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

Timothy A. Davis
Dept. of Computer and Information Science and Engineering
Univ. of Florida, Gainesville, FL

VERSION 1.7.3, Jan 25, 2011

Abstract

CHOLMOD\textsuperscript{1} is a set of routines for factorizing sparse symmetric positive definite matrices of the form $A$ or $AA^T$, updating/downdating a sparse Cholesky factorization, solving linear systems, updating/downdating the solution to the triangular system $Lx = b$, 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 $LDL^T$ factorization method that can factorize symmetric indefinite matrices if all of their leading submatrices are well-conditioned ($D$ 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.

CHOLMOD Copyright \textcopyright\textsuperscript{2005-2011} by Timothy A. Davis. Portions are also copyrighted by William W. Hager (the Modify Module), and the University of Florida (the Partition and Core Modules). All Rights Reserved. Some of CHOLMOD's Modules are distributed under the GNU General Public License, and others under the GNU Lesser General Public License. Refer to each Module for details. CHOLMOD is also available under other licenses that permit its use in proprietary applications; contact the authors for details. See http://www.cise.ufl.edu/research/sparse for the code and all documentation, including this User Guide.

\textsuperscript{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 .......................... 7
2 Primary routines and data structures 8
3 Simple example program 10
4 Installation of the C-callable library 11
5 Using CHOLMOD in MATLAB 14
   5.1 analyze: order and analyze ........................................... 15
   5.2 bisect: find a node separator ........................................ 16
   5.3 chol2: same as chol .................................................. 16
   5.4 cholmod2: supernodal backslash ................................... 17
   5.5 cholmod_demo: a short demo program .............................. 18
   5.6 cholmod_make: compile CHOLMOD in MATLAB .................... 18
   5.7 etree: same as etree ................................................. 20
   5.8 graph_demo: graph partitioning demo ............................. 20
   5.9 lchol: LL^T factorization ........................................... 22
   5.10 ld chol: LDL^T factorization .................................... 22
   5.11 ldisolve: solve using an LDL^T factorization ............... 23
   5.12 ldsplit: split an LDL^T factorization .......................... 23
   5.13 ldupdate: update/downdate an LDL^T factorization ........... 24
   5.14 mread: read a sparse or dense matrix from a Matrix Market file ............................................. 25
   5.15 mwrite: write a sparse or dense matrix to a Matrix Market file ............................................... 25
   5.16 metis: order with METIS ............................................. 26
   5.17 nesdis: order with CHOLMOD nested dissection .............. 27
   5.18 resymbol: re-do symbolic factorization .......................... 28
   5.19 smult: sparse matrix times dense matrix ...................... 28
   5.20 spsym: determine symmetry ...................................... 29
   5.21 sparse2: same as sparse .......................................... 31
   5.22 symbfact2: same as symbfact .................................. 32
6 Installation for use in MATLAB 33
   6.1 cholmod_make: compiling CHOLMOD in MATLAB ............... 33
   6.2 Unix make for compiling CHOLMOD ............................... 33
7 Integer and floating-point types, and notation used 34
8 The CHOLMOD Modules, objects, and functions 36
   8.1 Core Module: basic data structures and definitions .......... 37
      8.1.1 cholmod_common: parameters, statistics, and workspace ............. 37
      8.1.2 cholmod_sparse: a sparse matrix in compressed column form .... 38
      8.1.3 cholmod_factor: a symbolic or numeric factorization .......... 39
      8.1.4 cholmod_dense: a dense matrix ................................ 39
      8.1.5 cholmod_triplet: a sparse matrix in “triplet” form .......... 40
      8.1.6 Memory management routines .................................. 40
   8.2 Check Module: print/check the CHOLMOD objects .......... 41
8.3 Cholesky Module: sparse Cholesky factorization .................................... 42
8.4 Modify Module: update/downdate a sparse Cholesky factorization .......... 43
8.5 MatrixOps Module: basic sparse matrix operations ............................... 43
8.6 Supernodal Module: supernodal sparse Cholesky factorization ............... 44
8.7 Partition Module: graph-partitioning-based orderings ........................... 44

9 CHOLMOD naming convention, parameters, and return values 45

10 Core Module: cholmod_common object ................................................. 46
  10.1 Constant definitions ................................................................. 46
  10.2 cholmod_common: parameters, statistics, and workspace .................... 48
  10.3 cholmod_start: start CHOLMOD .................................................. 58
  10.4 cholmod_finish: finish CHOLMOD ................................................ 58
  10.5 cholmod_defaults: set default parameters ..................................... 58
  10.6 cholmod_maxrank: maximum update/downdate rank ............................ 58
  10.7 cholmod_allocate_work: allocate workspace ................................... 59
  10.8 cholmod_free_work: free workspace ............................................. 59
  10.9 cholmod_clear_flag: clear Flag array .......................................... 59
  10.10 cholmod_error: report error ...................................................... 60
  10.11 cholmod_dbound: bound diagonal of \( L \) ..................................... 60
  10.12 cholmod_hypot: \( \sqrt{x^2+y^2} \) .................................................... 60
  10.13 cholmod_divcomplex: complex divide .......................................... 61

11 Core Module: cholmod_sparse object ................................................. 62
  11.1 cholmod_sparse: compressed-column sparse matrix ........................... 62
  11.2 cholmod_allocate_sparse: allocate sparse matrix ............................. 63
  11.3 cholmod_free_sparse: free sparse matrix ....................................... 63
  11.4 cholmod_reallocate_sparse: reallocate sparse matrix ......................... 63
  11.5 cholmod_nnz: number of entries in sparse matrix ............................. 64
  11.6 cholmod_speye: sparse identity matrix ......................................... 64
  11.7 cholmod_szeros: sparse zero matrix ............................................. 64
  11.8 cholmod_transpose: transpose sparse matrix ................................... 65
  11.9 cholmod_ptranspose: transpose/permute sparse matrix ....................... 65
  11.10 cholmod_sort: sort columns of a sparse matrix ............................... 65
  11.11 cholmod_transpose_unsym: transpose/permute unsymmetric sparse matrix .... 66
  11.12 cholmod_transpose_sym: transpose/permute symmetric sparse matrix ....... 67
  11.13 cholmod_band: extract band of a sparse matrix .............................. 68
  11.14 cholmod_band_inplace: extract band, in place ................................ 68
  11.15 cholmod_aat: compute \( AA^T \) .................................................. 69
  11.16 cholmod_copy_sparse: copy sparse matrix ..................................... 69
  11.17 cholmod_copy: copy (and change) sparse matrix ............................. 70
  11.18 cholmod_add: add sparse matrices ............................................. 71
  11.19 cholmod_sparse xtype: change sparse xtype .................................... 71

12 Core Module: cholmod_factor object ................................................. 72
  12.1 cholmod_factor object: a sparse Cholesky factorization ...................... 72
  12.2 cholmod_free_factor: free factor ............................................... 75
  12.3 cholmod_allocate_factor: allocate factor ...................................... 75
12.4 cholmod_reallocate_factor: reallocate factor ........................................... 75
12.5 cholmod_change_factor: change factor ....................................................... 76
12.6 cholmod_pack_factor: pack the columns of a factor .................................... 78
12.7 cholmod_reallocate_column: reallocate one column of a factor .................... 78
12.8 cholmod_factor_to_sparse: sparse matrix copy of a factor ............................ 79
12.9 cholmod_copy_factor: copy factor ............................................................. 79
12.10 cholmod_factor xtype: change factor xtype .............................................. 79

13 Core Module: cholmod_dense object 80
13.1 cholmod_dense object: a dense matrix ...................................................... 80
13.2 cholmod_allocate_dense: allocate dense matrix ......................................... 80
13.3 cholmod_free_dense: free dense matrix ...................................................... 80
13.4 cholmod_zeros: dense zero matrix ............................................................ 81
13.5 cholmod_ones: dense matrix, all ones ....................................................... 81
13.6 cholmod_eye: dense identity matrix .......................................................... 81
13.7 cholmod_sparse_to_dense: dense matrix copy of a sparse matrix ................. 82
13.8 cholmod_dense_to_sparse: sparse matrix copy of a dense matrix ................... 82
13.9 cholmod_copy_dense: copy dense matrix .................................................... 82
13.10 cholmod_copy_dense2: copy dense matrix (preallocated) ............................ 83
13.11 cholmod_dense xtype: change dense matrix xtype .................................... 83

14 Core Module: cholmod_triplet object 84
14.1 cholmod_triplet object: sparse matrix in triplet form .................................. 84
14.2 cholmod_allocate_triplet: allocate triplet matrix ....................................... 85
14.3 cholmod_free_triplet: free triplet matrix ................................................... 85
14.4 cholmod_reallocate_triplet: reallocate triplet matrix .................................. 86
14.5 cholmod_sparse_to_triplet: triplet matrix copy of a sparse matrix ............... 86
14.6 cholmod_triplet_to_sparse: sparse matrix copy of a triplet matrix ................. 86
14.7 cholmod_copy_triplet: copy triplet matrix .................................................. 87
14.8 cholmod_triplet xtype: change triplet xtype ............................................. 87

15 Core Module: memory management 88
15.1 cholmod_malloc: allocate memory ............................................................ 88
15.2 cholmodcalloc: allocate and clear memory ............................................... 88
15.3 cholmod_free: free memory .................................................................... 89
15.4 cholmod realloc: reallocate memory .......................................................... 89
15.5 cholmod realloc multiple: reallocate memory ........................................... 90

16 Check Module routines 91
16.1 cholmod_check_common: check Common object ........................................ 91
16.2 cholmod_print_common: print Common object .......................................... 91
16.3 cholmod_check_sparse: check sparse matrix ............................................. 92
16.4 cholmod_print_sparse: print sparse matrix .............................................. 92
16.5 cholmod_check_dense: check dense matrix ............................................. 93
16.6 cholmod_print_dense: print dense matrix ............................................... 93
16.7 cholmod_check_factor: check factor ......................................................... 94
16.8 cholmod_print_factor: print factor ............................................................ 94
16.9 cholmod_check_triplet: check triplet matrix ............................................ 95
16.10 cholmod_print_triplet: print triplet matrix ........................................ 95
16.11 cholmod_check_subset: check subset .................................................. 96
16.12 cholmod_print_subset: print subset .................................................... 96
16.13 cholmod_check_perm: check permutation ............................................. 97
16.14 cholmod_print_perm: print permutation .............................................. 97
16.15 cholmod_check_parent: check elimination tree ...................................... 98
16.16 cholmod_print_parent: print elimination tree ....................................... 98
16.17 cholmod_read_triplet: read triplet matrix from file ............................. 99
16.18 cholmod_read_sparse: read sparse matrix from file ............................... 100
16.19 cholmod_read_dense: read dense matrix from file ................................. 101
16.20 cholmod_read_matrix: read a matrix from file .................................... 101
16.21 cholmod_write_sparse: write a sparse matrix to a file ......................... 102
16.22 cholmod_write_dense: write a dense matrix to a file ............................ 102

17 Cholesky Module routines .......................................................................... 103
  17.1 cholmod_analyze: symbolic factorization .............................................. 103
  17.2 cholmod_factorize: numeric factorization ............................................ 105
  17.3 cholmod_analyze_p: symbolic factorization, given permutation ................ 106
  17.4 cholmod_factorize_p: numeric factorization, given permutation ................ 106
  17.5 cholmod_solve: solve a linear system ................................................ 107
  17.6 cholmod_spsolve: solve a linear system ............................................. 107
  17.7 cholmod_etree: find elimination tree .................................................. 108
  17.8 cholmod_rowcolcounts: nonzeros counts of a factor ............................... 108
  17.9 cholmod_analyze_ordering: analyze a permutation ................................. 109
  17.10 cholmod_amd: interface to AMD ......................................................... 110
  17.11 cholmod_colamd: interface to COLAMD ............................................. 110
  17.12 cholmod_rowfac: row-oriented Cholesky factorization ......................... 111
  17.13 cholmod_rowfac_mask: row-oriented Cholesky factorization .................. 112
  17.14 cholmod_row_subtree: pattern of row of a factor ................................. 113
  17.15 cholmod_row_lsubtree: pattern of row of a factor ............................... 114
  17.16 cholmod_resymbol: re-do symbolic factorization .................................. 115
  17.17 cholmod_resymbol_noperm: re-do symbolic factorization ...................... 115
  17.18 cholmod_postorder: tree postorder .................................................... 116
  17.19 cholmod_rcond: reciprocal condition number ...................................... 116

18 Modify Module routines .............................................................................. 117
  18.1 cholmod_updown: update/downdate ..................................................... 117
  18.2 cholmod_updown_solve: update/downdate .......................................... 118
  18.3 cholmod_updown_mark: update/downdate ............................................ 119
  18.4 cholmod_updown_mask: update/downdate ............................................ 119
  18.5 cholmod_rowadd: add row to factor ................................................... 120
  18.6 cholmod_rowadd_solve: add row to factor ......................................... 120
  18.7 cholmod_rowdel: delete row from factor ............................................ 121
  18.8 cholmod_rowdel_solve: delete row from factor .................................... 121
  18.9 cholmod_rowadd_mark: add row to factor ........................................... 122
  18.10 cholmod_rowdel_mark: delete row from factor .................................... 122
19 MatrixOps Module routines 123
19.1 cholmod_drop: drop small entries ................................. 123
19.2 cholmod_norm_dense: dense matrix norm ...................... 123
19.3 cholmod_norm_sparse: sparse matrix norm .................... 123
19.4 cholmod_scale: scale sparse matrix ............................ 124
19.5 cholmod_sdmult: sparse-times-dense matrix .................. 125
19.6 cholmod_ssmult: sparse-times-sparse matrix ................. 125
19.7 cholmod_submatrix: sparse submatrix ........................ 126
19.8 cholmod_horzcat: horizontal concatenation .................. 127
19.9 cholmod vertcat: vertical concatenation ...................... 127
19.10 cholmod_symmetry: compute the symmetry of a matrix .......... 128

20 Supernodal Module routines 130
20.1 cholmod_super_symbolic: supernodal symbolic factorization ........ 130
20.2 cholmod_super_numeric: supernodal numeric factorization ...... 131
20.3 cholmod_super_lsolve: supernodal forward solve ............... 132
20.4 cholmod_super_ltsolve: supernodal backsolve ................ 132

21 Partition Module routines 133
21.1 cholmod_nested_dissection: nested dissection ordering .......... 133
21.2 cholmod_metis: interface to METIS nested dissection .......... 134
21.3 cholmod_camd: interface to CAMD .............................. 135
21.4 cholmod_ccolamd: interface to CCOLAMD ...................... 136
21.5 cholmod_csymamd: interface to CSYMAMD ..................... 136
21.6 cholmod_bisect: graph bisector ................................ 137
21.7 cholmod_metis_bisector: interface to METIS node bisector .... 137
21.8 cholmodCollapse septree: prune a separator tree .............. 138
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. 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 $\text{LDL}^T$ 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 has been submitted to the ACM Transactions on Mathematical Software: [4, 11].

CHOLMOD 1.0 replaces $\text{chol}$ (the sparse case), $\text{symbfact}$, and $\text{etree}$ in MATLAB 7.2 (R2006a), and is used for $\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$ when $\mathbf{A}$ is symmetric positive definite [14]. It will replace $\text{sparse}$ 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.

---

2Some support is provided for symmetric indefinite matrices.
2 Primary routines and data structures

Five primary CHOLMOD routines are required to factorize $A$ or $AA^T$ and solve the related system $Ax = b$ or $AA^Tx = b$, 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, $LL^T$ or $LDL^T$ using either the symbolic factorization from **cholmod_analyze** or the numerical factorization from a prior call to **cholmod_factorize**.

4. **cholmod_solve**: Solves $Ax = b$, or many other related systems, where $x$ and $b$ are dense matrices. The **cholmod_sp.solve** routine handles the sparse case. Any mixture of real and complex $A$ and $b$ are allowed.

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

Additional routines are also required to create and destroy the matrices $A$, $x$, $b$, and the $LL^T$ or $LDL^T$ 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 10 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:
   - $A->i$, 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 $A->i \ [A->p \ [j] \ldots \ A->p \ [j+1]-1]$. 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 11 for more details.

3. **cholmod_factor**: A symbolic or numeric factorization, either real, complex, or zomplex. It can be either an $LL^T$ or $LDL^T$ factorization, and either simplicial or supernodal. You will normally not need to examine its contents. See Section 12 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 13 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 14 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.
#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 */
    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) */
    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.
4 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 `-DNPARTITION` 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 `-DNPARTITION` flag.

- **UFconfig**: a single place where all sparse matrix packages authored or co-authored by Davis are configured. Also includes a version of the `xerbla` routine for the BLAS.

Three other packages are required for optimal performance:

- **METIS 4.0.1** is 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.

You must first obtain and install METIS, LAPACK, and the BLAS. Next edit the system-dependent configurations in the `UFconfig/UFconfig.mk` file. Sample configurations are provided for Linux, Macintosh, Sun Solaris, SGI IRIX, IBM AIX, and the DEC/Compaq Alpha. The most important configuration is the location of the BLAS, LAPACK, and METIS packages, since in its default configuration CHOLMOD cannot be compiled without them.

Here are the various parameters that you can control in your `UFconfig/UFconfig.mk` file:

- **CC** = your C compiler, such as `cc`.

- **CFLAGS** = optimization flags, such as `-O`.

- **RANLIB** = your system’s `ranlib` program, if needed.

- **AR** = the command to create a library (such as `ar`).

- **RM** = the command to delete a file.
- MV = the command to rename a file.
- F77 = the command to compile a Fortran program (optional).
- F77FLAGS = the Fortran compiler flags (optional).
- F77LIB = the Fortran libraries (optional).
- LIB = basic libraries, such as -lm.
- MEX = the command to compile a MATLAB mexFunction.
- BLAS = your BLAS library.
- LAPACK = your LAPACK library.
- XERBLA = a library containing the BLAS xerbla routine, if required.
- METIS_PATH = the path to your copy of the METIS 4.0.1 source code.
- METIS = your METIS library.
- CHOLMOD_CONFIG = configuration settings specific to CHOLMOD.

CHOLMOD’s specific settings are given by the CHOLMOD_CONFIG string:
- -DNCHECK: do not include the Check module. License: GNU LGPL.
- -DNCHOLESKY: do not include the Cholesky module. License: GNU LGPL.
- -DNPARTITION: do not include the Partition module. License: GNU LGPL.
- -DNGPL: do not include any GNU GPL Modules in the CHOLMOD library.
- -DNMATRIXOPS: do not include the MatrixOps module. License: GNU GPL.
- -DNMODIFY: do not include the Modify module. License: GNU GPL.
- -DNSUPERNODAL: do not include the Supernodal module. License: GNU GPL.
- -DNPEDPRINT: do not print anything.
- -D’LONGBLAS=long’ or -DLONGBLAS='long long’ defines the integers used by LAPACK and the BLAS (defaults to int).
- -DNSUNPERF: for Solaris only. If defined, do not use the Sun Performance Library.
- -DNLARGEFILE: CHOLMOD now assumes support for large files (2GB or larger). If this causes problems, you can compile CHOLMOD with -DNLARGEFILE. To use large files, you should #include "cholmod.h" (or at least #include "cholmod_io64.h") before any other #include statements, in your application that uses CHOLMOD. You may need to use fopen64 to create a file pointer to pass to CHOLMOD, if you are using a non-gcc compiler.
Type `make` in the CHOLMOD directory. The AMD, COLAMD, CAMD, CCOLAMD, and CHOLMOD libraries will be compiled, as will the C version of the null-output `xerbla` routine in case you need it. No Fortran compiler is required in this case. A short demo program will be compiled and tested on a few matrices. The residuals should all be small. Compare your output with the CHOLMOD/Demo/make.out file.

CHOLMOD is now ready for use in your own applications. You must link your programs with the CHOLMOD/Lib/libcholmod.a, AMD/Lib/libamd.a, COLAMD/libcolamd.a, CAMD/libcamd.a, CCOLAMD/libccolamd.a, metis-4.0/libmetis.a, LAPACK, and BLAS libraries, as well as the `xerbla` library if you need it (UFconfig/xerlib/libcerbla.a for the C version or UFconfig/xerlib/libxerbla.a for the Fortran version). Your compiler needs to know the location of the CHOLMOD Include directory, so that it can find the `cholmod.h` include file, by adding the `-ICHOLMOD/Include` to your C compiler options (modified appropriately to reflect the location of your copy of CHOLMOD).
5 Using CHOLMOD in MATLAB

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

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

Each function is described in the next sections.
5.1 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:

\[
\text{count} = \text{symbfact2} \left(A (p,p)\right) \quad \text{if ordering A}
\]
\[
\text{count} = \text{symbfact2} \left(A (p,:),\text{'row'}\right) \quad \text{if ordering A*A'}
\]
\[
\text{count} = \text{symbfact2} \left(A (:,p),\text{'col'}\right) \quad \text{if ordering A'*A}
\]

CHOLMOD uses the following ordering strategy:

\begin{align*}
\text{k} &= 0: \text{Try AMD. If that ordering gives a flop count} \geq 500 \times \text{nnz(L)} \\
& \quad \text{and a fill-in of} \ \text{nnz(L)} \
& \quad \geq 5 \times \text{nnz(C)}, \text{then try METIS_NodeND (where C} = A, A*A', \text{or A'*A the matrix being ordered. Selects the best ordering tried. This is the default.}}
\end{align*}

if \text{k} > 0, \text{then multiple orderings are attempted.}

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

\begin{align*}
\text{k} &= -1: \text{just use AMD} \\
\text{k} &= -2: \text{just use METIS} \\
\text{k} &= -3: \text{just use NESDIS}
\end{align*}

The method returning the smallest \text{nnz(L)} is used for \text{p} and \text{count}.

\text{k} = 4 \text{ takes much longer than (say) k} = 0, \text{but it can reduce nnz(L) by a typical 5% to 10%. k} = 5 \text{ 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:

\begin{align*}
\text{k} &= 1 \text{ to 4: just try AMD} \\
\text{k} &= 5 \text{ to 8: also try the natural ordering (p = 1:n)} \\
\text{k} &= 9: \text{also try COLAMD if ordering A'*A or A*A', (AMD if ordering A).} \\
\text{k} &> 9 \text{ is treated as k} = 9
\end{align*}
5.2 bisect: find a node separator

BISECT computes a node separator based on METIS_NodeComputeSeparator.

Example:

\[
\begin{align*}
\text{s = bisect}(A) & \quad \text{bisects A. Uses tril(A) and assumes A is symmetric.} \\
\text{s = bisect}(A,'\text{sym}') & \quad \text{the same as p=bisect(A).} \\
\text{s = bisect}(A,'\text{col}') & \quad \text{bisects A'*A.} \\
\text{s = bisect}(A,'\text{row}') & \quad \text{bisects A*A'.}
\end{align*}
\]

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 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse

5.3 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:

\[
\begin{align*}
\text{R = chol2 (A)} & \quad \text{same as R = chol (A), just faster} \\
\text{[R,p] = chol2 (A)} & \quad \text{save as [R,p] = chol(A), just faster} \\
\text{[R,p,q] = chol2 (A)} & \quad \text{factorizes A(q,q) into R'*R, where q is a fill-reducing ordering}
\end{align*}
\]

A must be sparse.

See also LCHOL, LDLCHOL, CHOL, LDLUPDATE.
Copyright 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse
5.4 cholmod2: supernodal backslash

CHOLMOD2 supernodal sparse Cholesky backslash, \( x = A\backslash b \)

Example:
\[
x = \text{cholmod2}(A,b)
\]

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, \([x,\text{stats}] = \text{cholmod2}(A,b)\), returns statistics:

- \( \text{stats}(1) \): estimate of the reciprocal of the condition number
- \( \text{stats}(2) \): ordering used:
- \( \text{stats}(3) \): \( \text{nnz}(L) \)
- \( \text{stats}(4) \): flop count in Cholesky factorization. Excludes solution of upper/lower triangular systems, which can be easily computed from \( \text{stats}(3) \) (roughly \( 4*\text{nnz}(L)*\text{size}(b,2) \)).
- \( \text{stats}(5) \): 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 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse
5.5 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 UF sparse matrix collection.

Example:
  cholmod_demo

See also BENCH
  Copyright 2006-2007, Timothy A. Davis
  http://www.cise.ufl.edu/research/sparse
  try_matrix: try a matrix with CHOLMOD

5.6 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. All but METIS are distributed with CHOLMOD. To compile CHOLMOD to use METIS you must first place a copy of the metis-4.0 directory (METIS version 4.0.1) in same directory that contains the AMD, COLAMD, CCOLAMD, and CHOLMOD directories. Next, type

  cholmod_make

in the MATLAB command window. Alternatively, use this command:

  cholmod_make ('path to your copy of metis-4.0 here')

See http://www-users.cs.umn.edu/~karypis/metis for a copy of METIS 4.0.1. If you do not have METIS, use either of the following:

  cholmod_make ('')
  cholmod_make ('no metis')

You must type the cholmod_make command while in the CHOLMOD/MATLAB directory.

See also analyze, bisect, chol2, cholmod2, etree2, lchol, ldichol, ldisolve, ldupdate, metis, spsym, nesdis, septe, symbsolve, spmv, spmv, symfct2, mread, mwrite
  Copyright 2006-2007, Timothy A. Davis
  http://www.cise.ufl.edu/research/sparse
Determine the METIS path, and whether or not METIS is available
fix the METIS 4.0.1 rename.h file

-------------------------------------------------------------------
BLAS option
This is exceedingly ugly. The MATLAB mex command needs to be told where to find the LAPACK and BLAS libraries, which is a real portability nightmare.

compile each library source file
compile each mexFunction
clean up

determine the MATLAB version, and return it as a double.
5.7 etree2: same as etree

ETREE2 sparse elimination tree.
Finds the elimination tree of A, A'*A, or A*A', and optionally 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,:)') or chol(A(p,:)*A(p,:)'):

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

See also TREELAYOUT, TREEPLOT, ETREEPLOT, ETREE

Copyright 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse

5.8 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 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse

my_gplot : like gplot, just a lot faster
5.9 lchol: $L L^T$ 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 $L*L'$ and LDL' factorization methods use tril($A$). $A$ must be sparse.

Example:
L = lchol (A) same as $L = \text{chol} (A')'$, just faster
[L,p] = lchol (A) save as $[R,p] = \text{chol}(A')$ ; $L=R'$, 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 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse

5.10 ldlchol: $LDL^T$ 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 $L*L'$ 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 $[R,p] = \text{chol}(A)$, 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,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 $[L,D] = \text{ldlsplit} (LD)$. 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 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse
5.11  ldlsolve: solve using an LDL$^T$ factorization

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

Example:
\[ x = ldlsolve (\text{LD}, b) \]

solves the system \( L \cdot D \cdot L'^T x = b \) for \( x \). This is equivalent to

\[
[L, D] = ldlsplit (\text{LD}) ;
\]
\[
x = L'^T \cdot (D \cdot (L \cdot b)) ;
\]

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

See also LDLCHOL, LDLUPDATE, LDLSPLIT
Copyright 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse

5.12  ldlsplit: split an LDL$^T$ factorization

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

Example:
\[
[L, D] = ldlsplit (\text{LD})
\]

LD contains an LDL'$ factorization, computed with \( LD = \text{ldlchol}(A) \), 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 \( L \cdot D \cdot L'^T = A \).

See also LDLCHOL, LDLSOLVE, LDLUPDATE.
Copyright 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse
5.13 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 (LD*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 property holds.

===========================================================================
Copyright 2006-2007, Timothy A. Davis, William W. Hager
http://www.cise.ufl.edu/research/sparse
5.14 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 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse

5.15 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 2006-2007, Timothy A. Davis
5.16 \texttt{metis}: order with METIS

METIS nested dissection ordering via METIS\_NodeND.

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

A must be square for \(p=\text{metis}(A)\) or \(\text{metis}(A,'sym')\)

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

See also NESDIS, BISECT
Copyright 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse
5.17 nesdis: order with CHOLMOD nested dissection

**NESDIS nested dissection ordering via CHOLMOD’s nested dissection.**

Example:
- `p = nesdis(A)` returns `p` such that `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 2006–2007, Timothy A. Davis

http://www.cise.ufl.edu/research/sparse
5.18 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 \text{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 (\text{lchol} or \text{ldlchol}). resymbol is useful after a series of downdates via \text{ldlupdate}, since downdates do not remove any entries in L. The numerical values of A are ignored; only its nonzero pattern is used.

See also \text{LCHOL}, \text{LDLUPDATE}

Copyright 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse

5.19 sdmult: sparse matrix times dense matrix

SDMULT sparse matrix times dense matrix

Compute \( C = S \cdot F \) or \( S' \cdot F \) 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 \( C = S \cdot F \) when F has many columns.

Example:
\[
C = \text{sdmult} (S,F) ; \quad C = S \cdot F \\
C = \text{sdmult} (S,F,0) ; \quad C = S \cdot F \\
C = \text{sdmult} (S,F,1) ; \quad C = S' \cdot F
\]

See also \text{MTIMES}

Copyright 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse
5.20  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
endif
elseif (nnz (A-A.')) == 0)
if (posdiag)
result = 6 ; % symmetric with positive diagonal
else
result = 3 ; % symmetric, nonpositive diagonal
endif
elseif (nnz (A+A.')) == 0)
result = 5 ; % skew symmetric
else
result = 2 ; % unsymmetric
endif
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 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse
5.21  \texttt{sparse2: same as sparse}

\texttt{SPARSE2 replacement for SPARSE}

Example:
\texttt{S = sparse2 (i,j,s,m,n,nzmax)}

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

\texttt{[S Z] = sparse2 (i,j,s,m,n,nzmax)}

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

\begin{verbatim}
  S = sparse (i,j,s, ...)
  Z = spones (sparse (i,j,1, ...)) - spones (S)
\end{verbatim}

See also \texttt{spars}.

Copyright 2006-2007, Timothy A. Davis

http://www.cise.ufl.edu/research/sparse
SYMBFACT2 symbolic factorization

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

Example:

- `count = symbfact2(A)` returns row counts of $R=\text{chol}(A)$
- `count = symbfact2(A,'col')` returns row counts of $R=\text{chol}(A'*A)$
- `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 $R=\text{chol}(A*A')$

The flop count for a subsequent $LL'$ factorization is $\text{sum(count.^2)}$

```
[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 $\text{chol}(A)$ for the symmetric case, $\text{chol}(A'*A)$ for the 'col' case, or $\text{chol}(A*A')$ for the 'row' case.

`symbfact2(A)` and `symbfact2(A,'sym')` uses the upper triangular part of $A$ ($\text{triu}(A)$) 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 $\text{nnz}(A)+n$ where $n$ is the dimension of $R$, and memory proportional to $\text{nnz}(A)$. Computing the 5th argument takes more time and memory, both $O(\text{nnz}(L))$. 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 2006-2007, Timothy A. Davis
http://www.cise.ufl.edu/research/sparse
6 Installation for use in MATLAB

If you wish to use METIS within CHOLMOD, you should first obtain a copy of METIS 4.0.1. See http://www-users.cs.umn.edu/~karypis/metis. Place your copy of the metis-4.0 directory (folder, for Windows users) in the same directory that contains your copy of the CHOLMOD directory. If you do not have METIS, however, you can still use CHOLMOD. Some of the CHOLMOD functions will not be available (metis, bisect, and nesdis), and you may experience higher fill-in for large matrices (particularly those arising in 3D finite-element problems) when using analyze, cho12, cholmod2, lcho1, and l1cho1. There are two methods for compiling CHOLMOD for use in MATLAB; both are described below.

6.1 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. It is also simpler for Windows users, who do not have the make command (unless you obtain a copy of Cygwin).

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. If you do not have METIS, type cholmod_make(''). If your copy of METIS is in another location, type cholmod_make ('path') where path is the pathname of your copy of the metis-4.0 directory.

When METIS is compiled malloc, free, calloc, and realloc are redefined to the MATLAB-equivalents (mxMalloc, ...). These memory-management functions safely terminate a mexFunction if they fail, and will free all memory allocated by the mexFunction. Thus, METIS will safely abort without terminating MATLAB, if it runs out of memory. The cholmod_make handles this redefinition without making any changes to your METIS source code.

6.2 Unix make for compiling CHOLMOD

You can also compile the CHOLMOD mexFunctions using the Unix/Linux make command. When using the gcc compiler, I strongly recommend editing the metis-4.0/Makefile.in file and changing OPTIONS to

```
OPTIONS = -fexceptions
```

Also ensure -fexceptions is in the CFLAGS option in the UFconfig/UFconfig.mk file that comes with CHOLMOD. If you do not make these modifications, the CHOLMOD mexFunctions will terminate MATLAB if they encounter an error.

If you have MATLAB 7.2 or earlier and use make mex in the CHOLMOD directory (equivalently, make in CHOLMOD/MATLAB), you must first edit UFconfig/UFconfig.h to remove the -largeArrayDims option from the MEX command (or just use cholmod_make.m inside MATLAB).

Next, compile your METIS 4.0.1 library by typing make in themetis-4.0 directory. Then type make in the CHOLMOD/MATLAB directory. This will compile the C-callable libraries for AMD, COLAMD, CAMD, CCOLAMD, METIS, and CHOLMOD, and then compile the mexFunction interfaces to those libraries. If METIS tries malloc and encounters an out-of-memory condition, it calls abort, which will terminate MATLAB. This problem does not occur using the method described in the previous section.
7 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 UFconfig.h. That file defines a C preprocessor token UF_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 UF_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

\[ C = \text{cholmod\_copy\_sparse} \left( A, &\text{Common} \right) ; \]

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:

\[ C = \text{cholmod\_l\_copy\_sparse} \left( A, &\text{Common} \right) ; \]

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 \( C = A \times B \), 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 $PAP^T$.

- $\text{tril}(A)$ is the lower triangular part of $A$, including the diagonal.
- $\text{tril}(A,k)$ is the lower triangular part of $A$, including entries on and below the $k$th diagonal.
- $\text{triu}(A)$ is the upper triangular part of $A$, including the diagonal.
- $\text{triu}(A,k)$ is the upper triangular part of $A$, including entries on and above the $k$th diagonal.
- $\text{size}(A)$ returns the dimensions of $A$.
- $\text{find}(x)$ 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 $A$.
- $\text{diag}(A)$ is the diagonal of $A$ if $A$ is a matrix.
- $C=\text{diag}(s)$ is a diagonal matrix if $s$ is a vector, with the values of $s$ on the diagonal of $C$.
- $S=\text{spones}(A)$ returns a binary matrix $S$ with the same nonzero pattern of $A$.
- $\text{nnz}(A)$ 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.
8 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. **Core**: 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

Two additional Modules are required to compile the CHOLMOD library:

1. **Include**: include files for CHOLMOD and programs that use CHOLMOD
2. **Lib**: where the CHOLMOD library is built

Five additional Modules provide support functions and documentation:

1. **Demo**: simple programs that illustrate the use of CHOLMOD
2. **Doc**: documentation (including this document)
3. **MATLAB**: CHOLMOD’s interface to MATLAB
4. **Tcov**: an exhaustive test coverage (requires Linux or Solaris)
5. **Valgrind**: runs the Tcov test under valgrind (requires Linux)

The following Modules are licensed under the GNU Lesser General Public License: Check, Cholesky, Core, and Partition. The following Modules are licensed under the GNU General Public License: Demo, Modify, MatrixOps, Supernodal, the MATLAB Module (not MATLAB itself!), Tcov, and Valgrind. The files in the Include Module are licensed according to their respective Modules. The Lib and Doc Modules need no license; the compiled binaries are licensed the same as their source code.
8.1 Core Module: basic data structures and definitions

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

8.1.1 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 $\sqrt{x^2+y^2}$ accurately.
- `cholmod_divcomplex`: complex divide.
8.1.2 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 \( n \times n \) 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 \( i[p[j] \ldots p[j+1]-1] \), and their corresponding numerical values are held in \( x[p[j] \ldots p[j+1]-1] \). 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_pttranspose: 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.
8.1.3 cholmod_factor: a symbolic or numeric factorization

A factor can be in $LL^T$ or $LDL^T$ 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 ($LDL^T$ to $LL^T$, 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.

8.1.4 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.
8.1.5 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.

8.1.6 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.
8.2 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 Core 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
The primary routines are all that a user requires to order, analyze, and factorize a sparse symmetric positive definite matrix $A$ (or $AA^T$), and to solve $Ax = b$ (or $AA^Tx = b$). The primary routines rely on the secondary routines, the Core 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 $x$ and $b$).
- **cholmod_spsolve**: solve a linear system (simplicial or supernodal, sparse $x$ and $b$).

Secondary routines:

- **cholmod_analyze_p**: analyze, with user-provided permutation or $f$ set.
- **cholmod_factorize_p**: factorize, with user-provided permutation or $f$.
- **cholmod_analyze_ordering**: analyze a permutation
- **cholmod_etree**: find the elimination tree.
- **cholmod_rowcolcounts**: compute the row/column counts of $L$.
- **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 $L$.
- **cholmod_row_lsubtree**: find the nonzero pattern of a row of $L$.
- **cholmod_resymbol**: recompute the symbolic pattern of $L$.
- **cholmod_resymbol_noperm**: recompute the symbolic pattern of $L$, no permutation.
- **cholmod_postorder**: postorder a tree.
- **cholmod_rcond**: compute the reciprocal condition number estimate.
- **cholmod_rowfac_mask**: for use in LPDASA only.
8.4 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 \( Lx = b \) 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 Core module can create an identity Cholesky factorization (\( LDL^T \) where \( L = D = I \)) that can then be modified by these routines. Requires the Core 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 \( Lx = b \)
- cholmod_updown_mark: update/downdate, and modify solution to partial \( Lx = b \)
- cholmod_updown_mask: for use in LPDASA only.
- cholmod_rowadd: add a row to an \( LDL^T \) factorization
- cholmod_rowadd_solve: add a row, and update solution to \( Lx = b \)
- cholmod_rowadd_mark: add a row, and update solution to partial \( Lx = b \)
- cholmod_rowdel: delete a row from an \( LDL^T \) factorization
- cholmod_rowdel_solve: delete a row, and downdate \( Lx = b \)
- cholmod_rowdel_mark: delete a row, and downdate solution to partial \( Lx = b \)

8.5 MatrixOps Module: basic sparse matrix operations

The MatrixOps Module provides basic operations on sparse and dense matrices. Requires the Core 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: \( s = \text{norm}(X) \), 1-norm, infinity-norm, or 2-norm
- cholmod_norm_sparse: \( s = \text{norm}(A) \), 1-norm or infinity-norm
- cholmod_horzcat: \( C = [A,B] \)
- cholmod_scale: \( A = \text{diag}(s)*A, A*\text{diag}(s), s*A \) or \( \text{diag}(s)*A*\text{diag}(s) \).
- 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.
8.6 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 Core 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 $Lx = b$ solve
- `cholmod_super_ltsolve`: supernodal $L^T x = b$ solve

8.7 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 Core and Cholesky Modules, and two packages: METIS 4.0.1, 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`.
- `cholmodCollapseSepTree`: pruned a separator tree from `cholmod_nested_dissection`.
9 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 \( \geq 0 \) if successful, or -1 otherwise.
- **double**: a value \( \geq 0 \) 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).
10 Core Module: cholmod_common object

10.1 Constant definitions

/* itype defines the types of integer used: */
#define CHOLMOD_INT 0 /* all integer arrays are int */
#define CHOLMOD_INTLONG 1 /* most are int, some are UF_long */
#define CHOLMOD_LONG 2 /* all integer arrays are UF_long */

/* The itype of all parameters for all CHOLMOD routines must match.
 * FUTURE WORK: CHOLMOD_INTLONG is not yet supported.
 */

/* dtype defines what the numerical type is (double or float): */
#define CHOLMOD_DOUBLE 0 /* all numerical values are double */
#define CHOLMOD_SINGLE 1 /* all numerical values are float */

/* The dtype of all parameters for all CHOLMOD routines must match.
 * Scalar floating-point values are always passed as double arrays of size 2
 * (for the real and imaginary parts). They are typecast to float as needed.
 * FUTURE WORK: the float case is not supported yet.
 */

/* xtype defines the kind of numerical values used: */
#define CHOLMOD_PATTERN 0 /* pattern only, no numerical values */
#define CHOLMOD_REAL 1 /* a real matrix */
#define CHOLMOD_COMPLEX 2 /* a complex matrix (ANSI C99 compatible) */
#define CHOLMOD_ZOMPLEX 3 /* a complex matrix (MATLAB compatible) */

/* The xtype of all parameters for all CHOLMOD routines must match.
 * CHOLMOD_PATTERN: x and z are ignored.
 * CHOLMOD_DOUBLE: x is non-null of size nzmax, z is ignored.
 * CHOLMOD_COMPLEX: x is non-null of size 2*nzmax doubles, z is ignored.
 * CHOLMOD_ZOMPLEX: x and z are non-null of size nzmax
 * In the real case, z is ignored. The kth entry in the matrix is x [k].
 * There are two methods for the complex case. In the ANSI C99-compatible
 * CHOLMOD_COMPLEX case, the real and imaginary parts of the kth entry
 * are in x [2*k] and x [2*k+1], respectively. z is ignored. In the
 * MATLAB-compatible CHOLMOD_ZOMPLEX case, the real and imaginary
 * parts of the kth entry are in x [k] and z [k].
 * Scalar floating-point values are always passed as double arrays of size 2
 * (real and imaginary parts). The imaginary part of a scalar is ignored if
 * the routine operates on a real matrix.
 * These Modules support complex and zomplex matrices, with a few exceptions:
 * Check all routines
 * Cholesky all routines
 * Core all except cholmod_at, add, band, copy
 * Demo all routines
 * Partition all routines
 * Supernodal all routines support any real, complex, or zomplex input.
 * There will never be a supernodal zomplex L; a complex
 * supernodal L is created if A is zomplex. */
These Modules provide partial support for complex and zomplex matrices:

MATLAB all routines support real and zomplex only, not complex, with the exception of ldlupdate, which supports real matrices only. This is a minor constraint since MATLAB’s matrices are all real or zomplex.

MatrixOps only norm_dense, norm_sparse, and sdmult support complex and zomplex

These Modules do not support complex and zomplex matrices at all:

Modify all routines support real matrices only

Purpose: These definitions are used within the cholmod_common object, called Common both here and throughout the code.
typedef struct cholmod_common_struct
{
    /* ------------------------------------------------------------- */
    /* parameters for symbolic/numeric factorization and update/downdate */
    /* ------------------------------------------------------------ */

double dbound ; /* Smallest absolute value of diagonal entries of D
    * for LDL' factorization and update/downdate/rowadd/
    * rowdel, or the diagonal of L for an LL' factorization.
    * Entries in the range 0 to dbound are replaced with dbound.
    * Entries in the range -dbound to 0 are replaced with -dbound. No
    * changes are made to the diagonal if dbound <= 0. Default: zero */

double grow0 ; /* For a simplicial factorization, L->i and L->x can
    * grow if necessary. grow0 is the factor by which
    * it grows. For the initial space, L is of size MAX (1,grow0) times
    * the required space. If L runs out of space, the new size of L is
    * MAX(1.2,grow0) times the new required space. If you do not plan on
    * modifying the LDL' factorization in the Modify module, set grow0 to
    * zero (or set grow2 to 0, see below). Default: 1.2 */

double grow1 ;

double grow2 ; /* For a simplicial factorization, each column j of L
    * is initialized with space equal to
    * grow1*L->ColCount[j] + grow2. If grow0 < 1, grow1 < 1, or grow2 == 0,
    * then the space allocated is exactly equal to L->ColCount[j]. If the
    * column j runs out of space, it increases to grow1*need + grow2 in
    * size, where need is the total # of nonzeros in that column. If you do
    * not plan on modifying the factorization in the Modify module, set
    * grow2 to zero. Default: grow1 = 1.2, grow2 = 5. */

size_t maxrank ; /* rank of maximum update/downdate. Valid values:
    * 2, 4, or 8. A value < 2 is set to 2, and a
    * value > 8 is set to 8. It is then rounded up to the next highest
    * power of 2. If not already a power of 2. Workspace (Xwork, below) of
    * size nrow-by-maxrank double's is allocated for the update/downdate.
    * If an update/downdate of rank-k is requested, with k > maxrank,
    * it is done in steps of maxrank. Default: 8, which is fastest.
    * Memory usage can be reduced by setting maxrank to 2 or 4.
    */

size_t supernodal_switch ; /* supernodal vs simplicial factorization */
int supernodal ; /* If Common->supernodal <= CHOLMOD_SIMPLICIAL
    * (0) then cholmod_analyze performs a
    * simplicial analysis. If >= CHOLMOD_SUPERNODAL (2), then a supernodal
    * analysis is performed. If == CHOLMOD_AUTO (1) and
    * flop/nnz(L) < Common->supernodal_switch, then a simplicial analysis
    * is done. A supernodal analysis done otherwise.
    * Default: CHOLMOD_AUTO. Default supernodal_switch = 40 */

int final_asis ; /* If TRUE, then ignore the other final_* parameters
    * (except for final_pack).
    * The factor is left as-is when done. Default: TRUE. */

int final_super ; /* If TRUE, leave a factor in supernodal form when
    * supernodal factorization is finished. If FALSE,
* then convert to a simplicial factor when done.
  * Default: TRUE */

int final_ll ; /* If TRUE, leave factor in LL' form when done.
  * Otherwise, leave in LDL' form. Default: FALSE */

int final_pack ; /* If TRUE, pack the columns when done. If TRUE, and
  * cholmod_factorize is called with a symbolic L, L is
  * allocated with exactly the space required, using L->ColCount. If you
  * plan on modifying the factorization, set Common->final_pack to FALSE,
  * and each column will be given a little extra slack space for future
  * growth in fill-in due to updates. Default: TRUE */

int final_monotonic ; /* If TRUE, ensure columns are monotonic when done.
  * Default: TRUE */

int final_resymbol ;/* if cholmod_factorize performed a supernodal
  * factorization, final_resymbol is true, and
  * final_super is FALSE (convert a simplicial numeric factorization),
  * then numerically zero entries that resulted from relaxed supernodal
  * amalgamation are removed. This does not remove entries that are zero
  * due to exact numeric cancellation, since doing so would break the
  * update/downdate rowadd/rowdel routines. Default: FALSE. */

/* supernodal relaxed amalgamation parameters: */
double zrelax [3] ;
size_t nrelax [3] ;

/* Let ns be the total number of columns in two adjacent supernodes.
 * Let z be the fraction of zero entries in the two supernodes if they
 * are merged (z includes zero entries from prior amalgations). The
 * two supernodes are merged if:
 * (ns <= nrelax [0]) || (no new zero entries added) ||
 * (ns <= nrelax [1] && z < zrelax [0]) ||
 * (ns <= nrelax [2] && z < zrelax [1]) || (z < zrelax [2])
 * *
 * Default parameters result in the following rule:
 * (ns <= 4) || (no new zero entries added) ||
 * (ns <= 16 && z < 0.8) || (ns <= 48 && z < 0.1) || (z < 0.05)
 */

int prefer_zomplex ; /* X = cholmod_solve (sys, L, B, Common) computes
  * x=A\b or solves a related system. If L and B are
  * both real, then X is real. Otherwise, X is returned as
  * CHOLMOD_COMPLEX if Common->prefer_zomplex is FALSE, or
  * CHOLMOD_ZOMPLEX if Common->prefer_zomplex is TRUE. This parameter
  * is needed because there is no supernodal zomplex L. Suppose the
  * caller wants all complex matrices to be stored in zomplex form
  * (MATLAB, for example). A supernodal L is returned in complex form
  * if A is zomplex. B can be real, and thus X = cholmod_solve (L,B)
  * should return X as zomplex. This cannot be inferred from the input
  * arguments L and B. Default: FALSE, since all data types are
  * supported in CHOLMOD_COMPLEX form and since this is the native type
  * of LAPACK and the BLAS. Note that the MATLAB/cholmod.c mexFunction
  * sets this parameter to TRUE, since MATLAB matrices are in
  * CHOLMOD_ZOMPLEX form. */

int prefer_upper ; /* cholmod_analyze and cholmod_factorize work
* fastest when a symmetric matrix is stored in
* upper triangular form when a fill-reducing ordering is used. In
* MATLAB, this corresponds to how x=A\b works. When the matrix is
* ordered as-is, they work fastest when a symmetric matrix is in lower
* triangular form. In MATLAB, R=chol(A) does the opposite. This
* parameter affects only how cholmod_read returns a symmetric matrix.
* If TRUE (the default case), a symmetric matrix is always returned in
* upper-triangular form (A->stype = 1). */

int quick_return_if_not_posdef ; /* if TRUE, the supernodal numeric
* the matrix is not positive definite. Default: FALSE. */

/* printing and error handling options */
/* ---------------------------------------------------------------------- */
/* ordering options */
/* ---------------------------------------------------------------------- */

/* The cholmod_analyze routine can try many different orderings and select
* the best one. It can also try one ordering method multiple times, with
* different parameter settings. The default is to use three orderings,
* the user's permutation (if provided), AMD which is the fastest ordering
* and generally gives good fill-in, and METIS. CHOLMOD's nested dissection
* (METIS with a constrained AMD) usually gives a better ordering than METIS
* alone (by about 5% to 10%) but it takes more time.
*
* If you know the method that is best for your matrix, set Common->nmethods
* to 1 and set Common->method [0] to the set of parameters for that method.
* If you set it to 1 and do not provide a permutation, then only AMD will
* be called.
*
* If METIS is not available, the default # of methods tried is 2 (the user
* permutation, if any, and AMD).
*
* To try other methods, set Common->nmethods to the number of methods you
* want to try. The suite of default methods and their parameters is
* described in the cholmod_defaults routine, and summarized here:
* Common->method[i]:
  * i = 0: user-provided ordering (cholmod_analyze_p only)
  * i = 1: AMD (for both A and A*A')
  * i = 2: METIS
  * i = 3: CHOLMOD’s nested dissection (NESDIS), default parameters
  * i = 4: natural
  * i = 5: NESDIS with nd_small = 20000
  * i = 6: NESDIS with nd_small = 4, no constrained minimum degree
  * i = 7: NESDIS with no dense node removal
  * i = 8: AMD for A, COLAMD for A*A'

* You can modify the suite of methods you wish to try by modifying
  * Common.method[...]] after calling cholmod_start or cholmod_defaults.

* For example, to use AMD, followed by a weighted postordering:
  *
  * Common->nmethods = 1;
  * Common->method[0].ordering = CHOLMOD_AMD;
  * Common->postorder = TRUE;

* To use the natural ordering (with no postordering):
  *
  * Common->nmethods = 1;
  * Common->method[0].ordering = CHOLMOD_NATURAL;
  * Common->postorder = FALSE;

* If you are going to factorize hundreds or more matrices with the same
  * nonzero pattern, you may wish to spend a great deal of time finding a
  * good permutation. In this case, try setting Common->nmethods to 9.
  * The time spent in cholmod_analyze will be very high, but you need to
  * call it only once.
  * cholmod_analyze sets Common->current to a value between 0 and nmethods-1.
  * Each ordering method uses the set of options defined by this parameter.

int nmethods;        /* The number of ordering methods to try. Default: 0.
  * nmethods = 0 is a special case. cholmod_analyze
  * will try the user-provided ordering (if given) and AMD. Let fl and
  * lnz be the flop count and nonzeros in L from AMD’s ordering. Let
  * anz be the number of nonzeros in the upper or lower triangular part
  * of the symmetric matrix A. If fl/lnz < 500 or lnz/anz < 5, then this
  * is a good ordering, and METIS is not attempted. Otherwise, METIS is
  * tried. The best ordering found is used. If nmethods > 0, the
  * methods used are given in the method[ ] array, below. The first
  * three methods in the default suite of orderings is (1) use the given
  * permutation (if provided), (2) use AMD, and (3) use METIS. Maximum
  * allowed value is CHOLMOD_MAXMETHODS. */

int current;        /* The current method being tried. Default: 0. Valid
  * range is 0 to nmethods-1. */

int selected;       /* The best method found. */

/* The suite of ordering methods and parameters: */

struct cholmod_method_struct
{
  /* statistics for this method */

51
double lnnz; /* nnz(L) excl. zeros from supernodal amalgamation, * for a "pure" L */

double fl; /* flop count for a "pure", real simplicial LL' * factorization, with no extra work due to * amalgamation. Subtract n to get the LDL' flop count. Multiply * by about 4 if the matrix is complex or zomplex. */

/* ordering method parameters */
double prune_dense; /* dense row/col control for AMD, SYMAMD, CSYMAMD, * and NESDIS (cholmod_nested_dissection). For a * symmetric n-by-n matrix, rows/columns with more than * MAX (16, prune_dense * sqrt (n)) entries are removed prior to * ordering. They appear at the end of the re-ordered matrix. * * If prune_dense < 0, only completely dense rows/cols are removed. * * This parameter is also the dense column control for COLAMD and * CCOLAMD. For an m-by-n matrix, columns with more than * MAX (16, prune_dense * sqrt (MIN (m,n))) entries are removed prior * to ordering. They appear at the end of the re-ordered matrix. * CHOLMOD factorizes A*A', so it calls COLAMD and CCOLAMD with A', * not A. Thus, this parameter affects the dense *row* control for * CHOLMOD's matrix, and the dense *column* control for COLAMD and * CCOLAMD. * * Removing dense rows and columns improves the run-time of the * ordering methods. It has some impact on ordering quality * (usually minimal, sometimes good, sometimes bad). * * Default: 10. */

double prune_dense2; /* dense row control for COLAMD and CCOLAMD. * Rows with more than MAX (16, dense2 * sqrt (n)) * for an m-by-n matrix are removed prior to ordering. CHOLMOD's * matrix is transposed before ordering it with COLAMD or CCOLAMD, * so this controls the dense *columns* of CHOLMOD's matrix, and * the dense *rows* of COLAMD's or CCOLAMD's matrix. * * If prune_dense2 < 0, only completely dense rows/cols are removed. * * Default: -1. Note that this is not the default for COLAMD and * CCOLAMD. -1 is best for Cholesky. 10 is best for LU. */

double nd_oksep; /* in NESDIS, when a node separator is computed, it * discarded if nsep >= nd_oksep*n, where nsep is * the number of nodes in the separator, and n is the size of the * graph being cut. Valid range is 0 to 1. If 1 or greater, the * separator is discarded if it consists of the entire graph. * Default: 1 */

double other1 [4]; /* future expansion */

size_t nd_small; /* do not partition graphs with fewer nodes than * nd_small, in NESDIS. Default: 200 (same as * METIS) */

size_t other2 [4]; /* future expansion */
int aggressive ; /* Aggresive absorption in AMD, COAMD, SYMAMD, * COCOLAMD, and CSYMAMD. Default: TRUE */

int order_for_lu ; /* COCOLAMD can be optimized to produce an ordering * for LU or Cholesky factorization. CHOLMOD only * performs a Cholesky factorization. However, you may wish to use * CHOLMOD as an interface for COCOLAMD but use it for your own LU * factorization. In this case, order_for_lu should be set to FALSE. * When factorizing in CHOLMOD itself, you should *** NEVER *** set * this parameter FALSE. Default: TRUE */

int nd_compress ; /* If TRUE, compress the graph and subgraphs before * partitioning them in NESDIS. Default: TRUE */

int nd_camd ; /* If 1, follow the nested dissection ordering * with a constrained minimum degree ordering that * respects the partitioning just found (using CAMD). If 2, use * CSYMAMD instead. If you set nd_small very small, you may not need * this ordering, and can save time by setting it to zero (no * constrained minimum degree ordering). Default: 1. */

int nd_components ; /* The nested dissection ordering finds a node * separator that splits the graph into two parts, * which may be unconnected. If nd_components is TRUE, each of * these connected components is split independently. If FALSE, * each part is split as a whole, even if it consists of more than * one connected component. Default: FALSE */

/* fill-reducing ordering to use */
int ordering;

size_t other3 [4] ; /* future expansion */

} method [CHOLMOD_MAXMETHODS + 1] ;

int postorder ; /* If TRUE, cholmod_analyze follows the ordering with a * weighted postorder of the elimination tree. Improves * supernode amalgamation. Does not affect fundamental nnz(L) and * flop count. Default: TRUE. */

/* memory management routines */
/* ---------------------------------------------------------------------- */
void *(*malloc_memory) (size_t) ; /* pointer to malloc */
void *(*realloc_memory) (void *, size_t) ; /* pointer to realloc */
void (*free_memory) (void *) ; /* pointer to free */
void *(*calloc_memory) (size_t, size_t) ; /* pointer to calloc */

/* routines for complex arithmetic */
/* ---------------------------------------------------------------------- */
int (*complex_divide) (double ax, double az, double bx, double bz, double *cx, double *cz);

/* flag = complex_divide (ax, az, bx, bz, &cx, &cz) computes the complex * division c = a/b, where ax and az hold the real and imaginary part * of a, and b and c are stored similarly. flag is returned as 1 if */

53
* a divide-by-zero occurs, or 0 otherwise. By default, the function
* pointer Common->complex_divide is set equal to cholmod_divcomplex.
*/
double (*hypotenuse) (double x, double y);

/* s = hypotenuse (x,y) computes s = sqrt (x*x + y*y), but does so more
* accurately. By default, the function pointer Common->hypotenuse is
* set equal to cholmod_hypot. See also the hypot function in the C99
* standard, which has an identical syntax and function. If you have
* a C99-compliant compiler, you can set Common->hypotenuse = hypot. */

double metis_memory; /* This is a parameter for CHOLMOD’s interface to
  * METIS, not a parameter to METIS itself. METIS
  * uses an amount of memory that is difficult to estimate precisely
  * beforehand. If it runs out of memory, it terminates your program.
  * All routines in CHOLMOD except for CHOLMOD’s interface to METIS
  * return an error status and safely return to your program if they run
  * out of memory. To mitigate this problem, the CHOLMOD interface
  * can allocate a single block of memory equal in size to an empirical
  * upper bound of METIS’s memory usage times the Common->metis_memory
  * parameter, and then immediately free it. It then calls METIS. If
  * this pre-allocation fails, it is possible that METIS will fail as
  * well, and so CHOLMOD returns with an out-of-memory condition without
  * calling METIS.
  *
  * METIS_NodeND (used in the CHOLMOD_METIS ordering option) with its
  * default parameter settings typically uses about (4*nz+40n+4096)
  * times sizeof(int) memory, where nz is equal to the number of entries
  * in A for the symmetric case or AA' if an unsymmetric matrix is
  * being ordered (where nz includes both the upper and lower parts
  * of A or AA'). The observed "upper bound" (with 2 exceptions),
  * measured in an instrumented copy of METIS 4.0.1 on thousands of
  * matrices, is (10*nz+50*n+4096) * sizeof(int). Two large matrices
  * exceeded this bound, one by almost a factor of 2 (Gupta/gupta2).
  *
  * If your program is terminated by METIS, try setting metis_memory to
  * 2.0, or even higher if needed. By default, CHOLMOD assumes that METIS
  * does not have this problem (so that CHOLMOD will work correctly when
  * this issue is fixed in METIS). Thus, the default value is zero.
  * This work-around is not guaranteed anyway.
  *
  * If a matrix exceeds this predicted memory usage, AMD is attempted
  * instead. It, too, may run out of memory, but if it does so it will
  * not terminate your program.
*/

double metis_dswitch; /* METIS_NodeND in METIS 4.0.1 gives a seg */
size_t metis_nswitch; /* fault with one matrix of order n = 3005 and
  * nz = 6,036,025. This is a very dense graph.
  * The workaround is to use AMD instead of METIS for matrices of dimension
  * greater than Common->metis_nswitch (default 3000) or more and with
  * density of Common->metis_dswitch (default 0.66) or more.
  * cholmod_nested_dissection has no problems with the same matrix, even
  * though it uses METIS_NodeComputeSeparator on this matrix. If this
* seg fault does not affect you, set metis_nswitch to zero or less,
* and CHOLMOD will not switch to AMD based just on the density of the
* matrix (it will still switch to AMD if the metis_memory parameter
* causes the switch).
*/

/* ---------------------------------------------------------------------- */
/* workspace */
/* ---------------------------------------------------------------------- */

/* CHOLMOD has several routines that take less time than the size of
* workspace they require. Allocating and initializing the workspace would
* dominate the run time, unless workspace is allocated and initialized
* just once. CHOLMOD allocates this space when needed, and holds it here
* between calls to CHOLMOD. cholmod_start sets these pointers to NULL
* (which is why it must be the first routine called in CHOLMOD).
* cholmod_finish frees the workspace (which is why it must be the last
* call to CHOLMOD).
*/

size_t nrow ; /* size of Flag and Head */
UF_long mark ; /* mark value for Flag array */
size_t ivworksize ; /* size of Ivwork. Upper bound: 6*nrow+ncol */
size_t xworksize ; /* size of Xwork, in bytes.
  * maxrank*nrow*sizeof(double) for update/downdate.
  * 2*nrow*sizeof(double) otherwise */

/* initialized workspace: contents needed between calls to CHOLMOD */
void *Flag ; /* size nrow, an integer array. Kept cleared between
  * calls to cholmod routines (Flag [i] < mark) */
void *Head ; /* size nrow+1, an integer array. Kept cleared between
  * calls to cholmod routines (Head [i] = EMPTY) */
void *Xwork ; /* a double array. Its size varies. It is nrow for
  * most routines (cholmod_rowfac, cholmod_add,
  * cholmod_aat, cholmod_norm, cholmod_ssmult) for the real case, twice
  * that when the input matrices are complex or complex. It is of size
  * 2*nrow for cholmod_rowadd and cholmod_routine. For cholmod_updown,
  * its size is maxrank*nrow where maxrank is 2, 4, or 8. Kept cleared
  * between calls to cholmod (set to zero). */

/* uninitialized workspace, contents not needed between calls to CHOLMOD */
void *Iwork ; /* size ivworksize, 2*nrow+ncol for most routines,
  * up to 6*nrow+ncol for cholmod_analyze. */

int itype ; /* If CHOLMOD_LONG, Flag, Head, and Iwork are UF_long.
  * Otherwise all three arrays are int. */
int dtype ; /* double or float */

/* Common->itype and Common->dtype are used to define the types of all
 * sparse matrices, triplet matrices, dense matrices, and factors
 * created using this Common struct. The itypes and dtypes of all
 * parameters to all CHOLMOD routines must match. */

int no_workspace_reallocate ; /* this is an internal flag, used as a
  * precaution by cholmod_analyze. It is normally false. If true,
  * cholmod_allocate_work is not allowed to reallocate any workspace;
they must use the existing workspace in Common (Iwork, Flag, Head, and Xwork). Added for CHOLMOD v1.1 */

/* statistics */

/* fl and lnz are set only in cholmod_analyze and cholmod_rowcolcounts, */
/* in the Cholesky module. modfl is set only in the Modify module. */

int status ; /* error code */
double fl ; /* LL' flop count from most recent analysis */
double lnz ; /* fundamental nz in L */
double anz ; /* nonzeros in tril(A) if A is symmetric/lower, */
/* triu(A) if symmetric/upper, or tril(A*A') if */
/* unsymmetric, in last call to cholmod_analyze. */
double modfl ; /* flop count from most recent update/downdate/ */
/* rovadd/rowdel (excluding flops to modify the */
/* solution to Lx=b, if computed) */
size_t malloc_count ; /* # of objects malloc'ed minus the # free'd*/
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 ; /* # of flops in last call to cholmod_rowfac */
double aatfl ; /* # of flops to compute A(:,f)*A(:,f)' */

/* future expansion */

/* To allow CHOLMOD to be updated without recompiling the user application, */
/* additional space is set aside here for future statistics, parameters, */
/* and workspace. Note: additional entries were added in v1.1 to the */
/* method array, above, and thus v1.0 and v1.1 are not binary compatible. */
/* * v1.1 to the current version are binary compatible. */

/* --------------------------------------------------------------- */
double other1 [10] ;
double SPQR_xstat [4] ; /* for SuiteSparseQR statistics */

/* SuiteSparseQR control parameters: */
double SPQR_grain ; /* task size is >= max (total flops / grain) */
double SPQR_small ; /* task size is >= small */

/* --------------------------------------------------------------- */
UF_long SPQR_istat [10] ; /* for SuiteSparseQR statistics */
UF_long other2 [6] ; /* reduced from size 16 in v1.6 */

/* --------------------------------------------------------------- */
int other3 [10] ; /* reduced from size 16 in v1.1 */
int prefer_binary ; /* cholmod_read_triplet converts a symmetric
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.
10.3 cholmod_start: start CHOLMOD

```c
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.

10.4 cholmod_finish: finish CHOLMOD

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

int cholmod_l_finish (cholmod_common *);
```

**Purpose:** This must be the last call to CHOLMOD.

10.5 cholmod_defaults: set default parameters

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

int cholmod_l_defaults (cholmod_common *);
```

**Purpose:** Sets the default parameters.

10.6 cholmod_maxrank: maximum update/downdate rank

```c
size_t cholmod_maxrank /* returns validated value of Common->maxrank */
(
    /* ---- input ---- */
    size_t n, /* A and L will have n rows */
    /* --------------- */
    cholmod_common *Common
);

size_t cholmod_l_maxrank (size_t, cholmod_common *);
```

**Purpose:** Returns the maximum rank for an update/downdate.
10.7 cholmod_allocate_work: allocate workspace

```c
int cholmod_allocate_work
(
    size_t nrow,  /* size: Common->Flag (nrow), Common->Head (nrow+1) */
    size_t iworksize,  /* size of Common->Iwork */
    size_t xworksize,  /* size of Common->Xwork */
    cholmod_common *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. 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.

10.8 cholmod_free_work: free workspace

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

**Purpose:** Frees the workspace in Common.

10.9 cholmod_clear_flag: clear Flag array

```c
UF_long cholmod_clear_flag
(
    cholmod_common *Common
);
```

**Purpose:** Increments Common->mark so that the Flag array is now cleared.
10.10 cholmod_error: report error

```c
int cholmod_error
(
/* ---- input ---- */
int status,     /* error status */
const char *file, /* name of source code file where error occurred */
int line,       /* line number in source code file where error occurred */
const char *message,/* error message */
/* --------------- */
cholmod_common *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.

10.11 cholmod_dbound: bound diagonal of L

```c
double cholmod_dbound  /* returns modified diagonal entry of D or L */
(
/* ---- input ---- */
double dj,       /* diagonal entry of D for LDL' or L for LL' */
/* --------------- */
cholmod_common *Common
);
```

Purpose: Ensures that entries on the diagonal of L for an LLᵀ factorization are greater than or equal to `Common->dbound`. For an LDLᵀ factorization, it ensures that the magnitude of the entries of D are greater than or equal to `Common->dbound`.

10.12 cholmod_hypot: sqrt(x*x+y*y)

```c
double cholmod_hypot
(
/* ---- input ---- */
double x, double y
);
```

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`.  

60
10.13 cholmod_divcomplex: complex divide

```c
int cholmod_divcomplex /* return 1 if divide-by-zero, 0 otherwise */
    ( /* ---- input ---- */
        double ar, double ai, /* real and imaginary parts of a */
        double br, double bi, /* real and imaginary parts of b */
    /* ---- output --- */
        double *cr, double *ci /* real and imaginary parts of c */
    );
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.
11 Core Module: cholmod_sparse object

11.1 cholmod_sparse: compressed-column sparse matrix

```c
typedef struct cholmod_sparse_struct {
    size_t nrow ; /* the matrix is nrow-by-ncol */
    size_t ncol ;
    size_t nzmax ; /* maximum number of entries in the matrix */
    /* pointers to int or UF_long: */
    void *p ; /* p [0..ncol], the column pointers */
    void *i ; /* i [0..nzmax-1], the row indices */
    /* for unpacked matrices only: */
    void *nz ; /* nz [0..ncol-1], the # of nonzeros in each col. In *
        * packed form, the nonzero pattern of column j is in *
        * A->i [A->p [j] ... A->p [j+1]-1]. In unpacked form, column j in *
        * A->i [A->p [j] ... A->p [j]+A->nz[j]-1] instead. In both cases, the *
        * numerical values (if present) are in the corresponding locations in *
        * the array x (or z if A->xtype is CHOLMOD_ZOMPLEX). */
    void *x ; /* size nzmax or 2*nzmax, if present */
    void *z ; /* size nzmax, if present */
    int stype ; /* Describes what parts of the matrix are considered: *
        * 0: matrix is "unsymmetric": use both upper and lower triangular parts *
        * (the matrix may actually be symmetric in pattern and value, but *
        * both parts are explicitly stored and used). May be square or *
        * rectangular. *
        * >0: matrix is square and symmetric, use upper triangular part. *
        * Entries in the lower triangular part are ignored. *
        * <0: matrix is square and symmetric, use lower triangular part. *
        * Entries in the upper triangular part are ignored. *
        * Note that stype>0 and stype<0 are different for cholmod_sparse and *
        * cholmod_triplet. See the cholmod_triplet data structure for more *
        * details. */
    int itype ; /* CHOLMOD_INT: p, i, and nz are int. *
        * CHOLMOD_INTLONG: p is UF_long, i and nz are int. *
        * CHOLMOD_LONG: p, i, and nz are UF_long. */
    int xtype ; /* pattern, real, complex, or zomplex */
    int dtype ; /* x and z are double or float */
    int sorted ; /* TRUE if columns are sorted, FALSE otherwise */
    int packed ; /* TRUE if packed (nz ignored), FALSE if unpacked *
        * (nz is required) */
} cholmod_sparse ;
```

**Purpose:** Stores a sparse matrix in compressed-column form.
11.2 cholmod_allocate_sparse: allocate sparse matrix

**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.

```
11.3 cholmod_free_sparse: free sparse matrix

int cholmod_free_sparse
(
    /* ---- in/out --- */
    cholmod_sparse **A, /* matrix to deallocate, NULL on output */
    /* --------------- */
    cholmod_common *Common
);

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

**Purpose:** Frees a sparse matrix.

11.4 cholmod_reallocate_sparse: reallocate sparse matrix

```
11.4 cholmod_reallocate_sparse: reallocate sparse matrix

int cholmod_reallocate_sparse
(
    /* ---- input ---- */
    size_t nznew, /* new # of entries in A */
    /* ---- in/out --- */
    cholmod_sparse *A, /* 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.
11.5 cholmod_nnz: number of entries in sparse matrix

```
UF_long cholmod_nnz
(
    /* ---- input ---- */
    cholmod_sparse *A,
    /* --------------- */
    cholmod_common *Common
);
```

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

11.6 cholmod_speye: sparse identity matrix

**Purpose:** Returns the sparse identity matrix.

11.7 cholmod_spzeros: sparse zero matrix

**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).
11.8 cholmod_transpose: transpose sparse matrix

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

11.9 cholmod_ptranspose: transpose/permute sparse matrix

**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.

11.10 cholmod_sort: sort columns of a sparse matrix

```c
int cholmod_sort
(
    /* ---- in/out --- */
    cholmod_sparse *A, /* 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.
### 11.11 cholmod_transpose_unsym: transpose/permute unsymmetric sparse matrix

```c
int cholmod_transpose_unsym
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to transpose */
    int values, /* 0: pattern, 1: array transpose, 2: conj. transpose */
    int *Perm, /* size nrow, if present (can be NULL) */
    int *fset, /* subset of 0:(A->ncol)-1 */
    size_t fsize, /* size of fset */
    /* ---- output --- */
    cholmod_sparse *F, /* F = A', A(:,f)', or A(p,f)' */
    /* --------------- */
    cholmod_common *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:

```plaintext
[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 \( F = A(\mathbf{p}, f) \) operation, use \texttt{cholmod_submatrix} instead (which does not compute the transpose). \( F \rightarrow \text{nzmax} \) must be large enough to hold the matrix \( F \). If \( F \rightarrow \text{nz} \) is present then \( F \rightarrow \text{nz} [j] \) 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 \( F \rightarrow \text{packed} \) on input. If \( F \rightarrow \text{packed} \) is \texttt{FALSE}, then \( F \) is returned in unpacked form (\( F \rightarrow \text{nz} \) 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, \( F \rightarrow i[1] + F \rightarrow \text{nz}[1] - 1 \) contain all entries in \( A(1,f) \), but \( F \rightarrow p[i+1] - F \rightarrow p[i] \) is equal to the number of nonzeros in \( A (i,:) \), not just \( A (i,f) \). The \texttt{cholmod_transpose_unsym} routine is the only operation in \texttt{CHOLMOD} that can produce an unpacked matrix.

11.12 cholmod_transpose_sym: transpose/permute symmetric sparse matrix

```c
int cholmod_transpose_sym
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to transpose */
    int values, /* 0: pattern, 1: array transpose, 2: conj. transpose */
    int *Perm, /* size nrow, if present (can be NULL) */
    /* ---- output --- */
    cholmod_sparse *F, /* F = A' or A(p,p)' */
    /* ---------------- */
    cholmod_common *Common
)
;

int cholmod_l_transpose_sym (cholmod_sparse *, int, UF_long *, cholmod_sparse *,
    cholmod_common *)

Purpose: Computes \( F = A' \) or \( F = A(p,p)' \), the transpose or permuted transpose, where \( A \rightarrow \text{stype} \) is nonzero. \( A \) must be square and symmetric. If \( A \rightarrow \text{stype} > 0 \), 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 \( A \rightarrow \text{stype} < 0 \), 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 \texttt{values} parameter:

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

For \texttt{cholmod_transpose_unsym} and \texttt{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 \( F \rightarrow p \) contain the column pointers of the resulting matrix, where \( F \rightarrow p [F \rightarrow \text{ncol}] > F \rightarrow \text{nzmax} \). In this case, the remaining contents of \( F \) are not modified. \( F \) can still be properly freed with \texttt{cholmod_free_spare}.
11.13 cholmod_band: extract band of a sparse matrix

Purpose: Returns \( C = \text{tril} (\text{triu} (A, k_1), k_2) \). \( C \) is a matrix consisting of the diagonals of \( A \) from \( k_1 \) to \( k_2 \). \( k=0 \) 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:

- \( k_1=-m \) means \( C = \text{tril} (A, k_2) \)
- \( k_2=n \) means \( C = \text{triu} (A, k_1) \)
- \( k_1=0 \) and \( k_2=0 \) means \( C = \text{diag}(A) \), except \( C \) is a matrix, not a vector

Values of \( k_1 \) and \( k_2 \) 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 \( A->\text{stype} > 0 \), entries in the lower triangular part of \( A \) are ignored, and the opposite is true if \( A->\text{stype} < 0 \). 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 \( \text{mode} > 0 \)), as a pattern-only (\( \text{mode} = 0 \)), or as a pattern-only but with the diagonal entries removed (\( \text{mode} < 0 \)).

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

11.14 cholmod_band_inplace: extract band, in place

```c
int cholmod_band_inplace
(  /* ---- input ---- */  UF_long k1, /* ignore entries below the k1-st diagonal */  UF_long k2, /* ignore entries above the k2-nd diagonal */  int mode,  /* >0: numerical, 0: pattern, <0: pattern (no diag) */  /* ---- in/out --- */  cholmod_sparse *A, /* matrix from which entries not in band are removed */  /* --------------- */  cholmod_common *Common  ) ;
```

Purpose: Same as cholmod_band, except that it always operates in place. Only packed matrices can be converted in place.
11.15  cholmod_aat: compute $AA^T$

Purpose: Computes $C = A\cdot A^T$ or $C = A(:,f)\cdot A(:,f)^T$. $A$ 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 $\leq 0$ (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.

11.16  cholmod_copy_sparse: copy sparse matrix

Purpose: Returns an exact copy of the input sparse matrix $A$. 
11.17 cholmod_copy: copy (and change) sparse matrix

**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 $A \rightarrow \text{stype}$:

- $< 0$, lower: assume $A$ is symmetric with just $\text{tril}(A)$ 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 $\text{triu}(A)$ stored; the rest of $A$ is ignored

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

- $< 0$, lower: return just $\text{tril}(C)$
- $0$, unsymmetric: return all of $C$
- $> 0$, upper: return just $\text{triu}(C)$

This gives a total of nine combinations:

<table>
<thead>
<tr>
<th>Equivalent MATLAB statements</th>
<th>Using cholmod_copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C = A$ ;</td>
<td>$A$ unsymmetric, $C$ unsymmetric</td>
</tr>
<tr>
<td>$C = \text{tril} (A)$ ;</td>
<td>$A$ unsymmetric, $C$ lower</td>
</tr>
<tr>
<td>$C = \text{triu} (A)$ ;</td>
<td>$A$ unsymmetric, $C$ upper</td>
</tr>
<tr>
<td>$U = \text{triu} (A)$ ; $L = \text{tril} (U',-1)$ ; $C = L+U$ ;</td>
<td>$A$ upper, $C$ unsymmetric</td>
</tr>
<tr>
<td>$C = \text{triu} (A)'$ ;</td>
<td>$A$ upper, $C$ lower</td>
</tr>
<tr>
<td>$C = \text{triu} (A)$ ;</td>
<td>$A$ upper, $C$ upper</td>
</tr>
<tr>
<td>$L = \text{tril} (A)$ ; $U = \text{triu} (L',1)$ ; $C = L+U$ ;</td>
<td>$A$ lower, $C$ unsymmetric</td>
</tr>
<tr>
<td>$C = \text{tril} (A)$ ;</td>
<td>$A$ lower, $C$ lower</td>
</tr>
<tr>
<td>$C = \text{tril} (A)'$ ;</td>
<td>$A$ lower, $C$ upper</td>
</tr>
</tbody>
</table>

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).
11.18 cholmod_add: add sparse matrices

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).

11.19 cholmod_sparse xtype: change sparse xtype

```
int cholmod_sparse xtype
(
  /* ---- input ---- */
  int to xtype, /* requested xtype (pattern, real, complex, zomplex) */
  /* ---- in/out --- */
  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.
12 Core Module: cholmod_factor object

12.1 cholmod_factor object: a sparse Cholesky factorization

```c
typedef struct cholmod_factor_struct
{
  /* ----------------------------- */
  /* for both simplicial and supernodal factorizations */
  /* ----------------------------- */

  size_t n ;    /* L is n-by-n */

  size_t minor ;    /* If the factorization failed, L->minor is the column */
  /* at which it failed (in the range 0 to n-1). A value */
  /* of n means the factorization was successful or */
  /* the matrix has not yet been factorized. */
  /* ----------------------------- */
  /* symbolic ordering and analysis */
  /* ----------------------------- */

  void *Perm ;   /* size n, permutation used */
  void *ColCount ;    /* size n, column counts for simplicial L */

  /* ----------------------------- */
  /* simplicial factorization */
  /* ----------------------------- */

  size_t nzmax ;    /* size of i and x */

  void *p ;    /* p [0..ncol], the column pointers */
  void *i ;    /* i [0..nzmax-1], the row indices */
  void *x ;    /* x [0..nzmax-1], the numerical values */
  void *z ;
  void *nz ;    /* nz [0..ncol-1], the # of nonzeros in each column. */
  /* i [p[j] ... p[j]+nz[j]-1] contains the row indices, */
  /* and the numerical values are in the same locatins */
  /* in x. The value of i [p[k]] is always k. */

  void *next ;    /* size ncol+2. next [j] is the next column in i/x */
  void *prev ;    /* size ncol+2. prev [j] is the prior column in i/x. */
  /* head of the list is ncol+1, and the tail is ncol. */

  /* ----------------------------- */
  /* supernodal factorization */
  /* ----------------------------- */

  size_t nsuper ;    /* number of supernodes */
  size_t ssize ;    /* size of s, integer part of supernodes */
  size_t xsise ;    /* size of x, real part of supernodes */
  size_t maxcsize ;    /* size of largest update matrix */
  size_t maxesize ;    /* max # of rows in supernodes, excl. triangular part */

  void *super ;    /* size nsuper+1, first col in each supernode */
```

void *pi; /* size nsuper+1, pointers to integer patterns */
void *px; /* size nsuper+1, pointers to real parts */
void *s; /* size ssize, integer part of supernodes */

/* factorization type */
/* ---------------------------------------------------------------------- */
int ordering; /* ordering method used */
int is_ll; /* TRUE if LL', FALSE if LDL' */
int is_super; /* TRUE if supernodal, FALSE if simplicial */
int is_monotonic; /* TRUE if columns of L appear in order 0..n-1.
  * Only applicable to simplicial numeric types. */

/* There are 8 types of factor objects that cholmod_factor can represent
  * (only 6 are used):
  * *
  * Numeric types (xtype is not CHOLMOD_PATTERN)
  * -------------------------------
  * *
  * 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 LDL': (is_ll FALSE, is_super TRUE). Not used.
  * FUTURE WORK: add support for supernodal LDL'
  * *
  * 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).
  * *
  * Symbolic types (xtype is CHOLMOD_PATTERN)
  * -------------------------------
  * *
  * 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 LDL': (is_ll FALSE, is_super TRUE). Not used.
  * FUTURE WORK: add support for supernodal LDL'
  * *
  * 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).
  */

int itype; /* The integer arrays are Perm, ColCount, p, i, nz,
  * next, prev, super, pi, px, and s. If itype is
Purpose: An $LL^T$ or $LDL^T$ factorization in simplicial or supernodal form. A simplicial factor is very similar to a cholmod\_sparse matrix. For an $LDL^T$ factorization, the diagonal matrix $D$ is stored as the diagonal of $L$; the unit-diagonal of $L$ is not stored.
12.2 cholmod_free_factor: free factor

```c
int cholmod_free_factor
(
    /* ---- in/out --- */
    cholmod_factor **L, /* factor to free, NULL on output */
    /* --------------- */
    cholmod_common *Common
);
```

Purpose: Frees a factor.

12.3 cholmod_allocate_factor: allocate factor

```c
int cholmod_l_free_factor (cholmod_factor **, cholmod_common *);
```

Purpose: Allocates a factor and sets it to identity.

12.4 cholmod_reallocate_factor: reallocate factor

```c
int cholmod_reallocate_factor
(
    /* ---- input ---- */
    size_t nznew, /* new # of entries in L */
    /* ---- in/out --- */
    cholmod_factor *L, /* factor to modify */
    /* --------------- */
    cholmod_common *Common
);
```

Purpose: Reallocates a simplicial factor so that it can contain \texttt{nznew} entries.
12.5 cholmod_change_factor: change factor

```c
int cholmod_change_factor
(
    /* ---- input ---- */
    int to xtype, /* to CHOLMOD_PATTERN, _REAL, _COMPLEX, _ZOMPLEX */
    int to ll, /* TRUE: convert to LL', FALSE: LDL' */
    int to super, /* TRUE: convert to supernodal, FALSE: simplicial */
    int to packed, /* TRUE: pack simplicial columns, FALSE: do not pack */
    int to monotonic, /* TRUE: put simplicial columns in order, FALSE: not */
    /* ---- in/out --- */
    cholmod_factor *L, /* factor to modify */
    cholmod_common *Common
);
int cholmod_l_change_factor ( int, int, int, int, int, cholmod_factor *,
    cholmod_common *Common ) ;
```

**Purpose:** Change the numeric or symbolic, $\text{LL}^T$ or $\text{LDL}^T$, 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 ($L->\text{Perm}$) and the nonzero count for each column of $L$ ($L->\text{ColCount}$). All other factor types also include this information. $L->\text{ColCount}$ 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. $L->\text{ColCount}$ is used to allocate space. $L->\text{ColCount}$ 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:
   - $\text{LDL}^T$: $L$ is unit diagonal. Row indices in column $j$ are located in $L->i [L->p [j] \ldots L->p [j] + L->nz [j]]$, 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 ($L->\text{prev}$ and $L->\text{next}$). $L->p$ 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 $2*\text{lnz}*\text{sizeof(double)}$. For the zomplex case, $L->x$ is of size $\text{lnz}*\text{sizeof(double)}$ and holds the real part; $L->z$ is the same size and holds the imaginary part.
   - $\text{LL}^T$: This is identical to the $\text{LDL}^T$ 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 $L->n\text{super}$ supernodes. Columns $L->\text{super} [k]$ to $L->\text{super} [k+1]-1$ are in the $k$th supernode. The row indices for the $k$th supernode are in $L->s [L->\text{pi} [k] \ldots L->\text{pi} [k+1]-1]$. The numerical values are not allocated ($L->x$), but when they are they will be located in $L->x [L->\text{px} [k] \ldots L->\text{px} [k+1]-1]$, and the $L->\text{px}$ array is defined in this factor type.
For the complex case, \(L \rightarrow x\) is stored interleaved with real/imaginary parts, and is of size 
\(2 \times L \rightarrow x\text{size} \times \text{sizeof(double)}\). The complex supernodal case is not supported, since it is not compatible with LAPACK and the BLAS.

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

In all cases, the row indices in each column (\(L \rightarrow i\) for simplicial \(L\) and \(L \rightarrow s\) for supernodal \(L\)) are kept sorted from low indices to high indices. This means the diagonal of \(L\) (or \(D\) for a \(LDL^T\) 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 \(j\)th column. If column \(j\) has no off-diagonal entries then node \(j\) is a root of the elimination tree.

The \texttt{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 \texttt{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 \texttt{cholmod\_factor\_xtype} instead.
12.6 cholmod_pack_factor: pack the columns of a factor

```c
int cholmod_pack_factor
(
   /* ---- in/out --- */
   cholmod_factor *L, /* factor to modify */
   /* --------------- */
   cholmod_common *Common
);
```

**Purpose:** Pack the columns of a simplicial LDL$^T$ or LL$^T$ 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_11, FALSE, TRUE, TRUE, L, Common)

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

12.7 cholmod_reallocate_column: reallocate one column of a factor

```c
int cholmod_reallocate_column
(
   /* ---- input ---- */
   size_t j,        /* the column to reallocate */
   size_t need,     /* required size of column j */
   /* ---- in/out --- */
   cholmod_factor *L, /* factor to modify */
   /* --------------- */
   cholmod_common *Common
);
```

**Purpose:** Reallocation of the space allotted to a single column of L.
12.8 cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.

12.9 cholmod_copy_factor: copy factor

Purpose: Returns an exact copy of a factor.

12.10 cholmod_factor_xtypes: change factor xtype

int cholmod_factor_xtypes
(int cholmod_factor_to_sparse: sparse matrix copy of a factor

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.
13 Core Module: cholmod_dense object

13.1 cholmod_dense object: a dense matrix

```c
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 float */
} cholmod_dense ;
```

**Purpose:** Contains a dense matrix.

13.2 cholmod_allocate_dense: allocate dense matrix

**Purpose:** Allocates a dense matrix.

13.3 cholmod_free_dense: free dense matrix

```c
int cholmod_free_dense
(
    /* ---- in/out --- */
    cholmod_dense **X, /* dense matrix to deallocate, NULL on output */
    /* --------------- */
    cholmod_common *Common
) ;

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

**Purpose:** Frees a dense matrix.
13.4 cholmod_zeros: dense zero matrix

Purpose: Returns an all-zero dense matrix.

13.5 cholmod.ones: dense matrix, all ones

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

13.6 cholmod_eye: dense identity matrix

Purpose: Returns a dense identity matrix.
13.7 cholmod_sparse_to_dense: dense matrix copy of a sparse matrix

Purpose: Returns a dense copy of a sparse matrix.

13.8 cholmod_dense_to_sparse: sparse matrix copy of a dense matrix

Purpose: Returns a sparse copy of a dense matrix.

13.9 cholmod_copy_dense: copy dense matrix

Purpose: Returns a copy of a dense matrix.
13.10 cholmod_copy_dense2: copy dense matrix (preallocated)

```
int cholmod_copy_dense2
(
    /* ---- input ---- */
    cholmod_dense *X, /* matrix to copy */
    /* ---- output --- */
    cholmod_dense *Y, /* copy of matrix X */
    /* --------------- */
    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.

13.11 cholmod_dense xtype: change dense matrix xtype

```
int cholmod_dense xtype
(
    /* ---- input ---- */
    int to xtype, /* requested xtype (real, complex, or zomplex) */
    /* ---- in/out --- */
    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.
14 Core Module: cholmod_triplet object

14.1 cholmod_triplet object: sparse matrix in triplet form

```c
typedef struct cholmod_triplet_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 nnz ; /* number of nonzeros in the matrix */
    void *i ; /* i [0..nzmax-1], the row indices */
    void *j ; /* j [0..nzmax-1], the column indices */
    void *x ; /* size nzmax or 2*nzmax, if present */
    void *z ; /* size nzmax, if present */
    int stype ; /* Describes what parts of the matrix are considered:
        */
        * 0: matrix is "unsymmetric": use both upper and lower triangular parts
        * (the matrix may actually be symmetric in pattern and value, but
        * both parts are explicitly stored and used). May be square or
        * rectangular.
        * >0: matrix is square and symmetric. Entries in the lower triangular
        * part are transposed and added to the upper triangular part when
        * the matrix is converted to cholmod_sparse form.
        * <0: matrix is square and symmetric. Entries in the upper triangular
        * part are transposed and added to the lower triangular part when
        * the matrix is converted to cholmod_sparse form.
        *
        * Note that stype>0 and stype<0 are different for cholmod_sparse and
        * cholmod_triplet. The reason is simple. You can permute a symmetric
        * triplet matrix by simply replacing a row and column index with their
        * new row and column indices, via an inverse permutation. Suppose
        * P = L->Perm is your permutation, and Pinv is an array of size n.
        * Suppose a symmetric matrix A is represent by a triplet matrix T, with
        * entries only in the upper triangular part. Then the following code:
        * *
        *       Ti = T->i ;
        *       Tj = T->j ;
        *       for (k = 0 ; k < n ; k++) Pinv [P [k]] = k ;
        *       for (k = 0 ; k < nz ; k++) Ti [k] = Pinv [Ti [k]] ;
        *       for (k = 0 ; k < nz ; k++) Tj [k] = Pinv [Tj [k]] ;
        *
        * creates the triplet form of C=P*A*P'. However, if T initially
        * contains just the upper triangular entries (T->stype = 1), after
        * permutation it has entries in both the upper and lower triangular
        * parts. These entries should be transposed when constructing the
        * cholmod_sparse form of A, which is what cholmod_triplet_to_sparse
        * does. Thus:
        * *
        *       C = cholmod_triplet_to_sparse (T, 0, &Common) ;
        * *
        * will return the matrix C = P*A*P'.
        *
        * Since the triplet matrix T is so simple to generate, it’s quite easy
        * to remove entries that you do not want, prior to converting T to the
        * cholmod_sparse form. So if you include these entries in T, CHOLMOD
        * assumes that there must be a reason (such as the one above). Thus,
Purpose: Contains a sparse matrix in triplet form.

14.2 cholmod_allocate_triplet: allocate triplet matrix

Purpose: Allocates a triplet matrix.

14.3 cholmod_free_triplet: free triplet matrix

Purpose: Frees a triplet matrix.
14.4 cholmod_reallocate_triplet: reallocate triplet matrix

```c
int cholmod_reallocate_triplet
(
    /* ---- input ---- */
    size_t nznew, /* new # of entries in T */
    /* ---- in/out --- */
    cholmod_triplet *T, /* triplet matrix to modify */
    /* --------------- */
    cholmod_common *Common
)
```

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

14.5 cholmod_sparse_to_triplet: triplet matrix copy of a sparse matrix

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

14.6 cholmod_triplet_to_sparse: sparse matrix copy of a triplet matrix

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.
14.7 cholmod_copy_triplet: copy triplet matrix

**Purpose:** Returns an exact copy of a triplet matrix.

14.8 cholmod_triplet_xtype: change triplet xtype

```c
int cholmod_triplet_xtype
(
    /* ---- input ---- */
    int to_xtype,    /* requested xtype (pattern, real, complex,or zomplex)*/
    /* ---- in/out --- */
    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.
15 Core Module: memory management

15.1 cholmod_malloc: allocate memory

Purpose: Allocates a block of memory of size \( n \times \text{size} \), using the \texttt{Common->malloc} memory function pointer (default is to use the ANSI C \texttt{malloc} routine). A value of \( n=0 \) is treated as \( n=1 \). If not successful, NULL is returned and \texttt{Common->status} is set to CHOLMOD\_OUT\_OF\_MEMORY.

15.2 cholmod_calloc: allocate and clear memory

Purpose: Allocates a block of memory of size \( n \times \text{size} \), using the \texttt{Common->calloc} memory function pointer (default is to use the ANSI C \texttt{calloc} routine). A value of \( n=0 \) is treated as \( n=1 \). If not successful, NULL is returned and \texttt{Common->status} is set to CHOLMOD\_OUT\_OF\_MEMORY.
15.3 cholmod_free: free memory

Purpose: Frees a block of memory of size \( n \times \text{size} \), using the Common->free_memory function pointer (default is to use the ANSI C free routine). The size of the block (\( n \) and \( \text{size} \)) 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.

15.4 cholmod_realloc: reallocate memory

Purpose: Reallocates a block of memory whose current size \( n \times \text{size} \), and whose new size will be \( n_{\text{new}} \times \text{size} \) if successful, using the Common->calloc_memory 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 \( n_{\text{new}} \) if successful, or left unchanged otherwise. A value of \( n_{\text{new}}=0 \) is treated as \( n_{\text{new}}=1 \).
15.5 cholmod_realloc_multiple: reallocate memory

```c
int cholmod_realloc_multiple
(
    /* ---- input ---- */
    size_t nnew,    /* requested # of items in reallocated blocks */
    int nint,      /* number of int/UF_long blocks */
    int xtype,      /* CHOLMOD_PATTERN, _REAL, _COMPLEX, or _ZOMPLEX */
    /* ---- in/out --- */
    void **Iblock, /* int or UF_long block */
    void **Jblock, /* int or UF_long block */
    void **Xblock, /* complex, double, or float block */
    void **Zblock, /* zomplex case only: double or float block */
    size_t *n,    /* current size of the I,J,X,Z blocks on input, */
                   /*   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.
16 Check Module routines

No CHOLMOD routines print anything, except for the cholmod_print_* routines in the Check Module, and the cholmod_error routine. The Common->print_function is a pointer to printf by default; you can redirect the output of CHOLMOD by redefining this pointer. If Common->print_function 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.

16.1 cholmod_check_common: check Common object

```c
int cholmod_check_common
(
    cholmod_common *Common
) ;
```

**Purpose:** Check if the Common object is valid.

16.2 cholmod_print_common: print Common object

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

**Purpose:** Print the Common object and check if it is valid. This prints the CHOLMOD parameters and statistics.
16.3 cholmod_check_sparse: check sparse matrix

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

**Purpose:** Check if a sparse matrix is valid.

16.4 cholmod_print_sparse: print sparse matrix

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

**Purpose:** Print a sparse matrix and check if it is valid.
16.5 cholmod_check_dense: check dense matrix

```c
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.

16.6 cholmod_print_dense: print dense matrix

```c
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.
16.7 cholmod_check_factor: check factor

```c
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.

16.8 cholmod_print_factor: print factor

```c
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.
16.9 cholmod_check_triplet: check triplet matrix

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

**Purpose:** Check if a triplet matrix is valid.

16.10 cholmod_print_triplet: print triplet matrix

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

**Purpose:** Print a triplet matrix and check if it is valid.
16.11 cholmod_check_subset: check subset

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

Purpose: Check if a subset is valid.

16.12 cholmod_print_subset: print subset

```c
int cholmod_print_subset
(
    /* ---- input ---- */
    int *Set, /* Set [0:len-1] is a subset of 0:n-1. Duplicates OK */
    UF_long 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
);
```

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

```
int cholmod_check_perm
(
    /* ---- input ---- */
    int *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 (UF_long *, size_t, size_t, cholmod_common *) ;
```

**Purpose:** Check if a permutation is valid.

16.14 cholmod_print_perm: print permutation

```
int cholmod_print_perm
(
    /* ---- input ---- */
    int *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 (UF_long *, size_t, size_t, const char *,
cholmod_common *) ;
```

**Purpose:** Print a permutation and check if it is valid.
16.15 cholmod_check_parent: check elimination tree

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

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

**Purpose:** Check if an elimination tree is valid.

16.16 cholmod_print_parent: print elimination tree

```c
int cholmod_print_parent
(
    /* ---- input ---- */
    int *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 (UF_long *, size_t, const char *, cholmod_common *);
```

**Purpose:** Print an elimination tree and check if it is valid.
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 \texttt{%%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:

\begin{verbatim}
%%MatrixMarket matrix coord type storage
\end{verbatim}

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:

\begin{verbatim}
nrow ncol nnz stype
\end{verbatim}

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 ($A = \text{mmread} ('file')$) 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.

16.18 cholmod_read_sparse: read sparse matrix from file

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.
16.19 cholmod_read_dense: read dense matrix from file

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

16.20 cholmod_read_matrix: read a matrix from file

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.
16.21 cholmod_write_sparse: write a sparse matrix to a file

```c
int cholmod_write_sparse
(
    /* ---- 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
);
```

```c
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.

16.22 cholmod_write_dense: write a dense matrix to a file

```c
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
);
```

```c
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.
17 Cholesky Module routines

17.1 cholmod_analyze: symbolic factorization

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 $\text{nnz}(L)$ is greater than or equal to $5*\text{nnz}(\text{tril}(A))$ (or $5*\text{nnz}(\text{tril}(A*A'))$ if $A$ is unsymmetric) and the floating-point operation count for the subsequent factorization is greater than or equal to $500*\text{nnz}(L)$, then METIS is tried (if installed). For cholmod_analyze.p, the user-provided ordering is also tried. This default behavior is obtained when Common->nmeths 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 $\text{nnz}(L)$ 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->nmeths 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 $n_{d,small} = 200$ or less, remove nodes with more than $\max(16, \text{prune\_dense} \times \sqrt{n})$ nodes where $\text{prune\_dense} = 10$, 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, $n_{d,small} = 20000$, $\text{prune\_dense} = 10$.

7. NESDIS, $n_{d,small} = 4$, $\text{prune\_dense} = 10$, no constrained minimum degree.

8. NESDIS, $n_{d,small} = 200$, $\text{prune\_dense} = 0$.

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 $L$ 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 $LDL^T$ by default, since this is the kind required by the Modify Module. CHOLMOD does not include a supernodal $LDL^T$ factorization, so if a supernodal factorization is selected, it will be in the form $LL^T$. The $LDL^T$ 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 $LL^T$ method is restricted to positive definite matrices. To factorize a large indefinite matrix, set **Common->supernodal** to **CHOLMOD_SIMPLICIAL**, and the simplicial $LDL^T$ method will always be used. This will be significantly slower than a supernodal $LL^T$ factorization, however.

Refer to cholmod_transpose_unsym for a description of $f$. 

104
17.2 cholmod_factorize: numeric factorization

```c
int cholmod_factorize
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to factorize */
    /* ---- in/out --- */
    cholmod_factor *L, /* resulting factorization */
    /* ----------------- */
    cholmod_common *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 `LL^T` factorization is computed. Otherwise, a simplicial numeric factorization is computed, either `LL^T` or `LDL^T`, depending on `Common->final_ll` (the default for the simplicial case is to compute an `LDL^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.
17.3 cholmod_analyze_p: symbolic factorization, given permutation

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 unsymmetric case.

17.4 cholmod_factorize_p: numeric factorization, given permutation

```c
int cholmod_factorize_p
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to factorize */
    double beta [2], /* factorize beta*I+A or beta*I+A'*A */
    int *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 *, UF_long *, 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.
17.5 cholmod_solve: solve a linear system

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

<table>
<thead>
<tr>
<th>system</th>
<th>sys parameter</th>
<th>system</th>
<th>sys parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Ax = b$</td>
<td>0: CHOLMOD_A</td>
<td>$L^T x = b$</td>
<td>5: CHOLMOD_Lt</td>
</tr>
<tr>
<td>$LDL^T x = b$</td>
<td>1: CHOLMOD_LDLt</td>
<td>$L^T = b$</td>
<td>5: CHOLMOD_Lt</td>
</tr>
<tr>
<td>$LD^T x = b$</td>
<td>2: CHOLMOD_LD</td>
<td>$Dx = b$</td>
<td>6: CHOLMOD_D</td>
</tr>
<tr>
<td>$DL^T x = b$</td>
<td>3: CHOLMOD_DLt</td>
<td>$x = Pb$</td>
<td>7: CHOLMOD_P</td>
</tr>
<tr>
<td>$Lx = b$</td>
<td>4: CHOLMOD_L</td>
<td>$x = P^T b$</td>
<td>8: CHOLMOD_Pt</td>
</tr>
</tbody>
</table>

The factorization can be simplicial $LDL^T$, simplicial $LL^T$, or supernodal $LL^T$. For an $LL^T$ factorization, $D$ is the identity matrix. Thus CHOLMOD_LD and CHOLMOD_L solve the same system if an $LL^T$ 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 $L$ or $D$ 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 $LL^T$ factorization, or $D(k,k)$ is zero for an $LDL^T$ factorization.

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

17.6 cholmod_spsolve: solve a linear system

Purpose: Identical to cholmod_spsolve, except that $B$ and $X$ are sparse.
17.7 cholmod_etree: find elimination tree

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

17.8 cholmod_rowcolcounts: nonzeros counts of a factor

```c
int cholmod_rowcolcounts
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to analyze */
    int *fset, /* subset of 0:(A->ncol)-1 */
    size_t fsize, /* size of fset */
    int *Parent, /* size nrow. Parent [i] = p if p is the parent of i */
    int *Post, /* size nrow. Post [k] = i if i is the kth node in */
    /* the postordered etree. */
    /* ---- output --- */
    int *RowCount, /* size nrow. RowCount [i] = # entries in the ith row of */
    /* L, including the diagonal. */
    int *ColCount, /* size nrow. ColCount [i] = # entries in the ith */
    /* column of L, including the diagonal. */
    int *First, /* size nrow. First [i] = k is the least postordering */
    /* of any descendant of i. */
    int *Level, /* size nrow. Level [i] is the length of the path from */
    /* i to the root, with Level [root] = 0. */
    /* --------------- */
    cholmod_common *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 ($LL^T = A$), $A$ 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->f1 (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].
17.9  cholmod_analyze_ordering: analyze a permutation

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

int cholmod_l_analyze_ordering (cholmod_sparse *, int, UF_long *, UF_long *,
    size_t, UF_long *, UF_long *, UF_long *, UF_long *,
    cholmod_common *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 \( \text{tril}(A) \) (\( \text{Common->fl} \), \( \text{Common->lnz} \), and \( \text{Common->anz} \)). In the unsymmetric case, \( A(p,f)A(p,f)' \) is analyzed, and \( \text{Common->anz} \) 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.
17.10 cholmod_amd: interface to AMD

```c
int cholmod_amd
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to order */
    int *fset, /* subset of 0:(A->ncol)-1 */
    size_t fsize, /* size of fset */
    /* ---- output --- */
    int *Perm, /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *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 \( k \)th 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 \( k \)th 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 \( LL^T \) 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.

17.11 cholmod_colamd: interface to COLAMD

```c
int cholmod_colamd
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to order */
    int *fset, /* subset of 0:(A->ncol)-1 */
    size_t fsize, /* size of fset */
    int postorder, /* if TRUE, follow with a coletree postorder */
    /* ---- output --- */
    int *Perm, /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *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 (\( A->stype = 0 \)).
17.12 cholmod_rowfac: row-oriented Cholesky factorization

```c
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 */
    int *mask, /* if mask[i] >= 0, then set row i to zero */
    int *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, UF_long *, UF_long *, cholmod_factor *, cholmod_common *) ;
```

Purpose: Full or incremental numerical LDLᵀ or LLᵀ 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 LDLᵀ or LLᵀ factorization of A+β*I (where A is symmetric) or A*F+β*I (where A and F are unsymmetric and only the upper triangular part of A*F+β*I is used). It computes L (and D, for LDLᵀ) one row at a time. The input scalar β is real; only the real part (β[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 complex 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.

17.13 cholmod_rowfac_mask: row-oriented Cholesky factorization

```c
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 */
int *mask, /* if mask[i] >= 0, then set row i to zero */
int *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, UF_long *, UF_long *, cholmod_factor *, cholmod_common *);
```

**Purpose:** For use in LPDASA only.
17.14 cholmod_row_subtree: pattern of row of a factor

```c
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 */
    int *Parent, /* elimination tree */
    /* ---- output --- */
    cholmod_sparse *R, /* pattern of L(k,:), 1-by-n with R->nzmax >= n */
    /* --------------- */
    cholmod_common *Common
) ;
```

```c
int cholmod_l_row_subtree (cholmod_sparse *, cholmod_sparse *, size_t,
    UF_long *, cholmod_sparse *, cholmod_common *) ;
```

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

$$L(0:k-1,0:k-1) \times x = A(0:k-1,k)$$

if $A$ is symmetric, or

$$L(0:k-1,0:k-1) \times x = A(0:k-1,:) \times A(:,k)'$$

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 $\text{nrow}$-by-1, with $R->\text{nzmax} \geq \text{nrow}$. $R$ is assumed to be packed ($R\text{nz}[0]$ is not updated); the number of entries in $R$ is given by $R\text{p}[0]$. 

### 17.15 cholmod_row_lsubtree: pattern of row of a factor

```c
int cholmod_row_lsubtree
(
    cholmod_sparse *A, /* matrix to analyze */
    int *Fi, size_t fnz, /* nonzero pattern of kth row of A', not required */
    size_t k, /* row k of L */
    cholmod_factor *L, /* the factor L from which parent(i) is derived */
    cholmod_sparse *R, /* pattern of L(k,:), 1-by-n with R->nzmax >= n */
    cholmod_common *Common
);

int cholmod_l_row_lsubtree (cholmod_sparse *, UF_long *, 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.
17.16 cholmod_resymbol: re-do symbolic factorization

```c
int cholmod_resymbol
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to analyze */
    int *fset, /* subset of 0:(A->ncol)-1 */
    size_t fsise, /* 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
) ;
```

**Purpose:** Recompute the symbolic pattern of L. Entries not in the symbolic pattern of the factorization of \(A(p,p)\) or \(F^TF'\), 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.

17.17 cholmod_resymbol_noperm: re-do symbolic factorization

```c
int cholmod_resymbol_noperm
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to analyze */
    int *fset, /* subset of 0:(A->ncol)-1 */
    size_t fsise, /* 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
) ;
```

**Purpose:** Identical to cholmod_resymbol, except that the fill-reducing ordering \(L->Perm\) is not used.
17.18 cholmod_postorder: tree postorder

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

UF_long cholmod_l_postorder (UF_long *, size_t, UF_long *, UF_long *,
    cholmod_common *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 kth 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 kth 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 ≥ n are treated as n-1).

17.19 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 LDL^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.
18 Modify Module routines

18.1 cholmod_updown: update/downdate

```c
int cholmod_updown
(
    /* ---- 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 LDLᵀ factorization (symbolic, then numeric), by computing a new factorization of

\[ \text{LDL}^T = \text{LDL}^T \pm \text{CC}^T \]

where \(\text{L}\) denotes the new factor. \(\text{C}\) must be sorted. It can be either packed or unpacked. As in all CHOLMOD routines, the columns of \(\text{L}\) are sorted on input, and also on output. If \(\text{L}\) does not contain a simplicial numeric \(\text{LDL}^T\) factorization, it is converted into one. Thus, a supernodal \(\text{LL}^T\) factorization can be passed to `cholmod_updown`. A symbolic \(\text{L}\) 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 \(\text{L}\rightarrow\text{Perm}\) is not used. The row indices of \(\text{C}\) refer to the rows of \(\text{L}\), not \(A\). If your original system is \(\text{LDL}^T = \text{PAP}^T\) (where \(\text{P} = \text{L}\rightarrow\text{Perm}\)), and you want to compute the \(\text{LDL}^T\) factorization of \(A + \text{CC}^T\), then you must permute \(\text{C}\) first. That is, if

\[ \text{PAP}^T = \text{LDL}^T \]

is the initial factorization, then

\[ \text{P}(A + \text{CC}^T)\text{P}^T = \text{PAP}^T + \text{PCC}^T\text{P}^T = \text{LDL}^T + (\text{PC})(\text{PC})^T = \text{LDL}^T + \text{CC}^T \]

where \(\text{C} = \text{PC}\).

You can use the `cholmod_submatrix` routine in the MatrixOps Module to permute \(\text{C}\), with:

```c
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 \(\text{C}\) with sorted columns. Only real matrices are supported. The algorithms are described in [8, 9].

18.2 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 \( Lx = b \) is also updated/downdated. The new system is \( Lx = b + \Delta b \). The old solution \( x \) is overwritten with \( x \). Note that as in the update/downdate of \( L \) itself, the fill-reducing permutation \( L->Perm \) is not used. The vectors \( x \) and \( b \) are in the permuted ordering, not your original ordering. This routine does not handle multiple right-hand-sides.
18.3 cholmod_updown_mask: update/downdate

```c
int cholmod_updown_mask
(
    /* ---- input ---- */
    int update,   /* TRUE for update, FALSE for downdate */
    cholmod_sparse *C, /* the incoming sparse update */
    int *colmark, /* int 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
); 
```

Purpose: Identical to cholmod_updown_solve, except that only part of \( L \) is used in the update of the solution to \( Lx = b \). 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.

18.4 cholmod_updown_mask: update/downdate

```c
int cholmod_updown_mask
(
    /* ---- input ---- */
    int update,   /* TRUE for update, FALSE for downdate */
    cholmod_sparse *C, /* the incoming sparse update */
    int *colmark, /* int array of size n. See cholmod_updown.c */
    int *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
); 
```

Purpose: For use in LPDASA only.
18.5 cholmod_rowadd: add row to factor

```c
int cholmod_rowadd
(
    /* ---- 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
);
```

**Purpose:** Adds a row and column to an LDLᵀ 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].

18.6 cholmod_rowadd_solve: add row to factor

```c
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
);
```

**Purpose:** Identical to cholmod_rowadd, except the system Lx = b is also updated/downdated, just like cholmod_updown_solve.
18.7 cholmod_rowdel: delete row from factor

```c
int cholmod_rowdel
(
    /* ---- 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
)
```

**Purpose:** Deletes a row and column from an LDLᵀ 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.

18.8 cholmod_rowdel_solve: delete row from factor

```c
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
)
```

**Purpose:** Identical to cholmod_rowdel, except the system Lx = b is also updated/downdated, just like cholmod_updown_solve. When row/column k of A is deleted from the system Ay = b, this can induce a change to x, in addition to changes arising when L and b are modified. If this is the case, the kth entry of y is required as input (yk). The algorithm is described in [10].
18.9 cholmod_rowadd_mark: add row to factor

```c
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 */
    int *colmark, /* int 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 *, UF_long *,
    cholmod_factor *, cholmod_dense *, cholmod_dense *,
    cholmod_common *);  
```

**Purpose:** Identical to cholmod_rowadd_solve, except that only part of L is used in the update of the solution to Lx = 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.

18.10 cholmod_rowdel_mark: delete row from factor

```c
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 */
    int *colmark, /* int 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 *, UF_long *,
    cholmod_factor *, cholmod_dense *, cholmod_dense *,
    cholmod_common *);  
```

**Purpose:** Identical to cholmod_rowadd_solve, except that only part of L is used in the update of the solution to Lx = 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.
19 MatrixOps Module routines

19.1 cholmod_drop: drop small entries

```c
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.

19.2 cholmod_norm_dense: dense matrix norm

```c
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.

19.3 cholmod_norm_sparse: sparse matrix norm

```c
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.
19.4 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.
19.5 cholmod_sdmult: sparse-times-dense matrix

```c
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
)
);
```

**Purpose:** Sparse matrix times dense matrix: \( Y = \alpha(A\times X) + \beta Y \) or \( Y = \alpha(A'\times X) + \beta Y \), 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 \( \text{transpose} = 0 \), then \( A \) is used; otherwise, \( A' \) is used (the complex conjugate transpose). The \( \text{transpose} \) 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.

19.6 cholmod_ssmult: sparse-times-sparse matrix

```c
int cholmod_ssmult (cholmod_sparse *, int, double *, double *,
   cholmod_dense *, cholmod_dense *, cholmod_common *) ;
```

**Purpose:** Computes \( C = A\times B \); multiplying two sparse matrices. \( C \) is returned as packed, and either unsorted or sorted, depending on the \( \text{sorted} \) input parameter. If \( C \) is returned sorted, then either \( C = (B'\times A')' \) or \( C = (A\times B)' \) is computed, depending on the number of nonzeros in \( A \), \( B \), and \( C \). The stype of \( C \) is determined by the \( \text{stype} \) parameter. Only pattern and real matrices are supported. Complex and zomplex matrices are supported only when the numerical values are not computed (\( \text{values} \) is FALSE).

125
19.7 cholmod_submatrix: sparse submatrix

**Purpose:** Returns $C = A(rset, cset)$, where $C$ becomes $\text{length}(rset)$-by-$\text{length}(cset)$ in dimension. $rset$ and $cset$ can have duplicate entries. $A$ must be unsymmetric. $C$ unsymmetric and is packed. If $\text{sorted}$ 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 $\text{rs}\text{ize}$ or $\text{cs}\text{ize}$ is negative, it denotes “;” in MATLAB notation. Thus, if both $\text{rs}\text{ize}$ and $\text{cs}\text{ize}$ 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 $O(A->n\text{row})$ if the Common workspace needs to be initialized, plus $O(C->n\text{row} + C->n\text{col} + \text{nnz}(A(:, cset)))$. 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 $\text{values}$ is FALSE.
19.8 cholmod_horzcat: horizontal concatenation

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.

19.9 cholmod_vertcat: vertical concatenation

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.
19.10 cholmod_symmetry: compute the symmetry of a matrix

```c
int cholmod_symmetry
(
    /* ---- input ---- */
    cholmod_sparse *A,
    int option,
    /* ---- output ---- */
    int *xmatched,
    int *pmatched,
    int *nzoffdiag,
    int *nzdiag,
    /* --------------- */
    cholmod_common *Common
);
```

```c
int cholmod_l_symmetry (cholmod_sparse *, int, UF_long *, UF_long *, UF_long *,
    UF_long *, 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 UF Sparse Matrix Collection.

- **Numerical symmetry:** number of matched offdiagonal nonzeros over the total number of offdiagonal 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 $\text{conj}(a_{ji}) = a_{ij}$.

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:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTY (-1)</td>
<td>out of memory, stype not zero, A not sorted</td>
</tr>
<tr>
<td>CHOLMOD_MM_RECTANGULAR 1</td>
<td>A is rectangular</td>
</tr>
<tr>
<td>CHOLMOD_MM_UNSymmetric 2</td>
<td>A is unsymmetric</td>
</tr>
<tr>
<td>CHOLMOD_MM_SYMMETRIC 3</td>
<td>A is symmetric, but with non-pos. diagonal</td>
</tr>
<tr>
<td>CHOLMOD_MM_HERMITIAN 4</td>
<td>A is Hermitian, but with non-pos. diagonal</td>
</tr>
<tr>
<td>CHOLMOD_MM_SKEW_SYMMETRIC 5</td>
<td>A is skew symmetric</td>
</tr>
<tr>
<td>CHOLMOD_MM_SYMMETRIC_POSDIAG 6</td>
<td>A is symmetric with positive diagonal</td>
</tr>
<tr>
<td>CHOLMOD_MM_HERMITIAN_POSDIAG 7</td>
<td>A is Hermitian with positive diagonal</td>
</tr>
</tbody>
</table>

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.
20 Supernodal Module routines

20.1 cholmod_super_symbolic: supernodal symbolic factorization

```c
int cholmod_super_symbolic
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to analyze */
    cholmod_sparse *F, /* F = Aᵀ or A(:,f)ᵀ */
    int *Parent, /* elimination tree */
    /* ---- in/out --- */
    cholmod_factor *L, /* simplicial symbolic on input, *
    * supernodal symbolic on output */
    /* --------------- */
    cholmod_common *Common
);
```

```c
int cholmod_l_super_symbolic (cholmod_sparse *, cholmod_sparse *, UF_long *,
    cholmod_factor *, cholmod_common *) ;
int cholmod_super_symbolic2
(
    /* ---- input ---- */
    int for_cholesky, /* Cholesky if TRUE, QR if FALSE */
    cholmod_sparse *A, /* matrix to analyze */
    cholmod_sparse *F, /* F = Aᵀ or A(:,f)ᵀ */
    int *Parent, /* elimination tree */
    /* ---- in/out --- */
    cholmod_factor *L, /* simplicial symbolic on input, *
    * supernodal symbolic on output */
    /* --------------- */
    cholmod_common *Common
);
```

```c
int cholmod_l_super_symbolic2 (int, cholmod_sparse *, cholmod_sparse *,
    UF_long *, cholmod_factor *, cholmod_common *) ;
```

**Purpose:** Supernodal symbolic analysis of the $LLᵀ$ 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 $\text{Parent}$ and $\text{L->ColCount}$ to be defined on input; these are the simplicial $\text{Parent}$ and $\text{ColCount}$ arrays as computed by **cholmod_rowcolcounts**. Does not use $\text{L->Perm}$; the input matrices $A$ and $F$ must already be properly permuted. Allocates and computes the supernodal pattern of $L$ ($L->\text{super}$, $L->\text{pi}$, $L->\text{px}$, and $L->\text{s}$). Does not allocate the real part ($L->x$).
20.2 cholmod_super_numeric: supernodal numeric factorization

```c
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 $p$ 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 $LL^T$ factorization. Supernodal $LDL^T$ 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.
20.3 cholmod_super_lsolve: supernodal forward solve

```c
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
) ;
```

**Purpose:** Solve $Lx = b$ for a supernodal factorization. This routine does not apply the permutation $L\rightarrow Perm$. 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.

20.4 cholmod_super_ltsolve: supernodal backsolve

```c
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
) ;
```

**Purpose:** Solve $L'Tx = b$ for a supernodal factorization. This routine does not apply the permutation $L\rightarrow Perm$. 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.
21 Partition Module routines

21.1 cholmod_nested_dissection: nested dissection ordering

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 (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_NodesND, except for how it treats the leaf nodes. METIS_NodesND 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.
21.2 cholmod_metis: interface to METIS nested dissection

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

```
int cholmod_l_metis (cholmod_sparse *, UF_long *, size_t, int, UF_long *,
    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.
21.3 cholmod_camd: interface to CAMD

```c
int cholmod_camd
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to order */
    int *fset, /* subset of 0:(A->ncol)-1 */
    size_t fsize, /* size of fset */
    /* ---- output --- */
    int *Cmember, /* size nrow. see cholmod_ccolamd above */
    int *Perm, /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *Common
);
```

```c
int cholmod_l_camd (cholmod_sparse *, UF_long *, size_t, UF_long *, UF_long *,
    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.
21.4 cholmod_ccolamd: interface to CCOLAMD

```c
int cholmod_ccolamd
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to order */
    int *fset, /* subset of 0:(A->n) - 1 */
    size_t fsize, /* size of fset */
    int *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 interpreted as Cmember [i] = 0 for
                   * all i */
    /* ---- output --- */
    int *Perm, /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *Common
)
);

int cholmod_l_ccolamd (cholmod_sparse *, UF_long *, size_t, UF_long *,
                        UF_long *, 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.

21.5 cholmod_csymamd: interface to CSYMAMD

```c
int cholmod_csymamd
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to order */
    /* ---- output --- */
    int *Cmember, /* size nrow. see cholmod_ccolamd above */
    int *Perm, /* size A->nrow, output permutation */
    /* --------------- */
    cholmod_common *Common
)
);

int cholmod_l_csymamd (cholmod_sparse *, UF_long *, UF_long *,
                        UF_long *, cholmod_common *) ;
```

**Purpose:** CHOLMOD interface to the CSYMAMD ordering routine. Finds a permutation $p$ such that the Cholesky factorization of $A(p,p)*A(p,p)'$ is sparser than $A*A'$. 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.
21.6 cholmod_bisect: graph bisector

UF_long cholmod_bisect /* returns # of nodes in separator */
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to bisect */
    int *fset, /* subset of 0:(A->ncol)-1 */
    size_t fsize, /* size of fset */
    int compress, /* if TRUE, compress the graph first */
    /* ---- output --- */
    int *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
);

UF_long cholmod_l_bisect (cholmod_sparse *, UF_long *, size_t, int, UF_long *,
    cholmod_common *) ;

Purpose: Finds a node bisector of A, A*\textasciitilde{A}', A(:,f)*A(:,f)': a set of nodes that partitions the graph into two parts. Compresses the graph first, and then calls METIS.

21.7 cholmod_metis_bisector: interface to METIS node bisector

UF_long cholmod_metis_bisector /* returns separator size */
(
    /* ---- input ---- */
    cholmod_sparse *A, /* matrix to bisect */
    int *Anw, /* size A->nrow, node weights */
    int *Aew, /* size nz, edge weights */
    /* ---- output --- */
    int *Partition, /* size A->nrow. see cholmod_bisect above. */
    /* --------------- */
    cholmod_common *Common
);

UF_long cholmod_l_metis_bisector (cholmod_sparse *, UF_long *, UF_long *,
    UF_long *, cholmod_common *) ;

Purpose: Finds a set of nodes that bisects the graph of A or A*\textasciitilde{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.
21.8 cholmod Collapse Septree: Prune a Separator Tree

```
UF_long 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 --- */
    int *CParent,    /* size ncomponents; from cholmod_nested_dissection */
    int *Cmember,    /* size n; from cholmod_nested_dissection */
    /* --------------- */
    cholmod_common *Common
)
;
UF_long cholmod_l_collapse_septree (size_t, size_t, double, size_t, UF_long *,
    UF_long *, cholmod_common *) ;
```

**Purpose:** Prunes a separator tree obtained from cholmod_nested_dissection.
References


