UMFPACK Version 3.2 User Guide

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

January 1, 2002

Abstract

UMFPACK is a set of routines for solving unsymmetric sparse linear systems, $\mathbf{Ax} = \mathbf{b}$, 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.

Acknowledgments:

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

Contents

1 Overview

UMFPACK Version 3.2 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).

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 [3, 2], available at www.cise.ufl.edu/tech-reports.

2 Availability

UMFPACK Version 3.2 is available at www.cise.ufl.edu/research/sparse, and has been submitted as a collected algorithm of the ACM [3, 2]. It makes use of a modified version of COLAMD V2.0 by

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 ;`

Timothy A. Davis, Stefan Larimore, John Gilbert, and Esmond Ng. The original COLAMD V2.0 is available 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. Prior versions of UMFPACK, 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, you can either type `make umfpack` in the Unix system shell, or type `umfpack_make` in MATLAB (which should work on any system, including Windows). See Section 8 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 `A` to be square, sparse, nonsingular and not complex, and it requires `b` to be a dense column vector (and not complex). This is more restrictive than what you can do with MATLAB's backslash or `LU`. Future releases of UMFPACK may remove some of these restrictions.

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) ;
```

```
n = size (A,1) ;
I = speye (n) ;
P = I (P,:) ;
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 two examples do not make use of the iterative refinement that is built into `x = umfpack (A, '\', b)` however.

Since the numeric factorization refines its column pre-ordering, the `Q` in `[L,U,P,Q] = umfpack (A)` and `[Q,F,C] = umfpack (A, 'symbolic')` will in general be different.

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 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_demo.c`. You may want to type `more on` before running the `umfpack_simple` demo since it generates lots of output.

A simple M-file (`umfpack_btf`) is provided that first permutes the matrix to upper block triangular form, using MATLAB's `dmperm` routine. It solves $\mathbf{Ax} = \mathbf{b}$; the LU factors are not returned. Its usage is simple: `x = umfpack_btf (A, b)`. Type `help umfpack_btf` for more options.

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), with some elbow room so that it can run more efficiently. If this fails, it reduces its request and uses less memory. 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 routines. See the `-DMATHWORKS` option in `umf_config.h` in Section 18 for details.

Memory allocation on a PC is notoriously bad, so I recommend setting `Control(7)` to a non-default value of 1.0 or even 1.05. This will avoid most reallocations of memory.

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 dual versions, with different sizes of integers. All user-callable routine names begin with `umfpack_` or `umfpack_l_`; other routine names beginning with `umf_` or `umfl_` are internal to the package, and should not be called by the user. The include file `umfpack.h`, listed in Section 16, must be included in any C program that uses UMFPACK.

4.1 The size of an integer

There are two versions of each user-callable routine (except for one routine). The routine names starting with just `umfpack_` use `int` integer arguments; those starting with `umfpack_l_` use `long` integer arguments. If you compile UMFPACK in the standard ILP32 mode (32-bit `int`'s, `long`'s, an pointers) then the versions are essentially identical. You will be able to solve problems using up to 4GB 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_l_*` routines can solve huge problems (not limited to 4GB), 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.

Both versions are discussed below. 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.

4.2 Primary routines, and a simple example

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

- `umfpack_symbolic`, `umfpack_l_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 $6n$ integers (where $\mathbf{A}$ is n-by-n). The matrix must be structurally non-singular (more precisely, each row and column must have at least one entry).

- `umfpack_numeric`, `umfpack_l_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 must be non-singular.

- `umfpack_solve`, `umfpack_l_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.

- `umfpack_free_symbolic`, `umfpack_l_free_symbolic`:

  Frees the `Symbolic` object created by `umfpack_symbolic` or `umfpack_qsymbolic`.

- `umfpack_free_numeric`, `umfpack_l_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 arrays, where the matrix is `n`-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] ;
```

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

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

Here is a simple main program, `umfpack_simple.c`, that illustrates the basic usage of 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 (int argc, char **argv)
{
    double *Control = (double *) NULL, *Info = (double *) NULL ;
    int i ;
    void *Symbolic, *Numeric ;
    (void) umfpack_symbolic (n, Ap, Ai, &Symbolic, Control, Info) ;
    (void) umfpack_numeric (Ap, Ai, Ax, Symbolic, &Numeric, Control, Info) ;
    umfpack_free_symbolic (&Symbolic) ;
    (void) umfpack_solve ("Ax=b", Ap, Ai, Ax, x, b, Numeric, Control, Info) ;
    umfpack_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

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

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

4.3 Alternative routines

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

- `umfpack_defaults`, `umfpack_l_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 fully described in Section 11.1. 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 `x = A\b` in MATLAB for nearly all matrices.

- `umfpack_qsymbolic`, `umfpack_l_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`, `umfpack_l_wsolve`:

  An alternative to `umfpack_solve` which does not dynamically allocate any memory. Requires the user to pass several additional size-n work arrays.

4.4 Matrix manipulation routines

The compressed column data structure is compact, and simplifies the UMFPACK routines that operate on the sparse matrix $\mathbf{A}$. 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 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 `k`-th triplet is (i, j, a_{ij}), where i = `Ti [k]`, j = `Tj [k]`, and a_{ij} = `Tx [k]`. The triplets can be in any order in the `Ti`, `Tj`, and `Tx` arrays, 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`, `umfpack_l_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.

- `umfpack_col_to_triplet`, `umfpack_l_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`, `umfpack_l_transpose`:

  Transposes and optionally permutes a column form matrix [22]. Identical to `B = A (P,Q)'` in MATLAB, except for the presence of numerically zero entries.

It is quite easy to add matrices in triplet form, transpose them, and permute them. See the discussion in `umfpack_triplet_to_col` in Section 12 for more details. All of the matrix manipulation routines can correctly operate on singular matrices.

4.5 Getting the contents of opaque objects

There are cases where the user would like 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. Thus, 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`, `umfpack_l_get_lunz`:

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

- `umfpack_get_numeric`, `umfpack_l_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 (also 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 `k`-th pivot row (or the `k`-th row of $\mathbf{PAQ}$), and where `Q [k] = j` means that column `j` of the original matrix is the `k`-th pivot column. This is identical to how MATLAB uses permutation vectors.

- `umfpack_get_symbolic`, `umfpack_l_get_symbolic`:

  Copies the contents of the `Symbolic` object (the initial column preordering, and 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.

4.6 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`, `umfpack_l_report_status`:

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

- `umfpack_report_info`, `umfpack_l_report_info`:

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

- `umfpack_report_control`, `umfpack_l_report_control`:

  Prints the `Control` settings.

- `umfpack_report_matrix`, `umfpack_l_report_matrix`:

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

- `umfpack_report_triplet`, `umfpack_l_report_triplet`:

  Verifies and prints a matrix in triplet form.

- `umfpack_report_symbolic`, `umfpack_l_report_symbolic`:

  Verifies and prints a `Symbolic` object.

- `umfpack_report_numeric`, `umfpack_l_report_numeric`:

  Verifies and prints a `Numeric` object.

- `umfpack_report_perm`, `umfpack_l_report_perm`:

  Verifies and prints a permutation vector.

- `umfpack_report_vector`, `umfpack_l_report_vector`:

  Verifies and prints a real 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 are in the range `0` to `n-1`, but it prints them as if they are in the range `1` to `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`, `umfpack_l_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`, `umfpack_l_report_control`:

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

- umfpack_report_info, umfpack_l_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.

4.7 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 both int and long versions (there is no umfpack_l_timer routine).

4.8 Control parameters

UMFPACK uses an optional double array, Control, to modify its control parameters. These may be modified by the user (see umfpack_defaults and umfpack_l_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.

4.9 Larger examples

A full example of all user-callable UMFPACK routines (the int routines) is available in the C main program, umfpack_demo.c listed in Section 17. A nearly identical program that uses the long integer version of UMFPACK is in umfpack_l_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 all C-callable routines (int version)

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

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:			
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(13)	Control[UMFPACK_PIVOT_OPTION]	0	symmetric pivot preference
Control(14)	Control[UMFPACK_RELAXED2_AMALGAMATION]	0.1	amalgamation
Control(15)	Control[UMFPACK_RELAXED3_AMALGAMATION]	0.125	amalgamation
Used by umfpack_*solve:			
Control(8)	Control[UMFPACK_IRSTEP]	2	max iter. refinement 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

a prior category have the same size and type.

5.1 Primary routines

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

status = umfpack_symbolic (n, Ap, Ai, &Symbolic, Control, Info) ;
status = umfpack_numeric (Ap, Ai, Ax, Symbolic, &Numeric, Control, Info) ;
status = umfpack_solve (sys, Ap, Ai, Ax, X, B, Numeric, Control, Info) ;
umfpack_free_symbolic (&Symbolic) ;
umfpack_free_numeric (&Numeric) ;
```

5.2 Alternative routines

```
int Qinit [n], Wi [n] ;
double W [n], Y [n], Z [n], S [n] ;

umfpack_defaults (Control) ;
status = umfpack_qsymbolic (n, Ap, Ai, Qinit, &Symbolic, Control, Info) ;
status = umfpack_wsolve (sys, Ap, Ai, Ax, X, B, Numeric, Control, Info,
    Wi, W, Y, Z, S) ;
```

5.3 Matrix manipulation routines

```
int Ti [nz], Tj [nz], Bp [n+1], Bi [max(n,nz)], P [n], Q [n], Cp [n+1], Ci [nz] ;
double Tx [nz], Cx [nz], Bx [nz] ;

status = umfpack_col_to_triplet (n, Ap, Tj) ;
status = umfpack_triplet_to_col (n, nz, Ti, Tj, Tx, Bp, Bi, Bx) ;
status = umfpack_transpose (n, Ap, Ai, Ax, P, Q, Cp, Ci, Cx) ;
```

5.4 Getting the contents of opaque objects

```
int lnz, unz, Lp [n+1], Li [lnz], Up [n+1], Ui [unz] ;
double Lx [lnz], Ux [unz] ;
int nfr, nchains, nsparse_col, Qtree [n], Front_npivots [n], Front_parent [n],
    Chain_start [n], Chain_maxrows [n], Chain_maxcols [n] ;

status = umfpack_get_lunz (&lnz, &unz, &n, Numeric) ;
status = umfpack_get_numeric (Lp, Li, Lx, Up, Ui, Ux, P, Q, Numeric) ;
status = umfpack_get_symbolic (&n, &nz, &nfr, &nchains, &nsparse_col,
    Qtree, Front_npivots, Front_parent, Chain_start, Chain_maxrows,
    Chain_maxcols, Symbolic) ;
```

Note: the `nsparse_col` argument is no longer relevant. It is always equal to `n` in this version.

5.5 Reporting routines

```
char *name, *form ;

umfpack_report_status (Control, status) ;
umfpack_report_control (Control) ;
umfpack_report_info (Control, Info) ;
status = umfpack_report_matrix (name, n, Ap, Ai, Ax, form, Control) ;
status = umfpack_report_numeric (name, Numeric, Control) ;
status = umfpack_report_perm (name, n, P, Control) ;
status = umfpack_report_symbolic (name, Symbolic, Control) ;
status = umfpack_report_triplet (name, n, nz, Ti, Tj, Tx, Control) ;
status = umfpack_report_vector (name, n, X, Control) ;
```

6 Synopsis of all C-callable routines (`long` version)

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. Note that the include file, `umfpack.h`, is the same for both `int` and `long` versions of UMFPACK.

6.1 Primary routines

```
#include "umfpack.h"
long status, n, nz, Ap [n+1], Ai [nz] ;
```

```
double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO], Ax [nz], X [n], B [n] ;
void *Symbolic, *Numeric ;
char *sys ;

status = umfpack_l_symbolic (n, Ap, Ai, &Symbolic, Control, Info) ;
status = umfpack_l_numeric (Ap, Ai, Ax, Symbolic, &Numeric, Control, Info) ;
status = umfpack_l_solve (sys, Ap, Ai, Ax, X, B, Numeric, Control, Info) ;
umfpack_l_free_symbolic (&Symbolic) ;
umfpack_l_free_numeric (&Numeric) ;
```

6.2 Alternative routines

```
long Qinit [n], Wi [n] ;
double W [n], Y [n], Z [n], S [n] ;

umfpack_l_defaults (Control) ;
status = umfpack_l_qsymbolic (n, Ap, Ai, Qinit, &Symbolic, Control, Info) ;
status = umfpack_l_wsolve (sys, Ap, Ai, Ax, X, B, Numeric, Control, Info,
    Wi, W, Y, Z, S) ;
```

6.3 Matrix manipulation routines

```
long Ti [nz], Tj [nz], Bp [n+1], Bi [max(n,nz)], P [n], Q [n], Cp [n+1], Ci [nz] ;
double Tx [nz], Cx [nz], Bx [nz] ;

status = umfpack_l_col_to_triplet (n, Ap, Tj) ;
status = umfpack_l_triplet_to_col (n, nz, Ti, Tj, Tx, Bp, Bi, Bx) ;
status = umfpack_l_transpose (n, Ap, Ai, Ax, P, Q, Cp, Ci, Cx) ;
```

6.4 Getting the contents of opaque objects

```
long lnz, unz, Lp [n+1], Li [lnz], Up [n+1], Ui [unz] ;
double Lx [lnz], Ux [unz] ;
long nfr, nchains, nsparse_col, Qtree [n], Front_npivots [n], Front_parent [n],
    Chain_start [n], Chain_maxrows [n], Chain_maxcols [n] ;

status = umfpack_l_get_lunz (&lnz, &unz, &n, Numeric) ;
status = umfpack_l_get_numeric (Lp, Li, Lx, Up, Ui, Ux, P, Q, Numeric) ;
status = umfpack_l_get_symbolic (&n, &nz, &nfr, &nchains, &nsparse_col,
    Qtree, Front_npivots, Front_parent, Chain_start, Chain_maxrows,
    Chain_maxcols, Symbolic) ;
```

Note: the `nsparse_col` argument is no longer relevant. It is always equal to `n` in this version.

6.5 Reporting routines

```
char *name, *form ;

umfpack_l_report_status (Control, status) ;
umfpack_l_report_control (Control) ;
umfpack_l_report_info (Control, Info) ;
```

```
status = umfpack_l_report_matrix (name, n, Ap, Ai, Ax, form, Control) ;
status = umfpack_l_report_numeric (name, Numeric, Control) ;
status = umfpack_l_report_perm (name, n, P, Control) ;
status = umfpack_l_report_symbolic (name, Symbolic, Control) ;
status = umfpack_l_report_triplet (name, n, nz, Ti, Tj, Tx, Control) ;
status = umfpack_l_report_vector (name, n, X, Control) ;
```

7 Synopsis of utility routines

This routine is the same in both `int` and `long` versions of UMFPACK.

```
double t ;

t = umfpack_timer ( ) ;
```

8 Installation

UMFPACK comes with a `Makefile` for compiling the C-callable `umfpack.a` library and the `umfpack` mexFunction on Unix. System-dependent configurations are controlled by the `Makefile`, and defined in `umf_config.h` listed in Section 18. 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 three demo programs will be executed, and the output of `umfpack_demo.c` and `umfpack_l_demo.c` will be compared with `umfpack_demo.out` and `umfpack_l_demo.out`. These two demo programs are identical, except that `umfpack_demo.c` uses the `int` version, while `umfpack_l_demo.out` uses the `long` version of UMFPACK. Expect to see a few differences, such as residual norms, compile-time control settings, and perhaps memory usage differences. (The Compaq Alpha uses the LP64 model by default, so if you're using that computer compare your output with the 64-bit Solaris output in `umfpack_demo.out64` and `umfpack_l_demo.out64`). The BLAS [9, 11, 24] will not be used, so the performance of UMFPACK will not be as high as possible. For better performance, edit the `Makefile` and un-comment the `include Make.*` statement that is specific to your computer. 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` or your system-specific `Make.*` file, be sure to type `make purge` before recompiling. Here are the various parameters that you can control in your `Make.*` file; more details are in `umf_config.h` listed in Section 18:

- 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.

The CONFIG string can include combinations of the following:

- -DNBLAS if you do not have any BLAS at all. By default, umf_config.h assumes you have some version of the BLAS. The BLAS are de-selected in Make.generic with the statement CONFIG = -DNBLAS.
- -DNCBLAS if you do not have the C-BLAS [24]. The interface to the C-BLAS is identical on any system (Unix or Windows). By default, umf_config.h assumes you have the C-BLAS, except for Solaris (which has sunperf) and MATLAB, which has its own BLAS for compiling the MATLAB mexFunction on any system.
- -DNSUNPERF if you are on Solaris but do not have sunperf.
- -DLONGBLAS if your BLAS can take long integer input arguments. If not defined, then the umfpack_l_* version of UMFPACK that uses long integers does not call the BLAS.
- -DGETRUSAGE if you have the getrusage function. This should exist on any UNIX system.
- Options for controlling how C calls the Fortran BLAS: -DBLAS_BY_VALUE, -DBLAS_NO_UNDERSCORE, and -DBLAS_CHAR_ARG. These are set automatically for Sun Solaris, SGI Irix, Red Hat Linux, Compaq Alpha, and AIX (the IBM RS 6000).

To compile the umfpack mexFunction on Unix, type make umfpack. The MATLAB mex command will select the appropriate compiler and compiler flags for your system, and the BLAS internal to MATLAB will be used. 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.

If you're running Windows, and all you want to do is use UMFPACK in MATLAB, then just type umfpack_make in MATLAB. MATLAB Version 6.0 or higher is required. You won't be able to use the BLAS when compiling with the lcc compiler provided with MATLAB Version 6.0); you will get an error stating that _dgemm is undefined. There is no work-around for this problem. Either use a different C compiler, or don't use the BLAS.

9 Future work

Here are a few features that are not in UMFPACK Version 3.2, 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.
2. a condition number estimator. You can write your own in MATLAB by making a copy of the built-in MATLAB `condest.m` routine and replacing `LU` with `umfpack`. Be sure to do so only if your MATLAB license allows you to, and do not distribute the derivative MATLAB code without direct permission from The Mathworks, Inc.
3. an estimate of the 1-norm of $\mathbf{PAQ} - \mathbf{LU}$.
   See `ftp://ftp.mathworks.com/pub/contrib/v4/linalg/normest1.m` for a similar algorithm that computes the 1-norm estimate of $\sigma\mathbf{I} + \mathbf{AA}^\mathsf{T} - \mathbf{LL}^\mathsf{T}$. It can easily be modified to compute the 1-norm estimate of $\mathbf{PAQ} - \mathbf{LU}$. See also [8].
4. a complex version.
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 symbolic and numeric factorization could handle singular matrices, just like MATLAB's `LU`.
10. the ability to use user-provided `malloc`, `free`, and `realloc` memory allocation routines. Note that UMFPACK makes very few calls to these routines.
11. 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.
12. future versions may return more statistics in the `Info` array, and they may use more entries in the `Control` array.
13. 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 $\mathbf{U}$.
14. an option of extracting the diagonal of $\mathbf{U}$ (or other subsets of $\mathbf{L}$ and $\mathbf{U}$) from the `Numeric` object without having to extract the entire LU factorization.

15. a Fortran interface (this would probably require modifying UMFPACK to use user-provided work arrays).
16. a C++ interface.
17. a parallel version using MPI.

10 The primary UMFPACK routines

The include files are the same for both `int` and `long` 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 and umfpack_l_symbolic

```
int umfpack_symbolic
(
    int n,
    const int Ap [ ],
    const int Ai [ ],
    void **Symbolic,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

long umfpack_l_symbolic
(
    long n,
    const long Ap [ ],
    const long Ai [ ],
    void **Symbolic,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    long n, *Ap, *Ai, status ;
    double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
    status = umfpack_l_symbolic (n, 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, use umfpack_qsymbolic instead.
```

```
Returns:

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

Arguments:

    Int n ;             Input argument, not modified.

        A is an n-by-n matrix.  Restriction: n > 0.

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

        Ap is an integer array of size n+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-1.  The value nz = Ap [n] is thus the total number of entries
        in the pattern of the matrix A.  nz must be greater than zero.

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

        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-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.  The following Control parameters
        are used:

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

        Control [UMFPACK_DENSE_COL]:  columns with more than
            max (16, Control [UMFPACK_DENSE_COL] * 16 * sqrt (n))
            entries are placed placed last in the column pre-ordering by
```

```
        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_singular_matrix

            Matrix is singular.

        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 (Ap and/or Ai) is missing.

    UMFPACK_ERROR_internal_error

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

Info [UMFPACK_N]:  the value of the input argument n.

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

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.  Note that all rows are stored in the
    same data structure, regardless of whether they are "sparse",
    "dense", or "empty".

Info [UMFPACK_NEMPTY_ROW]:  number of "empty" rows in A.  These are
    rows 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.
    Any given column is classified as either "dense" or "empty" or
    "sparse".  All columns are stored in the same data structure,
```

however (Version 3.0 stored dense columns in a separate dense array, but this is no longer true for Version 3.1 and following).

Info [UMFPACK_NEMPTY_COL]: number of "empty" columns in A. These are columns whose entries are all in "dense" rows. These columns are ordered last in the factorization, along with "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 + (20 to 25)*n integers, 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 (1 to 6)*n integers, depending on the matrix.

Info [UMFPACK_VARIABLE_INIT_ESTIMATE]: the Numeric object contains two components. The first is fixed in size (O (n) information, plus the "dense" part of the LU factors). 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 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 flop counts include add, subtract, multiply, and divide.  They
    exclude max, absolute value computations, and comparisons.

    The actual "true flop" count found by umfpack_numeric will be less
    than this estimate.

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 and umfpack_l_numeric

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

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    void *Symbolic, *Numeric ;
    long *Ap, *Ai, status ;
    double *Ax, Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
    status = umfpack_l_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+1] ;      Input argument, not modified.

        This must be identical to the Ap array passed to umfpack_symbolic.
        The value of n 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].

        This must be identical to the Ai array passed to umfpack_symbolic.

        Not all changes to Ai and Ap are detected; if the matrix has the same
        number of nonzeros and can be factorized in the existing frontal
        matrices as defined in the Symbolic object, then umfpack_numeric will
        not complain, and will successfully factorize the matrix and return a
        valid Numeric object.

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

        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)].

    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 addres 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.  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 it 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.

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, 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 10% 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.  I may try to address this is
        issue in a future release - in the mean time, 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 affect on fill-in or operation count.
        It has a small impact on run-time (the extra time required to do
        the garbage collection and memory reallocation).

    Control [UMFPACK_PIVOT_OPTION]:  If this is nonzero, then entries on
        the diagonal of A are given preference over off-diagonal entries.
        This can improve the fill-in on matrices with symmmetric nonzero
        pattern.  Default: 0 (do not give preference to the diagonal of A).
        This parameter was added for UMFPACK Version 3.1.

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.  Only in this case
        does umfpack_numeric compute a valid numeric factorization.
        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 (Ap, Ai, and/or Ax) are missing.

    UMFPACK_ERROR_singular_matrix

        The input matrix is singular.

    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_N]:  the value of n 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.  It does
    include the flops performed to factorize the "dense" and "empty"
    columns.

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.  It does
    include entries in "dense" or "empty" columns.  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.  It does
    include entries in "dense" or "empty" columns.  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 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 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 "sparse" part L and U.  The "sparse" part
    of L and U excludes entries on the diagonal, which is stored
    separately.  It excludes entries in the "dense" and "empty"
    columns.  Those are stored together in a single dense array of
    size n by (Info [UMFPACK_NDENSE_COL] + Info [UMFPACK_NEMPTY_COL]),
    and no integers are required to represent their pattern.

Info [UMFPACK_LU_ENTRIES]:  The total number of numerical values that
    are stored for the LU factors, including the dense array for "dense"
    and "empty" columns.  Some of the values may be explicitly zero.

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.

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 and umfpack_l_solve

```
int umfpack_solve
(
    const char 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_l_solve
(
    const char 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 Syntax:

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

long Syntax:

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

```
Returns:

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

Arguments:

    char sys [ ] ;              Input argument, not modified.

        A string that defines which system to solve.  Iterative refinement can
        be optionally performed when the sys argument is:

            "Ax=b"
            "A'x=b"

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

            "P'Lx=b"
            "L'Px=b"
            "UQ'x=b"
            "QU'x=b"
            "Lx=b"
            "L'x=b"
            "Ux=b"
            "U'x=b"

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

        If iterative refinement is requested (Control [UMFPACK_IRSTEP] >= 1 and
        Ax=b or A'x=b is being solved), 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 three arguments, so the results
        are undefined if Ap, Ai, and/or Ax 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 or A'x=b is being solved.

    double X [n] ;              Output argument.

        The solution to the linear system.

    double B [n] ;              Input argument, not modified.

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

    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.  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 or A'x=b is not being solved, then the Ap,
        Ai, and Ax 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_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, and
            sys arguments are always required.  Info and Control are
            not required.  Ap, Ai, and Ax are required if Ax=b or A'x=b
            is to be solve and the (default) iterative refinement is
            requested.

        UMFPACK_ERROR_invalid_Numeric_object

            The Numeric object is not valid.

    Info [UMFPACK_N]:  the value of n stored in the Numeric object.

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

    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.  This is set to -1 if
    iterative refinement was not requested.

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).  This
    statistic is set to -1 if iterative refinement was not requested.

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 and umfpack_l_free_symbolic

```
void umfpack_free_symbolic
(
    void **Symbolic
) ;

void umfpack_l_free_symbolic
(
    void **Symbolic
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    umfpack_l_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 and umfpack_l_free_numeric

```
void umfpack_free_numeric
(
    void **Numeric
) ;

void umfpack_l_free_numeric
(
    void **Numeric
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    umfpack_l_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 and umfpack_l_defaults

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

void umfpack_l_defaults
(
    double Control [UMFPACK_CONTROL]
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL] ;
    umfpack_l_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 and umfpack_l_qsymbolic

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

long umfpack_l_qsymbolic
(
    long n,
    const long Ap [ ],
    const long Ai [ ],
    const long Qinit [ ],
    void **Symbolic,
    const double Control [UMFPACK_CONTROL],
    double Info [UMFPACK_INFO]
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    long n, *Ap, *Ai, *Qinit, status ;
    double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
    status = umfpack_l_qsymbolic (n, 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] ;             Input argument, not modified.

        The user's fill-reducing initial column preordering.  This must be a
        permutation of 0..n-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, Ap, Ai, Control, Info) ;

        is identical to

        Symbolic = umfpack_qsymbolic (n, 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]:  Let j be the leftmost column in
            A (:,Qinit) with more entries than the value determined by the
            dense column control parameter (see umfpack_symbolic), or j=n if
            there is no such column.  Columns j through n-1 are all treated as
            "dense", and factorized in a (n-j)-by-n dense array.  When
            determining Qinit, be sure the "dense" columns of A (:,Qinit) are
            as far to the right as possible.

    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]:  zero
Info [UMFPACK_NDENSE_COL]:  n-j, where j is defined above.
Info [UMFPACK_NEMPTY_COL]:  zero
```

11.3 umfpack_wsolve and umfpack_l_wsolve

```
int umfpack_wsolve
(
    const char 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 [ ],
    double Y [ ],
    double Z [ ],
    double S [ ]
) ;

long umfpack_l_wsolve
(
    const char 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 [ ],
    double Y [ ],
    double Z [ ],
    double S [ ]
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
```

```
    void *Numeric ;
    long status, *Ap, *Ai ;
    char *sys ;
    double *B, *X, *Ax, Info [UMFPACK_INFO], Control [UMFPACK_CONTROL] ;
    long *Wi ;
    double *W, *Y, *Z, *S ;
    status = umfpack_l_wsolve (sys, Ap, Ai, Ax, X, B, Numeric, Control, Info,
        Wi, W, Y, Z, S) ;

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 slightly faster
    than umfpack_solve, since the workspace (Wi, W, Y, Z, and S) needs to be
    allocated only once, prior to calling umfpack_wsolve.

Returns:

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

Arguments:

    char 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.

        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 [2*n] ;              Workspace.
    double W [n] ;              Workspace.
    double Y [n] ;              Workspace, only needed for iterative refinement.
    double Z [n] ;              Workspace, only needed for iterative refinement.
    double S [n] ;              Workspace, only needed for iterative refinement.

        The Wi, W, Y, Z, and S arguments are workspace used by umfpack_wsolve.
        Their contents are undefined on output.
```

12 Matrix manipulation routines

12.1 umfpack_col_to_triplet and umfpack_l_col_to_triplet

```
int umfpack_col_to_triplet
(
    int n,
    const int Ap [ ],
    int Tj [ ]
) ;

long umfpack_l_col_to_triplet
(
    long n,
    const long Ap [ ],
    long Tj [ ]
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    long n, *Tj, *Ap, status ;
    status = umfpack_l_col_to_triplet (n, 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.

Returns:

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

Arguments:

    Int n ;             Input argument, not modified.
```

```
    A is an n-by-n matrix.  Restriction: n > 0.

Int Ap [n+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].  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] must be true for all j in the
    range 0 to n-1.  Empty columns are OK (that is, Ap [j] may equal
    Ap [j+1] for any j in the range 0 to n-1).

Int Tj [nz] ;       Output argument.

    Tj is an integer array of size nz on input, where nz = Ap [n].
    Suppose the column-form of the matrix is held in Ap, Ai, and Ax
    (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 in order to do the conversion.
```

12.2 umfpack_triplet_to_col and umfpack_l_triplet_to_col

```
int umfpack_triplet_to_col
(
    int n,
    int nz,
    const int Ti [ ],
    const int Tj [ ],
    const double Tx [ ],
    int Bp [ ],
    int Bi [ ],
    double Bx [ ]
) ;

long umfpack_l_triplet_to_col
(
    long n,
    long nz,
    const long Ti [ ],
    const long Tj [ ],
    const double Tx [ ],
    long Bp [ ],
    long Bi [ ],
    double Bx [ ]
) ;

int Syntax:

    #include "umfpack.h"
    int n, nz, *Ti, *Tj, *Bp, *Bi, status ;
    double *Tx, *Bx ;
    status = umfpack_triplet_to_col (n, nz, Ti, Tj, Tx, Bp, Bi, Bx) ;

long Syntax:

    #include "umfpack.h"
    long n, nz, *Ti, *Tj, *Bp, *Bi, status ;
    double *Tx, *Bx ;
    status = umfpack_l_triplet_to_col (n, nz, Ti, Tj, Tx, Bp, Bi, Bx) ;

Purpose:

    Converts a sparse matrix from "triplet" form to compressed-column form.

    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.

    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]), 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.

    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).

Returns:

    UMFPACK_OK if successful.
    UMFPACK_ERROR_argument_missing if Bp, Bi, Ti, and/or Tj are missing.
    UMFPACK_ERROR_n_nonpositive if n <= 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-1.
    UMFPACK_ERROR_out_of_memory if unable to allocate sufficient workspace.

Arguments:

    Int n ;             Input argument, not modified.

        A is an n-by-n matrix.  Restriction: n > 0.  All row and column indices
        in the triplet form must be in the range 0 to n-1.

    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.

        Ti, Tj, and Tx hold the "triplet" form of a sparse matrix.  The kth
        nonzero entry is in row i = Ti [k], column j = Tj [k], and has a
        numerical value of a_ij = Tx [k].  The row and column indices i and j
        must be in the range 0 to n-1.  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 is optional; if Tx and/or Bx
```

```
    are not present (a (double *) NULL pointer), then Bx is not computed.

Int Bp [n+1] ;      Output argument, not modified.

    Bp is an integer array of size n+1 on input.
    On output, Bp holds the "pointers" for the column form of the sparse
    matrix A.  Column j of the matrix A is held in
    Bi [(Bp [j]) ... (Bp [j+1]-1)].  The first entry, Bp [0], is zero, and
    Bp [j] <= Bp [j+1] holds for all j in the range 0 to n-1.  The value
    nz2 = Bp [n] is thus the total number of entries in the pattern of the
    matrix A.  Equivalently, the number of duplicate triplets is
    nz - Bp [n].

Int Bi [max(n,nz2)] ;       Output argument, not modified.

    Bi is an integer array of size max (n,nz2) on input, where nz2 <= nz.
    Bi is also used as workspace during the conversion, and for this use
    the size of Bi must also be at least n.

    The nonzero pattern (row indices) for column j is stored in
    Bi [(Bp [j]) ... (Bp [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-1 (the matrix is 0-based).

double Bx [nz2] ;   Output argument, not modified, of size nz2 = Bp [n].

    Bx is a double array of size nz2 on input, where nz2 <= nz.  Bx is
    optional; if Tx and/or Bx are not present (a (double *) NULL pointer),
    then Bx is not computed.  If present, Bx holds the numerical values of
    the sparse matrix A.  The nonzero pattern (row indices) for column j is
    stored in Bi [(Bp [j]) ... (Bp [j+1]-1)], and the corresponding
    numerical values are stored in Bx [(Bp [j]) ... (Bp [j+1]-1)].
```

12.3 umfpack_transpose and umfpack_l_transpose

```
int umfpack_transpose
(
    int n,
    const int Ap [ ],
    const int Ai [ ],
    const double Ax [ ],
    const int P [ ],
    const int Q [ ],
    int Cp [ ],
    int Ci [ ],
    double Cx [ ]
) ;

long umfpack_l_transpose
(
    long n,
    const long Ap [ ],
    const long Ai [ ],
    const double Ax [ ],
    const long P [ ],
    const long Q [ ],
    long Cp [ ],
    long Ci [ ],
    double Cx [ ]
) ;

int Syntax:

    #include "umfpack.h"
    int n, status, *Ap, *Ai, *P, *Q, *Cp, *Ci ;
    double *Ax, *Cx ;
    status = umfpack_transpose (n, Ap, Ai, Ax, P, Q, Cp, Ci, Cx) ;

long Syntax:

    #include "umfpack.h"
    long n, status, *Ap, *Ai, *P, *Q, *Cp, *Ci ;
    double *Ax, *Cx ;
    status = umfpack_l_transpose (n, Ap, Ai, Ax, P, Q, Cp, Ci, Cx) ;

Purpose:

    Transposes and optionally permutes a sparse matrix in row or column-form,
    C = (PAQ)'.  In Matlab notation, C = (A (P,Q))'.  Alternatively, this
    routine can be viewed as converting A (P,Q) from column-form to row-form,
    or visa versa.  Empty rows and columns may exist.
    The matrix A may be singular.

    umfpack_transpose is useful if you want to factorize A' instead of A.
    Factorizing 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', by solving
```

```
    with the sys argument "A'x=b" in umfpack_*solve.

    The input A and output C can be printed with umfpack_report matrix, and the
    permutation vectors can be printed with umfpack_report_perm.

Returns:

    UMFPACK_OK if successful.
    UMFPACK_ERROR_out_of_memory if umfpack_transpose fails to allocate a
        size-n workspace.
    UMFPACK_ERROR_argument_missing if Ai, Ap, Ci, and/or Cp are missing.
    UMFPACK_ERROR_n_nonpositive if n <= 0.
    UMFPACK_ERROR_invalid_permutation if P and/or Q are invalid.
    UMFPACK_ERROR_nz_negative if Ap [n] < 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-1.
    UMFPACK_ERROR_row_index_out_of_bounds if any row index i is < 0 or >= n.
    UMFPACK_ERROR_jumbled_matrix if the row indices in any column are not in
        ascending order.

Arguments:

    Int n ;             Input argument, not modified.

        A is an n-by-n matrix.  Restriction: n > 0.

    Int Ap [n+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].  Ap [0] must be zero, Ap [n] must be > 0, and
        Ap [j] <= Ap [j+1] and Ap [j] <= Ap [n] must be true for all j in the
        range 0 to n-1.  Empty columns are OK (that is, Ap [j] may equal Ap
        [j+1] for any j in the range 0 to n-1).

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

        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-1 (the matrix is 0-based).

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

        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 numerical values
        are stored in Ax [(Ap [j]) ... (Ap [j+1]-1)].  If Ax and/or Cx are not
        present, then the output Cx [...] is not computed, and only the pattern
        is transposed.  This is not an error condition.

    Int P [n] ;         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.

Int Q [n] ;         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.

Int Cp [n+1] ;      Output argument.

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

Int Ci [nz] ;       Output argument.

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

double Cx [nz] ;    Output argument.

    If present, these are the numerical values of the sparse matrix C,
    in the same form as the values Ax of the matrix A.  If Ax and/or Cx
    are not present, then the output Cx [...] is not computed, and only
    the pattern is transposed.  This is not an error condition.
```

13 Getting the contents of opaque objects

13.1 umfpack_get_lunz and umfpack_l_get_lunz

```
int umfpack_get_lunz
(
    int *lnz,
    int *unz,
    int *n,
    void *Numeric
) ;

long umfpack_l_get_lunz
(
    long *lnz,
    long *unz,
    long *n,
    void *Numeric
) ;

int Syntax:

    #include "umfpack.h"
    void *Numeric ;
    int status, lnz, unz, n ;
    status = umfpack_get_lunz (&lnz, &unz, &n, Numeric) ;

long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    long status, lnz, unz, n ;
    status = umfpack_l_get_lunz (&lnz, &unz, &n, 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.

    This routine may seem redundant, since n is a value originally passed to
    umfpack_symbolic by the user, and lnz and unz are available from the Info
    array.  However, the Info array is not always returned (its use is
    optional).  This routine is also useful in the context of many sparse linear
    systems, with many Numeric handles.  The user could store an array of
    Numeric objects in an array of (void *) pointers, for example.  The Info
    array is re-initialized each time an UMFPACK routine is called, and thus the
    lnz and unz information could be lost.  Lnz and unz can differ from
    different calls to umfpack_numeric with different numerical values (Ax),
    even when using the same Symbolic object.   This routine allows the LU
    factors to be extracted from the Numeric object (with umfpack_get_numeric)
    without the use of the corresponding Info array.
```

```
Returns:

    UMFPACK_OK if successful.
    UMFPACK_ERROR_invalid_Numeric_object if Numeric is not a valid object.
    UMFPACK_ERROR_argument_missing if lnz, unz, or n are (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 Li 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 ;            Output argument.

        The order of the L and U matrices.  The size of Lp and Up, as required
        by umfpack_get_numeric, is n+1, and the size of P and Q are n.  The
        value of n is identical to that passed to umfpack_symbolic.

    void *Numeric ;     Input argument, not modified.

        Numeric must point to a valid Numeric object, computed by
        umfpack_numeric.
```

13.2 umfpack_get_numeric and umfpack_l_get_numeric

```
int umfpack_get_numeric
(
    int Lp [ ],
    int Li [ ],
    double Lx [ ],
    int Up [ ],
    int Ui [ ],
    double Ux [ ],
    int P [ ],
    int Q [ ],
    void *Numeric
) ;

long umfpack_l_get_numeric
(
    long Lp [ ],
    long Li [ ],
    double Lx [ ],
    long Up [ ],
    long Ui [ ],
    double Ux [ ],
    long P [ ],
    long Q [ ],
    void *Numeric
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    long *Lp, *Li, *Up, *Ui, *P, *Q, status ;
    double *Lx, *Ux ;
    status = umfpack_l_get_numeric (Lp, Li, Lx, Up, Ui, Ux, P, Q, 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, Li, Lx, Up, Ui, Ux, P, and
    Q are not allocated by umfpack_get_numeric; they must exist on input.

Returns:
```

```
    Returns UMFPACK_OK if successful.  Returns UMFPACK_ERROR_out_of_memory
    if insufficient memory is available for the 2*n 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+1] ;      Output argument.
    Int Li [lnz] ;      Output argument.
    double Lx [lnz] ;   Output argument.

        The matrix L is returned in compressed-row form.  The column indices
        of row i and corresponding numerical values are in:

            Li [Lp [i] ... Lp [i+1]-1]
            Lx [Lp [i] ... Lp [i+1]-1]

        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, Li, and Lx are returned
        by umfpack_get_lunz.  If any one of the Lp, Li, or Lx arrays are not
        present, the L matrix is not returned.  This is not an error condition.
        Thus, if you do not want the L matrix returned, simply pass
        (Int *) NULL for Lp and Li, and (double *) NULL for Lx.
        The L matrix can be printed if n, Lp, Li, and Lx are passed to
        umfpack_report_matrix (using the "row" form).

    Int Up [n+1] ;      Output argument.
    Int Ui [unz] ;      Output argument.
    double Ux [unz] ;   Output argument.

        The 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]

        respectively.  Each column is stored in sorted order, from low row
        indices to higher.  The last entry in each column is the diagonal.  The
        sizes of Up, Ui, and Ux are returned by umfpack_get_lunz.  If any one of
        the Up, Ui, or Ux arrays are not present, the U matrix is not returned.
        This is not an error condition.  Thus, if you do not want the U matrix
        returned, simply pass (Int *) NULL for Up and Ui, and (double *) NULL
        for Ux.  The U matrix can be printed if n, Up, Ui, and Ux are passed to
        umfpack_report_matrix (using the "column" form).

    Int P [n] ;         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] ;          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_npivots in umfpack_get_symbolic for
    details.

void *Numeric ;      Input argument, not modified.

    Numeric must point to a valid Numeric object, computed by
    umfpack_numeric.
```

13.3 umfpack_get_symbolic and umfpack_l_get_symbolic

```
int umfpack_get_symbolic
(
    int *n,
    int *nz,
    int *nfr,
    int *nchains,
    int *nsparse_col,
    int Qtree [ ],
    int Front_npivots [ ],
    int Front_parent [ ],
    int Chain_start [ ],
    int Chain_maxrows [ ],
    int Chain_maxcols [ ],
    void *Symbolic
) ;

long umfpack_l_get_symbolic
(
    long *n,
    long *nz,
    long *nfr,
    long *nchains,
    long *nsparse_col,
    long Qtree [ ],
    long Front_npivots [ ],
    long Front_parent [ ],
    long Chain_start [ ],
    long Chain_maxrows [ ],
    long Chain_maxcols [ ],
    void *Symbolic
) ;

int Syntax:

    #include "umfpack.h"
    int status, n, nz, nfr, nchains, nsparse_col, *Qtree,
        *Front_npivots, *Front_parent, *Chain_start, *Chain_maxrows,
        *Chain_maxcols ;
    void *Symbolic ;
    status = umfpack_get_symbolic (&n, &nz, &nfr, &nchains, &nsparse_col,
        Qtree, Front_npivots, Front_parent, Chain_start, Chain_maxrows,
        Chain_maxcols, Symbolic) ;

long Syntax:

    #include "umfpack.h"
    long status, n, nz, nfr, nchains, nsparse_col, *Qtree,
        *Front_npivots, *Front_parent, *Chain_start, *Chain_maxrows,
        *Chain_maxcols ;
    void *Symbolic ;
    status = umfpack_l_get_symbolic (&n, &nz, &nfr, &nchains, &nsparse_col,
```

```
    Qtree, Front_npivots, Front_parent, 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 Qtree, Front_npivots, Front_parent, Chain_start, Chain_maxrows,
    and Chain_maxcols are not allocated by umfpack_get_symbolic;  they must
    exist on input.

    The Symbolic object is small.  It size, in integers, is
    (3*nchains + n + 2*nfr + 20), which is no greater than 6*n+20.  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 component of the Symbolic object to be
    returned to you, simply pass a (Int *) NULL pointer for that particular
    output argument.

    Int *n ;            Output argument.

        The dimension 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.  One or more pivots are contained in each
        frontal matrix, and the total number of pivots in the frontal matrices
        is n (see the description of nsparse_col, below).
        Thus, nfr is in the range 1 to n.

    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
    1 to nfr.

Int *nsparse_col ;  Output argument.

    This is equal to n.  It differed from n in Version 3.0.

Int Qtree [n] ;     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_npivots [nfr] ;   Output argument.

    This array should be of size at least n, in order to guarantee that it
    will be large enough to hold the output.  Only the first nfr entries
    are used, however.  The kth frontal matrix holds Front_npivots [k] pivot
    columns.  Thus, the first frontal matrix, front 0, is used to factorize
    the first Front_npivots [0] columns; these correspond to the original
    columns Qtree [0] through Qtree [Front_npivots [0]-1].  The next frontal
    matrix is used to factorize the next Front_npivots [1] columns, which
    are thus the original columns Qtree [Front_npivots [0]] through
    Qtree [Front_npivots [0] + Front_npivots [1] - 1], and so on.
    The sum of Front_npivots [0..nfr-1] is equal to n.

    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_npivots [0]-1] is
    some permutation of the columns Qtree [0] through
    Qtree [Front_npivots [0]-1].  For second frontal matrix,
    Q [Front_npivots [0]] through Q [Front_npivots [0] + Front_npivots[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 [nfr] ;    Output argument.

    This array should be of size at least n, in order to guarantee that it
    will be large enough to hold the output.  Only the first nfr entries
    are used, however.  Front_parent [0..nfr-1] holds the supernodal column
    elimination tree.  Each node in the tree corresponds to a single frontal
    matrix.  The parent of node f is Front_parent [f].

Int Chain_start [nchains+1] ;       Output argument.

    This array should be of size at least n+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 [nchains] ;       Output argument.
Int Chain_maxcols [nchains] ;       Output argument.

    These arrays should be of size at least n, in order to guarantee that
    they will be large enough to hold the output.  Only the first nchains
    entries of Chain_maxrows and Chain_maxcols 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 and umfpack_l_report_status

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

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

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL] ;
    long status ;
    umfpack_l_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.  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 and umfpack_l_report_control

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

void umfpack_l_report_control
(
    const double Control [UMFPACK_CONTROL]
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL] ;
    umfpack_l_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.  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 and umfpack_l_report_info

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

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

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
    umfpack_l_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.  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 and umfpack_l_report_matrix

```
int umfpack_report_matrix
(
    const char name [ ],
    int n,
    const int Ap [ ],
    const int Ai [ ],
    const double Ax [ ],
    const char form [ ],
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_l_report_matrix
(
    const char name [ ],
    long n,
    const long Ap [ ],
    const long Ai [ ],
    const double Ax [ ],
    const char form [ ],
    const double Control [UMFPACK_CONTROL]
) ;

int Syntax:

    #include "umfpack.h"
    int n, *Ap, *Ai, status ;
    double *Ax, Control [UMFPACK_CONTROL] ;
    status = umfpack_report_matrix ("A", n, Ap, Ai, Ax, "column", Control) ;
or:
    status = umfpack_report_matrix ("A", n, Ap, Ai, Ax, "row", Control) ;

long Syntax:

    #include "umfpack.h"
    long n, *Ap, *Ai, status ;
    double *Ax, Control [UMFPACK_CONTROL] ;
    status = umfpack_l_report_matrix ("A", n, Ap, Ai, Ax, "column", Control) ;
or:
    status = umfpack_l_report_matrix ("A", n, Ap, Ai, Ax, "row", 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:

    UMFPACK_OK if the matrix is valid and non-singular.
```

```
    UMFPACK_ERROR_singular_matrix if the matrix is structurally singular but
        otherwise valid.  It has one or more rows or columns with no entries.
        This test is made without considering the numerical values, but by just
        looking at the pattern of the entries.  Thus, all structurally singular
        matrices are numerically singular, but not all numerically singular
        matrices are structurally singular.  The matrix may be operated on by
        the matrix manipulation routines (umfpack_transpose,
        umfpack_col_to_triplet) but it may not be analyzed by umfpack_*symbolic
        or factorized by umfpack_numeric).

    UMFPACK_ERROR_n_nonpositive if n <= 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 n-1.
    UMFPACK_ERROR_jumbled_matrix if the row indices in any column are not in
        ascending order, or contain duplicates.

Arguments:

    char name [ ] ;     Input argument, not modified.

        The name of the matrix.  This is optional; no name is printed if
        a (char *) NULL pointer is passed.

    Int n ;             Input argument, not modified.

        A is an n-by-n matrix.  Restriction: n > 0.

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

        Ap is an integer array of size n+1.  If the form argument is "column",
        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)].  If form is "row", then Ap holds the
        row pointers.  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.
        Restriction:  Ap [0] == 0 and Ap [n] > 0.

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

        If form is "column", 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-1 (the matrix is 0-based).
```

```
    If form is "row", then 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-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 form is "row", then 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)].

    If form is "column", then the nonzero pattern (column indices) for row 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)].

    No numerical values are printed if Ax is a (double *) NULL pointer.

char *form ;        Input argument, not modified.

    The matrix is in row-oriented form if form is "row".  Otherwise,
    the matrix is in column-oriented form.  The form argument may be
    (char *) NULL, in which case 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.  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 and umfpack_l_report_numeric

```
int umfpack_report_numeric
(
    const char name [ ],
    void *Numeric,
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_l_report_numeric
(
    const char name [ ],
    void *Numeric,
    const double Control [UMFPACK_CONTROL]
) ;

int Syntax:

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

long Syntax:

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

Purpose:

    Verifies and prints a Numeric object.  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:

    char name [ ] ;     Input argument, not modified.

        The name of the Numeric object.  This is optional; no name is printed
        if a (char *) NULL pointer is passed.

    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.  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 and umfpack_l_report_perm

```
int umfpack_report_perm
(
    const char name [ ],
    int n,
    const int P [ ],
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_l_report_perm
(
    const char name [ ],
    long n,
    const long P [ ],
    const double Control [UMFPACK_CONTROL]
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    long n, *P, status ;
    double Control [UMFPACK_CONTROL] ;
    status = umfpack_l_report_perm ("P", n, P, 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 P is (Int *) NULL, which is not an error condition.

    UMFPACK_ERROR_n_nonpositive if n <= 0.
    UMFPACK_ERROR_out_of_memory if out of memory.
    UMFPACK_ERROR_invalid_permutation if P is not a valid permutation vector.

Arguments:

    char name [ ] ;     Input argument, not modified.
```

```
    The name of the permutation vector.  This is optional; no name is
    printed if a (char *) NULL pointer is passed.

Int n ;             Input argument, not modified.

    P is an integer vector of size n.  Restriction: n > 0.

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

    A permutation vector of size n.  If P is not present (a (Int *) NULL
    pointer, then P is assumed to be the identity permutation.  This is
    consistent with its use as an input argument to umfpack_qsymbolic.
    If P is present, the entries in P must range between 0 and n-1, and no
    duplicates may exists.

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.  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 and umfpack_l_report_symbolic

```
int umfpack_report_symbolic
(
    const char name [ ],
    void *Symbolic,
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_l_report_symbolic
(
    const char name [ ],
    void *Symbolic,
    const double Control [UMFPACK_CONTROL]
) ;

int Syntax:

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

long Syntax:

    #include "umfpack.h"
    void *Symbolic ;
    double Control [UMFPACK_CONTROL] ;
    long status ;
    status = umfpack_l_report_symbolic ("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:

    char name [ ] ;     Input argument, not modified.

        The name of the Symbolic object.  This is optional; no name is printed
        if a (char *) NULL pointer is passed.

    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.  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 and umfpack_l_report_triplet

```
int umfpack_report_triplet
(
    const char name [ ],
    int n,
    int nz,
    const int Ti [ ],
    const int Tj [ ],
    const double Tx [ ],
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_l_report_triplet
(
    const char name [ ],
    long n,
    long nz,
    const long Ti [ ],
    const long Tj [ ],
    const double Tx [ ],
    const double Control [UMFPACK_CONTROL]
) ;

int Syntax:

    #include "umfpack.h"
    int n, nz, *Ti, *Tj, status ;
    double *Tx, Control [UMFPACK_CONTROL] ;
    status = umfpack_report_triplet ("Triplet", n, nz, Ti, Tj, Tx, Control) ;

long Syntax:

    #include "umfpack.h"
    long n, nz, *Ti, *Tj, status ;
    double *Tx, Control [UMFPACK_CONTROL] ;
    status = umfpack_l_report_triplet ("Triplet", n, nz, Ti, Tj, Tx, 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 <= 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-1.

    Arguments:

        char name [ ] ;     Input argument, not modified.

            The name of the matrix.  This is optional; no name is printed if
            a (char *) NULL pointer is passed.

        Int n ;             Input argument, not modified.

            A is an n-by-n 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.

            Ti, Tj, and Tx hold the "triplet" form of a sparse matrix.  The kth
            nonzero entry is in row i = Ti [k], column j = Tj [k], and has a
            numerical value of a_ij = Tx [k].  The row and column indices i and j
            must be in the range 0 to n-1.  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 is optional;
            if Tx is not present (a (double *) NULL pointer), then the numerical
            values are not printed.

        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.  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 and umfpack_l_report_vector

```
int umfpack_report_vector
(
    const char name [ ],
    int n,
    const double X [ ],
    const double Control [UMFPACK_CONTROL]
) ;

long umfpack_l_report_vector
(
    const char name [ ],
    long n,
    const double X [ ],
    const double Control [UMFPACK_CONTROL]
) ;

int Syntax:

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

long Syntax:

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

Purpose:

    Verifies and prints a real 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 is missing.
    UMFPACK_ERROR_n_nonpositive if n <= 0.

Arguments:

    char name [ ] ;      Input argument, not modified.

        The name of the vector.  This is optional; no name is
        printed if a (char *) NULL pointer is passed.
```

```
Int n ;             Input argument, not modified.

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

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

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

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.  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 both int and long versions):

    #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).

    This routine uses the Unix getrusage ( ) routine, if available.  It is not
    subject to overflow.  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.

Arguments:

    None.
```

16 `umfpack.h` include file

```
/*
    This is the umfpack.h include file, and should be included in all user code
    that uses UMFPACK.  Do not include any of the umf_* header files in user
    code.  All routines in UMFPACK starting with "umfpack_" are user-callable
    (the 24 routines listed below).  All other routines are prefixed "umf_",
    and are not user-callable.
*/

#ifndef UMFPACK_H
#define UMFPACK_H

/* -------------------------------------------------------------------------- */
/* size of Info and Control arrays */
/* -------------------------------------------------------------------------- */

#define UMFPACK_INFO 90
#define UMFPACK_CONTROL 20

/* -------------------------------------------------------------------------- */
/* User-callable routines */
/* -------------------------------------------------------------------------- */

/* Primary routines: */
#include "umfpack_symbolic.h"
#include "umfpack_numeric.h"
#include "umfpack_solve.h"
#include "umfpack_free_symbolic.h"
#include "umfpack_free_numeric.h"

/* Alternative routines: */
#include "umfpack_defaults.h"
#include "umfpack_qsymbolic.h"
#include "umfpack_wsolve.h"

/* Matrix manipulation routines: */
#include "umfpack_triplet_to_col.h"
#include "umfpack_col_to_triplet.h"
#include "umfpack_transpose.h"

/* Getting the contents of the Symbolic and Numeric opaque objects: */
#include "umfpack_get_lunz.h"
#include "umfpack_get_numeric.h"
#include "umfpack_get_symbolic.h"

/* Reporting routines (the above 14 routines print nothing): */
#include "umfpack_report_status.h"
#include "umfpack_report_info.h"
#include "umfpack_report_control.h"
#include "umfpack_report_matrix.h"
#include "umfpack_report_triplet.h"
#include "umfpack_report_symbolic.h"
```

```
#include "umfpack_report_numeric.h"
#include "umfpack_report_perm.h"
#include "umfpack_report_vector.h"

/* Utility routines: */
#include "umfpack_timer.h"


/* -------------------------------------------------------------------------- */
/* Version, copyright, and license */
/* -------------------------------------------------------------------------- */

#define UMFPACK_VERSION "UMFPACK V3.2"

#define UMFPACK_COPYRIGHT \
"UMFPACK:  Copyright (c) 2002 by Timothy A. Davis, University of Florida,\n" \
"davis@cise.ufl.edu.  All Rights Reserved.\n"

#define UMFPACK_LICENSE \
"\nUMFPACK License:\n" \
"\n" \
"    Your use or distribution of UMFPACK or any derivative code implies that\n"\
"    you agree to this License.\n" \
"\n" \
"    THIS MATERIAL IS PROVIDED AS IS, WITH ABSOLUTELY NO WARRANTY\n" \
"    EXPRESSED OR IMPLIED.  ANY USE IS AT YOUR OWN RISK.\n" \
"\n" \
"    Permission is hereby granted to use or copy this program, provided\n" \
"    that the Copyright, this License, and the Availability of the original\n" \
"    version is retained on all copies.  User documentation of any code that\n"\
"    uses this code or any derivative code must cite the Copyright, this\n" \
"    License, the Availability note, and \"Used by permission.\"  If this\n" \
"    code or any derivative code is accessible from within MATLAB, then\n" \
"    typing \"help umfpack\" must cite the Copyright, and \"type umfpack\"\n" \
"    must also cite this License and the Availability note.  Permission to\n" \
"    modify the code and to distribute modified code is granted, provided\n" \
"    the Copyright, this License, and the Availability note are retained,\n" \
"    and a notice that the code was modified is included.  This software\n" \
"    was developed with support from the National Science Foundation, and\n" \
"    is provided to you free of charge.\n" \
"\n" \
"Availability:  http://www.cise.ufl.edu/research/sparse\n" \
"\n"


/* -------------------------------------------------------------------------- */
/* contents of Info */
/* -------------------------------------------------------------------------- */

/* Note that umfpack_report.m must coincide with these definitions. */

/* returned by all routines that use Info: */
#define UMFPACK_STATUS 0
```

```
#define UMFPACK_N 1
#define UMFPACK_NZ 2

/* computed in UMFPACK_*symbolic and UMFPACK_numeric: */
#define UMFPACK_SIZE_OF_UNIT 3

/* computed in UMFPACK_*symbolic: */
#define UMFPACK_SIZE_OF_INT 4
#define UMFPACK_SIZE_OF_LONG 5
#define UMFPACK_SIZE_OF_POINTER 6
#define UMFPACK_SIZE_OF_ENTRY 7
#define UMFPACK_NDENSE_ROW 8
#define UMFPACK_NEMPTY_ROW 9
#define UMFPACK_NDENSE_COL 10
#define UMFPACK_NEMPTY_COL 11
#define UMFPACK_SYMBOLIC_DEFRAG 12
#define UMFPACK_SYMBOLIC_PEAK_MEMORY 13
#define UMFPACK_SYMBOLIC_SIZE 14
#define UMFPACK_SYMBOLIC_TIME 15

/* Info [16..19] unused */

/* estimates computed in UMFPACK_*symbolic: */
#define UMFPACK_NUMERIC_SIZE_ESTIMATE 20
#define UMFPACK_PEAK_MEMORY_ESTIMATE 21
#define UMFPACK_FLOPS_ESTIMATE 22
#define UMFPACK_LNZ_ESTIMATE 23
#define UMFPACK_UNZ_ESTIMATE 24
#define UMFPACK_VARIABLE_INIT_ESTIMATE 25
#define UMFPACK_VARIABLE_PEAK_ESTIMATE 26
#define UMFPACK_VARIABLE_FINAL_ESTIMATE 27
#define UMFPACK_MAX_FRONT_SIZE_ESTIMATE 28

/* Info [29..39] unused */

/* exact values, (estimates shown above) computed in UMFPACK_numeric: */
#define UMFPACK_NUMERIC_SIZE 40
#define UMFPACK_PEAK_MEMORY 41
#define UMFPACK_FLOPS 42
#define UMFPACK_LNZ 43
#define UMFPACK_UNZ 44
#define UMFPACK_VARIABLE_INIT 45
#define UMFPACK_VARIABLE_PEAK 46
#define UMFPACK_VARIABLE_FINAL 47
#define UMFPACK_MAX_FRONT_SIZE 48

/* Info [49..59] unused */

/* computed in UMFPACK_numeric: */
#define UMFPACK_NUMERIC_DEFRAG 60
#define UMFPACK_NUMERIC_REALLOC 61
#define UMFPACK_NUMERIC_COSTLY_REALLOC 62
#define UMFPACK_COMPRESSED_PATTERN 63
```

```
#define UMFPACK_LU_ENTRIES 64
#define UMFPACK_NUMERIC_TIME 65

/* Info [66..79] unused */

/* computed in UMFPACK_solve: */
#define UMFPACK_IR_TAKEN 80
#define UMFPACK_IR_ATTEMPTED 81
#define UMFPACK_OMEGA1 82
#define UMFPACK_OMEGA2 83
#define UMFPACK_SOLVE_FLOPS 84
#define UMFPACK_SOLVE_TIME 85

/* Info [86..89] unused */

/* Unused parts of Info may be used in future versions of UMFPACK. */


/* -------------------------------------------------------------------------- */
/* contents of Control */
/* -------------------------------------------------------------------------- */

/* used in all UMFPACK_report_* routines: */
#define UMFPACK_PRL 0

/* used in UMFPACK_*symbolic only: */
#define UMFPACK_DENSE_ROW 1
#define UMFPACK_DENSE_COL 2

/* used in UMFPACK_numeric only: */
#define UMFPACK_PIVOT_TOLERANCE 3
#define UMFPACK_BLOCK_SIZE 4
#define UMFPACK_RELAXED_AMALGAMATION 5
#define UMFPACK_ALLOC_INIT 6
#define UMFPACK_PIVOT_OPTION 12
#define UMFPACK_RELAXED2_AMALGAMATION 13
#define UMFPACK_RELAXED3_AMALGAMATION 14

/* used in UMFPACK_*solve only: */
#define UMFPACK_IRSTEP 7

/* compile-time settings - Control [8..11] cannot be changed at run time: */
#define UMFPACK_COMPILED_WITH_BLAS 8
#define UMFPACK_COMPILED_FOR_MATLAB 9
#define UMFPACK_COMPILED_WITH_GETRUSAGE 10
#define UMFPACK_COMPILED_IN_DEBUG_MODE 11

/* Control [15...19] unused */

/* Unused parts of Control may be used in future versions of UMFPACK. */


/* -------------------------------------------------------------------------- */
```

```
/* default values of Control [0..7,12..13]: */
/* -------------------------------------------------------------------------- */

/* Note that the default block sized changed for Version 3.1 and following. */

#define UMFPACK_DEFAULT_PRL 1
#define UMFPACK_DEFAULT_DENSE_ROW 0.2
#define UMFPACK_DEFAULT_DENSE_COL 0.2
#define UMFPACK_DEFAULT_PIVOT_TOLERANCE 0.1
#define UMFPACK_DEFAULT_BLOCK_SIZE 24
#define UMFPACK_DEFAULT_RELAXED_AMALGAMATION 0.25
#define UMFPACK_DEFAULT_RELAXED2_AMALGAMATION 0.1
#define UMFPACK_DEFAULT_RELAXED3_AMALGAMATION 0.125
#define UMFPACK_DEFAULT_ALLOC_INIT 0.7
#define UMFPACK_DEFAULT_IRSTEP 2
#define UMFPACK_DEFAULT_PIVOT_OPTION 0

/* default values of Control [0..7,12..13] may change in future versions */
/* of UMFPACK. */

/* -------------------------------------------------------------------------- */
/* status codes */
/* -------------------------------------------------------------------------- */

#define UMFPACK_OK (0)
#define UMFPACK_ERROR_out_of_memory (-1)
#define UMFPACK_ERROR_singular_matrix (-2)
#define UMFPACK_ERROR_invalid_Numeric_object (-3)
#define UMFPACK_ERROR_invalid_Symbolic_object (-4)
#define UMFPACK_ERROR_argument_missing (-5)
#define UMFPACK_ERROR_n_nonpositive (-6)
#define UMFPACK_ERROR_nz_negative (-7)
#define UMFPACK_ERROR_jumbled_matrix (-8)
#define UMFPACK_ERROR_Ap0_nonzero (-9)
#define UMFPACK_ERROR_row_index_out_of_bounds (-10)
#define UMFPACK_ERROR_different_pattern (-11)
#define UMFPACK_ERROR_col_length_negative (-12)
#define UMFPACK_ERROR_invalid_system (-13)
#define UMFPACK_ERROR_invalid_triplet (-14)
#define UMFPACK_ERROR_invalid_permutation (-15)
#define UMFPACK_ERROR_problem_too_large (-16)
#define UMFPACK_ERROR_internal_error (-911)

#endif /* UMFPACK_H */
```

17 Demo C main program, `umfpack_demo.c`

The `umfpack_l_demo.c` is identical except that all the routine names are changed, and `long`'s are used instead of `int`'s.

```
/*
  A demo of UMFPACK Version 3.2:  See umfpack_demo.m for a (roughly)
  equivalent Matlab version.  The only difference is that while the Matlab
  umfpack mexFunction provides separate access to umfpack_symbolic, via
  umfpack (A, 'symbolic'), it does not use its output for a subsequent
  numerical factorization.  Thus, you will find that the output of this
  program and the Matlab diary are slightly different.  The Matlab output also
  uses 1-based matrix row and column indices, not 0-based (the internal
  represention is the same).

  First, factor and solve a 5-by-5 system, Ax=b, using default parameters,

      [ 2  3  0  0  0 ]      [  8 ]                  [ 1 ]
      [ 3  0  4  0  6 ]      [ 45 ]                  [ 2 ]
  A = [ 0 -1 -3  2  0 ], b = [ -3 ]. Solution is x = [ 3 ].
      [ 0  0  1  0  0 ]      [  3 ]                  [ 4 ]
      [ 0  4  2  0  1 ]      [ 19 ]                  [ 5 ]

  Then solve A'x=b, with solution:
        x = [  1.8158  1.4561 1.5000 -24.8509 10.2632 ]'
  using the factors of A.   Modify one entry (A (1,4) = 0, where the row and
  column indices range from 0 to 4, obtaining the system:

      [ 2  3  0  0  0 ]      [  8 ]                  [ 11.0    ]
      [ 3  0  4  0  0 ]      [ 45 ]                  [ -4.6667 ]
  A = [ 0 -1 -3  2  0 ], b = [ -3 ]. Solution is x = [  3.0    ].
      [ 0  0  1  0  0 ]      [  3 ]                  [  0.6667 ]
      [ 0  4  2  0  1 ]      [ 19 ]                  [ 31.6667 ]

  The pattern of A has not changed (it has explicitly zero entry), so a
  reanalysis with umfpack_symbolic does not need to be done (the Matlab
  umfpack_demo.m will need to redo it, because the Matlab caller is not
  provided with the Symbolic object).  Refactorize (with umfpack_numeric),
  and solve Ax=b.  Note that the pivot ordering has changed.  Next, change all
  of the entries in A, but not the pattern.  The system becomes

      [ 2 13  0  0  0 ]      [  8 ]                  [  8.5012 ]
      [ 2  0 23  0 39 ]      [ 45 ]                  [ -0.6925 ]
  A = [ 0  7 15 30  0 ], b = [ -3 ]. Solution is x = [  0.1667 ].
      [ 0  0 18  0  0 ]      [  3 ]                  [ -0.0218 ]
      [ 0 10 18  0 37 ]      [ 19 ]                  [  0.6196 ]

  Finally, compute B = A', and do the symbolic and numeric factorization of B.
  Factorizing A' can sometimes be better than factorizing A itself (less work
  and memory usage).  Solve B'x=b twice; the solution is the same as the
  solution to Ax=b for the above A.
*/
```

```
/* -------------------------------------------------------------------------- */
/* definitions */
/* -------------------------------------------------------------------------- */

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

#define ABS(x) ((x) >= 0 ? (x) : -(x))
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
#ifndef TRUE
#define TRUE (1)
#endif
#ifndef FALSE
#define FALSE (0)
#endif

/* -------------------------------------------------------------------------- */
/* triplet form of the matrix.  The triplets can be in any order. */
/* -------------------------------------------------------------------------- */

static int    n = 5 ;
static int    nz = 12 ;
static int    Arow [ ] = { 0,  4,  1,  1,   2,   2,  0,  1,  2,  3,  4,  4} ;
static int    Acol [ ] = { 0,  4,  0,  2,   1,   2,  1,  4,  3,  2,  1,  2} ;
static double Aval [ ] = {2., 1., 3., 4., -1., -3., 3., 6., 2., 1., 4., 2.} ;
static double b [ ] = {8., 45., -3., 3., 19.} ;
static double x [5] ;
static double r [5] ;

/* -------------------------------------------------------------------------- */
/* error: print a message and exit */
/* -------------------------------------------------------------------------- */

static void error
(
    char *message
)
{
    printf ("\n\n====== error: %s =====\n\n", message) ;
    exit (1) ;
}


/* -------------------------------------------------------------------------- */
/* resid: compute the residual, r = Ax-b or r = A'x=b and return maxnorm (r) */
/* -------------------------------------------------------------------------- */

static double resid
(
    int n,
    int Ap [ ],
    int Ai [ ],
```

```
    double Ax [ ],
    double x [ ],
    double r [ ],
    int transpose
)
{
    int i, j, p ;
    double norm ;

    for (i = 0 ; i < n ; i++)
    {
        r [i] = -b [i] ;
    }
    if (transpose)
    {
        for (j = 0 ; j < n ; j++)
        {
            for (p = Ap [j] ; p < Ap [j+1] ; p++)
            {
                i = Ai [p] ;
                r [j] += Ax [p] * x [i] ;
            }
        }
    }
    else
    {
        for (j = 0 ; j < n ; j++)
        {
            for (p = Ap [j] ; p < Ap [j+1] ; p++)
            {
                i = Ai [p] ;
                r [i] += Ax [p] * x [j] ;
            }
        }
    }
    norm = 0. ;
    for (i = 0 ; i < n ; i++)
    {
        norm = MAX (norm, ABS (r [i])) ;
    }
    return (norm) ;
}


/* -------------------------------------------------------------------------- */
/* main program */
/* -------------------------------------------------------------------------- */

/*ARGSUSED0*/   /* argc and argv are unused */
int main
(
    int argc,
    char **argv
```

```
)
{
    double Info [UMFPACK_INFO], Control [UMFPACK_CONTROL], *Ax, *Bx, *Lx, *Ux,
        *W, *Y, *Z, *S, t ;
    int *Ap, *Ai, *Bp, *Bi, row, col, p, lnz, unz, nn, *Lp, *Li, *Ui, *Up,
        *P, *Q, *Lj, i, j, k, anz, nfr, nchains, nsparse_col, *Qtree, fnpiv,
        status, *Front_npivots, *Front_parent, *Chain_start, *Wi,
        *Chain_maxrows, *Chain_maxcols ;
    void *Symbolic, *Numeric ;

    /* ---------------------------------------------------------------------- */
    /* initializations */
    /* ---------------------------------------------------------------------- */

    t = umfpack_timer ( ) ;

    printf ("\n%s demo:\n", UMFPACK_VERSION) ;

    /* get the default control parameters */
    umfpack_defaults (Control) ;

    /* change the default print level for this demo */
    /* (otherwise, nothing will print) */
    Control [UMFPACK_PRL] = 6 ;

    /* print the license agreement */
    umfpack_report_status (Control, UMFPACK_OK) ;
    Control [UMFPACK_PRL] = 5 ;

    /* print the control parameters */
    umfpack_report_control (Control) ;

    /* ---------------------------------------------------------------------- */
    /* print A and b, and convert A to column-form */
    /* ---------------------------------------------------------------------- */

    /* print the right-hand-side */
    (void) umfpack_report_vector ("b", n, b, Control) ;

    /* print the triplet form of the matrix */
    (void) umfpack_report_triplet ("A", n, nz, Arow, Acol, Aval, Control) ;

    /* convert to column form */
    Ap = (int *) malloc ((n+1) * sizeof (int)) ;
    Ai = (int *) malloc (nz * sizeof (int)) ;
    Ax = (double *) malloc (nz * sizeof (double)) ;
    if (!Ap || !Ai || !Ax)
    {
        error ("out of memory") ;
    }
    status = umfpack_triplet_to_col (n, nz, Arow, Acol, Aval, Ap, Ai, Ax) ;
    if (status != UMFPACK_OK)
    {
```

```
    umfpack_report_status (Control, status) ;
    error ("umfpack_triplet_to_col failed") ;
}

/* print the column-form of A */
(void) umfpack_report_matrix ("A", n, Ap, Ai, Ax, "column", Control) ;

/* ---------------------------------------------------------------------- */
/* symbolic factorization */
/* ---------------------------------------------------------------------- */

status = umfpack_symbolic (n, Ap, Ai, &Symbolic, Control, Info) ;
if (status != UMFPACK_OK)
{
    umfpack_report_info (Control, Info) ;
    umfpack_report_status (Control, status) ;
    error ("umfpack_symbolic failed") ;
}

/* print the symbolic factorization */
(void) umfpack_report_symbolic ("Symbolic factorization of A",
    Symbolic, Control) ;

/* ---------------------------------------------------------------------- */
/* numeric factorization */
/* ---------------------------------------------------------------------- */

status = umfpack_numeric (Ap, Ai, Ax, Symbolic, &Numeric, Control, Info) ;
if (status != UMFPACK_OK)
{
    umfpack_report_info (Control, Info) ;
    umfpack_report_status (Control, status) ;
    error ("umfpack_numeric failed") ;
}

/* print the numeric factorization */
(void) umfpack_report_numeric ("Numeric factorization of A",
    Numeric, Control) ;

/* ---------------------------------------------------------------------- */
/* solve Ax=b */
/* ---------------------------------------------------------------------- */

status = umfpack_solve ("Ax=b", Ap, Ai, Ax, x, b, Numeric, Control, Info) ;
umfpack_report_info (Control, Info) ;
umfpack_report_status (Control, status) ;
if (status != UMFPACK_OK)
{
    error ("umfpack_solve failed") ;
}
(void) umfpack_report_vector ("x (solution of Ax=b)", n, x, Control) ;
printf ("maxnorm of residual: %g\n\n", resid (n, Ap, Ai, Ax, x, r, FALSE)) ;
```

```
/* -------------------------------------------------------------------------- */
/* solve A'x=b */
/* -------------------------------------------------------------------------- */

status = umfpack_solve ("A'x=b", Ap, Ai, Ax, x, b, Numeric, Control, Info) ;
umfpack_report_info (Control, Info) ;
if (status != UMFPACK_OK)
{
    error ("umfpack_solve failed") ;
}
(void) umfpack_report_vector ("x (solution of A'x=b)", n, x, Control) ;
printf ("maxnorm of residual: %g\n\n", resid (n, Ap, Ai, Ax, x, r, TRUE)) ;

/* -------------------------------------------------------------------------- */
/* modify one numerical value in the column-form of A */
/* -------------------------------------------------------------------------- */

/* change A (1,4), look for row index 1 in column 4. */
row = 1 ;
col = 4 ;
for (p = Ap [col] ; p < Ap [col+1] ; p++)
{
    if (row == Ai [p])
    {
        printf ("\nchanging A (%d,%d) from %g", row, col, Ax [p]) ;
        Ax [p] = 0.0 ;
        printf (" to %g\n", Ax [p]) ;
        break ;
    }
}
(void) umfpack_report_matrix ("modified A", n, Ap, Ai, Ax, "column",
    Control) ;

/* -------------------------------------------------------------------------- */
/* redo the numeric factorization */
/* -------------------------------------------------------------------------- */

/* The pattern (Ap and Ai) hasn't changed, so the symbolic factorization */
/* doesn't have to be redone, no matter how much we change Ax. */

/* We don't need the Numeric object any more, so free it. */
umfpack_free_numeric (&Numeric) ;

/* Note that a memory leak would have occured if the old Numeric */
/* had not been free'd with umfpack_free_numeric above. */
status = umfpack_numeric (Ap, Ai, Ax, Symbolic, &Numeric, Control, Info) ;
if (status != UMFPACK_OK)
{
    umfpack_report_info (Control, Info) ;
    umfpack_report_status (Control, status) ;
    error ("umfpack_numeric failed") ;
}
(void) umfpack_report_numeric ("Numeric factorization of modified A",
```

```
    Numeric, Control) ;

/* ---------------------------------------------------------------------- */
/* solve Ax=b, with the modified A */
/* ---------------------------------------------------------------------- */

status = umfpack_solve ("Ax=b", Ap, Ai, Ax, x, b, Numeric, Control, Info) ;
umfpack_report_info (Control, Info) ;
if (status != UMFPACK_OK)
{
    umfpack_report_status (Control, status) ;
    error ("umfpack_solve failed") ;
}
(void) umfpack_report_vector ("x (with modified A)", n, x, Control) ;
printf ("maxnorm of residual: %g\n\n", resid (n, Ap, Ai, Ax, x, r, FALSE)) ;

/* ---------------------------------------------------------------------- */
/* modify all of the numerical values of A, but not the pattern */
/* ---------------------------------------------------------------------- */

for (col = 0 ; col < n ; col++)
{
    for (p = Ap [col] ; p < Ap [col+1] ; p++)
    {
        row = Ai [p] ;
        printf ("changing A (%d,%d) from %g", row, col, Ax [p]) ;
        Ax [p] = Ax [p] + col*10 - row ;
        printf (" to %g\n", Ax [p]) ;
    }
}
(void) umfpack_report_matrix ("completely modified A (same pattern)",
    n, Ap, Ai, Ax, "column", Control) ;

/* ---------------------------------------------------------------------- */
/* redo the numeric factorization */
/* ---------------------------------------------------------------------- */

umfpack_free_numeric (&Numeric) ;
status = umfpack_numeric (Ap, Ai, Ax, Symbolic, &Numeric, Control, Info) ;
if (status != UMFPACK_OK)
{
    umfpack_report_info (Control, Info) ;
    umfpack_report_status (Control, status) ;
    error ("umfpack_numeric failed") ;
}
(void) umfpack_report_numeric (
"Numeric factorization of completely modified A", Numeric, Control) ;

/* ---------------------------------------------------------------------- */
/* solve Ax=b, with the modified A */
/* ---------------------------------------------------------------------- */

status = umfpack_solve ("Ax=b", Ap, Ai, Ax, x, b, Numeric, Control, Info) ;
```

```
umfpack_report_info (Control, Info) ;
if (status != UMFPACK_OK)
{
    umfpack_report_status (Control, status) ;
    error ("umfpack_solve failed") ;
}
(void) umfpack_report_vector ("x (with completely modified A)",
    n, x, Control) ;
printf ("maxnorm of residual: %g\n\n", resid (n, Ap, Ai, Ax, x, r, FALSE)) ;

/* ------------------------------------------------------------------------ */
/* free the symbolic and numeric factorization */
/* ------------------------------------------------------------------------ */

umfpack_free_symbolic (&Symbolic) ;
umfpack_free_numeric (&Numeric) ;

/* ------------------------------------------------------------------------ */
/* B = transpose of A */
/* ------------------------------------------------------------------------ */

Bp = (int *) malloc ((n+1) * sizeof (int)) ;
Bi = (int *) malloc (nz * sizeof (int)) ;
Bx = (double *) malloc (nz * sizeof (double)) ;
if (!Bp || !Bi || !Bx)
{
    error ("out of memory") ;
}
status = umfpack_transpose (n, Ap, Ai, Ax, (int *) NULL, (int *) NULL,
    Bp, Bi, Bx) ;
if (status != UMFPACK_OK)
{
    umfpack_report_status (Control, status) ;
    error ("umfpack_transpose failed") ;
}
(void) umfpack_report_matrix ("B (transpose of A)",
    n, Bp, Bi, Bx, "column", Control) ;

/* ------------------------------------------------------------------------ */
/* symbolic factorization of B */
/* ------------------------------------------------------------------------ */

status = umfpack_symbolic (n, Bp, Bi, &Symbolic, Control, Info) ;
if (status != UMFPACK_OK)
{
    umfpack_report_info (Control, Info) ;
    umfpack_report_status (Control, status) ;
    error ("umfpack_symbolic failed") ;
}
(void) umfpack_report_symbolic ("Symbolic factorization of B",
    Symbolic, Control) ;

/* ------------------------------------------------------------------------ */
```

```
/* copy the contents of Symbolic into user arrays print them */
/* ---------------------------------------------------------------------- */

printf ("\nGet the contents of the Symbolic object for B:\n") ;
printf ("(compare with umfpack_report_symbolic output, above)\n") ;
Qtree = (int *) malloc (n * sizeof (int)) ;
Front_npivots = (int *) malloc (n * sizeof (int)) ;
Front_parent = (int *) malloc (n * sizeof (int)) ;
Chain_start = (int *) malloc ((n+1) * sizeof (int)) ;
Chain_maxrows = (int *) malloc (n * sizeof (int)) ;
Chain_maxcols = (int *) malloc (n * sizeof (int)) ;
if (!Qtree || !Front_npivots || !Front_parent || !Chain_start ||
    !Chain_maxrows || !Chain_maxcols)
{
    error ("out of memory") ;
}

status = umfpack_get_symbolic (&nn, &anz, &nfr, &nchains, &nsparse_col,
    Qtree, Front_npivots, Front_parent, Chain_start,
    Chain_maxrows, Chain_maxcols, Symbolic) ;

printf ("From the Symbolic object, B is of dimension n = %d\n", nn) ;
printf ("   with nz = %d, number of fronts = %d,\n", nz, nfr) ;
printf ("   number of frontal matrix chains = %d\n", nchains) ;

printf ("\nPivot columns in each front, and parent of each front:\n") ;
k = 0 ;
for (i = 0 ; i < nfr ; i++)
{
    fnpiv = Front_npivots [i] ;
    printf ("    Front %d: parent front: %d number of pivots: %d\n",
            i, Front_parent [i], fnpiv) ;
    for (j = 0 ; j < fnpiv ; j++)
    {
        col = Qtree [k] ;
        printf (
            "        %d-th pivot column is column %d in original matrix\n",
            k, col) ;
        k++ ;
    }
}

printf ("\nNote that the column ordering, above, will be refined\n") ;
printf ("in the numeric factorization below.  The assignment of pivot\n") ;
printf ("columns to frontal matrices will always remain unchanged.\n") ;

printf ("\nTotal number of pivot columns in frontal matrices: %d\n", k) ;

printf ("\nFrontal matrix chains:\n") ;
for (j = 0 ; j < nchains ; j++)
{
    printf ("   Frontal matrices %d to %d are factorized in a single\n",
        Chain_start [j], Chain_start [j+1] - 1) ;
```

```
    printf ("        working array of size %d-by-%d\n",
        Chain_maxrows [j], Chain_maxcols [j]) ;
}

/* ------------------------------------------------------------------------ */
/* numeric factorization of B */
/* ------------------------------------------------------------------------ */

status = umfpack_numeric (Bp, Bi, Bx, Symbolic, &Numeric, Control, Info) ;
if (status != UMFPACK_OK)
{
    error ("umfpack_numeric failed") ;
}
(void) umfpack_report_numeric ("Numeric factorization of B",
    Numeric, Control) ;

/* ------------------------------------------------------------------------ */
/* extract the LU factors of B and print them */
/* ------------------------------------------------------------------------ */

if (umfpack_get_lunz (&lnz, &unz, &nn, Numeric) != UMFPACK_OK)
{
    error ("umfpack_get_lunz failed") ;
}
Lp = (int *) malloc ((n+1) * sizeof (int)) ;
Li = (int *) malloc (lnz * sizeof (int)) ;
Lx = (double *) malloc (lnz * sizeof (double)) ;
Up = (int *) malloc ((n+1) * sizeof (int)) ;
Ui = (int *) malloc (unz * sizeof (int)) ;
Ux = (double *) malloc (unz * sizeof (double)) ;
P = (int *) malloc (n * sizeof (int)) ;
Q = (int *) malloc (n * sizeof (int)) ;
if (!Lp || !Li || !Lx || !Up || !Ui || !Ux || !P || !Q)
{
    error ("out of memory") ;
}
status = umfpack_get_numeric (Lp, Li, Lx, Up, Ui, Ux, P, Q, Numeric) ;
if (status != UMFPACK_OK)
{
    error ("umfpack_get_numeric failed") ;
}
(void) umfpack_report_matrix ("L (lower triangular factor of B)",
    n, Lp, Li, Lx, "row", Control) ;
(void) umfpack_report_matrix ("U (upper triangular factor of B)",
    n, Up, Ui, Ux, "column", Control) ;
(void) umfpack_report_perm ("P", n, P, Control) ;
(void) umfpack_report_perm ("Q", n, Q, Control) ;

/* ------------------------------------------------------------------------ */
/* convert L to triplet form and print it */
/* ------------------------------------------------------------------------ */

printf ("\nConverting L to triplet form, and printing it:\n") ;
```

```
Lj = (int *) malloc (lnz * sizeof (int)) ;
if (!Lj)
{
    error ("out of memory") ;
}
if (umfpack_col_to_triplet (n, Lp, Lj) != UMFPACK_OK)
{
    error ("umfpack_col_to_triplet failed") ;
}
(void) umfpack_report_triplet ("L, in triplet form", n, lnz, Li, Lj, Lx,
    Control) ;

/* ------------------------------------------------------------------------ */
/* solve B'x=b */
/* ------------------------------------------------------------------------ */

status = umfpack_solve ("A'x=b", Bp, Bi, Bx, x, b, Numeric, Control, Info) ;
umfpack_report_info (Control, Info) ;
if (status != UMFPACK_OK)
{
    umfpack_report_status (Control, status) ;
    error ("umfpack_solve failed") ;
}
(void) umfpack_report_vector ("x (solution of B'x=b)", n, x, Control) ;
printf ("maxnorm of residual: %g\n\n", resid (n, Bp, Bi, Bx, x, r, TRUE)) ;

/* ------------------------------------------------------------------------ */
/* solve B'x=b again, using umfpack_wsolve instead */
/* ------------------------------------------------------------------------ */

printf ("\nSolving B'x=b again, using umfpack_wsolve instead:\n") ;
Wi = (int *) malloc (n * sizeof (int)) ;
W = (double *) malloc (n * sizeof (double)) ;
Y = (double *) malloc (n * sizeof (double)) ;
Z = (double *) malloc (n * sizeof (double)) ;
S = (double *) malloc (n * sizeof (double)) ;
if (!Wi || !W || !Y || !Z || !S)
{
    error ("out of memory") ;
}

status = umfpack_wsolve ("A'x=b", Bp, Bi, Bx, x, b, Numeric, Control, Info,
    Wi, W, Y, Z, S) ;
umfpack_report_info (Control, Info) ;
if (status != UMFPACK_OK)
{
    umfpack_report_status (Control, status) ;
    error ("umfpack_wsolve failed") ;
}
(void) umfpack_report_vector ("x (solution of B'x=b)", n, x, Control) ;
printf ("maxnorm of residual: %g\n\n", resid (n, Bp, Bi, Bx, x, r, TRUE)) ;

/* ------------------------------------------------------------------------ */
```

```
    /* free everything */
    /* ---------------------------------------------------------------------- */

    /* This is not strictly required since the process is exiting and the */
    /* system will reclaim the memory anyway.  It's useful, though, just as */
    /* a list of what is currently malloc'ed by this program.  Plus, it's */
    /* always a good habit to explicitly free whatever you malloc. */

    free (Ap) ;
    free (Ai) ;
    free (Ax) ;

    free (Bp) ;
    free (Bi) ;
    free (Bx) ;

    free (Qtree) ;
    free (Front_npivots) ;
    free (Front_parent) ;
    free (Chain_start) ;
    free (Chain_maxrows) ;
    free (Chain_maxcols) ;

    free (Lp) ;
    free (Li) ;
    free (Lx) ;

    free (Up) ;
    free (Ui) ;
    free (Ux) ;

    free (P) ;
    free (Q) ;

    free (Lj) ;

    free (Wi) ;
    free (W) ;
    free (Y) ;
    free (Z) ;
    free (S) ;

    umfpack_free_symbolic (&Symbolic) ;
    umfpack_free_numeric (&Numeric) ;

    /* ---------------------------------------------------------------------- */
    /* print the total time spent in this demo */
    /* ---------------------------------------------------------------------- */

    t = umfpack_timer ( ) - t ;
    printf ("\numfpack demo complete.  Total time: %5.2f (seconds)\n", t) ;
    return (0) ;
}
```

18 Configuration file umf_config.h

```
/*
    This file controls the compile-time configuration of UMFPACK.  Modify the
    Makefile, the architecture-dependent Make.* file, and this file if
    necessary, to control these options.  The following compile-time flags are
    available:

        -DNBLAS

            BLAS mode.  If -DNBLAS is set, then no BLAS will be used.  Vanilla
            C code will be used instead.  This is portable, and easier to
            install, but you won't get the best performance.

            If -DNBLAS is not set, then externally-available BLAS routines
            (dgemm, dger, and dgemv or the equivalent C-BLAS routines) will be
            used.  This will give you the best performance, but perhaps at the
            expense of portability.

            The default is to use the BLAS, for both the C-callable umfpack.a
            library and the Matlab mexFunction.  If you have trouble installing
            UMFPACK, set -DNBLAS.

        -DNCBLAS

            If -DNCBLAS is set, then the C-BLAS will not be called.  This is the
            default when compiling the Matlab mexFunction, or when compiling
            umfpack.a on Sun Solaris or SGI IRIX.

            If -DNCBLAS is not set, then the C-BLAS interface to the BLAS is
            used.  If your vendor-supplied BLAS library does not have a C-BLAS
            interface, you can obtain the ATLAS BLAS, available at
            http://www.netlib.org/atlas.

            Using the C-BLAS is the default when compiling umfpack.a on all
            architectures except Sun Solaris (the Sun Performance Library is
            somewhat faster).  The ANSI C interface to the BLAS is fully
            portable.

            This flag is ignored if -DNBLAS is set.

        -DLONGBLAS

            If not defined, then the BLAS are not called in the long integer
            version of UMFPACK (the umfpack_l_* routines).  The most common
            definitions of the BLAS, unfortunately, use int arguments, and
            are thus not suitable for use in the LP64 model.  Only the Sun
            Performance Library, as far as I can tell, has a version of the
            BLAS that allows long integer (64-bit) input arguments.  This
            flag is set automatically in Sun Solaris if you are using the
            Sun Performance BLAS.  You can set it yourself, too, if your BLAS
            routines can take long integer input arguments.
```

```
    -DNSUNPERF

        Applies only to Sun Solaris.  If -DNSUNPERF is set, then the Sun
        Performance Library BLAS will not be used.

        The Sun Performance Library BLAS is used by default when compiling
        the C-callable umfpack.a library on Sun Solaris.

        This flag is ignored if -DNBLAS is set.

    -DNSCSL

        Applies only to SGI IRIX.  If -DSCSL is set, then the SGI SCSL
        Scientific Library BLAS will not be used.

        The SGI SCSL Scientific Library BLAS is used by default when
        compiling the C-callable umfpack.a library on SGI IRIX.

        This flag is ignored if -DNBLAS is set.

    -DGETRUSAGE

        If -DGETRUSAGE is set, then your system's getrusage routine will be
        used for getting the process CPU time.  Otherwise the ANSI C clock
        routine will be used.  The default is to use getrusage on Sun
        Solaris, SGI Irix, Linux, and AIX (IBM RS 6000) and to use clock on
        all other architectures.

C-to-Fortran interface, for the Fortran BLAS (these are set automatically
    for the C-BLAS or Sun Performance BLAS):

    -DBLAS_BY_VALUE         if scalars are passed by value, not reference
    -DBLAS_NO_UNDERSCORE    if no underscore should be appended
    -DBLAS_CHAR_ARG         if BLAS options are single char's, not strings

You should normally not set these flags yourself:

    -DMATLAB_MEX_FILE

        This flag is turned on when compiling the umfpack mexFunction for
        use in Matlab.  When compiling the Matlab mexFunction, the Matlab
        BLAS are used by default (this is set in the Makefile).

    -DMATHWORKS

        This flag is turned on when compiling umfpack as a built-in routine
        in Matlab.  It can also be used when compiling umfpack as a
        mexFunction.  Internal routines utMalloc, utFree, utRealloc, and
        utPrintf are used, and the "util.h" file is included.  This avoids
        the problem discussed in the User Guide regarding memory allocation
        in Matlab.  utMalloc returns NULL on failure, instead of terminating
        the mexFunction (which is what mxMalloc does).  However, the ut*
        routines are not documented by The MathWorks, Inc., so I cannot
```

```
            guarantee that you will always be able to use them.

        -DNDEBUG

            Debugging mode (if NDEBUG is not defined).  The default, of course,
            is no debugging.  Turning on debugging takes some work (see below).

*/



/* ========================================================================== */
/* === NDEBUG =============================================================== */
/* ========================================================================== */

/*
    UMFPACK will be exceedingly slow when running in debug mode.  The next three
    lines ensure that debugging is turned off.  If you want to compile UMFPACK
    in debugging mode, you must comment out the three lines below:
*/
#ifndef NDEBUG
#define NDEBUG
#endif

/*
    Next, you must either remove the -DNDEBUG option in the Makefile, or simply
    add the following line:
#undef NDEBUG
*/


/* ========================================================================== */
/* === Memory allocator ===================================================== */
/* ========================================================================== */

/* The Matlab mexFunction uses Matlab's memory manager, while the C-callable */
/* umfpack.a library uses the ANSI C malloc, free, and realloc routines. */

#ifdef MATLAB_MEX_FILE
#include "matrix.h"
#define ALLOCATE mxMalloc
#define FREE mxFree
#define REALLOCATE mxRealloc
#else
#ifdef MATHWORKS
#include "util.h"
/* Compiling UMFPACK as a built-in routine. */
/* Since UMFPACK carefully checks for out-of-memory after every allocation, */
/* we can use ut* routines here. */
#define ALLOCATE utMalloc
#define FREE utFree
#define REALLOCATE utRealloc
#else
```

```
#define ALLOCATE malloc
#define FREE free
#define REALLOCATE realloc
#endif
#endif


/* ========================================================================== */
/* === PRINTF macro ========================================================= */
/* ========================================================================== */

/* All output goes through the PRINTF macro.  Printing occurs only from the */
/* UMFPACK_report_* routines. */

#ifdef MATLAB_MEX_FILE
#include "mex.h"
#define PRINTF(params) { (void) mexPrintf params ; }
#else
#ifdef MATHWORKS
/* Already #included "util.h" above in Memory allocator section */
#define PRINTF(params) { (void) utPrintf params ; }
#else
#define PRINTF(params) { (void) printf params ; }
#endif
#endif


/* ========================================================================== */
/* === 0-based or 1-based printing ========================================== */
/* ========================================================================== */

#if defined (MATLAB_MEX_FILE) || defined (MATHWORKS)
/* In Matlab, matrices are 1-based to the user, but 0-based internally. */
/* One is added to all row and column indices when printing matrices */
/* in UMFPACK_report_*.  */
#define INDEX(i) ((i)+1)
#else
/* In ANSI C, matrices are 0-based and indices are reported as such. */
#define INDEX(i) (i)
#endif


/* ========================================================================== */
/* === Architecture ========================================================= */
/* ========================================================================== */

#if defined (__sun)
#define UMFPACK_ARCHITECTURE "Sun Solaris"
#endif

#if defined (__sgi)
#define UMFPACK_ARCHITECTURE "SGI Irix"
#endif
```

```
#if defined (__linux)
#define UMFPACK_ARCHITECTURE "Linux"
#endif

#if defined (_AIX)
#define UMFPACK_ARCHITECTURE "IBM AIX"
#endif

#if defined (__alpha)
#define UMFPACK_ARCHITECTURE "Compaq Alpha"
#endif

/* ========================================================================== */
/* === Timer ================================================================ */
/* ========================================================================== */

/*
    If you have the getrusage routine (all Unix systems I've test do), then use
    that.  Otherwise, use the ANSI C clock function.   Note that on many
    systems, the ANSI clock function wraps around after only 2147 seconds, or
    about 36 minutes.  BE CAREFUL:  if you compare the run time of UMFPACK with
    other sparse matrix packages, be sure to use the same timer.  See
    umfpack_timer.c for the timer used by UMFPACK.
*/

/* Sun Solaris, SGI Irix, Linux, Compaq Alpha, and IBM RS 6000 all have */
/* getrusage.  It's in BSD unix, so perhaps all unix systems have it. */
#if defined (__sun) || defined (__sgi) || defined (__linux) \
|| defined (__alpha) || defined (_AIX)
#define GETRUSAGE
#endif


/* ========================================================================== */
/* === BLAS ================================================================= */
/* ========================================================================== */

/*
    Determine if the BLAS exists for the long integer version.  It exists if
    LONGBLAS is defined in the Makefile, or if using the BLAS from the
    Sun Performance Library, or SGI's SCSL Scientific Library.
*/

#if !defined (MATLAB_MEX_FILE) && defined (__sun) && !defined (NSUNPERF)
#define BLAS_SUNPERF
#ifndef LONGBLAS
#define LONGBLAS
#endif
#endif

#if !defined (MATLAB_MEX_FILE) && defined (__sgi) && !defined (NSCSL)
#define BLAS_SCSL
```

```
#ifndef LONGBLAS
#define LONGBLAS
#endif
#endif

#if defined (DLONG) && !defined (LONGBLAS) && !defined (NBLAS)
#define NBLAS
#endif

/* -------------------------------------------------------------------------- */

#ifndef NBLAS

/*
    If the compile-time flag -DNBLAS is defined, then the BLAS are not used,
    portable vanilla C code is used instead, and the remainder of this file
    is ignored.

    Using the BLAS is much faster, but how C calls the Fortran BLAS is
    machine-dependent and thus can cause portability problems.  Thus, use
    -DNBLAS to ensure portability (at the expense of speed).

    Preferences:

        *** The best interface to use, regardless of the option you select
            below, is the standard C-BLAS interface.  Not all vendor-supplied
            BLAS libraries use this interface (as of April 2001).  The only
            problem with this interface is that it does not extend to the LP64
            model.

        1) most preferred:  use the optimized vendor-supplied library (such as
            the Sun Performance Library, or IBM's ESSL).  This is often the
            fastest, but might not be portable and might not always be
            available.  When compiling a Matlab mexFunction it might be
            difficult get the mex compiler script to recognize the vendor-
            supplied BLAS (I was not able get my mexFunction to use the
            Sun Performance Library BLAS, for example, because of linking
            difficulties).

        2) When compiling the UMFPACK mexFunction to use UMFPACK in Matlab, use
            the BLAS provided by The Mathworks, Inc.  This assumes you are using
            Matlab V6 or higher, since the BLAS are not incorporated in V5 or
            earlier versions.  On my Sun workstation, the Matlab BLAS gave
            slightly worse performance than the Sun Perf. BLAS.  The advantage
            of using the Matlab BLAS is that it's available on any computer that
            has Matlab V6 or higher.  I have not tried using Matlab BLAS outside
            of a mexFunction in a stand-alone C code, but Matlab (V6) allows for
            this.  This is well worth trying if you have Matlab and don't want
            to bother installing the ATLAS BLAS (option 3a, below).  The only
            glitch to this is that Matlab does not provide a portable interface
            to the BLAS (an underscore is required for some but not all
            architectures).  These variations are taken into account in the
            mexopts.sh file provided with UMFPACK.
```

```
3) Use a portable high-performance BLAS library:

    (a) The ATLAS BLAS, available at http://www.netlib.org/atlas,
        by R. Clint Whaley, Antoine Petitet, and Jack Dongarra.
        This has a standard C interface, and thus the interface to it is
        fully portable.  Its performance rivals, and sometimes exceeds,
        the vendor-supplied BLAS on many computers.

    (b) The Fortran RISC BLAS by Michel Dayde', Iain Duff, Antoine
        Petitet, and Abderrahim Qrichi Aniba, available via anonymous
        ftp to ftp.enseeiht.fr in the pub/numerique/BLAS/RISC directory,
        See M. J. Dayde' and I. S. Duff, "The RISC BLAS:  A blocked
        implementation of level 3 BLAS for RISC processors, ACM Trans.
        Math. Software, vol. 25, no. 3., Sept. 1999.  This will give
        you good performance, but with the same C-to-Fortran portability
        problems as option (1).

4) Use UMFPACK's built-in vanilla C code by setting -DNBLAS at compile
    time.  The key advantage is portability, which is guaranteed if you
    have an ANSI C compliant compiler.  You also don't need to download
    any other package - UMFPACK is stand-alone.  No Fortran is used
    anywhere in UMFPACK.  UMFPACK will be much slower than when using
    options (1) through (3), however.

5) least preferred:  use the standard Fortran implementation of the
    BLAS, also available at Netlib (http://www.netlib.org/blas).  This
    will be no faster than option (4), and not portable because of
    C-to-Fortran calling conventions.  Don't bother trying option (5).

The mechanics of how C calls the BLAS on various computers are as follows:

    * C-BLAS (from the ATLAS library, for example):

        The same interface is used on all computers.  This is the default
        (except on Sun Solaris, or when compiling the Matlab mexFunction).
        SGI Irix provides a C-BLAS interface to its vendor-supplied BLAS.

    * Defaults for calling the Fortran BLAS:
        add underscore, pass scalars by reference, use string arguments.

    * The Fortran BLAS on Sun Solaris (when compiling the Matlab mexFunction
        or when using the Fortran RISC BLAS), SGI, Linux:
        use defaults.

    * Sun Solaris (when using the C-callable Sun Performance library):
        no underscore, pass scalars by value, use character arguments.

    * The Fortran BLAS (ESSL Library) on the IBM RS 6000:
        no underscore, pass scalars by reference, use string arguments.

    * The Fortran BLAS on the HP PA:
        no underscore, pass scalars by reference, use string arguments.
```

```
            This has not been tested.  For the umfpack.a library, I recommend
            using the C-BLAS in the ATLAS library instead.  The Matlab
            mexFunction needs to have the -DBLAS_NO_UNDERSCORE compile-time
            flag set.  I've modified the mexopts.sh file to do this, but have
            not tested it.

        * The Fortran BLAS on Windows:
            no underscore, pass scalars by reference, use string arguments.
            This has not been tested.  For the umfpack.a library, I recommend
            using the C-BLAS in the ATLAS library instead.  The Mathworks-
            provided lcc compiler prepends an underscore to all C routine names.
            Thus, dgemm becomes _dgemm.  However, the Mathworks BLAS library has
            dgemm, not _dgemm.  I've contacted Mathworks and so far there is
            no work-around for this problem.  Use another compiler.  If you must
            use lcc then either do not use the BLAS in Matlab or use the C-BLAS.

*/



#ifndef NCBLAS

/* -------------------------------------------------------------------------- */
/* use the C-BLAS (any computer) */
/* -------------------------------------------------------------------------- */

/* This is the default, except for Solaris and IRIX umfpack.a, and for the */
/* mexFunction on any architecture. */

/* If you use the ATLAS C-BLAS, then be sure to set the -I flag to */
/* -I/path/ATLAS/include, where /path/ATLAS is the ATLAS installation */
/* directory.  Note that UMFPACK uses column-major storage for its dense */
/* matrices, but these are not visible to the user. */

#include "cblas.h"

#define BLAS_DGEMM_ROUTINE cblas_dgemm
#define BLAS_DGEMV_ROUTINE cblas_dgemv
#define BLAS_DGER_ROUTINE  cblas_dger

#define BLAS_NO_TRANSPOSE CblasNoTrans
#define BLAS_TRANSPOSE CblasTrans

/* This argument is present only for the C-BLAS: */
#define BLAS_COLUMN_MAJOR_ORDER CblasColMajor,

#define BLAS_SCALAR(n) n
#define BLAS_INT_SCALAR(n) n

#else

/* No such argument when not using the C-BLAS */
#define BLAS_COLUMN_MAJOR_ORDER
```

```
/* -------------------------------------------------------------------------- */
/* use Fortran (or other architecture-specific) BLAS */
/* -------------------------------------------------------------------------- */

/* Determine which architecture we're on and set options accordingly. */

#ifdef BLAS_SUNPERF
/* Sun Solaris sunperf library - the default for Solaris umfpack.a */
#include <sunperf.h>
#define BLAS_BY_VALUE
#define BLAS_NO_UNDERSCORE
#define BLAS_CHAR_ARG
#endif

#ifdef BLAS_SCSL
/* SGI SCSL library - the default for SGI umfpack.a */
#if defined (LP64)
#include <scsl_blas_i8.h>
#else
#include <scsl_blas.h>
#endif
#define BLAS_BY_VALUE
#define BLAS_NO_UNDERSCORE
#endif

/* The IBM RS 6000 does not add the underscore */
#if defined (_AIX)
#define BLAS_NO_UNDERSCORE
#endif

/*
    Add your own architecture-dependent settings here.
    For example, to call the Fortran BLAS on Windows, or HP PA:

    #if defined (__win32) || defined (__hppa)
    #define BLAS_NO_UNDERSCORE
    #endif
*/


/* -------------------------------------------------------------------------- */
/* BLAS names */
/* -------------------------------------------------------------------------- */

#if defined (LP64) && defined (BLAS_SUNPERF)

/* 64-bit sunperf BLAS, for Sun Solaris only */
#define BLAS_DGEMM_ROUTINE dgemm_64
#define BLAS_DGEMV_ROUTINE dgemv_64
#define BLAS_DGER_ROUTINE  dger_64
```

```
#else

/* naming convention (use underscore, or not) */
#ifdef BLAS_NO_UNDERSCORE
#define BLAS_DGEMM_ROUTINE dgemm
#define BLAS_DGEMV_ROUTINE dgemv
#define BLAS_DGER_ROUTINE  dger
#else
/* default:  add underscore */
#define BLAS_DGEMM_ROUTINE dgemm_
#define BLAS_DGEMV_ROUTINE dgemv_
#define BLAS_DGER_ROUTINE  dger_
#endif

#endif /* LP64 && BLAS_SUNPERF */

/* -------------------------------------------------------------------------- */
/* BLAS scalars */
/* -------------------------------------------------------------------------- */

/* pass scalars by value or by reference */
#ifdef BLAS_BY_VALUE
#define BLAS_SCALAR(n) n
#else
/* default:  pass scalars by reference */
#define BLAS_SCALAR(n) &(n)
#endif

#if defined (BLAS_SCSL) && defined (LP64)
#define BLAS_INT_SCALAR(n) ((long long) n)
#else
#define BLAS_INT_SCALAR(n) BLAS_SCALAR(n)
#endif

/* -------------------------------------------------------------------------- */
/* BLAS strings */
/* -------------------------------------------------------------------------- */

/* pass strings or single characters */
#ifdef BLAS_CHAR_ARG
#define BLAS_NO_TRANSPOSE 'N'
#define BLAS_TRANSPOSE 'T'
#else
/* default:  use string arguments */
#define BLAS_NO_TRANSPOSE "N"
#define BLAS_TRANSPOSE "T"
#endif


#endif /* NCBLAS */
```

```
/* ------------------------------------------------------------------------- */
/* BLAS macros, for all interfaces */
/* ------------------------------------------------------------------------- */


/* C = C - A*B, where A is m-by-k, B is k-by-n, and leading dimension is d */
#define BLAS_DGEMM(m,n,k,A,B,C,d) \
{ \
    double alpha = -1.0 ; \
    double beta = 1.0 ; \
    (void) BLAS_DGEMM_ROUTINE (BLAS_COLUMN_MAJOR_ORDER \
        BLAS_NO_TRANSPOSE, BLAS_NO_TRANSPOSE, \
        BLAS_INT_SCALAR (m), BLAS_INT_SCALAR (n), BLAS_INT_SCALAR (k), \
        BLAS_SCALAR (alpha), \
        A, BLAS_INT_SCALAR (d), B, BLAS_INT_SCALAR (d), BLAS_SCALAR (beta), \
        C, BLAS_INT_SCALAR (d)) ; \
}


/* A = A - x*y', where A is m-by-n with leading dimension d */
#define BLAS_DGER(m,n,x,y,A,d) \
{ \
    double alpha = -1.0 ; \
    Int incx = 1 ; \
    (void) BLAS_DGER_ROUTINE (BLAS_COLUMN_MAJOR_ORDER \
        BLAS_INT_SCALAR (m), BLAS_INT_SCALAR (n), BLAS_SCALAR (alpha), \
        x, BLAS_INT_SCALAR (incx), y, BLAS_INT_SCALAR (d), A, BLAS_INT_SCALAR (d)) ; \
}


/* y = y - A'*x, where A is m-by-n with leading dimension d, */
/* and x and y are row vectors with stride d */
#define BLAS_DGEMV_ROW(m,n,A,x,y,d) \
{ \
    double alpha = -1.0 ; \
    double beta = 1.0 ; \
    (void) BLAS_DGEMV_ROUTINE (BLAS_COLUMN_MAJOR_ORDER \
        BLAS_TRANSPOSE, \
        BLAS_INT_SCALAR (m), BLAS_INT_SCALAR (n), BLAS_SCALAR (alpha), \
        A, BLAS_INT_SCALAR (d), x, BLAS_INT_SCALAR (d), BLAS_SCALAR (beta), \
        y, BLAS_INT_SCALAR (d)) ; \
}


/* y = y - A*x, where A is m-by-n with leading dimension d, */
/* and x and y are column vectors with stride 1 */
#define BLAS_DGEMV_COL(m,n,A,x,y,d) \
{ \
    double alpha = -1.0 ; \
    double beta = 1.0 ; \
    Int incx = 1 ; \
    Int incy = 1 ; \
    (void) BLAS_DGEMV_ROUTINE (BLAS_COLUMN_MAJOR_ORDER \
```

```
        BLAS_NO_TRANSPOSE, \
        BLAS_INT_SCALAR (m), BLAS_INT_SCALAR (n), BLAS_SCALAR (alpha), \
        A, BLAS_INT_SCALAR (d), x, BLAS_INT_SCALAR (incx), BLAS_SCALAR (beta), \
        y, BLAS_INT_SCALAR (incy)) ; \
}


#endif  /* NBLAS */
```

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, 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.