{T}$.

- `tril(A)` is the lower triangular part of `A`, including the diagonal.
- `tril(A,k)` is the lower triangular part of `A`, including entries on and below the kth diagonal.
- `triu(A)` is the upper triangular part of `A`, including the diagonal.
- `triu(A,k)` is the upper triangular part of `A`, including entries on and above the kth diagonal.
- `size(A)` returns the dimensions of `A`.
- `find(x)` if `x` is a vector returns a list of indices `i` for which `x(i)` is nonzero.
- `A'` is the transpose of `A` if `A` is real, or the complex conjugate transpose if `A` is complex.
- `A.'` is the array transpose of `A`.
- `diag(A)` is the diagonal of `A` if `A` is a matrix.
- `C=diag(s)` is a diagonal matrix if `s` is a vector, with the values of `s` on the diagonal of `C`.
- `S=spones(A)` returns a binary matrix `S` with the same nonzero pattern of `A`.
- `nnz(A)` is the number of nonzero entries in `A`.

Variations to MATLAB notation used in this document:

- CHOLMOD uses 0-based notation (the first entry in the matrix is `A(0,0)`). MATLAB is 1-based. The context is usually clear.
- `I` is the identity matrix.
- `A(:,f)`, where `f` is a set of columns, is interpreted differently in CHOLMOD, but just for the set named `f`. See `cholmod_transpose_unsym` for details.

11 The CHOLMOD Modules, objects, and functions

CHOLMOD contains a total of 133 `int`-based routines (and the same number of `long` routines), divided into a set of inter-related Modules. Each Module contains a set of related functions. The functions are divided into two types: Primary and Secondary, to reflect how a user will typically use CHOLMOD. Most users will find the Primary routines to be sufficient to use CHOLMOD in their programs. Each Module exists as a sub-directory (a folder for Windows users) within the CHOLMOD directory (or folder).

There are seven Modules that provide user-callable routines for CHOLMOD.

1. `Utility`: basic data structures and definitions
2. `Check`: prints/checks each of CHOLMOD's objects
3. `Cholesky`: sparse Cholesky factorization
4. `Modify`: sparse Cholesky update/downdate and row-add/row-delete
5. `MatrixOps`: sparse matrix operators (add, multiply, norm, scale)
6. `Supernodal`: supernodal sparse Cholesky factorization
7. `Partition`: graph-partitioning-based orderings, which uses a slightly modified copy of METIS 5.1.0 in the `SuiteSparse_metis` folder.

Additional directories provide support functions and documentation:

1. `Include`: include files for CHOLMOD and programs that use CHOLMOD
2. `Demo`: simple programs that illustrate the use of CHOLMOD
3. `Doc`: documentation (including this document)
4. `MATLAB`: CHOLMOD's interface to MATLAB
5. `Tcov`: an exhaustive test coverage (requires Linux or Solaris)
6. `cmake_modules`: how other packages can find CHOLMOD when using cmake.
7. `Config`: a folder containing the input files to create the `cholmod.h` include file, via cmake.

11.1 Utility Module: basic data structures and definitions

CHOLMOD includes five basic objects, defined in the Utility Module. The Utility Module provides basic operations for these objects and is required by all six other CHOLMOD library Modules:

11.1.1 cholmod_common: parameters, statistics, and workspace

You must call cholmod_start before calling any other CHOLMOD routine, and you must call cholmod_finish as your last call to CHOLMOD (with the exception of cholmod_print_common and cholmod_check_common in the Check Module). Once the cholmod_common object is initialized, the user may modify CHOLMOD's parameters held in this object, and obtain statistics on CHOLMOD's activity.

Primary routines for the cholmod_common object:

- cholmod_start: the first call to CHOLMOD.
- cholmod_finish: the last call to CHOLMOD (frees workspace in the cholmod_common object).

Secondary routines for the cholmod_common object:

- cholmod_defaults: restores default parameters
- cholmod_maxrank: determine maximum rank for update/downdate.
- cholmod_allocate_work: allocate workspace.
- cholmod_free_work: free workspace.
- cholmod_clear_flag: clear Flag array.
- cholmod_error: called when CHOLMOD encounters and error.
- cholmod_dbound: bounds the diagonal of **L** or **D**.
- cholmod_hypot: compute sqrt(x*x+y*y) accurately.
- cholmod_divcomplex: complex divide.

11.1.2 cholmod_sparse: a sparse matrix in compressed column form

A sparse matrix A is held in compressed column form. In the basic type ("packed," which corresponds to how MATLAB stores its sparse matrices), and nrow-by-ncol matrix with nzmax entries is held in three arrays: p of size ncol+1, i of size nzmax, and x of size nzmax. Row indices of nonzero entries in column j are held in i [p[j] ... p[j+1]-1], and their corresponding numerical values are held in x [p[j] ... p[j+1]-1]. The first column starts at location zero (p[0]=0). There may be no duplicate entries. Row indices in each column may be sorted or unsorted (the A->sorted flag must be false if the columns are unsorted). The A->stype determines the storage mode: 0 if the matrix is unsymmetric, 1 if the matrix is symmetric with just the upper triangular part stored, and -1 if the matrix is symmetric with just the lower triangular part stored.

In "unpacked" form, an additional array nz of size ncol is used. The end of column j in i and x is given by p[j]+nz[j]. Columns not need be in any particular order (p[0] need not be zero), and there may be gaps between the columns.

Primary routines for the cholmod_sparse object:

- cholmod_allocate_sparse: allocate a sparse matrix
- cholmod_free_sparse: free a sparse matrix

Secondary routines for the cholmod_sparse object:

- cholmod_reallocate_sparse: change the size (number of entries) of a sparse matrix.
- cholmod_nnz: number of nonzeros in a sparse matrix.
- cholmod_speye: sparse identity matrix.
- cholmod_spzeros: sparse zero matrix.
- cholmod_transpose: transpose a sparse matrix.
- cholmod_ptranspose: transpose/permute a sparse matrix.
- cholmod_transpose_unsym: transpose/permute an unsymmetric sparse matrix.
- cholmod_transpose_sym: transpose/permute a symmetric sparse matrix.
- cholmod_sort: sort row indices in each column of a sparse matrix.
- cholmod_band: extract a band of a sparse matrix.
- cholmod_band_inplace: remove entries not with a band.
- cholmod_aat: C = A*A'.
- cholmod_copy_sparse: C = A, create an exact copy of a sparse matrix.
- cholmod_copy: C = A, with possible change of stype.
- cholmod_add: C = alpha*A + beta*B.
- cholmod_sparse_xtype: change the xtype of a sparse matrix.

11.1.3 `cholmod_factor`: a symbolic or numeric factorization

A factor can be in $\mathbf{LL}^\mathsf{T}$ or $\mathbf{LDL}^\mathsf{T}$ form, and either supernodal or simplicial form. In simplicial form, this is very much like a packed or unpacked `cholmod_sparse` matrix. In supernodal form, adjacent columns with similar nonzero pattern are stored as a single block (a supernode).

Primary routine for the `cholmod_factor` object:

- `cholmod_free_factor`: free a factor

Secondary routines for the `cholmod_factor` object:

- `cholmod_allocate_factor`: allocate a factor. You will normally use `cholmod_analyze` to create a factor.
- `cholmod_reallocate_factor`: change the number of entries in a factor.
- `cholmod_change_factor`: change the type of a factor ($\mathbf{LDL}^\mathsf{T}$ to $\mathbf{LL}^\mathsf{T}$, supernodal to simplicial, etc.).
- `cholmod_pack_factor`: pack the columns of a factor.
- `cholmod_reallocate_column`: resize a single column of a factor.
- `cholmod_factor_to_sparse`: create a sparse matrix copy of a factor.
- `cholmod_copy_factor`: create a copy of a factor.
- `cholmod_factor_xtype`: change the xtype of a factor.

11.1.4 `cholmod_dense`: a dense matrix

This consists of a dense array of numerical values and its dimensions.

Primary routines for the `cholmod_dense` object:

- `cholmod_allocate_dense`: allocate a dense matrix.
- `cholmod_free_dense`: free a dense matrix.

Secondary routines for the `cholmod_dense` object:

- `cholmod_zeros`: allocate a dense matrix of all zeros.
- `cholmod_ones`: allocate a dense matrix of all ones.
- `cholmod_eye`: allocate a dense identity matrix .
- `cholmod_sparse_to_dense`: create a dense matrix copy of a sparse matrix.
- `cholmod_dense_to_sparse`: create a sparse matrix copy of a dense matrix.
- `cholmod_copy_dense`: create a copy of a dense matrix.
- `cholmod_copy_dense2`: copy a dense matrix (pre-allocated).
- `cholmod_dense_xtype`: change the `xtype` of a dense matrix.

11.1.5 `cholmod_triplet`: a sparse matrix in "triplet" form

The `cholmod_sparse` matrix is the basic sparse matrix used in CHOLMOD, but it can be difficult for the user to construct. It also does not easily support the inclusion of new entries in the matrix. The `cholmod_triplet` matrix is provided to address these issues. A sparse matrix in triplet form consists of three arrays of size `nzmax`: `i`, `j`, and `x`, and a `z` array for the zomplex case.

Primary routines for the `cholmod_triplet` object:

- `cholmod_allocate_triplet`: allocate a triplet matrix.
- `cholmod_free_triplet`: free a triplet matrix.
- `cholmod_triplet_to_sparse`: create a sparse matrix copy of a triplet matrix.

Secondary routines for the `cholmod_triplet` object:

- `cholmod_reallocate_triplet`: change the number of entries in a triplet matrix.
- `cholmod_sparse_to_triplet`: create a triplet matrix copy of a sparse matrix.
- `cholmod_copy_triplet`: create a copy of a triplet matrix.
- `cholmod_triplet_xtype`: change the `xtype` of a triplet matrix.

11.1.6 Memory management routines

By default, CHOLMOD uses the ANSI C `malloc`, `free`, `calloc`, and `realloc` routines. You may use different routines by modifying function pointers in the `cholmod_common` object.

Primary routines:

- `cholmod_malloc`: `malloc` wrapper.
- `cholmod_free`: `free` wrapper.

Secondary routines:

- `cholmod_calloc`: `calloc` wrapper.
- `cholmod_realloc`: `realloc` wrapper.
- `cholmod_realloc_multiple`: `realloc` wrapper for multiple objects.

11.1.7 `cholmod_version`: Version control

The `cholmod_version` function returns the current version of CHOLMOD.

11.2 Check Module: print/check the CHOLMOD objects

The Check Module contains routines that check and print the five basic objects in CHOLMOD, and three kinds of integer vectors (a set, a permutation, and a tree). It also provides a routine to read a sparse matrix from a file in Matrix Market format (http://www.nist.gov/MatrixMarket). Requires the Utility Module.

Primary routines:

- cholmod_print_common: print the cholmod_common object, including statistics on CHOLMOD's behavior (fill-in, flop count, ordering methods used, and so on).
- cholmod_write_sparse: write a sparse matrix to a file in Matrix Market format.
- cholmod_write_dense: write a sparse matrix to a file in Matrix Market format.
- cholmod_read_matrix: read a sparse or dense matrix from a file in Matrix Market format.

Secondary routines:

- cholmod_check_common: check the cholmod_common object
- cholmod_check_sparse: check a sparse matrix
- cholmod_print_sparse: print a sparse matrix
- cholmod_check_dense: check a dense matrix
- cholmod_print_dense: print a dense matrix
- cholmod_check_factor: check a Cholesky factorization
- cholmod_print_factor: print a Cholesky factorization
- cholmod_check_triplet: check a triplet matrix
- cholmod_print_triplet: print a triplet matrix
- cholmod_check_subset: check a subset (integer vector in given range)
- cholmod_print_subset: print a subset (integer vector in given range)
- cholmod_check_perm: check a permutation (an integer vector)
- cholmod_print_perm: print a permutation (an integer vector)
- cholmod_check_parent: check an elimination tree (an integer vector)
- cholmod_print_parent: print an elimination tree (an integer vector)
- cholmod_read_triplet: read a triplet matrix from a file
- cholmod_read_sparse: read a sparse matrix from a file
- cholmod_read_dense: read a dense matrix from a file

11.3 `Cholesky` Module: sparse Cholesky factorization

The primary routines are all that a user requires to order, analyze, and factorize a sparse symmetric positive definite matrix $\mathbf{A}$ (or $\mathbf{A}\mathbf{A}^\mathsf{T}$), and to solve $\mathbf{Ax} = \mathbf{b}$ (or $\mathbf{A}\mathbf{A}^\mathsf{T}\mathbf{x} = \mathbf{b}$). The primary routines rely on the secondary routines, the `Utility` Module, and the AMD and COLAMD packages. They make optional use of the `Supernodal` and `Partition` Modules, the METIS package, the CAMD package, and the CCOLAMD package. The `Cholesky` Module is required by the `Partition` Module.

Primary routines:

- `cholmod_analyze`: order and analyze (simplicial or supernodal).
- `cholmod_factorize`: simplicial or supernodal Cholesky factorization.
- `cholmod_solve`: solve a linear system (simplicial or supernodal, dense $\mathbf{x}$ and $\mathbf{b}$).
- `cholmod_spsolve`: solve a linear system (simplicial or supernodal, sparse $\mathbf{x}$ and $\mathbf{b}$).

Secondary routines:

- `cholmod_analyze_p`: analyze, with user-provided permutation or $\mathbf{f}$ set.
- `cholmod_factorize_p`: factorize, with user-provided permutation or $\mathbf{f}$.
- `cholmod_analyze_ordering`: analyze a permutation
- `cholmod_solve2`: solve a linear system, reusing workspace.
- `cholmod_etree`: find the elimination tree.
- `cholmod_rowcolcounts`: compute the row/column counts of $\mathbf{L}$.
- `cholmod_amd`: order using AMD.
- `cholmod_colamd`: order using COLAMD.
- `cholmod_rowfac`: incremental simplicial factorization.
- `cholmod_row_subtree`: find the nonzero pattern of a row of $\mathbf{L}$.
- `cholmod_row_lsubtree`: find the nonzero pattern of a row of $\mathbf{L}$.
- `cholmod_row_lsubtree`: find the nonzero pattern of $\mathbf{L}^{-1}b$.
- `cholmod_resymbol`: recompute the symbolic pattern of $\mathbf{L}$.
- `cholmod_resymbol_noperm`: recompute the symbolic pattern of $\mathbf{L}$, no permutation.
- `cholmod_postorder`: postorder a tree.
- `cholmod_rcond`: compute the reciprocal condition number estimate.
- `cholmod_rowfac_mask`: for use in LPDASA only.

11.4 `Modify` Module: update/downdate a sparse Cholesky factorization

The `Modify` Module contains sparse Cholesky modification routines: update, downdate, row-add, and row-delete. It can also modify a corresponding solution to $\mathbf{Lx} = \mathbf{b}$ when L is modified. This module is most useful when applied on a Cholesky factorization computed by the `Cholesky` module, but it does not actually require the `Cholesky` module. The `Utility` module can create an identity Cholesky factorization ($\mathbf{LDL^T}$ where $\mathbf{L} = \mathbf{D} = \mathbf{I}$) that can then be modified by these routines. Requires the `Utility` module. Not required by any other CHOLMOD Module.

Primary routine:

- `cholmod_updown`: multiple rank update/downdate

Secondary routines:

- `cholmod_updown_solve`: update/downdate, and modify solution to $\mathbf{Lx} = \mathbf{b}$
- `cholmod_updown_mark`: update/downdate, and modify solution to partial $\mathbf{Lx} = \mathbf{b}$
- `cholmod_updown_mask`: for use in LPDASA only.
- `cholmod_rowadd`: add a row to an $\mathbf{LDL^T}$ factorization
- `cholmod_rowadd_solve`: add a row, and update solution to $\mathbf{Lx} = \mathbf{b}$
- `cholmod_rowadd_mark`: add a row, and update solution to partial $\mathbf{Lx} = \mathbf{b}$
- `cholmod_rowdel`: delete a row from an $\mathbf{LDL^T}$ factorization
- `cholmod_rowdel_solve`: delete a row, and downdate $\mathbf{Lx} = \mathbf{b}$
- `cholmod_rowdel_mark`: delete a row, and downdate solution to partial $\mathbf{Lx} = \mathbf{b}$

11.5 `MatrixOps` Module: basic sparse matrix operations

The `MatrixOps` Module provides basic operations on sparse and dense matrices. Requires the `Utility` module. Not required by any other CHOLMOD module. In the descriptions below, `A`, `B`, and `C:` are sparse matrices (`cholmod_sparse`), `X` and `Y` are dense matrices (`cholmod_dense`), `s` is a scalar or vector, and `alpha` `beta` are scalars.

- `cholmod_drop`: drop entries from A with absolute value $\geq$ a given tolerance.
- `cholmod_norm_dense`: `s = norm (X)`, 1-norm, infinity-norm, or 2-norm
- `cholmod_norm_sparse`: `s = norm (A)`, 1-norm or infinity-norm
- `cholmod_horzcat`: `C = [A,B]`
- `cholmod_scale`: `A = diag(s)*A`, `A*diag(s)`, `s*A` or `diag(s)*A*diag(s)`.
- `cholmod_sdmult`: `Y = alpha*(A*X) + beta*Y` or `alpha*(A'*X) + beta*Y`.
- `cholmod_ssmult`: `C = A*B`

- `cholmod_submatrix`: `C = A (i,j)`, where `i` and `j` are arbitrary integer vectors.
- `cholmod_vertcat`: `C = [A ; B]`.
- `cholmod_symmetry`: determine symmetry of a matrix.

11.6 `Supernodal` Module: supernodal sparse Cholesky factorization

The `Supernodal` Module performs supernodal analysis, factorization, and solve. The simplest way to use these routines is via the `Cholesky` Module. This Module does not provide any fill-reducing orderings. It normally operates on matrices ordered by the `Cholesky` Module. It does not require the `Cholesky` Module itself, however. Requires the `Utility` Module, and two external packages: LAPACK and the BLAS. Optionally used by the `Cholesky` Module. All are secondary routines since these functions are more easily used via the `Cholesky` Module.

Secondary routines:

- `cholmod_super_symbolic`: supernodal symbolic analysis
- `cholmod_super_numeric`: supernodal numeric factorization
- `cholmod_super_lsolve`: supernodal $\mathbf{Lx} = \mathbf{b}$ solve
- `cholmod_super_ltsolve`: supernodal $\mathbf{L}^\mathsf{T}\mathbf{x} = \mathbf{b}$ solve

11.7 `Partition` Module: graph-partitioning-based orderings

The `Partition` Module provides graph partitioning and graph-partition-based orderings. It includes an interface to CAMD, CCOLAMD, and CSYMAMD, constrained minimum degree ordering methods which order a matrix following constraints determined via nested dissection. Requires the `Utility` and `Cholesky` Modules, and two packages: `METIS 5.1.0`, CAMD, and CCOLAMD. Optionally used by the `Cholesky` Module. All are secondary routines since these are more easily used by the `Cholesky` Module.

Note that METIS does not have a version that uses `long` integers. If you try to use these routines (except the CAMD, CCOLAMD, and CSYMAMD interfaces) on a matrix that is too large, an error code will be returned.

Secondary routines:

- `cholmod_nested_dissection`: CHOLMOD nested dissection ordering
- `cholmod_metis`: METIS nested dissection ordering (`METIS_NodeND`)
- `cholmod_camd`: interface to CAMD ordering
- `cholmod_ccolamd`: interface to CCOLAMD ordering
- `cholmod_csymamd`: interface to CSYMAMD ordering
- `cholmod_bisect`: graph partitioner (currently based on METIS)
- `cholmod_metis_bisector`: direct interface to `METIS_NodeComputeSeparator`.
- `cholmod_collapse_septree`: pruned a separator tree from `cholmod_nested_dissection`.

12 CHOLMOD naming convention, parameters, and return values

All routine names, data types, and CHOLMOD library files use the `cholmod_` prefix. All macros and other `#define` statements visible to the user program use the `CHOLMOD` prefix. The `cholmod.h` file must be included in user programs that use CHOLMOD:

```
#include "cholmod.h"
```

All CHOLMOD routines (in all modules) use the following protocol for return values:

- `int`: `TRUE` (1) if successful, or `FALSE` (0) otherwise. (exception: `cholmod_divcomplex`).
- `long`: a value ≥ 0 if successful, or -1 otherwise.
- `double`: a value ≥ 0 if successful, or -1 otherwise.
- `size_t`: a value > 0 if successful, or 0 otherwise.
- `void *`: a non-`NULL` pointer to newly allocated memory if successful, or `NULL` otherwise.
- `cholmod_sparse *`: a non-`NULL` pointer to a newly allocated sparse matrix if successful, or `NULL` otherwise.
- `cholmod_factor *`: a non-`NULL` pointer to a newly allocated factor if successful, or `NULL` otherwise.
- `cholmod_triplet *`: a non-`NULL` pointer to a newly allocated triplet matrix if successful, or `NULL` otherwise.
- `cholmod_dense *`: a non-`NULL` pointer to a newly allocated dense matrix if successful, or `NULL` otherwise.

`TRUE` and `FALSE` are not defined in `cholmod.h`, since they may conflict with the user program. A routine that described here returning `TRUE` or `FALSE` returns 1 or 0, respectively. Any `TRUE`/`FALSE` parameter is true if nonzero, false if zero.
Input, output, and input/output parameters:

- Input parameters appear first in the parameter lists of all CHOLMOD routines. They are not modified by CHOLMOD.
- Input/output parameters (except for `Common`) appear next. They must be defined on input, and are modified on output.
- Output parameters are listed next. If they are pointers, they must point to allocated space on input, but their contents are not defined on input.
- Workspace parameters appear next. They are used in only two routines in the Supernodal module.

- The `cholmod_common *Common` parameter always appears as the last parameter (with two exceptions: `cholmod_hypot` and `cholmod_divcomplex`). It is always an input/output parameter.

A floating-point scalar is passed to CHOLMOD as a pointer to a `double` array of size two. The first entry in this array is the real part of the scalar, and the second entry is the imaginary part. The imaginary part is only accessed if the other inputs are complex or zomplex. In some cases the imaginary part is always ignored (`cholmod_factor_p`, for example).

13 Utility Module: cholmod_common object

13.1 Constant definitions

```
// itype: integer sizes
// The itype is held in the Common object and must match the method used.
#define CHOLMOD_INT  0    /* int32, for cholmod_* methods (no _l_) */
#define CHOLMOD_LONG 2    /* int64, for cholmod_l_* methods       */

// dtype: floating point sizes (double or float)
// The dtype of all parameters for all CHOLMOD routines must match.
// NOTE: CHOLMOD_SINGLE is still under development.
#define CHOLMOD_DOUBLE 0  /* matrix or factorization is double precision */
#define CHOLMOD_SINGLE 4  /* matrix or factorization is single precision */

// xtype: pattern, real, complex, or zomplex
#define CHOLMOD_PATTERN 0 /* no numerical values                          */
#define CHOLMOD_REAL    1 /* real (double or single), not complex         */
#define CHOLMOD_COMPLEX 2 /* complex (double or single), interleaved      */
#define CHOLMOD_ZOMPLEX 3 /* complex (double or single), with real and imag */
                          /* parts held in different arrays               */

// xdtype is (xtype + dtype), which combines the two type parameters into
// a single number handling all 8 cases:
//
//  (0) CHOLMOD_DOUBLE + CHOLMOD_PATTERN    a pattern-only matrix
//  (1) CHOLMOD_DOUBLE + CHOLMOD_REAL       a double real matrix
//  (2) CHOLMOD_DOUBLE + CHOLMOD_COMPLEX    a double complex matrix
//  (3) CHOLMOD_DOUBLE + CHOLMOD_ZOMPLEX    a double zomplex matrix
//  (4) CHOLMOD_SINGLE + CHOLMOD_PATTERN    a pattern-only matrix
//  (5) CHOLMOD_SINGLE + CHOLMOD_REAL       a float real matrix
//  (6) CHOLMOD_SINGLE + CHOLMOD_COMPLEX    a float complex matrix
//  (7) CHOLMOD_SINGLE + CHOLMOD_ZOMPLEX    a float zomplex matrix

// max # of ordering methods in Common
#define CHOLMOD_MAXMETHODS 9

// Common->status for error handling: 0 is ok, negative is a fatal error,
// and positive is a warning
#define CHOLMOD_OK            (0)
#define CHOLMOD_NOT_INSTALLED (-1) /* module not installed                */
#define CHOLMOD_OUT_OF_MEMORY (-2) /* malloc, calloc, or realloc failed   */
#define CHOLMOD_TOO_LARGE     (-3) /* integer overflow                    */
#define CHOLMOD_INVALID       (-4) /* input invalid                       */
#define CHOLMOD_GPU_PROBLEM   (-5) /* CUDA error                          */
#define CHOLMOD_NOT_POSDEF    (1)  /* matrix not positive definite        */
#define CHOLMOD_DSMALL        (2)  /* diagonal entry very small           */

// ordering method
#define CHOLMOD_NATURAL     0 /* no preordering                           */
#define CHOLMOD_GIVEN       1 /* user-provided permutation                */
#define CHOLMOD_AMD         2 /* AMD: approximate minimum degree          */
#define CHOLMOD_METIS       3 /* METIS: mested dissection                 */
#define CHOLMOD_NESDIS      4 /* CHOLMOD's nested dissection              */
#define CHOLMOD_COLAMD      5 /* AMD for A, COLAMD for AA' or A'A         */
```

```
#define CHOLMOD_POSTORDERED 6 /* natural then postordered              */

// supernodal strategy
#define CHOLMOD_SIMPLICIAL 0 /* always use simplicial method           */
#define CHOLMOD_AUTO       1 /* auto select simplicial vs supernodal   */
#define CHOLMOD_SUPERNODAL 2 /* always use supernoda method            */
```

Purpose: These definitions are used within the `cholmod_common` object, called `Common` both here and throughout the code.

13.2 `cholmod_common`: parameters, statistics, and workspace

```
typedef struct cholmod_common_struct
{

    //------------------------------------------------------------------------
    // primary parameters for factorization and update/downdate
    //------------------------------------------------------------------------

    double dbound ; // Bounds the diagonal entries of D for LDL'
        // factorization and update/downdate/rowadd.  Entries outside this
        // bound are replaced with dbound.  Default: 0.
        // dbound is used for double precision factorization only.
        // See sbound for single precision factorization.

    double grow0 ;      // default: 1.2
    double grow1 ;      // default: 1.2
    size_t grow2 ;      // default: 5
        // Initial space for simplicial factorization is max(grow0,1) times the
        // required space.  If space is exhausted, L is grown by
        // max(grow0,1.2) times the required space.  grow1 and grow2 control
        // how each column of L can grow in an update/downdate; if space runs
        // out, then grow1*(required space) + grow2 is allocated.

    size_t maxrank ;    // maximum rank for update/downdate.  Valid values are
        // 2, 4, and 8.  Default is 8.  If a larger update/downdate is done,
        // it is done in steps of maxrank.

    double supernodal_switch ;  // default: 40
    int supernodal ;            // default: CHOLMOD_AUTO.
        // Controls supernodal vs simplicial factorization.  If
        // Common->supernodal is CHOLMOD_SIMPLICIAL, a simplicial factorization
        // is always done; if CHOLMOD_SUPERNODAL, a supernodal factorization is
        // always done.  If CHOLMOD_AUTO, then a simplicial factorization is
        // down if flops/nnz(L) < Common->supernodal_switch.

    int final_asis ;    // if true, other final_* parameters are ignored,
        // except for final_pack and the factors are left as-is when done.
        // Default: true.

    int final_super ;   // if true, leave factor in supernodal form.
        // if false, convert to simplicial.  Default: true.
```

```
int final_ll ;      // if true, simplicial factors are converted to LL',
    // otherwise left as LDL.  Default: false.

int final_pack ;    // if true, the factorize are allocated with exactly
    // the space required.  Set this to false if you expect future
    // updates/downdates (giving a little extra space for future growth),
    // Default: true.

int final_monotonic ;   // if true, columns are sorted when done, by
    // ascending row index.  Default: true.

int final_resymbol ;    // if true, a supernodal factorization converted
    // to simplicial is reanalyzed, to remove zeros added for relaxed
    // amalgamation.  Default: false.

double zrelax [3] ; size_t nrelax [3] ;
    // The zrelax and nrelax parameters control relaxed supernodal
    // amalgamation,  If ns is the # of columns in two adjacent supernodes,
    // and z is the fraction of zeros in the two supernodes if merged, then
    // the two supernodes are merged if any of the 5 following condition
    // are true:
    //
    //      no new zero entries added if the two supernodes are merged
    //      (ns <= nrelax [0])
    //      (ns <= nrelax [1] && z < zrelax [0])
    //      (ns <= nrelax [2] && z < zrelax [1])
    //      (z < zrelax [2])
    //
    // With the defaults, the rules become:
    //
    //      no new zero entries added if the two supernodes are merged
    //      (ns <=  4)
    //      (ns <= 16 && z < 0.8)
    //      (ns <= 48 && z < 0.1)
    //      (z < 0.05)

int prefer_zomplex ;    // if true, and a complex system is solved,
    // X is returned as zomplex (with two arrays, one for the real part
    // and one for the imaginary part).  If false, then X is returned as
    // a single array with interleaved real and imaginary parts.
    // Default: false.

int prefer_upper ;  // if true, then a preference is given for holding
    // a symmetric matrix by just its upper triangular form.  This gives
    // the best performance by the CHOLMOD analysis and factorization
    // methods.  Only used by cholmod_read.  Default: true.

int quick_return_if_not_posdef ;    // if true, a supernodal factorization
    // returns immediately if it finds the matrix is not positive definite.
    // If false, the failed supernode is refactorized, up to but not
    // including the failed column (required by MATLAB).

int prefer_binary ; // if true, cholmod_read_triplet converts a symmetric
    // pattern-only matrix to a real matrix with all values set to 1.
    // if false, diagonal entries A(k,k) are set to one plus the # of
```

```
    // entries in row/column k, and off-diagonals are set to -1.
    // Default: false.

int print ;     // print level.  Default is 3.
int precise ;   // if true, print 16 digits, otherwise 5. Default: false.

int try_catch ; // if true, ignore errors (CHOLMOD is assumed to be inside
    // a try/catch block.  No error messages are printed and the
    // error_handler function is not called.  Default: false.

void (*error_handler) (int status, const char *file, int line,
    const char *message) ;
    // User error handling routine; default is NULL.
    // This function is called if an error occurs, with parameters:
    // status: the Common->status result.
    // file: filename where the error occurred.
    // line: line number where the error occurred.
    // message: a string that describes the error.

//--------------------------------------------------------------------------
// ordering options
//--------------------------------------------------------------------------

// CHOLMOD can try many ordering options and then pick the best result it
// finds.  The default is to use one or two orderings: the user's
// permutation (if given), and AMD.

// Common->nmethods is the number of methods to try.  If the
// Common->method array is left unmodified, the methods are:

// (0) given (skipped if no user permutation)
// (1) amd
// (2) metis
// (3) nesdis with defaults (CHOLMOD's nested dissection, based on METIS)
// (4) natural
// (5) nesdis: stop at subgraphs of 20000 nodes
// (6) nesdis: stop at subgraphs of 4 nodes, do not use CAMD
// (7) nesdis: no pruning on of dense rows/cols
// (8) colamd

// To use all 9 of the above methods, set Common->nmethods to 9.  The
// analysis will take a long time, but that might be worth it if the
// ordering will be reused many many times.

// Common->nmethods and Common->methods can be revised to use a different
// set of orderings.  For example, to use just a single method
// (AMD with a weighted postordering):
//
//      Common->nmethods = 1 ;
//      Common->method [0].ordering = CHOLMOD_AMD ;
//      Common->postorder = TRUE ;
//
//

int nmethods ;  // Number of methods to try, default is 0.
```

```
    // The value of 0 is a special case, and tells CHOLMOD to use the user
    // permutation (if not NULL) and then AMD.  Next, if fl is lnz are the
    // flop counts and number of nonzeros in L as found by AMD, then the
    // this ordering is used if fl/lnz < 500 or lnz/anz < 5, where anz is
    // the number of entries in A.  If this condition fails, METIS is tried
    // as well.
    //
    // Otherwise, if Common->nmethods > 0, then the methods defined by
    // Common->method [0 ... Common->nmethods-1] are used.

int current ;   // The current method being tried in the analysis.
int selected ;  // The selected method: Common->method [Common->selected]

// The Common->method parameter is an array of structs that defines up
// to 9 methods:
struct cholmod_method_struct
{

    //----------------------------------------------------------------------
    // statistics from the ordering
    //----------------------------------------------------------------------

    double lnz ;    // number of nonzeros in L
    double fl ;     // Cholesky flop count for this ordering (each
        // multiply and each add counted once (doesn't count complex
        // flops).

    //----------------------------------------------------------------------
    // ordering parameters:
    //----------------------------------------------------------------------

    double prune_dense ;    // dense row/col control.  Default: 10.
        // Rows/cols with more than max (prune_dense*sqrt(n),16) are
        // removed prior to orderingm and placed last.  If negative,
        // only completely dense rows/cols are removed.  Removing these
        // rows/cols with many entries can speed up the ordering, but
        // removing too many can reduce the ordering quality.
        //
        // For AMD, SYMAMD, and CSYMAMD, this is the only dense row/col
        // parameter.  For COLAMD and CCOLAMD, this parameter controls
        // how dense columns are handled.

    double prune_dense2 ;   // dense row control for COLAMD and CCOLAMD.
        // Default -1.  When computing the Cholesky factorization of AA'
        // rows with more than max(prune_dense2*sqrt(n),16) entries
        // are removed prior to ordering.  If negative, only completely
        // dense rows are removed.

    double nd_oksep ;   // for CHOLMOD's nesdis method. Default 1.
        // A node separator with nsep nodes is discarded if
        // nsep >= nd_oksep*n.

    double other_1 [4] ;    // unused, for future expansion

    size_t nd_small ;   // for CHOLMOD's nesdis method. Default 200.
```

```
        // Subgraphs with fewer than nd_small nodes are not partitioned.

    double other_2 [4] ;    // unused, for future expansion

    int aggressive ;    // if true, AMD, COLAMD, SYMAMD, CCOLAMD, and
        // CSYMAMD perform aggresive absorption.  Default: true

    int order_for_lu ;  // Default: false.  If the CHOLMOD analysis/
        // ordering methods are used as an ordering method for an LU
        // factorization, then set this to true.  For use in a Cholesky
        // factorization by CHOLMOD itself, never set this to true.

    int nd_compress ;   // if true, then the graph and subgraphs are
        // compressed before partitioning them in CHOLMOD's nesdis
        // method.  Default: true.

    int nd_camd ;   // if 1, then CHOLMOD's nesdis is followed by
        // CAMD.  If 2: followed by CSYMAMD.  If nd_small is very small,
        // then use 0, which skips CAMD or CSYMAMD.  Default: 1.

    int nd_components ; // CHOLMOD's nesdis can partition a graph and then
        // find that the subgraphs are unconnected.  If true, each of these
        // components is partitioned separately.  If false, the whole
        // subgraph is partitioned.  Default: false.

    int ordering ;  // ordering method to use

    size_t other_3 [4] ;    // unused, for future expansion

}
method [CHOLMOD_MAXMETHODS + 1] ;

int postorder ; // if true, CHOLMOD performs a weighted postordering
    // after its fill-reducing ordering, which improves supernodal
    // amalgamation.  Has no effect on flop count or nnz(L).
    // Default: true.

int default_nesdis ;    //  If false, then the default ordering strategy
    // when Common->nmethods is zero is to try the user's permutation
    // if given, then AMD, and then METIS if the AMD ordering results in
    // a lot of fill-in.  If true, then nesdis is used instead of METIS.
    // Default: false.

//----------------------------------------------------------------------------
// METIS workarounds
//----------------------------------------------------------------------------

// These workarounds were put into place for METIS 4.0.1.  They are safe
// to use with METIS 5.1.0, but they might not longer be necessary.

double metis_memory ;   // default: 0.  If METIS terminates your
    // program when it runs out of memory, try 2, or higher.
double metis_dswitch ;  // default: 0.66
size_t metis_nswitch ;  // default: 3000
    // If a matrix has n > metis_nswitch and a density (nnz(A)/n^2) >
```

```
    // metis_dswitch, then METIS is not used.

//------------------------------------------------------------------------
// workspace
//------------------------------------------------------------------------

// This workspace is kept in the CHOLMOD Common object.  cholmod_start
// sets these arrays to NULL, and cholmod_finish frees them.

size_t nrow ;       // Flag has size nrow, Head has size nrow+1
int64_t mark ;      // Flag is cleared if Flag [0..nrow-1] < mark.
size_t iworksize ;  // size of Iwork, in Ints (int32 or int64).
                    // This is at most 6*nrow + ncol.
size_t xworkbytes ; // size of Xwork, in bytes.  for update/downdate:
    // maxrank*nrow*sizeof(double or float), 2*nrow*sizeof(double or float)
    // otherwise.  NOTE: in CHOLMOD v4 and earlier, xworkwise was in terms
    // of # of doubles, not # of bytes.

void *Flag ;    // size nrow.  If this is "cleared" then
    // Flag [i] < mark for all i = 0:nrow-1.  Flag is kept cleared between
    // calls to CHOLMOD.

void *Head ;    // size nrow+1.  If Head [i] = EMPTY (-1) then that
    // entry is "cleared".  Head is kept cleared between calls to CHOLMOD.

void *Xwork ;   // a double or float array.  It has size nrow for most
    // routines, or 2*nrow if complex matrices are being handled.
    // It has size 2*nrow for cholmod_rowadd/rowdel, and maxrank*nrow for
    // cholmod_updown, where maxrank is 2, 4, or 8.  Xwork is kept all
    // zero between calls to CHOLMOD.

void *Iwork ;   // size iworksize integers (int32's or int64's).
    // Uninitialized integer workspace, of size at most 6*nrow+ncol.

int itype ;     // cholmod_start (for int32's) sets this to CHOLMOD_INT,
    // and cholmod_l_start sets this to CHOLMOD_LONG.  It defines the
    // integer sizes for th Flag, Head, and Iwork arrays, and also
    // defines the integers for all objects created by CHOLMOD.
    // The itype of the Common object must match the function name
    // and all objects passed to it.

int other_5 ;   // unused: for future expansion

int no_workspace_reallocate ;   // an internal flag, usually false.
    // This is set true to disable any reallocation of the workspace
    // in the Common object.

//------------------------------------------------------------------------
// statistics
//------------------------------------------------------------------------

int status ;    // status code (0: ok, negative: error, pos: warning
double fl ;     // flop count from last analysis
double lnz ;    // nnz(L) from last analysis
double anz ;    // in last analysis: nnz(tril(A)) or nnz(triu(A)) if A
```

```
                // symmetric, or tril(A*A') if A is unsymmetric.
double modfl ;  // flop count from last update/downdate/rowadd/rowdel,
                // not included the flops to revise the solution to Lx=b,
                // if that was performed.

size_t malloc_count ;   // # of malloc'd objects not yet freed
size_t memory_usage ;   // peak memory usage in bytes
size_t memory_inuse ;   // current memory usage in bytes

double nrealloc_col ;   // # of column reallocations
double nrealloc_factor ;// # of factor reallocations due to col. reallocs
double ndbounds_hit ;   // # of times diagonal modified by dbound

double rowfacfl ;       // flop count of cholmod_rowfac
double aatfl ;          // flop count to compute A(:,f)*A(:,f)'

int called_nd ; // true if last analysis used nesdis or METIS.
int blas_ok ;   // true if no integer overflow has occured when trying to
    // call the BLAS.  The typical BLAS library uses 32-bit integers for
    // its input parameters, even on a 64-bit platform.  CHOLMOD uses int64
    // in its cholmod_l_* methods, and these must be typecast to the BLAS
    // integer.  If integer overflow occurs, this is set false.

//----------------------------------------------------------------------
// SuiteSparseQR control parameters and statistics
//----------------------------------------------------------------------

// SPQR uses the CHOLMOD Common object for its control and statistics.
// These parameters are not used by CHOLMOD itself.

// control parameters:
double SPQR_grain ;     // task size is >= max (total flops / grain)
double SPQR_small ;     // task size is >= small
int SPQR_shrink ;       // controls stack realloc method
int SPQR_nthreads ;     // number of TBB threads, 0 = auto

// statistics:
double SPQR_flopcount ;         // flop count for SPQR
double SPQR_analyze_time ;      // analysis time in seconds for SPQR
double SPQR_factorize_time ;    // factorize time in seconds for SPQR
double SPQR_solve_time ;        // backsolve time in seconds
double SPQR_flopcount_bound ;   // upper bound on flop count
double SPQR_tol_used ;          // tolerance used
double SPQR_norm_E_fro ;        // Frobenius norm of dropped entries

//----------------------------------------------------------------------
// Revised for CHOLMOD v5.0
//----------------------------------------------------------------------

// was size 10 in CHOLMOD v4.2; reduced to 8 in CHOLMOD v5:
int64_t SPQR_istat [8] ;        // other statistics

//----------------------------------------------------------------------
// Added for CHOLMOD v5.0
//----------------------------------------------------------------------
```

```
// These terms have been added to the CHOLMOD Common struct for v5.0, and
// on most systems they will total 16 bytes.  The preceding term,
// SPQR_istat, was reduced by 16 bytes, since those last 2 entries were
// unused in CHOLMOD v4.2.  As a result, the Common struct in v5.0 has the
// same size as v4.0, and all entries would normally be in the same offset,
// as well.  This mitigates any changes between v4.0 and v5.0, and may make
// it easier to upgrade from v4 to v5.

double nsbounds_hit ;   // # of times diagonal modified by sbound.  This
                // ought to be int64_t, but ndbounds_hit was double in v4
                // (see above), so nsbounds_hit is made the same type for
                // consistency.
float sbound ;  // Same as dbound, but for single precision factorization.
float other_6 ; // for future expansion

//--------------------------------------------------------------------------
// GPU configuration and statistics
//--------------------------------------------------------------------------

int useGPU ; // 1 if GPU is requested for CHOLMOD
             // 0 if GPU is not requested for CHOLMOD
             // -1 if the use of the GPU is in CHOLMOD controled by the
             // CHOLMOD_USE_GPU environment variable.

size_t maxGpuMemBytes ;     // GPU control for CHOLMOD
double maxGpuMemFraction ;  // GPU control for CHOLMOD

// for SPQR:
size_t gpuMemorySize ;      // Amount of memory in bytes on the GPU
double gpuKernelTime ;      // Time taken by GPU kernels
int64_t gpuFlops ;          // Number of flops performed by the GPU
int gpuNumKernelLaunches ;  // Number of GPU kernel launches

#ifdef SUITESPARSE_CUDA
    // these three types are pointers defined by CUDA:
    #define CHOLMOD_CUBLAS_HANDLE cublasHandle_t
    #define CHOLMOD_CUDASTREAM    cudaStream_t
    #define CHOLMOD_CUDAEVENT     cudaEvent_t
#else
    // they are (void *) if CUDA is not in use:
    #define CHOLMOD_CUBLAS_HANDLE void *
    #define CHOLMOD_CUDASTREAM    void *
    #define CHOLMOD_CUDAEVENT     void *
#endif

CHOLMOD_CUBLAS_HANDLE cublasHandle ;

// a set of streams for general use
CHOLMOD_CUDASTREAM  gpuStream [CHOLMOD_HOST_SUPERNODE_BUFFERS] ;

CHOLMOD_CUDAEVENT   cublasEventPotrf [3] ;
CHOLMOD_CUDAEVENT   updateCKernelsComplete ;
CHOLMOD_CUDAEVENT   updateCBuffersFree [CHOLMOD_HOST_SUPERNODE_BUFFERS] ;
```

```
    void *dev_mempool ; // pointer to single allocation of device memory
    size_t dev_mempool_size ;

    void *host_pinned_mempool ; // pointer to single alloc of pinned mem
    size_t host_pinned_mempool_size ;

    size_t devBuffSize ;
    int    ibuffer ;
    double syrkStart ;          // time syrk started

    // run times of the different parts of CHOLMOD (GPU and CPU):
    double cholmod_cpu_gemm_time ;
    double cholmod_cpu_syrk_time ;
    double cholmod_cpu_trsm_time ;
    double cholmod_cpu_potrf_time ;
    double cholmod_gpu_gemm_time ;
    double cholmod_gpu_syrk_time ;
    double cholmod_gpu_trsm_time ;
    double cholmod_gpu_potrf_time ;
    double cholmod_assemble_time ;
    double cholmod_assemble_time2 ;

    // number of times the BLAS are called on the CPU and the GPU:
    size_t cholmod_cpu_gemm_calls ;
    size_t cholmod_cpu_syrk_calls ;
    size_t cholmod_cpu_trsm_calls ;
    size_t cholmod_cpu_potrf_calls ;
    size_t cholmod_gpu_gemm_calls ;
    size_t cholmod_gpu_syrk_calls ;
    size_t cholmod_gpu_trsm_calls ;
    size_t cholmod_gpu_potrf_calls ;

    double chunk ;      // chunksize for computing # of OpenMP threads to use.
        // Given nwork work to do, # of threads is
        // max (1, min (floor (work / chunk), nthreads_max))

    int nthreads_max ; // max # of OpenMP threads to use in CHOLMOD.
        // Defaults to SUITESPARSE_OPENMP_MAX_THREADS.

} cholmod_common ;
```

Purpose: The `cholmod_common Common` object contains parameters, statistics, and workspace used within CHOLMOD. The first call to CHOLMOD must be `cholmod_start`, which initializes this object.

13.3 `cholmod_start`: start CHOLMOD

```
int cholmod_start (cholmod_common *Common) ;
int cholmod_l_start (cholmod_common *) ;
```

Purpose: Sets the default parameters, clears the statistics, and initializes all workspace pointers to `NULL`. The `int`/`long` type is set in `Common->itype`.

13.4 `cholmod_finish`: finish CHOLMOD

```
int cholmod_finish (cholmod_common *Common) ;
int cholmod_l_finish (cholmod_common *) ;
```

Purpose: This must be the last call to CHOLMOD.

13.5 `cholmod_defaults`: set default parameters

```
int cholmod_defaults (cholmod_common *Common) ;
int cholmod_l_defaults (cholmod_common *) ;
```

Purpose: Sets the default parameters.

13.6 `cholmod_maxrank`: maximum update/downdate rank

```
size_t cholmod_maxrank      // return validated Common->maxrank
(
    size_t n,               // # of rows of L and A
    cholmod_common *Common
) ;
size_t cholmod_l_maxrank (size_t, cholmod_common *) ;
```

Purpose: Returns the maximum rank for an update/downdate.

13.7 `cholmod_allocate_work`: allocate workspace

```
int cholmod_allocate_work
(
    size_t nrow,        // size of Common->Flag (nrow int32's)
                        // and Common->Head (nrow+1 int32's)
    size_t iworksize,   // size of Common->Iwork (# of int32's)
    size_t xworksize,   // size of Common->Xwork (# of double's)
    cholmod_common *Common
) ;
int cholmod_l_allocate_work (size_t, size_t, size_t, cholmod_common *) ;
```

Purpose: Allocates workspace in `Common`. The workspace consists of the integer `Head`, `Flag`, and `Iwork` arrays, of size `nrow+1`, `nrow`, and `iworksize`, respectively, and a `double` array `Xwork` of size `xworksize` entries. The `Head` array is normally equal to `-1` when it is cleared. If the `Flag` array is cleared, all entries are less than `Common->mark`. The `Iwork` array is not kept in any particular state. The integer type is `int` or `long`, depending on whether the `cholmod_` or `cholmod_l_` routines are used.

13.8 `cholmod_free_work`: free workspace

```
int cholmod_free_work (cholmod_common *Common) ;
int cholmod_l_free_work (cholmod_common *) ;
```

Purpose: Frees the workspace in `Common`.

13.9 `cholmod_clear_flag`: clear Flag array

```
int64_t cholmod_clear_flag (cholmod_common *Common) ;
int64_t cholmod_l_clear_flag (cholmod_common *) ;
```

Purpose: Increments `Common->mark` so that the `Flag` array is now cleared.

13.10 `cholmod_error`: report error

```
int cholmod_error
(
    int status,             // Common->status
    const char *file,       // source file where error occurred
    int line,               // line number where error occurred
    const char *message,    // error message to print
    cholmod_common *Common
) ;
int cholmod_l_error (int, const char *, int, const char *, cholmod_common *) ;
```

Purpose: This routine is called when CHOLMOD encounters an error. It prints a message (if printing is enabled), sets `Common->status`. It then calls the user error handler routine `Common->error_handler`, if it is not `NULL`.

13.11 `cholmod_dbound`: bound diagonal of L

```
double cholmod_dbound   (double, cholmod_common *) ;
double cholmod_l_dbound (double, cholmod_common *) ;
float  cholmod_sbound   (float,  cholmod_common *) ;
float  cholmod_l_sbound (float,  cholmod_common *) ;
```

Purpose: Ensures that entries on the diagonal of $\mathbf{L}$ for an $\mathbf{LL}^\mathsf{T}$ factorization are greater than or equal to `Common->dbound`. For an $\mathbf{LDL}^\mathsf{T}$ factorization, it ensures that the magnitude of the entries of $\mathbf{D}$ are greater than or equal to `Common->dbound`.

13.12 `cholmod_hypot`: `sqrt(x*x+y*y)`

```
double cholmod_hypot (double x, double y) ;
double cholmod_l_hypot (double, double) ;
```

Purpose: Computes the magnitude of a complex number. This routine is the default value for the `Common->hypotenuse` function pointer. See also `hypot`, in the standard `math.h` header. If you have the ANSI C99 `hypot`, you can use `Common->hypotenuse = hypot`. The `cholmod_hypot` routine is provided in case you are using the ANSI C89 standard, which does not have `hypot`.

13.13 `cholmod_divcomplex`: complex divide

```
int cholmod_divcomplex          // return 1 if divide-by-zero, 0 if OK
(
    double ar, double ai,       // a (real, imaginary)
    double br, double bi,       // b (real, imaginary)
    double *cr, double *ci      // c (real, imaginary)
) ;
int cholmod_l_divcomplex (double, double, double, double, double *, double *) ;
```

Purpose: Divides two complex numbers. It returns 1 if a divide-by-zero occurred, or 0 otherwise. This routine is the default value for the `Common->complex_divide` function pointer. This return value is the single exception to the CHOLMOD rule that states all `int` return values are `TRUE` if successful or `FALSE` otherwise. The exception is made to match the return value of a different complex divide routine that is not a part of CHOLMOD, but can be used via the function pointer.

14 Utility Module: cholmod_sparse object

14.1 cholmod_sparse: compressed-column sparse matrix

```
typedef struct cholmod_sparse_struct
{
    size_t nrow ;   // # of rows of the matrix
    size_t ncol ;   // # of colums of the matrix
    size_t nzmax ;  // max # of entries that can be held in the matrix

    // int32_t or int64_t arrays:
    void *p ;   // A->p [0..ncol], column "pointers" of the CSC matrix
    void *i ;   // A->i [0..nzmax-1], the row indices

    // for unpacked matrices only:
    void *nz ;  // A->nz [0..ncol-1], is the # of nonzeros in each col.
        // This is NULL for a "packed" matrix (conventional CSC).
        // For a packed matrix, the jth column is held in A->i and A->x in
        // postions A->p [j] to A->p [j+1]-1, with no gaps between columns.
        // For an "unpacked" matrix, there can be gaps between columns, so
        // the jth columns appears in positions A-p [j] to
        // A->p [j] + A->nz [j] - 1.

    // double or float arrays:
    void *x ;   // size nzmax or 2*nzmax, or NULL
    void *z ;   // size nzmax, or NULL

    int stype ; // A->stype defines what parts of the matrix is held:
        // 0:  the matrix is unsymmetric with both lower and upper parts stored.
        // >0: the matrix is square and symmetric, with just the upper
        //     triangular part stored.
        // <0: the matrix is square and symmetric, with just the lower
        //     triangular part stored.

    int itype ; // A->itype defines the integers used for A->p, A->i, and A->nz.
        // if CHOLMOD_INT, these arrays are all of type int32_t.
        // if CHOLMOD_LONG, these arrays are all of type int64_t.
    int xtype ;     // pattern, real, complex, or zomplex
    int dtype ;     // x and z are double or single
    int sorted ;    // true if columns are sorted, false otherwise
    int packed ;    // true if packed (A->nz ignored), false if unpacked

} cholmod_sparse ;
```

Purpose: Stores a sparse matrix in compressed-column form.

14.2 cholmod_allocate_sparse: allocate sparse matrix

```
cholmod_sparse *cholmod_allocate_sparse
(
    size_t nrow,    // # of rows
    size_t ncol,    // # of columns
    size_t nzmax,   // max # of entries the matrix can hold
```

```
    int sorted,     // true if columns are sorted
    int packed,     // true if A is be packed (A->nz NULL), false if unpacked
    int stype,      // the stype of the matrix (unsym, tril, or triu)
    int xdtype,     // xtype + dtype of the matrix:
                    // (CHOLMOD_DOUBLE, _SINGLE) +
                    // (CHOLMOD_PATTERN, _REAL, _COMPLEX, or _ZOMPLEX)
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_allocate_sparse (size_t, size_t, size_t, int, int,
    int, int, cholmod_common *) ;
```

Purpose: Allocates a sparse matrix. `A->i`, `A->x`, and `A->z` are not initialized. The matrix returned is all zero, but it contains space enough for `nzmax` entries.

14.3 `cholmod_free_sparse`: free sparse matrix

```
int cholmod_free_sparse
(
    cholmod_sparse **A,         // handle of sparse matrix to free
    cholmod_common *Common
) ;
int cholmod_l_free_sparse (cholmod_sparse **, cholmod_common *) ;
```

Purpose: Frees a sparse matrix.

14.4 `cholmod_reallocate_sparse`: reallocate sparse matrix

```
int cholmod_reallocate_sparse
(
    size_t nznew,       // new max # of nonzeros the sparse matrix can hold
    cholmod_sparse *A,  // sparse matrix to reallocate
    cholmod_common *Common
) ;
int cholmod_l_reallocate_sparse (size_t, cholmod_sparse *, cholmod_common *) ;
```

Purpose: Reallocates a sparse matrix, so that it can contain `nznew` entries.

14.5 `cholmod_nnz`: number of entries in sparse matrix

```
int64_t cholmod_nnz             // return # of entries in the sparse matrix
(
    cholmod_sparse *A,          // sparse matrix to query
    cholmod_common *Common
) ;
int64_t cholmod_l_nnz (cholmod_sparse *, cholmod_common *) ;
```

Purpose: Returns the number of entries in a sparse matrix.

14.6 cholmod_speye: sparse identity matrix

```
cholmod_sparse *cholmod_speye
(
    size_t nrow,    // # of rows
    size_t ncol,    // # of columns
    int xdtype,     // xtype + dtype of the matrix:
                    // (CHOLMOD_DOUBLE, _SINGLE) +
                    // (CHOLMOD_PATTERN, _REAL, _COMPLEX, or _ZOMPLEX)
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_speye (size_t, size_t, int, cholmod_common *) ;
```

Purpose: Returns the sparse identity matrix.

14.7 cholmod_spzeros: sparse zero matrix

```
cholmod_sparse *cholmod_spzeros     // return a sparse matrix with no entries
(
    size_t nrow,    // # of rows
    size_t ncol,    // # of columns
    size_t nzmax,   // max # of entries the matrix can hold
    int xdtype,     // xtype + dtype of the matrix:
                    // (CHOLMOD_DOUBLE, _SINGLE) +
                    // (CHOLMOD_PATTERN, _REAL, _COMPLEX, or _ZOMPLEX)
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_spzeros (size_t, size_t, size_t, int,
    cholmod_common *) ;
```

Purpose: Returns the sparse zero matrix. This is another name for `cholmod_allocate_sparse`, but with fewer parameters (the matrix is packed, sorted, and unsymmetric).

14.8 cholmod_transpose: transpose sparse matrix

```
cholmod_sparse *cholmod_transpose       // return new sparse matrix C
(
    cholmod_sparse *A,  // input matrix
    int mode,           // 2: numerical (conj), 1: numerical (non-conj.),
                        // <= 0: pattern (with diag)
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_transpose (cholmod_sparse *, int, cholmod_common *) ;
```

Purpose: Returns the transpose or complex conjugate transpose of a sparse matrix.

14.9 cholmod_ptranspose: transpose/permute sparse matrix

```
cholmod_sparse *cholmod_ptranspose      // return new sparse matrix C
(
    cholmod_sparse *A,  // input matrix
    int mode,           // 2: numerical (conj), 1: numerical (non-conj.),
                        // <= 0: pattern (with diag)
    int32_t *Perm,      // permutation for C=A(p,f)', or NULL
    int32_t *fset,      // a list of column indices in range 0:A->ncol-1
    size_t fsize,       // # of entries in fset
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_ptranspose (cholmod_sparse *, int, int64_t *,
    int64_t *, size_t, cholmod_common *) ;
```

Purpose: Returns `A'` or `A(p,p)'` if `A` is symmetric. Returns `A'`, `A(:,f)'`, or `A(p,f)'` if `A` is unsymmetric. See `cholmod_transpose_unsym` for a discussion of how `f` is used; this usage deviates from the MATLAB notation. Can also return the array transpose.

14.10 cholmod_sort: sort columns of a sparse matrix

```
int cholmod_sort
(
    cholmod_sparse *A,      // input/output matrix to sort
    cholmod_common *Common
) ;
int cholmod_l_sort (cholmod_sparse *, cholmod_common *) ;
```

Purpose: Sorts the columns of the matrix `A`. Returns `A` in packed form, even if it starts as unpacked. Removes entries in the ignored part of a symmetric matrix.

14.11 cholmod_transpose_unsym: transpose/permute unsymmetric sparse matrix

```
int cholmod_transpose_unsym
(
    cholmod_sparse *A,  // input matrix
    int mode,           // 2: numerical (conj), 1: numerical (non-conj.),
                        // <= 0: pattern (with diag)
    int32_t *Perm,      // permutation for C=A(p,f)', or NULL
    int32_t *fset,      // a list of column indices in range 0:A->ncol-1
    size_t fsize,       // # of entries in fset
    cholmod_sparse *C,  // output matrix, must be allocated on input
    cholmod_common *Common
) ;
int cholmod_l_transpose_unsym (cholmod_sparse *, int, int64_t *, int64_t *,
    size_t, cholmod_sparse *, cholmod_common *) ;
```

Purpose: Transposes and optionally permutes an unsymmetric sparse matrix. The output matrix must be preallocated before calling this routine.

Computes `F=A'`, `F=A(:,f)'` or `F=A(p,f)'`, except that the indexing by `f` does not work the same as the MATLAB notation (see below). `A->stype` is zero, which denotes that both the upper and lower triangular parts of A are present (and used). The matrix `A` may in fact be symmetric in pattern and/or value; `A->stype` just denotes which part of `A` are stored. `A` may be rectangular.

The integer vector `p` is a permutation of `0:m-1`, and `f` is a subset of `0:n-1`, where A is `m`-by-`n`. There can be no duplicate entries in `p` or `f`.
Three kinds of transposes are available, depending on the `values` parameter:

- 0: do not transpose the numerical values; create a `CHOLMOD_PATTERN` matrix
- 1: array transpose
- 2: complex conjugate transpose (same as 2 if input is real or pattern)

The set `f` is held in fset and fsize:

- `fset = NULL` means ":" in MATLAB. `fset` is ignored.
- `fset != NULL` means `f = fset [0..fsize-1]`.
- `fset != NULL` and `fsize = 0` means `f` is the empty set.

Columns not in the set `f` are considered to be zero. That is, if `A` is 5-by-10 then `F=A(:,[3 4])'` is not 2-by-5, but 10-by-5, and rows 3 and 4 of `F` are equal to columns 3 and 4 of `A` (the other rows of `F` are zero). More precisely, in MATLAB notation:

```
[m n] = size (A)
F = A
notf = ones (1,n)
notf (f) = 0
F (:, find (notf)) = 0
F = F'
```

If you want the MATLAB equivalent F=A(p,f) operation, use cholmod_submatrix instead (which does not compute the transpose). F->nzmax must be large enough to hold the matrix F. If F->nz is present then F->nz [j] is equal to the number of entries in column j of F. A can be sorted or unsorted, with packed or unpacked columns. If f is present and not sorted in ascending order, then F is unsorted (that is, it may contain columns whose row indices do not appear in ascending order). Otherwise, F is sorted (the row indices in each column of F appear in strictly ascending order).

F is returned in packed or unpacked form, depending on F->packed on input. If F->packed is FALSE, then F is returned in unpacked form (F->nz must be present). Each row i of F is large enough to hold all the entries in row i of A, even if f is provided. That is, F->i and F->x [F->p [i] .. F->p [i] + F->nz [i] - 1] contain all entries in A(i,f), but F->p [i+1] - F->p [i] is equal to the number of nonzeros in A (i,:), not just A (i,f). The cholmod_transpose_unsym routine is the only operation in CHOLMOD that can produce an unpacked matrix.

14.12 cholmod_transpose_sym: transpose/permute symmetric sparse matrix

```
int cholmod_transpose_sym
(
    cholmod_sparse *A,  // input matrix
    int mode,           // 2: numerical (conj), 1: numerical (non-conj.),
                        // <= 0: pattern (with diag)
    int32_t *Perm,      // permutation for C=A(p,p)', or NULL
    cholmod_sparse *C,  // output matrix, must be allocated on input
    cholmod_common *Common
) ;
int cholmod_l_transpose_sym (cholmod_sparse *, int, int64_t *, cholmod_sparse *,
    cholmod_common *) ;
```

Purpose: Computes F = A' or F = A(p,p)', the transpose or permuted transpose, where A->stype is nonzero. A must be square and symmetric. If A->stype > 0, then A is a symmetric matrix where just the upper part of the matrix is stored. Entries in the lower triangular part may be present, but are ignored. If A->stype < 0, then A is a symmetric matrix where just the lower part of the matrix is stored. Entries in the upper triangular part may be present, but are ignored. If F=A', then F is returned sorted; otherwise F is unsorted for the F=A(p,p)' case. There can be no duplicate entries in p.

Three kinds of transposes are available, depending on the values parameter:

- 0: do not transpose the numerical values; create a CHOLMOD_PATTERN matrix
- 1: array transpose
- 2: complex conjugate transpose (same as 2 if input is real or pattern)

For cholmod_transpose_unsym and cholmod_transpose_sym, the output matrix F must already be pre-allocated by the caller, with the correct dimensions. If F is not valid or has the wrong dimensions, it is not modified. Otherwise, if F is too small, the transpose is not computed; the contents of F->p contain the column pointers of the resulting matrix, where F->p [F->ncol] > F->nzmax. In this case, the remaining contents of F are not modified. F can still be properly freed with cholmod_free_sparse.

14.13 `cholmod_band`: extract band of a sparse matrix

```
cholmod_sparse *cholmod_band    // return a new matrix C
(
    cholmod_sparse *A,      // input matrix
    int64_t k1,             // count entries in k1:k2 diagonals
    int64_t k2,
    int mode,               // >0: numerical, 0: pattern, <0: pattern (no diag)
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_band (cholmod_sparse *, int64_t, int64_t, int,
    cholmod_common *) ;
```

Purpose: Returns `C = tril (triu (A,k1), k2)`. `C` is a matrix consisting of the diagonals of A from `k1` to `k2`. `k=0` is the main diagonal of `A`, `k=1` is the superdiagonal, `k=-1` is the subdiagonal, and so on. If `A` is `m`-by-`n`, then:

- `k1=-m` means `C = tril (A,k2)`
- `k2=n` means `C = triu (A,k1)`
- `k1=0` and `k2=0` means `C = diag(A)`, except `C` is a matrix, not a vector

Values of `k1` and `k2` less than `-m` are treated as `-m`, and values greater than `n` are treated as `n`.

`A` can be of any symmetry (upper, lower, or unsymmetric); `C` is returned in the same form, and packed. If `A->stype` > 0, entries in the lower triangular part of `A` are ignored, and the opposite is true if `A->stype` < 0. If `A` has sorted columns, then so does `C`. `C` has the same size as `A`.

`C` can be returned as a numerical valued matrix (if `A` has numerical values and `mode` > 0), as a pattern-only (`mode` $= 0$), or as a pattern-only but with the diagonal entries removed (`mode` < 0).

The xtype of `A` can be pattern or real. Complex or zomplex cases are supported only if `mode` is ≤ 0 (in which case the numerical values are ignored).

14.14 `cholmod_band_inplace`: extract band, in place

```
int cholmod_band_inplace
(
    int64_t k1,             // count entries in k1:k2 diagonals
    int64_t k2,
    int mode,               // >0: numerical, 0: pattern, <0: pattern (no diag)
    cholmod_sparse *A,      // input/output matrix
    cholmod_common *Common
) ;
int cholmod_l_band_inplace (int64_t, int64_t, int, cholmod_sparse *,
    cholmod_common *) ;
```

Purpose: Same as `cholmod_band`, except that it always operates in place. Only packed matrices can be converted in place.

14.15 cholmod_aat: compute AA^T

```
cholmod_sparse *cholmod_aat     // return sparse matrix C
(
    cholmod_sparse *A,  // input matrix
    int32_t *fset,      // a list of column indices in range 0:A->ncol-1
    size_t fsize,       // # of entries in fset
    int mode,           // 2: numerical (conj), 1: numerical (non-conj.),
                        // 0: pattern (with diag), -1: pattern (remove diag),
                        // -2: pattern (remove diag; add ~50% extra space in C)
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_aat (cholmod_sparse *, int64_t *, size_t, int,
    cholmod_common *) ;
```

Purpose: Computes `C = A*A'` or `C = A(:,f)*A(:,f)'`. `A` can be packed or unpacked, sorted or unsorted, but must be stored with both upper and lower parts (`A->stype` of zero). `C` is returned as packed, `C->stype` of zero (both upper and lower parts present), and unsorted. See `cholmod_ssmult` in the `MatrixOps` Module for a more general matrix-matrix multiply. The xtype of `A` can be pattern or real. Complex or zomplex cases are supported only if `mode` is ≤ 0 (in which case the numerical values are ignored). You can trivially convert `C` to a symmetric upper/lower matrix by changing `C->stype` to 1 or -1, respectively, after calling this routine.

14.16 cholmod_copy_sparse: copy sparse matrix

```
cholmod_sparse *cholmod_copy_sparse  // return new sparse matrix
(
    cholmod_sparse *A,     // sparse matrix to copy
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_copy_sparse (cholmod_sparse *, cholmod_common *) ;
```

Purpose: Returns an exact copy of the input sparse matrix `A`.

14.17 cholmod_copy: copy (and change) sparse matrix

```
cholmod_sparse *cholmod_copy_sparse  // return new sparse matrix
(
    cholmod_sparse *A,     // sparse matrix to copy
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_copy_sparse (cholmod_sparse *, cholmod_common *) ;
cholmod_sparse *cholmod_copy        // return new sparse matrix
(
    cholmod_sparse *A,  // input matrix, not modified
    int stype,          // stype of C
    int mode,           // 2: numerical (conj), 1: numerical (non-conj.),
                        // 0: pattern (with diag), -1: pattern (remove diag),
                        // -2: pattern (remove diag; add ~50% extra space in C)
    cholmod_common *Common
```

```
) ;
cholmod_sparse *cholmod_l_copy (cholmod_sparse *, int, int, cholmod_common *) ;
```

Purpose: `C = A`, which allocates `C` and copies `A` into `C`, with possible change of `stype`. The diagonal can optionally be removed. The numerical entries can optionally be copied. This routine differs from `cholmod_copy_sparse`, which makes an exact copy of a sparse matrix.

`A` can be of any type (packed/unpacked, upper/lower/unsymmetric). `C` is packed and can be of any stype (upper/lower/unsymmetric), except that if `A` is rectangular `C` can only be unsymmetric. If the stype of A and C differ, then the appropriate conversion is made.
There are three cases for `A->stype`:

- < 0, lower: assume `A` is symmetric with just `tril(A)` stored; the rest of `A` is ignored
- 0, unsymmetric: assume `A` is unsymmetric; consider all entries in A
- > 0, upper: assume `A` is symmetric with just `triu(A)` stored; the rest of `A` is ignored

There are three cases for the requested symmetry of `C` (`stype` parameter):

- < 0, lower: return just `tril(C)`
- 0, unsymmetric: return all of `C`
- > 0, upper: return just `triu(C)`

This gives a total of nine combinations:

Equivalent MATLAB statements	Using `cholmod_copy`
`C = A ;`	`A` unsymmetric, `C` unsymmetric
`C = tril (A) ;`	`A` unsymmetric, `C` lower
`C = triu (A) ;`	`A` unsymmetric, `C` upper
`U = triu (A) ; L = tril (U',-1) ; C = L+U ;`	`A` upper, `C` unsymmetric
`C = triu (A)' ;`	`A` upper, `C` lower
`C = triu (A) ;`	`A` upper, `C` upper
`L = tril (A) ; U = triu (L',1) ; C = L+U ;`	`A` lower, `C` unsymmetric
`C = tril (A) ;`	`A` lower, `C` lower
`C = tril (A)' ;`	`A` lower, `C` upper

The xtype of `A` can be pattern or real. Complex or zomplex cases are supported only if `values` is `FALSE` (in which case the numerical values are ignored).

14.18 `cholmod_add`: add sparse matrices

```
cholmod_sparse *cholmod_add     // return C = alpha*A + beta*B
(
    cholmod_sparse *A,  // input matrix
    cholmod_sparse *B,  // input matrix
    double alpha [2],   // scale factor for A (two entires used if complex)
    double beta [2],    // scale factor for A (two entires used if complex)
    int values,         // if TRUE compute the numerical values of C
    int sorted,         // ignored; C is now always returned as sorted
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_add (cholmod_sparse *, cholmod_sparse *, double *,
    double *, int, int, cholmod_common *) ;
```

Purpose: Returns `C = alpha*A + beta*B`. If the `stype` of `A` and `B` match, then `C` has the same `stype`. Otherwise, `C->stype` is zero (`C` is unsymmetric).

14.19 `cholmod_sparse_xtype`: change sparse xtype

```
int cholmod_sparse_xtype
(
    int to_xdtype,      // requested xtype and dtype
    cholmod_sparse *A,  // sparse matrix to change
    cholmod_common *Common
) ;
int cholmod_l_sparse_xtype (int, cholmod_sparse *, cholmod_common *) ;
```

Purpose: Changes the `xtype` of a sparse matrix, to pattern, real, complex, or zomplex. Changing from complex or zomplex to real discards the imaginary part.

15 Utility Module: cholmod_factor object

15.1 cholmod_factor object: a sparse Cholesky factorization

```
typedef struct cholmod_factor_struct
{
    size_t n ;      // L is n-by-n

    size_t minor ;  // If the factorization failed because of numerical issues
        // (the matrix being factorized is found to be singular or not positive
        // definte), then L->minor is the column at which it failed.  L->minor
        // = n means the factorization was successful.

    //-------------------------------------------------------------------------
    // symbolic ordering and analysis
    //-------------------------------------------------------------------------

    void *Perm ;    // int32/int64, size n, fill-reducing ordering
    void *ColCount ;// int32/int64, size n, # entries in each column of L
    void *IPerm ;   // int32/int64, size n, created by cholmod_solve2;
                    // containing the inverse of L->Perm

    //-------------------------------------------------------------------------
    // simplicial factorization (not supernodal)
    //-------------------------------------------------------------------------

    size_t nzmax ;  // # of entries that L->i, L->x, and L->z can hold

    void *p ;       // int32/int64, size n+1, column pointers
    void *i ;       // int32/int64, size nzmax, row indices
    void *x ;       // float/double, size nzmax or 2*nzmax, numerical values
    void *z ;       // float/double, size nzmax or empty, imaginary values
    void *nz ;      // int32/int64, size ncol, # of entries in each column

    // The row indices of L(:,j) are held in
    // L->i [L->p [j] ... L->p [j] + L->nz [j] - 1].

    // The numeical values of L(:,j) are held in the same positions in L->x
    // (and L->z if L is zomplex)

    // L->next and L->prev hold a link list of columns of L, that tracks the
    // order they appear in the arrays L->i, L->x, and L->z.  The head and tail
    // of the list is n+1 and n, respectively.

    void *next ;    // int32/int64, size n+2
    void *prev ;    // int32/int64, size n+2

    //-------------------------------------------------------------------------
    // supernodal factorization (not simplicial)
    //-------------------------------------------------------------------------

    // L->x is shared with the simplicial structure above.  L->z is not used
    // for the supernodal case since a supernodal factor cannot be zomplex.

    size_t nsuper ;     // # of supernodes
```

```
size_t ssize ;      // # of integers in L->s
size_t xsize ;      // # of entries in L->x
size_t maxcsize ;   // size of largest update matrix
size_t maxesize ;   // max # of rows in supernodes, excl. triangular part

// the following are int32/int64 and are size nsuper+1:
void *super ;       // first column in each supernode
void *pi ;          // index into L->s for integer part of a supernode
void *px ;          // index into L->x for numeric part of a supernode

void *s ;           // int32/int64, ssize, integer part of supernodes

//--------------------------------------------------------------------------
// type of the factorization
//--------------------------------------------------------------------------

int ordering ;      // the fill-reducing method used (CHOLMOD_NATURAL,
    // CHOLMOD_GIVEN, CHOLMOD_AMD, CHOLMOD_METIS, CHOLMOD_NESDIS,
    // CHOLMOD_COLAMD, or CHOLMOD_POSTORDERED).

int is_ll ;         // true: an LL' factorization; false: LDL' instead
int is_super ;      // true: supernodal; false: simplicial
int is_monotonic ;  // true: columns appear in order 0 to n-1 in L, for a
                    // simplicial factorization only

// Two boolean values above (is_ll, is_super) and L->xtype (pattern or
// otherwise, define eight types of factorizations, but only 6 are used:

// If L->xtype is CHOLMOD_PATTERN, then L is a symbolic factor:
//
// simplicial LDL': (is_ll false, is_super false).  Nothing is present
//      except Perm and ColCount.
//
// simplicial LL': (is_ll true, is_super false).  Identical to the
//      simplicial LDL', except for the is_ll flag.
//
// supernodal LL': (is_ll true, is_super true).  A supernodal symbolic
//      factorization.  The simplicial symbolic information is present
//      (Perm and ColCount), as is all of the supernodal factorization
//      except for the numerical values (x and z).
//
// If L->xtype is CHOLMOD_REAL, CHOLMOD_COMPLEX, or CHOLMOD_ZOMPLEX,
// then L is a numeric factor:
//
//
// simplicial LDL':  (is_ll false, is_super false).  Stored in compressed
//      column form, using the simplicial components above (nzmax, p, i,
//      x, z, nz, next, and prev).  The unit diagonal of L is not stored,
//      and D is stored in its place.  There are no supernodes.
//
// simplicial LL': (is_ll true, is_super false).  Uses the same storage
//      scheme as the simplicial LDL', except that D does not appear.
//      The first entry of each column of L is the diagonal entry of
//      that column of L.
//
```

```
    // supernodal LL': (is_ll true, is_super true).  A supernodal factor,
    //      using the supernodal components described above (nsuper, ssize,
    //      xsize, maxcsize, maxesize, super, pi, px, s, x, and z).
    //      A supernodal factorization is never zomplex.

    int itype ; // integer type for L->Perm, L->ColCount, L->p, L->i, L->nz,
                // L->next, L->prev, L->super, L->pi, L->px, and L->s.
                // These are all int32 if L->itype is CHOLMOD_INT, or all int64
                // if L->itype is CHOLMOD_LONG.

    int xtype ; // pattern, real, complex, or zomplex
    int dtype ; // x and z are double or single

    int useGPU; // if true, symbolic factorization allows for use of the GPU

} cholmod_factor ;
```

Purpose: An $\mathbf{LL}^\mathsf{T}$ or $\mathbf{LDL}^\mathsf{T}$ factorization in simplicial or supernodal form. A simplicial factor is very similar to a `cholmod_sparse` matrix. For an $\mathbf{LDL}^\mathsf{T}$ factorization, the diagonal matrix $\mathbf{D}$ is stored as the diagonal of $\mathbf{L}$; the unit-diagonal of $\mathbf{L}$ is not stored.

15.2 `cholmod_free_factor`: free factor

```
int cholmod_free_factor
(
    cholmod_factor **L,         // handle of sparse factorization to free
    cholmod_common *Common
) ;
int cholmod_l_free_factor (cholmod_factor **, cholmod_common *) ;
```

Purpose: Frees a factor.

15.3 `cholmod_allocate_factor`: allocate factor

```
cholmod_factor *cholmod_allocate_factor         // return the new factor L
(
    size_t n,               // L is factorization of an n-by-n matrix
    cholmod_common *Common
) ;
cholmod_factor *cholmod_l_allocate_factor (size_t, cholmod_common *) ;
```

Purpose: Allocates a factor and sets it to identity.

15.4 `cholmod_reallocate_factor`: reallocate factor

```
int cholmod_reallocate_factor
(
    size_t nznew,       // new max # of nonzeros the factor matrix can hold
    cholmod_factor *L,  // factor to reallocate
    cholmod_common *Common
) ;
int cholmod_l_reallocate_factor (size_t, cholmod_factor *, cholmod_common *) ;
```

Purpose: Reallocates a simplicial factor so that it can contain `nznew` entries.

15.5 cholmod_change_factor: change factor

```
int cholmod_change_factor
(
    int to_xtype,       // CHOLMOD_PATTERN, _REAL, _COMPLEX, or _ZOMPLEX
    int to_ll,          // if true: convert to LL'; else to LDL'
    int to_super,       // if true: convert to supernodal; else to simplicial
    int to_packed,      // if true: pack simplicial columns' else: do not pack
    int to_monotonic,   // if true, put simplicial columns in order
    cholmod_factor *L,  // factor to change.
    cholmod_common *Common
) ;
int cholmod_l_change_factor (int, int, int, int, int, cholmod_factor *,
    cholmod_common *) ;
```

Purpose: Change the numeric or symbolic, $\mathbf{LL}^\mathsf{T}$ or $\mathbf{LDL}^\mathsf{T}$, simplicial or super, packed or unpacked, and monotonic or non-monotonic status of a `cholmod_factor` object.

There are four basic classes of factor types:

1. simplicial symbolic: Consists of two size-`n` arrays: the fill-reducing permutation (`L->Perm`) and the nonzero count for each column of L (`L->ColCount`). All other factor types also include this information. `L->ColCount` may be exact (obtained from the analysis routines), or it may be a guess. During factorization, and certainly after update/downdate, the columns of `L` can have a different number of nonzeros. `L->ColCount` is used to allocate space. `L->ColCount` is exact for the supernodal factorizations. The nonzero pattern of `L` is not kept.

2. simplicial numeric: These represent `L` in a compressed column form. The variants of this type are:

   - $\mathbf{LDL}^\mathsf{T}$: `L` is unit diagonal. Row indices in column `j` are located in `L->i [L->p [j] ... L->p [j] + L->nz [j]]`, and corresponding numeric values are in the same locations in `L->x`. The total number of entries is the sum of `L->nz [j]`. The unit diagonal is not stored; `D` is stored on the diagonal of `L` instead. `L->p` may or may not be monotonic. The order of storage of the columns in `L->i` and `L->x` is given by a doubly-linked list (`L->prev` and `L->next`). `L->p` is of size `n+1`, but only the first `n` entries are used.

     For the complex case, `L->x` is stored interleaved with real and imaginary parts, and is of size `2*lnz*sizeof(double)`. For the zomplex case, `L->x` is of size `lnz*sizeof(double)` and holds the real part; `L->z` is the same size and holds the imaginary part.

   - $\mathbf{LL}^\mathsf{T}$: This is identical to the $\mathbf{LDL}^\mathsf{T}$ form, except that the non-unit diagonal of `L` is stored as the first entry in each column of `L`.

3. supernodal symbolic: A representation of the nonzero pattern of the supernodes for a supernodal factorization. There are `L->nsuper` supernodes. Columns `L->super [k]` to `L->super [k+1]-1` are in the `k`th supernode. The row indices for the `k`th supernode are in `L->s [L->pi [k] ... L->pi [k+1]-1]`. The numerical values are not allocated (`L->x`), but when they are they will be located in `L->x [L->px [k] ... L->px [k+1]-1]`, and the `L->px` array is defined in this factor type.

For the complex case, L->x is stored interleaved with real/imaginary parts, and is of size 2*L->xsize*sizeof(double). The zomplex supernodal case is not supported, since it is not compatible with LAPACK and the BLAS.

4. supernodal numeric: Always an $\mathbf{LL}^\mathsf{T}$ factorization. L has a non-unit diagonal. L->x contains the numerical values of the supernodes, as described above for the supernodal symbolic factor. For the complex case, L->x is stored interleaved, and is of size 2*L->xsize*sizeof(double). The zomplex supernodal case is not supported, since it is not compatible with LAPACK and the BLAS.

In all cases, the row indices in each column (L->i for simplicial L and L->s for supernodal L) are kept sorted from low indices to high indices. This means the diagonal of L (or D for a $\mathbf{LDL}^\mathsf{T}$ factorization) is always kept as the first entry in each column. The elimination tree is not kept. The parent of node j can be found as the second row index in the jth column. If column j has no off-diagonal entries then node j is a root of the elimination tree.

The cholmod_change_factor routine can do almost all possible conversions. It cannot do the following conversions:

- Simplicial numeric types cannot be converted to a supernodal symbolic type. This would simultaneously deallocate the simplicial pattern and numeric values and reallocate uninitialized space for the supernodal pattern. This isn't useful for the user, and not needed by CHOLMOD's own routines either.
- Only a symbolic factor (simplicial to supernodal) can be converted to a supernodal numeric factor.

Some conversions are meant only to be used internally by other CHOLMOD routines, and should not be performed by the end user. They allocate space whose contents are undefined:

- converting from simplicial symbolic to supernodal symbolic.
- converting any factor to supernodal numeric.

Supports all xtypes, except that there is no supernodal zomplex L.

The to_xtype parameter is used only when converting from symbolic to numeric or numeric to symbolic. It cannot be used to convert a numeric xtype (real, complex, or zomplex) to a different numeric xtype. For that conversion, use cholmod_factor_xtype instead.

15.6 cholmod_pack_factor: pack the columns of a factor

```
int cholmod_pack_factor
(
    cholmod_factor *L,      // factor to pack
    cholmod_common *Common
) ;
int cholmod_l_pack_factor (cholmod_factor *, cholmod_common *) ;
```

Purpose: Pack the columns of a simplicial $\mathbf{LDL}^\mathsf{T}$ or $\mathbf{LL}^\mathsf{T}$ factorization. This can be followed by a call to `cholmod_reallocate_factor` to reduce the size of `L` to the exact size required by the factor, if desired. Alternatively, you can leave the size of `L->i` and `L->x` the same, to allow space for future updates/rowadds. Each column is reduced in size so that it has at most `Common->grow2` free space at the end of the column. Does nothing and returns silently if given any other type of factor. Does not force the columns of `L` to be monotonic. It thus differs from

```
cholmod_change_factor (xtype, L->is_ll, FALSE, TRUE, TRUE, L, Common)
```

which packs the columns and ensures that they appear in monotonic order.

15.7 cholmod_reallocate_column: reallocate one column of a factor

```
int cholmod_reallocate_column
(
    size_t j,                  // reallocate L(:,j)
    size_t need,               // space in L(:,j) for this # of entries
    cholmod_factor *L,         // L factor modified, L(:,j) resized
    cholmod_common *Common
) ;
int cholmod_l_reallocate_column (size_t, size_t, cholmod_factor *,
    cholmod_common *) ;
```

Purpose: Reallocates the space allotted to a single column of **L**.

15.8 cholmod_factor_to_sparse: sparse matrix copy of a factor

```
cholmod_sparse *cholmod_factor_to_sparse    // return a new sparse matrix
(
    cholmod_factor *L,  // input: factor to convert; output: L is converted
                        // to a simplicial symbolic factor
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_factor_to_sparse (cholmod_factor *,
    cholmod_common *) ;
```

Purpose: Returns a column-oriented sparse matrix containing the pattern and values of a simplicial or supernodal numerical factor, and then converts the factor into a simplicial symbolic factor. If `L` is already packed, monotonic, and simplicial (which is the case when `cholmod_factorize` uses the simplicial Cholesky factorization algorithm) then this routine requires only a small amount of time and memory, independent of `n`. It only operates on numeric factors (real, complex, or zomplex). It does not change `L->xtype` (the resulting sparse matrix has the same `xtype` as `L`). If this routine fails, `L` is left unmodified.

15.9 cholmod_copy_factor: copy factor

```
cholmod_factor *cholmod_copy_factor     // return a copy of the factor
(
    cholmod_factor *L,      // factor to copy (not modified)
    cholmod_common *Common
) ;
cholmod_factor *cholmod_l_copy_factor (cholmod_factor *, cholmod_common *) ;
```

Purpose: Returns an exact copy of a factor.

15.10 cholmod_factor_xtype: change factor xtype

```
int cholmod_factor_xtype
(
    int to_xdtype,      // requested xtype and dtype
    cholmod_factor *L,  // factor to change
    cholmod_common *Common
) ;
int cholmod_l_factor_xtype (int, cholmod_factor *, cholmod_common *) ;
```

Purpose: Changes the `xtype` of a factor, to pattern, real, complex, or zomplex. Changing from complex or zomplex to real discards the imaginary part. You cannot change a supernodal factor to the zomplex xtype.

16 Utility Module: cholmod_dense object

16.1 cholmod_dense object: a dense matrix

```
typedef struct cholmod_dense_struct
{
    size_t nrow ;       // the matrix is nrow-by-ncol
    size_t ncol ;
    size_t nzmax ;      // maximum number of entries in the matrix
    size_t d ;          // leading dimension (d >= nrow must hold)
    void *x ;           // size nzmax or 2*nzmax, if present
    void *z ;           // size nzmax, if present
    int xtype ;         // pattern, real, complex, or zomplex
    int dtype ;         // x and z double or single

} cholmod_dense ;
```

Purpose: Contains a dense matrix.

16.2 cholmod_allocate_dense: allocate dense matrix

```
cholmod_dense *cholmod_allocate_dense
(
    size_t nrow,    // # of rows
    size_t ncol,    // # of columns
    size_t d,       // leading dimension
    int xdtype,     // xtype + dtype of the matrix:
                    // (CHOLMOD_DOUBLE, _SINGLE) +
                    // (CHOLMOD_REAL, _COMPLEX, or _ZOMPLEX)
    cholmod_common *Common
) ;
cholmod_dense *cholmod_l_allocate_dense (size_t, size_t, size_t, int,
    cholmod_common *) ;
```

Purpose: Allocates a dense matrix.

16.3 cholmod_free_dense: free dense matrix

```
int cholmod_free_dense
(
    cholmod_dense **X,          // handle of dense matrix to free
    cholmod_common *Common
) ;
int cholmod_l_free_dense (cholmod_dense **, cholmod_common *) ;
```

Purpose: Frees a dense matrix.

16.4 `cholmod_ensure_dense`: ensure dense matrix has a given size and type

```
cholmod_dense *cholmod_ensure_dense
(
    cholmod_dense **X,  // matrix to resize as needed (*X may be NULL)
    size_t nrow,    // # of rows
    size_t ncol,    // # of columns
    size_t d,       // leading dimension
    int xdtype,     // xtype + dtype of the matrix:
                    // (CHOLMOD_DOUBLE, _SINGLE) +
                    // (CHOLMOD_REAL, _COMPLEX, or _ZOMPLEX)
    cholmod_common *Common
) ;
cholmod_dense *cholmod_l_ensure_dense (cholmod_dense **, size_t, size_t,
    size_t, int, cholmod_common *) ;
```

Purpose: Ensures a dense matrix has a given size and type.

16.5 cholmod_zeros: dense zero matrix

```
cholmod_dense *cholmod_zeros
(
    size_t nrow,     // # of rows
    size_t ncol,     // # of columns
    int xdtype,      // xtype + dtype of the matrix:
                     // (CHOLMOD_DOUBLE, _SINGLE) +
                     // (CHOLMOD_REAL, _COMPLEX, or _ZOMPLEX)
    cholmod_common *Common
) ;
cholmod_dense *cholmod_l_zeros (size_t, size_t, int, cholmod_common *) ;
```

Purpose: Returns an all-zero dense matrix.

16.6 cholmod_ones: dense matrix, all ones

```
cholmod_dense *cholmod_ones
(
    size_t nrow,     // # of rows
    size_t ncol,     // # of columns
    int xdtype,      // xtype + dtype of the matrix:
                     // (CHOLMOD_DOUBLE, _SINGLE) +
                     // (_REAL, _COMPLEX, or _ZOMPLEX)
    cholmod_common *Common
) ;
cholmod_dense *cholmod_l_ones (size_t, size_t, int, cholmod_common *) ;
```

Purpose: Returns a dense matrix with each entry equal to one.

16.7 cholmod_eye: dense identity matrix

```
cholmod_dense *cholmod_eye      // return a dense identity matrix
(
    size_t nrow,     // # of rows
    size_t ncol,     // # of columns
    int xdtype,      // xtype + dtype of the matrix:
                     // (CHOLMOD_DOUBLE, _SINGLE) +
                     // (_REAL, _COMPLEX, or _ZOMPLEX)
    cholmod_common *Common
) ;
cholmod_dense *cholmod_l_eye (size_t, size_t, int, cholmod_common *) ;
```

Purpose: Returns a dense identity matrix.

16.8 `cholmod_sparse_to_dense`: dense matrix copy of a sparse matrix

```
cholmod_dense *cholmod_sparse_to_dense      // return a dense matrix
(
    cholmod_sparse *A,      // input matrix
    cholmod_common *Common
) ;
cholmod_dense *cholmod_l_sparse_to_dense (cholmod_sparse *, cholmod_common *) ;
```

Purpose: Returns a dense copy of a sparse matrix.

16.9 `cholmod_dense_nnz`: number of nonzeros in a dense matrix

```
int64_t cholmod_dense_nnz       // return # of entries in the dense matrix
(
    cholmod_dense *X,       // input matrix
    cholmod_common *Common
) ;
int64_t cholmod_l_dense_nnz (cholmod_dense *, cholmod_common *) ;
```

Purpose: Returns a count of the number of nonzero entries in a dense matrix.

16.10 `cholmod_dense_to_sparse`: sparse matrix copy of a dense matrix

```
cholmod_sparse *cholmod_dense_to_sparse        // return a sparse matrix C
(
    cholmod_dense *X,       // input matrix
    int values,             // if true, copy the values; if false, C is pattern
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_dense_to_sparse (cholmod_dense *, int,
    cholmod_common *) ;
```

Purpose: Returns a sparse copy of a dense matrix.

16.11 `cholmod_copy_dense`: copy dense matrix

```
cholmod_dense *cholmod_copy_dense   // returns new dense matrix
(
    cholmod_dense *X,   // input dense matrix
    cholmod_common *Common
) ;
cholmod_dense *cholmod_l_copy_dense (cholmod_dense *, cholmod_common *) ;
```

Purpose: Returns a copy of a dense matrix.

16.12 cholmod_copy_dense2: copy dense matrix (preallocated)

```
int cholmod_copy_dense2
(
    cholmod_dense *X,   // input dense matrix
    cholmod_dense *Y,   // output dense matrix (already allocated on input)
    cholmod_common *Common
) ;
int cholmod_l_copy_dense2 (cholmod_dense *, cholmod_dense *, cholmod_common *) ;
```

Purpose: Returns a copy of a dense matrix, placing the result in a preallocated matrix Y.

16.13 cholmod_dense_xtype: change dense matrix xtype

```
int cholmod_dense_xtype
(
    int to_xdtype,      // requested xtype and dtype
    cholmod_dense *X,   // dense matrix to change
    cholmod_common *Common
) ;
int cholmod_l_dense_xtype (int, cholmod_dense *, cholmod_common *) ;
```

Purpose: Changes the xtype of a dense matrix, to real, complex, or zomplex. Changing from complex or zomplex to real discards the imaginary part.

17 Utility Module: cholmod_triplet object

17.1 cholmod_triplet object: sparse matrix in triplet form

```
typedef struct cholmod_triplet_struct
{
    size_t nrow ;   // # of rows of the matrix
    size_t ncol ;   // # of colums of the matrix
    size_t nzmax ;  // max # of entries that can be held in the matrix
    size_t nnz ;    // current # of entries can be held in the matrix

    // int32 or int64 arrays (depending on T->itype)
    void *i ;   // i [0..nzmax-1], the row indices
    void *j ;   // j [0..nzmax-1], the column indices

    // double or float arrays:
    void *x ;   // size nzmax or 2*nzmax, or NULL
    void *z ;   // size nzmax, or NULL

    int stype ; // T->stype defines what parts of the matrix is held:
        // 0:  the matrix is unsymmetric with both lower and upper parts stored.
        // >0: the matrix is square and symmetric, where entries in the lower
        //      triangular part are transposed and placed in the upper
        //      triangular part of A if T is converted into a sparse matrix A.
        // <0: the matrix is square and symmetric, where entries in the upper
        //      triangular part are transposed and placed in the lower
        //      triangular part of A if T is converted into a sparse matrix A.
        //
        // Note that A->stype (for a sparse matrix) and T->stype (for a
        // triplet matrix) are handled differently.  In a triplet matrix T,
        // no entry is ever ignored.  For a sparse matrix A, if A->stype < 0
        // or A->stype > 0, then entries not in the correct triangular part
        // are ignored.

    int itype ; // T->itype defines the integers used for T->i and T->j.
        // if CHOLMOD_INT, these arrays are all of type int32_t.
        // if CHOLMOD_LONG, these arrays are all of type int64_t.
    int xtype ;     // pattern, real, complex, or zomplex
    int dtype ;     // x and z are double or single

} cholmod_triplet ;
```

Purpose: Contains a sparse matrix in triplet form.

17.2 cholmod_allocate_triplet: allocate triplet matrix

```
cholmod_triplet *cholmod_allocate_triplet      // return triplet matrix T
(
    size_t nrow,    // # of rows
    size_t ncol,    // # of columns
    size_t nzmax,   // max # of entries the matrix can hold
    int stype,      // the stype of the matrix (unsym, tril, or triu)
    int xdtype,     // xtype + dtype of the matrix:
```

```
                    // (CHOLMOD_DOUBLE, _SINGLE) +
                    // (CHOLMOD_PATTERN, _REAL, _COMPLEX, or _ZOMPLEX)
    cholmod_common *Common
) ;
cholmod_triplet *cholmod_l_allocate_triplet (size_t, size_t, size_t, int, int,
    cholmod_common *) ;
```

Purpose: Allocates a triplet matrix.

17.3 `cholmod_free_triplet`: free triplet matrix

```
int cholmod_free_triplet
(
    cholmod_triplet **T,        // handle of triplet matrix to free
    cholmod_common *Common
) ;
int cholmod_l_free_triplet (cholmod_triplet **, cholmod_common *) ;
```

Purpose: Frees a triplet matrix.

17.4 `cholmod_reallocate_triplet`: reallocate triplet matrix

```
int cholmod_reallocate_triplet
(
    size_t nznew,       // new max # of nonzeros the triplet matrix can hold
    cholmod_triplet *T, // triplet matrix to reallocate
    cholmod_common *Common
) ;
int cholmod_l_reallocate_triplet (size_t, cholmod_triplet *, cholmod_common *) ;
```

Purpose: Reallocates a triplet matrix so that it can hold `nznew` entries.

17.5 `cholmod_sparse_to_triplet`: triplet matrix copy of a sparse matrix

```
cholmod_triplet *cholmod_sparse_to_triplet
(
    cholmod_sparse *A,      // matrix to copy into triplet form T
    cholmod_common *Common
) ;
cholmod_triplet *cholmod_l_sparse_to_triplet (cholmod_sparse *,
    cholmod_common *) ;
```

Purpose: Returns a triplet matrix copy of a sparse matrix.

17.6 `cholmod_triplet_to_sparse`: sparse matrix copy of a triplet matrix

```
cholmod_sparse *cholmod_triplet_to_sparse      // return sparse matrix A
(
    cholmod_triplet *T,     // input triplet matrix
    size_t nzmax,           // allocate space for max(nzmax,nnz(A)) entries
    cholmod_common *Common
) ;
cholmod_sparse *cholmod_l_triplet_to_sparse (cholmod_triplet *, size_t,
    cholmod_common *) ;
```

Purpose: Returns a sparse matrix copy of a triplet matrix. If the triplet matrix is symmetric with just the lower part present (`T->stype` < 0), then entries in the upper part are transposed and placed in the lower part when converting to a sparse matrix. Similarly, if the triplet matrix is symmetric with just the upper part present (`T->stype` > 0), then entries in the lower part are transposed and placed in the upper part when converting to a sparse matrix. Any duplicate entries are summed.

17.7 `cholmod_copy_triplet`: copy triplet matrix

```
cholmod_triplet *cholmod_copy_triplet   // return new triplet matrix
(
    cholmod_triplet *T,     // triplet matrix to copy
    cholmod_common *Common
) ;
cholmod_triplet *cholmod_l_copy_triplet (cholmod_triplet *, cholmod_common *) ;
```

Purpose: Returns an exact copy of a triplet matrix.

17.8 `cholmod_triplet_xtype`: change triplet xtype

```
int cholmod_triplet_xtype
(
    int to_xdtype,      // requested xtype and dtype
    cholmod_triplet *T, // triplet matrix to change
    cholmod_common *Common
) ;
int cholmod_l_triplet_xtype (int, cholmod_triplet *, cholmod_common *) ;
```

Purpose: Changes the `xtype` of a dense matrix, to real, complex, or zomplex. Changing from complex or zomplex to real discards the imaginary part.

18 Utility Module: memory management

18.1 cholmod_malloc: allocate memory

```
void *cholmod_malloc    // return pointer to newly allocated memory
(
    size_t n,           // number of items
    size_t size,        // size of each item
    cholmod_common *Common
) ;
void *cholmod_l_malloc (size_t, size_t, cholmod_common *) ;
```

Purpose: Allocates a block of memory of size n*size, using the SuiteSparse_config.malloc_func function pointer (default is to use the ANSI C malloc routine). A value of n=0 is treated as n=1. If not successful, NULL is returned and Common->status is set to CHOLMOD_OUT_OF_MEMORY.

18.2 cholmod_calloc: allocate and clear memory

```
void *cholmod_calloc    // return pointer to newly allocated memory
(
    size_t n,           // number of items
    size_t size,        // size of each item
    cholmod_common *Common
) ;
void *cholmod_l_calloc (size_t, size_t, cholmod_common *) ;
```

Purpose: Allocates a block of memory of size n*size, using the SuiteSparse_config.calloc_func function pointer (default is to use the ANSI C calloc routine). A value of n=0 is treated as n=1. If not successful, NULL is returned and Common->status is set to CHOLMOD_OUT_OF_MEMORY.

18.3 `cholmod_free`: free memory

```
void *cholmod_free      // returns NULL to simplify its usage
(
    size_t n,           // number of items
    size_t size,        // size of each item
    void *p,            // memory to free
    cholmod_common *Common
) ;
void *cholmod_l_free (size_t, size_t, void *, cholmod_common *) ;
```

Purpose: Frees a block of memory of size `n*size`, using the `SuiteSparse_config.free_func` function pointer (default is to use the ANSI C `free` routine). The size of the block (`n` and `size`) is only required so that CHOLMOD can keep track of its current and peak memory usage. This is a useful statistic, and it can also help in tracking down memory leaks. After the call to `cholmod_finish`, the count of allocated blocks (`Common->malloc_count`) should be zero, and the count of bytes in use (`Common->memory_inuse`) also should be zero. If you allocate a block with one size and free it with another, the `Common->memory_inuse` count will be wrong, but CHOLMOD will not have a memory leak.

18.4 `cholmod_realloc`: reallocate memory

```
void *cholmod_realloc   // return newly reallocated block of memory
(
    size_t nnew,        // # of items in newly reallocate memory
    size_t size,        // size of each item
    void *p,            // pointer to memory to reallocate (may be NULL)
    size_t *n,          // # of items in p on input; nnew on output if success
    cholmod_common *Common
) ;
void *cholmod_l_realloc (size_t, size_t, void *, size_t *, cholmod_common *) ;
```

Purpose: Reallocates a block of memory whose current size `n*size`, and whose new size will be `nnew*size` if successful, using the `SuiteSparse_config.calloc_func` function pointer (default is to use the ANSI C `realloc` routine). If the reallocation is not successful, `p` is returned unchanged and `Common->status` is set to `CHOLMOD_OUT_OF_MEMORY`. The value of `n` is set to `nnew` if successful, or left unchanged otherwise. A value of `nnew=0` is treated as `nnew=1`.

18.5 cholmod_realloc_multiple: reallocate memory

```
int cholmod_realloc_multiple    // returns true if successful, false otherwise
(
    size_t nnew,    // # of items in newly reallocate memory
    int nint,       // 0: do not allocate I_block or J_block, 1: just I_block,
                    // 2: both I_block and J_block
    int xdtype,     // xtype + dtype of the matrix:
                    // (CHOLMOD_DOUBLE, _SINGLE) +
                    // (CHOLMOD_PATTERN, _REAL, _COMPLEX, or _ZOMPLEX)
    // input/output:
    void **I_block,       // integer block of memory (int32_t or int64_t)
    void **J_block,       // integer block of memory (int32_t or int64_t)
    void **X_block,       // real or complex, double or single, block
    void **Z_block,       // zomplex only: double or single block
    size_t *n,      // current size of I_block, J_block, X_block, and/or Z_block
                    // on input,  changed to nnew on output, if successful
    cholmod_common *Common
) ;
int cholmod_l_realloc_multiple (size_t, int, int, void **, void **, void **,
    void **, size_t *, cholmod_common *) ;
```

Purpose: Reallocates multiple blocks of memory, all with the same number of items (but with different item sizes). Either all reallocations succeed, or all are returned to their original size.

19 Utility Module: version control

19.1 cholmod_version: return current CHOLMOD version

```
int cholmod_version     // returns CHOLMOD_VERSION, defined above
(
    // if version is not NULL, then cholmod_version returns its contents as:
    // version [0] = CHOLMOD_MAIN_VERSION
    // version [1] = CHOLMOD_SUB_VERSION
    // version [2] = CHOLMOD_SUBSUB_VERSION
    int version [3]
) ;
int cholmod_l_version (int version [3]) ;
```

Purpose: Returns the CHOLMOD version number, so that it can be tested at run time, even if the caller does not have access to the CHOLMOD include files. For example, for a CHOLMOD version 3.2.1, the `version` array will contain 3, 2, and 1, in that order. This function appears in CHOLMOD 2.1.1 and later. You can check if the function exists with the `CHOLMOD_HAS_VERSION_FUNCTION` macro, so that the following code fragment works in any version of CHOLMOD:

```
#ifdef CHOLMOD_HAS_VERSION_FUNCTION
v = cholmod_version (NULL) ;
#else
v = CHOLMOD_VERSION ;
#endif
```

Note that `cholmod_version` and `cholmod_l_version` have identical prototypes. Both use `int`'s. Unlike all other CHOLMOD functions, this function does not take the `Common` object as an input parameter, and it does not use any definitions from any include files. Thus, the caller can access this function even if the caller does not include any CHOLMOD include files.

The above code fragment does require the `#include "cholmod.h"`, of course, but `cholmod_version` can be called without it, if necessary.

20 Check Module routines

No CHOLMOD routines print anything, except for the cholmod_print_* routines in the Check Module, and the cholmod_error routine. The SuiteSparse_config.printf_function is a pointer to printf by default; you can redirect the output of CHOLMOD by redefining this pointer. If the function pointer is NULL, CHOLMOD does not print anything.

The Common->print parameter determines how much detail is printed. Each value of Common->print listed below also prints the items listed for smaller values of Common->print:

- 0: print nothing; check the data structures and return TRUE or FALSE.
- 1: print error messages.
- 2: print warning messages.
- 3: print a one-line summary of the object.
- 4: print a short summary of the object (first and last few entries).
- 5: print the entire contents of the object.

Values less than zero are treated as zero, and values greater than five are treated as five.

20.1 cholmod_check_common: check Common object

```
int cholmod_check_common
(
    cholmod_common *Common
) ;

int cholmod_l_check_common (cholmod_common *) ;
```

Purpose: Check if the Common object is valid.

20.2 cholmod_print_common: print Common object

```
int cholmod_print_common
(
    /* ---- input ---- */
    const char *name,   /* printed name of Common object */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_print_common (const char *, cholmod_common *) ;
```

Purpose: Print the Common object and check if it is valid. This prints the CHOLMOD parameters and statistics.

20.3 cholmod_check_sparse: check sparse matrix

```
int cholmod_check_sparse
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* sparse matrix to check */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_check_sparse (cholmod_sparse *, cholmod_common *) ;
```

Purpose: Check if a sparse matrix is valid.

20.4 cholmod_print_sparse: print sparse matrix

```
int cholmod_print_sparse
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* sparse matrix to print */
    const char *name,   /* printed name of sparse matrix */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_print_sparse (cholmod_sparse *, const char *, cholmod_common *) ;
```

Purpose: Print a sparse matrix and check if it is valid.

20.5 cholmod_check_dense: check dense matrix

```
int cholmod_check_dense
(
    /* ---- input ---- */
    cholmod_dense *X,   /* dense matrix to check */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_check_dense (cholmod_dense *, cholmod_common *) ;
```

Purpose: Check if a dense matrix is valid.

20.6 cholmod_print_dense: print dense matrix

```
int cholmod_print_dense
(
    /* ---- input ---- */
    cholmod_dense *X,   /* dense matrix to print */
    const char *name,   /* printed name of dense matrix */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_print_dense (cholmod_dense *, const char *, cholmod_common *) ;
```

Purpose: Print a dense matrix and check if it is valid.

20.7 `cholmod_check_factor`: check factor

```
int cholmod_check_factor
(
    /* ---- input ---- */
    cholmod_factor *L,  /* factor to check */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_check_factor (cholmod_factor *, cholmod_common *) ;
```

Purpose: Check if a factor is valid.

20.8 `cholmod_print_factor`: print factor

```
int cholmod_print_factor
(
    /* ---- input ---- */
    cholmod_factor *L,  /* factor to print */
    const char *name,   /* printed name of factor */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_print_factor (cholmod_factor *, const char *, cholmod_common *) ;
```

Purpose: Print a factor and check if it is valid.

20.9 cholmod_check_triplet: check triplet matrix

```
int cholmod_check_triplet
(
    /* ---- input ---- */
    cholmod_triplet *T, /* triplet matrix to check */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_check_triplet (cholmod_triplet *, cholmod_common *) ;
```

Purpose: Check if a triplet matrix is valid.

20.10 cholmod_print_triplet: print triplet matrix

```
int cholmod_print_triplet
(
    /* ---- input ---- */
    cholmod_triplet *T, /* triplet matrix to print */
    const char *name,   /* printed name of triplet matrix */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_print_triplet (cholmod_triplet *, const char *,
    cholmod_common *) ;
```

Purpose: Print a triplet matrix and check if it is valid.

20.11 `cholmod_check_subset`: check subset

```
int cholmod_check_subset
(
    /* ---- input ---- */
    int32_t *Set,       /* Set [0:len-1] is a subset of 0:n-1.  Duplicates OK */
    int64_t len,        /* size of Set (an integer array) */
    size_t n,           /* 0:n-1 is valid range */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_check_subset (int64_t *, int64_t, size_t,
    cholmod_common *) ;
```

Purpose: Check if a subset is valid.

20.12 `cholmod_print_subset`: print subset

```
int cholmod_print_subset
(
    /* ---- input ---- */
    int32_t *Set,       /* Set [0:len-1] is a subset of 0:n-1.  Duplicates OK */
    int64_t len,        /* size of Set (an integer array) */
    size_t n,           /* 0:n-1 is valid range */
    const char *name,   /* printed name of Set */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_print_subset (int64_t *, int64_t, size_t,
    const char *, cholmod_common *) ;
```

Purpose: Print a subset and check if it is valid.

20.13 cholmod_check_perm: check permutation

```
int cholmod_check_perm
(
    /* ---- input ---- */
    int32_t *Perm,      /* Perm [0:len-1] is a permutation of subset of 0:n-1 */
    size_t len,         /* size of Perm (an integer array) */
    size_t n,           /* 0:n-1 is valid range */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_check_perm (int64_t *, size_t, size_t, cholmod_common *);

/* ------------------------------------------------------------------------ */
/* cholmod_print_perm:  print a permutation vector */
/* ------------------------------------------------------------------------ */

int cholmod_print_perm
(
    /* ---- input ---- */
    int32_t *Perm,      /* Perm [0:len-1] is a permutation of subset of 0:n-1 */
    size_t len,         /* size of Perm (an integer array) */
    size_t n,           /* 0:n-1 is valid range */
    const char *name,   /* printed name of Perm */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_print_perm (int64_t *, size_t, size_t, const char *,
    cholmod_common *) ;
```

Purpose: Check if a permutation is valid.

20.14 cholmod_print_perm: print permutation

```
int cholmod_print_perm
(
    /* ---- input ---- */
    int32_t *Perm,      /* Perm [0:len-1] is a permutation of subset of 0:n-1 */
    size_t len,         /* size of Perm (an integer array) */
    size_t n,           /* 0:n-1 is valid range */
    const char *name,   /* printed name of Perm */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_print_perm (int64_t *, size_t, size_t, const char *,
    cholmod_common *) ;
```

Purpose: Print a permutation and check if it is valid.

20.15 `cholmod_check_parent`: check elimination tree

```
int cholmod_check_parent
(
    /* ---- input ---- */
    int32_t *Parent,    /* Parent [0:n-1] is an elimination tree */
    size_t n,           /* size of Parent */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_check_parent (int64_t *, size_t, cholmod_common *) ;
```

Purpose: Check if an elimination tree is valid.

20.16 `cholmod_print_parent`: print elimination tree

```
int cholmod_print_parent
(
    /* ---- input ---- */
    int32_t *Parent,    /* Parent [0:n-1] is an elimination tree */
    size_t n,           /* size of Parent */
    const char *name,   /* printed name of Parent */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_print_parent (int64_t *, size_t, const char *,
    cholmod_common *) ;
```

Purpose: Print an elimination tree and check if it is valid.

20.17 cholmod_read_triplet: read triplet matrix from file

```
cholmod_triplet *cholmod_read_triplet
(
    /* ---- input ---- */
    FILE *f,            /* file to read from, must already be open */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_triplet *cholmod_l_read_triplet (FILE *, cholmod_common *) ;
```

Purpose: Read a sparse matrix in triplet form, using the the `coord` Matrix Market format (http://www.nist.gov/MatrixMarket). Skew-symmetric and complex symmetric matrices are returned with both upper and lower triangular parts present (an stype of zero). Real symmetric and complex Hermitian matrices are returned with just their upper or lower triangular part, depending on their stype. The Matrix Market `array` data type for dense matrices is not supported (use `cholmod_read_dense` for that case).

If the first line of the file starts with `%%MatrixMarket`, then it is interpreted as a file in Matrix Market format. The header line is optional. If present, this line must have the following format:

`%%MatrixMarket matrix coord` *type storage*

where *type* is one of: `real`, `complex`, `pattern`, or `integer`, and *storage* is one of: `general`, `hermitian`, `symmetric`, or `skew-symmetric`. In CHOLMOD, these roughly correspond to the `xtype` (pattern, real, complex, or zomplex) and `stype` (unsymmetric, symmetric/upper, and symmetric/lower). The strings are case-insensitive. Only the first character (or the first two for skew-symmetric) is significant. The `coord` token can be replaced with `array` in the Matrix Market format, but this format not supported by `cholmod_read_triplet`. The `integer` type is converted to real. The *type* is ignored; the actual type (real, complex, or pattern) is inferred from the number of tokens in each line of the file (2: pattern, 3: real, 4: complex). This is compatible with the Matrix Market format.

A storage of `general` implies an stype of zero (see below). A storage of `symmetric` and `hermitian` imply an stype of -1. Skew-symmetric and complex symmetric matrices are returned with an stype of 0. Blank lines, any other lines starting with "`%`" are treated as comments, and are ignored.

The first non-comment line contains 3 or 4 integers:

nrow ncol nnz stype

where *stype* is optional (stype does not appear in the Matrix Market format). The matrix is *nrow*-by-*ncol*. The following *nnz* lines (excluding comments) each contain a single entry. Duplicates are permitted, and are summed in the output matrix.

If stype is present, it denotes the storage format for the matrix.

- stype = 0 denotes an unsymmetric matrix (same as Matrix Market `general`).
- stype = -1 denotes a symmetric or Hermitian matrix whose lower triangular entries are stored. Entries may be present in the upper triangular part, but these are ignored (same as Matrix Market `symmetric` for the real case, `hermitian` for the complex case).

- stype = 1 denotes a symmetric or Hermitian matrix whose upper triangular entries are stored. Entries may be present in the lower triangular part, but these are ignored. This format is not available in the Matrix Market format.

If neither the stype nor the Matrix Market header are present, then the stype is inferred from the rest of the data. If the matrix is rectangular, or has entries in both the upper and lower triangular parts, then it is assumed to be unsymmetric (stype=0). If only entries in the lower triangular part are present, the matrix is assumed to have stype = -1. If only entries in the upper triangular part are present, the matrix is assumed to have stype = 1.

Each nonzero consists of one line with 2, 3, or 4 entries. All lines must have the same number of entries. The first two entries are the row and column indices of the nonzero. If 3 entries are present, the 3rd entry is the numerical value, and the matrix is real. If 4 entries are present, the 3rd and 4th entries in the line are the real and imaginary parts of a complex value.

The matrix can be either 0-based or 1-based. It is first assumed to be one-based (compatible with Matrix Market), with row indices in the range 1 to ncol and column indices in the range 1 to nrow. If a row or column index of zero is found, the matrix is assumed to be zero-based (with row indices in the range 0 to ncol-1 and column indices in the range 0 to nrow-1). This test correctly determines that all Matrix Market matrices are in 1-based form.

For symmetric pattern-only matrices, the kth diagonal (if present) is set to one plus the degree of the row k or column k (whichever is larger), and the off-diagonals are set to -1. A symmetric pattern-only matrix with a zero-free diagonal is thus converted into a symmetric positive definite matrix. All entries are set to one for an unsymmetric pattern-only matrix. This differs from the MatrixMarket format (`A = mmread ('file')` returns a binary pattern for A for symmetric pattern-only matrices). To return a binary format for all pattern-only matrices, use `A = mread('file',1)`.

Example matrices that follow this format can be found in the `CHOLMOD/Demo/Matrix` and `CHOLMOD/Tcov/Matrix` directories. You can also try any of the matrices in the Matrix Market collection at http://www.nist.gov/MatrixMarket.

20.18 `cholmod_read_sparse`: read sparse matrix from file

```
cholmod_sparse *cholmod_read_sparse
(
    /* ---- input ---- */
    FILE *f,            /* file to read from, must already be open */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_sparse *cholmod_l_read_sparse (FILE *, cholmod_common *) ;
```

Purpose: Read a sparse matrix in triplet form from a file (using `cholmod_read_triplet`) and convert to a CHOLMOD sparse matrix. The Matrix Market format is used. If `Common->prefer_upper` is `TRUE` (the default case), a symmetric matrix is returned stored in upper-triangular form (`A->stype` is 1). Otherwise, it is left in its original form, either upper or lower.

20.19 `cholmod_read_dense`: read dense matrix from file

```
cholmod_dense *cholmod_read_dense
(
    /* ---- input ---- */
    FILE *f,            /* file to read from, must already be open */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_dense *cholmod_l_read_dense (FILE *, cholmod_common *) ;
```

Purpose: Read a dense matrix from a file, using the the `array` Matrix Market format (http://www.nist.gov/MatrixMarket).

20.20 `cholmod_read_matrix`: read a matrix from file

```
void *cholmod_read_matrix
(
    /* ---- input ---- */
    FILE *f,            /* file to read from, must already be open */
    int prefer,         /* If 0, a sparse matrix is always return as a
                         *      cholmod_triplet form.  It can have any stype
                         *      (symmetric-lower, unsymmetric, or
                         *      symmetric-upper).
                         * If 1, a sparse matrix is returned as an unsymmetric
                         *      cholmod_sparse form (A->stype == 0), with both
                         *      upper and lower triangular parts present.
                         *      This is what the MATLAB mread mexFunction does,
                         *      since MATLAB does not have an stype.
                         * If 2, a sparse matrix is returned with an stype of 0
                         *      or 1 (unsymmetric, or symmetric with upper part
                         *      stored).
                         * This argument has no effect for dense matrices.
                         */
    /* ---- output---- */
    int *mtype,         /* CHOLMOD_TRIPLET, CHOLMOD_SPARSE or CHOLMOD_DENSE */
    /* --------------- */
    cholmod_common *Common
) ;

void *cholmod_l_read_matrix (FILE *, int, int *, cholmod_common *) ;
```

Purpose: Read a sparse or dense matrix from a file, in Matrix Market format. Returns a `void` pointer to either a `cholmod_triplet`, `cholmod_sparse`, or `cholmod_dense` object.

20.21 cholmod_write_sparse: write a sparse matrix to a file

```
int cholmod_write_sparse    // returns the same result as cholmod_symmetry
(
    /* ---- input ---- */
    FILE *f,                /* file to write to, must already be open */
    cholmod_sparse *A,      /* matrix to print */
    cholmod_sparse *Z,      /* optional matrix with pattern of explicit zeros */
    const char *comments,   /* optional filename of comments to include */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_write_sparse (FILE *, cholmod_sparse *, cholmod_sparse *,
    const char *c, cholmod_common *) ;
```

Purpose: Write a sparse matrix to a file in Matrix Market format. Optionally include comments, and print explicit zero entries given by the pattern of the `Z` matrix. If not NULL, the `Z` matrix must have the same dimensions and stype as `A`.

Returns the symmetry in which the matrix was printed (1 to 7) or -1 on failure. See the `cholmod_symmetry` function for a description of the return codes.

If `A` and `Z` are sorted on input, and either unsymmetric (stype = 0) or symmetric-lower (stype ¡ 0), and if `A` and `Z` do not overlap, then the triplets are sorted, first by column and then by row index within each column, with no duplicate entries. If all the above holds except stype ¿ 0, then the triplets are sorted by row first and then column.

20.22 cholmod_write_dense: write a dense matrix to a file

```
int cholmod_write_dense
(
    /* ---- input ---- */
    FILE *f,                /* file to write to, must already be open */
    cholmod_dense *X,       /* matrix to print */
    const char *comments,   /* optional filename of comments to include */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_write_dense (FILE *, cholmod_dense *, const char *,
    cholmod_common *) ;
```

Purpose: Write a dense matrix to a file in Matrix Market format. Optionally include comments. Returns ¿ 0 if successful, -1 otherwise (1 if rectangular, 2 if square). A dense matrix is written in "general" format; symmetric formats in the Matrix Market standard are not exploited.

21 Cholesky Module routines

21.1 cholmod_analyze: symbolic factorization

```
cholmod_factor *cholmod_analyze     // order and analyze
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to order and analyze */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_factor *cholmod_l_analyze (cholmod_sparse *, cholmod_common *) ;
```

Purpose: Orders and analyzes a matrix (either simplicial or supernodal), in preparation for numerical factorization via `cholmod_factorize` or via the "expert" routines `cholmod_rowfac` and `cholmod_super_numeric`.

In the symmetric case, `A` or `A(p,p)` is analyzed, where `p` is the fill-reducing ordering. In the unsymmetric case, `A*A'` or `A(p,:)*A(p,:)'` is analyzed. The `cholmod_analyze_p` routine can be given a user-provided permutation `p` (see below).

The default ordering strategy is to first try AMD. The ordering quality is analyzed, and if AMD obtains an ordering where `nnz(L)` is greater than or equal to `5*nnz(tril(A))` (or `5*nnz(tril(A*A'))` if `A` is unsymmetric) and the floating-point operation count for the subsequent factorization is greater than or equal to `500*nnz(L)`, then METIS is tried (if installed). For `cholmod_analyze_p`, the user-provided ordering is also tried. This default behavior is obtained when `Common->nmethods` is zero. In this case, methods 0, 1, and 2 in `Common->method[...]` are reset to user-provided, AMD, and METIS, respectively. The ordering with the smallest `nnz(L)` is kept.

If `Common->default_nesdis` is true (nonzero), then CHOLMOD's nested dissection (NESDIS) is used for the default strategy described above, in place of METIS.

Other ordering options can be requested. These include:

1. natural: A is not permuted to reduce fill-in.
2. user-provided: a permutation can be provided to `cholmod_analyze_p`.
3. AMD: approximate minimum degree (AMD for the symmetric case, COLAMD for the `A*A'` case).
4. METIS: nested dissection with `METIS_NodeND`
5. `NESDIS`: CHOLMOD's nested dissection using `METIS_NodeComputeSeparator`, followed by a constrained minimum degree (CAMD or CSYMAMD for the symmetric case, CCOLAMD for the `A*A'` case). This is typically slower than METIS, but typically provides better orderings.

Multiple ordering options can be tried (up to 9 of them), and the best one is selected (the one that gives the smallest number of nonzeros in the simplicial factor L). If one method fails, `cholmod_analyze` keeps going, and picks the best among the methods that succeeded. This routine fails (and returns `NULL`) if either the initial memory allocation fails, all ordering methods fail, or the supernodal analysis (if requested) fails. Change `Common->nmethods` to the number of methods you wish to try. By default, the 9 methods available are:

1. user-provided permutation (only for cholmod_analyze_p).

2. AMD with default parameters.

3. METIS with default parameters.

4. NESDIS with default parameters: stopping the partitioning when the graph is of size nd_small = 200 or less, remove nodes with more than max (16, prune_dense * sqrt (n)) nodes where prune_dense = 10, and follow partitioning with constrained minimum degree ordering (CAMD for the symmetric case, CCOLAMD for the unsymmetric case).

5. natural ordering (with weighted postorder).

6. NESDIS, nd_small = 20000, prune_dense = 10.

7. NESDIS, nd_small = 4, prune_dense = 10, no constrained minimum degree.

8. NESDIS, nd_small = 200, prune_dense = 0.

9. COLAMD for A*A' or AMD for A

You can modify these 9 methods and the number of methods tried by changing parameters in the Common argument. If you know the best ordering for your matrix, set Common->nmethods to 1 and set Common->method[0].ordering to the requested ordering method. Parameters for each method can also be modified (refer to the description of cholmod_common for details).

Note that it is possible for METIS to terminate your program if it runs out of memory. This is not the case for any CHOLMOD or minimum degree ordering routine (AMD, COLAMD, CAMD, CCOLAMD, or CSYMAMD). Since NESDIS relies on METIS, it too can terminate your program.

The selected ordering is followed by a weighted postorder of the elimination tree by default (see cholmod_postorder for details), unless Common->postorder is set to FALSE. The postorder does not change the number of nonzeros in $\mathbf{L}$ or the floating-point operation count. It does improve performance, particularly for the supernodal factorization. If you truly want the natural ordering with no postordering, you must set Common->postorder to FALSE.

The factor L is returned as simplicial symbolic if Common->supernodal is CHOLMOD_SIMPLICIAL (zero) or as supernodal symbolic if Common->supernodal is CHOLMOD_SUPERNODAL (two). If Common->supernodal is CHOLMOD_AUTO (one), then L is simplicial if the flop count per nonzero in L is less than Common->supernodal_switch (default: 40), and supernodal otherwise. In both cases, L->xtype is CHOLMOD_PATTERN. A subsequent call to cholmod_factorize will perform a simplicial or supernodal factorization, depending on the type of L.

For the simplicial case, L contains the fill-reducing permutation (L->Perm) and the counts of nonzeros in each column of L (L->ColCount). For the supernodal case, L also contains the nonzero pattern of each supernode.

If a simplicial factorization is selected, it will be $\mathbf{LDL}^\mathsf{T}$ by default, since this is the kind required by the Modify Module. CHOLMOD does not include a supernodal $\mathbf{LDL}^\mathsf{T}$ factorization, so if a supernodal factorization is selected, it will be in the form $\mathbf{LL}^\mathsf{T}$. The $\mathbf{LDL}^\mathsf{T}$ method can be used to factorize positive definite matrices and indefinite matrices whose leading minors are well-conditioned (2-by-2 pivoting is not supported). The $\mathbf{LL}^\mathsf{T}$ method is restricted to positive definite matrices. To factorize a large indefinite matrix, set Common->supernodal to CHOLMOD_SIMPLICIAL, and the simplicial $\mathbf{LDL}^\mathsf{T}$ method will always be used. This will be significantly slower than a supernodal $\mathbf{LL}^\mathsf{T}$ factorization, however.

Refer to `cholmod_transpose_unsym` for a description of `f`.

21.2 `cholmod_factorize`: numeric factorization

```
int cholmod_factorize       // simplicial or superodal Cholesky factorization
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to factorize */
    /* ---- in/out --- */
    cholmod_factor *L,  /* resulting factorization */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_factorize (cholmod_sparse *, cholmod_factor *, cholmod_common *) ;
```

Purpose: Computes the numerical factorization of a symmetric matrix. The inputs to this routine are a sparse matrix `A` and the symbolic factor `L` from `cholmod_analyze` or a prior numerical factor `L`. If `A` is symmetric, this routine factorizes `A(p,p)`. where p is the fill-reducing permutation (`L->Perm`). If `A` is unsymmetric, `A(p,:)*A(p,:)'` is factorized. The nonzero pattern of the matrix `A` must be the same as the matrix passed to `cholmod_analyze` for the supernodal case. For the simplicial case, it can be different, but it should be the same for best performance.

A simplicial factorization or supernodal factorization is chosen, based on the type of the factor `L`. If `L->is_super` is `TRUE`, a supernodal $\mathbf{LL}^\mathsf{T}$ factorization is computed. Otherwise, a simplicial numeric factorization is computed, either $\mathbf{LL}^\mathsf{T}$ or $\mathbf{LDL}^\mathsf{T}$, depending on `Common->final_ll` (the default for the simplicial case is to compute an $\mathbf{LDL}^\mathsf{T}$ factorization).

Once the factorization is complete, it can be left as is or optionally converted into any simplicial numeric type, depending on the `Common->final_*` parameters. If converted from a supernodal to simplicial type, and `Common->final_resymbol` is `TRUE`, then numerically zero entries in `L` due to relaxed supernodal amalgamation are removed from the simplicial factor (they are always left in the supernodal form of `L`). Entries that are numerically zero but present in the simplicial symbolic pattern of `L` are left in place (the graph of `L` remains chordal). This is required for the update/downdate/rowadd/rowdel routines to work properly.

If the matrix is not positive definite the routine returns `TRUE`, but `Common->status` is set to `CHOLMOD_NOT_POSDEF` and `L->minor` is set to the column at which the failure occurred. Columns `L->minor` to `L->n-1` are set to zero.

Supports any xtype (pattern, real, complex, or zomplex), except that the input matrix `A` cannot be pattern-only. If `L` is simplicial, its numeric xtype matches `A` on output. If `L` is supernodal, its xtype is real if `A` is real, or complex if `A` is complex or zomplex. CHOLMOD does not provide a supernodal zomplex factor, since it is incompatible with how complex numbers are stored in LAPACK and the BLAS.

21.3 `cholmod_analyze_p`: symbolic factorization, given permutation

```
cholmod_factor *cholmod_analyze_p
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to order and analyze */
    int32_t *UserPerm,  /* user-provided permutation, size A->nrow */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_factor *cholmod_l_analyze_p (cholmod_sparse *, int64_t *,
    int64_t *, size_t, cholmod_common *) ;
cholmod_factor *cholmod_analyze_p2
(
    /* ---- input ---- */
    int for_whom,       /* FOR_SPQR     (0): for SPQR but not GPU-accelerated
                           FOR_CHOLESKY (1): for Cholesky (GPU or not)
                           FOR_SPQRGPU  (2): for SPQR with GPU acceleration */
    cholmod_sparse *A,  /* matrix to order and analyze */
    int32_t *UserPerm,  /* user-provided permutation, size A->nrow */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_factor *cholmod_l_analyze_p2 (int, cholmod_sparse *, int64_t *,
    int64_t *, size_t, cholmod_common *) ;
```

Purpose: Identical to `cholmod_analyze`, except that a user-provided permutation `p` can be provided, and the set `f` for the unsymmetric case can be provided. The matrices `A(:,f)*A(:,f)'` or `A(p,f)*A(p,f)'` can be analyzed in the the unsymmetric case.

21.4 `cholmod_factorize_p`: numeric factorization, given permutation

```
int cholmod_factorize_p
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to factorize */
    double beta [2],    /* factorize beta*I+A or beta*I+A'*A */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    /* ---- in/out --- */
    cholmod_factor *L,  /* resulting factorization */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_factorize_p (cholmod_sparse *, double *, int64_t *,
    size_t, cholmod_factor *, cholmod_common *) ;
```

Purpose: Identical to `cholmod_factorize`, but with additional options. The set `f` can be provided for the unsymmetric case; `A(p,f)*A(p,f)'` is factorized. The term `beta*I` can be added to the matrix before it is factorized, where `beta` is real. Only the real part, `beta[0]`, is used.

21.5 `cholmod_solve`: solve a linear system

```
cholmod_dense *cholmod_solve    /* returns the solution X */
(
    /* ---- input ---- */
    int sys,            /* system to solve */
    cholmod_factor *L,  /* factorization to use */
    cholmod_dense *B,   /* right-hand-side */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_dense *cholmod_l_solve (int, cholmod_factor *, cholmod_dense *,
    cholmod_common *) ;
```

Purpose: Returns a solution `X` that solves one of the following systems:

system	`sys` parameter	system	`sys` parameter
$\mathbf{Ax} = \mathbf{b}$	0: `CHOLMOD_A`		
$\mathbf{LDL}^\mathsf{T}\mathbf{x} = \mathbf{b}$	1: `CHOLMOD_LDLt`	$\mathbf{L}^\mathsf{T}\mathbf{x} = \mathbf{b}$	5: `CHOLMOD_Lt`
$\mathbf{LDx} = \mathbf{b}$	2: `CHOLMOD_LD`	$\mathbf{Dx} = \mathbf{b}$	6: `CHOLMOD_D`
$\mathbf{DL}^\mathsf{T}\mathbf{x} = \mathbf{b}$	3: `CHOLMOD_DLt`	$\mathbf{x} = \mathbf{Pb}$	7: `CHOLMOD_P`
$\mathbf{Lx} = \mathbf{b}$	4: `CHOLMOD_L`	$\mathbf{x} = \mathbf{P}^\mathsf{T}\mathbf{b}$	8: `CHOLMOD_Pt`

The factorization can be simplicial $\mathbf{LDL}^\mathsf{T}$, simplicial $\mathbf{LL}^\mathsf{T}$, or supernodal $\mathbf{LL}^\mathsf{T}$. For an $\mathbf{LL}^\mathsf{T}$ factorization, $\mathbf{D}$ is the identity matrix. Thus `CHOLMOD_LD` and `CHOLMOD_L` solve the same system if an $\mathbf{LL}^\mathsf{T}$ factorization was performed, for example. This is one of the few routines in CHOLMOD for which the xtype of the input arguments need not match. If both `L` and `B` are real, then `X` is returned real. If either is complex or zomplex, `X` is returned as either complex or zomplex, depending on the `Common->prefer_zomplex` parameter (default is complex).

This routine does not check to see if the diagonal of $\mathbf{L}$ or $\mathbf{D}$ is zero, because sometimes a partial solve can be done with an indefinite or singular matrix. If you wish to check in your own code, test `L->minor`. If `L->minor == L->n`, then the matrix has no zero diagonal entries. If `k = L->minor < L->n`, then `L(k,k)` is zero for an $\mathbf{LL}^\mathsf{T}$ factorization, or `D(k,k)` is zero for an $\mathbf{LDL}^\mathsf{T}$ factorization.

Iterative refinement is not performed, but this can be easily done with the `MatrixOps` Module. See `Demo/cholmod_demo.c` for an example.

21.6 `cholmod_spsolve`: solve a linear system

```
cholmod_sparse *cholmod_spsolve
(
    /* ---- input ---- */
    int sys,            /* system to solve */
    cholmod_factor *L,  /* factorization to use */
    cholmod_sparse *B,  /* right-hand-side */
    /* --------------- */
```

```
    cholmod_common *Common
) ;

cholmod_sparse *cholmod_l_spsolve (int, cholmod_factor *, cholmod_sparse *,
    cholmod_common *) ;
```

Purpose: Identical to `cholmod_solve`, except that `B` and `X` are sparse. This function converts `B` to full format, solves the system, and then converts `X` back to sparse. If you want to solve with a sparse `B` and get just a partial solution back in `X` (corresponding to the pattern of `B`), use `cholmod_solve2` below.

21.7 `cholmod_solve2`: solve a linear system, reusing workspace

```
int cholmod_solve2     /* returns TRUE on success, FALSE on failure */
(
    /* ---- input ---- */
    int sys,                        /* system to solve */
    cholmod_factor *L,              /* factorization to use */
    cholmod_dense *B,               /* right-hand-side */
    cholmod_sparse *Bset,
    /* ---- output --- */
    cholmod_dense **X_Handle,       /* solution, allocated if need be */
    cholmod_sparse **Xset_Handle,
    /* ---- workspace  */
    cholmod_dense **Y_Handle,       /* workspace, or NULL */
    cholmod_dense **E_Handle,       /* workspace, or NULL */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_solve2 (int, cholmod_factor *, cholmod_dense *, cholmod_sparse *,
    cholmod_dense **, cholmod_sparse **, cholmod_dense **, cholmod_dense **,
    cholmod_common *) ;
```

Purpose: Solve a linear system, optionally reusing workspace from a prior call to `cholmod_solve2`.

The inputs to this function are the same as `cholmod_solve`, with the addition of three parameters: `X`, `Y`, and `E`. The dense matrix `X` is the solution on output. On input, `&X` can point to a NULL matrix, or be the wrong size. If that is the case, it is freed and allocated to be the proper size. If `X` has the right size and type on input, then the allocation is skipped. In contrast, the `cholmod_solve` function always allocates its output `X`. This `cholmod_solve2` function allows you to reuse the memory space of a prior `X`, thereby saving time.

The two workspace matrices `Y` and `E` can also be reused between calls. You must free `X` `Y`, and `E` yourself, when your computations are done. Below is an example of usage. Note that `X` `Y`, and `E` must be defined on input (either NULL, or valid dense matrices).

```
cholmod_dense *X = NULL, *Y = NULL, *E = NULL ;
...
cholmod_l_solve2 (sys, L, B1, NULL, &X, NULL, &Y, &E, Common) ;
cholmod_l_solve2 (sys, L, B2, NULL, &X, NULL, &Y, &E, Common) ;
```

```
cholmod_l_solve2 (sys, L, B3, NULL, &X, NULL, &Y, &E, Common) ;
cholmod_l_free_dense (&X, Common) ;
cholmod_l_free_dense (&Y, Common) ;
cholmod_l_free_dense (&E, Common) ;
```

The equivalent when using cholmod_solve is:

```
cholmod_dense *X = NULL, *Y = NULL, *E = NULL ;
...
X = cholmod_l_solve (sys, L, B1, Common) ;
cholmod_l_free_dense (&X, Common) ;
X = cholmod_l_solve (sys, L, B2, Common) ;
cholmod_l_free_dense (&X, Common) ;
X = cholmod_l_solve (sys, L, B3, Common) ;
cholmod_l_free_dense (&X, Common) ;
```

Both methods work fine, but in the second method with cholmod_solve, the internal workspaces (Y and E) and the solution (X) are allocated and freed on each call.

The cholmod_solve2 function can also solve for a subset of the solution vector X, if the optional Bset parameter is non-NULL. The right-hand-side B must be a single column vector, and its complexity (real, complex, zomplex) must match that of L. The vector B is dense, but it is assumed to be zero except for row indices specified in Bset. The vector Bset must be a sparse column vector, of dimension the same as B. Only the pattern of Bset is used. The solution X (a dense column vector) is modified on output, but is defined only in the rows defined by the sparse vector Xset. The entries in Bset are a subset of Xset (except if sys is CHOLMOD_P or CHOLMOD_Pt).

No memory allocations are done if the outputs and internal workspaces (X, Xset, Y, and E) have been allocated by a prior call (or if allocated by the user). To let cholmod_solve2 allocate these outputs and workspaces for you, simply initialize them to NULL (as in the example above). Since it is possible for this function to reallocate these 4 arrays, you should always re-acquire the pointers to their internal data (X->x for example) after calling cholmod_solve2), since they may change. They normally will not change except in the first call to this function.

On the first call to cholmod_solve2 when Bset is NULL, the factorization is converted from supernodal to simplicial, if needed. The inverse permutation is also computed and stored in the factorization object, L. This can take a modest amount of time. Subsequent calls to cholmod_solve2 with a small Bset are very fast (both asymptotically and in practice).

You can find an example of how to use cholmod_solve2 in the two demo programs, cholmod_demo and cholmod_l_demo.

21.8 cholmod_etree: find elimination tree

```
int cholmod_etree
(
    /* ---- input ---- */
    cholmod_sparse *A,
    /* ---- output --- */
    int32_t *Parent,    /* size ncol.  Parent [j] = p if p is the parent of j */
    /* --------------- */
```

```
    cholmod_common *Common
) ;

int cholmod_l_etree (cholmod_sparse *, int64_t *, cholmod_common *) ;
```

Purpose: Computes the elimination tree of `A` or `A'*A`. In the symmetric case, the upper triangular part of `A` is used. Entries not in this part of the matrix are ignored. Computing the etree of a symmetric matrix from just its lower triangular entries is not supported. In the unsymmetric case, all of `A` is used, and the etree of `A'*A` is computed. Refer to [20] for a discussion of the elimination tree and its use in sparse Cholesky factorization.

21.9 `cholmod_rowcolcounts`: nonzeros counts of a factor

```
int cholmod_rowcolcounts
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to analyze */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    int32_t *Parent,    /* size nrow.  Parent [i] = p if p is the parent of i */
    int32_t *Post,      /* size nrow.  Post [k] = i if i is the kth node in
                         * the postordered etree. */
    /* ---- output --- */
    int32_t *RowCount,  /* size nrow. RowCount [i] = # entries in the ith row of
                         * L, including the diagonal. */
    int32_t *ColCount,  /* size nrow. ColCount [i] = # entries in the ith
                         * column of L, including the diagonal. */
    int32_t *First,     /* size nrow.  First [i] = k is the least postordering
                         * of any descendant of i. */
    int32_t *Level,     /* size nrow.  Level [i] is the length of the path from
                         * i to the root, with Level [root] = 0. */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowcolcounts (cholmod_sparse *, int64_t *, size_t,
    int64_t *, int64_t *, int64_t *,
    int64_t *, int64_t *, int64_t *,
    cholmod_common *) ;
```

Purpose: Compute the row and column counts of the Cholesky factor `L` of the matrix `A` or `A*A'`. The etree and its postordering must already be computed (see `cholmod_etree` and `cholmod_postorder`) and given as inputs to this routine. For the symmetric case ($\mathbf{LL}^\mathsf{T} = \mathbf{A}$), `A` must be stored in symmetric/lower form (`A->stype = -1`). In the unsymmetric case, `A*A'` or `A(:,f)*A(:,f)'` can be analyzed. The fundamental floating-point operation count is returned in `Common->fl` (this excludes extra flops due to relaxed supernodal amalgamation). Refer to `cholmod_transpose_unsym` for a description of `f`. The algorithm is described in [13, 15].

21.10 `cholmod_analyze_ordering`: analyze a permutation

```
int cholmod_analyze_ordering
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to analyze */
    int ordering,       /* ordering method used */
    int32_t *Perm,      /* size n, fill-reducing permutation to analyze */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    /* ---- output --- */
    int32_t *Parent,    /* size n, elimination tree */
    int32_t *Post,      /* size n, postordering of elimination tree */
    int32_t *ColCount,  /* size n, nnz in each column of L */
    /* ---- workspace  */
    int32_t *First,     /* size nworkspace for cholmod_postorder */
    int32_t *Level,     /* size n workspace for cholmod_postorder */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_analyze_ordering (cholmod_sparse *, int, int64_t *,
    int64_t *, size_t, int64_t *, int64_t *, int64_t *, int64_t *, int64_t *,
    cholmod_common *) ;
```

Purpose: Given a matrix A and its fill-reducing permutation, compute the elimination tree, its (non-weighted) postordering, and the number of nonzeros in each column of L. Also computes the flop count, the total nonzeros in L, and the nonzeros in tril(A) (Common->fl, Common->lnz, and Common->anz). In the unsymmetric case, A(p,f)*A(p,f)' is analyzed, and Common->anz is the number of nonzero entries in the lower triangular part of the product, not in A itself.

Refer to cholmod_transpose_unsym for a description of f.

The column counts of L, flop count, and other statistics from cholmod_rowcolcounts are not computed if ColCount is NULL.

21.11 cholmod_amd: interface to AMD

```
int cholmod_amd
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to order */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    /* ---- output --- */
    int32_t *Perm,      /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_amd (cholmod_sparse *, int64_t *, size_t,
    int64_t *, cholmod_common *) ;
```

Purpose: CHOLMOD interface to the AMD ordering package. Orders A if the matrix is symmetric. On output, Perm [k] = i if row/column i of A is the kth row/column of P*A*P'. This corresponds to A(p,p) in MATLAB notation. If A is unsymmetric, cholmod_amd orders A*A' or A(:,f)*A(:,f)'. On output, Perm [k] = i if row/column i of A*A' is the kth row/column of P*A*A'*P'. This corresponds to A(p,:)*A(p,:)' in MATLAB notation. If f is present, A(p,f)*A(p,f)' is the permuted matrix. Refer to cholmod_transpose_unsym for a description of f.

Computes the flop count for a subsequent $\mathbf{LL}^\mathsf{T}$ factorization, the number of nonzeros in L, and the number of nonzeros in the matrix ordered (A, A*A' or A(:,f)*A(:,f)'). These statistics are returned in Common->fl, Common->lnz, and Common->anz, respectively.

21.12 cholmod_colamd: interface to COLAMD

```
int cholmod_colamd
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to order */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    int postorder,      /* if TRUE, follow with a coletree postorder */
    /* ---- output --- */
    int32_t *Perm,      /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_colamd (cholmod_sparse *, int64_t *, size_t, int,
    int64_t *, cholmod_common *) ;
```

Purpose: CHOLMOD interface to the COLAMD ordering package. Finds a permutation p such that the Cholesky factorization of P*A*A'*P' is sparser than A*A', using COLAMD. If the postorder input parameter is TRUE, the column elimination tree is found and postordered, and the COLAMD ordering is then combined with its postordering (COLAMD itself does not perform this postordering). A must be unsymmetric (A->stype = 0).

21.13 cholmod_rowfac: row-oriented Cholesky factorization

```
int cholmod_rowfac
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to factorize */
    cholmod_sparse *F,  /* used for A*A' case only. F=A' or A(:,fset)' */
    double beta [2],    /* factorize beta*I+A or beta*I+A'*A */
    size_t kstart,      /* first row to factorize */
    size_t kend,        /* last row to factorize is kend-1 */
    /* ---- in/out --- */
    cholmod_factor *L,
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowfac (cholmod_sparse *, cholmod_sparse *, double *, size_t,
```

```
    size_t, cholmod_factor *, cholmod_common *) ;
int cholmod_rowfac_mask
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to factorize */
    cholmod_sparse *F,  /* used for A*A' case only. F=A' or A(:,fset)' */
    double beta [2],    /* factorize beta*I+A or beta*I+A'*A */
    size_t kstart,      /* first row to factorize */
    size_t kend,        /* last row to factorize is kend-1 */
    int32_t *mask,      /* if mask[i] >= 0, then set row i to zero */
    int32_t *RLinkUp,   /* link list of rows to compute */
    /* ---- in/out --- */
    cholmod_factor *L,
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowfac_mask (cholmod_sparse *, cholmod_sparse *, double *, size_t,
    size_t, int64_t *, int64_t *, cholmod_factor *,
    cholmod_common *) ;
int cholmod_rowfac_mask2
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to factorize */
    cholmod_sparse *F,  /* used for A*A' case only. F=A' or A(:,fset)' */
    double beta [2],    /* factorize beta*I+A or beta*I+A'*A */
    size_t kstart,      /* first row to factorize */
    size_t kend,        /* last row to factorize is kend-1 */
    int32_t *mask,      /* if mask[i] >= maskmark, then set row i to zero */
    int32_t maskmark,
    int32_t *RLinkUp,   /* link list of rows to compute */
    /* ---- in/out --- */
    cholmod_factor *L,
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowfac_mask2 (cholmod_sparse *, cholmod_sparse *, double *,
    size_t, size_t, int64_t *, int64_t, int64_t *,
    cholmod_factor *, cholmod_common *) ;
```

Purpose: Full or incremental numerical $\mathbf{LDL}^\mathsf{T}$ or $\mathbf{LL}^\mathsf{T}$ factorization (simplicial, not supernodal). `cholmod_factorize` is the "easy" wrapper for this code, but it does not provide access to incremental factorization. The algorithm is the row-oriented, up-looking method described in [5]. See also [19]. No 2-by-2 pivoting (or any other pivoting) is performed.

`cholmod_rowfac` computes the full or incremental $\mathbf{LDL}^\mathsf{T}$ or $\mathbf{LL}^\mathsf{T}$ factorization of `A+beta*I` (where `A` is symmetric) or `A*F+beta*I` (where `A` and `F` are unsymmetric and only the upper triangular part of `A*F+beta*I` is used). It computes `L` (and `D`, for $\mathbf{LDL}^\mathsf{T}$) one row at a time. The input scalar `beta` is real; only the real part (`beta[0]`) is used.

`L` can be a simplicial symbolic or numeric (`L->is_super` must be `FALSE`). A symbolic factor is converted immediately into a numeric factor containing the identity matrix.

For a full factorization, use `kstart = 0` and `kend = nrow`. The existing nonzero entries (numerical values in L->x and L->z for the zomplex case, and indices in L->i) are overwritten.

To compute an incremental factorization, select kstart and kend as the range of rows of L you wish to compute. Rows kstart to kend-1 of L will be computed. A correct factorization will be computed only if all descendants of all nodes kstart to kend-1 in the elimination tree have been factorized by a prior call to this routine, and if rows kstart to kend-1 have not been factorized. This condition is **not** checked on input.

In the symmetric case, A must be stored in upper form (A->stype is greater than zero). The matrix F is not accessed and may be NULL. Only columns kstart to kend-1 of A are accessed.

In the unsymmetric case, the typical case is F=A'. Alternatively, if F=A(:,f)', then this routine factorizes the matrix S = beta*I + A(:,f)*A(:,f)'. The product A*F is assumed to be symmetric; only the upper triangular part of A*F is used. F must be of size A->ncol by A->nrow.

21.14 cholmod_rowfac_mask: row-oriented Cholesky factorization

```
int cholmod_rowfac_mask
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to factorize */
    cholmod_sparse *F,  /* used for A*A' case only. F=A' or A(:,fset)' */
    double beta [2],    /* factorize beta*I+A or beta*I+A'*A */
    size_t kstart,      /* first row to factorize */
    size_t kend,        /* last row to factorize is kend-1 */
    int32_t *mask,      /* if mask[i] >= 0, then set row i to zero */
    int32_t *RLinkUp,   /* link list of rows to compute */
    /* ---- in/out --- */
    cholmod_factor *L,
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowfac_mask (cholmod_sparse *, cholmod_sparse *, double *, size_t,
    size_t, int64_t *, int64_t *, cholmod_factor *,
    cholmod_common *) ;
int cholmod_rowfac_mask2
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to factorize */
    cholmod_sparse *F,  /* used for A*A' case only. F=A' or A(:,fset)' */
    double beta [2],    /* factorize beta*I+A or beta*I+A'*A */
    size_t kstart,      /* first row to factorize */
    size_t kend,        /* last row to factorize is kend-1 */
    int32_t *mask,      /* if mask[i] >= maskmark, then set row i to zero */
    int32_t maskmark,
    int32_t *RLinkUp,   /* link list of rows to compute */
    /* ---- in/out --- */
    cholmod_factor *L,
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowfac_mask2 (cholmod_sparse *, cholmod_sparse *, double *,
    size_t, size_t, int64_t *, int64_t, int64_t *,
```

```
    cholmod_factor *, cholmod_common *) ;
```

Purpose: For use in LPDASA only.

21.15 `cholmod_row_subtree`: pattern of row of a factor

```
int cholmod_row_subtree
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to analyze */
    cholmod_sparse *F,  /* used for A*A' case only. F=A' or A(:,fset)' */
    size_t k,           /* row k of L */
    int32_t *Parent,    /* elimination tree */
    /* ---- output --- */
    cholmod_sparse *R,  /* pattern of L(k,:), n-by-1 with R->nzmax >= n */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_row_subtree (cholmod_sparse *, cholmod_sparse *, size_t,
    int64_t *, cholmod_sparse *, cholmod_common *) ;
```

Purpose: Compute the nonzero pattern of the solution to the lower triangular system

```
L(0:k-1,0:k-1) * x = A (0:k-1,k)
```

if `A` is symmetric, or

```
L(0:k-1,0:k-1) * x = A (0:k-1,:) * A (:,k)'
```

if `A` is unsymmetric. This gives the nonzero pattern of row `k` of `L` (excluding the diagonal). The pattern is returned postordered, according to the subtree of the elimination tree rooted at node `k`.

The symmetric case requires `A` to be in symmetric-upper form.

The result is returned in `R`, a pre-allocated sparse matrix of size `nrow`-by-1, with `R->nzmax >= nrow`. `R` is assumed to be packed (`Rnz [0]` is not updated); the number of entries in `R` is given by `Rp [0]`.

21.16 `cholmod_row_lsubtree`: pattern of row of a factor

```
int cholmod_row_lsubtree
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to analyze */
    int32_t *Fi, size_t fnz, /* nonzero pattern of kth row of A', not required
                              * for the symmetric case.  Need not be sorted. */
    size_t k,           /* row k of L */
    cholmod_factor *L,  /* the factor L from which parent(i) is derived */
    /* ---- output --- */
    cholmod_sparse *R,  /* pattern of L(k,:), n-by-1 with R->nzmax >= n */
    /* --------------- */
```

```
    cholmod_common *Common
) ;

int cholmod_l_row_lsubtree (cholmod_sparse *, int64_t *, size_t,
    size_t, cholmod_factor *, cholmod_sparse *, cholmod_common *) ;
```

Purpose: Identical to cholmod_row_subtree, except the elimination tree is found from L itself, not Parent. Also, F=A' is not provided; the nonzero pattern of the kth column of F is given by Fi and fnz instead.

21.17 cholmod_resymbol: re-do symbolic factorization

```
int cholmod_resymbol    // recompute symbolic pattern of L
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to analyze */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    int pack,           /* if TRUE, pack the columns of L */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factorization, entries pruned on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_resymbol (cholmod_sparse *, int64_t *, size_t, int,
    cholmod_factor *, cholmod_common *) ;
```

Purpose: Recompute the symbolic pattern of L. Entries not in the symbolic pattern of the factorization of A(p,p) or F*F', where F=A(p,f) or F=A(:,f), are dropped, where p = L->Perm is used to permute the input matrix A.

Refer to cholmod_transpose_unsym for a description of f.

If an entry in L is kept, its numerical value does not change.

This routine is used after a supernodal factorization is converted into a simplicial one, to remove zero entries that were added due to relaxed supernode amalgamation. It can also be used after a series of downdates to remove entries that would no longer be present if the matrix were factorized from scratch. A downdate (cholmod_updown) does not remove any entries from L.

21.18 cholmod_resymbol_noperm: re-do symbolic factorization

```
int cholmod_resymbol_noperm
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to analyze */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    int pack,           /* if TRUE, pack the columns of L */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factorization, entries pruned on output */
```

```
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_resymbol_noperm (cholmod_sparse *, int64_t *, size_t, int,
    cholmod_factor *, cholmod_common *) ;
```

Purpose: Identical to `cholmod_resymbol`, except that the fill-reducing ordering `L->Perm` is not used.

21.19 `cholmod_postorder`: tree postorder

```
int32_t cholmod_postorder       /* return # of nodes postordered */
(
    /* ---- input ---- */
    int32_t *Parent,    /* size n. Parent [j] = p if p is the parent of j */
    size_t n,
    int32_t *Weight_p,  /* size n, optional. Weight [j] is weight of node j */
    /* ---- output --- */
    int32_t *Post,      /* size n. Post [k] = j is kth in postordered tree */
    /* --------------- */
    cholmod_common *Common
) ;

int64_t cholmod_l_postorder (int64_t *, size_t, int64_t *, int64_t *,
    cholmod_common *) ;
```

Purpose: Postorder a tree. The tree is either an elimination tree (the output from `cholmod_etree`) or a component tree (from `cholmod_nested_dissection`).

An elimination tree is a complete tree of `n` nodes with `Parent [j] > j` or `Parent [j] = -1` if `j` is a root. On output `Post [0..n-1]` is a complete permutation vector; `Post [k] = j` if node `j` is the `k`th node in the postordered elimination tree, where `k` is in the range 0 to `n-1`.

A component tree is a subset of `0:n-1`. `Parent [j] = -2` if node `j` is not in the component tree. `Parent [j] = -1` if `j` is a root of the component tree, and `Parent [j]` is in the range 0 to `n-1` if `j` is in the component tree but not a root. On output, `Post [k]` is defined only for nodes in the component tree. `Post [k] = j` if node `j` is the `k`th node in the postordered component tree, where `k` is in the range 0 to the number of components minus 1. Node `j` is ignored and not included in the postorder if `Parent [j] < -1`. As a result, `cholmod_check_parent (Parent, ...)` and `cholmod_check_perm (Post, ...)` fail if used for a component tree and its postordering.

An optional node weight can be given. When starting a postorder at node `j`, the children of `j` are ordered in decreasing order of their weight. If no weights are given (`Weight` is `NULL`) then children are ordered in decreasing order of their node number. The weight of a node must be in the range 0 to `n-1`. Weights outside that range are silently converted to that range (weights < 0 are treated as zero, and weights $\geq$ `n` are treated as `n-1`).

21.20 `cholmod_rcond`: reciprocal condition number

```
double cholmod_rcond        /* return min(diag(L)) / max(diag(L)) */
(
    /* ---- input ---- */
    cholmod_factor *L,
    /* --------------- */
    cholmod_common *Common
) ;

double cholmod_l_rcond (cholmod_factor *, cholmod_common *) ;
```

Purpose: Returns a rough estimate of the reciprocal of the condition number: the minimum entry on the diagonal of `L` (or absolute entry of `D` for an $\mathbf{LDL}^\mathsf{T}$ factorization) divided by the maximum entry. `L` can be real, complex, or zomplex. Returns -1 on error, 0 if the matrix is singular or has a zero or NaN entry on the diagonal of `L`, 1 if the matrix is 0-by-0, or `min(diag(L))/max(diag(L))` otherwise. Never returns NaN; if `L` has a NaN on the diagonal it returns zero instead.

22 `Modify` Module routines

22.1 `cholmod_updown`: update/downdate

```
int cholmod_updown          // update/downdate
(
    /* ---- input ---- */
    int update,         /* TRUE for update, FALSE for downdate */
    cholmod_sparse *C,  /* the incoming sparse update */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    /* --------------- */
    cholmod_common *Common
) ;
int cholmod_l_updown (int, cholmod_sparse *, cholmod_factor *, cholmod_common *) ;
```

Purpose: Updates/downdates the $\mathbf{LDL}^\mathsf{T}$ factorization (symbolic, then numeric), by computing a new factorization of

$$\overline{\mathbf{LDL}}^\mathsf{T} = \mathbf{LDL}^\mathsf{T} \pm \mathbf{CC}^\mathsf{T}$$

where $\overline{\mathbf{L}}$ denotes the new factor. `C` must be sorted. It can be either packed or unpacked. As in all CHOLMOD routines, the columns of `L` are sorted on input, and also on output. If `L` does not contain a simplicial numeric $\mathbf{LDL}^\mathsf{T}$ factorization, it is converted into one. Thus, a supernodal $\mathbf{LL}^\mathsf{T}$ factorization can be passed to `cholmod_updown`. A symbolic `L` is converted into a numeric identity matrix. If the initial conversion fails, the factor is returned unchanged.

If memory runs out during the update, the factor is returned as a simplicial symbolic factor. That is, everything is freed except for the fill-reducing ordering and its corresponding column counts (typically computed by `cholmod_analyze`).

Note that the fill-reducing permutation `L->Perm` is not used. The row indices of `C` refer to the rows of `L`, not `A`. If your original system is $\mathbf{LDL}^\mathsf{T} = \mathbf{PAP}^\mathsf{T}$ (where $\mathbf{P}$ = `L->Perm`), and you want to compute the $\mathbf{LDL}^\mathsf{T}$ factorization of $\mathbf{A} + \mathbf{CC}^\mathsf{T}$, then you must permute $\mathbf{C}$ first. That is, if

$$\mathbf{PAP}^\mathsf{T} = \mathbf{LDL}^\mathsf{T}$$

is the initial factorization, then

$$\mathbf{P}(\mathbf{A} + \mathbf{CC}^\mathsf{T})\mathbf{P}^\mathsf{T} = \mathbf{PAP}^\mathsf{T} + \mathbf{PCC}^\mathsf{T}\mathbf{P}^\mathsf{T} = \mathbf{LDL}^\mathsf{T} + (\mathbf{PC})(\mathbf{PC})^\mathsf{T} = \mathbf{LDL}^\mathsf{T} + \overline{\mathbf{CC}}^\mathsf{T}$$

where $\overline{\mathbf{C}} = \mathbf{PC}$.

You can use the `cholmod_submatrix` routine in the `MatrixOps` Module to permute `C`, with:

```
Cnew = cholmod_submatrix (C, L->Perm, L->n, NULL, -1, TRUE, TRUE, Common) ;
```

Note that the `sorted` input parameter to `cholmod_submatrix` must be `TRUE`, because `cholmod_updown` requires `C` with sorted columns. Only real matrices are supported. The algorithms are described in [8, 9].

22.2 `cholmod_updown_solve`: update/downdate

```
int cholmod_updown_solve
(
    /* ---- input ---- */
    int update,         /* TRUE for update, FALSE for downdate */
    cholmod_sparse *C,  /* the incoming sparse update */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    cholmod_dense *X,   /* solution to Lx=b (size n-by-1) */
    cholmod_dense *DeltaB,  /* change in b, zero on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_updown_solve (int, cholmod_sparse *, cholmod_factor *,
    cholmod_dense *, cholmod_dense *, cholmod_common *) ;
```

Purpose: Identical to `cholmod_updown`, except the system $\mathbf{Lx} = \mathbf{b}$ is also updated/downdated. The new system is $\overline{\mathbf{L}}\overline{\mathbf{x}} = \mathbf{b} + \Delta\mathbf{b}$. The old solution $\mathbf{x}$ is overwritten with $\overline{\mathbf{x}}$. Note that as in the update/downdate of $\mathbf{L}$ itself, the fill- reducing permutation `L->Perm` is not used. The vectors $\mathbf{x}$ and $\mathbf{b}$ are in the permuted ordering, not your original ordering. This routine does not handle multiple right-hand-sides.

22.3 `cholmod_updown_mark`: update/downdate

```
int cholmod_updown_mark
(
    /* ---- input ---- */
    int update,         /* TRUE for update, FALSE for downdate */
    cholmod_sparse *C,  /* the incoming sparse update */
    int32_t *colmark,   /* array of size n.  See cholmod_updown.c */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    cholmod_dense *X,   /* solution to Lx=b (size n-by-1) */
    cholmod_dense *DeltaB,  /* change in b, zero on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_updown_mark (int, cholmod_sparse *, int64_t *,
    cholmod_factor *, cholmod_dense *, cholmod_dense *, cholmod_common *) ;
```

Purpose: Identical to `cholmod_updown_solve`, except that only part of $\mathbf{L}$ is used in the update of the solution to $\mathbf{Lx} = \mathbf{b}$. For more details, see the source code file `CHOLMOD/Modify/cholmod_updown.c`. This routine is meant for use in the `LPDASA` linear program solver only, by Hager and Davis.

22.4 `cholmod_updown_mask`: update/downdate

```
int cholmod_updown_mask
(
    /* ---- input ---- */
    int update,         /* TRUE for update, FALSE for downdate */
    cholmod_sparse *C,  /* the incoming sparse update */
    int32_t *colmark,   /* array of size n.  See cholmod_updown.c */
    int32_t *mask,      /* size n */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    cholmod_dense *X,   /* solution to Lx=b (size n-by-1) */
    cholmod_dense *DeltaB,  /* change in b, zero on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_updown_mask (int, cholmod_sparse *, int64_t *,
    int64_t *, cholmod_factor *, cholmod_dense *, cholmod_dense *,
    cholmod_common *) ;
int cholmod_updown_mask2
(
    /* ---- input ---- */
    int update,         /* TRUE for update, FALSE for downdate */
    cholmod_sparse *C,  /* the incoming sparse update */
    int32_t *colmark,   /* array of size n.  See cholmod_updown.c */
    int32_t *mask,      /* size n */
    int32_t maskmark,
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    cholmod_dense *X,   /* solution to Lx=b (size n-by-1) */
    cholmod_dense *DeltaB,  /* change in b, zero on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_updown_mask2 (int, cholmod_sparse *, int64_t *,
    int64_t *, int64_t, cholmod_factor *, cholmod_dense *,
    cholmod_dense *, cholmod_common *) ;
```

Purpose: For use in LPDASA only.

22.5 `cholmod_rowadd`: add row to factor

```
int cholmod_rowadd      // add a row to an LDL' factorization
(
    /* ---- input ---- */
    size_t k,           /* row/column index to add */
    cholmod_sparse *R,  /* row/column of matrix to factorize (n-by-1) */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowadd (size_t, cholmod_sparse *, cholmod_factor *,
```

```
    cholmod_common *) ;
```

Purpose: Adds a row and column to an $\mathbf{LDL}^\mathsf{T}$ factorization. The kth row and column of L must be equal to the kth row and column of the identity matrix on input. Only real matrices are supported. The algorithm is described in [10].

22.6 cholmod_rowadd_solve: add row to factor

```
int cholmod_rowadd_solve
(
    /* ---- input ---- */
    size_t k,           /* row/column index to add */
    cholmod_sparse *R,  /* row/column of matrix to factorize (n-by-1) */
    double bk [2],      /* kth entry of the right-hand-side b */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    cholmod_dense *X,   /* solution to Lx=b (size n-by-1) */
    cholmod_dense *DeltaB,  /* change in b, zero on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowadd_solve (size_t, cholmod_sparse *, double *,
    cholmod_factor *, cholmod_dense *, cholmod_dense *, cholmod_common *) ;
```

Purpose: Identical to cholmod_rowadd, except the system $\mathbf{Lx} = \mathbf{b}$ is also updated/downdated, just like cholmod_updown_solve.

22.7 cholmod_rowdel: delete row from factor

```
int cholmod_rowdel      // delete a rw from an LDL' factorization
(
    /* ---- input ---- */
    size_t k,           /* row/column index to delete */
    cholmod_sparse *R,  /* NULL, or the nonzero pattern of kth row of L */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowdel (size_t, cholmod_sparse *, cholmod_factor *,
    cholmod_common *) ;
```

Purpose: Deletes a row and column from an $\mathbf{LDL}^\mathsf{T}$ factorization. The kth row and column of L is equal to the kth row and column of the identity matrix on output. Only real matrices are supported.

22.8 `cholmod_rowdel_solve`: delete row from factor

```
int cholmod_rowdel_solve
(
    /* ---- input ---- */
    size_t k,           /* row/column index to delete */
    cholmod_sparse *R,  /* NULL, or the nonzero pattern of kth row of L */
    double yk [2],      /* kth entry in the solution to A*y=b */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    cholmod_dense *X,   /* solution to Lx=b (size n-by-1) */
    cholmod_dense *DeltaB,  /* change in b, zero on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowdel_solve (size_t, cholmod_sparse *, double *,
    cholmod_factor *, cholmod_dense *, cholmod_dense *, cholmod_common *) ;
```

Purpose: Identical to `cholmod_rowdel`, except the system $\mathbf{Lx} = \mathbf{b}$ is also updated/downdated, just like `cholmod_updown_solve`. When row/column k of $\mathbf{A}$ is deleted from the system $\mathbf{Ay} = \mathbf{b}$, this can induce a change to $\mathbf{x}$, in addition to changes arising when $\mathbf{L}$ and $\mathbf{b}$ are modified. If this is the case, the kth entry of $\mathbf{y}$ is required as input (`yk`). The algorithm is described in [10].

22.9 `cholmod_rowadd_mark`: add row to factor

```
int cholmod_rowadd_mark
(
    /* ---- input ---- */
    size_t k,           /* row/column index to add */
    cholmod_sparse *R,  /* row/column of matrix to factorize (n-by-1) */
    double bk [2],      /* kth entry of the right hand side, b */
    int32_t *colmark,   /* array of size n.  See cholmod_updown.c */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    cholmod_dense *X,   /* solution to Lx=b (size n-by-1) */
    cholmod_dense *DeltaB,  /* change in b, zero on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowadd_mark (size_t, cholmod_sparse *, double *,
    int64_t *, cholmod_factor *, cholmod_dense *, cholmod_dense *,
    cholmod_common *) ;
```

Purpose: Identical to `cholmod_rowadd_solve`, except that only part of $\mathbf{L}$ is used in the update of the solution to $\mathbf{Lx} = \mathbf{b}$. For more details, see the source code file `CHOLMOD/Modify/cholmod_rowadd.c`. This routine is meant for use in the `LPDASA` linear program solver only.

22.10 `cholmod_rowdel_mark`: delete row from factor

```
int cholmod_rowdel_mark
(
    /* ---- input ---- */
    size_t k,           /* row/column index to delete */
    cholmod_sparse *R,  /* NULL, or the nonzero pattern of kth row of L */
    double yk [2],      /* kth entry in the solution to A*y=b */
    int32_t *colmark,   /* array of size n.  See cholmod_updown.c */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factor to modify */
    cholmod_dense *X,   /* solution to Lx=b (size n-by-1) */
    cholmod_dense *DeltaB,  /* change in b, zero on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_rowdel_mark (size_t, cholmod_sparse *, double *,
    int64_t *, cholmod_factor *, cholmod_dense *, cholmod_dense *,
    cholmod_common *) ;
```

Purpose: Identical to `cholmod_rowadd_solve`, except that only part of $\mathbf{L}$ is used in the update of the solution to $\mathbf{Lx} = \mathbf{b}$. For more details, see the source code file `CHOLMOD/Modify/cholmod_rowdel.c`. This routine is meant for use in the `LPDASA` linear program solver only.

23 MatrixOps Module routines

23.1 cholmod_drop: drop small entries

```
int cholmod_drop
(
    /* ---- input ---- */
    double tol,         /* keep entries with absolute value > tol */
    /* ---- in/out --- */
    cholmod_sparse *A,  /* matrix to drop entries from */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_drop (double, cholmod_sparse *, cholmod_common *) ;
```

Purpose: Drop small entries from A, and entries in the ignored part of A if A is symmetric. No CHOLMOD routine drops small numerical entries from a matrix, except for this one. NaN's and Inf's are kept.

Supports pattern and real matrices; complex and zomplex matrices are not supported.

23.2 cholmod_norm_dense: dense matrix norm

```
double cholmod_norm_dense
(
    /* ---- input ---- */
    cholmod_dense *X,   /* matrix to compute the norm of */
    int norm,           /* type of norm: 0: inf. norm, 1: 1-norm, 2: 2-norm */
    /* --------------- */
    cholmod_common *Common
) ;

double cholmod_l_norm_dense (cholmod_dense *, int, cholmod_common *) ;
```

Purpose: Returns the infinity-norm, 1-norm, or 2-norm of a dense matrix. Can compute the 2-norm only for a dense column vector. All xtypes are supported.

23.3 cholmod_norm_sparse: sparse matrix norm

```
double cholmod_norm_sparse
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to compute the norm of */
    int norm,           /* type of norm: 0: inf. norm, 1: 1-norm */
    /* --------------- */
    cholmod_common *Common
) ;

double cholmod_l_norm_sparse (cholmod_sparse *, int, cholmod_common *) ;
```

Purpose: Returns the infinity-norm or 1-norm of a sparse matrix. All xtypes are supported.

23.4 cholmod_scale: scale sparse matrix

```
#define CHOLMOD_SCALAR 0        /* A = s*A */
#define CHOLMOD_ROW 1           /* A = diag(s)*A */
#define CHOLMOD_COL 2           /* A = A*diag(s) */
#define CHOLMOD_SYM 3           /* A = diag(s)*A*diag(s) */

int cholmod_scale
(
    /* ---- input ---- */
    cholmod_dense *S,   /* scale factors (scalar or vector) */
    int scale,          /* type of scaling to compute */
    /* ---- in/out --- */
    cholmod_sparse *A,  /* matrix to scale */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_scale (cholmod_dense *, int, cholmod_sparse *, cholmod_common *) ;
```

Purpose: Scales a matrix: `A = diag(s)*A`, `A*diag(s)`, `s*A`, or `diag(s)*A*diag(s)`.

`A` can be of any type (packed/unpacked, upper/lower/unsymmetric). The symmetry of `A` is ignored; all entries in the matrix are modified.

If `A` is `m`-by-`n` unsymmetric but scaled symmetrically, the result is

```
A = diag (s (1:m)) * A * diag (s (1:n))
```

Row or column scaling of a symmetric matrix still results in a symmetric matrix, since entries are still ignored by other routines. For example, when row-scaling a symmetric matrix where just the upper triangular part is stored (and lower triangular entries ignored) `A = diag(s)*triu(A)` is performed, where the result `A` is also symmetric-upper. This has the effect of modifying the implicit lower triangular part. In MATLAB notation:

```
U = diag(s)*triu(A) ;
L = tril (U',-1)
A = L + U ;
```

The scale parameter determines the kind of scaling to perform and the size of `S`:

`scale`	operation	size of `S`
`CHOLMOD_SCALAR`	`s[0]*A`	1
`CHOLMOD_ROW`	`diag(s)*A`	`nrow`-by-1 or 1-by-`nrow`
`CHOLMOD_COL`	`A*diag(s)`	`ncol`-by-1 or 1-by-`ncol`
`CHOLMOD_SYM`	`diag(s)*A*diag(s)`	`max(nrow,ncol)`-by-1, or 1-by-`max(nrow,ncol)`

Only real matrices are supported.

23.5 cholmod_sdmult: sparse-times-dense matrix

```
int cholmod_sdmult
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* sparse matrix to multiply */
    int transpose,      /* use A if 0, or A' otherwise */
    double alpha [2],   /* scale factor for A */
    double beta [2],    /* scale factor for Y */
    cholmod_dense *X,   /* dense matrix to multiply */
    /* ---- in/out --- */
    cholmod_dense *Y,   /* resulting dense matrix */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_sdmult (cholmod_sparse *, int, double *, double *,
    cholmod_dense *, cholmod_dense *Y, cholmod_common *) ;
```

Purpose: Sparse matrix times dense matrix: `Y = alpha*(A*X) + beta*Y` or `Y = alpha*(A'*X) + beta*Y`, where `A` is sparse and `X` and `Y` are dense. When using `A`, `X` has `A->ncol` rows and `Y` has `A->nrow` rows. When using `A'`, `X` has `A->nrow` rows and `Y` has `A->ncol` rows. If `transpose = 0`, then `A` is used; otherwise, `A'` is used (the complex conjugate transpose). The `transpose` parameter is ignored if the matrix is symmetric or Hermitian. (the array transpose `A.'` is not supported). Supports real, complex, and zomplex matrices, but the xtypes of `A`, `X`, and `Y` must all match.

23.6 cholmod_ssmult: sparse-times-sparse matrix

```
cholmod_sparse *cholmod_ssmult
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* left matrix to multiply */
    cholmod_sparse *B,  /* right matrix to multiply */
    int stype,          /* requested stype of C */
    int values,         /* TRUE: do numerical values, FALSE: pattern only */
    int sorted,         /* if TRUE then return C with sorted columns */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_sparse *cholmod_l_ssmult (cholmod_sparse *, cholmod_sparse *, int, int,
    int, cholmod_common *) ;
```

Purpose: Computes `C = A*B`; multiplying two sparse matrices. `C` is returned as packed, and either unsorted or sorted, depending on the `sorted` input parameter. If `C` is returned sorted, then either `C = (B'*A')'` or `C = (A*B)''` is computed, depending on the number of nonzeros in `A`, `B`, and `C`. The stype of `C` is determined by the `stype` parameter. Only pattern and real matrices are supported. Complex and zomplex matrices are supported only when the numerical values are not computed (`values` is `FALSE`).

23.7 cholmod_submatrix: sparse submatrix

```
cholmod_sparse *cholmod_submatrix
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to subreference */
    int32_t *rset,      /* set of row indices, duplicates OK */
    int64_t rsize,      /* size of r; rsize < 0 denotes ":" */
    int32_t *cset,      /* set of column indices, duplicates OK */
    int64_t csize,      /* size of c; csize < 0 denotes ":" */
    int values,         /* if TRUE compute the numerical values of C */
    int sorted,         /* if TRUE then return C with sorted columns */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_sparse *cholmod_l_submatrix (cholmod_sparse *, int64_t *,
    int64_t, int64_t *, int64_t, int, int, cholmod_common *) ;
```

Purpose: Returns `C = A (rset,cset)`, where `C` becomes `length(rset)`-by-`length(cset)` in dimension. `rset` and `cset` can have duplicate entries. `A` must be unsymmetric. `C` unsymmetric and is packed. If `sorted` is `TRUE` on input, or `rset` is sorted and `A` is sorted, then `C` is sorted; otherwise `C` is unsorted.

If `rset` is `NULL`, it means "`[ ]`" in MATLAB notation, the empty set. The number of rows in the result `C` will be zero if `rset` is `NULL`. Likewise if `cset` means the empty set; the number of columns in the result `C` will be zero if `cset` is `NULL`. If `rsize` or `csize` is negative, it denotes "`:`" in MATLAB notation. Thus, if both `rsize` and `csize` are negative `C = A(:,:)  = A` is returned.

For permuting a matrix, this routine is an alternative to `cholmod_ptranspose` (which permutes and transposes a matrix and can work on symmetric matrices).

The time taken by this routine is O(`A->nrow`) if the `Common` workspace needs to be initialized, plus O(`C->nrow + C->ncol + nnz (A (:,cset))`). Thus, if `C` is small and the workspace is not initialized, the time can be dominated by the call to `cholmod_allocate_work`. However, once the workspace is allocated, subsequent calls take less time.

Only pattern and real matrices are supported. Complex and zomplex matrices are supported only when `values` is `FALSE`.

23.8 cholmod_horzcat: horizontal concatenation

```
cholmod_sparse *cholmod_horzcat
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* left matrix to concatenate */
    cholmod_sparse *B,  /* right matrix to concatenate */
    int values,         /* if TRUE compute the numerical values of C */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_sparse *cholmod_l_horzcat (cholmod_sparse *, cholmod_sparse *, int,
    cholmod_common *) ;
```

Purpose: Horizontal concatenation, returns `C = [A,B]` in MATLAB notation. `A` and `B` can have any stype. `C` is returned unsymmetric and packed. `A` and `B` must have the same number of rows. `C` is sorted if both `A` and `B` are sorted. `A` and `B` must have the same numeric xtype, unless `values` is `FALSE`. `A` and `B` cannot be complex or zomplex, unless `values` is `FALSE`.

23.9 cholmod_vertcat: vertical concatenation

```
cholmod_sparse *cholmod_vertcat
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* left matrix to concatenate */
    cholmod_sparse *B,  /* right matrix to concatenate */
    int values,         /* if TRUE compute the numerical values of C */
    /* --------------- */
    cholmod_common *Common
) ;

cholmod_sparse *cholmod_l_vertcat (cholmod_sparse *, cholmod_sparse *, int,
    cholmod_common *) ;
```

Purpose: Vertical concatenation, returns `C = [A;B]` in MATLAB notation. `A` and `B` can have any stype. `C` is returned unsymmetric and packed. `A` and `B` must have the same number of columns. `C` is sorted if both `A` and `B` are sorted. `A` and `B` must have the same numeric xtype, unless `values` is `FALSE`. `A` and `B` cannot be complex or zomplex, unless `values` is `FALSE`.

23.10 `cholmod_symmetry`: compute the symmetry of a matrix

```
int cholmod_symmetry
(
    /* ---- input ---- */
    cholmod_sparse *A,
    int option,
    /* ---- output ---- */
    int32_t *xmatched,
    int32_t *pmatched,
    int32_t *nzoffdiag,
    int32_t *nzdiag,
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_symmetry (cholmod_sparse *, int, int64_t *, int64_t *, int64_t *,
    int64_t *, cholmod_common *) ;
```

Purpose:

Determines if a sparse matrix is rectangular, unsymmetric, symmetric, skew-symmetric, or Hermitian. It does so by looking at its numerical values of both upper and lower triangular parts of a CHOLMOD "unsymmetric" matrix, where A-¿stype == 0. The transpose of A is NOT constructed.

If not unsymmetric, it also determines if the matrix has a diagonal whose entries are all real and positive (and thus a candidate for sparse Cholesky if A-¿stype is changed to a nonzero value).

Note that a Matrix Market "general" matrix is either rectangular or unsymmetric.

The row indices in the column of each matrix MUST be sorted for this function to work properly (A-¿sorted must be TRUE). This routine returns EMPTY if A-¿stype is not zero, or if A-¿sorted is FALSE. The exception to this rule is if A is rectangular.

If option == 0, then this routine returns immediately when it finds a non-positive diagonal entry (or one with nonzero imaginary part). If the matrix is not a candidate for sparse Cholesky, it returns the value `CHOLMOD_MM_UNSYMMETRIC`, even if the matrix might in fact be symmetric or Hermitian.

This routine is useful inside the MATLAB backslash, which must look at an arbitrary matrix (A-¿stype == 0) and determine if it is a candidate for sparse Cholesky. In that case, option should be 0.

This routine is also useful when writing a MATLAB matrix to a file in Rutherford/Boeing or Matrix Market format. Those formats require a determination as to the symmetry of the matrix, and thus this routine should not return upon encountering the first non-positive diagonal. In this case, option should be 1.

If option is 2, this function can be used to compute the numerical and pattern symmetry, where 0 is a completely unsymmetric matrix, and 1 is a perfectly symmetric matrix. This option is used when computing the following statistics for the matrices in the SuiteSparse Matrix Collection.

numerical symmetry: number of matched off-diagonal nonzeros over the total number of off-diagonal entries. A real entry a_{ij}, $i \neq j$, is matched if $a_{ji} = a_{ij}$, but this is only counted if both a_{ji} and a_{ij} are nonzero. This does not depend on `Z`. (If A is complex, then the above test is modified; a_{ij} is matched if $\mathrm{conj}(a_{ji}) = a_{ij}$.

Then numeric symmetry = xmatched / nzoffdiag, or 1 if nzoffdiag = 0.

pattern symmetry: number of matched offdiagonal entries over the total number of offdiagonal entries. An entry a_{ij}, $i \neq j$, is matched if a_{ji} is also an entry.

Then pattern symmetry = pmatched / nzoffdiag, or 1 if nzoffdiag = 0.

The symmetry of a matrix with no offdiagonal entries is equal to 1.

A workspace of size ncol integers is allocated; EMPTY is returned if this allocation fails.

Summary of return values:

`EMPTY (-1)`	out of memory, stype not zero, A not sorted
`CHOLMOD_MM_RECTANGULAR 1`	A is rectangular
`CHOLMOD_MM_UNSYMMETRIC 2`	A is unsymmetric
`CHOLMOD_MM_SYMMETRIC 3`	A is symmetric, but with non-pos. diagonal
`CHOLMOD_MM_HERMITIAN 4`	A is Hermitian, but with non-pos. diagonal
`CHOLMOD_MM_SKEW_SYMMETRIC 5`	A is skew symmetric
`CHOLMOD_MM_SYMMETRIC_POSDIAG 6`	A is symmetric with positive diagonal
`CHOLMOD_MM_HERMITIAN_POSDIAG 7`	A is Hermitian with positive diagonal

See also the `spsym` mexFunction, which is a MATLAB interface for this code.

If the matrix is a candidate for sparse Cholesky, it will return a result `CHOLMOD_MM_SYMMETRIC_POSDIAG` if real, or `CHOLMOD_MM_HERMITIAN_POSDIAG` if complex. Otherwise, it will return a value less than this. This is true regardless of the value of the option parameter.

24 Supernodal Module routines

24.1 cholmod_super_symbolic: supernodal symbolic factorization

```
int cholmod_super_symbolic
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to analyze */
    cholmod_sparse *F,  /* F = A' or A(:,f)' */
    int32_t *Parent,    /* elimination tree */
    /* ---- in/out --- */
    cholmod_factor *L,  /* simplicial symbolic on input,
                         * supernodal symbolic on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_super_symbolic (cholmod_sparse *, cholmod_sparse *, int64_t *,
    cholmod_factor *, cholmod_common *) ;
int cholmod_super_symbolic2
(
    /* ---- input ---- */
    int for_whom,       /* FOR_SPQR     (0): for SPQR but not GPU-accelerated
                           FOR_CHOLESKY (1): for Cholesky (GPU or not)
                           FOR_SPQRGPU  (2): for SPQR with GPU acceleration */
    cholmod_sparse *A,  /* matrix to analyze */
    cholmod_sparse *F,  /* F = A' or A(:,f)' */
    int32_t *Parent,    /* elimination tree */
    /* ---- in/out --- */
    cholmod_factor *L,  /* simplicial symbolic on input,
                         * supernodal symbolic on output */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_super_symbolic2 (int, cholmod_sparse *, cholmod_sparse *,
    int64_t *, cholmod_factor *, cholmod_common *) ;
```

Purpose: Supernodal symbolic analysis of the $\mathbf{LL}^\mathsf{T}$ factorization of `A`, `A*A'`, or `A(:,f)*A(:,f)'`. This routine must be preceded by a simplicial symbolic analysis (`cholmod_rowcolcounts`). See `Cholesky/cholmod_analyze.c` for an example of how to use this routine. The user need not call this directly; `cholmod_analyze` is a "simple" wrapper for this routine. `A` can be symmetric (upper), or unsymmetric. The symmetric/lower form is not supported. In the unsymmetric case `F` is the normally transpose of `A`. Alternatively, if `F=A(:,f)'` then `F*F'` is analyzed. Requires `Parent` and `L->ColCount` to be defined on input; these are the simplicial `Parent` and `ColCount` arrays as computed by `cholmod_rowcolcounts`. Does not use `L->Perm`; the input matrices `A` and `F` must already be properly permuted. Allocates and computes the supernodal pattern of `L` (`L->super`, `L->pi`, `L->px`, and `L->s`). Does not allocate the real part (`L->x`).

24.2 cholmod_super_numeric: supernodal numeric factorization

```
int cholmod_super_numeric
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to factorize */
    cholmod_sparse *F,  /* F = A' or A(:,f)' */
    double beta [2],    /* beta*I is added to diagonal of matrix to factorize */
    /* ---- in/out --- */
    cholmod_factor *L,  /* factorization */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_super_numeric (cholmod_sparse *, cholmod_sparse *, double *,
    cholmod_factor *, cholmod_common *) ;
```

Purpose: Computes the numerical Cholesky factorization of A+beta*I or A*F+beta*I. Only the lower triangular part of A+beta*I or A*F+beta*I is accessed. The matrices A and F must already be permuted according to the fill-reduction permutation L->Perm. cholmod_factorize is an "easy" wrapper for this code which applies that permutation. The input scalar beta is real; only the real part (beta[0] is used.

Symmetric case: A is a symmetric (lower) matrix. F is not accessed and may be NULL. With a fill-reducing permutation, A(p,p) should be passed for A, where is p is L->Perm.

Unsymmetric case: A is unsymmetric, and F must be present. Normally, F=A'. With a fill-reducing permutation, A(p,f) and A(p,f)' should be passed as the parameters A and F, respectively, where f is a list of the subset of the columns of A.

The input factorization L must be supernodal (L->is_super is TRUE). It can either be symbolic or numeric. In the first case, L has been analyzed by cholmod_analyze or cholmod_super_symbolic, but the matrix has not yet been numerically factorized. The numerical values are allocated here and the factorization is computed. In the second case, a prior matrix has been analyzed and numerically factorized, and a new matrix is being factorized. The numerical values of L are replaced with the new numerical factorization.

L->is_ll is ignored on input, and set to TRUE on output. This routine always computes an $\mathbf{LL}^\mathsf{T}$ factorization. Supernodal $\mathbf{LDL}^\mathsf{T}$ factorization is not supported.

If the matrix is not positive definite the routine returns TRUE, but sets Common->status to CHOLMOD_NOT_POSDEF and L->minor is set to the column at which the failure occurred. Columns L->minor to L->n-1 are set to zero.

If L is supernodal symbolic on input, it is converted to a supernodal numeric factor on output, with an xtype of real if A is real, or complex if A is complex or zomplex. If L is supernodal numeric on input, its xtype must match A (except that L can be complex and A zomplex). The xtype of A and F must match.

24.3 cholmod_super_lsolve: supernodal forward solve

```
int cholmod_super_lsolve
(
    /* ---- input ---- */
    cholmod_factor *L,  /* factor to use for the forward solve */
    /* ---- output ---- */
    cholmod_dense *X,   /* b on input, solution to Lx=b on output */
    /* ---- workspace   */
    cholmod_dense *E,   /* workspace of size nrhs*(L->maxesize) */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_super_lsolve (cholmod_factor *, cholmod_dense *, cholmod_dense *,
    cholmod_common *) ;
```

Purpose: Solve $\mathbf{Lx} = \mathbf{b}$ for a supernodal factorization. This routine does not apply the permutation `L->Perm`. See `cholmod_solve` for a more general interface that performs that operation. Only real and complex xtypes are supported. `L`, `X`, and `E` must have the same xtype.

24.4 cholmod_super_ltsolve: supernodal backsolve

```
int cholmod_super_ltsolve
(
    /* ---- input ---- */
    cholmod_factor *L,  /* factor to use for the backsolve */
    /* ---- output ---- */
    cholmod_dense *X,   /* b on input, solution to L'x=b on output */
    /* ---- workspace   */
    cholmod_dense *E,   /* workspace of size nrhs*(L->maxesize) */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_super_ltsolve (cholmod_factor *, cholmod_dense *, cholmod_dense *,
    cholmod_common *) ;
```

Purpose: Solve $\mathbf{L}^\mathsf{T}\mathbf{x} = \mathbf{b}$ for a supernodal factorization. This routine does not apply the permutation `L->Perm`. See `cholmod_solve` for a more general interface that performs that operation. Only real and complex xtypes are supported. `L`, `X`, and `E` must have the same xtype.

25 Partition Module routines

25.1 cholmod_nested_dissection: nested dissection ordering

```
int64_t cholmod_nested_dissection      /* returns # of components */
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to order */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    /* ---- output --- */
    int32_t *Perm,      /* size A->nrow, output permutation */
    int32_t *CParent,   /* size A->nrow.  On output, CParent [c] is the parent
                         * of component c, or EMPTY if c is a root, and where
                         * c is in the range 0 to # of components minus 1 */
    int32_t *Cmember,   /* size A->nrow.  Cmember [j] = c if node j of A is
                         * in component c */
    /* --------------- */
    cholmod_common *Common
) ;

int64_t cholmod_l_nested_dissection (cholmod_sparse *, int64_t *, size_t,
    int64_t *, int64_t *, int64_t *, cholmod_common *) ;
```

Purpose: CHOLMOD's nested dissection algorithm: using its own compression and connected-components algorithms, an external graph partitioner (METIS), and a constrained minimum degree ordering algorithm (CAMD, CCOLAMD, or CSYMAMD). Typically gives better orderings than METIS_NodeND (about 5% to 10% fewer nonzeros in L).

This method uses a node bisector, applied recursively (but using a non-recursive implementation). Once the graph is partitioned, it calls a constrained minimum degree code (CAMD or CSYMAMD for A+A', and CCOLAMD for A*A') to order all the nodes in the graph - but obeying the constraints determined by the separators. This routine is similar to METIS_NodeND, except for how it treats the leaf nodes. METIS_NodeND orders the leaves of the separator tree with MMD, ignoring the rest of the matrix when ordering a single leaf. This routine orders the whole matrix with CAMD, CSYMAMD, or CCOLAMD, all at once, when the graph partitioning is done.

25.2 cholmod_metis: interface to METIS nested dissection

```
int cholmod_metis
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to order */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    int postorder,      /* if TRUE, follow with etree or coletree postorder */
    /* ---- output --- */
    int32_t *Perm,      /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_metis (cholmod_sparse *, int64_t *, size_t, int, int64_t *,
    cholmod_common *) ;
```

Purpose: CHOLMOD wrapper for the `METIS_NodeND` ordering routine. Creates `A+A'`, `A*A'` or `A(:,f)*A(:,f)'` and then calls `METIS_NodeND` on the resulting graph. This routine is comparable to `cholmod_nested_dissection`, except that it calls `METIS_NodeND` directly, and it does not return the separator tree.

25.3 `cholmod_camd`: interface to CAMD

```
int cholmod_camd
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to order */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    /* ---- output --- */
    int32_t *Cmember,   /* size nrow.  see cholmod_ccolamd above */
    int32_t *Perm,      /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_camd (cholmod_sparse *, int64_t *, size_t,
    int64_t *, int64_t *, cholmod_common *) ;
```

Purpose: CHOLMOD interface to the CAMD ordering routine. Finds a permutation `p` such that the Cholesky factorization of `A(p,p)` is sparser than `A`. If `A` is unsymmetric, `A*A'` is ordered. If `Cmember[i]=c` then node `i` is in set `c`. All nodes in set 0 are ordered first, followed by all nodes in set 1, and so on.

25.4 `cholmod_ccolamd`: interface to CCOLAMD

```
int cholmod_ccolamd
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to order */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    int32_t *Cmember,   /* size A->nrow.  Cmember [i] = c if row i is in the
                         * constraint set c.  c must be >= 0.  The # of
                         * constraint sets is max (Cmember) + 1.  If Cmember is
                         * NULL, then it is interpretted as Cmember [i] = 0 for
                         * all i */
    /* ---- output --- */
    int32_t *Perm,      /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_ccolamd (cholmod_sparse *, int64_t *, size_t,
    int64_t *, int64_t *, cholmod_common *) ;
```

Purpose: CHOLMOD interface to the CCOLAMD ordering routine. Finds a permutation `p` such that the Cholesky factorization of `A(p,:)*A(p,:)'` is sparser than `A*A'`. The column elimination is found and postordered, and the CCOLAMD ordering is then combined with its postordering. `A` must be unsymmetric. If `Cmember[i]=c` then node `i` is in set `c`. All nodes in set 0 are ordered first, followed by all nodes in set 1, and so on.

25.5 `cholmod_csymamd`: interface to CSYMAMD

```
int cholmod_csymamd
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to order */
    /* ---- output --- */
    int32_t *Cmember,   /* size nrow.  see cholmod_ccolamd above */
    int32_t *Perm,      /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *Common
) ;

int cholmod_l_csymamd (cholmod_sparse *, int64_t *,
    int64_t *, cholmod_common *) ;
```

Purpose: CHOLMOD interface to the CSYMAMD ordering routine. Finds a permutation `p` such that the Cholesky factorization of `A(p,p)` is sparser than `A`. The elimination tree is found and postordered, and the CSYMAMD ordering is then combined with its postordering. If `A` is unsymmetric, `A+A'` is ordered (`A` must be square). If `Cmember[i]=c` then node `i` is in set `c`. All nodes in set 0 are ordered first, followed by all nodes in set 1, and so on.

25.6 `cholmod_bisect`: graph bisector

```
int64_t cholmod_bisect  /* returns # of nodes in separator */
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to bisect */
    int32_t *fset,      /* subset of 0:(A->ncol)-1 */
    size_t fsize,       /* size of fset */
    int compress,       /* if TRUE, compress the graph first */
    /* ---- output --- */
    int32_t *Partition, /* size A->nrow.  Node i is in the left graph if
                         * Partition [i] = 0, the right graph if 1, and in the
                         * separator if 2. */
    /* --------------- */
    cholmod_common *Common
) ;

int64_t cholmod_l_bisect (cholmod_sparse *, int64_t *, size_t, int, int64_t *,
    cholmod_common *) ;
```

Purpose: Finds a node bisector of `A`, `A*A'`, `A(:,f)*A(:,f)'`: a set of nodes that partitions the graph into two parts. Compresses the graph first, ensures the graph is symmetric with no diagonal entries, and then calls METIS.

25.7 `cholmod_metis_bisector`: interface to METIS node bisector

```
int64_t cholmod_metis_bisector  /* returns separator size */
(
    /* ---- input ---- */
    cholmod_sparse *A,  /* matrix to bisect */
    int32_t *Anw,       /* size A->nrow, node weights, can be NULL, */
                        /* which means the graph is unweighted. */
    int32_t *Aew,       /* size nz, edge weights (silently ignored). */
                        /* This option was available with METIS 4, but not */
                        /* in METIS 5.  This argument is now unused, but */
                        /* it remains for backward compatibilty, so as not */
                        /* to change the API for cholmod_metis_bisector. */
    /* ---- output --- */
    int32_t *Partition, /* size A->nrow */
    /* --------------- */
    cholmod_common *Common
) ;

int64_t cholmod_l_metis_bisector (cholmod_sparse *, int64_t *, int64_t *,
    int64_t *, cholmod_common *) ;
```

Purpose: Finds a set of nodes that bisects the graph of `A` or `A*A'` (a direct interface to `METIS_NodeComputeSeparator`).

The input matrix `A` must be square, symmetric (with both upper and lower parts present) and with no diagonal entries. These conditions are not checked. Use cholmod_bisect to check these conditions.

25.8 cholmod_collapse_septree: prune a separator tree

```
int64_t cholmod_collapse_septree
(
    /* ---- input ---- */
    size_t n,           /* # of nodes in the graph */
    size_t ncomponents, /* # of nodes in the separator tree (must be <= n) */
    double nd_oksep,    /* collapse if #sep >= nd_oksep * #nodes in subtree */
    size_t nd_small,    /* collapse if #nodes in subtree < nd_small */
    /* ---- in/out --- */
    int32_t *CParent,   /* size ncomponents; from cholmod_nested_dissection */
    int32_t *Cmember,   /* size n; from cholmod_nested_dissection */
    /* --------------- */
    cholmod_common *Common
) ;

int64_t cholmod_l_collapse_septree (size_t, size_t, double, size_t, int64_t *,
    int64_t *, cholmod_common *) ;
```

Purpose: Prunes a separator tree obtained from `cholmod_nested_dissection`.

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Applic.*, 17(4):886–905, 1996.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):381–388, 2004.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenny, and D. Sorensen. *LAPACK Users' Guide, 3rd ed.* SIAM, 1999.

[4] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 8xx: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, submitted in 2006.

[5] T. A. Davis. Algorithm 849: A concise sparse Cholesky algorithm. *ACM Trans. Math. Softw.*, 31(4):587–591, 2005.

[6] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):377–380, 2004.

[7] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):353–376, 2004.

[8] T. A. Davis and W. W. Hager. Modifying a sparse Cholesky factorization. *SIAM J. Matrix Anal. Applic.*, 20(3):606–627, 1999.

[9] T. A. Davis and W. W. Hager. Multiple-rank modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Applic.*, 22(4):997–1013, 2001.

[10] T. A. Davis and W. W. Hager. Row modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Applic.*, 26(3):621–639, 2005.

[11] T. A. Davis and W. W. Hager. Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Trans. Math. Softw.*, submitted in 2006.

[12] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level-3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[13] J. R. Gilbert, X. S. Li, E. G. Ng, and B. W. Peyton. Computing row and column counts for sparse QR and LU factorization. *BIT*, 41(4):693–710, 2001.

[14] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Applic.*, 13(1):333–356, 1992.

[15] J. R. Gilbert, E. G. Ng, and B. W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Anal. Applic.*, 15(4):1075–1091, 1994.

[16] N. I. M. Gould, Y. Hu, and J. A. Scott. Complete results from a numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. Technical Report Internal report 2005-1 (revision 1), CCLRC, Rutherford Appleton Laboratory, 2005.

[17] N. I. M. Gould, Y. Hu, and J. A. Scott. A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Softw.*, to appear.

[18] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[19] J. W. H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Trans. Math. Softw.*, 12(2):127–148, 1986.

[20] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Applic.*, 11(1):134–172, 1990.

[21] E. Ng and B. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14:1034–1056, 1993.