UMFPACK Version 4.0 User Guide

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

April 11, 2002

Abstract

UMFPACK is a set of routines for solving unsymmetric sparse linear systems, $\mathbf{A}\mathbf{x} = \mathbf{b}$, using the Unsymmetric MultiFrontal method and direct sparse LU factorization. It is written in ANSI/ISO C, with a MATLAB (Version 6.0 or 6.1) interface. UMFPACK relies on the Level-3 Basic Linear Algebra Subprograms (dense matrix multiply) for its performance. This code works on Windows and many versions of Unix (Sun Solaris, Red Hat Linux, IBM AIX, SGI IRIX, and Compaq Alpha).

Acknowledgments:

This work was supported by the National Science Foundation, under grants DMS-9504974 and DMS-9803599.

Contents

1 Overview

UMFPACK Version 4.0 is a set of routines for solving systems of linear equations, $\mathbf{Ax} = \mathbf{b}$, when $\mathbf{A}$ is sparse and unsymmetric. It is based on the Unsymmetric MultiFrontal method [4, 5], which factorizes $\mathbf{PAQ}$ into the product $\mathbf{LU}$, where $\mathbf{L}$ and $\mathbf{U}$ are lower and upper triangular, respectively, and $\mathbf{P}$ are $\mathbf{Q}$ are permutation matrices. Both $\mathbf{P}$ and $\mathbf{Q}$ are chosen to reduce fill-in (new nonzeros in $\mathbf{L}$ and $\mathbf{U}$ that are not present in $\mathbf{A}$). The permutation $\mathbf{P}$ has the dual role of reducing fill-in and maintaining numerical accuracy (via relaxed partial pivoting and row interchanges).

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 first finds a column pre-ordering that reduces fill-in, without regard to numerical values, with a modified version of COLAMD [6, 7, 23]. The method finds a symmetric permutation $\mathbf{Q}$ of the matrix $\mathbf{A}^\mathsf{T}\mathbf{A}$ (without forming $\mathbf{A}^\mathsf{T}\mathbf{A}$ explicitly). This is a good choice for $\mathbf{Q}$, since the Cholesky factors of $(\mathbf{AQ})^\mathsf{T}(\mathbf{AQ})$ are an upper bound (in terms of nonzero pattern) of the factor $\mathbf{U}$ for the unsymmetric LU factorization ($\mathbf{PAQ} = \mathbf{LU}$) regardless of the choice of $\mathbf{P}$ [16, 17, 19].

Next, the method breaks the factorization of the matrix $\mathbf{A}$ down into a sequence of dense rectangular frontal matrices. The frontal matrices are related to each other by a supernodal column elimination tree, in which each node in the tree represents one frontal matrix. This analysis phase also determines upper bounds on the memory usage, the floating-point operation count, and the number of nonzeros in the LU factors.

UMFPACK factorizes each *chain* of frontal matrices in a single working array, similar to how the unifrontal method [14] factorizes the whole matrix. A chain of frontal matrices is a sequence of fronts where the parent of front i is i+1 in the supernodal column elimination tree. UMFPACK is an outer-product based, right-looking method. At the k-th step of Gaussian elimination, it represents the updated submatrix $\mathbf{A}_k$ as an implicit summation of a set of dense submatrices (referred to as *elements*, borrowing a phrase from finite-element methods) that arise when the frontal matrices are factorized and their pivot rows and columns eliminated.

Each frontal matrix represents the elimination of one or more columns; each column of $\mathbf{A}$ will be eliminated in a specific frontal matrix, and which frontal matrix will be used for each column is determined by the pre-analysis phase. The pre-analysis phase also determines the worst-case size of each frontal matrix so that they can hold any candidate pivot column and any candidate pivot row. From the perspective of the analysis phase, any candidate pivot column in the frontal matrix is identical (in terms of nonzero pattern), and so is any row. However, the numerical factorization phase has more information than the analysis phase. It uses this information to reorder the columns within each frontal matrix to reduce fill-in. Similarly, since the number of nonzeros in each row and column are maintained (more precisely, COLMMD-style approximate degrees [18]), a pivot row can be selected based on sparsity-preserving criteria (low degree) as well as numerical considerations (relaxed threshold partial pivoting). This information about row and column degrees is not available to left-looking methods such as SuperLU [10] or MATLAB's LU [18, 21].

More details of the method, including experimental results, are described in [2, 3], available at www.cise.ufl.edu/tech-reports.

Table 1: Using UMFPACK's MATLAB interface

Function	Using UMFPACK	MATLAB 6.0 equivalent
Solve $\mathbf{Ax} = \mathbf{b}$.	`x = umfpack (A,'\',b) ;`	`x = A \ b ;`

Solve $\mathbf{Ax} = \mathbf{b}$ using a different column pre-ordering.	`S = spones (A) ;`
	`Q = symamd (S+S') ;`
	`x = umfpack (A,Q,'\',b) ;`
		`spparms ('autommd',0) ;`
		`S = spones (A) ;`
		`Q = symamd (S+S') ;`
		`x = A (:,Q) \ b ;`
		`x (Q) = x ;`
		`spparms ('autommd',1) ;`

Solve $\mathbf{A}^\mathsf{T}\mathbf{x}^\mathsf{T} = \mathbf{b}^\mathsf{T}$.	`x = umfpack (b,'/',A) ;`	`x = b / A ;`

Factorize $\mathbf{A}$, then solve $\mathbf{Ax} = \mathbf{b}$.	`[L,U,P,Q] = umfpack (A) ;`
	`x = U \ (L \ (b (P))) ;`
	`x (Q) = x ;`
		`Q = colamd (A) ;`
		`[L,U,P] = lu (A (:,Q)) ;`
		`x = U \ (L \ (P*b)) ;`
		`x (Q) = x ;`

2 Availability

UMFPACK Version 4.0 is available at www.cise.ufl.edu/research/sparse. An earlier version (3.2) has been submitted as a collected algorithm of the ACM [2, 3]. Version 3.2 handles only real, square, non-singular matrices. Version 3.0 and following make use of a modified version of COLAMD V2.0 by Timothy A. Davis, Stefan Larimore, John Gilbert, and Esmond Ng. The original COLAMD V2.1 is available in as a built-in routine in MATLAB V6.0 (or later), and at www.cise.ufl.edu/research/sparse. These codes are also available in Netlib [12] at www.netlib.org. UMFPACK Versions 2.2.1 and earlier, co-authored with Iain Duff, are available at www.cise.ufl.edu/research/sparse and as MA38 (functionally equivalent to Version 2.2.1) in the Harwell Subroutine Library.

3 Using UMFPACK in MATLAB

The easiest way to use UMFPACK is within MATLAB. This discussion assumes that you have MATLAB Version 6.0 or later (which includes the BLAS, and the `colamd` ordering routine). To compile the UMFPACK mexFunction, just type `make umfpack` in the Unix system shell. You can also type `umfpack_make` in MATLAB (which should work on any system, including Windows). See Section 7 for more details on how to install UMFPACK. Once installed, the UMFPACK mexFunction can analyze, factor, and solve linear systems. Table 1 summarizes some of the more common uses of UMFPACK within MATLAB.

UMFPACK requires `b` to be a dense vector (real or complex) of the appropriate dimension. This is more restrictive than what you can do with MATLAB's backslash or forward slash. Future releases of UMFPACK may allow for b to be sparse, or allow it to be a matrix rather than just a vector.

MATLAB's [L,U,P] = lu(A) returns a lower triangular L, an upper triangular U, and a permutation matrix P such that P*A is equal to L*U. UMFPACK behaves differently; it returns P and Q such that A(P,Q) is equal to L*U, where P and Q are permutation vectors. If you prefer permutation matrices, use the following MATLAB code:

```
[L,U,P,Q] = umfpack (A) ;
[m n] = size (A) ;
I = speye (m) ; P = I (P,:) ;
I = speye (n) ; Q = I (:,Q) ;
```

Now P*A*Q is equal to L*U. Note that x = umfpack(A,'\',b) requires that b be a dense column vector. If you wish to use the LU factors from UMFPACK to solve a linear system, $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{b}$ is a either a dense or sparse matrix with more than one column, do this:

```
[L,U,P,Q] = umfpack (A) ;
x = U \ (L \ (b (P,:))) ;
x (Q,:) = x ;
```

The above example does not make use of the iterative refinement that is built into x = umfpack (A,'\',b) however.

There are more options; you can provide your own column pre-ordering (in which case UMFPACK does not call COLAMD), you can modify other control settings (similar to the spparms in MATLAB), and you can get various statistics on the analysis, factorization, and solution of the linear system. Type help umfpack_details and help umfpack_report in MATLAB for more information. Two demo m-files are provided. Just type umfpack_simple and umfpack_demo to run them. They roughly correspond to the C programs umfpack_simple.c and umfpack_di_demo.c. You may want to type more on before running umfpack_demo since it generates lots of output. The output of these two programs should be about the same as the files umfpack_simple.m.out and umfpack_demo.m.out that are provided.

Factorizing A' (or A.') and using the transposed factors can sometimes be faster than factorizing A. It can also be preferable to factorize A' if A is rectangular. UMFPACK preorders the columns to maintain sparsity; the row ordering is not determined until the matrix is factorized. Thus, if A is m by n with rank m and m $<$ n, then umfpack might not find a factor U with a zero-free diagonal. Unless the matrix ill-conditioned or poorly scaled, factorizing A' in this case will guarantee that both factors will have zero-free diagonals. Here's how you can factorize A' and get the factors of A instead:

```
[l,u,p,q] = umfpack (A') ;
L = u' ;
U = l' ;
P = q ;
Q = p ;
clear l u p q
```

This is an alternative to [L,U,P,Q]=umfpack(A). The above code is an excerpt from the umfpack_factorize function. It orders and analyzes both A and A', and then computes the numerical factorization that has a lower bound on the number of floating-point operations required.

A simple M-file (umfpack_btf) is provided that first permutes the matrix to upper block triangular form, using MATLAB's dmperm routine, and then solves each block. The LU factors are not returned. Its usage is simple: x = umfpack_btf(A,b). Type help umfpack_btf for more options. An estimate of the 1-norm of L*U-A(P,Q) can be computed in MATLAB as lu_normest(A(P,Q),L,U), using the lu_normest.m M-file by Hager and Davis [8] that is included with the UMFPACK V4.0 distribution.

One issue you may encounter is how UMFPACK allocates its memory when being used in a mexFunction. One part of its working space is of variable size. The symbolic analysis phase determines an upper bound on the size of this memory, but not all of this memory will typically be used in the numerical factorization. UMFPACK tries to allocate a decent amount of working space (70% of the upper bound, by default). If this fails, it reduces its request and uses less memory. If the space is not large enough during factorization, it is increased via realloc.

However, mxMalloc aborts the umfpack mexFunction if it fails, so this strategy doesn't work in MATLAB. The strategy works fine when malloc is used instead. If you run out of memory in MATLAB, try reducing Control(7) to be less than 0.70, and try again. Alternatively, set Control(7) to 1.0 or 1.05 to avoid all reallocations of memory. Type help umfpack_details and umfpack_report for more information, and refer to the Control [UMFPACK_ALLOC_INIT] parameter described in umfpack_*_numeric in Section 10, below.

There is a solution to this problem, but it relies on undocumented internal MATLAB routines (utMalloc, utFree, and utRealloc). See the -DMATHWORKS option in the file umf_config.h for details.

4 Using UMFPACK in a C program

The C-callable UMFPACK library consists of 24 user-callable routines and one include file. Twenty-three of the routines come in four versions, with different sizes of integers and for real or complex floating-point numbers:

1. umfpack_di_*: real double precision, int integers.
2. umfpack_dl_*: real double precision, long integers.
3. umfpack_zi_*: complex double precision, int integers.
4. umfpack_zl_*: complex double precision, long integers.

where * denotes the specific name of one of the 23 routines. Routine names beginning with umf_ are internal to the package, and should not be called by the user. The include file umfpack.h must be included in any C program that uses UMFPACK.

Use only one version for any one problem; do not attempt to use one version to analyze the matrix and another version to factorize the matrix, for example.

The notation umfpack_di_* refers to all 23 user-callable routines for the real double precision and int integer case. The notation umfpack_*_numeric, for example, refers all four versions (real/complex, int/long) of a single operation (in this case numerical factorization).

4.1 The size of an integer

The `umfpack_di_*` and `umfpack_zi_*` routines use `int` integer arguments; those starting with `umfpack_dl_` or `umfpack_zl_` use `long` integer arguments. If you compile UMFPACK in the standard ILP32 mode (32-bit `int`'s, `long`'s, and pointers) then the versions are essentially identical. You will be able to solve problems using up to 2GB of memory. If you compile UMFPACK in the standard LP64 mode, the size of an `int` remains 32-bits, but the size of a `long` and a pointer both get promoted to 64-bits. In the LP64 mode, the `umfpack_dl_*` and `umfpack_zl_*` routines can solve huge problems (not limited to 2GB), limited of course by the amount of available memory. The only drawback to the 64-bit mode is that few BLAS libraries support 64-bit integers. This limits the performance you will obtain.

4.2 Real and complex floating-point

The `umfpack_di_*` and `umfpack_dl_*` routines take (real) double precision arguments, and return double precision arguments. In the `umfpack_zi_*` and `umfpack_zl_*` routines, these same arguments hold the real part of the matrices; and second double precision array holds the imaginary part of the input and output matrices. Internally, complex numbers are stored in arrays with their real and imaginary parts interleaved, as required by the BLAS.

4.3 Primary routines, and a simple example

Five primary UMFPACK routines are required to factorize $\mathbf{A}$ or solve $\mathbf{Ax} = \mathbf{b}$. They are fully described in Section 10:

- `umfpack_*_symbolic`:

  Pre-orders the columns of $\mathbf{A}$ to reduce fill-in, based on its sparsity pattern only, finds the supernodal column elimination tree, and post-orders the tree. Returns an opaque `Symbolic` object as a `void *` pointer. The object contains the symbolic analysis and is needed for the numerical factorization. This routine requires only $O(|\mathbf{A}|)$ space, where $|\mathbf{A}|$ is the number of nonzero entries in the matrix. It computes upper bounds on the nonzeros in $\mathbf{L}$ and $\mathbf{U}$, the floating-point operations required, and the memory usage of `umfpack_*_numeric`. The `Symbolic` object is small; it contains just the column pre-ordering, the supernodal column elimination tree, and information about each frontal matrix, and is no larger than about $m + 8n$ integers (where $\mathbf{A}$ is m-by-n).

- `umfpack_*_numeric`:

  Numerically factorizes a sparse matrix into $\mathbf{PAQ} = \mathbf{LU}$. Requires the symbolic ordering and analysis computed by `umfpack_*_symbolic` or `umfpack_*_qsymbolic`. Returns an opaque `Numeric` object as a `void *` pointer. The object contains the numerical factorization and is used by `umfpack_*_solve`. You can factorize a new matrix with a different values (but identical pattern) as the matrix analyzed by `umfpack_*_symbolic` or `umfpack_*_qsymbolic` by re-using the `Symbolic` object (this feature is available when using UMFPACK in a C program, but not in MATLAB). The matrix $\mathbf{L}$ is unit lower triangular. The matrix $\mathbf{U}$ will have zeros on the diagonal if $\mathbf{A}$ is singular; this produces a warning, but the factorization is still valid.

- `umfpack_*_solve`:

  Solves a sparse linear system ($\mathbf{Ax} = \mathbf{b}$, $\mathbf{A}^\mathsf{T}\mathbf{x} = \mathbf{b}$, or systems involving just $\mathbf{L}$ or $\mathbf{U}$), using the numeric factorization computed by `umfpack_*_numeric`. Iterative refinement with sparse backward error [1] is used by default. The matrix $\mathbf{A}$ must be square. If it is singular, then a divide-by-zero will occur, and your solution with contain IEEE Inf's or NaN's.

- `umfpack_*_free_symbolic`:

  Frees the `Symbolic` object created by `umfpack_*_symbolic` or `umfpack_*_qsymbolic`.

- `umfpack_*_free_numeric`:

  Frees the `Numeric` object created by `umfpack_*_numeric`.

Be careful not to free a `Symbolic` object with `umfpack_*_free_numeric`. Nor should you attempt to free a `Numeric` object with `umfpack_*_free_symbolic`. Failure to free these objects will lead to memory leaks.

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 or four arrays, where the matrix is `m`-by-`n`, with `nz` entries. For the `int` version of UMFPACK:

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

For the `long` version of UMFPACK:

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

The complex versions add a second array for the imaginary part:

```
double Az [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]`. The imaginary part, for the complex versions, is stored in `Az[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 UMFPACK.

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

int    n = 5 ;
int    Ap [ ] = {0, 2, 5, 9, 10, 12} ;
int    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 *Control = (double *) NULL, *Info = (double *) NULL ;
    int i ;
    void *Symbolic, *Numeric ;
    (void) umfpack_di_symbolic (n, n, Ap, Ai, &Symbolic, Control, Info) ;
    (void) umfpack_di_numeric (Ap, Ai, Ax, Symbolic, &Numeric, Control, Info) ;
    umfpack_di_free_symbolic (&Symbolic) ;
    (void) umfpack_di_solve (UMFPACK_A, Ap, Ai, Ax, x, b, Numeric,
        Control, Info) ;
    umfpack_di_free_numeric (&Numeric) ;
    for (i = 0 ; i < n ; i++) printf ("x [%d] = %g\n", i, x [i]) ;
    return (0) ;
}
```

It solves the same linear system as the `umfpack_simple.m` MATLAB m-file. The `Ap`, `Ai`, and `Ax` arrays represent the matrix

$$\mathbf{A} = \left[\begin{array}{rrrrr} 2 & 3 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 & 6 \\ 0 & -1 & -3 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 4 & 2 & 0 & 1 \end{array} \right].$$

and the solution is $\mathbf{x} = [1\,2\,3\,4\,5]^{\mathsf{T}}$. 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`).

4.4 A note about zero-sized arrays

UMFPACK uses many user-provided arrays of size `n_row` or `n_col` (the order of the matrix), and of size `nz` (the number of nonzeros in a matrix). UMFPACK does not handle zero-dimensioned arrays; it returns an error code if `n_row` or `n_col` are zero. However, `nz` can be zero, since all singular matrices are handled correctly. If you attempt to `malloc` an array of size `nz` = 0, however, `malloc` will return a null pointer which UMFPACK will report as a missing argument. If you `malloc` an array of size `nz` to pass to UMFPACK, make sure that you handle the `nz` = 0 case correctly (use a size equal to the maximum of `nz` and 1, or use a size of `nz+1`).

4.5 Alternative routines

Three alternative routines are provided that modify UMFPACK's default behavior. They are fully described in Section 11:

- `umfpack_*_defaults`:

  Sets the default control parameters in the `Control` array. These can then be modified as desired before passing the array to the other UMFPACK routines. Control parameters are summarized in Section 4.10. One particular parameter deserves special notice. UMFPACK uses relaxed partial pivoting, where a candidate pivot entry is numerically acceptable if its magnitude is greater than or equal to a tolerance parameter times the magnitude of the largest entry in the same column. The parameter `Info[UMFPACK_PIVOT_TOLERANCE]` has a default value of 0.1. This may be too small for some matrices, particularly for ill-conditioned or poorly scaled ones. With the default pivot tolerance and default iterative refinement, `x = umfpack (A,'\',b)` is just as accurate as (or more accurate) than `x = A\b` in MATLAB for nearly all matrices. For complex matrices, a cheap approximation of the absolute value is used for the threshold pivoting test ($|a| \approx |a_{\mathrm{real}}| + |a_{\mathrm{imag}}|$).

- `umfpack_*_qsymbolic`:

  An alternative to `umfpack_*_symbolic`. Allows the user to specify his or her own column pre-ordering, rather than using the default COLAMD pre-ordering.

- `umfpack_*_wsolve`:

  An alternative to `umfpack_*_solve` which does not dynamically allocate any memory. Requires the user to pass two additional work arrays.

4.6 Matrix manipulation routines

The compressed column data structure is compact, and simplifies the UMFPACK routines that operate on the sparse matrix **A**. However, it can be inconvenient for the user to generate. Section 12 presents the details of routines for manipulating sparse matrices in *triplet* form, compressed column form, and compressed row form (the transpose of the compressed column form). The triplet form of a matrix consists of three or four arrays. For the `int` version of UMFPACK:

```
int Ti [nz] ;
int Tj [nz] ;
double Tx [nz] ;
```

For the `long` version:

```
long Ti [nz] ;
long Tj [nz] ;
double Tx [nz] ;
```

The complex versions use a second array to hold the imaginary part:

```
double Tz [nz] ;
```

The `k`-th triplet is (i, j, a_{ij}), where $i =$ `Ti[k]`, $j =$ `Tj[k]`, and $a_{ij} =$ `Tx[k]`. For the complex versions, `Tx[k]` is the real part of a_{ij} and `Tz[k]` is the imaginary part. The triplets can be in any order in the `Ti`, `Tj`, and `Tx` arrays (and `Tz` for the complex versions), and duplicate entries may exist. Any duplicate entries are summed when the triplet form is converted to compressed column form. This is a convenient way to create a matrix arising in finite-element methods, for example.

Three routines are provided for manipulating sparse matrices:

- `umfpack_*_triplet_to_col`:

  Converts a triplet form of a matrix to compressed column form (ready for input to `umfpack_*_symbolic`, `umfpack_*_qsymbolic`, and `umfpack_*_numeric`). Identical to `A = spconvert(i,j,x)` in MATLAB, except that zero entries are not removed, so that the pattern of entries in the compressed column form of $\mathbf{A}$ are fully under user control. This is important if you want to factorize a new matrix with the `Symbolic` object from a prior matrix with the same pattern as the new one. MATLAB never stores explicitly zero entries, and does not support the reuse of the `Symbolic` object.

- `umfpack_*_col_to_triplet`:

  The opposite of `umfpack_*_triplet_to_col`. Identical to `[i,j,x] = find(A)` in MATLAB, except that numerically zero entries may be included.

- `umfpack_*_transpose`:

  Transposes and optionally permutes a column form matrix [22]. Identical to `R = A(P,Q)'` (linear algebraic transpose, using the complex conjugate) or `R = A(P,Q).'` (the array transpose) in MATLAB, except for the presence of numerically zero entries.

  Factorizing $\mathbf{A}^\mathsf{T}$ and then solving $\mathbf{Ax} = \mathbf{b}$ with the transposed factors can sometimes be much faster or much slower than factorizing $\mathbf{A}$. It is highly dependent on your particular matrix, however. See the `umfpack_factorize` MATLAB function for an example.

It is quite easy to add matrices in triplet form, subtract them, transpose them, permute them, and construct a submatrix. Refer to the discussion of `umfpack_*_triplet_to_col` in Section 12 for more details. The only primary matrix operation not provided by UMFPACK is sparse matrix multiplication [22].

4.7 Getting the contents of opaque objects

There are cases where you may wish to do more with the LU factorization of a matrix than solve a linear system. The opaque `Symbolic` and `Numeric` objects are just that - opaque. In addition, the LU factors are stored in the `Numeric` object in a compact way that does not store all of the row and column indices [15]. These objects may not be dereferenced by the user, and even if they were, it would be difficult for a typical user to understand how the LU factors are stored. Three routines are provided for copying their contents into user-provided arrays using simpler data structures. They are fully described in Section 13:

- `umfpack_*_get_lunz`:

  Returns the number of nonzeros in $\mathbf{L}$ and $\mathbf{U}$.

- `umfpack_*_get_numeric`:

  Copies $\mathbf{L}$, $\mathbf{U}$, $\mathbf{P}$, and $\mathbf{Q}$ from the `Numeric` object into arrays provided by the user. The matrix $\mathbf{L}$ is returned in compressed row form (with the column indices in each row sorted in ascending order). The matrix $\mathbf{U}$ is returned in compressed column form (with sorted columns). There are no explicit zero entries in $\mathbf{L}$ and $\mathbf{U}$, but such entries may exist in the `Numeric` object. The permutations $\mathbf{P}$ and $\mathbf{Q}$ are represented as permutation vectors, where `P[k] = i` means that row `i` of the original matrix is the the `k`-th row of $\mathbf{PAQ}$, and where `Q[k] = j` means that column `j` of the original matrix is the `k`-th column of $\mathbf{PAQ}$. This is identical to how MATLAB uses permutation vectors.

- `umfpack_*_get_symbolic`:

  Copies the contents of the `Symbolic` object (the initial row and column preordering, supernodal column elimination tree, and information about each frontal matrix) into arrays provided by the user.

UMFPACK itself does not make use of the output of the `umfpack_*_get_*` routines; they are provided solely for returning the contents of the opaque `Symbolic` and `Numeric` objects to the user.

4.8 Reporting routines

None of the UMFPACK routines discussed so far prints anything, even when an error occurs. UMFPACK provides you with nine routines for printing the input and output arguments (including the `Control` settings and `Info` statistics) of UMFPACK routines discussed above. They are fully described in Section 14:

- `umfpack_*_report_status`:

  Prints the status (return value) of other `umfpack_*` routines.

- `umfpack_*_report_info`:

  Prints the statistics returned in the `Info` array by `umfpack_*_*symbolic`, `umfpack_*_numeric`, and `umfpack_*_*solve`.

- `umfpack_*_report_control`:

  Prints the `Control` settings.

- `umfpack_*_report_matrix`:

  Verifies and prints a compressed column-form or compressed row-form sparse matrix.

- `umfpack_*_report_triplet`:

  Verifies and prints a matrix in triplet form.

- `umfpack_*_report_symbolic`:

  Verifies and prints a `Symbolic` object.

- `umfpack_*_report_numeric`:

  Verifies and prints a `Numeric` object.

- `umfpack_*_report_perm`:

  Verifies and prints a permutation vector.

- `umfpack_*_report_vector`:

  Verifies and prints a real or complex vector.

The `umfpack_*_report_*` routines behave slightly differently when compiled into the C-callable UMFPACK library than when used in the MATLAB mexFunction. MATLAB stores its sparse matrices using the same compressed column data structure discussed above, where row and column indices of an m-by-n matrix are in the range 0 to $m-1$ or $n-1$, respectively. It prints them as if they are in the range 1 to m or n. The UMFPACK mexFunction behaves the same way.

You can control how much the `umfpack_*_report_*` routines print by modifying the `Control [UMFPACK_PRL]` parameter. Its default value is `UMFPACK_DEFAULT_PRL` which is equal to 1. Here is a summary of how the routines use this print level parameter:

- `umfpack_*_report_status`:

  No output if the print level is 0 or less, even when an error occurs. If 1, then error messages are printed, and nothing is printed if the status is `UMFPACK_OK`. If 2 or more, then the status is always printed. If 4 or more, then the UMFPACK Copyright is printed. If 6 or more, then the UMFPACK License is printed. See also the first page of this User Guide for the Copyright and License.

- `umfpack_*_report_control`:

  No output if the print level is 1 or less. If 2 or more, all of `Control` is printed.

- `umfpack_*_report_info`:

  No output if the print level is 1 or less. If 2 or more, all of `Info` is printed.

- all other `umfpack_*_report_*` routines:

  If the print level is 2 or less, then these routines return silently without checking their inputs. If 3 or more, the inputs are fully verified and a short status summary is printed. If 4, then the first few entries of the input arguments are printed. If 5, then all of the input arguments are printed.

4.9 Utility routines

UMFPACK includes a routine that returns the time used by the process, `umfpack_timer`. The routine uses either `getrusage` (which is preferred), or the ANSI C `clock` routine if that is not available. It is fully described in Section 15. It is the only routine that is identical in all four `int`/`long`, real/complex versions (there is no `umfpack_di_timer` routine, for example).

Table 2: UMFPACK Control parameters

MATLAB	ANSI C	default	description
Used by reporting routines:			
`Control(1)`	`Control[UMFPACK_PRL]`	1	printing level
Used by `umfpack_*_symbolic` and `umfpack_*_qsymbolic:`			
`Control(2)`	`Control[UMFPACK_DENSE_ROW]`	0.2	dense row threshold
`Control(3)`	`Control[UMFPACK_DENSE_COL]`	0.2	dense column threshold
Used by `umfpack_*_numeric:`			
`Control(4)`	`Control[UMFPACK_PIVOT_TOLERANCE]`	0.1	partial pivoting tolerance
`Control(5)`	`Control[UMFPACK_BLOCK_SIZE]`	24	BLAS block size
`Control(6)`	`Control[UMFPACK_RELAXED_AMALGAMATION]`	0.25	amalgamation
`Control(7)`	`Control[UMFPACK_ALLOC_INIT]`	0.7	initial memory allocation
`Control(14)`	`Control[UMFPACK_RELAXED2_AMALGAMATION]`	0.1	amalgamation
`Control(15)`	`Control[UMFPACK_RELAXED3_AMALGAMATION]`	0.125	amalgamation
Used by `umfpack_*_solve` and `umfpack_*_wsolve:`			
`Control(8)`	`Control[UMFPACK_IRSTEP]`	2	max iter. refi nement steps
Can only be changed at compile time:			
`Control(9)`	`Control[UMFPACK_COMPILED_WITH_BLAS]`	-	true if BLAS is used
`Control(10)`	`Control[UMFPACK_COMPILED_FOR_MATLAB]`	-	true for mexFunction
`Control(11)`	`Control[UMFPACK_COMPILED_WITH_GETRUSAGE]`	-	true if `getrusage` used
`Control(12)`	`Control[UMFPACK_COMPILED_IN_DEBUG_MODE]`	-	true if debug mode enabled

4.10 Control parameters

UMFPACK uses an optional `double` array of size 20, `Control`, to modify its control parameters. These may be modified by the user (see `umfpack_*_defaults`). Each user-callable routine includes a complete description of how each control setting modifies its behavior. Table 2 summarizes the entire contents of the `Control` array. Future versions may make use of additional entries in the `Control` array. Note that ANSI C uses 0-based indexing, while MATLAB user's 1-based indexing. Thus, `Control(1)` in MATLAB is the same as `Control[0]` or `Control[UMFPACK_PRL]` in ANSI C.

4.11 Larger examples

Full examples of all user-callable UMFPACK routines are available in four C main programs, `umfpack_*_demo.c`. Another example is the UMFPACK mexFunction, `umfpackmex.c`. The mexFunction accesses only the user-callable C interface to UMFPACK. The only features that it does not use are the support for the triplet form (MATLAB's sparse arrays are already in the compressed column form) and the ability to reuse the `Symbolic` object to numerically factorize a matrix whose pattern is the same as a prior matrix analyzed by `umfpack_*_symbolic` or `umfpack_*_qsymbolic`. The latter is an important feature, but the mexFunction does not return its opaque `Symbolic` and `Numeric` objects to MATLAB. Instead, it gets the contents of these objects after extracting them via the `umfpack_*_get_*` routines, and returns them as MATLAB sparse matrices.

5 Synopsis of C-callable routines

Each subsection, below, summarizes the input variables, output variables, return values, and calling sequences of the routines in one category. Variables with the same name as those already listed in a prior category have the same size and type.

The real, `long` integer `umfpack_dl_*` routines are identical to the real, `int` routines, except that `_di_` is replaced with `_dl_` in the name, and all `int` arguments become `long`. Similarly, the complex, `long` integer `umfpack_zl_*` routines are identical to the complex, `int` routines, except that `_zi_` is replaced with `_zl_` in the name, and all `int` arguments become `long`. Only the real and complex `int` versions are listed in the synopsis below.

The matrix **A** is `n_row`-by-`n_col` with `nz` entries. If it is square then `n` = `n_row` = `n_col`.

5.1 Primary routines: real/`int`

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

status = umfpack_di_symbolic (n_row, n_col, Ap, Ai, &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) ;
```

5.2 Alternative routines: real/`int`

```
int Qinit [n_col], Wi [n] ;
double W [5*n] ;

umfpack_di_defaults (Control) ;
status = umfpack_di_qsymbolic (n_row, n_col, Ap, Ai, Qinit, &Symbolic,
    Control, Info) ;
status = umfpack_di_wsolve (sys, Ap, Ai, Ax, X, B, Numeric,
    Control, Info, Wi, W) ;
```

5.3 Matrix manipulation routines: real/`int`

```
int Ti [nz], Tj [nz], P [n_row], Q [n_col], Rp [n_row+1], Ri [nz], Map [nz] ;
double Tx [nz], Rx [nz] ;

status = umfpack_di_col_to_triplet (n_col, Ap, Tj) ;
status = umfpack_di_triplet_to_col (n_row, n_col, nz, Ti, Tj, Tx, Ap, Ai, Ax,
    Map) ;
status = umfpack_di_transpose (n_row, n_col, Ap, Ai, Ax, P, Q, Rp, Ri, Rx) ;
```

5.4 Getting the contents of opaque objects: real/`int`

```
int lnz, unz, Lp [n_row+1], Lj [lnz], Up [n_col+1], Ui [unz] ;
```

```
double Lx [lnz], Ux [unz], D [min (n_row,n_col)] ;
int nfr, nchains, Ptree [n_row], Qtree [n_col], Front_npivcol [n_col+1],
    Front_parent [n_col+1], Front_1strow [n_col+1],
    Front_leftmostdesc [n_col+1], Chain_start [n_col+1],
    Chain_maxrows [n_col+1], Chain_maxcols [n_col+1] ;

status = umfpack_di_get_lunz (&lnz, &unz, &n_row, &n_col, &nz_udiag, Numeric) ;
status = umfpack_di_get_numeric (Lp, Lj, Lx, Up, Ui, Ux, P, Q, D, Numeric) ;
status = umfpack_di_get_symbolic (&n_row, &n_col, &nz, &nfr, &nchains,
    Ptree, Qtree, Front_npivcol, Front_parent, Front_1strow, Front_leftmostdesc,
    Chain_start, Chain_maxrows, Chain_maxcols, Symbolic) ;
```

5.5 Reporting routines: real/`int`

```
umfpack_di_report_status (Control, status) ;
umfpack_di_report_control (Control) ;
umfpack_di_report_info (Control, Info) ;
status = umfpack_di_report_matrix (n_row, n_col, Ap, Ai, Ax, 1, Control) ;
status = umfpack_di_report_matrix (n_row, n_col, Rp, Ri, Rx, 0, Control) ;
status = umfpack_di_report_numeric (Numeric, Control) ;
status = umfpack_di_report_perm (n_row, P, Control) ;
status = umfpack_di_report_perm (n_col, Q, Control) ;
status = umfpack_di_report_symbolic (Symbolic, Control) ;
status = umfpack_di_report_triplet (n_row, n_col, nz, Ti, Tj, Tx, Control) ;
status = umfpack_di_report_vector (n, X, Control) ;
```

5.6 Primary routines: complex/`int`

```
double Az [nz], Xx [n], Xz [n], Bx [n], Bz [n] ;

status = umfpack_zi_symbolic (n_row, n_col, Ap, Ai, &Symbolic, Control, Info) ;
status = umfpack_zi_numeric (Ap, Ai, Ax, Az, Symbolic, &Numeric,
    Control, Info) ;
status = umfpack_zi_solve (sys, Ap, Ai, Ax, Az, Xx, Xz, Bx, Bz, Numeric,
    Control, Info) ;
umfpack_zi_free_symbolic (&Symbolic) ;
umfpack_zi_free_numeric (&Numeric) ;
```

5.7 Alternative routines: complex/`int`

```
double Wz [10*n] ;

umfpack_zi_defaults (Control) ;
status = umfpack_zi_qsymbolic (n_row, n_col, Ap, Ai, Qinit, &Symbolic,
    Control, Info) ;
status = umfpack_zi_wsolve (sys, Ap, Ai, Ax, Az, Xx, Xz, Bx, Bz, Numeric,
    Control, Info, Wi, Wz) ;
```

5.8 Matrix manipulation routines: complex/`int`

```
double Tz [nz], Rz [nz] ;

status = umfpack_zi_col_to_triplet (n_col, Ap, Tj) ;
status = umfpack_zi_triplet_to_col (n_row, n_col, nz, Ti, Tj, Tx, Tz,
    Ap, Ai, Ax, Az, Map) ;
status = umfpack_zi_transpose (n_row, n_col, Ap, Ai, Ax, Az, P, Q,
    Rp, Ri, Rx, Rz, 1) ;
status = umfpack_zi_transpose (n_row, n_col, Ap, Ai, Ax, Az, P, Q,
    Rp, Ri, Rx, Rz, 0) ;
```

5.9 Getting the contents of opaque objects: complex/`int`

```
double Lz [lnz], Uz [unz], Dx [min (n_row,n_col)], Dz [min (n_row,n_col)] ;

status = umfpack_zi_get_lunz (&lnz, &unz, &n_row, &n_col, &nz_udiag, Numeric) ;
status = umfpack_zi_get_numeric (Lp, Lj, Lx, Lz, Up, Ui, Ux, Uz, P, Q,
    Dx, Dz, Numeric) ;
status = umfpack_zi_get_symbolic (&n_row, &n_col, &nz, &nfr, &nchains,
Ptree, Qtree, Front_npivcol, Front_parent, Front_1strow,
    Front_leftmostdesc, Chain_start, Chain_maxrows, Chain_maxcols, Symbolic) ;
```

5.10 Reporting routines: complex/`int`

```
umfpack_zi_report_status (Control, status) ;
umfpack_zi_report_control (Control) ;
umfpack_zi_report_info (Control, Info) ;
status = umfpack_zi_report_matrix (n_row, n_col, Ap, Ai, Ax, Az, 1, Control) ;
status = umfpack_zi_report_matrix (n_row, n_col, Rp, Ri, Rx, Rz, 0, Control) ;
status = umfpack_zi_report_numeric (Numeric, Control) ;
status = umfpack_zi_report_perm (n_row, P, Control) ;
status = umfpack_zi_report_perm (n_col, Q, Control) ;
status = umfpack_zi_report_symbolic (Symbolic, Control) ;
status = umfpack_zi_report_triplet (n_row, n_col, nz, Ti, Tj, Tx, Tz, Control) ;
status = umfpack_zi_report_vector (n, Xx, Xz, Control) ;
```

6 Synopsis of utility routines

This routine is the same in all four versions of UMFPACK.

```
double t ;

t = umfpack_timer ( ) ;
```

7 Installation

UMFPACK comes with a `Makefile` and a `GNUmakefile` for compiling the C-callable `umfpack.a` library and the `umfpack` mexFunction on Unix. System-dependent configurations are controlled by the `Makefile` (or `GNUmakefile`), and defined in the `umf_config.h` file. You should not have to modify `umf_config.h`.

To compile `umfpack.a` on most Unix systems, all you need to do is to type `make`. This will use the generic configuration, in `Make.generic`. The five demo programs will be executed, and their output will be compared with output files in the distribution. Expect to see a few differences, such as residual norms, compile-time control settings, and perhaps memory usage differences. The BLAS [9, 11, 24] will not be used, so the performance of UMFPACK will not be as high as possible. It will also attempt to compile the `umfpack` mexFunction, but this will fail if you do not have MATLAB V6.0 or later. For better performance, edit the `Makefile` (or `GNUmakefile` if you use the GNU version of make) and un-comment the `include Make.*` statement that is specific to your computer. If you don't know if you have GNU make or not, then simply edit both `Makefile` and `GNUmakefile`; otherwise, you might get the wrong file when you type `make`. For example,

```
# include Make.generic
# include Make.linux
# include Make.sgi
include Make.solaris
# include Make.alpha
# include Make.rs6000
```

will include the Solaris-specific configurations, which uses the Sun Performance Library BLAS (`sunperf`), and compiler optimizations that are different than the generic settings. If you change the `Makefile`, `GNUmakefile`, or your system-specific `Make.*` file, be sure to type `make purge` before recompiling. A draft `Make.windows` file is provided; it has not been tested. The http://www.cise.ufl.edu/research/sparse/umfpack web page provides more sample make files for Windows. Here are the various parameters that you can control in your `Make.*` file; many more details are in `umf_config.h`:

- `CC =` your C compiler, usually, `cc`. If you don't modify this string at all in your `Make.*`, then the `make` program will use your default C compiler (if `make` is installed properly).
- `RANLIB =` your system's `ranlib` program, if needed.
- `CFLAGS =` optimization flags, such as `-O`.
- `CONFIG =` configuration settings.
- `LIB =` your libraries, such as `-lm` or `-lblas`.

You can control these options in your `Make.*` file if you are using the `GNUmakefile` only:

- `OBJEXT =` the filename extension for object files (defaults to `.o` for Unix). Set this to `.obj` for Windows.
- `OUTPUT =` your compiler's `-o` option. At least one Windows compiler requires this to be `/Fo`.

The `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.
- `-DNCBLAS` if you do not have the C-BLAS [24].
- `-DNSUNPERF` if you are on Solaris but do not have the Sun Performance Library.
- `-DNSCSL` if you on SGI IRIX but do not have the SCSL library.
- `-DLONGBLAS` if your BLAS can take `long` integer input arguments. If not defined, then the `umfpack_*l_*` versions of UMFPACK that use `long` integers do not call the BLAS. This flag is set internally when using the Sun Performance BLAS or SGI's SCSL BLAS (both have 64-bit versions of the BLAS).
- Options for controlling how C calls the Fortran BLAS: `-DBLAS_BY_VALUE`, `-DBLAS_NO_UNDERSCORE`, and `-DBLAS_CHAR_ARG`. These are set automatically for Windows, Sun Solaris, SGI Irix, Red Hat Linux, Compaq Alpha, and AIX (the IBM RS 6000).
- `-DGETRUSAGE` if you have the `getrusage` function.
- `-DLP64` if you are compiling in the LP64 model (32 bit int's, 64 bit long's, and 64 bit pointers).

If you use the `gcc` compiler and call a Fortran BLAS package or the Sun Performance BLAS you may see compiler warnings. The BLAS routines `dgemm`, `dgemv`, `dger`, `zgemm`, `zgemv`, and `zgeru` may be implicitly declared. Header files are not provided for the Fortran BLAS. The Sun Performance BLAS header (`sunperf.h`) is not used because it is incorrect for `zgemm`, `zgemv`, and `zgeru`, and causes a failure when compiling in the LP64 mode. You may also see warnings about arguments to the C-BLAS on older SGI computers. **Ignore all of these warnings.**

To compile the `umfpack` mexFunction on Unix, you must first modify the `Makefile` to select your architecture. Then type `make`. The MATLAB `mex` command will select the appropriate compiler and compiler flags for your system, and the BLAS internal to MATLAB will be used. If you compare the performance of UMFPACK with other packages that use the BLAS, be sure to use the same BLAS library for your comparisons; MATLAB's BLAS is slightly slower than the Sun Performance BLAS, for example. The `mexopts.sh` file in your UMFPACK directory has been modified from the MATLAB default; the unmodified version is in `mexopts.sh.orig` for comparison, in case The MathWorks makes changes to its default `mexopts.sh` file at a subsequent date. You may wish to modify `mexopts.sh` to increase the optimization level (`COPTIMFLAGS`). This has been done for Solaris only.

You may also compile the mexFunction from within MATLAB, on any system, by typing `umfpack_make` in MATLAB. If you're running Windows and are using the `lcc` compiler bundled with MATLAB, then you must first copy the `umfpack`\lcc_lib\libmwlapack.lib file into the `<matlab>`\extern\lib\win32\lcc\ directory. Next, type `mex -setup` at the MATLAB prompt, and ask MATLAB to select the `lcc` compiler. MATLAB has built-in BLAS, but it cannot be accessed by a program compiled by `lcc` without first copying this file.

8 Known Issues

The Microsoft C or C++ compilers on a Pentium badly break the IEEE 754 standard, and do not treat NaN's properly. According to IEEE 754, the expression `x != x` is supposed to be true if and only if `x` is NaN. For non-compliant compilers in Windows that expression is always false, and another test must be used: `((x < x)` is true if and only if `x` is NaN. For compliant compilers, `x < x` is always false, for any value of `xx` (including NaN). To cover both cases, UMFPACK when running under Microsoft Windows defines the following macro, which is true if and only if `x` is NaN, regardless of whether your compiler is compliant or not:

```
#define SCALAR_IS_NAN(x) (((x) != (x)) || ((x) < (x)))
```

If your compiler breaks this test, then UMFPACK will fail catastrophically if it encounters a NaN. In that case, you might try to see if the common (but non-ANSI C) routine `isnan` is available, and modify the macro `SCALAR_IS_NAN` in `umf_version.h` accordingly. The simpler (and IEEE 754-compliant) test `x != x` is always true with Linux on a PC, and on every Unix compiler I've tested.

9 Future work

Here are a few features that are not in UMFPACK Version 4.0, in no particular order. They may appear in a future release of UMFPACK. If you are interested, let me know and I could consider including them:

1. Future versions may have different default `Control` parameters. Future versions may return more statistics in the `Info` array, and they may use more entries in the `Control` array.

2. A simple C function could be written that orders and analyzes both $\mathbf{A}$ and $\mathbf{A}^\mathsf{T}$ and performs the numerical factorization on the one with the smaller upper bound on floating-point operations or memory usage. See `umfpack_factorize.m` for a MATLAB version.

3. Forward/back solvers for the conventional row or column-form data structure for $\mathbf{L}$ and $\mathbf{U}$. This would enable a seperate solver that could be used to write a MATLAB mexFunction `x = lu_refine (A, b, L, U, P, Q)` that gives MATLAB access to the iterative refinement algorithm with sparse backward error analysis. It would also be easier to handle sparse right-hand-sides in this data structure, and end up with good asymptotic run-time in this case (particularly for $\mathbf{Lx} = \mathbf{b}$; see [21]).

4. Complex absolute value computations could be based on FDLIBM (see http://www.netlib.org/fdlibm), using the `hypot(x,y)` routine.

5. When using iterative refinement, the residual $\mathbf{Ax}-\mathbf{b}$ could be returned by `umfpack_solve` (`umfpack_wsolve` already does so, but this is not documented).

6. The solve routines could handle multiple right-hand sides, and sparse right-hand sides.

7. An option to redirect the error and diagnostic output to something other than standard output.

8. Permutation to block-triangular-form [13] for the C-callable interface.

9. The ability to use user-provided `malloc`, `free`, and `realloc` memory allocation routines. Note that UMFPACK makes very few calls to these routines. You can do this at compile-time by modifying the definitions of `ALLOCATE`, `FREE`, and `REALLOCATE` in the file `umf_config.h`.

10. The ability to use user-provided work arrays, so that `malloc`, `free`, and `realloc` realloc are not called. The `umfpack_*_wsolve` routine is one example.

11. Use a method that takes time proportional to the number of nonzeros in $\mathbf{A}$ to analyze $\mathbf{A}$ when `Qinit` is provided (or when `Qinit` is not provided and `umf_colamd` ignores "dense" rows) [20]. The current method in `umf_analyze.c` takes time proportional to the number of nonzeros in the upper bound of $\mathbf{U}$.

12. The complex versions could use ANSI C99 `double _Complex` arguments, and support the use of interleaved real/imaginary parts as input and output arguments. The `umfpack_*_report_vector` routine is one example.

13. Other basic sparse matrix operations, such as sparse matrix multiplication.

14. A Fortran interface. This is easy, but highly non-portable. See the `ChangeLog` for some hints.

15. A C++ interface.

16. A parallel version using MPI.

10 The primary UMFPACK routines

The include files are the same for all four versions of UMFPACK. The generic integer type is `Int`, which is an `int` or `long`, depending on which version of UMFPACK you are using.

10.1 umfpack_*_symbolic

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

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

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

long umfpack_zl_symbolic
(
    long n_row,
    long n_col,
    const long Ap [ ],
    const long Ai [ ],
    void **Symbolic,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;
```

```
double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

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 UMF_colamd, modified from colamd V2.0 for UMFPACK), and a symbolic
    factorization.  This is required before the matrix can be numerically
    factorized with umfpack_*_numeric.  If you wish to bypass the UMF_colamd
    pre-ordering and provide your own ordering, use umfpack_*_qsymbolic instead.

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.

    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.  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_DENSE_COL]:  columns with more than
            max (16, Control [UMFPACK_DENSE_COL] * 16 * sqrt (n_row))
            entries are placed placed last in the column pre-ordering by
            UMF_colamd.  Default: 0.2.

        Control [UMFPACK_DENSE_ROW]:  rows with more than
            max (16, Control [UMFPACK_DENSE_ROW] * 16 * sqrt (n_col))
            entries (after "dense" columns are removed) are ignored in the
            column pre-ordering, UMF_colamd.  Default: 0.2.

        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: 24.  Modifying this
            parameter affects when updates are applied to the working frontal
            matrix, and can indirectly affect fill-in and operation count.
            As long as the block size is large enough (8 or so), this parameter
            has modest effect on performance.  In Version 3.0, this parameter
            was an input to umfpack_*_numeric, and had a default value of 16.
            On a Sun UltraSparc, a block size of 24 is better for larger
            matrices (16 is better for smaller ones, but not by much).  In the
            current version, it is required in the symbolic analysis phase, and
            is thus an input to this phase.

    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_jumbled_matrix

                Columns of input matrix were jumbled (unsorted columns or
                duplicate entries).

            UMFPACK_ERROR_n_nonpositive

                n is less than or equal to zero.

            UMFPACK_ERROR_nz_negative

                Number of entries in the matrix is negative.

            UMFPACK_ERROR_Ap0_nonzero

                Ap [0] is nonzero.

            UMFPACK_ERROR_col_length_negative

                A column has a negative number of entries.

            UMFPACK_ERROR_row_index_out_of_bounds

                A row index is out of bounds.
            UMFPACK_ERROR_out_of_memory

                Insufficient memory to perform the symbolic analysis.

            UMFPACK_ERROR_argument_missing

                One or more required arguments is missing.

            UMFPACK_ERROR_problem_too_large

                Problem is too large; memory usage estimate causes an integer
                overlow.  If you are using umfpack_*i_symbolic, try using
                the long versions instead, umfpack_*l_symbolic.

            UMFPACK_ERROR_internal_error

                Something very serious went wrong.  This is a bug.
                Please contact the author (davis@cise.ufl.edu).

        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 int.

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

        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 UMF_colamd.
            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.  These are
            rows that either have no entries, or whose entries are all in
            "dense" columns.  Any given row is classified as either "dense"
            or "empty" or "sparse".

        Info [UMFPACK_NDENSE_COL]:  number of "dense" columns in A.  These
            columns are ordered last in the factorization, but before "empty"
            columns.  Any given column is classified as either "dense" or
            "empty" or "sparse".
```

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

Info [UMFPACK_SYMBOLIC_DEFRAG]:  number of garbage collections
    performed in UMF_colamd, the column pre-ordering routine, and in
    UMF_analyze, which is called if UMF_colamd isn’t, or if UMF_colamd
    ignores one or more "dense" rows.

Info [UMFPACK_SYMBOLIC_PEAK_MEMORY]:  the amount of memory (in Units)
    required for umfpack_*_symbolic to complete.  This is roughly
    2.2*nz + (26 to 31)*n integers for a square matrix, depending on the
    matrix.  This count includes the size of the Symbolic object itself,
    which is 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 9*n integers,
    depending on the matrix.

Info [UMFPACK_VARIABLE_INIT_ESTIMATE]: the Numeric object contains two
    parts.  The first is fixed in size (O (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.

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_*_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_SYMBOLIC_TIME]:  The time taken by umfpack_*_symbolic, in
    seconds.  In the ANSI C version, this may be invalid if the time
    taken is more than about 36 minutes, because of wrap-around in the
    ANSI C clock function.  Compile UMFPACK with -DGETRUSAGE if you have
    the more accurate getrusage function.

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.
```

10.2 umfpack_*_numeric

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

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, where P and Q are permutation matrices
    (represented as permutation vectors), 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.
    umfpack_*_numeric 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].

        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)].

        Future complex version:  if Ax is present and Az is NULL, then both real
        and imaginary parts will be 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.

    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.  A value of zero is treated as 1.0.  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_real| + |a_imag|
            instead of the more expensive-to-compute exact absolute value
            sqrt (a_real^2 + a_imag^2)).

        Control [UMFPACK_RELAXED_AMALGAMATION]:  This controls the creation of
            "elements" (small dense submatrices) that are formed when a frontal
            matrix is factorized.  A new element is created if the current one,
            plus the new pivot, contains "too many" explicitly zero numerical
            entries.  The two elements are merged if the number of extra zero
            entries is < Control [UMFPACK_RELAXED_AMALGAMATION] times the
            size of the merged element.  A lower setting decreases fill-in, but
            run-time and memory usage can increase.  A larger setting increases
            fill-in (because the extra zeros are treated as normal entries
            during pivot selection), but this can lead to an increase in
            run-time but (paradoxically) a decrease in memory usage (one merged
            elements can take less space than two separate elements).  Except
            for the initial column ordering, this parameter has the most impact
            on the run-time, fill-in, operation count, and memory usage.
            Default: 0.25, which is fine for nearly all matrices.
            (For nearly all matrices that I've tested, different values of this
            parameter can decrease the run-time by at most 5%, but can also
            dramatically increase the run time for some matrices).

        Control [UMFPACK_RELAXED2_AMALGAMATION]:  This, along with the block
            size (Control [UMFPACK_BLOCK_SIZE]), controls how often the
            pending updates are applied when the next pivot entry resides in
            the current frontal matrix.  If the number of zero entries in the
            LU part of the current frontal matrix would exceed this parameter
            times the size of the LU part, then the pending updates are applied
            before the next pivot is included in the frontal matrix.
            Default: 0.20 (that is, more than 20% zero entries causes the
            pending updates to be applied).  This input parameter is new
            since Version 3.1.

        Control [UMFPACK_RELAXED3_AMALGAMATION]:  This, along with the block
            size (Control [UMFPACK_BLOCK_SIZE]), controls how often the pending
            updates are applied when the next pivot entry does NOT reside
            in the current frontal matrix.  If the number of zero entries in the
            LU part of the current frontal matrix would exceed this parameter
            times the size of the LU part, then the pending updates are applied
            before the next pivot is included in the frontal matrix.
            Default: 0.10 (that is, more than 10% zero entries causes the
            pending updates to be applied).  This input parameter is new
            since Version 3.1.

        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].
        umfpack_*_numeric initially allocates space for the variable-sized
        part equal to this estimate times Control [UMFPACK_ALLOC_INIT].
        Typically, 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.  A value less than
        zero is treated as zero (in which case, just the bare minimum
        amount of memory needed to start the factorization is initially
        allocated).  The bare initial memory required is returned by
        umfpack_*_*symbolic in Info [UMFPACK_VARIABLE_INIT_ESTIMATE] (which
        in fact not an estimate, but exact).  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.  These two
        factors (1.2 and 0.95) are fixed control parameters defined in
        umf_internal.h and cannot be changed at run-time.  You would need
        to edit umf_internal.h to modify them.  If you do this, be sure that
        the two factors are greater than 1 and less than 1, respectively.

        The strategy of attempting to malloc a working space, and re-trying
        with a smaller space, may not work under MATLAB, since mxMalloc
        aborts the mexFunction if it fails.  The built-in umfpack routine
        uses utMalloc instead, which does cause this problem.  If you are
        using the umfpack mexFunction, decrease Control [UMFPACK_ALLOC_INIT]
        if you run out of memory in MATLAB.

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

        Changing this parameter has no 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).

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 (Ap and/or Ai) 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 (Ap and/or
        Ai) 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 <= 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 time taken by umfpack_*_numeric, in
    seconds.  In the ANSI C version, this may be invalid if the time
    taken is more than about 36 minutes, because of wrap-around in the
    ANSI C clock function.  Compile UMFPACK with -DGETRUSAGE if you have
    the more accurate getrusage function.

Info [UMFPACK_RCOND]:  A rough estimate of the condition number, equal
    to min (abs (diag (U))) / max (abs (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.

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.
```

10.3 umfpack_*_solve

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

long umfpack_dl_solve
(
    long sys,
    const long Ap [ ],
    const long 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 int Ap [ ],
    const int 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]
) ;

long umfpack_zl_solve
(
    long sys,
    const long Ap [ ],
    const long 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 int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:

    Given LU factors computed by umfpack_*_numeric (PAQ=LU) and the
    right-hand-side, B, solve a linear system for the solution X.  Iterative
    refinement is optionally performed.  This routine dynamically allocates
    workspace of size O(n).  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).

        Earlier versions used a string argument for sys.  It was changed to an
        integer to make it easier for a Fortran code to call UMFPACK.

    Int Ap [n+1] ;      Input argument, not modified.
    Int Ai [nz] ;       Input argument, not modified.
    double Ax [nz] ;    Input argument, not modified.
    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.

        Future complex version:  if Ax is present and Az is NULL, then both real
        and imaginary parts will be 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.
    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.

        Future complex version:  if Xx is present and Xz is NULL, then both real
        and imaginary parts will be 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.
    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.

        Future complex version:  if Bx is present and Bz is NULL, then both real
        and imaginary parts will be 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 occured.  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 = [2 0]  b = [ 1 ]  returns x = [ 0.5 ]
            [0 0]      [ 0 ]              [ Inf ]

    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, Bz, Xx and Xz for the complex versions) arguments
        are always required.  Info and Control are not required.  Ap,
        Ai, Ax (and Az for complex versions) 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 by umfpack_*_solve, in
    seconds.  In the ANSI C version, this may be invalid if the time
    taken is more than about 36 minutes, because of wrap-around in the
    ANSI C clock function.  Compile UMFPACK with -DGETRUSAGE if you have
    the more accurate getrusage function.

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.
```

10.4 umfpack_*_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 int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long 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;
    any other action (such as using "free (Symbolic) ;" or not freeing Symbolic
    at all) will lead to memory leaks.

Arguments:

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

        Symbolic must point to a valid Symbolic object, computed by
        umfpack_*_symbolic.  No action is taken if Symbolic is a (void *) NULL
        pointer.
```

10.5 umfpack_*_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 int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long 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;
    any other action (such as using "free (Numeric) ;" or not freeing Numeric
    at all) will lead to memory leaks.

Arguments:

    void **Numeric ;            Input argument, deallocated and Numeric is
                                set to (void *) NULL on output.

        Numeric must point to a valid Numeric object, computed by
        umfpack_*_numeric.  No action is taken if Numeric is a (void *) NULL
        pointer.
```

11 Alternatives routines

11.1 umfpack_*_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 int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long 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).
```

11.2 umfpack_*_qsymbolic

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

long umfpack_dl_qsymbolic
(
    long n_row,
    long n_col,
    const long Ap [ ],
    const long Ai [ ],
    const long Qinit [ ],
    void **Symbolic,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

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

long umfpack_zl_qsymbolic
(
    long n_row,
    long n_col,
    const long Ap [ ],
    const long Ai [ ],
    const long Qinit [ ],
    void **Symbolic,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

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

double long Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    long n_row, n_col, *Ap, *Ai, *Qinit, status ;
    double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
    status = umfpack_dl_qsymbolic (n_row, n_col, Ap, Ai, Qinit, &Symbolic,
        Control, Info) ;

complex int Syntax:

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

complex long Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    long n_row, n_col, *Ap, *Ai, *Qinit, status ;
    double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
    status = umfpack_zl_qsymbolic (n_row, n_col, Ap, Ai, Qinit, &Symbolic,
        Control, Info) ;

Purpose:

    Given the nonzero pattern of a sparse matrix A in column-oriented form, and
    a sparsity preserving column preordering Qinit, umfpack_*_qsymbolic performs
    the symbolic factorization of A*Qinit (or A (:,Qinit) in MATLAB notation).
    It also computes the column elimination tree post-ordering.  This is
    identical to umfpack_*_symbolic, except that colamd is not called and the
    user input column order Qinit is used instead.  Note that in general, the
    Qinit passed to umfpack_*_qsymbolic will differ from the final Q found in
    umfpack_*_numeric, because of the column etree postordering done in
    umfpack_*_qsymbolic and sparsity-preserving modifications made within each
    frontal matrix during umfpack_*_numeric.

    *** WARNING ***  A poor choice of Qinit can easily cause umfpack_*_numeric
    to use a huge amount of memory and do a lot of work.  The "default" symbolic
    analysis method is umfpack_*_symbolic, not this routine.  If you use this
    routine, the performance of UMFPACK is your responsibility;  UMFPACK will
    not try to second-guess a poor choice of Qinit.  If you are unsure about
    the quality of your Qinit, then call both umfpack_*_symbolic and
    umfpack_*_qsymbolic, and pick the one with lower estimates of work and
    memory usage (Info [UMFPACK_FLOPS_ESTIMATE] and
    Info [UMFPACK_PEAK_MEMORY_ESTIMATE]).  Don't forget to call
    umfpack_*_free_symbolic to free the Symbolic object that you don't need.

Returns:

    The value of Info [UMFPACK_STATUS]; see below.

Arguments:

    All arguments are the same as umfpack_*_symbolic, except for the following:

    Int Qinit [n_col] ;         Input argument, not modified.

        The user's fill-reducing initial column preordering.  This must be a
        permutation of 0..n_col-1.  If Qinit [k] = j, then column j is the kth
        column of the matrix A (:,Qinit) to be factorized.  If Qinit is an
        (Int *) NULL pointer, then colamd is called instead.  In fact,

        Symbolic = umfpack_*_symbolic (n_row, n_col, Ap, Ai, Control, Info) ;

        is identical to

        Symbolic = umfpack_*_qsymbolic (n_row, n_col, Ap, Ai, (Int *) NULL,
            Control, Info) ;

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

        Identical to umfpack_*_symbolic if Qinit is (Int *) NULL.  Otherwise,
        if Qinit is present, it is identical to umfpack_*_symbolic except for
        the following:

        Control [UMFPACK_DENSE_ROW]:  ignored.

        Control [UMFPACK_DENSE_COL]:  ignored.

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

        Identical to umfpack_*_symbolic if Qinit is (Int *) NULL.  Otherwise,
        if Qinit is present, it is identical to umfpack_*_symbolic except for
        the following:

        Info [UMFPACK_NDENSE_ROW]:  zero
        Info [UMFPACK_NEMPTY_ROW]:  number of empty rows.
        Info [UMFPACK_NDENSE_COL]:  zero
        Info [UMFPACK_NEMPTY_COL]:  number of empty columns.
```

11.3 umfpack_*_wsolve

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

long umfpack_dl_wsolve
(
    long sys,
    const long Ap [ ],
    const long Ai [ ],
    const double Ax [ ],
    double X [ ],
    const double B [ ],
    void *Numeric,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO],
    long Wi [ ],
    double W [ ]
) ;

int umfpack_zi_wsolve
(
    int sys,
    const int Ap [ ],
    const int 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 Wi [ ],
    double W [ ]
) ;

long umfpack_zl_wsolve
(
    long sys,
    const long Ap [ ],
    const long 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],
    long Wi [ ],
    double W [ ]
) ;

double int Syntax:

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

double long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    long status, *Ap, *Ai, *Wi, sys ;
    double *B, *X, *Ax, *W, Info [UMFPACK_INFO], Control [UMFPACK_CONTROL] ;
    status = umfpack_dl_wsolve (sys, Ap, Ai, Ax, X, B, Numeric,
        Control, Info, Wi, W) ;

complex int Syntax:

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

complex long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    long status, *Ap, *Ai, *Wi, sys ;
    double *Bx, *Bz, *Xx, *Xz, *Ax, *Az, *W,
        Info [UMFPACK_INFO], Control [UMFPACK_CONTROL] ;
    status = umfpack_zl_wsolve (sys, Ap, Ai, Ax, Az, Xx, Xz, Bx, Bz, Numeric,
        Control, Info, Wi, W) ;

Purpose:

    Given LU factors computed by umfpack_*_numeric (PAQ=LU) and the
    right-hand-side, B, solve a linear system for the solution X.  Iterative
    refinement is optionally performed.  This routine is identical to
    umfpack_*_solve, except that it does not dynamically allocate any workspace.
    When you have many linear systems to solve, this routine is faster than
    umfpack_*_solve, since the workspace (Wi, W) needs to be allocated only
    once, prior to calling umfpack_*_wsolve.

Returns:

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

Arguments:

    Int sys ;           Input argument, not modified.
    Int Ap [n+1] ;      Input argument, not modified.
    Int Ai [nz] ;       Input argument, not modified.
    double Ax [nz] ;    Input argument, not modified.
    double X [n] ;      Output argument.
    double B [n] ;      Input argument, not modified.
    void *Numeric ;     Input argument, not modified.
    double Control [UMFPACK_CONTROL] ;  Input argument, not modified.
    double Info [UMFPACK_INFO] ;        Output argument.

    for complex versions:
    double Az [nz] ;    Input argument, not modified, imaginary part
    double Xx [n] ;     Output argument, real part.
    double Xz [n] ;     Output argument, imaginary part
    double Bx [n] ;     Input argument, not modified, real part
    double Bz [n] ;     Input argument, not modified, imaginary part

        The above arguments are identical to umfpack_*_solve, except that the
        error code UMFPACK_ERROR_out_of_memory will not be returned in
        Info [UMFPACK_STATUS], since umfpack_*_wsolve does not allocate any
        memory.

    Int Wi [n] ;                Workspace.
    double W [c*n] ;            Workspace, where c is defined below.

        The Wi and W arguments are workspace used by umfpack_*_wsolve.  They
        need not be initialized on input, and their contents are undefined on
        output.  The size of W depends on whether or not iterative refinement is
        used, and which version (real or complex) is called.  Iterative
        refinement is performed if Ax=b, A'x=b, or A.'x=b is being solved,
        Control [UMFPACK_IRSTEP] > 0, and A is nonsingular.  The size of W is
        given below:

                                no iter.        with iter.
                                refinement      refinement
        umfpack_di_wsolve       n               5*n
        umfpack_dl_wsolve       n               5*n
        umfpack_zi_wsolve       4*n             10*n
        umfpack_zl_wsolve       4*n             10*n
```

12 Matrix manipulation routines

12.1 umfpack_*_col_to_triplet

```
int umfpack_di_col_to_triplet
(
    int n_col,
    const int Ap [ ],
    int Tj [ ]
) ;

long umfpack_dl_col_to_triplet
(
    long n_col,
    const long Ap [ ],
    long Tj [ ]
) ;

int umfpack_zi_col_to_triplet
(
    int n_col,
    const int Ap [ ],
    int Tj [ ]
) ;

long umfpack_zl_col_to_triplet
(
    long n_col,
    const long Ap [ ],
    long Tj [ ]
) ;

double int Syntax:

    #include "umfpack.h"
    int n_col, *Tj, *Ap, status ;
    status = umfpack_di_col_to_triplet (n_col, Ap, Tj) ;

double long Syntax:

    #include "umfpack.h"
    long n_col, *Tj, *Ap, status ;
    status = umfpack_dl_col_to_triplet (n_col, Ap, Tj) ;

complex int Syntax:

    #include "umfpack.h"
    int n_col, *Tj, *Ap, status ;
    status = umfpack_zi_col_to_triplet (n_col, Ap, Tj) ;

complex long Syntax:
    #include "umfpack.h"
    long n_col, *Tj, *Ap, status ;
    status = umfpack_zl_col_to_triplet (n_col, Ap, Tj) ;

Purpose:

    Converts a column-oriented matrix to a triplet form.  Only the column
    pointers, Ap, are required, and only the column indices of the triplet form
    are constructed.   This routine is the opposite of umfpack_*_triplet_to_col.
    The matrix may be singular and/or rectangular.  Analogous to [i, Tj, x] =
    find (A) in MATLAB, except that zero entries present in the column-form of
    A are present in the output, and i and x are not created (those are just Ai
    and Ax+Az*1i, respectively, for a column-form matrix A).

Returns:

    UMFPACK_OK if successful
    UMFPACK_ERROR_argument_missing if Ap or Tj is missing
    UMFPACK_ERROR_n_nonpositive if n_col <= 0
    UMFPACK_ERROR_Ap0_nonzero if Ap [0] != 0
    UMFPACK_ERROR_nz_negative if Ap [n_col] < 0
    UMFPACK_ERROR_col_length_negative if Ap [j] > Ap [j+1] for any j in the
        range 0 to n-1.
    Unsorted columns and duplicate entries do not cause an error (these would
    only be evident by examining Ai).  Empty rows and columns are OK.

Arguments:

    Int n_col ;         Input argument, not modified.

        A is an n_row-by-n_col matrix.  Restriction: n_col > 0.
        (n_row is not required)

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

        The column pointers of the column-oriented form of the matrix.  See
        umfpack_*_*symbolic for a description.  The number of entries in
        the matrix is nz = Ap [n_col].  Restrictions on Ap are the same as those
        for umfpack_*_transpose.  Ap [0] must be zero, nz must be >= 0, and
        Ap [j] <= Ap [j+1] and Ap [j] <= Ap [n_col] must be true for all j in
        the range 0 to n_col-1.  Empty columns are OK (that is, Ap [j] may equal
        Ap [j+1] for any j in the range 0 to n_col-1).

    Int Tj [nz] ;       Output argument.

        Tj is an integer array of size nz on input, where nz = Ap [n_col].
        Suppose the column-form of the matrix is held in Ap, Ai, Ax, and Az
        (see umfpack_*_*symbolic for a description).  Then on output, the
        triplet form of the same matrix is held in Ai (row indices), Tj (column
        indices), and Ax (numerical values).  Note, however, that this routine
        does not require Ai and Ax (or Az for the complex version) in order to
        do the conversion.
```

12.2 umfpack_*_triplet_to_col

```
int umfpack_di_triplet_to_col
(
    int n_row,
    int n_col,
    int nz,
    const int Ti [ ],
    const int Tj [ ],
    const double Tx [ ],
    int Ap [ ],
    int Ai [ ],
    double Ax [ ],
    int Map [ ]
) ;

long umfpack_dl_triplet_to_col
(
    long n_row,
    long n_col,
    long nz,
    const long Ti [ ],
    const long Tj [ ],
    const double Tx [ ],
    long Ap [ ],
    long Ai [ ],
    double Ax [ ],
    long Map [ ]
) ;

int umfpack_zi_triplet_to_col
(
    int n_row,
    int n_col,
    int nz,
    const int Ti [ ],
    const int Tj [ ],
    const double Tx [ ], const double Tz [ ],
    int Ap [ ],
    int Ai [ ],
    double Ax [ ], double Az [ ],
    int Map [ ]
) ;

long umfpack_zl_triplet_to_col
(
    long n_row,
    long n_col,
    long nz,
    const long Ti [ ],
    const long Tj [ ],
    const double Tx [ ], const double Tz [ ],
    long Ap [ ],
    long Ai [ ],
    double Ax [ ], double Az [ ],
    long Map [ ]
) ;

double int Syntax:

    #include "umfpack.h"
    int n_row, n_col, nz, *Ti, *Tj, *Ap, *Ai, status, *Map ;
    double *Tx, *Ax ;
    status = umfpack_di_triplet_to_col (n_row, n_col, nz, Ti, Tj, Tx,
        Ap, Ai, Ax, Map) ;

double long Syntax:

    #include "umfpack.h"
    long n_row, n_col, nz, *Ti, *Tj, *Ap, *Ai, status, *Map ;
    double *Tx, *Ax ;
    status = umfpack_dl_triplet_to_col (n_row, n_col, nz, Ti, Tj, Tx,
        Ap, Ai, Ax, Map) ;

complex int Syntax:

    #include "umfpack.h"
    int n_row, n_col, nz, *Ti, *Tj, *Ap, *Ai, status, *Map ;
    double *Tx, *Tz, *Ax, *Az ;
    status = umfpack_zi_triplet_to_col (n_row, n_col, nz, Ti, Tj, Tx, Tz,
        Ap, Ai, Ax, Az, Map) ;

long Syntax:

    #include "umfpack.h"
    long n_row, n_col, nz, *Ti, *Tj, *Ap, *Ai, status, *Map ;
    double *Tx, *Tz, *Ax, *Az ;
    status = umfpack_zl_triplet_to_col (n_row, n_col, nz, Ti, Tj, Tx, Tz,
        Ap, Ai, Ax, Az, Map) ;

Purpose:

    Converts a sparse matrix from "triplet" form to compressed-column form.
    Analogous to A = spconvert (Ti, Tj, Tx + Tx*1i) in MATLAB, except that
    zero entries present in the triplet form are present in A.

    The triplet form of a matrix is a very simple data structure for basic
    sparse matrix operations.  For example, suppose you wish to factorize a
    matrix A coming from a finite element method, in which A is a sum of
    dense submatrices, A = E1 + E2 + E3 + ... .  The entries in each element
    matrix Ei can be concatenated together in the three triplet arrays, and
    any overlap between the elements will be correctly summed by
    umfpack_*_triplet_to_col.
    Transposing a matrix in triplet form is simple; just interchange the
    use of Ti and Tj.  You can construct the complex conjugate transpose by
    negating Tz, for the complex versions.

    Permuting a matrix in triplet form is also simple.  If you want the matrix
    PAQ, or A (P,Q) in MATLAB notation, where P [k] = i means that row i of
    A is the kth row of PAQ and Q [k] = j means that column j of A is the kth
    column of PAQ, then do the following.  First, create inverse permutations
    Pinv and Qinv such that Pinv [i] = k if P [k] = i and Qinv [j] = k if
    Q [k] = j.  Next, for the mth triplet (Ti [m], Tj [m], Tx [m], Tz [m]),
    replace Ti [m] with Pinv [Ti [m]] and replace Tj [m] with Qinv [Tj [m]].

    If you have a column-form matrix with duplicate entries or unsorted
    columns, you can sort it and sum up the duplicates by first converting it
    to triplet form with umfpack_*_col_to_triplet, and then coverting it back
    with umfpack_*_triplet_to_col.

    Constructing a submatrix is also easy.  Just scan the triplets and remove
    those entries outside the desired subset of 0...n_row-1 and 0...n_col-1,
    and renumber the indices according to their position in the subset.

    You can do all these operations on a column-form matrix by first
    converting it to triplet form with umfpack_*_col_to_triplet, doing the
    operation on the triplet form, and then converting it back with
    umfpack_*_triplet_to_col.

    The only operation not supported easily in the triplet form is the
    multiplication of two sparse matrices (UMFPACK does not provide this
    operation).

    You can print the input triplet form with umfpack_*_report_triplet, and
    the output matrix with umfpack_*_report_matrix.

    The matrix may be singular (nz can be zero, and empty rows and/or columns
    may exist).  It may also be rectangular and/or complex.

Returns:

    UMFPACK_OK if successful.
    UMFPACK_ERROR_argument_missing if Ap, Ai, Ti, and/or Tj are missing.
    UMFPACK_ERROR_n_nonpositive if n_row <= 0 or n_col <= 0.
    UMFPACK_ERROR_nz_negative if nz < 0.
    UMFPACK_ERROR_invalid_triplet if for any k, Ti [k] and/or Tj [k] are not in
        the range 0 to n_row-1 or 0 to n_col-1, respectively.
    UMFPACK_ERROR_out_of_memory if unable to allocate sufficient workspace.

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.
        All row and column indices in the triplet form must be in the range
        0 to n_row-1 and 0 to n_col-1, respectively.

    Int nz ;            Input argument, not modified.

        The number of entries in the triplet form of the matrix.  Restriction:
        nz >= 0.

    Int Ti [nz] ;       Input argument, not modified.
    Int Tj [nz] ;       Input argument, not modified.
    double Tx [nz] ;    Input argument, not modified.
    double Tz [nz] ;    Input argument, not modified, for complex versions.

        Ti, Tj, Tx, and Tz hold the "triplet" form of a sparse matrix.  The kth
        nonzero entry is in row i = Ti [k], column j = Tj [k], and the real part
        of a_ij is Tx [k].  The imaginary part of a_ij is Tz [k], for complex
        versions.  The row and column indices i and j must be in the range 0 to
        n_row-1 and 0 to n_col-1, respectively.  Duplicate entries may be
        present; they are summed in the output matrix.  This is not an error
        condition.  The "triplets" may be in any order.  Tx, Tz, Ax, and Az
        are optional.  For the real version, Ax is computed only if both Ax
        and Tx are present (not (double *) NULL).  For the complex version, Ax
        and Az are computed only if Tx, Tz, Ax, and Az are all present.  These
        are not error conditions; the routine can create just the pattern of
        the output matrix from the pattern of the triplets.

        Future complex version:  if Tx is present and Tz is NULL, then both real
        and imaginary parts will be contained in Tx[0..2*nz-1], with Tx[2*k]
        and Tx[2*k+1] being the real and imaginary part of the kth entry.

    Int Ap [n_col+1] ;  Output argument.

        Ap is an integer array of size n_col+1 on input.  On output, Ap 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], is zero, and Ap [j] <= Ap [j+1] holds for all j in the
        range 0 to n_col-1.  The value nz2 = Ap [n_col] is thus the total
        number of entries in the pattern of the matrix A.  Equivalently, the
        number of duplicate triplets is nz - Ap [n_col].

    Int Ai [nz] ;       Output argument.

        Ai is an integer array of size nz on input.  Note that only the first
        Ap [n_col] entries are used.

        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
        are in ascending order, and no duplicate row indices are present.
        Row indices are in the range 0 to n_col-1 (the matrix is 0-based).

    double Ax [nz] ;    Output argument.
    double Az [nz] ;    Output argument for complex versions.

        Ax and Az (for the complex versions) are double arrays of size nz on
        input.  Note that only the first Ap [n_col] entries are used
        in both arrays.

        Ax is optional; if Tx and/or Ax are not present (a (double *) NULL
        pointer), then Ax is not computed.  Az is also optional; if Tz and/or
        Az are not present, then Az is not computed.  If present, Ax holds the
        numerical values of the the real part of the sparse matrix A and Az
        holds the imaginary parts.  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)].  The imaginary parts are stored in
        Az [(Ap [j]) ... (Ap [j+1]-1)], for the complex versions.

        Future complex version:  if Ax is present and Az is NULL, then both real
        and imaginary parts will be returned in Ax[0..2*nz2-1], with Ax[2*k]
        and Ax[2*k+1] being the real and imaginary part of the kth entry.

    int Map [nz] ;      Optional output argument.

        If Map is present (a non-NULL pointer to an Int array of size nz), then
        on output it holds the position of the triplets in the column-form
        matrix.  That is, suppose p = Map [k], and the k-th triplet is i=Ti[k],
        j=Tj[k], and aij=Tx[k].  Then i=Ai[p], and aij will have been summed
        into Ax[p] (or simply aij=Ax[p] if there were no duplicate entries also
        in row i and column j).  Also, Ap[j] <= p < Ap[j+1].  The Map array is
        not computed if it is (Int *) NULL.  The Map array is useful for
        converting a subsequent triplet form matrix with the same pattern as the
        first one, without calling this routine.  If Ti and Tj do not change,
        then Ap, and Ai can be reused from the prior call to
        umfpack_*_triplet_to_col.  You only need to recompute Ax (and Az for the
        complex version).  This code excerpt properly sums up all duplicate
        values (for the real version):

            for (p = 0 ; p < Ap [n_col] ; p++) Ax [p] = 0 ;
            for (k = 0 ; k < nz ; k++) Ax [Map [k]] += Tx [k] ;

        This feature is useful (along with the reuse of the Symbolic object) if
        you need to factorize a sequence of triplet matrices with identical
        nonzero pattern (the order of the triplets in the Ti,Tj,Tx arrays must
        also remain unchanged).  It is faster than calling this routine for
        each matrix, and requires no workspace.
```

12.3 umfpack_*_transpose

```
int umfpack_di_transpose
(
    int n_row,
    int n_col,
    const int Ap [ ],
    const int Ai [ ],
    const double Ax [ ],
    const int P [ ],
    const int Q [ ],
    int Rp [ ],
    int Ri [ ],
    double Rx [ ]
) ;

long umfpack_dl_transpose
(
    long n_row,
    long n_col,
    const long Ap [ ],
    const long Ai [ ],
    const double Ax [ ],
    const long P [ ],
    const long Q [ ],
    long Rp [ ],
    long Ri [ ],
    double Rx [ ]
) ;

int umfpack_zi_transpose
(
    int n_row,
    int n_col,
    const int Ap [ ],
    const int Ai [ ],
    const double Ax [ ], const double Az [ ],
    const int P [ ],
    const int Q [ ],
    int Rp [ ],
    int Ri [ ],
    double Rx [ ], double Rz [ ],
    int do_conjugate
) ;

long umfpack_zl_transpose
(
    long n_row,
    long n_col,
    const long Ap [ ],
    const long Ai [ ],
    const double Ax [ ], const double Az [ ],
    const long P [ ],
    const long Q [ ],
    long Rp [ ],
    long Ri [ ],
    double Rx [ ], double Rz [ ],
    long do_conjugate
) ;

double int Syntax:

    #include "umfpack.h"
    int n_row, n_col, status, *Ap, *Ai, *P, *Q, *Rp, *Ri ;
    double *Ax, *Rx ;
    status = umfpack_di_transpose (n_row, n_col, Ap, Ai, Ax, P, Q, Rp, Ri, Rx) ;

double long Syntax:

    #include "umfpack.h"
    long n_row, n_col, status, *Ap, *Ai, *P, *Q, *Rp, *Ri ;
    double *Ax, *Rx ;
    status = umfpack_dl_transpose (n_row, n_col, Ap, Ai, Ax, P, Q, Rp, Ri, Rx) ;

complex int Syntax:

    #include "umfpack.h"
    int n_row, n_col, status, *Ap, *Ai, *P, *Q, *Rp, *Ri, do_conjugate ;
    double *Ax, *Az, *Rx, *Rz ;
    status = umfpack_zi_transpose (n_row, n_col, Ap, Ai, Ax, Az, P, Q,
        Rp, Ri, Rx, Rz, do_conjugate) ;

complex long Syntax:

    #include "umfpack.h"
    long n_row, n_col, status, *Ap, *Ai, *P, *Q, *Rp, *Ri, do_conjugate ;
    double *Ax, *Az, *Rx, *Rz ;
    status = umfpack_zl_transpose (n_row, n_col, Ap, Ai, Ax, Az, P, Q,
        Rp, Ri, Rx, Rz, do_conjugate) ;

Purpose:

    Transposes and optionally permutes a sparse matrix in row or column-form,
    R = (PAQ)'.  In MATLAB notation, R = (A (P,Q))' or R = (A (P,Q)).' doing
    either the linear algebraic transpose or the array transpose. Alternatively,
    this routine can be viewed as converting A (P,Q) from column-form to
    row-form, or visa versa (for the array transpose).  Empty rows and columns
    may exist.  The matrix A may be singular and/or rectangular.

    umfpack_*_transpose is useful if you want to factorize A' or A.' instead of
    A.  Factorizing A' or A.' instead of A can be much better, particularly if
    AA' is much sparser than A'A.  You can still solve Ax=b if you factorize
    A' or A.', by solving with the sys argument UMFPACK_At or UMFPACK_Aat,
    respectively, in umfpack_*_*solve.  The umfpack mexFunction (umfpackmex.c)
    is one example.  To compute x = A/b, it computes x = (A.'\b.').' instead,
    by factorizing A.'.  It then uses the regular solve, since b.' and x.' are
    stored identically as b and x, respectively (both b.' and b are dense
    vectors).  If b and x were arrays, the umfpack mexFunction would need to
    first compute b.' and then transpose the resulting solution.

Returns:

    UMFPACK_OK if successful.
    UMFPACK_ERROR_out_of_memory if umfpack_*_transpose fails to allocate a
        size-max (n_row,n_col) workspace.
    UMFPACK_ERROR_argument_missing if Ai, Ap, Ri, and/or Rp are missing.
    UMFPACK_ERROR_n_nonpositive if n_row <= 0 or n_col <= 0
    UMFPACK_ERROR_invalid_permutation if P and/or Q are invalid.
    UMFPACK_ERROR_nz_negative if Ap [n_col] < 0.
    UMFPACK_ERROR_Ap0_nonzero if Ap [0] != 0.
    UMFPACK_ERROR_col_length_negative if Ap [j] > Ap [j+1] for any j in the
        range 0 to n_col-1.
    UMFPACK_ERROR_row_index_out_of_bounds if any row index i is < 0 or >= n_row.
    UMFPACK_ERROR_jumbled_matrix if the row indices in any column are not in
        ascending order.

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.

        The column pointers of the column-oriented form of the matrix A.  See
        umfpack_*_symbolic for a description.  The number of entries in
        the matrix is nz = Ap [n_col].  Ap [0] must be zero, Ap [n_col] must be
        => 0, and Ap [j] <= Ap [j+1] and Ap [j] <= Ap [n_col] must be true for
        all j in the range 0 to n_col-1.  Empty columns are OK (that is, Ap [j]
        may equal Ap [j+1] for any j in the range 0 to n_col-1).

    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).

    double Ax [nz] ;    Input argument, not modified, of size nz = Ap [n_col].
    double Az [nz] ;    Input argument, not modified, for complex versions.

        If present, these are 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 real numerical
        values are stored in Ax [(Ap [j]) ... (Ap [j+1]-1)].  The imaginary
        values are stored in Az [(Ap [j]) ... (Ap [j+1]-1)].  The values are
        transposed only if Ax and Rx are present (for the real version), and
        only if all four (Ax, Az, Rx, and Rz) are present for the complex
        version.  These are not an error conditions; you are able to transpose
        and permute just the pattern of a matrix.

        Future complex version:  if Ax is present and Az is NULL, then both real
        and imaginary parts will be 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.

    Int P [n_row] ;             Input argument, not modified.

        The permutation vector P is defined as P [k] = i, where the original
        row i of A is the kth row of PAQ.  If you want to use the identity
        permutation for P, simply pass (Int *) NULL for P.  This is not an error
        condition.  P is a complete permutation of all the rows of A; this
        routine does not support the creation of a transposed submatrix of A
        (R = A (1:3,:)' where A has more than 3 rows, for example, cannot be
        done; a future version might support this operation).

    Int Q [n_col] ;             Input argument, not modified.

        The permutation vector Q is defined as Q [k] = j, where the original
        column j of A is the kth column of PAQ.  If you want to use the identity
        permutation for Q, simply pass (Int *) NULL for Q.  This is not an error
        condition.  Q is a complete permutation of all the columns of A; this
        routine does not support the creation of a transposed submatrix of A.

    Int Rp [n_row+1] ;  Output argument.

        The column pointers of the matrix R = (A (P,Q))' or (A (P,Q)).', in the
        same form as the column pointers Ap for the matrix A.

    Int Ri [nz] ;       Output argument.

        The row indices of the matrix R = (A (P,Q))' or (A (P,Q)).' , in the
        same form as the row indices Ai for the matrix A.

    double Rx [nz] ;    Output argument.
    double Rz [nz] ;    Output argument, imaginary part for complex versions.

        If present, these are the numerical values of the sparse matrix R,
        in the same form as the values Ax and Az of the matrix A.

        Future complex version:  if Rx is present and Rz is NULL, then both real
        and imaginary parts will be contained in Rx[0..2*nz-1], with Rx[2*k]
        and Rx[2*k+1] being the real and imaginary part of the kth entry.

    Int do_conjugate ;  Input argument for complex versions only.

        If true, and if Ax, Az, Rx, and Rz are all present, then the linear
        algebraic transpose is computed (complex conjugate).  If false, the
        array transpose is computed instead.
```

13 Getting the contents of opaque objects

13.1 umfpack_*_get_lunz

```
int umfpack_di_get_lunz
(
    int *lnz,
    int *unz,
    int *n_row,
    int *n_col,
    int *nz_udiag,
    void *Numeric
) ;

long umfpack_dl_get_lunz
(
    long *lnz,
    long *unz,
    long *n_row,
    long *n_col,
    long *nz_udiag,
    void *Numeric
) ;

int umfpack_zi_get_lunz
(
    int *lnz,
    int *unz,
    int *n_row,
    int *n_col,
    int *nz_udiag,
    void *Numeric
) ;

long umfpack_zl_get_lunz
(
    long *lnz,
    long *unz,
    long *n_row,
    long *n_col,
    long *nz_udiag,
    void *Numeric
) ;

double int Syntax:

    #include "umfpack.h"
    void *Numeric ;
    int status, lnz, unz, n_row, n_col ;
    status = umfpack_di_get_lunz (&lnz, &unz, &n_row, &n_col, Numeric) ;
```

```
double long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    long status, lnz, unz, n_row, n_col ;
    status = umfpack_dl_get_lunz (&lnz, &unz, &n_row, &n_col, Numeric) ;

complex int Syntax:

    #include "umfpack.h"
    void *Numeric ;
    int status, lnz, unz, n_row, n_col ;
    status = umfpack_zi_get_lunz (&lnz, &unz, &n_row, &n_col, Numeric) ;

complex long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    long status, lnz, unz, n_row, n_col ;
    status = umfpack_zl_get_lunz (&lnz, &unz, &n_row, &n_col, Numeric) ;

Purpose:

    Determines the size and number of nonzeros in the LU factors held by the
    Numeric object.  These are also the sizes of the output arrays required
    by umfpack_*_get_numeric.

    The matrix L is n_row -by- min(n_row,n_col), with lnz nonzeros, including
    the entries on the unit diagonal of L.

    The matrix U is min(n_row,n_col) -by- n_col, with unz nonzeros, including
    nonzeros on the diagonal of U.

Returns:

    UMFPACK_OK if successful.
    UMFPACK_ERROR_invalid_Numeric_object if Numeric is not a valid object.
    UMFPACK_ERROR_argument_missing if any other argument is (Int *) NULL.

Arguments:

    Int *lnz ;          Output argument.

        The number of nonzeros in L, including the diagonal (which is all
        one's).  This value is the required size of the Lj and Lx arrays as
        computed by umfpack_*_get_numeric.  The value of lnz is identical to
        Info [UMFPACK_LNZ], if that value was returned by umfpack_*_numeric.

    Int *unz ;          Output argument.

        The number of nonzeros in U, including the diagonal.  This value is the
        required size of the Ui and Ux arrays as computed by
        umfpack_*_get_numeric.  The value of unz is identical to
        Info [UMFPACK_UNZ], if that value was returned by umfpack_*_numeric.

    Int *n_row ;        Output argument.
    Int *n_col ;        Output argument.

        The order of the L and U matrices.  L is n_row -by- min(n_row,n_col)
        and U is min(n_row,n_col) -by- n_col.

    Int *nz_udiag ;     Output argument.

        The number of numerically nonzero values on the diagonal of U.  The
        matrix is singular if nz_diag < min(n_row,n_col).  A divide-by-zero
        will occur if nz_diag < n_row == n_col when solving a sparse system
        involving the matrix U in umfpack_*_*solve.  The value of nz_udiag is
        identical to Info [UMFPACK_UDIAG_NZ] if that value was returned by
        umfpack_*_numeric.

    void *Numeric ;     Input argument, not modified.

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

13.2 umfpack_*_get_numeric

```
int umfpack_di_get_numeric
(
    int Lp [ ],
    int Lj [ ],
    double Lx [ ],
    int Up [ ],
    int Ui [ ],
    double Ux [ ],
    int P [ ],
    int Q [ ],
    double Dx [ ],
    void *Numeric
) ;

long umfpack_dl_get_numeric
(
    long Lp [ ],
    long Lj [ ],
    double Lx [ ],
    long Up [ ],
    long Ui [ ],
    double Ux [ ],
    long P [ ],
    long Q [ ],
    double Dx [ ],
    void *Numeric
) ;

int umfpack_zi_get_numeric
(
    int Lp [ ],
    int Lj [ ],
    double Lx [ ], double Lz [ ],
    int Up [ ],
    int Ui [ ],
    double Ux [ ], double Uz [ ],
    int P [ ],
    int Q [ ],
    double Dx [ ], double Dz [ ],
    void *Numeric
) ;

long umfpack_zl_get_numeric
(
    long Lp [ ],
    long Lj [ ],
    double Lx [ ], double Lz [ ],
    long Up [ ],
    long Ui [ ],
    double Ux [ ], double Uz [ ],
    long P [ ],
    long Q [ ],
    double Dx [ ], double Dz [ ],
    void *Numeric
) ;

double int Syntax:

    #include "umfpack.h"
    void *Numeric ;
    int *Lp, *Lj, *Up, *Ui, *P, *Q, status ;
    double *Lx, *Ux, *Dx ;
    status = umfpack_di_get_numeric (Lp, Lj, Lx, Up, Ui, Ux, P, Q, Dx, Numeric);

double long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    long *Lp, *Lj, *Up, *Ui, *P, *Q, status ;
    double *Lx, *Ux, *Dx ;
    status = umfpack_dl_get_numeric (Lp, Lj, Lx, Up, Ui, Ux, P, Q, Dx, Numeric);

complex int Syntax:

    #include "umfpack.h"
    void *Numeric ;
    int *Lp, *Lj, *Up, *Ui, *P, *Q, status ;
    double *Lx, *Lz, *Ux, *Uz, *Dx, *Dz ;
    status = umfpack_zi_get_numeric (Lp, Lj, Lx, Lz, Up, Ui, Ux, Uz, P, Q,
        Dx, Dz, Numeric) ;

complex long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    long *Lp, *Lj, *Up, *Ui, *P, *Q, status ;
    double *Lx, *Lz, *Ux, *Uz, *Dx, *Dz ;
    status = umfpack_zl_get_numeric (Lp, Lj, Lx, Lz, Up, Ui, Ux, Uz, P, Q,
        Dx, Dz, Numeric) ;

Purpose:

    This routine copies the LU factors and permutation vectors from the Numeric
    object into user-accessible arrays.  This routine is not needed to solve a
    linear system.  Note that the output arrays Lp, Lj, Lx, Up, Ui, Ux, P, Q,
    and Dx are not allocated by umfpack_*_get_numeric; they must exist on input.
    Similarly, Lz, Uz and Dz must exist on input for the complex versions.

Returns:

    Returns UMFPACK_OK if successful.  Returns UMFPACK_ERROR_out_of_memory
    if insufficient memory is available for the 2*max(n_row,n_col) integer
    workspace that umfpack_*_get_numeric allocates to construct L and/or U.
    Returns UMFPACK_ERROR_invalid_Numeric_object if the Numeric object provided
    as input is invalid.

Arguments:

    Int Lp [n_row+1] ;  Output argument.
    Int Lj [lnz] ;      Output argument.
    double Lx [lnz] ;   Output argument.
    double Lz [lnz] ;   Output argument for complex versions.

        The n_row-by-min(n_row,n_col) matrix L is returned in compressed-row
        form.  The column indices of row i and corresponding numerical values
        are in:

            Lj [Lp [i] ... Lp [i+1]-1]
            Lx [Lp [i] ... Lp [i+1]-1]  real part
            Lz [Lp [i] ... Lp [i+1]-1]  imaginary part (complex versions)

        respectively.  Each row is stored in sorted order, from low column
        indices to higher.  The last entry in each row is the diagonal, which
        is numerically equal to one.  The sizes of Lp, Lj, Lx, and Lz are
        returned by umfpack_*_get_lunz.    If Lp, Lj, or Ux (or Uz for the
        complex version) are not present, then the matrix L is not returned.
        This is not an error condition.  The L matrix can be printed if n_row,
        Lp, Lj, Lx (and Lz for the complex versions) are passed to
        umfpack_*_report_matrix (using the "row" form).

        Future complex version:  if Lx is present and Lz is NULL, then both real
        and imaginary parts will be returned in Lx[0..2*lnz-1], with Lx[2*k]
        and Lx[2*k+1] being the real and imaginary part of the kth entry.

    Int Up [n_col+1] ;  Output argument.
    Int Ui [unz] ;      Output argument.
    double Ux [unz] ;   Output argument.
    double Uz [unz] ;   Output argument for complex versions.

        The min(n_row,n_co)-by-n_col matrix U is returned in compressed-column
        form.  The row indices of column j and corresponding numerical values
        are in

            Ui [Up [j] ... Up [j+1]-1]
            Ux [Up [j] ... Up [j+1]-1]  real part
            Uz [Up [j] ... Up [j+1]-1]  imaginary part (complex versions)

        respectively.  Each column is stored in sorted order, from low row
        indices to higher.  The last entry in each column is the diagonal
        (assuming that it is nonzero).  The sizes of Up, Ui, Ux, and Uz are
        returned by umfpack_*_get_lunz.  If Up, Ui, or Ux (or Uz for the complex
        version) are not present, then the matrix U is not returned.  This is
        not an error condition.  The U matrix can be printed if n_col, Up, Ui,
        Ux (and Uz for the complex versions) are passed to
        umfpack_*_report_matrix (using the "column" form).
        Future complex version:  if Ux is present and Uz is NULL, then both real
        and imaginary parts will be returned in Ux[0..2*unz-1], with Ux[2*k]
        and Ux[2*k+1] being the real and imaginary part of the kth entry.

    Int P [n_row] ;             Output argument.

        The permutation vector P is defined as P [k] = i, where the original
        row i of A is the kth pivot row in PAQ.  If you do not want the P vector
        to be returned, simply pass (Int *) NULL for P.  This is not an error
        condition.  You can print P and Q with umfpack_*_report_perm.

    Int Q [n_col] ;             Output argument.

        The permutation vector Q is defined as Q [k] = j, where the original
        column j of A is the kth pivot column in PAQ.  If you not want the Q
        vector to be returned, simply pass (Int *) NULL for Q.  This is not
        an error condition.  Note that Q is not necessarily identical to
        Qtree, the column preordering held in the Symbolic object.  Refer to
        the description of Qtree and Front_npivcol in umfpack_*_get_symbolic for
        details.

    double Dx [min(n_row,n_col)] ;      Output argument.
    double Dz [min(n_row,n_col)] ;      Output argument for complex versions.

        The diagonal of U is also returned in Dx and Dz.  You can extract the
        diagonal of U without getting all of U by passing a non-NULL Dx (and
        Dz for the complex version) and passing Up, Ui, and Ux as NULL.  Dx is
        the real part of the diagonal, and Dz is the imaginary part.

        Future complex version:  if Dx is present and Dz is NULL, then both real
        and imaginary parts will be returned in Dx[0..2*min(n_row,n_col)-1],
        with Dx[2*k] and Dx[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.
```

13.3 umfpack_*_get_symbolic

```
int umfpack_di_get_symbolic
(
    int *n_row,
    int *n_col,
    int *nz,
    int *nfr,
    int *nchains,
    int Ptree [ ],
    int Qtree [ ],
    int Front_npivcol [ ],
    int Front_parent [ ],
    int Front_1strow [ ],
    int Front_leftmostdesc [ ],
    int Chain_start [ ],
    int Chain_maxrows [ ],
    int Chain_maxcols [ ],
    void *Symbolic
) ;

long umfpack_dl_get_symbolic
(
    long *n_row,
    long *n_col,
    long *nz,
    long *nfr,
    long *nchains,
    long Ptree [ ],
    long Qtree [ ],
    long Front_npivcol [ ],
    long Front_parent [ ],
    long Front_1strow [ ],
    long Front_leftmostdesc [ ],
    long Chain_start [ ],
    long Chain_maxrows [ ],
    long Chain_maxcols [ ],
    void *Symbolic
) ;

int umfpack_zi_get_symbolic
(
    int *n_row,
    int *n_col,
    int *nz,
    int *nfr,
    int *nchains,
    int Ptree [ ],
    int Qtree [ ],
    int Front_npivcol [ ],
    int Front_parent [ ],
    int Front_1strow [ ],
    int Front_leftmostdesc [ ],
    int Chain_start [ ],
    int Chain_maxrows [ ],
    int Chain_maxcols [ ],
    void *Symbolic
) ;

long umfpack_zl_get_symbolic
(
    long *n_row,
    long *n_col,
    long *nz,
    long *nfr,
    long *nchains,
    long Ptree [ ],
    long Qtree [ ],
    long Front_npivcol [ ],
    long Front_parent [ ],
    long Front_1strow [ ],
    long Front_leftmostdesc [ ],
    long Chain_start [ ],
    long Chain_maxrows [ ],
    long Chain_maxcols [ ],
    void *Symbolic
) ;

double int Syntax:

    #include "umfpack.h"
    int status, n_row, n_col, nz, nfr, nchains, *Ptree, *Qtree,
        *Front_npivcol, *Front_parent, *Front_1strow, *Front_leftmostdesc,
        *Chain_start, *Chain_maxrows, *Chain_maxcols ;
    void *Symbolic ;
    status = umfpack_di_get_symbolic (&n_row, &n_col, &nz, &nfr, &nchains,
        Ptree, Qtree, Front_npivcol, Front_parent, Front_1strow,
        Front_leftmostdesc, Chain_start, Chain_maxrows, Chain_maxcols,
        Symbolic) ;

double long Syntax:

    #include "umfpack.h"
    long status, n_row, n_col, nz, nfr, nchains, *Ptree, *Qtree,
        *Front_npivcol, *Front_parent, *Front_1strow, *Front_leftmostdesc,
        *Chain_start, *Chain_maxrows, *Chain_maxcols ;
    void *Symbolic ;
    status = umfpack_dl_get_symbolic (&n_row, &n_col, &nz, &nfr, &nchains,
        Ptree, Qtree, Front_npivcol, Front_parent, Front_1strow,
        Front_leftmostdesc, Chain_start, Chain_maxrows, Chain_maxcols,
        Symbolic) ;

complex int Syntax:
    #include "umfpack.h"
    int status, n_row, n_col, nz, nfr, nchains, *Ptree, *Qtree,
        *Front_npivcol, *Front_parent, *Front_1strow, *Front_leftmostdesc,
        *Chain_start, *Chain_maxrows, *Chain_maxcols ;
    void *Symbolic ;
    status = umfpack_zi_get_symbolic (&n_row, &n_col, &nz, &nfr, &nchains,
        Ptree, Qtree, Front_npivcol, Front_parent, Front_1strow,
        Front_leftmostdesc, Chain_start, Chain_maxrows, Chain_maxcols,
        Symbolic) ;

complex long Syntax:

    #include "umfpack.h"
    long status, n_row, n_col, nz, nfr, nchains, *Ptree, *Qtree,
        *Front_npivcol, *Front_parent, *Front_1strow, *Front_leftmostdesc,
        *Chain_start, *Chain_maxrows, *Chain_maxcols ;
    void *Symbolic ;
    status = umfpack_zl_get_symbolic (&n_row, &n_col, &nz, &nfr, &nchains,
        Ptree, Qtree, Front_npivcol, Front_parent, Front_1strow,
        Front_leftmostdesc, Chain_start, Chain_maxrows, Chain_maxcols,
        Symbolic) ;

Purpose:

    Copies the contents of the Symbolic object into simple integer arrays
    accessible to the user.  This routine is not needed to factorize and/or
    solve a sparse linear system using UMFPACK.  Note that the output arrays
    Ptree, Qtree, Front_npivcol, Front_parent, Front_1strow, Front_leftmostdesc,
    Chain_start, Chain_maxrows, and Chain_maxcols are not allocated by
    umfpack_*_get_symbolic; they must exist on input.

    The Symbolic object is small.  Its size for an n-by-n square matrix is about
    (3*nchains + 2*n + 4*nfr + 30) integers plus 5 double’s, which is no greater
    than about 9*n+40 int’s.  The object holds the initial column permutation,
    the supernodal column elimination tree, and information about each frontal
    matrix.  You can print it with umfpack_*_report_symbolic.

Returns:

    Returns UMFPACK_OK if successful, UMFPACK_ERROR_invalid_Symbolic_object
    if Symbolic is an invalid object.

Arguments:

    Note that if any of the output arguments are (Int *) NULL pointers, then
    that argument is not returned.  This is not an error condition.  Thus,
    if you do not want a particular part of the Symbolic object to be
    returned to you, simply pass a (Int *) NULL pointer for that particular
    output argument.

    Int *n_row ;                Output argument.
    Int *n_col ;                Output argument.
        The dimensions of the matrix A analyzed by the call to
        umfpack_*_symbolic that generated the Symbolic object.

    Int *nz ;           Output argument.

        The number of nonzeros in A.

    Int *nfr ;  Output argument.

        The number of frontal matrices that will be used by umfpack_*_numeric
        to factorize the matrix A.  It is in the range 0 to n_col.

    Int *nchains ;      Output argument.

        The frontal matrices are related to one another by the supernodal
        column elimination tree.  Each node in this tree is one frontal matrix.
        The tree is partitioned into a set of disjoint paths, and a frontal
        matrix chain is one path in this tree.  Each chain is factorized using
        a unifrontal technique, with a single working array that holds each
        frontal matrix in the chain, one at a time.  nchains is in the range
        0 to nfr.

    Int Ptree [n_row] ; Output argument.

        The initial row permutation.  If Ptree [k] = i, then this means that
        row i is the kth row in the preordered matrix.  Row i typically will
        not be the kth pivot row, however.  Ptree is the result of a sorting of
        the rows according to the smallest column index of entries in the
        matrix permuted by Qtree, below.

    Int Qtree [n_col] ; Output argument.

        The initial column permutation.  If Qtree [k] = j, then this means that
        column j is the kth pivot column in the preordered matrix.  Qtree is
        not necessarily the same as the final column permutation Q, computed by
        umfpack_*_numeric.  The numeric factorization may reorder the pivot
        columns within each frontal matrix to reduce fill-in.

    Int Front_npivcol [n_col+1] ;       Output argument.

        This array should be of size at least n_col+1, in order to guarantee
        that it will be large enough to hold the output.  Only the first nfr+1
        entries are used, however.

        The kth frontal matrix holds Front_npivcol [k] pivot columns.  Thus, the
        first frontal matrix, front 0, is used to factorize the first
        Front_npivcol [0] columns; these correspond to the original columns
        Qtree [0] through Qtree [Front_npivcol [0]-1].  The next frontal matrix
        is used to factorize the next Front_npivcol [1] columns, which are thus
        the original columns Qtree [Front_npivcol [0]] through
        Qtree [Front_npivcol [0] + Front_npivcol [1] - 1], and so on.  Columns
        with no entries at all are put in a placeholder "front",
        Front_npivcol [nfr].  The sum of Front_npivcol [0..nfr] is equal to
        n_col.

        Any modifications that umfpack_*_numeric makes to the initial column
        permutation are constrained to within each frontal matrix.  Thus, for
        the first frontal matrix, Q [0] through Q [Front_npivcol [0]-1] is some
        permutation of the columns Qtree [0] through
        Qtree [Front_npivcol [0]-1].  For second frontal matrix,
        Q [Front_npivcol [0]] through Q [Front_npivcol [0] + Front_npivcol[1]-1]
        is some permutation of the same portion of Qtree, and so on.  All pivot
        columns are numerically factorized within the frontal matrix originally
        determined by the symbolic factorization; there is no delayed pivoting
        across frontal matrices.

    Int Front_parent [n_col+1] ;        Output argument.

        This array should be of size at least n_col+1, in order to guarantee
        that it will be large enough to hold the output.  Only the first nfr+1
        entries are used, however.

        Front_parent [0..nfr] holds the supernodal column elimination tree
        (including the placeholder front nfr, which may be empty).  Each node in
        the tree corresponds to a single frontal matrix.  The parent of node f
        is Front_parent [f].

    Int Front_1strow [n_col+1] ;        Output argument.

        This array should be of size at least n_col+1, in order to guarantee
        that it will be large enough to hold the output.  Only the first nfr+1
        entries are used, however.

        Front_1strow [k] is the row index of the first row in A (Ptree,Qtree)
        whose leftmost entry is in a pivot column for the kth front.  This is
        necessary only to properly factorize singular matrices.  It is new to
        Version 4.0.  Rows in the range Front_1strow [k] to
        Front_1strow [k+1]-1 first become pivot row candidates at the kth front.
        Any rows not eliminated in the kth front may be selected as pivot rows
        in the parent of k (Front_parent [k]) and so on up the tree.

    Int Front_leftmostdesc [n_col+1] ;  Output argument.

        This array should be of size at least n_col+1, in order to guarantee
        that it will be large enough to hold the output.  Only the first nfr+1
        entries are used, however.

        Front_leftmostdesc [k] is the leftmost descendant of front k, or k
        if the front has no children in the tree.  Since the rows and columns
        (Ptree and Qtree) have been post-ordered via a depth-first-search of
        the tree, rows in the range Front_1strow [Front_leftmostdesc [k]] to
        Front_1strow [k+1]-1 form the entire set of candidate pivot rows for
        the kth front (some of these will typically have already been selected
        by fronts in the range Front_leftmostdesc [k] to front k-1, before
        the factorization reaches front k).
```

```
Chain_start [n_col+1] ;     Output argument.

    This array should be of size at least n_col+1, in order to guarantee
    that it will be large enough to hold the output.  Only the first
    nchains+1 entries are used, however.

    The kth frontal matrix chain consists of frontal matrices Chain_start[k]
    through Chain_start [k+1]-1.  Thus, Chain_start [0] is always 0, and
    Chain_start [nchains] is the total number of frontal matrices, nfr.  For
    two adjacent fronts f and f+1 within a single chain, f+1 is always the
    parent of f (that is, Front_parent [f] = f+1).

Int Chain_maxrows [n_col+1] ;       Output argument.
Int Chain_maxcols [n_col+1] ;       Output argument.

    These arrays should be of size at least n_col+1, in order to guarantee
    that they will be large enough to hold the output.  Only the first
    nchains entries are used, however.

    The kth frontal matrix chain requires a single working array of
    dimension Chain_maxrows [k] by Chain_maxcols [k], for the unifrontal
    technique that factorizes the frontal matrix chain.  Since the symbolic
    factorization only provides an upper bound on the size of each frontal
    matrix, not all of the working array is necessarily used during the
    numerical factorization.

    Note that the upper bound on the number of rows and columns of each
    frontal matrix is computed by umfpack_*_symbolic, but all that is
    required by umfpack_*_numeric is the maximum of these two sets of
    values for each frontal matrix chain.  Thus, the size of each
    individual frontal matrix is not preserved in the Symbolic object.

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_*_get_symbolic.
```

14 Reporting routines

14.1 umfpack_*_report_status

```
void umfpack_di_report_status
(
    const double Control [UMFPACK_CONTROL],
    int status
) ;

void umfpack_dl_report_status
(
    const double Control [UMFPACK_CONTROL],
    long status
) ;

void umfpack_zi_report_status
(
    const double Control [UMFPACK_CONTROL],
    int status
) ;

void umfpack_zl_report_status
(
    const double Control [UMFPACK_CONTROL],
    long status
) ;

double int Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL] ;
    int status ;
    umfpack_di_report_status (Control, status) ;

double long Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL] ;
    long status ;
    umfpack_dl_report_status (Control, status) ;

complex int Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL] ;
    int status ;
    umfpack_zi_report_status (Control, status) ;

complex long Syntax:
    #include "umfpack.h"
    double Control [UMFPACK_CONTROL] ;
    long status ;
    umfpack_zl_report_status (Control, status) ;

Purpose:

    Prints the status (return value) of other umfpack_* routines.

Arguments:

    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_PRL]:  printing level.

            0 or less: no output, even when an error occurs
            1: error messages only
            2 or more: print status, whether or not an error occured
            4 or more: also print the UMFPACK Copyright
            6 or more: also print the UMFPACK License
            Default: 1

    Int status ;                        Input argument, not modified.

        The return value from another umfpack_* routine.
```

14.2 umfpack_*_report_control

```
void umfpack_di_report_control
(
    const double Control [UMFPACK_CONTROL]
) ;

void umfpack_dl_report_control
(
    const double Control [UMFPACK_CONTROL]
) ;

void umfpack_zi_report_control
(
    const double Control [UMFPACK_CONTROL]
) ;

void umfpack_zl_report_control
(
    const double Control [UMFPACK_CONTROL]
) ;

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

double long Syntax:

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

Purpose:

    Prints the current control settings.  Note that with the default print
    level, nothing is printed.  Does nothing if Control is (double *) NULL.
```

```
Arguments:

    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_PRL]:  printing level.

            1 or less: no output
            2 or more: print all of Control
            Default: 1
```

14.3 umfpack_*_report_info

```
void umfpack_di_report_info
(
    const double Control [UMFPACK_CONTROL],
    const double Info [UMFPACK_INFO]
) ;

void umfpack_dl_report_info
(
    const double Control [UMFPACK_CONTROL],
    const double Info [UMFPACK_INFO]
) ;

void umfpack_zi_report_info
(
    const double Control [UMFPACK_CONTROL],
    const double Info [UMFPACK_INFO]
) ;

void umfpack_zl_report_info
(
    const double Control [UMFPACK_CONTROL],
    const double Info [UMFPACK_INFO]
) ;

double int Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
    umfpack_di_report_info (Control, Info) ;

double long Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
    umfpack_dl_report_info (Control, Info) ;

complex int Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
    umfpack_zi_report_info (Control, Info) ;

complex long Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
    umfpack_zl_report_info (Control, Info) ;

Purpose:
    Reports statistics from the umfpack_*_*symbolic, umfpack_*_numeric, and
    umfpack_*_*solve routines.

Arguments:

    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_PRL]:  printing level.

            0 or less: no output, even when an error occurs
            1: error messages only
            2 or more: error messages, and print all of Info
            Default: 1

    double Info [UMFPACK_INFO] ;                Input argument, not modified.

        Info is an output argument of several UMFPACK routines.
        The contents of Info are printed on standard output.
```

14.4 umfpack_*_report_matrix

```
int umfpack_di_report_matrix
(
    int n_row,
    int n_col,
    const int Ap [ ],
    const int Ai [ ],
    const double Ax [ ],
    int col_form,
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_dl_report_matrix
(
    long n_row,
    long n_col,
    const long Ap [ ],
    const long Ai [ ],
    const double Ax [ ],
    long col_form,
    const double Control [UMFPACK_CONTROL]
) ;

int umfpack_zi_report_matrix
(
    int n_row,
    int n_col,
    const int Ap [ ],
    const int Ai [ ],
    const double Ax [ ], const double Az [ ],
    int col_form,
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_zl_report_matrix
(
    long n_row,
    long n_col,
    const long Ap [ ],
    const long Ai [ ],
    const double Ax [ ], const double Az [ ],
    long col_form,
    const double Control [UMFPACK_CONTROL]
) ;

double int Syntax:

    #include "umfpack.h"
    int n_row, n_col, *Ap, *Ai, status ;
    double *Ax, Control [UMFPACK_CONTROL] ;
    status = umfpack_di_report_matrix (n_row, n_col, Ap, Ai, Ax, 1, Control) ;
or:
    status = umfpack_di_report_matrix (n_row, n_col, Ap, Ai, Ax, 0, Control) ;

double long Syntax:

    #include "umfpack.h"
    long n_row, n_col, *Ap, *Ai, status ;
    double *Ax, Control [UMFPACK_CONTROL] ;
    status = umfpack_dl_report_matrix (n_row, n_col, Ap, Ai, Ax, 1, Control) ;
or:
    status = umfpack_dl_report_matrix (n_row, n_col, Ap, Ai, Ax, 0, Control) ;

complex int Syntax:

    #include "umfpack.h"
    int n_row, n_col, *Ap, *Ai, status ;
    double *Ax, *Az, Control [UMFPACK_CONTROL] ;
    status = umfpack_zi_report_matrix (n_row, n_col, Ap, Ai, Ax, Az, 1,
        Control) ;
or:
    status = umfpack_zi_report_matrix (n_row, n_col, Ap, Ai, Ax, Az, 0,
        Control) ;

complex long Syntax:

    #include "umfpack.h"
    long n_row, n_col, *Ap, *Ai, status ;
    double *Ax, Control [UMFPACK_CONTROL] ;
    status = umfpack_zl_report_matrix (n_row, n_col, Ap, Ai, Ax, Az, 1,
        Control) ;
or:
    status = umfpack_zl_report_matrix (n_row, n_col, Ap, Ai, Ax, Az, 0,
        Control) ;

Purpose:

    Verifies and prints a row or column-oriented sparse matrix.

Returns:

    UMFPACK_OK if Control [UMFPACK_PRL] <= 2 (the input is not checked).

    Otherwise (where n is n_col for the column form and n_row for row
    and let ni be n_row for the column form and n_col for row):

    UMFPACK_OK if the matrix is valid.

    UMFPACK_ERROR_n_nonpositive if n_row <= 0 or n_col <= 0.
    UMFPACK_ERROR_argument_missing if Ap and/or Ai are missing.
    UMFPACK_ERROR_nz_negative if Ap [n] < 0.
    UMFPACK_ERROR_Ap0_nonzero if Ap [0] is not zero.
    UMFPACK_ERROR_col_length_negative if Ap [j+1] < Ap [j] for any j in the
```

```
        range 0 to n-1.
    UMFPACK_ERROR_out_of_memory if out of memory.
    UMFPACK_ERROR_row_index_out_of_bounds if any row index in Ai is not in
        the range 0 to ni-1.
    UMFPACK_ERROR_jumbled_matrix if the row indices in any column are not in
        ascending order, or contain duplicates.

Arguments:

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

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

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

        n is n_row for a row-form matrix, and n_col for a column-form matrix.

        Ap is an integer array of size n+1.  If col_form is true (nonzero),
        then on input, it holds the "pointers" for the column form of the
        sparse matrix A.  The row indices of column j of the matrix A are held
        in Ai [(Ap [j]) ... (Ap [j+1]-1)].  Otherwise, Ap holds the
        row pointers, and the column indices of row j of the matrix are 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-1.  The value nz = Ap [n] is thus the
        total number of entries in the pattern of the matrix A.

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

        If col_form is true (nonzero), then the nonzero pattern (row indices)
        for column j is stored in Ai [(Ap [j]) ... (Ap [j+1]-1)].  Row indices
        must be in the range 0 to n_row-1 (the matrix is 0-based).

        Otherwise, the nonzero pattern (column indices) for row j is stored in
        Ai [(Ap [j]) ... (Ap [j+1]-1)]. Column indices must be in the range 0
        to n_col-1 (the matrix is 0-based).

    double Ax [nz] ;    Input argument, not modified, of size nz = Ap [n].

        The numerical values of the sparse matrix A.

        If col_form is true (nonzero), then the nonzero pattern (row indices)
        for column j is stored in Ai [(Ap [j]) ... (Ap [j+1]-1)], and the
        corresponding (real) numerical values are stored in
        Ax [(Ap [j]) ... (Ap [j+1]-1)].  The imaginary parts are stored in
        Az [(Ap [j]) ... (Ap [j+1]-1)], for the complex versions.

        Otherwise, the nonzero pattern (column indices) for row j
        is stored in Ai [(Ap [j]) ... (Ap [j+1]-1)], and the corresponding
        (real) numerical values are stored in Ax [(Ap [j]) ... (Ap [j+1]-1)].
        The imaginary parts are stored in Az [(Ap [j]) ... (Ap [j+1]-1)],
        for the complex versions.

        No numerical values are printed if Ax or Az are (double *) NULL.

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

        The imaginary values of the sparse matrix A.   See the description
        of Ax, above.  No numerical values are printed if Az is NULL.

        Future complex version:  if Ax is present and Az is NULL, then both real
        and imaginary parts will be 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.

    Int col_form ;      Input argument, not modified.

        The matrix is in row-oriented form if form is col_form is false (0).
        Otherwise, the matrix is in column-oriented form.

    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_PRL]:  printing level.

            2 or less: no output.  returns silently without checking anything.
            3: fully check input, and print a short summary of its status
            4: as 3, but print first few entries of the input
            5: as 3, but print all of the input
            Default: 1
```

14.5 umfpack_*_report_numeric

```
int umfpack_di_report_numeric
(
    void *Numeric,
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_dl_report_numeric
(
    void *Numeric,
    const double Control [UMFPACK_CONTROL]
) ;

int umfpack_zi_report_numeric
(
    void *Numeric,
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_zl_report_numeric
(
    void *Numeric,
    const double Control [UMFPACK_CONTROL]
) ;

double int Syntax:

    #include "umfpack.h"
    void *Numeric ;
    double Control [UMFPACK_CONTROL] ;
    int status ;
    status = umfpack_di_report_numeric (Numeric, Control) ;

double long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    double Control [UMFPACK_CONTROL] ;
    long status ;
    status = umfpack_dl_report_numeric (Numeric, Control) ;

complex int Syntax:

    #include "umfpack.h"
    void *Numeric ;
    double Control [UMFPACK_CONTROL] ;
    int status ;
    status = umfpack_zi_report_numeric (Numeric, Control) ;

complex long Syntax:
    #include "umfpack.h"
    void *Numeric ;
    double Control [UMFPACK_CONTROL] ;
    long status ;
    status = umfpack_zl_report_numeric (Numeric, Control) ;

Purpose:

    Verifies and prints a Numeric object (the LU factorization, both its pattern
    numerical values, and permutation vectors P and Q).  This routine checks the
    object more carefully than the computational routines.  Normally, this check
    is not required, since umfpack_*_numeric either returns (void *) NULL, or a
    valid Numeric object.  However, if you suspect that your own code has
    corrupted the Numeric object (by overruning memory bounds, for example),
    then this routine might be able to detect a corrupted Numeric object.  Since
    this is a complex object, not all such user-generated errors are guaranteed
    to be caught by this routine.

Returns:

    UMFPACK_OK if Control [UMFPACK_PRL] <= 2 (the input is not checked).

    Otherwise:

    UMFPACK_OK if the Numeric object is valid.
    UMFPACK_ERROR_invalid_Numeric_object if the Numeric object is invalid.
    UMFPACK_ERROR_out_of_memory if out of memory.

Arguments:

    void *Numeric ;                     Input argument, not modified.

        The Numeric object, which holds the numeric factorization 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_PRL]:  printing level.

            2 or less: no output.  returns silently without checking anything.
            3: fully check input, and print a short summary of its status
            4: as 3, but print first few entries of the input
            5: as 3, but print all of the input
            Default: 1
```

14.6 umfpack_*_report_perm

```
int umfpack_di_report_perm
(
    int np,
    const int Perm [ ],
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_dl_report_perm
(
    long np,
    const long Perm [ ],
    const double Control [UMFPACK_CONTROL]
) ;

int umfpack_zi_report_perm
(
    int np,
    const int Perm [ ],
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_zl_report_perm
(
    long np,
    const long Perm [ ],
    const double Control [UMFPACK_CONTROL]
) ;

double int Syntax:

    #include "umfpack.h"
    int np, *Perm, status ;
    double Control [UMFPACK_CONTROL] ;
    status = umfpack_di_report_perm (np, Perm, Control) ;

double long Syntax:

    #include "umfpack.h"
    long np, *Perm, status ;
    double Control [UMFPACK_CONTROL] ;
    status = umfpack_dl_report_perm (np, Perm, Control) ;

complex int Syntax:

    #include "umfpack.h"
    int np, *Perm, status ;
    double Control [UMFPACK_CONTROL] ;
    status = umfpack_zi_report_perm (np, Perm, Control) ;
```

```
complex long Syntax:

    #include "umfpack.h"
    long np, *Perm, status ;
    double Control [UMFPACK_CONTROL] ;
    status = umfpack_zl_report_perm (np, Perm, Control) ;

Purpose:

    Verifies and prints a permutation vector.

Returns:

    UMFPACK_OK if Control [UMFPACK_PRL] <= 2 (the input is not checked).

    Otherwise:
    UMFPACK_OK if the permutation vector is valid (this includes that case
        when Perm is (Int *) NULL, which is not an error condition).
    UMFPACK_ERROR_n_nonpositive if np <= 0.
    UMFPACK_ERROR_out_of_memory if out of memory.
    UMFPACK_ERROR_invalid_permutation if Perm is not a valid permutation vector.

Arguments:

    Int np ;            Input argument, not modified.

        Perm is an integer vector of size np.  Restriction: np > 0.

    Int Perm [np] ;     Input argument, not modified.

        A permutation vector of size np.  If Perm is not present (an (Int *)
        NULL pointer), then it is assumed to be the identity permutation.  This
        is consistent with its use as an input argument to umfpack_*_qsymbolic,
        and is not an error condition.  If Perm is present, the entries in Perm
        must range between 0 and np-1, and no duplicates may exist.

    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_PRL]:  printing level.

            2 or less: no output.  returns silently without checking anything.
            3: fully check input, and print a short summary of its status
            4: as 3, but print first few entries of the input
            5: as 3, but print all of the input
            Default: 1
```

14.7 umfpack_*_report_symbolic

```
int umfpack_di_report_symbolic
(
    void *Symbolic,
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_dl_report_symbolic
(
    void *Symbolic,
    const double Control [UMFPACK_CONTROL]
) ;

int umfpack_zi_report_symbolic
(
    void *Symbolic,
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_zl_report_symbolic
(
    void *Symbolic,
    const double Control [UMFPACK_CONTROL]
) ;

double int Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    double Control [UMFPACK_CONTROL] ;
    int status ;
    status = umfpack_di_report_symbolic (Symbolic, Control) ;

double long Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    double Control [UMFPACK_CONTROL] ;
    long status ;
    status = umfpack_dl_report_symbolic (Symbolic, Control) ;

complex int Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    double Control [UMFPACK_CONTROL] ;
    int status ;
    status = umfpack_zi_report_symbolic (Symbolic, Control) ;

complex long Syntax:
    #include "umfpack.h"
    void *Symbolic ;
    double Control [UMFPACK_CONTROL] ;
    long status ;
    status = umfpack_zl_report_symbolic (Symbolic, Control) ;

Purpose:

    Verifies and prints a Symbolic object.  This routine checks the object more
    carefully than the computational routines.  Normally, this check is not
    required, since umfpack_*_*symbolic either returns (void *) NULL, or a valid
    Symbolic object.  However, if you suspect that your own code has corrupted
    the Symbolic object (by overruning memory bounds, for example), then this
    routine might be able to detect a corrupted Symbolic object.  Since this is
    a complex object, not all such user-generated errors are guaranteed to be
    caught by this routine.

Returns:

    UMFPACK_OK if Control [UMFPACK_PRL] is <= 2 (no inputs are checked).

    Otherwise:

    UMFPACK_OK if the Symbolic object is valid.
    UMFPACK_ERROR_invalid_Symbolic_object if the Symbolic object is invalid.
    UMFPACK_ERROR_out_of_memory if out of memory.

Arguments:

    void *Symbolic ;                    Input argument, not modified.

        The Symbolic object, which holds the symbolic factorization computed by
        umfpack_*_*symbolic.

    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_PRL]:  printing level.

            2 or less: no output.  returns silently without checking anything.
            3: fully check input, and print a short summary of its status
            4: as 3, but print first few entries of the input
            5: as 3, but print all of the input
            Default: 1
```

14.8 umfpack_*_report_triplet

```
int umfpack_di_report_triplet
(
    int n_row,
    int n_col,
    int nz,
    const int Ti [ ],
    const int Tj [ ],
    const double Tx [ ],
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_dl_report_triplet
(
    long n_row,
    long n_col,
    long nz,
    const long Ti [ ],
    const long Tj [ ],
    const double Tx [ ],
    const double Control [UMFPACK_CONTROL]
) ;

int umfpack_zi_report_triplet
(
    int n_row,
    int n_col,
    int nz,
    const int Ti [ ],
    const int Tj [ ],
    const double Tx [ ], const double Tz [ ],
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_zl_report_triplet
(
    long n_row,
    long n_col,
    long nz,
    const long Ti [ ],
    const long Tj [ ],
    const double Tx [ ], const double Tz [ ],
    const double Control [UMFPACK_CONTROL]
) ;

double int Syntax:

    #include "umfpack.h"
    int n_row, n_col, nz, *Ti, *Tj, status ;
    double *Tx, Control [UMFPACK_CONTROL] ;
    status = umfpack_di_report_triplet (n_row, n_col, nz, Ti, Tj, Tx, Control) ;

double long Syntax:

    #include "umfpack.h"
    long n_row, n_col, nz, *Ti, *Tj, status ;
    double *Tx, Control [UMFPACK_CONTROL] ;
    status = umfpack_dl_report_triplet (n_row, n_col, nz, Ti, Tj, Tx, Control) ;

complex int Syntax:

    #include "umfpack.h"
    int n_row, n_col, nz, *Ti, *Tj, status ;
    double *Tx, *Tz, Control [UMFPACK_CONTROL] ;
    status = umfpack_zi_report_triplet (n_row, n_col, nz, Ti, Tj, Tx, Tz,
        Control) ;

complex long Syntax:

    #include "umfpack.h"
    long n_row, n_col, nz, *Ti, *Tj, status ;
    double *Tx, *Tz, Control [UMFPACK_CONTROL] ;
    status = umfpack_zl_report_triplet (n_row, n_col, nz, Ti, Tj, Tx, Tz,
        Control) ;

Purpose:

    Verifies and prints a matrix in triplet form.

Returns:

    UMFPACK_OK if Control [UMFPACK_PRL] <= 2 (the input is not checked).

    Otherwise:

    UMFPACK_OK if the Triplet matrix is OK.
    UMFPACK_ERROR_argument_missing if Ti and/or Tj are missing.
    UMFPACK_ERROR_n_nonpositive if n_row <= 0 or n_col <= 0.
    UMFPACK_ERROR_nz_negative if nz < 0.
    UMFPACK_ERROR_invalid_triplet if any row or column index in Ti and/or Tj
        is not in the range 0 to n_row-1 or 0 to n_col-1, respectively.

Arguments:

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

        A is an n_row-by-n_col matrix.

    Int nz ;            Input argument, not modified.

        The number of entries in the triplet form of the matrix.
```

```
Int Ti [nz] ;       Input argument, not modified.
Int Tj [nz] ;       Input argument, not modified.
double Tx [nz] ;    Input argument, not modified.
double Tz [nz] ;    Input argument, not modified, for complex versions.

    Ti, Tj, Tx (and Tz for complex versions) hold the "triplet" form of a
    sparse matrix.  The kth nonzero entry is in row i = Ti [k], column
    j = Tj [k], the real numerical value of a_ij is Tx [k], and the
    imaginary part of a_ij is Tz [k] (for complex versions).  The row and
    column indices i and j must be in the range 0 to n_row-1 or 0 to
    n_col-1, respectively.  Duplicate entries may be present.  The
    "triplets" may be in any order.  Tx and Tz are optional; if Tx or Tz are
    not present ((double *) NULL pointers), then the numerical values are
    not printed.

    Future complex version:  if Tx is present and Tz is NULL, then both real
    and imaginary parts will be contained in Tx[0..2*nz-1], with Tx[2*k]
    and Tx[2*k+1] being the real and imaginary part of the kth entry.

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_PRL]:  printing level.

        2 or less: no output.  returns silently without checking anything.
        3: fully check input, and print a short summary of its status
        4: as 3, but print first few entries of the input
        5: as 3, but print all of the input
        Default: 1
```

14.9 umfpack_*_report_vector

```
int umfpack_di_report_vector
(
    int n,
    const double X [ ],
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_dl_report_vector
(
    long n,
    const double X [ ],
    const double Control [UMFPACK_CONTROL]
) ;

int umfpack_zi_report_vector
(
    int n,
    const double Xx [ ], const double Xz [ ],
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_zl_report_vector
(
    long n,
    const double Xx [ ], const double Xz [ ],
    const double Control [UMFPACK_CONTROL]
) ;

double int Syntax:

    #include "umfpack.h"
    int n, status ;
    double *X, Control [UMFPACK_CONTROL] ;
    status = umfpack_di_report_vector (n, X, Control) ;

double long Syntax:

    #include "umfpack.h"
    long n, status ;
    double *X, Control [UMFPACK_CONTROL] ;
    status = umfpack_dl_report_vector (n, X, Control) ;

complex int Syntax:

    #include "umfpack.h"
    int n, status ;
    double *Xx, *Xz, Control [UMFPACK_CONTROL] ;
    status = umfpack_zi_report_vector (n, Xx, Xz, Control) ;
```

```
complex long Syntax:

    #include "umfpack.h"
    long n, status ;
    double *Xx, *Xz, Control [UMFPACK_CONTROL] ;
    status = umfpack_zl_report_vector (n, Xx, Xz, Control) ;

Purpose:

    Verifies and prints a dense vector.

Returns:

    UMFPACK_OK if Control [UMFPACK_PRL] <= 2 (the input is not checked).

    Otherwise:

    UMFPACK_OK if the vector is valid.
    UMFPACK_ERROR_argument_missing if X or Xx is missing.
    UMFPACK_ERROR_n_nonpositive if n <= 0.

Arguments:

    Int n ;             Input argument, not modified.

        X is a real or complex vector of size n.  Restriction: n > 0.

    double X [n] ;      Input argument, not modified.  For real versions.

        A real vector of size n.  X must not be (double *) NULL.

    double Xx [n or 2*n] ; Input argument, not modified.  For complex versions.
    double Xz [n or 0] ;   Input argument, not modified.  For complex versions.

        A complex vector of size n, in one of two storage formats.
        Xx must not be (double *) NULL.

        If Xz is not (double *) NULL, then Xx [i] is the real part of X (i) and
        Xz [i] is the imaginary part of X (i).  Both vectors are of length n.
        This is the "split" form of the complex vector X.

        If Xz is (double *) NULL, then Xx holds both real and imaginary parts,
        where Xx [2*i] is the real part of X (i) and Xx [2*i+1] is the imaginary
        part of X (i).  Xx is of length 2*n doubles.  If you have an ANSI C99
        compiler with the intrinsic double _Complex type, then Xx can be of
        type double _Complex in the calling routine and typecast to (double *)
        when passed to umfpack_*_report_vector (this is untested, however).
        This is the "merged" form of the complex vector X.

        Future work:  all complex routines in UMFPACK could use this same
        strategy for their complex arguments.  The split format is useful for
        MATLAB, which holds its real and imaginary parts in seperate arrays.
        The merged format is compatible with the intrinsic double _Complex
        type in ANSI C99, and is also compatible with SuperLU's method of
        storing complex matrices.  In the current version, only
        umfpack_*_report_vector supports both formats.

    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_PRL]:  printing level.

            2 or less: no output.  returns silently without checking anything.
            3: fully check input, and print a short summary of its status
            4: as 3, but print first few entries of the input
            5: as 3, but print all of the input
            Default: 1
```

15 Utility routines

15.1 umfpack_timer

```
double umfpack_timer ( void ) ;

Syntax (for all versions: di, dl, zi, and zl):

    #include "umfpack.h"
    double t ;
    t = umfpack_timer ( ) ;

Purpose:

    Returns the CPU time used by the process.  Includes both "user" and "system"
    time (the latter is time spent by the system on behalf of the process, and
    is thus charged to the process).  It does not return the wall clock time.

    This routine uses the Unix getrusage routine, if available.  It is less
    subject to overflow than the ANSI C clock routine.  If getrusage is not
    available, the portable ANSI C clock routine is used instead.
    Unfortunately, clock ( ) overflows if the CPU time exceeds 2147 seconds
    (about 36 minutes) when sizeof (clock_t) is 4 bytes.  If you have getrusage,
    be sure to compile UMFPACK with the -DGETRUSAGE flag set; see umf_config.h
    and the User Guide for details.  Even the getrusage routine can overlow.

Arguments:

    None.
```

References

[1] M. Arioli, J. W. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Applic.*, 10:165–190, 1989.

[2] T. A. Davis. Algorithm 8xx: UMFPACK V3.2, an unsymmetric-pattern multifrontal method with a column pre-ordering strategy. Technical Report TR-02-002, Univ. of Florida, CISE Dept., Gainesville, FL, January 2002. (www.cise.ufl.edu/tech-reports. Submitted to *ACM Trans. Math. Softw.)*.

[3] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. Technical Report TR-02-001, Univ. of Florida, CISE Dept., Gainesville, FL, January 2002. (www.cise.ufl.edu/tech-reports. Submitted to *ACM Trans. Math. Softw.)*.

[4] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Anal. Applic.*, 18(1):140–158, 1997.

[5] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Softw.*, 25(1):1–19, 1999.

[6] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. Algorithm 8xx: COLAMD, a column approximate minimum degree ordering algorithm. Technical Report TR-00-006, Univ. of Florida, CISE Dept., Gainesville, FL, October 2000. (www.cise.ufl.edu/tech-reports. Submitted to *ACM Trans. Math. Softw.)*.

[7] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. Technical Report TR-00-005, Univ. of Florida, CISE Dept., Gainesville, FL, October 2000. (www.cise.ufl.edu/tech-reports. Submitted to *ACM Trans. Math. Softw.)*.

[8] T. A. Davis and W. W. Hager. Modifying a sparse Cholesky factorization. *SIAM J. Matrix Anal. Applic.*, 20(3):606–627, 1999.

[9] M. J. Daydé and I. S. Duff. The RISC BLAS: A blocked implementation of level 3 BLAS for RISC processors. *ACM Trans. Math. Softw.*, 25(3), Sept. 1999.

[10] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Applic.*, 20(3):720–755, 1999. www.netlib.org.

[11] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level-3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[12] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Comm. ACM*, 30:403–407, 1987. www.netlib.org.

[13] I. S. Duff and J. K. Reid. An implementation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Trans. Math. Softw.*, 4(2):137–147, 1978.

[14] I. S. Duff and J. A. Scott. The design of a new frontal code for solving sparse unsymmetric systems. *ACM Trans. Math. Softw.*, 22(1):30–45, 1996.

[15] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[16] A. George and E. G. Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM J. Sci. Statist. Comput.*, 6(2):390–409, 1985.

[17] A. George and E. G. Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM J. Sci. Statist. Comput.*, 8(6):877–898, 1987.

[18] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Applic.*, 13(1):333–356, 1992.

[19] J. R. Gilbert and E. G. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In A. George, J. R. Gilbert, and J. W.H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, Volume 56 of the IMA Volumes in Mathematics and its Applications, pages 107–139. Springer-Verlag, 1993.

[20] J. R. Gilbert, E. G. Ng, and B. W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Anal. Applic.*, 15(4):1075–1091, 1994.

[21] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.*, 9:862–874, 1988.

[22] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.

[23] S. I. Larimore. An approximate minimum degree column ordering algorithm. Technical Report TR-98-016, Univ. of Florida, CISE Dept., Gainesville, FL, 1998. www.cise.ufl.edu/tech-reports.

[24] R. C Whaley, A. Petitet, and J. J. Dongarra. Automated emperical optimization of software and the ATLAS project. Technical Report LAPACK Working Note 147, Computer Science Department, The University of Tennessee, September 2000. www.netlib.org/atlas.