

UMFPACK Quick Start Guide

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

VERSION 6.2.2, Oct 23, 2023

Abstract

UMFPACK is a set of routines for solving unsymmetric sparse linear systems, $\mathbf{Ax} = \mathbf{b}$, using the Unsymmetric-pattern MultiFrontal method and direct sparse LU factorization. It is written in ANSI/ISO C, with a MATLAB interface. UMFPACK relies on the Level-3 Basic Linear Algebra Subprograms (dense matrix multiply) for its performance. This is a “quick start” guide for Unix/Linux users of the C interface.

UMFPACK, Copyright©2005-2023, Timothy A. Davis, All Rights Reserved.
SPDX-License-Identifier: GPL-2.0+
See <http://www.suitesparse.com> for the code and full documentation.

1 Overview

UMFPACK is a set of routines for solving systems of linear equations, $\mathbf{Ax} = \mathbf{b}$, when \mathbf{A} is sparse and unsymmetric. The sparse matrix \mathbf{A} can be square or rectangular, singular or non-singular, and real or complex (or any combination). Only square matrices \mathbf{A} can be used to solve $\mathbf{Ax} = \mathbf{b}$ or related systems. Rectangular matrices can only be factorized.

UMFPACK is a built-in routine in MATLAB used by the forward and backslash operator, and the `lu` routine. The following is a short introduction to Unix users of the C interface of UMFPACK.

The C-callable UMFPACK library consists of 32 user-callable routines and one include file. Twenty-eight of the routines come in four versions, with different sizes of integers and for real or complex floating-point numbers. This Quick Start Guide assumes you are working with real matrices (not complex) and with `int`'s as integers (not `long`'s). Refer to the User Guide for information about the complex and long integer versions. The include file `umfpack.h` must be included in any C program that uses UMFPACK.

For more details, see: *A column pre-ordering strategy for the unsymmetric-pattern multifrontal method*, Davis, T. A., ACM Trans. Math. Software, vol 30. no 2, 2004, pp. 165-195, and *Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method*, same issue, pp. 196-199.

2 Primary routines, and a simple example

Five primary UMFPACK routines are required to factorize \mathbf{A} or solve $\mathbf{Ax} = \mathbf{b}$. An overview of the primary features of the routines is given in Section 5. Additional routines are available for passing a different column ordering to UMFPACK, changing default parameters, manipulating

sparse matrices, getting the LU factors, save and loading the LU factors from a file, computing the determinant, and reporting results. See the User Guide for more information.

- `umfpack_di_symbolic`:

Pre-orders the columns of \mathbf{A} to reduce fill-in and performs a symbolic analysis. Returns an opaque `Symbolic` object as a `void *` pointer. The object contains the symbolic analysis and is needed for the numerical factorization.

- `umfpack_di_numeric`:

Numerically scales and then factorizes a sparse matrix \mathbf{PAQ} , \mathbf{PRAQ} , or $\mathbf{PR}^{-1}\mathbf{AQ}$ into the product \mathbf{LU} , where \mathbf{P} and \mathbf{Q} are permutation matrices, \mathbf{R} is a diagonal matrix of scale factors, \mathbf{L} is lower triangular with unit diagonal, and \mathbf{U} is upper triangular. Requires the symbolic ordering and analysis computed by `umfpack_di_symbolic`. Returns an opaque `Numeric` object as a `void *` pointer. The object contains the numerical factorization and is used by `umfpack_di_solve`.

- `umfpack_di_solve`:

Solves a sparse linear system ($\mathbf{Ax} = \mathbf{b}$, $\mathbf{A}^T\mathbf{x} = \mathbf{b}$, or systems involving just \mathbf{L} or \mathbf{U}), using the numeric factorization computed by `umfpack_di_numeric`.

- `umfpack_di_free_symbolic`:

Frees the `Symbolic` object created by `umfpack_di_symbolic`.

- `umfpack_di_free_numeric`:

Frees the `Numeric` object created by `umfpack_di_numeric`.

The matrix \mathbf{A} is represented in compressed column form, which is identical to the sparse matrix representation used by MATLAB. It consists of three arrays, where the matrix is m -by- n , with `nz` entries:

```
int32_t Ap [n+1] ;
int32_t Ai [nz] ;
double Ax [nz] ;
```

All nonzeros are entries, but an entry may be numerically zero. The row indices of entries in column j are stored in `Ai[Ap[j] ... Ap[j+1]-1]`. The corresponding numerical values are stored in `Ax[Ap[j] ... Ap[j+1]-1]`.

No duplicate row indices may be present, and the row indices in any given column must be sorted in ascending order. The first entry `Ap[0]` must be zero. The total number of entries in the matrix is thus `nz = Ap[n]`. Except for the fact that extra zero entries can be included, there is thus a unique compressed column representation of any given matrix \mathbf{A} .

Here is a simple main program, `umfpack_simple.c`, that illustrates the basic usage of UMF-PACK.

```
#include <stdio.h>
#include "umfpack.h"
```

```

int32_t n = 5 ;
int32_t Ap [ ] = {0, 2, 5, 9, 10, 12} ;
int32_t Ai [ ] = { 0, 1, 0, 2, 4, 1, 2, 3, 4, 2, 1, 4} ;
double Ax [ ] = {2., 3., 3., -1., 4., 4., -3., 1., 2., 2., 6., 1.} ;
double b [ ] = {8., 45., -3., 3., 19.} ;
double x [5] ;

int main (void)
{
    double *null = (double *) NULL ;
    int i ;
    void *Symbolic, *Numeric ;
    (void) umfpack_di_symbolic (n, n, Ap, Ai, Ax, &Symbolic, null, null) ;
    (void) umfpack_di_numeric (Ap, Ai, Ax, Symbolic, &Numeric, null, null) ;
    umfpack_di_free_symbolic (&Symbolic) ;
    (void) umfpack_di_solve (UMFPACK_A, Ap, Ai, Ax, x, b, Numeric, null, null) ;
    umfpack_di_free_numeric (&Numeric) ;
    for (i = 0 ; i < n ; i++) printf ("x [%d] = %g\n", i, x [i]) ;
    return (0) ;
}

```

The `Ap`, `Ai`, and `Ax` arrays represent the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 & 6 \\ 0 & -1 & -3 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 4 & 2 & 0 & 1 \end{bmatrix}.$$

and the solution is $\mathbf{x} = [1\ 2\ 3\ 4\ 5]^T$. The program uses default control settings and does not return any statistics about the ordering, factorization, or solution (`Control` and `Info` are both `(double *) NULL`).

For routines to manipulate a simpler “triplet-form” data structure for your sparse matrix \mathbf{A} , refer to the UMFPACK User Guide.

3 Synopsis of primary C-callable routines

The matrix \mathbf{A} is m -by- n with nz entries. The optional `umfpack_di_defaults` routine loads the default control parameters into the `Control` array. The settings can then be modified before passing the array to the other routines. Refer to the description of each function in `umfpack.h`.

```

#include "umfpack.h"
int status, sys ; int32_t n, m, nz, Ap [n+1], Ai [nz] ;
double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO], Ax [nz], X [n], B [n] ;
void *Symbolic, *Numeric ;

umfpack_di_defaults (Control) ;
status = umfpack_di_symbolic (m, n, Ap, Ai, Ax, &Symbolic, Control, Info) ;
status = umfpack_di_numeric (Ap, Ai, Ax, Symbolic, &Numeric, Control, Info) ;
status = umfpack_di_solve (sys, Ap, Ai, Ax, X, B, Numeric, Control, Info) ;
umfpack_di_free_symbolic (&Symbolic) ;
umfpack_di_free_numeric (&Numeric) ;

```

4 Installation

You will need to install both UMFPACK and AMD to use UMFPACK. The UMFPACK and AMD subdirectories must be placed side-by-side within the same parent directory. AMD is a stand-alone package that is required by UMFPACK. UMFPACK can be compiled without the BLAS but your performance will be much less than what it should be.

UMFPACK also requires CHOLMOD, CCAMD, CCOLAMD, COLAMD, and SuiteSparse_metis (a slightly modified version of the original METIS v5.1.0) by default. You can remove this dependency by compiling with `-DNCHOLMOD`; see the `CMakeLists.txt` file.

CMake is used to build the UMFPACK library. An optional top-level Makefile simplifies its use. To compile and install the library for system-wide usage:

```
make ; sudo make install
```

To compile/install for local usage (SuiteSparse/lib and SuiteSparse/include)

```
make local ; sudo make install
```

To run the demos

```
make demos
```

For Windows, simply import the `CMakeLists.txt` script into Visual Studio.

Use the MATLAB command `umfpack_make` in the MATLAB directory to compile UMFPACK and AMD for use in MATLAB.

The `UMFPACK_CONFIG` string can include combinations of the following; most deal with how the BLAS are called:

- `-DNBLAS` if you do not have any BLAS at all.
- `-DLONGBLAS` if your BLAS takes non-`int32_t` integer arguments.
- `-DBLAS_INT =` the integer used by the BLAS.
- `-DNRECIPROCAL` controls a trade-off between speed and accuracy. This is off by default (speed preferred over accuracy) except when compiling for MATLAB.

When you compile your program that uses the C-callable UMFPACK library, you need to link your program with all libraries: `-lumfpack -lamd -lcholmod -lcolamd -lccolamd -lcamd -lmetis -lsuitesparseconfig`. If you don't compile UMFPACK to use METIS, then you can just use `-lumfpack -lamd -lsuitesparseconfig`.

All libraries are placed in `SuiteSparse/lib` and all include files are placed in `SuiteSparse/include`, and `make install` will also place them where they are available system-wide. To install for just yourself, use `make local` and then `make install`.

You do not need to directly include any AMD include files in your program, unless you directly call AMD routines. You only need the

```
#include "umfpack.h"
```

statement, as described in Section 3.

5 The primary UMFPACK routines

5.1 umfpack_di_symbolic

```
int umfpack_di_symbolic
(
    int32_t n_row,
    int32_t n_col,
    const int32_t Ap [ ],
    const int32_t Ai [ ],
    const double Ax [ ],
    void **Symbolic,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

int umfpack_dl_symbolic
(
    int64_t n_row,
    int64_t n_col,
    const int64_t Ap [ ],
    const int64_t Ai [ ],
    const double Ax [ ],
    void **Symbolic,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

int umfpack_zi_symbolic
(
    int32_t n_row,
    int32_t n_col,
    const int32_t Ap [ ],
    const int32_t Ai [ ],
    const double Ax [ ], const double Az [ ],
    void **Symbolic,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

int umfpack_zl_symbolic
(
    int64_t n_row,
    int64_t n_col,
    const int64_t Ap [ ],
    const int64_t Ai [ ],
    const double Ax [ ], const double Az [ ],
    void **Symbolic,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

/*
double int32_t Syntax:
```

```

#include "umfpack.h"
void *Symbolic ;
int32_t n_row, n_col, *Ap, *Ai ;
double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO], *Ax ;
int status = umfpack_di_symbolic (n_row, n_col, Ap, Ai, Ax,
    &Symbolic, Control, Info) ;

```

double int64_t Syntax:

```

#include "umfpack.h"
void *Symbolic ;
int64_t n_row, n_col, *Ap, *Ai ;
double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO], *Ax ;
int status = umfpack_dl_symbolic (n_row, n_col, Ap, Ai, Ax,
    &Symbolic, Control, Info) ;

```

complex int32_t Syntax:

```

#include "umfpack.h"
void *Symbolic ;
int32_t n_row, n_col, *Ap, *Ai ;
double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO], *Ax, *Az ;
int status = umfpack_zi_symbolic (n_row, n_col, Ap, Ai, Ax, Az,
    &Symbolic, Control, Info) ;

```

complex int64_t Syntax:

```

#include "umfpack.h"
void *Symbolic ;
int64_t n_row, n_col, *Ap, *Ai ;
double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO], *Ax, *Az ;
int status = umfpack_zl_symbolic (n_row, n_col, Ap, Ai, Ax, Az,
    &Symbolic, Control, Info) ;

```

packed complex Syntax:

Same as above, except Az is NULL.

Purpose:

Given nonzero pattern of a sparse matrix A in column-oriented form, umfpack*_symbolic performs a column pre-ordering to reduce fill-in (using COLAMD, AMD or METIS) and a symbolic factorization. This is required before the matrix can be numerically factorized with umfpack*_numeric. If you wish to bypass the COLAMD/AMD/METIS pre-ordering and provide your own ordering, use umfpack*_qsymbolic instead. If you wish to pass in a pointer to a user-provided ordering function, use umfpack*_fsymbolic.

Since umfpack*_symbolic and umfpack*_qsymbolic are very similar, options for both routines are discussed below.

For the following discussion, let S be the submatrix of A obtained after eliminating all pivots of zero Markowitz cost. S has dimension (n_row-n1-nempty_row) -by- (n_col-n1-nempty_col), where

```
n1 = Info [UMFPACK_COL_SINGLETONS] + Info [UMFPACK_ROW_SINGLETONS],
nempty_row = Info [UMFPACK_NEMPTY_ROW] and
nempty_col = Info [UMFPACK_NEMPTY_COL].
```

Returns:

The status code is returned. See Info [UMFPACK_STATUS], below.

Arguments:

```
Int n_row ;          Input argument, not modified.
Int n_col ;          Input argument, not modified.
```

A is an n_row-by-n_col matrix. Restriction: n_row > 0 and n_col > 0.

```
Int Ap [n_col+1] ;  Input argument, not modified.
```

Ap is an integer array of size n_col+1. On input, it holds the "pointers" for the column form of the sparse matrix A. Column j of the matrix A is held in Ai [(Ap [j]) ... (Ap [j+1]-1)]. The first entry, Ap [0], must be zero, and Ap [j] <= Ap [j+1] must hold for all j in the range 0 to n_col-1. The value nz = Ap [n_col] is thus the total number of entries in the pattern of the matrix A. nz must be greater than or equal to zero.

```
Int Ai [nz] ;       Input argument, not modified, of size nz = Ap [n_col].
```

The nonzero pattern (row indices) for column j is stored in Ai [(Ap [j]) ... (Ap [j+1]-1)]. The row indices in a given column j must be in ascending order, and no duplicate row indices may be present. Row indices must be in the range 0 to n_row-1 (the matrix is 0-based). See umfpack*_triplet_to_col for how to sort the columns of a matrix and sum up the duplicate entries. See umfpack*_report_matrix for how to print the matrix A.

```
double Ax [nz] ;    Optional input argument, not modified. May be NULL.
                   Size 2*nz for packed complex case.
```

The numerical values of the sparse matrix A. The nonzero pattern (row indices) for column j is stored in Ai [(Ap [j]) ... (Ap [j+1]-1)], and the corresponding numerical values are stored in Ax [(Ap [j]) ... (Ap [j+1]-1)]. Used only for gathering statistics about how many nonzeros are placed on the diagonal by the fill-reducing ordering.

```
double Az [nz] ;    Optional input argument, not modified, for complex
                   versions. May be NULL.
```

For the complex versions, this holds the imaginary part of A. The imaginary part of column j is held in Az [(Ap [j]) ... (Ap [j+1]-1)].

If Az is NULL, then both real and imaginary parts are contained in Ax[0..2*nz-1], with Ax[2*k] and Ax[2*k+1] being the real and imaginary part of the kth entry.

Used for statistics only. See the description of Ax, above.

void **Symbolic ; Output argument.

**Symbolic is the address of a (void *) pointer variable in the user's calling routine (see Syntax, above). On input, the contents of this variable are not defined. On output, this variable holds a (void *) pointer to the Symbolic object (if successful), or (void *) NULL if a failure occurred.

double Control [UMFPACK_CONTROL] ; Input argument, not modified.

If a (double *) NULL pointer is passed, then the default control settings are used (the defaults are suitable for all matrices, ranging from those with highly unsymmetric nonzero pattern, to symmetric matrices). Otherwise, the settings are determined from the Control array. See umfpack*_defaults on how to fill the Control array with the default settings. If Control contains NaN's, the defaults are used. The following Control parameters are used:

Control [UMFPACK_STRATEGY]: This is the most important control parameter. It determines what kind of ordering and pivoting strategy that UMFPACK should use.

NOTE: the interaction of numerical and fill-reducing pivoting is a delicate balance, and a perfect heuristic is not possible because sparsity-preserving pivoting is an NP-hard problem. Selecting the wrong strategy can lead to catastrophic fill-in and/or numerical inaccuracy.

UMFPACK_STRATEGY_AUTO: This is the default. The input matrix is analyzed to determine how symmetric the nonzero pattern is, and how many entries there are on the diagonal. It then selects one of the following strategies. Refer to the User Guide for a description of how the strategy is automatically selected.

UMFPACK_STRATEGY_UNSYMMETRIC: Use the unsymmetric strategy. COLAMD is used to order the columns of A, followed by a postorder of the column elimination tree. No attempt is made to perform diagonal pivoting. The column ordering is refined during factorization.

In the numerical factorization, the Control [UMFPACK_SYM_PIVOT_TOLERANCE] parameter is ignored. A pivot is selected if its magnitude is \geq Control [UMFPACK_PIVOT_TOLERANCE] (default 0.1) times the largest entry in its column.

UMFPACK_STRATEGY_SYMMETRIC: Use the symmetric strategy
In this method, the approximate minimum degree ordering (AMD) is applied to $A+A'$, followed by a postorder of the elimination tree of $A+A'$. UMFPACK attempts to perform diagonal pivoting during numerical factorization. No refinement of the column pre-ordering is performed during factorization.

In the numerical factorization, a nonzero entry on the diagonal is selected as the pivot if its magnitude is \geq Control [UMFPACK_SYM_PIVOT_TOLERANCE] (default 0.001) times the largest entry in its column. If this is not acceptable, then an off-diagonal pivot is selected with magnitude \geq Control [UMFPACK_PIVOT_TOLERANCE] (default 0.1) times the largest entry in its column.

Control [UMFPACK_ORDERING]: The ordering method to use:

UMFPACK_ORDERING_CHOLMOD	try AMD/COLAMD, then METIS if needed
UMFPACK_ORDERING_AMD	just AMD or COLAMD
UMFPACK_ORDERING_GIVEN	just Qinit (umfpack*_qsymbolic only)
UMFPACK_ORDERING_NONE	no fill-reducing ordering
UMFPACK_ORDERING_METIS	just METIS(A+A') or METIS(A'A)
UMFPACK_ORDERING_BEST	try AMD/COLAMD, METIS, and NESDIS
UMFPACK_ORDERING_USER	just user function (*_fsymbolic only)
UMFPACK_ORDERING_METIS_GUARD	use METIS, AMD, or COLAMD.

Symmetric strategy: always use METIS on A+A'. Unsymmetric strategy: use METIS on A'A, unless A has one or more very dense rows. In that case, A'A is very costly to form. In this case, COLAMD is used instead of METIS.

Control [UMFPACK_SINGLETONS]: If false (0), then singletons are not removed prior to factorization. Default: true (1).

Control [UMFPACK_DENSE_COL]:
If COLAMD is used, columns with more than $\max(16, \text{Control [UMFPACK_DENSE_COL]} * 16 * \sqrt{n_row})$ entries are placed last in the column pre-ordering. Default: 0.2.

Control [UMFPACK_DENSE_ROW]:
Rows with more than $\max(16, \text{Control [UMFPACK_DENSE_ROW]} * 16 * \sqrt{n_col})$ entries are treated differently in the COLAMD pre-ordering, and in the internal data structures during the subsequent numeric factorization. Default: 0.2.
If any row exists with more than these number of entries, and if the unsymmetric strategy is selected, the METIS_GUARD ordering selects COLAMD instead of METIS.

Control [UMFPACK_AMD_DENSE]: rows/columns in A+A' with more than $\max(16, \text{Control [UMFPACK_AMD_DENSE]} * \sqrt{n})$ entries (where $n = n_row = n_col$) are ignored in the AMD pre-ordering. Default: 10.

Control [UMFPACK_BLOCK_SIZE]: the block size to use for Level-3 BLAS in the subsequent numerical factorization (umfpack*_numeric). A value less than 1 is treated as 1. Default: 32. Modifying this parameter affects when updates are applied to the working frontal matrix, and can indirectly affect fill-in and operation count. Assuming the block size is large enough (8 or so), this parameter has a modest effect on performance.

Control [UMFPACK_FIXQ]: If > 0 , then the pre-ordering Q is not modified during numeric factorization. If < 0 , then Q may be modified. If zero, then this is controlled automatically (the unsymmetric

strategy modifies Q, the others do not). Default: 0.

Note that the symbolic analysis will in general modify the input ordering Qinit to obtain Q; see umfpack_qsymbolic.h for details. This option ensures Q does not change, as found in the symbolic analysis, but Qinit is in general not the same as Q.

Control [UMFPACK_AGGRESSIVE]: If nonzero, aggressive absorption is used in COLAMD and AMD. Default: 1.

// added for v6.0.0:

Control [UMFPACK_STRATEGY_THRESH_SYM]: tsym, Default 0.5.

Control [UMFPACK_STRATEGY_THRESH_NNZDIAG]: tdiag, Default 0.9.

For the auto strategy, if the pattern of the submatrix S after removing singletons has a symmetry of tsym or more (0 being completely unsymmetric and 1 being completely symmetric, and if the fraction of entries present on the diagonal is \geq tdiag, then the symmetric strategy is chosen. Otherwise, the unsymmetric strategy is chosen.

double Info [UMFPACK_INFO] ; Output argument, not defined on input.

Contains statistics about the symbolic analysis. If a (double *) NULL pointer is passed, then no statistics are returned in Info (this is not an error condition). The entire Info array is cleared (all entries set to -1) and then the following statistics are computed:

Info [UMFPACK_STATUS]: status code. This is also the return value, whether or not Info is present.

UMFPACK_OK

Each column of the input matrix contained row indices in increasing order, with no duplicates. Only in this case does umfpack*_symbolic compute a valid symbolic factorization. For the other cases below, no Symbolic object is created (*Symbolic is (void *) NULL).

UMFPACK_ERROR_n_nonpositive

n is less than or equal to zero.

UMFPACK_ERROR_invalid_matrix

Number of entries in the matrix is negative, Ap [0] is nonzero, a column has a negative number of entries, a row index is out of bounds, or the columns of input matrix were jumbled (unsorted columns or duplicate entries).

UMFPACK_ERROR_out_of_memory

Insufficient memory to perform the symbolic analysis. If the analysis requires more than 2GB of memory and you are using the int32_t version of UMFPACK, then you are guaranteed to run out of memory. Try using the 64-bit version of UMFPACK.

UMFPACK_ERROR_argument_missing

One or more required arguments is missing.

UMFPACK_ERROR_internal_error

Something very serious went wrong. This is a bug.
Please contact the author (DrTimothyAldenDavis@gmail.com).

Info [UMFPACK_NROW]: the value of the input argument n_row.

Info [UMFPACK_NCOL]: the value of the input argument n_col.

Info [UMFPACK_NZ]: the number of entries in the input matrix
(Ap [n_col]).

Info [UMFPACK_SIZE_OF_UNIT]: the number of bytes in a Unit,
for memory usage statistics below.

Info [UMFPACK_SIZE_OF_INT]: the number of bytes in an int32_t.

Info [UMFPACK_SIZE_OF_LONG]: the number of bytes in a int64_t.

Info [UMFPACK_SIZE_OF_POINTER]: the number of bytes in a void *
pointer.

Info [UMFPACK_SIZE_OF_ENTRY]: the number of bytes in a numerical entry.

Info [UMFPACK_NDENSE_ROW]: number of "dense" rows in A. These rows are
ignored when the column pre-ordering is computed in COLAMD. They
are also treated differently during numeric factorization. If > 0,
then the matrix had to be re-analyzed by UMF_analyze, which does
not ignore these rows.

Info [UMFPACK_NEMPTY_ROW]: number of "empty" rows in A, as determined
These are rows that either have no entries, or whose entries are
all in pivot columns of zero-Markowitz-cost pivots.

Info [UMFPACK_NDENSE_COL]: number of "dense" columns in A. COLAMD
orders these columns are ordered last in the factorization, but
before "empty" columns.

Info [UMFPACK_NEMPTY_COL]: number of "empty" columns in A. These are
columns that either have no entries, or whose entries are all in
pivot rows of zero-Markowitz-cost pivots. These columns are
ordered last in the factorization, to the right of "dense" columns.

Info [UMFPACK_SYMBOLIC_DEFRAG]: number of garbage collections
performed during ordering and symbolic pre-analysis.

Info [UMFPACK_SYMBOLIC_PEAK_MEMORY]: the amount of memory (in Units)
required for umfpack*_symbolic to complete. This count includes
the size of the Symbolic object itself, which is also reported in
Info [UMFPACK_SYMBOLIC_SIZE].

Info [UMFPACK_SYMBOLIC_SIZE]: the final size of the Symbolic object (in Units). This is fairly small, roughly $2*n$ to $13*n$ integers, depending on the matrix.

Info [UMFPACK_VARIABLE_INIT_ESTIMATE]: the Numeric object contains two parts. The first is fixed in size (0 ($n_{row}+n_{col}$)). The second part holds the sparse LU factors and the contribution blocks from factorized frontal matrices. This part changes in size during factorization. Info [UMFPACK_VARIABLE_INIT_ESTIMATE] is the exact size (in Units) required for this second variable-sized part in order for the numerical factorization to start.

Info [UMFPACK_VARIABLE_PEAK_ESTIMATE]: the estimated peak size (in Units) of the variable-sized part of the Numeric object. This is usually an upper bound, but that is not guaranteed.

Info [UMFPACK_VARIABLE_FINAL_ESTIMATE]: the estimated final size (in Units) of the variable-sized part of the Numeric object. This is usually an upper bound, but that is not guaranteed. It holds just the sparse LU factors.

Info [UMFPACK_NUMERIC_SIZE_ESTIMATE]: an estimate of the final size (in Units) of the entire Numeric object (both fixed-size and variable-sized parts), which holds the LU factorization (including the L, U, P and Q matrices).

Info [UMFPACK_PEAK_MEMORY_ESTIMATE]: an estimate of the total amount of memory (in Units) required by `umfpack*_symbolic` and `umfpack*_numeric` to perform both the symbolic and numeric factorization. This is the larger of the amount of memory needed in `umfpack*_numeric` itself, and the amount of memory needed in `umfpack*_symbolic` (Info [UMFPACK_SYMBOLIC_PEAK_MEMORY]). The count includes the size of both the Symbolic and Numeric objects themselves. It can be a very loose upper bound, particularly when the symmetric strategy is used.

Info [UMFPACK_FLOPS_ESTIMATE]: an estimate of the total floating-point operations required to factorize the matrix. This is a "true" theoretical estimate of the number of flops that would be performed by a flop-parsimonious sparse LU algorithm. It assumes that no extra flops are performed except for what is strictly required to compute the LU factorization. It ignores, for example, the flops performed by `umfpack_di_numeric` to add contribution blocks of frontal matrices together. If L and U are the upper bound on the pattern of the factors, then this flop count estimate can be represented in MATLAB (for real matrices, not complex) as:

```
Lnz = full (sum (spones (L))) - 1 ;      % nz in each col of L
Unz = full (sum (spones (U')))' - 1 ;    % nz in each row of U
flops = 2*Lnz*Unz + sum (Lnz) ;
```

The actual "true flop" count found by `umfpack*_numeric` will be less than this estimate.

For the real version, only (+ - * /) are counted. For the complex version, the following counts are used:

operation	flops
c = 1/b	6
c = a*b	6
c -= a*b	8

Info [UMFPACK_LNZ_ESTIMATE]: an estimate of the number of nonzeros in L, including the diagonal. Since L is unit-diagonal, the diagonal of L is not stored. This estimate is a strict upper bound on the actual nonzeros in L to be computed by umfpack*_numeric.

Info [UMFPACK_UNZ_ESTIMATE]: an estimate of the number of nonzeros in U, including the diagonal. This estimate is a strict upper bound on the actual nonzeros in U to be computed by umfpack*_numeric.

Info [UMFPACK_MAX_FRONT_SIZE_ESTIMATE]: estimate of the size of the largest frontal matrix (# of entries), for arbitrary partial pivoting during numerical factorization.

Info [UMFPACK_SYMBOLIC_TIME]: The CPU time taken, in seconds.

Info [UMFPACK_SYMBOLIC_WALLTIME]: The wallclock time taken, in seconds.

Info [UMFPACK_STRATEGY_USED]: The ordering strategy used:
UMFPACK_STRATEGY_SYMMETRIC or UMFPACK_STRATEGY_UNSYMMETRIC

Info [UMFPACK_ORDERING_USED]: The ordering method used:
UMFPACK_ORDERING_AMD (AMD for sym. strategy, COLAMD for unsym.)
UMFPACK_ORDERING_GIVEN
UMFPACK_ORDERING_NONE
UMFPACK_ORDERING_METIS
UMFPACK_ORDERING_USER

Info [UMFPACK_QFIXED]: 1 if the column pre-ordering will be refined during numerical factorization, 0 if not.

Info [UMFPACK_DIAG_PREFERED]: 1 if diagonal pivoting will be attempted, 0 if not.

Info [UMFPACK_COL_SINGLETONS]: the matrix A is analyzed by first eliminating all pivots with zero Markowitz cost. This count is the number of these pivots with exactly one nonzero in their pivot column.

Info [UMFPACK_ROW_SINGLETONS]: the number of zero-Markowitz-cost pivots with exactly one nonzero in their pivot row.

Info [UMFPACK_PATTERN_SYMMETRY]: the symmetry of the pattern of S.

Info [UMFPACK_NZ_A_PLUS_AT]: the number of off-diagonal entries in S+S'.

Info [UMFPACK_NZDIAG]: the number of entries on the diagonal of S.

Info [UMFPACK_N2]: if S is square, and nempty_row = nempty_col, this is equal to n_row - n1 - nempty_row.

Info [UMFPACK_S_SYMMETRIC]: 1 if S is square and its diagonal has been preserved, 0 otherwise.

Info [UMFPACK_MAX_FRONT_NROWS_ESTIMATE]: estimate of the max number of rows in any frontal matrix, for arbitrary partial pivoting.

Info [UMFPACK_MAX_FRONT_NCOLS_ESTIMATE]: estimate of the max number of columns in any frontal matrix, for arbitrary partial pivoting.

The next four statistics are computed only if AMD is used:

Info [UMFPACK_SYMMETRIC_LUNZ]: The number of nonzeros in L and U, assuming no pivoting during numerical factorization, and assuming a zero-free diagonal of U. Excludes the entries on the diagonal of L. If the matrix has a purely symmetric nonzero pattern, this is often a lower bound on the nonzeros in the actual L and U computed in the numerical factorization, for matrices that fit the criteria for the "symmetric" strategy.

Info [UMFPACK_SYMMETRIC_FLOPS]: The floating-point operation count in the numerical factorization phase, assuming no pivoting. If the pattern of the matrix is symmetric, this is normally a lower bound on the floating-point operation count in the actual numerical factorization, for matrices that fit the criteria for the symmetric strategy.

Info [UMFPACK_SYMMETRIC_NDENSE]: The number of "dense" rows/columns of S+S' that were ignored during the AMD ordering. These are placed last in the output order. If > 0, then the Info [UMFPACK_SYMMETRIC_*] statistics, above are rough upper bounds.

Info [UMFPACK_SYMMETRIC_DMAX]: The maximum number of nonzeros in any column of L, if no pivoting is performed during numerical factorization. Excludes the part of the LU factorization for pivots with zero Markowitz cost.

At the start of umfpack*_symbolic, all of Info is set of -1, and then after that only the above listed Info [...] entries are accessed. Future versions might modify different parts of Info.

*/

5.2 umfpack_di_numeric

```
int umfpack_di_numeric
(
    const int32_t Ap [ ],
    const int32_t Ai [ ],
    const double Ax [ ],
    void *Symbolic,
    void **Numeric,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

int umfpack_dl_numeric
(
    const int64_t Ap [ ],
    const int64_t Ai [ ],
    const double Ax [ ],
    void *Symbolic,
    void **Numeric,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

int umfpack_zi_numeric
(
    const int32_t Ap [ ],
    const int32_t Ai [ ],
    const double Ax [ ], const double Az [ ],
    void *Symbolic,
    void **Numeric,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

int umfpack_zl_numeric
(
    const int64_t Ap [ ],
    const int64_t Ai [ ],
    const double Ax [ ], const double Az [ ],
    void *Symbolic,
    void **Numeric,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

/*
double int32_t Syntax:

#include "umfpack.h"
void *Symbolic, *Numeric ;
int32_t *Ap, *Ai, status ;
double *Ax, Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
int status = umfpack_di_numeric (Ap, Ai, Ax, Symbolic, &Numeric, Control,
    Info) ;
```

double int64_t Syntax:

```
#include "umfpack.h"
void *Symbolic, *Numeric ;
int64_t *Ap, *Ai ;
double *Ax, Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
int status = umfpack_dl_numeric (Ap, Ai, Ax, Symbolic, &Numeric, Control,
    Info) ;
```

complex int32_t Syntax:

```
#include "umfpack.h"
void *Symbolic, *Numeric ;
int32_t *Ap, *Ai ;
double *Ax, *Az, Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
int status = umfpack_zi_numeric (Ap, Ai, Ax, Az, Symbolic, &Numeric,
    Control, Info) ;
```

complex int64_t Syntax:

```
#include "umfpack.h"
void *Symbolic, *Numeric ;
int64_t *Ap, *Ai ;
double *Ax, *Az, Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
int status = umfpack_zl_numeric (Ap, Ai, Ax, Az, Symbolic, &Numeric,
    Control, Info) ;
```

packed complex Syntax:

Same as above, except that Az is NULL.

Purpose:

Given a sparse matrix A in column-oriented form, and a symbolic analysis computed by `umfpack_*_symbolic`, the `umfpack_*_numeric` routine performs the numerical factorization, PAQ=LU, PRAQ=LU, or P(R\A)Q=LU, where P and Q are permutation matrices (represented as permutation vectors), R is the row scaling, L is unit-lower triangular, and U is upper triangular. This is required before the system Ax=b (or other related linear systems) can be solved. `umfpack_*_numeric` can be called multiple times for each call to `umfpack_*_symbolic`, to factorize a sequence of matrices with identical nonzero pattern. Simply compute the Symbolic object once, with `umfpack_*_symbolic`, and reuse it for subsequent matrices. This routine safely detects if the pattern changes, and sets an appropriate error code.

Returns:

The status code is returned. See Info [UMFPACK_STATUS], below.

Arguments:

Int Ap [n_col+1] ; Input argument, not modified.

This must be identical to the Ap array passed to `umfpack_*_symbolic`.

The value of `n_col` is what was passed to `umfpack_*_symbolic` (this is held in the Symbolic object).

`Int Ai [nz] ;` Input argument, not modified, of size `nz = Ap [n_col]`.

This must be identical to the `Ai` array passed to `umfpack_*_symbolic`.

`double Ax [nz] ;` Input argument, not modified, of size `nz = Ap [n_col]`.
Size `2*nz` for packed complex case.

The numerical values of the sparse matrix `A`. The nonzero pattern (row indices) for column `j` is stored in `Ai [(Ap [j]) ... (Ap [j+1]-1)]`, and the corresponding numerical values are stored in `Ax [(Ap [j]) ... (Ap [j+1]-1)]`.

`double Az [nz] ;` Input argument, not modified, for complex versions.

For the complex versions, this holds the imaginary part of `A`. The imaginary part of column `j` is held in `Az [(Ap [j]) ... (Ap [j+1]-1)]`.

If `Az` is `NULL`, then both real and imaginary parts are contained in `Ax[0..2*nz-1]`, with `Ax[2*k]` and `Ax[2*k+1]` being the real and imaginary part of the `k`th entry.

`void *Symbolic ;` Input argument, not modified.

The Symbolic object, which holds the symbolic factorization computed by `umfpack_*_symbolic`. The Symbolic object is not modified by `umfpack_*_numeric`.

`void **Numeric ;` Output argument.

`**Numeric` is the address of a `(void *)` pointer variable in the user's calling routine (see Syntax, above). On input, the contents of this variable are not defined. On output, this variable holds a `(void *)` pointer to the Numeric object (if successful), or `(void *) NULL` if a failure occurred.

`double Control [UMFPACK_CONTROL] ;` Input argument, not modified.

If a `(double *) NULL` pointer is passed, then the default control settings are used. Otherwise, the settings are determined from the Control array. See `umfpack_*_defaults` on how to fill the Control array with the default settings. If Control contains NaN's, the defaults are used. The following Control parameters are used:

`Control [UMFPACK_PIVOT_TOLERANCE]`: relative pivot tolerance for threshold partial pivoting with row interchanges. In any given column, an entry is numerically acceptable if its absolute value is greater than or equal to `Control [UMFPACK_PIVOT_TOLERANCE]` times the largest absolute value in the column. A value of 1.0 gives true partial pivoting. If less than or equal to zero, then any nonzero entry is numerically acceptable as a pivot. Default: 0.1.

Smaller values tend to lead to sparser LU factors, but the solution

to the linear system can become inaccurate. Larger values can lead to a more accurate solution (but not always), and usually an increase in the total work.

For complex matrices, a cheap approximate of the absolute value is used for the threshold partial pivoting test ($|a_{\text{real}}| + |a_{\text{imag}}|$ instead of the more expensive-to-compute exact absolute value $\sqrt{a_{\text{real}}^2 + a_{\text{imag}}^2}$).

Control [UMFPACK_SYM_PIVOT_TOLERANCE]:

If diagonal pivoting is attempted (the symmetric strategy is used) then this parameter is used to control when the diagonal entry is selected in a given pivot column. The absolute value of the entry must be \geq Control [UMFPACK_SYM_PIVOT_TOLERANCE] times the largest absolute value in the column. A value of zero will ensure that no off-diagonal pivoting is performed, except that zero diagonal entries are not selected if there are any off-diagonal nonzero entries.

If an off-diagonal pivot is selected, an attempt is made to restore symmetry later on. Suppose $A(i,j)$ is selected, where $i \neq j$. If column i has not yet been selected as a pivot column, then the entry $A(j,i)$ is redefined as a "diagonal" entry, except that the tighter tolerance (Control [UMFPACK_PIVOT_TOLERANCE]) is applied. This strategy has an effect similar to 2-by-2 pivoting for symmetric indefinite matrices. If a 2-by-2 block pivot with nonzero structure

$$\begin{array}{cc} & i & j \\ i: & 0 & x \\ j: & x & 0 \end{array}$$

is selected in a symmetric indefinite factorization method, the 2-by-2 block is inverted and a rank-2 update is applied. In UMFPACK, this 2-by-2 block would be reordered as

$$\begin{array}{cc} & j & i \\ i: & x & 0 \\ j: & 0 & x \end{array}$$

In both cases, the symmetry of the Schur complement is preserved.

Control [UMFPACK_SCALE]: Note that the user's input matrix is never modified, only an internal copy is scaled.

There are three valid settings for this parameter. If any other value is provided, the default is used.

UMFPACK_SCALE_NONE: no scaling is performed.

UMFPACK_SCALE_SUM: each row of the input matrix A is divided by the sum of the absolute values of the entries in that row. The scaled matrix has an infinity norm of 1.

UMFPACK_SCALE_MAX: each row of the input matrix A is divided by

the maximum the absolute values of the entries in that row.
In the scaled matrix the largest entry in each row has
a magnitude exactly equal to 1.

Note that for complex matrices, a cheap approximate absolute value
is used, $|a_{\text{real}}| + |a_{\text{imag}}|$, instead of the exact absolute value
 $\sqrt{(a_{\text{real}})^2 + (a_{\text{imag}})^2}$.

Scaling is very important for the "symmetric" strategy when
diagonal pivoting is attempted. It also improves the performance
of the "unsymmetric" strategy.

Default: UMFPACK_SCALE_SUM.

Control [UMFPACK_ALLOC_INIT]:

When `umfpack*_numeric` starts, it allocates memory for the Numeric
object. Part of this is of fixed size (approximately n double's +
 $12*n$ integers). The remainder is of variable size, which grows to
hold the LU factors and the frontal matrices created during
factorization. A estimate of the upper bound is computed by
`umfpack*_symbolic`, and returned by `umfpack*_symbolic` in
Info [UMFPACK_VARIABLE_PEAK_ESTIMATE] (in Units).

If Control [UMFPACK_ALLOC_INIT] is ≥ 0 , `umfpack*_numeric` initially
allocates space for the variable-sized part equal to this estimate
times Control [UMFPACK_ALLOC_INIT]. Typically, for matrices for
which the "unsymmetric" strategy applies, `umfpack*_numeric` needs
only about half the estimated memory space, so a setting of 0.5 or
0.6 often provides enough memory for `umfpack*_numeric` to factorize
the matrix with no subsequent increases in the size of this block.

If the matrix is ordered via AMD, then this non-negative parameter
is ignored. The initial allocation ratio computed automatically,
as $1.2 * (nz + \text{Info}[\text{UMFPACK_SYMMETRIC_LUNZ}]) /$
 $(\text{Info}[\text{UMFPACK_LNZ_ESTIMATE}] + \text{Info}[\text{UMFPACK_UNZ_ESTIMATE}] -$
 $\min(n_{\text{row}}, n_{\text{col}}))$.

If Control [UMFPACK_ALLOC_INIT] is negative, then `umfpack*_numeric`
allocates a space with initial size (in Units) equal to
 $(-\text{Control}[\text{UMFPACK_ALLOC_INIT}])$.

Regardless of the value of this parameter, a space equal to or
greater than the the bare minimum amount of memory needed to start
the factorization is always initially allocated. The bare initial
memory required is returned by `umfpack*_symbolic` in
Info [UMFPACK_VARIABLE_INIT_ESTIMATE] (an exact value, not an
estimate).

If the variable-size part of the Numeric object is found to be too
small sometime after numerical factorization has started, the memory
is increased in size by a factor of 1.2. If this fails, the
request is reduced by a factor of 0.95 until it succeeds, or until
it determines that no increase in size is possible. Garbage
collection then occurs.

The strategy of attempting to "malloc" a working space, and re-trying with a smaller space, may not work when UMFPACK is used as a mexFunction in MATLAB, since mxMalloc aborts the mexFunction if it fails. This issue does not affect the use of UMFPACK as a part of the built-in $x=A\b$ in MATLAB 6.5 and later.

If you are using the umfpack mexFunction, decrease the magnitude of Control [UMFPACK_ALLOC_INIT] if you run out of memory in MATLAB.

Default initial allocation size: 0.7. Thus, with the default control settings and the "unsymmetric" strategy, the upper-bound is reached after two reallocations ($0.7 * 1.2 * 1.2 = 1.008$).

Changing this parameter has little effect on fill-in or operation count. It has a small impact on run-time (the extra time required to do the garbage collection and memory reallocation).

Control [UMFPACK_FRONT_ALLOC_INIT]:

When UMFPACK starts the factorization of each "chain" of frontal matrices, it allocates a working array to hold the frontal matrices as they are factorized. The symbolic factorization computes the size of the largest possible frontal matrix that could occur during the factorization of each chain.

If Control [UMFPACK_FRONT_ALLOC_INIT] is ≥ 0 , the following strategy is used. If the AMD ordering was used, this non-negative parameter is ignored. A front of size $(d+2)*(d+2)$ is allocated, where $d = \text{Info [UMFPACK_SYMMETRIC_DMAX]}$. Otherwise, a front of size Control [UMFPACK_FRONT_ALLOC_INIT] times the largest front possible for this chain is allocated.

If Control [UMFPACK_FRONT_ALLOC_INIT] is negative, then a front of size $(-\text{Control [UMFPACK_FRONT_ALLOC_INIT]})$ is allocated (where the size is in terms of the number of numerical entries). This is done regardless of the ordering method or ordering strategy used.

Default: 0.5.

Control [UMFPACK_DROPTOL]:

Entries in L and U with absolute value less than or equal to the drop tolerance are removed from the data structures (unless leaving them there reduces memory usage by reducing the space required for the nonzero pattern of L and U).

Default: 0.0.

double Info [UMFPACK_INFO] ; Output argument.

Contains statistics about the numeric factorization. If a (double *) NULL pointer is passed, then no statistics are returned in Info (this is not an error condition). The following statistics are computed in umfpack_*_numeric:

Info [UMFPACK_STATUS]: status code. This is also the return value, whether or not Info is present.

UMFPACK_OK

Numeric factorization was successful. `umfpack*_numeric` computed a valid numeric factorization.

UMFPACK_WARNING_singular_matrix

Numeric factorization was successful, but the matrix is singular. `umfpack*_numeric` computed a valid numeric factorization, but you will get a divide by zero in `umfpack*_solve`. For the other cases below, no Numeric object is created (*Numeric is (void *) NULL).

UMFPACK_ERROR_out_of_memory

Insufficient memory to complete the numeric factorization.

UMFPACK_ERROR_argument_missing

One or more required arguments are missing.

UMFPACK_ERROR_invalid_Symbolic_object

Symbolic object provided as input is invalid.

UMFPACK_ERROR_different_pattern

The pattern (A_p and/or A_i) has changed since the call to `umfpack*_symbolic` which produced the Symbolic object.

Info [UMFPACK_NROW]: the value of `n_row` stored in the Symbolic object.

Info [UMFPACK_NCOL]: the value of `n_col` stored in the Symbolic object.

Info [UMFPACK_NZ]: the number of entries in the input matrix. This value is obtained from the Symbolic object.

Info [UMFPACK_SIZE_OF_UNIT]: the number of bytes in a Unit, for memory usage statistics below.

Info [UMFPACK_VARIABLE_INIT]: the initial size (in Units) of the variable-sized part of the Numeric object. If this differs from Info [UMFPACK_VARIABLE_INIT_ESTIMATE], then the pattern (A_p and/or A_i) has changed since the last call to `umfpack*_symbolic`, which is an error condition.

Info [UMFPACK_VARIABLE_PEAK]: the peak size (in Units) of the variable-sized part of the Numeric object. This size is the amount of space actually used inside the block of memory, not the space allocated via `UMF_malloc`. You can reduce UMFPACK's memory requirements by setting Control [UMFPACK_ALLOC_INIT] to the ratio

Info [UMFPACK_VARIABLE_PEAK] / Info[UMFPACK_VARIABLE_PEAK_ESTIMATE]. This will ensure that no memory reallocations occur (you may want to add 0.001 to make sure that integer roundoff does not lead to a memory size that is 1 Unit too small; otherwise, garbage collection and reallocation will occur).

Info [UMFPACK_VARIABLE_FINAL]: the final size (in Units) of the variable-sized part of the Numeric object. It holds just the sparse LU factors.

Info [UMFPACK_NUMERIC_SIZE]: the actual final size (in Units) of the entire Numeric object, including the final size of the variable part of the object. Info [UMFPACK_NUMERIC_SIZE_ESTIMATE], an estimate, was computed by umfpack_*_symbolic. The estimate is normally an upper bound on the actual final size, but this is not guaranteed.

Info [UMFPACK_PEAK_MEMORY]: the actual peak memory usage (in Units) of both umfpack_*_symbolic and umfpack_*_numeric. An estimate, Info [UMFPACK_PEAK_MEMORY_ESTIMATE], was computed by umfpack_*_symbolic. The estimate is normally an upper bound on the actual peak usage, but this is not guaranteed. With testing on hundreds of matrix arising in real applications, I have never observed a matrix where this estimate or the Numeric size estimate was less than the actual result, but this is theoretically possible. Please send me one if you find such a matrix.

Info [UMFPACK_FLOPS]: the actual count of the (useful) floating-point operations performed. An estimate, Info [UMFPACK_FLOPS_ESTIMATE], was computed by umfpack_*_symbolic. The estimate is guaranteed to be an upper bound on this flop count. The flop count excludes "useless" flops on zero values, flops performed during the pivot search (for tentative updates and assembly of candidate columns), and flops performed to add frontal matrices together.

For the real version, only (+ - * /) are counted. For the complex version, the following counts are used:

operation	flops
c = 1/b	6
c = a*b	6
c -= a*b	8

Info [UMFPACK_LNZ]: the actual nonzero entries in final factor L, including the diagonal. This excludes any zero entries in L, although some of these are stored in the Numeric object. The Info [UMFPACK_LU_ENTRIES] statistic does account for all explicitly stored zeros, however. Info [UMFPACK_LNZ_ESTIMATE], an estimate, was computed by umfpack_*_symbolic. The estimate is guaranteed to be an upper bound on Info [UMFPACK_LNZ].

Info [UMFPACK_UNZ]: the actual nonzero entries in final factor U, including the diagonal. This excludes any zero entries in U, although some of these are stored in the Numeric object. The Info [UMFPACK_LU_ENTRIES] statistic does account for all

explicitly stored zeros, however. Info [UMFPACK_UNZ_ESTIMATE], an estimate, was computed by `umfpack_*_symbolic`. The estimate is guaranteed to be an upper bound on Info [UMFPACK_UNZ].

Info [UMFPACK_NUMERIC_DEFRAG]: The number of garbage collections performed during `umfpack_*_numeric`, to compact the contents of the variable-sized workspace used by `umfpack_*_numeric`. No estimate was computed by `umfpack_*_symbolic`. In the current version of UMFPACK, garbage collection is performed and then the memory is reallocated, so this statistic is the same as Info [UMFPACK_NUMERIC_REALLOC], below. It may differ in future releases.

Info [UMFPACK_NUMERIC_REALLOC]: The number of times that the Numeric object was increased in size from its initial size. A rough upper bound on the peak size of the Numeric object was computed by `umfpack_*_symbolic`, so reallocations should be rare. However, if `umfpack_*_numeric` is unable to allocate that much storage, it reduces its request until either the allocation succeeds, or until it gets too small to do anything with. If the memory that it finally got was small, but usable, then the reallocation count could be high. No estimate of this count was computed by `umfpack_*_symbolic`.

Info [UMFPACK_NUMERIC_COSTLY_REALLOC]: The number of times that the system `realloc` library routine (or `mxRealloc` for the `mexFunction`) had to move the workspace. `Realloc` can sometimes increase the size of a block of memory without moving it, which is much faster. This statistic will always be \leq Info [UMFPACK_NUMERIC_REALLOC]. If your memory space is fragmented, then the number of "costly" `realloc`'s will be equal to Info [UMFPACK_NUMERIC_REALLOC].

Info [UMFPACK_COMPRESSED_PATTERN]: The number of integers used to represent the pattern of L and U.

Info [UMFPACK_LU_ENTRIES]: The total number of numerical values that are stored for the LU factors. Some of the values may be explicitly zero in order to save space (allowing for a smaller compressed pattern).

Info [UMFPACK_NUMERIC_TIME]: The CPU time taken, in seconds.

Info [UMFPACK_RCOND]: A rough estimate of the condition number, equal to $\min(\text{abs}(\text{diag}(U))) / \max(\text{abs}(\text{diag}(U)))$, or zero if the diagonal of U is all zero.

Info [UMFPACK_UDIAG_NZ]: The number of numerically nonzero values on the diagonal of U.

Info [UMFPACK_UMIN]: the smallest absolute value on the diagonal of U.

Info [UMFPACK_UMAX]: the smallest absolute value on the diagonal of U.

Info [UMFPACK_MAX_FRONT_SIZE]: the size of the largest frontal matrix (number of entries).

Info [UMFPACK_NUMERIC_WALLTIME]: The wallclock time taken, in seconds.

Info [UMFPACK_MAX_FRONT_NROWS]: the max number of rows in any frontal matrix.

Info [UMFPACK_MAX_FRONT_NCOLS]: the max number of columns in any frontal matrix.

Info [UMFPACK_WAS_SCALED]: the scaling used, either UMFPACK_SCALE_NONE, UMFPACK_SCALE_SUM, or UMFPACK_SCALE_MAX.

Info [UMFPACK_RSMIN]: if scaling is performed, the smallest scale factor for any row (either the smallest sum of absolute entries, or the smallest maximum of absolute entries).

Info [UMFPACK_RSMAX]: if scaling is performed, the largest scale factor for any row (either the largest sum of absolute entries, or the largest maximum of absolute entries).

Info [UMFPACK_ALLOC_INIT_USED]: the initial allocation parameter used.

Info [UMFPACK_FORCED_UPDATES]: the number of BLAS-3 updates to the frontal matrices that were required because the frontal matrix grew larger than its current working array.

Info [UMFPACK_NOFF_DIAG]: number of off-diagonal pivots selected, if the symmetric strategy is used.

Info [UMFPACK_NZDROPPED]: the number of entries smaller in absolute value than Control [UMFPACK_DROPTOL] that were dropped from L and U. Note that entries on the diagonal of U are never dropped.

Info [UMFPACK_ALL_LNZ]: the number of entries in L, including the diagonal, if no small entries are dropped.

Info [UMFPACK_ALL_UNZ]: the number of entries in U, including the diagonal, if no small entries are dropped.

Only the above listed Info [...] entries are accessed. The remaining entries of Info are not accessed or modified by umfpack*_numeric. Future versions might modify different parts of Info.

*/

5.3 umfpack_di_solve

```
int umfpack_di_solve
(
    int sys,
    const int32_t Ap [ ],
    const int32_t Ai [ ],
    const double Ax [ ],
    double X [ ],
    const double B [ ],
    void *Numeric,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
);

int umfpack_dl_solve
(
    int sys,
    const int64_t Ap [ ],
    const int64_t Ai [ ],
    const double Ax [ ],
    double X [ ],
    const double B [ ],
    void *Numeric,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
);

int umfpack_zi_solve
(
    int sys,
    const int32_t Ap [ ],
    const int32_t Ai [ ],
    const double Ax [ ], const double Az [ ],
    double Xx [ ],      double Xz [ ],
    const double Bx [ ], const double Bz [ ],
    void *Numeric,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
);

int umfpack_zl_solve
(
    int sys,
    const int64_t Ap [ ],
    const int64_t Ai [ ],
    const double Ax [ ], const double Az [ ],
    double Xx [ ],      double Xz [ ],
    const double Bx [ ], const double Bz [ ],
    void *Numeric,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
);

/*
```

double int32_t Syntax:

```
#include "umfpack.h"
void *Numeric ;
int32_t *Ap, *Ai ;
int sys ;
double *B, *X, *Ax, Info [UMFPACK_INFO], Control [UMFPACK_CONTROL] ;
int status = umfpack_di_solve (sys, Ap, Ai, Ax, X, B, Numeric, Control,
    Info) ;
```

double int64_t Syntax:

```
#include "umfpack.h"
void *Numeric ;
int64_t *Ap, *Ai ;
int sys ;
double *B, *X, *Ax, Info [UMFPACK_INFO], Control [UMFPACK_CONTROL] ;
int status = umfpack_dl_solve (sys, Ap, Ai, Ax, X, B, Numeric, Control,
    Info) ;
```

complex int32_t Syntax:

```
#include "umfpack.h"
void *Numeric ;
int32_t *Ap, *Ai ;
int sys ;
double *Bx, *Bz, *Xx, *Xz, *Ax, *Az, Info [UMFPACK_INFO],
    Control [UMFPACK_CONTROL] ;
int status = umfpack_zi_solve (sys, Ap, Ai, Ax, Az, Xx, Xz, Bx, Bz,
    Numeric, Control, Info) ;
```

complex int64_t Syntax:

```
#include "umfpack.h"
void *Numeric ;
int64_t *Ap, *Ai ;
int sys ;
double *Bx, *Bz, *Xx, *Xz, *Ax, *Az, Info [UMFPACK_INFO],
    Control [UMFPACK_CONTROL] ;
int status = umfpack_zl_solve (sys, Ap, Ai, Ax, Az, Xx, Xz, Bx, Bz,
    Numeric, Control, Info) ;
```

packed complex Syntax:

Same as above, Xz, Bz, and Az are NULL.

Purpose:

Given LU factors computed by umfpack_*_numeric (PAQ=LU, PRAQ=LU, or P(R\A)Q=LU) and the right-hand-side, B, solve a linear system for the solution X. Iterative refinement is optionally performed. Only square systems are handled. Singular matrices result in a divide-by-zero for all systems except those involving just the matrix L. Iterative refinement is not performed for singular matrices. In the discussion below, n is equal to n_row and n_col, because only square systems are handled.

Returns:

The status code is returned. See Info [UMFPACK_STATUS], below.

Arguments:

int sys ; Input argument, not modified.

Defines which system to solve. (') is the linear algebraic transpose (complex conjugate if A is complex), and (.) is the array transpose.

sys value	system solved
UMFPACK_A	Ax=b
UMFPACK_At	A'x=b
UMFPACK_Aat	A.'x=b
UMFPACK_Pt_L	P'Lx=b
UMFPACK_L	Lx=b
UMFPACK_Lt_P	L'Px=b
UMFPACK_Lat_P	L.'Px=b
UMFPACK_Lt	L'x=b
UMFPACK_U_Qt	UQ'x=b
UMFPACK_U	Ux=b
UMFPACK_Q_Ut	QU'x=b
UMFPACK_Q_Uat	QU.'x=b
UMFPACK_Ut	U'x=b
UMFPACK_Uat	U.'x=b

Iterative refinement can be optionally performed when sys is any of the following:

UMFPACK_A	Ax=b
UMFPACK_At	A'x=b
UMFPACK_Aat	A.'x=b

For the other values of the sys argument, iterative refinement is not performed (Control [UMFPACK_IRSTEP], Ap, Ai, Ax, and Az are ignored).

Int Ap [n+1] ; Input argument, not modified.
Int Ai [nz] ; Input argument, not modified.
double Ax [nz] ; Input argument, not modified.
 Size 2*nz for packed complex case.
double Az [nz] ; Input argument, not modified, for complex versions.

If iterative refinement is requested (Control [UMFPACK_IRSTEP] >= 1, Ax=b, A'x=b, or A.'x=b is being solved, and A is nonsingular), then these arrays must be identical to the same ones passed to umfpack*_numeric. The umfpack*_solve routine does not check the contents of these arguments, so the results are undefined if Ap, Ai, Ax, and/or Az are modified between the calls the umfpack*_numeric and umfpack*_solve. These three arrays do not need to be present (NULL pointers can be passed) if Control [UMFPACK_IRSTEP] is zero, or if a system other than Ax=b, A'x=b, or A.'x=b is being solved, or if A is singular, since in each of these cases A is not accessed.

If Az, Xz, or Bz are NULL, then both real and imaginary parts are contained in Ax[0..2*nz-1], with Ax[2*k] and Ax[2*k+1] being the real and imaginary part of the kth entry.

double X [n] ; Output argument.
or:
double Xx [n] ; Output argument, real part
 Size 2*n for packed complex case.
double Xz [n] ; Output argument, imaginary part.

The solution to the linear system, where $n = n_row = n_col$ is the dimension of the matrices A, L, and U.

If Az, Xz, or Bz are NULL, then both real and imaginary parts are returned in Xx[0..2*n-1], with Xx[2*k] and Xx[2*k+1] being the real and imaginary part of the kth entry.

double B [n] ; Input argument, not modified.
or:
double Bx [n] ; Input argument, not modified, real part.
 Size 2*n for packed complex case.
double Bz [n] ; Input argument, not modified, imaginary part.

The right-hand side vector, b, stored as a conventional array of size n (or two arrays of size n for complex versions). This routine does not solve for multiple right-hand-sides, nor does it allow b to be stored in a sparse-column form.

If Az, Xz, or Bz are NULL, then both real and imaginary parts are contained in Bx[0..2*n-1], with Bx[2*k] and Bx[2*k+1] being the real and imaginary part of the kth entry.

void *Numeric ; Input argument, not modified.

Numeric must point to a valid Numeric object, computed by umfpack*_numeric.

double Control [UMFPACK_CONTROL] ; Input argument, not modified.

If a (double *) NULL pointer is passed, then the default control settings are used. Otherwise, the settings are determined from the Control array. See umfpack*_defaults on how to fill the Control array with the default settings. If Control contains NaN's, the defaults are used. The following Control parameters are used:

Control [UMFPACK_IRSTEP]: The maximum number of iterative refinement steps to attempt. A value less than zero is treated as zero. If less than 1, or if Ax=b, A'x=b, or A.'x=b is not being solved, or if A is singular, then the Ap, Ai, Ax, and Az arguments are not accessed. Default: 2.

double Info [UMFPACK_INFO] ; Output argument.

Contains statistics about the solution factorization. If a (double *) NULL pointer is passed, then no statistics are returned in

Info (this is not an error condition). The following statistics are computed in umfpack*_solve:

Info [UMFPACK_STATUS]: status code. This is also the return value, whether or not Info is present.

UMFPACK_OK

The linear system was successfully solved.

UMFPACK_WARNING_singular_matrix

A divide-by-zero occurred. Your solution will contain Inf's and/or NaN's. Some parts of the solution may be valid. For example, solving $Ax=b$ with

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{returns } x = \begin{bmatrix} 0.5 \\ \text{Inf} \end{bmatrix}$$

UMFPACK_ERROR_out_of_memory

Insufficient memory to solve the linear system.

UMFPACK_ERROR_argument_missing

One or more required arguments are missing. The B, X, (or Bx and Xx for the complex versions) arguments are always required. Info and Control are not required. Ap, Ai, Ax are required if $Ax=b$, $A'x=b$, $A.'x=b$ is to be solved, the (default) iterative refinement is requested, and the matrix A is nonsingular.

UMFPACK_ERROR_invalid_system

The sys argument is not valid, or the matrix A is not square.

UMFPACK_ERROR_invalid_Numeric_object

The Numeric object is not valid.

Info [UMFPACK_NROW], Info [UMFPACK_NCOL]:

The dimensions of the matrix A (L is n_row-by-n_inner and U is n_inner-by-n_col, with n_inner = min(n_row,n_col)).

Info [UMFPACK_NZ]: the number of entries in the input matrix, Ap [n], if iterative refinement is requested ($Ax=b$, $A'x=b$, or $A.'x=b$ is being solved, Control [UMFPACK_IRSTEP] ≥ 1 , and A is nonsingular).

Info [UMFPACK_IR_TAKEN]: The number of iterative refinement steps effectively taken. The number of steps attempted may be one more than this; the refinement algorithm backtracks if the last refinement step worsens the solution.

Info [UMFPACK_IR_ATTEMPTED]: The number of iterative refinement steps attempted. The number of times a linear system was solved is one

more than this (once for the initial $Ax=b$, and once for each $Ay=r$ solved for each iterative refinement step attempted).

Info [UMFPACK_OMEGA1]: sparse backward error estimate, omega1, if iterative refinement was performed, or -1 if iterative refinement not performed.

Info [UMFPACK_OMEGA2]: sparse backward error estimate, omega2, if iterative refinement was performed, or -1 if iterative refinement not performed.

Info [UMFPACK_SOLVE_FLOPS]: the number of floating point operations performed to solve the linear system. This includes the work taken for all iterative refinement steps, including the backtrack (if any).

Info [UMFPACK_SOLVE_TIME]: The time taken, in seconds.

Info [UMFPACK_SOLVE_WALLTIME]: The wallclock time taken, in seconds.

Only the above listed Info [...] entries are accessed. The remaining entries of Info are not accessed or modified by umfpack*_solve. Future versions might modify different parts of Info.

*/

5.4 umfpack_di_free_symbolic

```
void umfpack_di_free_symbolic
(
    void **Symbolic
) ;

void umfpack_dl_free_symbolic
(
    void **Symbolic
) ;

void umfpack_zi_free_symbolic
(
    void **Symbolic
) ;

void umfpack_zl_free_symbolic
(
    void **Symbolic
) ;

/*
double int32_t Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    umfpack_di_free_symbolic (&Symbolic) ;

double int64_t Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    umfpack_dl_free_symbolic (&Symbolic) ;

complex int32_t Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    umfpack_zi_free_symbolic (&Symbolic) ;

complex int64_t Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    umfpack_zl_free_symbolic (&Symbolic) ;
```

Purpose:

Deallocates the Symbolic object and sets the Symbolic handle to NULL. This routine is the only valid way of destroying the Symbolic object.

Arguments:

void **Symbolic ; Input argument, set to (void *) NULL on output.

```
        Points to a valid Symbolic object computed by umfpack*_symbolic.
        No action is taken if Symbolic is a (void *) NULL pointer.
*/
```

5.5 umfpack_di_free_numeric

```
void umfpack_di_free_numeric
(
    void **Numeric
);

void umfpack_dl_free_numeric
(
    void **Numeric
);

void umfpack_zi_free_numeric
(
    void **Numeric
);

void umfpack_zl_free_numeric
(
    void **Numeric
);

/*
double int32_t Syntax:

    #include "umfpack.h"
    void *Numeric ;
    umfpack_di_free_numeric (&Numeric) ;

double int64_t Syntax:

    #include "umfpack.h"
    void *Numeric ;
    umfpack_dl_free_numeric (&Numeric) ;

complex int32_t Syntax:

    #include "umfpack.h"
    void *Numeric ;
    umfpack_zi_free_numeric (&Numeric) ;

complex int64_t Syntax:

    #include "umfpack.h"
    void *Numeric ;
    umfpack_zl_free_numeric (&Numeric) ;
```

Purpose:

Deallocates the Numeric object and sets the Numeric handle to NULL. This routine is the only valid way of destroying the Numeric object.

Arguments:

```
void **Numeric ;          Input argument, set to (void *) NULL on output.

    Numeric points to a valid Numeric object, computed by umfpack*_numeric.
    No action is taken if Numeric is a (void *) NULL pointer.
*/
}
```

5.6 umfpack_di_defaults

```
void umfpack_di_defaults
(
    double Control [UMFPACK_CONTROL]
) ;

void umfpack_dl_defaults
(
    double Control [UMFPACK_CONTROL]
) ;

void umfpack_zi_defaults
(
    double Control [UMFPACK_CONTROL]
) ;

void umfpack_zl_defaults
(
    double Control [UMFPACK_CONTROL]
) ;

/*
double int32_t Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL] ;
    umfpack_di_defaults (Control) ;

double int64_t Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL] ;
    umfpack_dl_defaults (Control) ;

complex int32_t Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL] ;
    umfpack_zi_defaults (Control) ;
```

complex int64_t Syntax:

```
#include "umfpack.h"
double Control [UMFPACK_CONTROL] ;
umfpack_zl_defaults (Control) ;
```

Purpose:

Sets the default control parameter settings.

Arguments:

double Control [UMFPACK_CONTROL] ; Output argument.

Control is set to the default control parameter settings. You can then modify individual settings by changing specific entries in the Control array. If Control is a (double *) NULL pointer, then umfpack*_defaults returns silently (no error is generated, since passing a NULL pointer for Control to any UMFPACK routine is valid).

*/