

UMFPACK Version 4.1 User Guide

Timothy A. Davis

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

April 30, 2003

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 interface. UMFPACK relies on the Level-3 Basic Linear Algebra Subprograms (dense matrix multiply) for its performance. This code works on Windows and many versions of Unix (Sun Solaris, Red Hat Linux, IBM AIX, SGI IRIX, and Compaq Alpha).

Technical Report TR-03-008.

UMFPACK Version 4.1 (Apr. 30, 2003), Copyright©2003 by Timothy A. Davis. All Rights Reserved.

UMFPACK License: Your use or distribution of UMFPACK or any modified version of UMFPACK implies that you agree to this License.

THIS MATERIAL IS PROVIDED AS IS, WITH ABSOLUTELY NO WARRANTY EXPRESSED OR IMPLIED. ANY USE IS AT YOUR OWN RISK.

Permission is hereby granted to use or copy this program, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses UMFPACK or any modified version of UMFPACK code must cite the Copyright, this License, the Availability note, and “Used by permission.” Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. This software was developed with support from the National Science Foundation, and is provided to you free of charge.

Availability: <http://www.cise.ufl.edu/research/sparse/umfpack>

Acknowledgments:

This work was supported by the National Science Foundation, under grants DMS-9504974, DMS-9803599, and DMS-0203270. The upgrade to Version 4.1 and the inclusion of the symmetric and 2-by-2 pivoting strategies were done while the author was on sabbatical at Stanford University and Lawrence Berkeley National Laboratory.

Contents

1	Overview	4
2	Availability	6
3	Primary changes since Version 4.0	6
4	Using UMFPACK in MATLAB	7
5	Using UMFPACK in a C program	9
5.1	The size of an integer	10
5.2	Real and complex floating-point	10
5.3	Primary routines, and a simple example	10
5.4	A note about zero-sized arrays	13
5.5	Alternative routines	13
5.6	Matrix manipulation routines	14
5.7	Getting the contents of opaque objects	15
5.8	Reporting routines	16
5.9	Utility routines	17
5.10	Control parameters	18
5.11	Error codes	19
5.12	Larger examples	20
6	Synopsis of C-callable routines	21
6.1	Primary routines: real/int	21
6.2	Alternative routines: real/int	21
6.3	Matrix manipulation routines: real/int	22
6.4	Getting the contents of opaque objects: real/int	22
6.5	Reporting routines: real/int	22
6.6	Primary routines: complex/int	22
6.7	Alternative routines: complex/int	23
6.8	Matrix manipulation routines: complex/int	23
6.9	Getting the contents of opaque objects: complex/int	23
6.10	Reporting routines: complex/int	23
7	Synopsis of utility routines	23
8	Synopsis of AMD ordering routines	24
9	Using UMFPACK in a Fortran program	24
10	Installing the C library	26
11	Installing the MATLAB interface	28
12	Installing the Fortran interface	29
13	Known Issues	29

14 Future work	30
15 The primary UMFPACK routines	32
15.1 umfpack_*_symbolic	32
15.2 umfpack_*_numeric	42
15.3 umfpack_*_solve	51
15.4 umfpack_*_free_symbolic	56
15.5 umfpack_*_free_numeric	57
16 Alternative routines	58
16.1 umfpack_*_defaults	58
16.2 umfpack_*_qsymbolic	60
16.3 umfpack_*_wsolve	63
17 Matrix manipulation routines	66
17.1 umfpack_*_col_to_triplet	66
17.2 umfpack_*_triplet_to_col	68
17.3 umfpack_*_transpose	73
17.4 umfpack_*_scale	77
18 Getting the contents of opaque objects	79
18.1 umfpack_*_get_lunz	79
18.2 umfpack_*_get_numeric	82
18.3 umfpack_*_get_symbolic	87
18.4 umfpack_*_save_numeric	93
18.5 umfpack_*_load_numeric	95
18.6 umfpack_*_save_symbolic	97
18.7 umfpack_*_load_symbolic	99
19 Reporting routines	101
19.1 umfpack_*_report_status	101
19.2 umfpack_*_report_control	103
19.3 umfpack_*_report_info	105
19.4 umfpack_*_report_matrix	107
19.5 umfpack_*_report_numeric	111
19.6 umfpack_*_report_perm	113
19.7 umfpack_*_report_symbolic	115
19.8 umfpack_*_report_triplet	117
19.9 umfpack_*_report_vector	120
20 Utility routines	123
20.1 umfpack_timer	123
20.2 umfpack_tic and umfpack_toc	124

1 Overview

UMFPACK¹ Version 4.1 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-pattern MultiFrontal method [6, 7]. UMFPACK factorizes \mathbf{PAQ} , \mathbf{PRAQ} , or $\mathbf{PR}^{-1}\mathbf{AQ}$ into the product \mathbf{LU} , where \mathbf{L} and \mathbf{U} are lower and upper triangular, respectively, \mathbf{P} and \mathbf{Q} are permutation matrices, and \mathbf{R} is a diagonal matrix of row scaling factors (or $\mathbf{R} = \mathbf{I}$ if row-scaling is not used). Both \mathbf{P} and \mathbf{Q} are chosen to reduce fill-in (new nonzeros in \mathbf{L} and \mathbf{U} that are not present in \mathbf{A}). The permutation \mathbf{P} has the dual role of reducing fill-in and maintaining numerical accuracy (via relaxed partial pivoting and row interchanges).

The sparse matrix \mathbf{A} can be square or rectangular, singular or non-singular, and real or complex (or any combination). Only square matrices \mathbf{A} can be used to solve $\mathbf{Ax} = \mathbf{b}$ or related systems. Rectangular matrices can only be factorized.

UMFPACK first finds a column pre-ordering that reduces fill-in, without regard to numerical values. It scales and analyzes the matrix, and then automatically selects one of three strategies for pre-ordering the rows and columns: *unsymmetric*, *2-by-2*, and *symmetric*. These strategies are described below.

First, all pivots with zero Markowitz cost are eliminated and placed in the LU factors. The remaining submatrix \mathbf{S} is then analyzed. The following rules are applied, and the first one that matches defines the strategy.

- Rule 1: \mathbf{A} rectangular \rightarrow unsymmetric.
- Rule 2: If the zero-Markowitz elimination results in a rectangular \mathbf{S} , or an \mathbf{S} whose diagonal has not been preserved, the unsymmetric strategy is used.
- The symmetry σ_1 of \mathbf{S} is computed. It is defined as the number of *matched* off-diagonal entries, divided by the total number of off-diagonal entries. An entry s_{ij} is matched if s_{ji} is also an entry. They need not be numerically equal. An *entry* is a value in \mathbf{A} which is present in the input data structure. All nonzeros are entries, but some entries may be numerically zero. Rule 3: $\sigma_1 < 0.1 \rightarrow$ unsymmetric. The matrix is very unsymmetric.
- Let d be the number of nonzero entries on the diagonal of \mathbf{S} . Let \mathbf{S} be ν -by- ν . Rule 4: $(\sigma_1 \geq 0.7) \wedge (d = \nu) \rightarrow$ symmetric. The matrix has a nearly symmetric nonzero pattern, and a zero-free diagonal.

If the strategy has not yet been determined, the 2-by-2 strategy is attempted. A row permutation \mathbf{P}_2 is found which attempts to reduce the number of small diagonal entries of $\mathbf{P}_2\mathbf{S}$. An entry s_{ij} is determined to be small if $|s_{ij}| < 0.01 \max |s_{*j}|$, or large otherwise. If s_{ii} is numerically small, the method attempts to swap two rows i and j , such that both s_{ij} and s_{ji} are large. Once these rows are swapped, they remain in place. Let σ_2 be the symmetry of $\mathbf{P}_2\mathbf{S}$, and let d_2 be the number of nonzero entries (either small or large) on the diagonal of $\mathbf{P}_2\mathbf{S}$.

- Rule 5: $(\sigma_2 > 1.1\sigma_1) \wedge (d_2 > 0.9\nu) \rightarrow$ 2-by-2. The 2-by-2 permutation has made the matrix significantly more symmetric.
- Rule 6: $\sigma_2 < 0.7\sigma_1 \rightarrow$ unsymmetric. The 2-by-2 strategy has significantly deteriorated the symmetry,

¹Pronounced with two syllables: umph-pack

- Rule 7: $\sigma_2 < 0.25 \rightarrow$ unsymmetric. The matrix is still very unsymmetric.
- Rule 8: $\sigma_2 \geq 0.51 \rightarrow$ 2-by-2. The matrix is roughly symmetric.
- Rule 9: $\sigma_2 \geq 0.999\sigma_1 \rightarrow$ 2-by-2. The 2-by-2 permutation has preserved symmetry, or made it only slightly worse.
- Rule 10: if no rule has yet triggered, use the unsymmetric strategy.

Each strategy is described below:

- *unsymmetric*: The column pre-ordering of \mathbf{S} is computed by a modified version of COLAMD [8, 9, 27]. The method finds a symmetric permutation \mathbf{Q} of the matrix $\mathbf{S}^T\mathbf{S}$ (without forming $\mathbf{S}^T\mathbf{S}$ explicitly). This is a good choice for \mathbf{Q} , since the Cholesky factors of $(\mathbf{S}\mathbf{Q})^T(\mathbf{S}\mathbf{Q})$ are an upper bound (in terms of nonzero pattern) of the factor \mathbf{U} for the unsymmetric LU factorization ($\mathbf{P}\mathbf{S}\mathbf{Q} = \mathbf{L}\mathbf{U}$) regardless of the choice of \mathbf{P} [19, 20, 22]. This modified version of COLAMD also computes the column elimination tree and post-orders the tree. It finds the upper bound on the number of nonzeros in \mathbf{L} and \mathbf{U} . It also has a different threshold for determining dense rows and columns. During factorization, the column pre-ordering can be modified. Columns within a single super-column can be reshuffled, to reduce fill-in. Threshold partial pivoting is used with no preference given to the diagonal entry. Within a given pivot column j , an entry a_{ij} can be chosen if $|a_{ij}| \geq 0.1 \max |a_{*j}|$. Among those numerically acceptable entries, the sparsest row i is chosen as the pivot row.
- *2-by-2*: The symmetric strategy (see below) is applied to the matrix $\mathbf{P}_2\mathbf{S}$, rather than \mathbf{S} .
- *symmetric*: The column ordering is computed from AMD [1, 2, ?], applied to the pattern of $\mathbf{S} + \mathbf{S}^T$ followed by a post-ordering of the supernodal elimination tree of $\mathbf{S} + \mathbf{S}^T$. No modification of the column pre-ordering is made during numerical factorization. Threshold partial pivoting is used, with a strong preference given to the diagonal entry. The diagonal entry is chosen if $a_{jj} \geq 0.001 \max |a_{*j}|$. Otherwise, a sparse row is selected, using the same method used by the unsymmetric strategy.

The symmetric and 2-by-2 strategies, and their automatic selection, are new to Version 4.1. Version 4.0 only used the unsymmetric strategy.

Once the strategy is selected, the factorization of the matrix \mathbf{A} is broken down into the factorization of 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 [18] 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. For the nonsingular matrices factorized with the unsymmetric strategy, there are exactly the same number of chains as there are leaves 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 sub-matrices (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 which 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 numeric 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 [21]), a pivot row can be selected based on sparsity-preserving criteria (low degree) as well as numerical considerations (relaxed threshold partial pivoting).

When the symmetric or 2-by-2 strategies are used, the column preordering is not refined during numeric factorization. Row pivoting for sparsity and numerical accuracy is performed if the diagonal entry is too small.

More details of the method, including experimental results, are described in [4, 5], available at <http://www.cise.ufl.edu/tech-reports>.

2 Availability

In addition to appearing as a Collected Algorithm of the ACM, UMFPACK Version 4.1 is available at <http://www.cise.ufl.edu/research/sparse>. An earlier version (Version 4.0) is included as a built-in routine in MATLAB 6.5. Version 4.0 does not have the symmetric or 2-by-2 strategies and it takes less advantage of the level-3 BLAS [11, 12, 28, 25]. Version 4.1 tends to be much faster than Version 4.0, particularly on unsymmetric matrices with mostly symmetric nonzero pattern (such as finite element and circuit simulation matrices). Version 3.0 and following make use of a modified version of COLAMD V2.0 by Timothy A. Davis, Stefan Larimore, John Gilbert, and Esmond Ng. The original COLAMD V2.1 is available in as a built-in routine in MATLAB V6.0 (or later), and at <http://www.cise.ufl.edu/research/sparse>. These codes are also available in Netlib [13] at <http://www.netlib.org>. UMFPACK Versions 2.2.1 and earlier, co-authored with Iain Duff, are available at <http://www.cise.ufl.edu/research/sparse> and as MA38 (functionally equivalent to Version 2.2.1) in the Harwell Subroutine Library.

3 Primary changes since Version 4.0

A detailed list of changes is in the **ChangeLog** file. The following is a summary of the main changes that are visible to the C or MATLAB user:

1. New ordering strategies added. No changes are required in user code (either C or MATLAB) to use the new default strategy, which is an automatic selection of the unsymmetric, symmetric, or 2-by-2 strategies.
2. Row scaling added. This is only visible to the MATLAB caller when using the form `[L,U,P,Q,R] = umfpack(A)`, to retrieve the LU factors. Likewise, it is only visible to the C caller when the LU factors are retrieved, or when solving systems with just **L** or **U**. New C-callable and MATLAB-callable routines are included to get and to apply the scale factors computed by UMFPACK. Row scaling is enabled by default, but can be disabled. Row scaling usually leads to a better factorization, particularly when the symmetric strategy is used.
3. Error code `UMFPACK_ERROR_problem_to_large` removed. Version 4.0 would generate this error when the upper bound memory usage exceeded 2GB (for the `int` version), even when the

actual memory usage was less than this. The new version properly handles this case, and can successfully factorize the matrix if sufficient memory is available.

4. New control parameters and statistics provided.
5. The AMD symmetric approximate minimum degree ordering routine added [1, 2, ?]. It is used by UMFPACK, and can also be called independently from C or MATLAB.
6. The `umfpack` mexFunction now returns permutation matrices, not permutation vectors, when using the form `[L,U,P,Q] = umfpack (A)` or the new form `[L,U,P,Q,R] = umfpack (A)`.
7. New arguments added to the user-callable routines `umfpack*_symbolic`, `umfpack*_qsymbolic`, `umfpack*_get_numeric`, and `umfpack*_get_symbolic`. The symbolic analysis now makes use of the numerical values of the matrix **A**, to guide the 2-by-2 strategy. The subsequent matrix passed to the numeric factorization step does not have to have the same numerical values. All of the new arguments are optional. If you do not wish to include them, simply pass `NULL` pointers instead. The 2-by-2 strategy will assume all entries are numerically large, for example.
8. New routines added to save and load the `Numeric` and `Symbolic` objects to and from a binary file.
9. A Fortran interface added. It provides access to a subset of UMFPACK's features.

4 Using UMFPACK in MATLAB

The easiest way to use UMFPACK is within MATLAB. Version 4.0 is a built-in routine in MATLAB 6.5, and is used in `x = A\b` when **A** is sparse, square, unsymmetric (or symmetric but not positive definite), and with nonzero entries that are not confined in a narrow band. It is also used for the `[L,U,P,Q] = lu (A)` usage of `lu`. Type `help lu` in MATLAB 6.5 or later for more details.

To use the UMFPACK v4.0 or v4.1 mexFunction, you must download and compile it, since the mexFunction itself is not part of MATLAB. The following discussion assumes that you have MATLAB Version 6.0 or later (which includes the BLAS, and the `colamd` ordering routine). To compile both the UMFPACK and AMD mexFunctions, just type `make` in the Unix system shell, while in the UMFPACK directory. You can also type `umfpack.make` in MATLAB, if you are in the UMFPACK/MATLAB directory, or if that directory is in your MATLAB path. This works on any system with MATLAB, including Windows. See Section 10 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 the UMFPACK mexFunction within MATLAB.

An optional input argument can be used to modify the control parameters for UMFPACK, and an optional output argument provides statistics on the factorization. UMFPACK will print these statistics if you turn on the sparse matrix monitor flag with `spparms ('spumoni', 1)`. The UMFPACK printing level parameter described in Section 5.8 is set to `spumoni` plus 1. Type `help umfpack` for more information.

Refer to the AMD User Guide for more details about the AMD mexFunction.

Note: in MATLAB 6.5, use `spparms ('autoamd',0)` in addition to `spparms ('autommd',0)`, in Table 1, to turn off MATLAB's default reordering.

UMFPACK requires **b** to be a dense vector (real or complex) of the appropriate dimension. This is more restrictive than what you can do with MATLAB's backslash or forward slash. See

Table 1: Using UMFPACK's MATLAB interface

Function	Using UMFPACK	MATLAB 6.0 equivalent
Solve $\mathbf{Ax} = \mathbf{b}$.	<code>x = umfpack (A, '\', b) ;</code>	<code>x = A \ b ;</code>
Solve $\mathbf{Ax} = \mathbf{b}$ using a different column pre-ordering.	<code>S = spones (A) ; Q = symamd (S+S') ; x = umfpack (A,Q, '\', b) ;</code>	<code>spparms ('autommd',0) ; S = spones (A) ; Q = symamd (S+S') ; x = A (:,Q) \ b ; x (Q) = x ; spparms ('autommd',1) ;</code>
Solve $\mathbf{A}^T \mathbf{x}^T = \mathbf{b}^T$.	<code>x = umfpack (b, '/', A) ;</code>	<code>x = b / A ;</code>
Scale and factorize \mathbf{A} , then solve $\mathbf{Ax} = \mathbf{b}$.	<code>[L,U,P,Q,R] = umfpack (A) ; c = P * (R \ b) ; x = Q * (U \ (L \ c)) ;</code>	<code>[m n] = size (A) ; r = full (sum (abs (A), 2)) ; r (find (r == 0)) = 1 ; R = spdiags (r, 0, m, m) ; I = speye (n) ; Q = I (:, colamd (A)) ; [L,U,P] = lu ((R\A)*Q) ; c = P * (R \ b) ; x = Q * (U \ (L \ c)) ;</code>

`umfpack_solve` for an M-file that removes this restriction. This restriction does not apply to the built-in backslash operator in MATLAB 6.5, which uses UMFPACK v4.0 to factorize the matrix. You can do this yourself in MATLAB:

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

or, with no row scaling:

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

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

MATLAB's `[L,U,P] = lu(A)` returns a lower triangular L, an upper triangular U, and a permutation matrix P such that $P \cdot A$ is equal to $L \cdot U$. UMFPACK behaves differently. By default, it scales the rows of A and reorders the columns of A prior to factorization, so that $L \cdot U$ is equal to $P \cdot (R \setminus A) \cdot Q$, where R is a diagonal sparse matrix of scale factors for the rows of A. The scale factors R are applied to A via the MATLAB expression `R \ A` to avoid multiplying by the reciprocal, which can be numerically inaccurate.

There are more options; you can provide your own column pre-ordering (in which case UMFPACK does not call COLAMD or AMD), you can modify other control settings (similar to the `spparms` in MATLAB), and you can get various statistics on the analysis, factorization, and solution of the linear system. Type `umfpack_details` and `umfpack_report` in MATLAB for more information. Two demo M-files are provided. Just type `umfpack_simple` and `umfpack_demo` to run them.

The output of these two programs should be about the same as the files `umfpack_simple.m.out` and `umfpack_demo.m.out` that are provided.

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

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

This is an alternative to `[L,U,P,Q]=umfpack(A)`.

A simple M-file (`umfpack_btf`) is provided that first permutes the matrix to upper block triangular form, using MATLAB's `dmperm` routine, and then solves each block. The LU factors are not returned. Its usage is simple: `x = umfpack_btf(A,b)`. Type `help umfpack_btf` for more options. An estimate of the 1-norm of $L*U-P*A*Q$ can be computed in MATLAB as `lu_normest(P*A*Q,L,U)`, using the `lu_normest.m` M-file by Hager and Davis [10] that is included with the UMFPACK distribution. With row scaling enabled, use `lu_normest(P*(R\A)*Q,L,U)` instead.

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. This is 70% of the upper bound, by default, for the unsymmetric strategy. For the symmetric strategy, the fraction of the upper bound is computed automatically (assuming a best-case scenario with no numerical pivoting required during numeric factorization). If this initial allocation fails, it reduces its request and uses less memory. If the space is not large enough during factorization, it is increased via `mxRealloc`.

However, `mxMalloc` and `mxRealloc` abort the `umfpack` mexFunction if they fail, so this strategy does not work in MATLAB. The strategy works fine when `malloc` or the internal memory allocator `utMalloc` are used instead, since those routines return `NULL` on failure, and do not terminate the mexFunction. The `umfpack` mexFunction can be compiled to use `utMalloc`, but this is an internal undocumented utility routine in MATLAB, and thus using `utMalloc` might not always be successful. To use the documented `mxMalloc` routine instead, compile the mexFunction with the `-DNUTIL` flag enabled.

5 Using UMFPACK in a C program

The C-callable UMFPACK library consists of 31 user-callable routines and one include file. Twenty-seven of the routines come in four versions, with different sizes of integers and for real or complex floating-point numbers:

1. `umfpack_di_*`: real double precision, `int` integers.

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

where `*` denotes the specific name of one of the 27 routines. Routine names beginning with `umf_` are internal to the package, and should not be called by the user. The include file `umfpack.h` must be included in any C program that uses UMFPACK. The other three routines are the same for all four versions.

In addition, the C-callable AMD library distributed with UMFPACK includes 4 user-callable routines (in two versions with `int` and `long` integers) and one include file. Refer to the AMD documentation for more details.

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

The notation `umfpack_di_*` refers to all 27 user-callable routines for the real double precision and `int` integer case. The notation `umfpack_*_numeric`, for example, refers all four versions (real/complex, int/long) of a single operation (in this case numeric factorization).

5.1 The size of an integer

The `umfpack_di_*` and `umfpack_zi_*` routines use `int` integer arguments; those starting with `umfpack_dl_` or `umfpack_zl_` use `long` integer arguments. If you compile UMFPACK in the standard ILP32 mode (32-bit `int`'s, `long`'s, and pointers) then the versions are essentially identical. You will be able to solve problems using up to 2GB of memory. If you compile UMFPACK in the standard LP64 mode, the size of an `int` remains 32-bits, but the size of a `long` and a pointer both get promoted to 64-bits. In the LP64 mode, the `umfpack_dl_*` and `umfpack_zl_*` routines can solve huge problems (not limited to 2GB), limited of course by the amount of available memory. The only drawback to the 64-bit mode is that not all BLAS libraries support 64-bit integers. This limits the performance you will obtain. Those that do support 64-bit integers are specific to particular architectures, and are not portable. UMFPACK and AMD should be compiled in the same mode. If you compile UMFPACK and AMD in the LP64 mode, be sure to add `-DLP64` to the compilation command. See the examples in `Make.alpha`, `Make.sgi`, and `Make.solaris`.

5.2 Real and complex floating-point

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

5.3 Primary routines, and a simple example

Five primary UMFPACK routines are required to factorize \mathbf{A} or solve $\mathbf{Ax} = \mathbf{b}$. They are fully described in Section 15:

- `umfpack_*_symbolic`:

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

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. It is no larger than about $13n$ integers if \mathbf{A} is n -by- n .

- `umfpack_*_numeric`:

Numerically scales and then factorizes a sparse matrix into \mathbf{PAQ} , \mathbf{PRAQ} , or $\mathbf{PR}^{-1}\mathbf{AQ}$ into the product \mathbf{LU} , where \mathbf{P} and \mathbf{Q} are permutation matrices, \mathbf{R} is a diagonal matrix of scale factors, \mathbf{L} is lower triangular with unit diagonal, and \mathbf{U} is upper triangular. Requires the symbolic ordering and analysis computed by `umfpack_*_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 or Fortran program, but not in MATLAB). The matrix \mathbf{U} will have zeros on the diagonal if \mathbf{A} is singular; this produces a warning, but the factorization is still valid.

- `umfpack_*_solve`:

Solves a sparse linear system ($\mathbf{Ax} = \mathbf{b}$, $\mathbf{A}^T\mathbf{x} = \mathbf{b}$, or systems involving just \mathbf{L} or \mathbf{U}), using the numeric factorization computed by `umfpack_*_numeric`. Iterative refinement with sparse backward error [3] is used by default. The matrix \mathbf{A} must be square. If it is singular, then a divide-by-zero will occur, and your solution will contain IEEE Inf's or NaN's in the appropriate places.

- `umfpack_*_free_symbolic`:

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

- `umfpack_*_free_numeric`:

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

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

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

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

For the `long` version of UMFPACK:

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

The complex versions add another array for the imaginary part:

```
double Az [nz] ;
```

All nonzeros are entries, but an entry may be numerically zero. The row indices of entries in column j are stored in $Ai[Ap[j] \dots Ap[j+1]-1]$. The corresponding numerical values are stored in $Ax[Ap[j] \dots Ap[j+1]-1]$. The imaginary part, for the complex versions, is stored in $Az[Ap[j] \dots Ap[j+1]-1]$.

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} . For a more flexible method for providing an input matrix to UMFPACK, see Section 5.6.

Here is a simple main program, `umfpack_simple.c`, that illustrates the basic usage of UMFPACK. See Section 6 for a short description of each calling sequence, including a list of options for the first argument of `umfpack_di_solve`.

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

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

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

The Ap , Ai , and Ax arrays represent the matrix

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

and the solution to $\mathbf{Ax} = \mathbf{b}$ is $\mathbf{x} = [12345]^T$. 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`). It also ignores the status value returned by most user-callable UMFPACK routines.

5.4 A note about zero-sized arrays

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

5.5 Alternative routines

Three alternative routines are provided that modify UMFPACK's default behavior. They are fully described in Section 16:

- `umfpack_*_defaults`:

Sets the default control parameters in the `Control` array. These can then be modified as desired before passing the array to the other UMFPACK routines. Control parameters are summarized in Section 5.10. Three particular parameters deserve 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 `Control [UMFPACK_PIVOT_TOLERANCE]` has a default value of 0.1, and is used for the unsymmetric strategy. For complex matrices, a cheap approximation of the absolute value is used for the threshold pivoting test ($|a| \approx |a_{\text{real}}| + |a_{\text{imag}}|$).

For the symmetric strategy, a second tolerance is used for diagonal entries:

`Control [UMFPACK_SYM_PIVOT_TOLERANCE]`, with a default value of 0.001. The first parameter (with a default of 0.1) is used for any off-diagonal candidate pivot entries.

These two parameters may be too small for some matrices, particularly for ill-conditioned or poorly scaled ones. With the default pivot tolerances and default iterative refinement, `x = umfpack (A, '\', b)` is just as accurate as (or more accurate) than `x = A\b` in MATLAB 6.1 for nearly all matrices.

If `Control [UMFPACK_PIVOT_TOLERANCE]` is zero, then any nonzero entry is acceptable as a pivot (this is changed from Version 4.0, which treated a value of 0.0 the same as 1.0). If the symmetric strategy is used, and `Control [UMFPACK_SYM_PIVOT_TOLERANCE]` is zero, then any nonzero entry on the diagonal is accepted as a pivot. Off-diagonal pivoting will still occur if the diagonal entry is exactly zero. The `Control [UMFPACK_SYM_PIVOT_TOLERANCE]` parameter is new to Version 4.1. It is similar in function to the pivot tolerance for left-looking methods (the MATLAB `THRESH` option in `[L,U,P] = lu (A, THRESH)`, and the pivot tolerance parameter in SuperLU).

The parameter `Control [UMFPACK_STRATEGY]` can be used to bypass UMFPACK's automatic strategy selection. The automatic strategy nearly always selects the best method. When it does not, the different methods nearly always give about the same quality of results. There may be cases where the automatic strategy fails to pick a good strategy. Also, you can save some computing time if you know the right strategy for your set of matrix problems.

- `umfpack_*_qsymbolic`:

An alternative to `umfpack*_symbolic`. Allows the user to specify his or her own column pre-ordering, rather than using the default COLAMD or AMD pre-orderings. For example, a graph partitioning-based order of $\mathbf{A}^T \mathbf{A}$ would be suitable for UMFPACK's unsymmetric strategy. A partitioning of $\mathbf{A} + \mathbf{A}^T$ would be suitable for UMFPACK's symmetric or 2-by-2 strategies.

- `umfpack*_wsolve`:

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

5.6 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 17 presents the details of routines for manipulating sparse matrices in *triplet* form, compressed column form, and compressed row form (the transpose of the compressed column form). The triplet form of a matrix consists of three or four arrays. For the `int` version of UMFPACK:

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

For the `long` version:

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

The complex versions use another array to hold the imaginary part:

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

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

Four routines are provided for manipulating sparse matrices:

- `umfpack*_triplet_to_col`:

Converts a triplet form of a matrix to compressed column form (ready for input to `umfpack*_symbolic`, `umfpack*_qsymbolic`, and `umfpack*_numeric`). Identical to `A = spconvert(i,j,x)` in MATLAB, except that zero entries are not removed, so that the pattern of entries in the compressed column form of \mathbf{A} are fully under user control. This is important if you want to factorize a new matrix with the `Symbolic` object from a prior matrix with the same pattern as the new one.

- `umfpack*_col_to_triplet`:

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

- `umfpack*_transpose`:

Transposes and optionally permutes a column form matrix [26]. Identical to $\mathbf{R} = \mathbf{A}(\mathbf{P}, \mathbf{Q})'$ (linear algebraic transpose, using the complex conjugate) or $\mathbf{R} = \mathbf{A}(\mathbf{P}, \mathbf{Q})'$ (the array transpose) in MATLAB, except for the presence of numerically zero entries.

Factorizing \mathbf{A}^T and then solving $\mathbf{Ax} = \mathbf{b}$ with the transposed factors can sometimes be much faster or much slower than factorizing \mathbf{A} . It is highly dependent on your particular matrix.

- `umfpack*_scale`:

Applies the row scale factors to a user-provided vector. This is not required to solve the sparse linear system $\mathbf{Ax} = \mathbf{b}$ or $\mathbf{A}^T \mathbf{x} = \mathbf{b}$, since `umfpack*_solve` applies the scale factors for those systems. This routine is new to Version 4.1.

It is quite easy to add matrices in triplet form, subtract them, transpose them, permute them, construct a submatrix, and multiply a triplet-form matrix times a vector. UMFPACK does not provide code for these basic operations, however. Refer to the discussion of `umfpack*_triplet_to_col` in Section 17 for more details on how to compute these operations in your own code. The only primary matrix operation not provided by UMFPACK is the multiplication of two sparse matrices [26].

5.7 Getting the contents of opaque objects

There are cases where you may wish to do more with the LU factorization of a matrix than solve a linear system. The opaque `Symbolic` and `Numeric` objects are just that - opaque. You cannot do anything with them except to pass them back to subsequent calls to UMFPACK. Three routines are provided for copying their contents into user-provided arrays using simpler data structures. Four routines are provided for saving and loading the `Numeric` and `Symbolic` objects to/from binary files. They are fully described in Section 18:

- `umfpack*_get_lunz`:

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

- `umfpack*_get_numeric`:

Copies \mathbf{L} , \mathbf{U} , \mathbf{P} , \mathbf{Q} , and \mathbf{R} from the `Numeric` object into arrays provided by the user. The matrix \mathbf{L} is returned in compressed row form (with the column indices in each row sorted in ascending order). The matrix \mathbf{U} is returned in compressed column form (with sorted columns). There are no explicit zero entries in \mathbf{L} and \mathbf{U} , but such entries may exist in the `Numeric` object. The permutations \mathbf{P} and \mathbf{Q} are represented as permutation vectors, where $\mathbf{P}[\mathbf{k}] = \mathbf{i}$ means that row \mathbf{i} of the original matrix is the \mathbf{k} -th row of \mathbf{PAQ} , and where $\mathbf{Q}[\mathbf{k}] = \mathbf{j}$ means that column \mathbf{j} of the original matrix is the \mathbf{k} -th column of \mathbf{PAQ} . This is identical to how MATLAB uses permutation vectors (type `help colamd` in MATLAB 6.1 or later).

- `umfpack*_get_symbolic`:

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

- `umfpack_*_save_numeric:`
Saves a copy of the `Numeric` object to a file, in binary format.
- `umfpack_*_load_numeric:`
Creates a `Numeric` object by loading it from a file created by `umfpack_*_save_numeric`.
- `umfpack_*_save_symbolic:`
Saves a copy of the `Symbolic` object to a file, in binary format.
- `umfpack_*_load_symbolic:`
Creates a `Symbolic` object by loading it from a file created by `umfpack_*_save_symbolic`.

UMFPACK itself does not make use of these routines; they are provided solely for returning the contents of the opaque `Symbolic` and `Numeric` objects to the user, and saving/loading them to/from a binary file.

5.8 Reporting routines

None of the UMFPACK routines discussed so far prints anything, even when an error occurs. UMFPACK provides you with nine routines for printing the input and output arguments (including the `Control` settings and `Info` statistics) of UMFPACK routines discussed above. They are fully described in Section 19:

- `umfpack_*_report_status:`
Prints the status (return value) of other `umfpack_*` routines.
- `umfpack_*_report_info:`
Prints the statistics returned in the `Info` array by `umfpack_*_symbolic`, `umfpack_*_numeric`, and `umfpack_*_solve`.
- `umfpack_*_report_control:`
Prints the `Control` settings.
- `umfpack_*_report_matrix:`
Verifies and prints a compressed column-form or compressed row-form sparse matrix.
- `umfpack_*_report_triplet:`
Verifies and prints a matrix in triplet form.
- `umfpack_*_report_symbolic:`
Verifies and prints a `Symbolic` object.
- `umfpack_*_report_numeric:`
Verifies and prints a `Numeric` object.
- `umfpack_*_report_perm:`
Verifies and prints a permutation vector.

- `umfpack_*_report_vector`:

Verifies and prints a real or complex vector.

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

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

- `umfpack_*_report_status`:

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

- `umfpack_*_report_control`:

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

- `umfpack_*_report_info`:

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

- all other `umfpack_*_report_*` routines:

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

This print level parameter has an additional effect on the MATLAB mexFunction. If zero, then no warnings of singular or nearly singular matrices are printed (similar to the MATLAB commands `warning off MATLAB:singularMatrix` and `warning off MATLAB:nearlySingularMatrix`).

5.9 Utility routines

UMFPACK v4.0 included 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 20. It is still available in UMFPACK v4.1, but not used internally. Two new timing routines are provided in UMFPACK Version 4.1, `umfpack_tic` and `umfpack_toc`. They use POSIX-compliant `sysconf` and `times` routines to find both the CPU time and wallclock time. These three routines are the only user-callable routine that is identical in all four `int/long`, real/complex versions (there is no `umfpack_di_timer` routine, for example).

Table 2: UMFPACK Control parameters

MATLAB	ANSI C	default	description
Control(1)	Control[UMFPACK_PRL]	1	printing level
Control(2)	Control[UMFPACK_DENSE_ROW]	0.2	dense row parameter
Control(3)	Control[UMFPACK_DENSE_COL]	0.2	dense column parameter
Control(4)	Control[UMFPACK_PIVOT_TOLERANCE]	0.1	partial pivoting tolerance
Control(5)	Control[UMFPACK_BLOCK_SIZE]	32	BLAS block size
Control(6)	Control[UMFPACK_STRATEGY]	0 (auto)	select strategy
Control(7)	Control[UMFPACK_ALLOC_INIT]	0.7	initial memory allocation
Control(8)	Control[UMFPACK_IRSTEP]	2	max iter. refinement steps
Control(13)	Control[UMFPACK_2BY2_TOLERANCE]	0.01	defines “large” entries
Control(14)	Control[UMFPACK_FIXQ]	0 (auto)	fix or modify Q
Control(15)	Control[UMFPACK_AMD_DENSE]	10	AMD dense row/column parameter
Control(16)	Control[UMFPACK_SYM_PIVOT_TOLERANCE]	0.001	for diagonal entries
Control(17)	Control[UMFPACK_SCALE]	1 (sum)	row scaling (none, sum, or max)
Control(18)	Control[UMFPACK_FRONT_ALLOC_INIT]	0.5	frontal matrix allocation ratio
Control(19)	Control[18]	-	(unused)
Control(20)	Control[UMFPACK_AGGRESSIVE]	1 (yes)	aggressive absorption in AMD and COLAMD
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]	-	1 if <code>getrusage</code> used
Control(12)	Control[UMFPACK_COMPILED_IN_DEBUG_MODE]	-	true if debug mode enabled

5.10 Control parameters

UMFPACK uses an optional `double` array (currently of size 20) to modify its control parameters. If you pass (`double *`) `NULL` instead of a `Control` array, then defaults are used. This array will almost certainly grow in size in future releases, so be sure to dimension your `Control` array to be of size `UMFPACK_CONTROL`. That constant is currently defined to be 20, but will increase in future versions.

The contents of this array may be modified by the user (see `umfpack*_defaults`). Each user-callable routine includes a complete description of how each control setting modifies its behavior. Table 2 summarizes the entire contents of the `Control` array. Note that ANSI C uses 0-based indexing, while MATLAB uses 1-based indexing. Thus, `Control(1)` in MATLAB is the same as `Control[0]` or `Control[UMFPACK_PRL]` in ANSI C.

Let $\alpha_r = \text{Control}[\text{UMFPACK_DENSE_ROW}]$, $\alpha_c = \text{Control}[\text{UMFPACK_DENSE_COL}]$, and $\alpha = \text{Control}[\text{UMFPACK_AMD_DENSE}]$. Suppose the submatrix \mathbf{S} , obtained after eliminating pivots with zero Markowitz cost, is m -by- n . Then a row is considered “dense” if it has more than $\max(16, 16\alpha_r\sqrt{n})$ entries. A column is considered “dense” if it has more than $\max(16, 16\alpha_c\sqrt{m})$ entries. These rows and columns are treated different in COLAMD and during numerical factorization. In COLAMD, dense columns are placed last in their natural order, and dense rows are ignored. During numerical factorization, dense rows are stored differently. In AMD, a row/column of the square matrix $\mathbf{S} + \mathbf{S}^T$ is considered “dense” if it has more than $\max(16, \alpha\sqrt{n})$ entries. These rows/columns are placed last in AMD’s output ordering. For more details on the control parameters, refer to the documentation of `umfpack*_qsymbolic`, `umfpack*_numeric`, `umfpack*_solve`, and the `umfpack*_report_*` routines, in Sections 15 through 19, below.

5.11 Error codes

Many of the routines return a `status` value. This is also returned as the first entry in the `Info` array, for those routines with that argument. The following list summarizes all of the error codes in UMFPACK v4.1. Each error code is given a specific name in the `umfpack.h` include file, so you can use those constants instead of hard-coded values in your program. Future versions may report additional error codes.

A value of zero means everything was successful, and the matrix is non-singular. A value greater than zero means the routine was successful, but a warning occurred. Currently, the only warning code is when the matrix is singular. A negative value means the routine was not successful. In this case, no `Symbolic` or `Numeric` object was created.

- `UMFPACK_OK`, (0): UMFPACK was successful.
- `UMFPACK_WARNING_singular_matrix`, (1): Matrix is singular. There are exact zeros on the diagonal of `U`.
- `UMFPACK_ERROR_out_of_memory`, (-1): Not enough memory. The ANSI C `malloc` or `realloc` routine failed.
- `UMFPACK_ERROR_invalid_Numeric_object`, (-3): Routines that take a `Numeric` object as input (or load it from a file) check this object and return this error code if it is invalid. This can be caused by a memory leak or overrun in your program, which can overwrite part of the `Numeric` object. It can also be caused by passing a `Symbolic` object by mistake, or some other pointer. If you try to factorize a matrix using one version of UMFPACK and then use the factors in another version, this error code will trigger as well. You cannot factor your matrix using version 4.0 and then solve with version 4.1. You cannot use different precisions of the same version (real and complex, for example). It is possible for the `Numeric` object to be corrupted by your program in subtle ways that are not detectable by this quick check. In this case, you may see an `UMFPACK_ERROR_different_pattern` error code, or even an `UMFPACK_ERROR_internal_error`.
- `UMFPACK_ERROR_invalid_Symbolic_object`, (-4): Routines that take a `Symbolic` object as input (or load it from a file) check this object and return this error code if it is invalid. The causes of this error are analogous to the `UMFPACK_ERROR_invalid_Numeric_object` error described above.
- `UMFPACK_ERROR_argument_missing`, (-5): Some arguments of some are optional (you can pass a `NULL` pointer instead of an array). This error code occurs if you pass a `NULL` pointer when that argument is required to be present.
- `UMFPACK_ERROR_n_nonpositive` (-6): The number of rows or columns of the matrix must be greater than zero.
- `UMFPACK_ERROR_invalid_matrix` (-8): The matrix is invalid. For the column-oriented input, this error code will occur if the contents of `Ap` and/or `Ai` are invalid.

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

The nonzero pattern (row indices) for column j is stored in $A_i \ [(A_p \ [j]) \dots (A_p \ [j+1]-1)]$. The row indices in a given column j must be in ascending order, and no duplicate row indices may be present. Row indices must be in the range 0 to n_row-1 (the matrix is 0-based).

Some routines take a triplet-form input, with arguments nz , T_i , and T_j . This error code is returned if nz is less than zero, if any row index in T_i is outside the range 0 to n_col-1 , or if any column index in T_j is outside the range 0 to n_row-1 .

- `UMFPACK_ERROR_different_pattern`, (-11): The most common cause of this error is that the pattern of the matrix has changed between the symbolic and numeric factorization. It can also occur if the `Numeric` or `Symbolic` object has been subtly corrupted by your program.
- `UMFPACK_ERROR_invalid_system`, (-13): The `sys` argument provided to one of the solve routines is invalid.
- `UMFPACK_ERROR_invalid_permutation`, (-15): The permutation vector provided as input is invalid.
- `UMFPACK_ERROR_file_IO`, (-17): This error code is returned by the routines that save and load the `Numeric` or `Symbolic` objects to/from a file, if a file I/O error has occurred. The file may not exist or may not be readable, you may be trying to create a file that you don't have permission to create, or you may be out of disk space. The file you are trying to read might be the wrong one, and an earlier end-of-file condition would then result in this error.
- `UMFPACK_ERROR_internal_error`, (-911): An internal error has occurred, of unknown cause. This is either a bug in UMFPACK, or the result of a memory overrun from your program. Try modifying the file `AMD/Source/amd_internal.h` and adding the statement `#undef NDEBBUG`, to enable the debugging mode. Recompile UMFPACK and rerun your program. A failed assertion might occur which can give you a better indication as to what is going wrong. Be aware that UMFPACK will be extraordinarily slow when running in debug mode. If all else fails, contact the developer (davis@cise.ufl.edu) with as many details as possible.

5.12 Larger examples

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

The `umf4.c` program for reading matrices in Harwell/Boeing format [15] is provided. It requires three Fortran 77 programs (`readhb.f`, `readhb_nozeros.f`, and `readhb_size.f`) for reading in the sample Harwell/Boeing files in the `UMFPACK/Demo/HB` directory. More matrices are available at <http://www.cise.ufl.edu/research/sparse/matrices>. Type `make hb` in the `UMFPACK/Demo/HB` directory to compile and run this demo. This program was used for the experimental results in [5].

Table 3: UMFPACK `sys` parameter

Value		system
UMFPACK_A	(0)	$\mathbf{Ax} = \mathbf{b}$
UMFPACK_At	(1)	$\mathbf{A}^H \mathbf{x} = \mathbf{b}$
UMFPACK_Aat	(2)	$\mathbf{A}^T \mathbf{x} = \mathbf{b}$
UMFPACK_Pt_L	(3)	$\mathbf{P}^T \mathbf{Lx} = \mathbf{b}$
UMFPACK_L	(4)	$\mathbf{Lx} = \mathbf{b}$
UMFPACK_Lt_P	(5)	$\mathbf{L}^H \mathbf{Px} = \mathbf{b}$
UMFPACK_Lat_P	(6)	$\mathbf{L}^T \mathbf{Px} = \mathbf{b}$
UMFPACK_Lt	(7)	$\mathbf{L}^H \mathbf{x} = \mathbf{b}$
UMFPACK_Lat	(8)	$\mathbf{L}^T \mathbf{x} = \mathbf{b}$
UMFPACK_U_Qt	(9)	$\mathbf{UQ}^T \mathbf{x} = \mathbf{b}$
UMFPACK_U	(10)	$\mathbf{Ux} = \mathbf{b}$
UMFPACK_Q_Ut	(11)	$\mathbf{QU}^H \mathbf{x} = \mathbf{b}$
UMFPACK_Q_Uat	(12)	$\mathbf{QU}^T \mathbf{x} = \mathbf{b}$
UMFPACK_Ut	(13)	$\mathbf{U}^H \mathbf{x} = \mathbf{b}$
UMFPACK_Uat	(14)	$\mathbf{U}^T \mathbf{x} = \mathbf{b}$

6 Synopsis of C-callable routines

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

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

The matrix \mathbf{A} is `m`-by-`n` with `nz` entries.

The `sys` argument of `umfpack_*_solve` is an integer in the range 0 to 14 which defines which linear system is to be solved. Valid values are listed in Table 3. The notation \mathbf{A}^H refers to the matrix transpose, which is the complex conjugate transpose for complex matrices (\mathbf{A}' in MATLAB). The array transpose is \mathbf{A}^T , which is $\mathbf{A}.'$ in MATLAB.

6.1 Primary routines: real/int

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

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

6.2 Alternative routines: real/int

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

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

6.3 Matrix manipulation routines: real/int

```
int Ti [nz], Tj [nz], P [m], Q [n], Rp [m+1], Ri [nz], Map [nz] ;
double Tx [nz], Rx [nz], Y [m], Z [m] ;

status = umfpack_di_col_to_triplet (n, Ap, Tj) ;
status = umfpack_di_triplet_to_col (m, n, nz, Ti, Tj, Tx, Ap, Ai, Ax, Map) ;
status = umfpack_di_transpose (m, n, Ap, Ai, Ax, P, Q, Rp, Ri, Rx) ;
status = umfpack_di_scale (Y, Z, Numeric) ;
```

6.4 Getting the contents of opaque objects: real/int

The filename string should be large enough to hold the name of a file.

```
int lnz, unz, Lp [m+1], Lj [lnz], Up [n+1], Ui [unz], do_recip ;
double Lx [lnz], Ux [unz], D [min (m,n)], Rs [m] ;
int nfr, nchains, P1 [m], Q1 [n], Front_npivcol [n+1], Front_parent [n+1], Front_1strow [n+1],
    Front_leftmostdesc [n+1], Chain_start [n+1], Chain_maxrows [n+1], Chain_maxcols [n+1] ;
char filename [100] ;

status = umfpack_di_get_lunz (&lnz, &unz, &m, &n, &nz_udiag, Numeric) ;
status = umfpack_di_get_numeric (Lp, Lj, Lx, Up, Ui, Ux, P, Q, D, &do_recip, Rs, Numeric) ;
status = umfpack_di_get_symbolic (&m, &n, &n1, &nz, &nfr, &nchains, P1, Q1,
    Front_npivcol, Front_parent, Front_1strow, Front_leftmostdesc,
    Chain_start, Chain_maxrows, Chain_maxcols, Symbolic) ;
status = umfpack_di_load_numeric (&Numeric, filename) ;
status = umfpack_di_save_numeric (Numeric, filename) ;
status = umfpack_di_load_symbolic (&Symbolic, filename) ;
status = umfpack_di_save_symbolic (Symbolic, filename) ;
```

6.5 Reporting routines: real/int

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

6.6 Primary routines: complex/int

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

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

6.7 Alternative routines: complex/int

```
double Wz [10*n] ;

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

6.8 Matrix manipulation routines: complex/int

```
double Tz [nz], Rz [nz], Yx [m], Yz [m], Zx [m], Zz [m] ;

status = umfpack_zi_col_to_triplet (n, Ap, Tj) ;
status = umfpack_zi_triplet_to_col (m, n, nz, Ti, Tj, Tx, Tz, Ap, Ai, Ax, Az, Map) ;
status = umfpack_zi_transpose (m, n, Ap, Ai, Ax, Az, P, Q, Rp, Ri, Rx, Rz, 1) ;
status = umfpack_zi_transpose (m, n, Ap, Ai, Ax, Az, P, Q, Rp, Ri, Rx, Rz, 0) ;
status = umfpack_zi_scale (Yx, Yz, Zx, Zz, Numeric) ;
```

6.9 Getting the contents of opaque objects: complex/int

```
double Lz [lnz], Uz [unz], Dx [min (m,n)], Dz [min (m,n)] ;

status = umfpack_zi_get_lunz (&lnz, &unz, &m, &n, &nz_udiag, Numeric) ;
status = umfpack_zi_get_numeric (Lp, Lj, Lx, Lz, Up, Ui, Ux, Uz, P, Q, Dx, Dz,
    &do_recip, Rs, Numeric) ;
status = umfpack_zi_get_symbolic (&m, &n, &n1, &nz, &nfr, &nchains, P1, Q1,
    Front_npivcol, Front_parent, Front_1strow, Front_leftmostdesc,
    Chain_start, Chain_maxrows, Chain_maxcols, Symbolic) ;
status = umfpack_zi_load_numeric (&Numeric, filename) ;
status = umfpack_zi_save_numeric (Numeric, filename) ;
status = umfpack_zi_load_symbolic (&Symbolic, filename) ;
status = umfpack_zi_save_symbolic (Symbolic, filename) ;
```

6.10 Reporting routines: complex/int

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

7 Synopsis of utility routines

These routines are the same in all four versions of UMFPACK.

```
double t, s [2] ;

t = umfpack_timer ( ) ;
umfpack_tic (s) ;
umfpack_toc (s) ;
```

8 Synopsis of AMD ordering routines

UMFPACK makes use of the AMD ordering package for its symmetric ordering strategy. You may also use these four user-callable routines in your own C programs. You need to include the `amd.h` file only if you make direct calls to the AMD routines themselves. The `int` versions are summarized below; `long` versions are also available. Refer to the AMD User Guide for more information, or to the file `amd.h` which documents these routines.

```
#include "amd.h"
double amd_control [AMD_CONTROL], amd_info [AMD_INFO] ;

amd_defaults (amd_control) ;
status = amd_order (n, Ap, Ai, P, amd_control, amd_info) ;
amd_control (amd_control) ;
amd_info (amd_info) ;
```

9 Using UMFPACK in a Fortran program

UMFPACK v4.1 includes a basic Fortran 77 interface to some of the C-callable UMFPACK routines. Since interfacing C and Fortran programs is not portable, this interface might not work with all C and Fortran compilers. Refer to Section 10 for more details. The following Fortran routines are provided. The list includes the C-callable routines that the Fortran interface routine calls. Refer to the corresponding C routines in Section 5 for more details on what the Fortran routine does.

- `umf4def`: sets the default control parameters (`umfpack_di_defaults`).
- `umf4sym`: pre-ordering and symbolic factorization (`umfpack_di_symbolic`).
- `umf4num`: numeric factorization (`umfpack_di_numeric`).
- `umf4solr`: solve a linear system with iterative refinement (`umfpack_di_solve`).
- `umf4sol`: solve a linear system without iterative refinement (`umfpack_di_solve`). Sets `Control [UMFPACK_IRSTEP]` to zero, and does not require the matrix **A**.
- `umf4scal`: scales a vector using UMFPACK's scale factors (`umfpack_di_scale`).
- `umf4fnum`: free the Numeric object (`umfpack_di_free_numeric`).
- `umf4fsym`: free the Symbolic object (`umfpack_di_free_symbolic`).
- `umf4pcon`: prints the control parameters (`umfpack_di_report_control`).
- `umf4pinf`: print statistics (`umfpack_di_report_info`).
- `umf4snum`: save the Numeric object to a file (`umfpack_di_save_numeric`).
- `umf4ssym`: save the Symbolic object to a file (`umfpack_di_save_symbolic`).
- `umf4lnum`: load the Numeric object from a file (`umfpack_di_load_numeric`).
- `umf4lsym`: load the Symbolic object from a file (`umfpack_di_load_symbolic`).

The matrix **A** is passed to UMFPACK in compressed column form, with 0-based indices. In Fortran, for an m -by- n matrix **A** with **nz** entries, the row indices of the first column (column 1) are in **Ai** (**Ap**(1)+1 ... **Ap**(2)), with values in **Ax** (**Ap**(1)+1 ... **Ap**(2)). The last column (column n) is in **Ai** (**Ap**(n)+1 ... **Ap**(n +1)) and **Ax** (**Ap**(n)+1 ... **Ap**(n +1)). The number of entries in the matrix is thus **nz** = **Ap** (n +1). The row indices in **Ai** are in the range 0 to m -1. They must be sorted, with no duplicate entries allowed. None of the UMFPACK routines modify the input matrix **A**. The following definitions apply for the Fortran routines:

```
integer m, n, Ap (n+1), Ai (nz), symbolic, numeric, filenum, status
double precision Ax (nz), control (20), info (90), x (n), b (n)
```

UMFPACK's status is returned in either a **status** argument, or in **info** (1). It is zero if UMFPACK was successful, 1 if the matrix is singular (this is a warning, not an error), and negative if an error occurred. Section 5.11 summarizes the possible values of **status** and **info** (1). See Table 3 for a list of the values of the **sys** argument. See Table 2 for a list of the control parameters (the Fortran usage is the same as the MATLAB usage for this array).

For the **Numeric** and **Symbolic** handles, it is probably safe to assume that a Fortran **integer** is sufficient to store a C pointer. If that does not work, try defining **numeric** and **symbolic** in your Fortran program as integer arrays of size 2. You will need to define them as **integer*8** if you compile UMFPACK in the 64-bit mode.

To avoid passing strings between C and Fortran in the load/save routines, a file number is passed instead, and the C interface constructs a file name (if **filenum** is 42, the **Numeric** file name is **n42.umf**, and the **Symbolic** file name is **s42.umf**).

The following is a summary of the calling sequence of each Fortran interface routine. An example of their use is in the **Demo/umf4hb.f** file. That routine also includes an example of how to convert a 1-based sparse matrix into 0-based form. For more details on the arguments of each routine, refer to the arguments of the same name in the corresponding C-callable routine, in Sections 15 through 20. The only exception is the **control** argument of **umf4sol**, which sets **control** (8) to zero to disable iterative refinement. Note that the solve routines do not overwrite **b** with the solution, but return their solution in a different array, **x**.

```
call umf4def (control)
call umf4sym (m, n, Ap, Ai, Ax, symbolic, control, info)
call umf4num (Ap, Ai, Ax, symbolic, numeric, control, info)
call umf4solr (sys, Ap, Ai, Ax, x, b, numeric, control, info)
call umf4sol (sys, x, b, numeric, control, info)
call umf4scal (x, b, numeric, status)
call umf4fnum (numeric)
call umf4fsym (symbolic)
call umf4pcon (control)
call umf4pinf (control)
call umf4snum (numeric, filenum, status)
call umf4ssym (symbolic, filenum, status)
call umf4lnum (numeric, filenum, status)
call umf4lsym (symbolic, filenum, status)
```

Access to the complex routines in UMFPACK is provided by the interface routines in **umf4_f77wrapper.c**. The following is a synopsis of each routine. All the arguments are the same as the real versions, except **Az**, **xz**, and **bz** are the imaginary parts of the matrix, solution, and right-hand-side, respectively. The **Ax**, **x**, and **b** are the real parts.

```
call umf4zdef (control)
```

```

call umf4zsym (m, n, Ap, Ai, Ax, Az, symbolic, control, info)
call umf4znum (Ap, Ai, Ax, Az, symbolic, numeric, control, info)
call umf4zsolr (sys, Ap, Ai, Ax, Az, x, xz, b, bz, numeric, control, info)
call umf4zsol (sys, x, xz, b, bz, numeric, control, info)
call umf4zscal (x, xz, b, bz, numeric, status)
call umf4zfnum (numeric)
call umf4zfsym (symbolic)
call umf4zpcon (control)
call umf4zpinf (control)
call umf4zsnum (numeric, filenum, status)
call umf4zssym (symbolic, filenum, status)
call umf4zlnum (numeric, filenum, status)
call umf4zlsym (symbolic, filenum, status)

```

10 Installing the C library

The following discussion assumes you have the `make` program, either in Unix, or in Windows with Cygwin². You can skip this section and go to next one if all you want to use is the UMFPACK and AMD mexFunctions in MATLAB.

You will need to install both UMFPACK v4.1 and AMD v1.0 to use UMFPACK. The UMFPACK and AMD subdirectories must be placed side-by-side within the same directory. AMD is a stand-alone package that is required by UMFPACK. UMFPACK can be compiled without the BLAS [11, 12, 28, 25], but your performance will be much less than what it should be.

System-dependent configurations are in the AMD/Make and UMFPACK/Make directories (the UMFPACK/Make directory is actually just a symbolic link to AMD/Make³). You can edit the Make.include files in either of those directories to customize the compilation. The default settings will work on most systems, except that UMFPACK will be compiled so that it does not use the BLAS. Sample configuration files are provided for Linux, Sun Solaris, SGI IRIX, IBM AIX, and the DEC/Compaq Alpha.

To compile and install both packages, go to the UMFPACK directory and type `make`. This will compile the libraries (AMD/Lib/libamd.a and UMFPACK/Lib/libumfpack.a). A demo of the AMD ordering routine will be compiled and tested in the AMD/Demo directory, and five demo programs will then be compiled and tested in the UMFPACK/Demo directory. The outputs of these demo programs will then be compared with output files in the distribution. Expect to see a few differences, such as residual norms, compile-time control settings, and perhaps memory usage differences. The AMD and UMFPACK mexFunctions for use in MATLAB will also be compiled. If you do not have MATLAB 6.0 or later, type `make lib` instead.

If you have the GNU version of `make`, the Source/GNUMakefile and MATLAB/GNUMakefile files are used. These are much more concise than what the “old” version of `make` can handle. If you do not have GNU `make`, the Source/Makefile and MATLAB/Makefile files are used instead. Each UMFPACK source file is compiled into four versions (`double` / `complex`, and `int` / `long`). A proper old-style `Makefile` is cumbersome in this case, so these two `Makefile`’s have been constructed by brute force. They ignore dependencies, and simply compile everything. I highly recommend using GNU `make` if you wish to modify UMFPACK.

If you compile UMFPACK and AMD and then later change the Make.include file or your system-specific configuration file such as Make.linux, then you should type `make purge` and then `make` to recompile.

²www.cygwin.com

³Windows might not extract the symbolic link UMFPACK/Make correctly. If it doesn’t, simply create the UMFPACK/Make folder by copying it from AMD/Make.

Here are the various parameters that you can control in your `Make.include` file:

- `CC` = your C compiler, such as `cc`.
- `RANLIB` = your system's `ranlib` program, if needed.
- `CFLAGS` = optimization flags, such as `-O`. Add `-DLP64` if you are compiling in 64-bit mode (32 bit `int`'s, 64 bit `long`'s, and 64 bit pointers).
- `CONFIG` = configuration settings for the BLAS, memory allocation routines, and timing routines.
- `LIB` = your libraries, such as `-lm` or `-lblas`.
- `RM` = the command to delete a file.
- `MV` = the command to rename a file.
- `MEX` = the command to compile a MATLAB mexFunction. If you are using MATLAB 5, you need to add `-DNBLAS` and `-DNUTIL` to this command. An example is provided in the `Make/Make.include` file.
- `F77` = the command to compile a Fortran program (optional).
- `F77FLAGS` = the Fortran compiler flags (optional).
- `F77LIB` = the Fortran libraries (optional).

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

- `-DNBLAS` if you do not have any BLAS at all.
- `-DCBLAS` if you have the C-BLAS [28].
- `-DNSUNPERF` if you are on Solaris but do not have the Sun Performance Library (for the BLAS).
- `-DNSCSL` if you on SGI IRIX but do not have the SCSL BLAS library.
- `-DLONGBLAS` if your BLAS can take `long` integer input arguments. If not defined, then the `umfpack_*1_*` versions of UMFPACK that use `long` integers do not call the BLAS. This flag is set internally when using the Sun Performance BLAS or SGI's SCSL BLAS (both have 64-bit versions of the BLAS).
- Options for controlling how C calls the Fortran BLAS: `-DBLAS_BY_VALUE`, `-DBLAS_NO_UNDERSCORE`, and `-DBLAS_CHAR_ARG`. These are set automatically for Windows, Sun Solaris, SGI Irix, Red Hat Linux, Compaq Alpha, and AIX (the IBM RS 6000). They are ignored if you are using the C-BLAS interface to the BLAS.
- `-DGETRUSAGE` if you have the `getrusage` function.
- `-DNUTIL` if you wish to compile the MATLAB-callable UMFPACK mexFunction with the `mxMalloc`, `mxRealloc` and `mxFree` routines, instead of the undocumented (but superior) `utMalloc`, `utRealloc`, and `utFree` routines. The default is to use the `ut*` routines on Unix, and the `mx*` routines on Windows.

- `-DNPOSIX` if you do not have the POSIX-compliant `sysconf` and `times` routines used by `umfpack_tic` and `umfpack_toc`.
- `-DNRECIPROCAL` controls a trade-off between speed and accuracy. If defined (or if the pivot value itself is less than 10^{-12}), then the pivot column is divided by the pivot value during numeric factorization. Otherwise, it is multiplied by the reciprocal of the pivot, which is faster but can be less accurate. The default is to multiply by the reciprocal unless the pivot value is small. This option also modifies how the rows of the matrix **A** are scaled. If `-DNRECIPROCAL` is defined (or if any scale factor is less than 10^{-12}), entries in the rows of **A** are divided by the scale factors. Otherwise, they are multiplied by the reciprocal. When compiling the complex routines with the GNU `gcc` compiler, the pivot column is always divided by the pivot entry, because of a numerical accuracy issue encountered with `gcc` version 3.2 with a few complex matrices on a Pentium 4M (running Linux). You can still use `-DNRECIPROCAL` to control how the scale factors for the rows of **A** are applied.

If a Fortran BLAS package is used you may see compiler warnings. The BLAS routines `dgemm`, `dgemv`, `dger`, `dtrsm`, `dtrsv`, `dscal` and their corresponding complex versions are used. Header files are not provided for the Fortran BLAS. You may safely ignore all of these warnings.

I highly recommend the recent BLAS by Goto and van de Geijn [25]. Using this BLAS increased the performance of UMFPACK by up to 50% on a Dell Latitude C840 laptop (2GHz Pentium 4M, 512K L2 cache, 1GB main memory). The peak performance of `umfpack_di_numeric` with Goto and van de Geijn's BLAS is 1.6 Gflops on this computer. In MATLAB, the peak performance of UMFPACK on a dense matrix (stored in sparse format) is 900 Mflops, as compared to 1 Gflop for $\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$ when **A** is stored as a regular full matrix.

When you compile your program that uses the C-callable UMFPACK library, you need to link your program with both libraries (`UMFPACK/Lib/libumfpack.a` and `AMD/Lib/libamd.a`) and you need to tell your compiler to look in the directories `UMFPACK/Include` and `AMD/Include` for include files. See `UMFPACK/Demo/Makefile` for an example. You do not need to directly include any AMD include files in your program, unless you directly call AMD routines. You only need the

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

statement, as described in Section 6.

If you would like to compile both 32-bit and 64-bit versions of the libraries, you will need to do it in two steps. Modify your `Make/Make.<arch>` file, and select the 32-bit option. Type `make` in the UMFPACK directory, which creates the `UMFPACK/Lib/libumfpack.a` and `AMD/Lib/libamd.a` libraries. Rename those two files. Edit your `Make/Make.<arch>` and select the 64-bit option. Type `make purge`, and then `make`, and you will create the 64-bit libraries. You can use the same `umfpack.h` include file for both 32-bit and 64-bit versions. Simply link your program with the appropriate 32-bit or 64-bit compiled version of the UMFPACK and AMD libraries.

Type `make hb` in the `UMFPACK/Demo/HB` directory to compile and run a C program that reads in and factorizes Harwell/Boeing matrices. Note that this uses a stand-alone Fortran program to read in the Fortran-formatted Harwell/Boeing matrices and write them to a file which can be read by a C program.

11 Installing the MATLAB interface

If all you want to do is use the UMFPACK mexFunction in MATLAB, you can skip the use of the `make` command described above. Simply type `umfpack_make` in MATLAB while in the

UMFPACK/MATLAB directory. You can also type `amd_make` in the AMD/MATLAB directory to compile the stand-alone AMD mexFunction (this is not required to compile the UMFPACK mexFunction). This works on any computer with MATLAB, including Windows. This method of creating the UMFPACK mexFunction has been tested with MATLAB 5.3, 6.1, and 6.5.

You will be prompted to select several configuration options, including whether or not to use the BLAS. MATLAB 5.3 (or earlier) does not include the BLAS, so you either have to compile UMFPACK without the BLAS (UMFPACK will be slow), or modify your `<matlab>/bin/mexopts.sh` by adding your BLAS library to the CLIBS string, where `<matlab>` is the directory in which MATLAB is installed.

If you are using Windows and the `lcc` compiler bundled with MATLAB 6.1, then you may need to copy the `UMFPACK\MATLAB\lcc_lib\libmwlpack.lib` file into the `<matlab>\extern\lib\win32\lcc\` directory. Next, type `mex -setup` at the MATLAB prompt, and ask MATLAB to select the `lcc` compiler. MATLAB 6.1 has built-in BLAS, but in that version of MATLAB the BLAS cannot be accessed by a mexFunction compiled by `lcc` without first copying this file to the location listed above. If you have MATLAB 6.5 or later, you can probably skip this step.

12 Installing the Fortran interface

Once the 32-bit C-callable UMFPACK library is compiled, you can also compile the Fortran interface, by typing `make fortran`. This will create the `umf4hb` program, test it, and compare the output with the file `umf4hb.out` in the distribution. If you compiled UMFPACK in 64-bit mode, you need to use `make fortran64` instead, which compiles the `umf4hb64` program and compares its output with the file `umf4hb64.out`. Refer to the comments in the `Demo/umf4_f77wrapper.c` file for more details.

This interface is **highly** non-portable, since it depends on how C and Fortran are interfaced. Because of this issue, the interface is included in the `Demo` directory, and not as a primary part of the UMFPACK library. The interface routines are not included in the compiled `UMFPACK/Lib/libumfpack.a` library, but left as stand-alone compiled files (`umf4_f77wrapper.o` and `umf4_f77wrapper64.o` in the `Demo` directory). You may need to modify the interface routines in the file `umf4_f77wrapper.c` if you are using compilers for which this interface has not been tested.

13 Known Issues

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

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

If your compiler breaks this test, then UMFPACK will fail catastrophically if it encounters a NaN. You will not just see NaN's in your output; UMFPACK will probably crash with a segmentation fault. In that case, you might try to see if the common (but non-ANSI C) routine `isnan` is

available, and modify the macro `SCALAR_IS_NAN` in `umf_version.h` accordingly. The simpler (and IEEE 754-compliant) test `(x != x)` is always true with Linux on a PC, and on every Unix compiler I have tested.

Some compilers will complain about the Fortran BLAS being defined implicitly. C prototypes for the BLAS are not used, except the C-BLAS. Some compilers will complain about unrecognized `#pragma`'s. You may safely ignore all of these warnings.

14 Future work

Here are a few features that are not in UMFPACK Version 4.1, in no particular order. They may appear in a future release of UMFPACK. If you are interested, let me know and I could consider including them:

1. Future versions may have different default `Control` parameters. Future versions may return more statistics in the `Info` array, and they may use more entries in the `Control` array. These two arrays will probably become larger, since there are very few unused entries. If they change in size, the constants `UMFPACK_CONTROL` and `UMFPACK_INFO` defined in `umfpack.h` will be changed to reflect their new size. Your C program should use these constants when declaring the size of these two arrays. Do not define them as `Control` [20] and `Info` [90].
2. Forward/back solvers for the conventional row or column-form data structure for **L** and **U** (the output of `umfpack*.di_get_numeric`). This would enable a separate solver that could be used to write a MATLAB mexFunction `x = lu_refine(A, b, L, U, P, Q, R)` that gives MATLAB access to the iterative refinement algorithm with sparse backward error analysis. It would also be easier to handle sparse right-hand-sides in this data structure, and end up with good asymptotic run-time in this case (particularly for $\mathbf{Lx} = \mathbf{b}$; see [24]).
3. Complex absolute value computations could be based on FDLIBM (see <http://www.netlib.org/fdlibm>), using the `hypot(x,y)` routine.
4. When using iterative refinement, the residual $\mathbf{Ax} - \mathbf{b}$ could be returned by `umfpack_solve`.
5. The solve routines could handle multiple right-hand sides, and sparse right-hand sides. See `umfpack_solve` for the MATLAB version of this feature.
6. An incomplete LU factorization, by dropping small entries from **L** and **U**.
7. An option to redirect the error and diagnostic output.
8. Permutation to block-triangular-form [17] for the C-callable interface. There are two routines in the ACM Collected Algorithms (529 and 575) [14, 16] that could be translated from Fortran to C and included in UMFPACK. This would result in better performance for matrices from circuit simulation and chemical process engineering. See `umfpack_btf.m` for the MATLAB version of this feature.
9. The ability to use user-provided `malloc`, `free`, and `realloc` memory allocation routines. Note that UMFPACK makes very few calls to these routines. You can do this at compile-time by modifying the definitions of `ALLOCATE`, `FREE`, and `REALLOCATE` in the file `umf_internal.h`. Be sure to document your changes carefully when you change UMFPACK source code.

10. The ability to use user-provided work arrays, so that `malloc`, `free`, and `realloc` are not called. The `umfpack*_wsolve` routine is one example.
11. A method that takes time proportional to the number of nonzeros in \mathbf{A} to compute the symbolic factorization [23]. This would improve the performance of the symmetric and 2-by-2 strategies, and the unsymmetric strategy when dense rows are present. The current method takes time proportional to the number of nonzeros in the upper bound of \mathbf{U} . The method used in UMFPACK exploits super-columns, however, so this bound is rarely reached.
12. The complex versions could use ANSI C99 `double _Complex` arguments, and support the use of interleaved real/imaginary parts as input and output arguments. The `umfpack*_report_vector` routine is one example.
13. Other basic sparse matrix operations, such as sparse matrix multiplication, could be included.
14. A more complete Fortran interface.
15. A C++ interface.
16. A parallel version using MPI. This would require a large amount of effort.

15 The primary UMFPACK routines

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

15.1 `umfpack_*_symbolic`

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

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

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

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

double int Syntax:
```



```

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

```

double long Syntax:

```

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

```

complex int Syntax:

```

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

```

complex long Syntax:

```

#include "umfpack.h"
void *Symbolic ;
long n_row, n_col, *Ap, *Ai, status ;
double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO], *Ax, *Az ;
status = umfpack_zl_symbolic (n_row, n_col, Ap, Ai, Ax, Az,
    &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 COLAMD or AMD) and a symbolic factorization. This is required before the matrix can be numerically factorized with umfpack*_numeric. If you wish to bypass the COLAMD or AMD pre-ordering and provide your own ordering, use umfpack*_qsymbolic instead.

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

For the following discussion, let S be the submatrix of A obtained after eliminating all pivots of zero Markowitz cost. S has dimension (n_row-n1-nempty_row) -by- (n_col-n1-nempty_col), where
n1 = Info [UMFPACK_COL_SINGLETONS] + Info [UMFPACK_ROW_SINGLETONS],
nempty_row = Info [UMFPACK_EMPTY_ROW] and
nempty_col = Info [UMFPACK_EMPTY_COL].

Returns:

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

Arguments:

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

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

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

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

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

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

double Ax [nz] ; Optional input argument, not modified.

The numerical values of the sparse matrix A. The nonzero pattern (row indices) for column j is stored in Ai [(Ap [j]) ... (Ap [j+1]-1)], and the corresponding numerical values are stored in Ax [(Ap [j]) ... (Ap [j+1]-1)]. Used only by the 2-by-2 strategy to determine whether entries are "large" or "small". You do not have to pass the same numerical values to umfpack*_numeric. If Ax is not present (a (double *) NULL pointer), then any entry in A is assumed to be "large".

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

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

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

Used by the 2-by-2 strategy only. See the description of Ax, above.

void **Symbolic ; Output argument.

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

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

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

Control [UMFPACK_STRATEGY]: This is the most important control parameter. It determines what kind of ordering and pivoting strategy that UMFPACK should use. It is new to Version 4.1
There are 4 options:

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

UMFPACK_STRATEGY_UNSYMMETRIC: Use the unsymmetric strategy. COLAMD is used to order the columns of A, followed by a postorder of the column elimination tree. No attempt is made to perform diagonal pivoting. The column ordering is refined during factorization. This strategy was the only one provided with UMFPACK V4.0.

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

UMFPACK_STRATEGY_SYMMETRIC: Use the symmetric strategy (new to Version 4.1). In this method, the approximate minimum degree ordering (AMD) is applied to $A+A'$, followed by a postorder of the elimination tree of $A+A'$. UMFPACK attempts to perform diagonal pivoting during numerical factorization. No refinement of the column pre-ordering is performed during factorization.

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

UMFPACK_STRATEGY_2BY2: a row permutation P2 is found that places large entries on the diagonal. The matrix $P2*A$ is then factorized using the symmetric strategy, described above. Refer to the User Guide for more information.

Control [UMFPACK_DENSE_COL]:

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

Control [UMFPACK_DENSE_ROW]:

Rows with more than $\max(16, \text{Control [UMFPACK_DENSE_ROW]} * 16 * \sqrt{n_{\text{col}}})$ entries are placed last in the row pre-ordering. Default: 0.2.

$\sqrt{n_{\text{col}}}$) entries are treated differently in the COLAMD pre-ordering, and in the internal data structures during the subsequent numeric factorization. Default: 0.2.

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

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

Control [UMFPACK_2BY2_TOLERANCE]: a diagonal entry $S(k,k)$ is considered "small" if it is $< \text{tol} * \max(\text{abs}(S(:,k)))$, where S a submatrix of the scaled input matrix, with pivots of zero Markowitz cost removed.

Control [UMFPACK_SCALE]: This parameter is new to V4.1. See umfpack_numeric.h for a description. Only affects the 2-by-2 strategy. Default: UMFPACK_SCALE_SUM.

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

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

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

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

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

UMFPACK_OK

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

UMFPACK_ERROR_n_nonpositive

n is less than or equal to zero.

UMFPACK_ERROR_invalid_matrix

Number of entries in the matrix is negative, $A_p[0]$ is nonzero,

a column has a negative number of entries, a row index is out of bounds, or the columns of input matrix were jumbled (unsorted columns or duplicate entries).

UMFPACK_ERROR_out_of_memory

Insufficient memory to perform the symbolic analysis. If the analysis requires more than 2GB of memory and you are using the 32-bit ("int") version of UMFPACK, then you are guaranteed to run out of memory. Try using the 64-bit version of UMFPACK.

UMFPACK_ERROR_argument_missing

One or more required arguments is missing.

UMFPACK_ERROR_internal_error

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

Note that the UMFPACK_ERROR_problem_too_large error code is no longer returned (it was in Version 4.0).

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

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

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

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

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

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

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

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

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

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

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

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

ordered last in the factorization, to the right of "dense" columns.

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

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

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

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

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

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

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

Info [UMFPACK_PEAK_MEMORY_ESTIMATE]: an estimate of the total amount of memory (in Units) required by umfpack_*_symbolic and umfpack_*_numeric to perform both the symbolic and numeric factorization. This is the larger of the amount of memory needed in umfpack_*_numeric itself, and the amount of memory needed in umfpack_*_symbolic (Info [UMFPACK_SYMBOLIC_PEAK_MEMORY]). The count includes the size of both the Symbolic and Numeric objects themselves. It can be a very loose upper bound, particularly when the symmetric or 2-by-2 strategies are used.

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

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

```
flops = 2*Lnz*Unz + sum (Lnz) ;
```

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

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

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

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

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

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

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

The rest of the statistics are new to Version 4.1:

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

Info [UMFPACK_STRATEGY_USED]: The ordering strategy used:
UMFPACK_STRATEGY_SYMMETRIC, UMFPACK_STRATEGY_UNSYMMETRIC, or
UMFPACK_STRATEGY_2BY2.

Info [UMFPACK_ORDERING_USED]: The ordering method used:
UMFPACK_ORDERING_COLAMD or UMFPACK_ORDERING_AMD. It can be
UMFPACK_ORDERING_GIVEN for umfpack*_qsymbolic.

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

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

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

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

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

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

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

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

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

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

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

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

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

Info [UMFPACK_SYMMETRIC_FLOPS]: The floating-point operation count in the numerical factorization phase, assuming no pivoting. If the pattern of the matrix is symmetric, this is normally a lower bound on the floating-point operation count in the actual numerical factorization, for matrices that fit the criteria for the symmetric or 2-by-2 strategies

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

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

The following statistics are computed only if the 2-by-2 strategy is used or attempted:

Info [UMFPACK_2BY2_NWEAK]: the number of small diagonal entries in S.

Info [UMFPACK_2BY2_UNMATCHED]: the number of small diagonal entries in $P2*S$.

Info [UMFPACK_2BY2_PATTERN_SYMMETRY]: the symmetry of $P2*S$.

Info [UMFPACK_2BY2_NZ_PA_PLUS_AT]: the number of off-diagonal entries in $(P2*S)+(P2*S)'$.

Info [UMFPACK_2BY2_NZDIAG]: the number of nonzero entries on the diagonal of $P2 \times S$.

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.

15.2 umfpack_*_numeric

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

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

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

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

double int Syntax:

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

double long Syntax:

#include "umfpack.h"
```

```

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

```

complex int Syntax:

```

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

```

complex long Syntax:

```

#include "umfpack.h"
void *Symbolic, *Numeric ;
long *Ap, *Ai, status ;
double *Ax, *Az, Control [UMFPACK_CONTROL], Info [UMFPACK_INFO] ;
status = umfpack_zl_numeric (Ap, Ai, Ax, Az, 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, PRAQ=LU, or P(R\A)Q=LU, where P and Q are permutation matrices (represented as permutation vectors), R is the row scaling, L is unit-lower triangular, and U is upper triangular. This is required before the system Ax=b (or other related linear systems) can be solved. umfpack_*_numeric can be called multiple times for each call to umfpack_*_symbolic, to factorize a sequence of matrices with identical nonzero pattern. Simply compute the Symbolic object once, with umfpack_*_symbolic, and reuse it for subsequent matrices. This routine safely detects if the pattern changes, and sets an appropriate error code.

Returns:

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

Arguments:

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

This must be identical to the Ap array passed to umfpack_*_symbolic. The value of n_col is what was passed to umfpack_*_symbolic (this is held in the Symbolic object).

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

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

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

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

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

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

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

void *Symbolic ; Input argument, not modified.

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

void **Numeric ; Output argument.

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

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

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

Control [UMFPACK_PIVOT_TOLERANCE]: relative pivot tolerance for threshold partial pivoting with row interchanges. In any given column, an entry is numerically acceptable if its absolute value is greater than or equal to Control [UMFPACK_PIVOT_TOLERANCE] times the largest absolute value in the column. A value of 1.0 gives true partial pivoting. If less than or equal to zero, then any nonzero entry is numerically acceptable as a pivot (this is changed from Version 4.0). Default: 0.1.

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

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

Control [UMFPACK_SYM_PIVOT_TOLERANCE]: This parameter is new to V4.1. If diagonal pivoting is attempted (the symmetric or symmetric-2by2 strategies are used) then this parameter is used to control when the diagonal entry is selected in a given pivot column. The absolute value of the entry must be \geq Control [UMFPACK_SYM_PIVOT_TOLERANCE] times the largest absolute value in the column. A value of zero will ensure that no off-diagonal pivoting is performed, except that zero diagonal entries are not selected if there are any off-diagonal

nonzero entries.

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

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

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

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

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

Control [UMFPACK_SCALE]: This parameter is new to V4.1. Version 4.0 did not scale the matrix. Note that the user's input matrix is never modified, only an internal copy is scaled.

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

UMFPACK_SCALE_NONE: no scaling is performed.

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

UMFPACK_SCALE_MAX: each row of the input matrix A is divided by the maximum the absolute values of the entries in that row. In the scaled matrix the largest entry in each row has a magnitude exactly equal to 1.

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

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

Default: UMFPACK_SCALE_SUM.

Control [UMFPACK_ALLOC_INIT]: This parameter has changed in V4.1.

When `umfpack_*_numeric` starts, it allocates memory for the Numeric object. Part of this is of fixed size (approximately n double's + $12*n$ integers). The remainder is of variable size, which grows to hold the LU factors and the frontal matrices created during

factorization. A estimate of the upper bound is computed by `umfpack_*_symbolic`, and returned by `umfpack_*_symbolic` in `Info [UMFPACK_VARIABLE_PEAK_ESTIMATE]` (in Units).

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

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

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

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

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

The strategy of attempting to "malloc" a working space, and re-trying with a smaller space, may not work under MATLAB, since `mxMalloc` aborts the `mexFunction` if it fails. The built-in `umfpack` routine (version 4.0) in MATLAB 6.5 uses `utMalloc` instead, which avoids this problem. As a `mexFunction`, `utMalloc` is used unless `-DNUTIL` is defined at compile time. The `utMalloc` routine, and `utFree` and `utRealloc`, are not documented. If the `mexFunction` doesn't work, then compile it with `-DNUTIL` instead.

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

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

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

Control `[UMFPACK_FRONT_ALLOC_INIT]`: This parameter is new to V4.1.

When `UMFPACK` starts the factorization of each "chain" of frontal

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

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

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

Default: 0.5.

double Info [UMFPACK_INFO] ; Output argument.

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

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

UMFPACK_OK

Numeric factorization was successful. umfpack_*_numeric computed a valid numeric factorization.

UMFPACK_WARNING_singular_matrix

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

UMFPACK_ERROR_out_of_memory

Insufficient memory to complete the numeric factorization.

UMFPACK_ERROR_argument_missing

One or more required arguments are missing.

UMFPACK_ERROR_invalid_Symbolic_object

Symbolic object provided as input is invalid.

UMFPACK_ERROR_different_pattern

The pattern (Ap and/or Ai) has changed since the call to umfpack_*_symbolic which produced the Symbolic object.

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

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

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

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

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

Info [UMFPACK_VARIABLE_PEAK]: the peak size (in Units) of the variable-sized part of the Numeric object. This size is the amount of space actually used inside the block of memory, not the space allocated via UMF_malloc. You can reduce UMFPACK's memory requirements by setting Control [UMFPACK_ALLOC_INIT] to the ratio Info [UMFPACK_VARIABLE_PEAK] / Info[UMFPACK_VARIABLE_PEAK_ESTIMATE]. This will ensure that no memory reallocations occur (you may want to add 0.001 to make sure that integer roundoff does not lead to a memory size that is 1 Unit too small; otherwise, garbage collection and reallocation will occur).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Info [UMFPACK_RCOND]: A rough estimate of the condition number, equal

to $\min(\text{abs}(\text{diag}(U))) / \max(\text{abs}(\text{diag}(U)))$, or zero if the diagonal of U is all zero.

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

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

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

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

The following statistics were added to Version 4.1:

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

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

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

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

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

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

Info [UMFPACK_ALLOC_INIT_USED]: the initial allocation parameter used.

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

Info [UMFPACK_NOFF_DIAG]: number of off-diagonal pivots selected, if the symmetric or 2-by-2 strategies are used.

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.

15.3 umfpack_*_solve

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

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

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

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

double int Syntax:

    #include "umfpack.h"
```

```

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

```

double long Syntax:

```

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

```

complex int Syntax:

```

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

```

complex long Syntax:

```

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

```

Purpose:

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

Returns:

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

Arguments:

Int sys ; Input argument, not modified.

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

sys value	system solved
UMFPACK_A	Ax=b
UMFPACK_At	A'x=b
UMFPACK_Aat	A.'x=b
UMFPACK_Pt_L	P'Lx=b

UMFPACK_L	Lx=b
UMFPACK_Lt_P	L'Px=b
UMFPACK_Lat_P	L.'Px=b
UMFPACK_Lt	L'x=b
UMFPACK_U_Qt	UQ'x=b
UMFPACK_U	Ux=b
UMFPACK_Q_Ut	QU'x=b
UMFPACK_Q_Uat	QU.'x=b
UMFPACK_Ut	U'x=b
UMFPACK_Uat	U.'x=b

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

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

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

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

```
Int Ap [n+1] ;      Input argument, not modified.
Int Ai [nz] ;       Input argument, not modified.
double Ax [nz] ;    Input argument, not modified.
double Az [nz] ;    Input argument, not modified, for complex versions.
```

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

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

```
double X [n] ;      Output argument.
or:
double Xx [n] ;     Output argument, real part.
double Xz [n] ;     Output argument, imaginary part.
```

The solution to the linear system, where $n = n_{\text{row}} = n_{\text{col}}$ is the dimension of the matrices A, L, and U.

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

```
double B [n] ;      Input argument, not modified.
or:
double Bx [n] ;     Input argument, not modified, real part.
```

double Bz [n] ; Input argument, not modified, imaginary part.

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

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

void *Numeric ; Input argument, not modified.

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

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

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

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

double Info [UMFPACK_INFO] ; Output argument.

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

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

UMFPACK_OK

The linear system was successfully solved.

UMFPACK_WARNING_singular_matrix

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

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

UMFPACK_ERROR_out_of_memory

Insufficient memory to solve the linear system.

UMFPACK_ERROR_argument_missing

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

UMFPACK_ERROR_invalid_system

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

UMFPACK_ERROR_invalid_Numeric_object

The Numeric object is not valid.

Info [UMFPACK_NROW], Info [UMFPACK_NCOL]:

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

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

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

Info [UMFPACK_IR_ATTEMPTED]: The number of iterative refinement steps attempted. The number of times a linear system was solved is one more than this (once for the initial Ax=b, and once for each Ay=r solved for each iterative refinement step attempted).

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

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

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

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

The following statistic was added to Version 4.1:

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

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

15.4 umfpack_*_free_symbolic

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

```
void umfpack_dl_free_symbolic
(
    void **Symbolic
) ;
```

```
void umfpack_zi_free_symbolic
(
    void **Symbolic
) ;
```

```
void umfpack_zl_free_symbolic
(
    void **Symbolic
) ;
```

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:

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

Arguments:

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

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

15.5 umfpack_*_free_numeric

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

```
void umfpack_dl_free_numeric
(
    void **Numeric
) ;
```

```
void umfpack_zi_free_numeric
(
    void **Numeric
) ;
```

```
void umfpack_zl_free_numeric
(
    void **Numeric
) ;
```

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:

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

Arguments:

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

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

16 Alternative routines

16.1 umfpack_*_defaults

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

void umfpack_dl_defaults
(
    double Control [UMFPACK_CONTROL]
) ;

void umfpack_zi_defaults
(
    double Control [UMFPACK_CONTROL]
) ;

void umfpack_zl_defaults
(
    double Control [UMFPACK_CONTROL]
) ;

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:

Sets the default control parameter settings.

NOTE: new control parameters have been added to the Control array for Version 4.1. These entries were unused in Version 4.0. The default block size for the BLAS has increased from 24 to 32. Some rarely used control parameters have been removed (those that controlled relaxed amalgamation).

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).

16.2 umfpack_*_qsymbolic

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

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

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

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

double int Syntax:

    #include "umfpack.h"
```

```

void *Symbolic ;
int n_row, n_col, *Ap, *Ai, *Qinit, status ;
double Control [UMFPACK_CONTROL], Info [UMFPACK_INFO], *Ax ;
status = umfpack_di_ksymbolic (n_row, n_col, Ap, Ai, Ax, Qinit,
    &Symbolic, Control, Info) ;

```

double long Syntax:

```

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

```

complex int Syntax:

```

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

```

complex long Syntax:

```

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

```

Purpose:

Given the nonzero pattern of a sparse matrix A in column-oriented form, and a sparsity preserving column pre-ordering Qinit, umfpack_*_ksymbolic performs the symbolic factorization of A*Qinit (or A (:,Qinit) in MATLAB notation). This is identical to umfpack_*_symbolic, except that neither COLAMD nor AMD are called and the user input column order Qinit is used instead. Note that in general, the Qinit passed to umfpack_*_ksymbolic can differ from the final Q found in umfpack_*_numeric. The unsymmetric strategy will perform a column etree postordering done in umfpack_*_ksymbolic and sparsity-preserving modifications are made within each frontal matrix during umfpack_*_numeric. The symmetric and 2-by-2 strategies will preserve Qinit, unless the matrix is structurally singular.

See umfpack_*_symbolic for more information.

*** 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.

Returns:

The value of Info [UMFPACK_STATUS]; see umfpack_*_symbolic.
Also returns UMFPACK_ERROR_invalid_permutation if Qinit is not a valid

permutation vector.

Arguments:

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

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

The user's fill-reducing initial column pre-ordering. This must be a permutation of `0..n_col-1`. If `Qinit [k] = j`, then column `j` is the `k`th column of the matrix `A(:,Qinit)` to be factorized. If `Qinit` is an `(Int *)` NULL pointer, then `COLAMD` or `AMD` are called instead.

16.3 umfpack_*_wsolve

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

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

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

long umfpack_zl_wsolve
(
    long sys,
    const long Ap [ ],
    const long Ai [ ],
    const double Ax [ ], const double Az [ ],
    double Xx [ ],      double Xz [ ],
    const double Bx [ ], const double Bz [ ],
    void *Numeric,
    const double Control [UMFPACK_CONTROL],
```

```

    double Info [UMFPACK_INFO],
    long Wi [ ],
    double W [ ]
) ;

```

double int Syntax:

```

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

```

double long Syntax:

```

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

```

complex int Syntax:

```

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

```

complex long Syntax:

```

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

```

Purpose:

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

Returns:

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

Arguments:


```

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

```

for complex versions:

```

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

```

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

```

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

```

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

	no iter. refinement	with iter. refinement
<code>umfpack_di_wsolve</code>	n	$5*n$
<code>umfpack_dl_wsolve</code>	n	$5*n$
<code>umfpack_zi_wsolve</code>	$4*n$	$10*n$
<code>umfpack_zl_wsolve</code>	$4*n$	$10*n$

17 Matrix manipulation routines

17.1 umfpack*_col_to_triplet

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:
```

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

Returns:

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

Arguments:

Int n_col ; Input argument, not modified.

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

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

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

Int Tj [nz] ; Output argument.

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

17.2 umfpack*_triplet_to_col

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

long Syntax:

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

Purpose:

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

The triplet form of a matrix is a very simple data structure for basic sparse matrix operations. For example, suppose you wish to factorize a matrix A coming from a finite element method, in which A is a sum of dense submatrices, $A = E1 + E2 + E3 + \dots$. The entries in each element matrix Ei can be concatenated together in the three triplet arrays, and any overlap between the elements will be correctly summed by `umfpack_*_triplet_to_col`.

Transposing a matrix in triplet form is simple; just interchange the use of Ti and Tj . You can construct the complex conjugate transpose by negating Tz , for the complex versions.

Permuting a matrix in triplet form is also simple. If you want the matrix PAQ, or A (P,Q) in MATLAB notation, where $P[k] = i$ means that row i of A is the k th row of PAQ and $Q[k] = j$ means that column j of A is the k th 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 m th triplet $(Ti[m], Tj[m], Tx[m], Tz[m])$, replace $Ti[m]$ with $Pinv[Ti[m]]$ and replace $Tj[m]$ with $Qinv[Tj[m]]$.

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

Constructing a submatrix is also easy. Just scan the triplets and remove those entries outside the desired subset of $0 \dots n_{\text{row}}-1$ and $0 \dots n_{\text{col}}-1$, and renumber the indices according to their position in the subset.

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

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

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

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

Returns:

UMFPACK_OK if successful.
 UMFPACK_ERROR_argument_missing if A_p , A_i , T_i , and/or T_j are missing.
 UMFPACK_ERROR_n_nonpositive if $n_{\text{row}} \leq 0$ or $n_{\text{col}} \leq 0$.
 UMFPACK_ERROR_invalid_matrix if $n_z < 0$, or if for any k , $T_i[k]$ and/or $T_j[k]$ are not in the range 0 to $n_{\text{row}}-1$ or 0 to $n_{\text{col}}-1$, respectively.
 UMFPACK_ERROR_out_of_memory if unable to allocate sufficient workspace.

Arguments:

Int n_{row} ; Input argument, not modified.
 Int n_{col} ; Input argument, not modified.

A is an n_{row} -by- n_{col} matrix. Restriction: $n_{\text{row}} > 0$ and $n_{\text{col}} > 0$.
 All row and column indices in the triplet form must be in the range 0 to $n_{\text{row}}-1$ and 0 to $n_{\text{col}}-1$, respectively.

Int n_z ; Input argument, not modified.

The number of entries in the triplet form of the matrix. Restriction:
 $n_z \geq 0$.

Int $T_i[n_z]$; Input argument, not modified.
 Int $T_j[n_z]$; Input argument, not modified.
 double $T_x[n_z]$; Input argument, not modified.
 double $T_z[n_z]$; Input argument, not modified, for complex versions.

T_i , T_j , T_x , and T_z hold the "triplet" form of a sparse matrix. The k th nonzero entry is in row $i = T_i[k]$, column $j = T_j[k]$, and the real part of a_{ij} is $T_x[k]$. The imaginary part of a_{ij} is $T_z[k]$, for complex versions. The row and column indices i and j must be in the range 0 to $n_{\text{row}}-1$ and 0 to $n_{\text{col}}-1$, respectively. Duplicate entries may be present; they are summed in the output matrix. This is not an error condition. The "triplets" may be in any order. T_x , T_z , A_x , and A_z

are optional. For the real version, Ax is computed only if both Tx and Tz are present (not (double *) NULL). For the complex version, Ax and Az are computed only if Tx, Tz, Ax, and Az are all present. These are not error conditions; the routine can create just the pattern of the output matrix from the pattern of the triplets.

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

Int Ap [n_col+1] ; Output argument.

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

Int Ai [nz] ; Output argument.

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

The nonzero pattern (row indices) for column j is stored in Ai [(Ap [j]) ... (Ap [j+1]-1)]. The row indices in a given column j are in ascending order, and no duplicate row indices are present. Row indices are in the range 0 to n_col-1 (the matrix is 0-based).

double Ax [nz] ; Output argument.

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

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

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

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

int Map [nz] ; Optional output argument.

If Map is present (a non-NULL pointer to an Int array of size nz), then on output it holds the position of the triplets in the column-form matrix. That is, suppose p = Map [k], and the k-th triplet is i=Ti[k], j=Tj[k], and aij=Tx[k]. Then i=Ai[p], and aij will have been summed into Ax[p] (or simply aij=Ax[p] if there were no duplicate entries also in row i and column j). Also, Ap[j] <= p < Ap[j+1]. The Map array is

not computed if it is (Int *) NULL. The Map array is useful for converting a subsequent triplet form matrix with the same pattern as the first one, without calling this routine. If Ti and Tj do not change, then Ap, and Ai can be reused from the prior call to umfpack*_triplet_to_col. You only need to recompute Ax (and Az for the complex version). This code excerpt properly sums up all duplicate values (for the real version):

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

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

17.3 umfpack*_transpose

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

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

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

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

```

    long do_conjugate
) ;

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

```

Purpose:

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

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

Returns:

`UMFPACK_OK` if successful.
`UMFPACK_ERROR_out_of_memory` if `umfpack*_transpose` fails to allocate a `size-max (n_row,n_col)` workspace.
`UMFPACK_ERROR_argument_missing` if `Ai`, `Ap`, `Ri`, and/or `Rp` are missing.

UMFPACK_ERROR_n_nonpositive if $n_row \leq 0$ or $n_col \leq 0$
 UMFPACK_ERROR_invalid_permutation if P and/or Q are invalid.
 UMFPACK_ERROR_invalid_matrix if $Ap[n_col] < 0$, if $Ap[0] \neq 0$,
 if $Ap[j] > Ap[j+1]$ for any j in the range 0 to n_col-1 ,
 if any row index i is < 0 or $\geq n_row$, or if the row indices
 in any column are not in ascending order.

Arguments:

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

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

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

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

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

The nonzero pattern (row indices) for column j is stored in $Ai[(Ap[j]) \dots (Ap[j+1]-1)]$. The row indices in a given column j must be in ascending order, and no duplicate row indices may be present. Row indices must be in the range 0 to n_row-1 (the matrix is 0-based).

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

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

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

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

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

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

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

Int Rp [n_row+1] ; Output argument.

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

Int Ri [nz] ; Output argument.

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

double Rx [nz] ; Output argument.

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

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

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

Int do_conjugate ; Input argument for complex versions only.

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

17.4 umfpack*_scale

```
int umfpack_di_scale
(
    double X [ ],
    const double B [ ],
    void *Numeric
) ;

long umfpack_dl_scale
(
    double X [ ],
    const double B [ ],
    void *Numeric
) ;

int umfpack_zi_scale
(
    double Xx [ ],      double Xz [ ],
    const double Bx [ ], const double Bz [ ],
    void *Numeric
) ;

long umfpack_zl_scale
(
    double Xx [ ],      double Xz [ ],
    const double Bx [ ], const double Bz [ ],
    void *Numeric
) ;

double int Syntax:

    #include "umfpack.h"
    void *Numeric ;
    double *B, *X ;
    status = umfpack_di_scale (X, B, Numeric) ;

double long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    double *B, *X ;
    status = umfpack_dl_scale (X, B, Numeric) ;

complex int Syntax:

    #include "umfpack.h"
    void *Numeric ;
    double *Bx, *Bz, *Xx, *Xz ;
    status = umfpack_zi_scale (Xx, Xz, Bx, Bz, Numeric) ;

complex long Syntax:

    #include "umfpack.h"
    void *Numeric ;
    double *Bx, *Bz, *Xx, *Xz ;
    status = umfpack_zl_scale (Xx, Xz, Bx, Bz, Numeric) ;
```

Purpose:

Given LU factors computed by `umfpack*_numeric` (`PAQ=LU`, `PRAQ=LU`, or `P(R\A)Q=LU`), and a vector `B`, this routine computes $X = B$, $X = R*B$, or $X = R\backslash B$, as appropriate. `X` and `B` must be vectors equal in length to the number of rows of `A`.

Returns:

The status code is returned. `UMFPACK_OK` is returned if successful. `UMFPACK_ERROR_invalid_Numeric_object` is returned in the `Numeric` object is invalid. `UMFPACK_ERROR_argument_missing` is returned if any of the input vectors are missing (`X` and `B` for the real version, and `Xx`, `Xz`, `Bx`, and `Bz` for the complex version).

Arguments:

`double X [n_row] ;` Output argument.
or:
`double Xx [n_row] ;` Output argument, real part.
`double Xz [n_row] ;` Output argument, imaginary part.

The output vector `X`.

`double B [n_row] ;` Input argument, not modified.
or:
`double Bx [n_row] ;` Input argument, not modified, real part.
`double Bz [n_row] ;` Input argument, not modified, imaginary part.

The input vector `B`.

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

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

18 Getting the contents of opaque objects

18.1 umfpack_*_get_lunz

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:

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

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

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

Returns:

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

Arguments:

Int *lnz ; Output argument.

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

Int *unz ; Output argument.

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

Int *n_row ; Output argument.

Int *n_col ; Output argument.

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

Int *nz_udia ; Output argument.

The number of numerically nonzero values on the diagonal of U. The matrix is singular if nz_diag < min(n_row,n_col). A divide-by-zero

will occur if `nz_diag < n_row == n_col` when solving a sparse system involving the matrix `U` in `umfpack_*_solve`. The value of `nz_udia` is identical to `Info [UMFPACK_UDIAG_NZ]` if that value was returned by `umfpack_*_numeric`.

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

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

18.2 umfpack_*_get_numeric

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

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

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

long umfpack_zl_get_numeric
(
    long Lp [ ],
    long Lj [ ],
    double Lx [ ], double Lz [ ],
    long Up [ ],
    long Ui [ ],
```

```

double Ux [ ], double Uz [ ],
long P [ ],
long Q [ ],
double Dx [ ], double Dz [ ],
long *do_recip,
double Rs [ ],
void *Numeric
) ;

```

double int Syntax:

```

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

```

double long Syntax:

```

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

```

complex int Syntax:

```

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

```

complex long Syntax:

```

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

```

Purpose:

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

All output arguments are optional. If any of them are NULL on input, then that part of the LU factorization is not copied. You can use this routine to extract just the parts of the LU factorization that you want. For example, to retrieve just the column permutation Q, use:

```

#define noD (double *) NULL
#define noI (int *) NULL
status = umfpack_di_get_numeric (noI, noI, noD, noI, noI, noD, noI,
                                Q, noD, noI, noD, Numeric) ;

```

Returns:

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

Arguments:

```

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

```

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

```

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

```

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

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

```

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

```

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

```

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

```

respectively. Each column is stored in sorted order, from low row indices to higher. The last entry in each column is the diagonal (assuming that it is nonzero). The sizes of Up, Ui, Ux, and Uz are returned by umfpack*_get_lunz. If Up, Ui, or Ux (or Uz for the complex version) are not present, then the matrix U is not returned. This is

not an error condition. The U matrix can be printed if `n_col`, `Up`, `Ui`, `Ux` (and `Uz` for the complex versions) are passed to `umfpack*_report_matrix` (using the "column" form).

Future complex version: if `Ux` is present and `Uz` is NULL, then both real and imaginary parts will be returned in `Ux[0..2*unz-1]`, with `Ux[2*k]` and `Ux[2*k+1]` being the real and imaginary part of the *k*th entry.

`Int P [n_row] ;` Output argument.

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

`Int Q [n_col] ;` Output argument.

The permutation vector `Q` is defined as `Q [k] = j`, where the original column *j* of `A` is the *k*th pivot column in `PAQ`. If you do 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 `Qtrees`, the column pre-ordering held in the Symbolic object. Refer to the description of `Qtrees` and `Front_npivotcol` in `umfpack*_get_symbolic` for details.

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

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

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

`Int *do_recip ;` Output argument.

If `do_recip` is returned as zero (false), then the scale factors `Rs [i]` are to be used by multiplying row *i* by `Rs [i]`. Otherwise, the entries in row *i* are to be divided by `Rs [i]`.

If UMFPACK has been compiled with gcc, or for MATLAB as either a built-in routine or as a mexFunction, then the `NRECIPROCAL` flag is set, and `do_recip` will always be zero (false).

NOTE: this argument is new to version 4.1.

`double Rs [n_row] ;` Output argument.

The row scale factors are returned in `Rs [0..n_row-1]`. Row *i* of `A` is scaled by dividing or multiplying its values by `Rs [i]`. If default scaling is in use, `Rs [i]` is the sum of the absolute values of row *i* (or its reciprocal). If max row scaling is in use, then `Rs [i]` is the maximum absolute value in row *i* (or its reciprocal). Otherwise, `Rs [i] = 1`. If row *i* is all zero, `Rs [i] = 1` as well. For the complex version, an approximate absolute value is used

$(|x_{\text{real}}| + |x_{\text{imag}}|)$.

NOTE: this argument is new to version 4.1.

void *Numeric ; Input argument, not modified.

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

18.3 umfpack_*_get_symbolic

```
int umfpack_di_get_symbolic
(
    int *n_row,
    int *n_col,
    int *n1,
    int *nz,
    int *nfr,
    int *nchains,
    int P [ ],
    int Q [ ],
    int Front_npivcol [ ],
    int Front_parent [ ],
    int Front_1strow [ ],
    int Front_leftmostdesc [ ],
    int Chain_start [ ],
    int Chain_maxrows [ ],
    int Chain_maxcols [ ],
    void *Symbolic
) ;

long umfpack_dl_get_symbolic
(
    long *n_row,
    long *n_col,
    long *n1,
    long *nz,
    long *nfr,
    long *nchains,
    long P [ ],
    long Q [ ],
    long Front_npivcol [ ],
    long Front_parent [ ],
    long Front_1strow [ ],
    long Front_leftmostdesc [ ],
    long Chain_start [ ],
    long Chain_maxrows [ ],
    long Chain_maxcols [ ],
    void *Symbolic
) ;

int umfpack_zi_get_symbolic
(
    int *n_row,
    int *n_col,
    int *n1,
    int *nz,
    int *nfr,
    int *nchains,
    int P [ ],
    int Q [ ],
    int Front_npivcol [ ],
    int Front_parent [ ],
    int Front_1strow [ ],
    int Front_leftmostdesc [ ],
    int Chain_start [ ],
```

```

    int Chain_maxrows [ ],
    int Chain_maxcols [ ],
    void *Symbolic
) ;

long umfpack_zl_get_symbolic
(
    long *n_row,
    long *n_col,
    long *n1,
    long *nz,
    long *nfr,
    long *nchains,
    long P [ ],
    long Q [ ],
    long Front_npivcol [ ],
    long Front_parent [ ],
    long Front_1strow [ ],
    long Front_leftmostdesc [ ],
    long Chain_start [ ],
    long Chain_maxrows [ ],
    long Chain_maxcols [ ],
    void *Symbolic
) ;

```

double int Syntax:

```

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

```

double long Syntax:

```

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

```

complex int Syntax:

```

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

```


complex long Syntax:

Purpose:

All output arguments are optional. If any of them are NULL on input, then that part of the symbolic analysis is not copied. You can use this routine to extract just the parts of the symbolic analysis that you want. For example, to retrieve just the column permutation Q, use:

The only required argument the last one, the pointer to the Symbolic object.

The Symbolic object is small. Its size for an n -by- n square matrix varies from $4*n$ to $13*n$, depending on the matrix. 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 UMPACK_OK if successful, UMPACK_ERROR_invalid_Symbolic_object if Symbolic is an invalid object.

Arguments:

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

The dimensions of the matrix A analyzed by the call to `umfpack*_symbolic` that generated the Symbolic object.

```
Int *n1 ;           Output argument.
```

The number of pivots with zero Markowitz cost (they have just one entry in the pivot row, or the pivot column, or both). These appear first in the output permutations P and Q.

NOTE: this argument is new to version 4.1.

Int *nz ; Output argument.

The number of nonzeros in A.

Int *nfr ; Output argument.

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

Int *nchains ; Output argument.

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

Int P [n_row] ; Output argument.

The initial row permutation. If $P[k] = i$, then this means that row i is the k th row in the pre-ordered matrix. In general, this P is not the same as the final row permutation computed by umfpack*_numeric.

For the unsymmetric strategy, P defines the row-merge order. Let j be the column index of the leftmost nonzero entry in row i of $A*Q$. Then P defines a sort of the rows according to this value. A row can appear earlier in this ordering if it is aggressively absorbed before it can become a pivot row. If $P[k] = i$, row i typically will not be the k th pivot row.

For the symmetric strategy, $P = Q$. For the 2-by-2 strategy, P is the row permutation that places large entries on the diagonal of $P*A*Q$. If no pivoting occurs during numerical factorization, $P[k] = i$ also defines the final permutation of umfpack*_numeric, for either the symmetric or 2-by-2 strategies.

Int Q [n_col] ; Output argument.

The initial column permutation. If $Q[k] = j$, then this means that column j is the k th pivot column in the pre-ordered matrix. Q 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. If the matrix is structurally singular, and if the symmetric or 2-by-2 strategies are used (or if Control [UMFPACK_FIXQ] > 0), then this Q will be the same as the final column permutation computed in umfpack*_numeric.

Int Front_npivcol [n_col+1] ; Output argument.

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

The k th frontal matrix holds Front_npivcol [k] pivot columns. Thus, the first frontal matrix, front 0, is used to factorize the first

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

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

Int Front_parent [n_col+1] ; Output argument.

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

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

Int Front_1strow [n_col+1] ; Output argument.

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

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

Int Front_leftmostdesc [n_col+1] ; Output argument.

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

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

Chain_start [n_col+1] ; Output argument.

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

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

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

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

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

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

void *Symbolic ; Input argument, not modified.

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

18.4 umfpack_*_save_numeric

```
int umfpack_di_save_numeric
(
    void *Numeric,
    char *filename
) ;
```

```
long umfpack_dl_save_numeric
(
    void *Numeric,
    char *filename
) ;
```

```
int umfpack_zi_save_numeric
(
    void *Numeric,
    char *filename
) ;
```

```
long umfpack_zl_save_numeric
(
    void *Numeric,
    char *filename
) ;
```

double int Syntax:

```
#include "umfpack.h"
int status ;
char *filename ;
void *Numeric ;
status = umfpack_di_save_numeric (Numeric, filename) ;
```

double long Syntax:

```
#include "umfpack.h"
long status ;
char *filename ;
void *Numeric ;
status = umfpack_dl_save_numeric (Numeric, filename) ;
```

complex int Syntax:

```
#include "umfpack.h"
int status ;
char *filename ;
void *Numeric ;
status = umfpack_zi_save_numeric (Numeric, filename) ;
```

complex long Syntax:

```
#include "umfpack.h"
long status ;
char *filename ;
void *Numeric ;
status = umfpack_zl_save_numeric (Numeric, filename) ;
```

Purpose:

Saves a Numeric object to a file, which can later be read by `umfpack*_load_numeric`. The Numeric object is not modified. You need to call `umfpack*_free_numeric` if you to delete the Numeric object after saving it to a file.

Returns:

UMFPACK_OK if successful.
UMFPACK_ERROR_invalid_Numeric_object if Numeric is not valid.
UMFPACK_ERROR_file_IO if an I/O error occurred.

Arguments:

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

Numeric must point to a valid Numeric object, computed by `umfpack*_numeric` or loaded by `umfpack*_load_numeric`.

`char *filename ;` Input argument, not modified.

A string that contains the filename to which the Numeric object is written.

18.5 umfpack_*_load_numeric

```
int umfpack_di_load_numeric
(
    void **Numeric,
    char *filename
) ;
```

```
long umfpack_dl_load_numeric
(
    void **Numeric,
    char *filename
) ;
```

```
int umfpack_zi_load_numeric
(
    void **Numeric,
    char *filename
) ;
```

```
long umfpack_zl_load_numeric
(
    void **Numeric,
    char *filename
) ;
```

double int Syntax:

```
#include "umfpack.h"
int status ;
char *filename ;
void *Numeric ;
status = umfpack_di_load_numeric (&Numeric, filename) ;
```

double long Syntax:

```
#include "umfpack.h"
long status ;
char *filename ;
void *Numeric ;
status = umfpack_dl_load_numeric (&Numeric, filename) ;
```

complex int Syntax:

```
#include "umfpack.h"
int status ;
char *filename ;
void *Numeric ;
status = umfpack_zi_load_numeric (&Numeric, filename) ;
```

complex long Syntax:

```
#include "umfpack.h"
long status ;
char *filename ;
void *Numeric ;
status = umfpack_zl_load_numeric (&Numeric, filename) ;
```

Purpose:

Loads a Numeric object from a file created by `umfpack*_save_numeric`. The Numeric handle passed to this routine is overwritten with the new object. If that object exists prior to calling this routine, a memory leak will occur. The contents of Numeric are ignored on input.

Returns:

UMFPACK_OK if successful.
UMFPACK_ERROR_out_of_memory if not enough memory is available.
UMFPACK_ERROR_file_IO if an I/O error occurred.

Arguments:

`void **Numeric ;` Output argument.

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

`char *filename ;` Input argument, not modified.

A string that contains the filename from which to read the Numeric object.

18.6 umfpack_*_save_symbolic

```
int umfpack_di_save_symbolic
(
    void *Symbolic,
    char *filename
) ;
```

```
long umfpack_dl_save_symbolic
(
    void *Symbolic,
    char *filename
) ;
```

```
int umfpack_zi_save_symbolic
(
    void *Symbolic,
    char *filename
) ;
```

```
long umfpack_zl_save_symbolic
(
    void *Symbolic,
    char *filename
) ;
```

double int Syntax:

```
#include "umfpack.h"
int status ;
char *filename ;
void *Symbolic ;
status = umfpack_di_save_symbolic (Symbolic, filename) ;
```

double long Syntax:

```
#include "umfpack.h"
long status ;
char *filename ;
void *Symbolic ;
status = umfpack_dl_save_symbolic (Symbolic, filename) ;
```

complex int Syntax:

```
#include "umfpack.h"
int status ;
char *filename ;
void *Symbolic ;
status = umfpack_zi_save_symbolic (Symbolic, filename) ;
```

complex long Syntax:

```
#include "umfpack.h"
long status ;
char *filename ;
void *Symbolic ;
status = umfpack_zl_save_symbolic (Symbolic, filename) ;
```

Purpose:

Saves a Symbolic object to a file, which can later be read by `umfpack*_load_symbolic`. The Symbolic object is not modified. You need to call `umfpack*_free_symbolic` if you to delete the Symbolic object after saving it to a file.

Returns:

UMFPACK_OK if successful.
UMFPACK_ERROR_invalid_Symbolic_object if Symbolic is not valid.
UMFPACK_ERROR_file_IO if an I/O error occurred.

Arguments:

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

Symbolic must point to a valid Symbolic object, computed by `umfpack*_symbolic` or loaded by `umfpack*_load_symbolic`.

`char *filename ;` Input argument, not modified.

A string that contains the filename to which the Symbolic object is written.

18.7 umfpack_*_load_symbolic

```
int umfpack_di_load_symbolic
(
    void **Symbolic,
    char *filename
) ;
```

```
long umfpack_dl_load_symbolic
(
    void **Symbolic,
    char *filename
) ;
```

```
int umfpack_zi_load_symbolic
(
    void **Symbolic,
    char *filename
) ;
```

```
long umfpack_zl_load_symbolic
(
    void **Symbolic,
    char *filename
) ;
```

double int Syntax:

```
#include "umfpack.h"
int status ;
char *filename ;
void *Symbolic ;
status = umfpack_di_load_symbolic (&Symbolic, filename) ;
```

double long Syntax:

```
#include "umfpack.h"
long status ;
char *filename ;
void *Symbolic ;
status = umfpack_dl_load_symbolic (&Symbolic, filename) ;
```

complex int Syntax:

```
#include "umfpack.h"
int status ;
char *filename ;
void *Symbolic ;
status = umfpack_zi_load_symbolic (&Symbolic, filename) ;
```

complex long Syntax:

```
#include "umfpack.h"
long status ;
char *filename ;
void *Symbolic ;
status = umfpack_zl_load_symbolic (&Symbolic, filename) ;
```

Purpose:

Loads a Symbolic object from a file created by `umfpack*_save_symbolic`. The Symbolic handle passed to this routine is overwritten with the new object. If that object exists prior to calling this routine, a memory leak will occur. The contents of Symbolic are ignored on input.

Returns:

UMFPACK_OK if successful.
UMFPACK_ERROR_out_of_memory if not enough memory is available.
UMFPACK_ERROR_file_IO if an I/O error occurred.

Arguments:

`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.

`char *filename ;` Input argument, not modified.

A string that contains the filename from which to read the Symbolic object.

19 Reporting routines

19.1 umfpack_*_report_status

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:

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

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

Control [UMFPACK_PRL]: printing level.

Int status ;	Input argument, not modified.
--------------	-------------------------------

19.2 umfpack_*_report_control

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

double long Syntax:

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

Purpose:

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

Arguments:

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

If a (double *) NULL pointer is passed, then the default control settings are used. Otherwise, the settings are determined from the

Control array. See `umfpack*_defaults` on how to fill the Control array with the default settings. If Control contains NaN's, the defaults are used. The following Control parameters are used:

Control [UMFPACK_PRL]: printing level.

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

19.3 umfpack_*_report_info

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:

Reports statistics from the umfpack_*_symbolic, umfpack_*_numeric, and umfpack_*_solve routines.

Arguments:

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

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

Control [UMFPACK_PRL]: printing level.

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

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

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

19.4 umfpack_*_report_matrix

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

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

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

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

double int Syntax:

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

double long Syntax:
```

```

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

```

complex int Syntax:

```

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

```

complex long Syntax:

```

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

```

Purpose:

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

Returns:

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

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

UMFPACK_OK if the matrix is valid.

UMFPACK_ERROR_n_nonpositive if n_row <= 0 or n_col <= 0.

UMFPACK_ERROR_argument_missing if Ap and/or Ai are missing.

UMFPACK_ERROR_invalid_matrix if Ap [n] < 0, if Ap [0] is not zero,
if Ap [j+1] < Ap [j] for any j in the range 0 to n-1,
if any row index in Ai is not in the range 0 to ni-1, or
if the row indices in any column are not in
ascending order, or contain duplicates.

UMFPACK_ERROR_out_of_memory if out of memory.

Arguments:

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

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

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

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

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

The first entry, `Ap [0]`, must be zero, and `Ap [j] <= Ap [j+1]` must hold for all `j` in the range 0 to `n-1`. The value `nz = Ap [n]` is thus the total number of entries in the pattern of the matrix `A`.

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

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

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

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

The numerical values of the sparse matrix `A`.

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

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

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

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

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

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

`Int col_form ; Input argument, not modified.`

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

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

If a (double *) NULL pointer is passed, then the default control

settings are used. Otherwise, the settings are determined from the Control array. See `umfpack*_defaults` on how to fill the Control array with the default settings. If Control contains NaN's, the defaults are used. The following Control parameters are used:

Control [UMFPACK_PRL]: printing level.

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

Default: 1

19.5 umfpack_*_report_numeric

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:

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

Returns:

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

Otherwise:

UMFPACK_OK if the Numeric object is valid.

UMFPACK_ERROR_invalid_Numeric_object if the Numeric object is invalid.

UMFPACK_ERROR_out_of_memory if out of memory.

Arguments:

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

The Numeric object, which holds the numeric factorization computed by `umfpack_*_numeric`.

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

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

```
Control [UMFPACK_PRL]:  printing level.
```

2 or less: no output. returns silently without checking anything.

3: fully check input, and print a short summary of its status

4: as 3, but print first few entries of the input

```
5: as 3, but print all of the input
```

Default: 1

19.6 umfpack_*_report_perm

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:

Verifies and prints a permutation vector.

Returns:

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

Otherwise:

UMFPACK_OK if the permutation vector is valid (this includes that case when Perm is (Int *) NULL, which is not an error condition).

UMFPACK_ERROR_n_nonpositive if $np \leq 0$.

UMFPACK_ERROR_out_of_memory if out of memory.

UMFPACK_ERROR_invalid_permutation if Perm is not a valid permutation vector.

Arguments:

Int np ; Input argument, not modified.

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

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

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

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

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

Control [UMFPACK_PRL]: printing level.

2 or less: no output. returns silently without checking anything.

3: fully check input, and print a short summary of its status

4: as 3, but print first few entries of the input

5: as 3, but print all of the input

Default: 1

19.7 umfpack_*_report_symbolic

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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

Purpose:

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

Returns:

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

Otherwise:

UMFPACK_OK if the Symbolic object is valid.

UMFPACK_ERROR_invalid_Symbolic_object if the Symbolic object is invalid.

UMFPACK_ERROR_out_of_memory if out of memory.

Arguments:

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

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

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

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

Control [UMFPACK_PRL]: printing level.

2 or less: no output. returns silently without checking anything.

3: fully check input, and print a short summary of its status

4: as 3, but print first few entries of the input

5: as 3, but print all of the input

Default: 1

19.8 umfpack_*_report_triplet

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

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

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

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

double int Syntax:

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

double long Syntax:

#include "umfpack.h"
long n_row, n_col, nz, *Ti, *Tj, status ;
```

```
double *Tx, Control [UMFPACK_CONTROL] ;
status = umfpack_dl_report_triplet (n_row, n_col, nz, Ti, Tj, Tx, Control) ;
```

complex int Syntax:

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

complex long Syntax:

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

Purpose:

Verifies and prints a matrix in triplet form.

Returns:

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

Otherwise:

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

Arguments:

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

A is an n_row-by-n_col matrix.

Int nz ; Input argument, not modified.

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

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

Ti, Tj, Tx (and Tz for complex versions) hold the "triplet" form of a sparse matrix. The kth nonzero entry is in row i = Ti [k], column j = Tj [k], the real numerical value of a_{ij} is Tx [k], and the imaginary part of a_{ij} is Tz [k] (for complex versions). The row and column indices i and j must be in the range 0 to n_row-1 or 0 to n_col-1, respectively. Duplicate entries may be present. The "triplets" may be in any order. Tx and Tz are optional; if Tx or Tz are

not present ((double *) NULL pointers), then the numerical values are not printed.

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

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

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

Control [UMFPACK_PRL]: printing level.

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

19.9 umfpack_*_report_vector

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

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

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

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

double int Syntax:

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

double long Syntax:

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

complex int Syntax:

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

complex long Syntax:

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


Purpose:

Verifies and prints a dense vector.

Returns:

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

Otherwise:

UMFPACK_OK if the vector is valid.

UMFPACK_ERROR_argument_missing if X or Xx is missing.

UMFPACK_ERROR_n_nonpositive if n <= 0.

Arguments:

Int n ; Input argument, not modified.

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

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

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

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

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

A complex vector of size n, in one of two storage formats.

Xx must not be (double *) NULL.

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

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

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

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

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

Control [UMFPACK_PRL]: printing level.

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

20 Utility routines

20.1 umfpack_timer

```
double umfpack_timer ( void ) ;
```

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

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

Purpose:

Returns the CPU time used by the process. Includes both "user" and "system" time (the latter is time spent by the system on behalf of the process, and is thus charged to the process). It does not return the wall clock time. This was the timer used internally in UMFPACK V4.0. See `umfpack_tic` and `umfpack_toc` (the file `umfpack_tictoc.h`) for the timer used internally by UMFPACK V4.1.

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

Arguments:

None.

20.2 umfpack_tic and umfpack_toc

```
void umfpack_tic (double stats [2]) ;
```

```
void umfpack_toc (double stats [2]) ;
```

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

```
#include "umfpack.h"
double stats [2] ;
umfpack_tic (stats) ;
...
umfpack_toc (stats) ;
```

Purpose:

umfpack_tic returns the CPU time and wall clock time used by the process. The CPU time 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). umfpack_toc returns the CPU time and wall clock time since the last call to umfpack_tic with the same stats array.

Typical usage:

```
umfpack_tic (stats) ;
... do some work ...
umfpack_toc (stats) ;
```

then stats [1] contains the time in seconds used by the code between umfpack_tic and umfpack_toc, and stats [0] contains the wall clock time elapsed between the umfpack_tic and umfpack_toc. These two routines act just like tic and toc in MATLAB, except that the both process time and wall clock time are returned.

This routine normally uses the sysconf and times routines in the POSIX standard. If -DNPOSIX is defined at compile time, then the ANSI C clock routine is used instead, and only the CPU time is returned (stats [0] is set to zero).

umfpack_tic and umfpack_toc are the routines used internally in UMFPACK to time the symbolic analysis, numerical factorization, and the forward/backward solve.

Arguments:

```
double stats [2]:

stats [0]: wall clock time, in seconds
stats [1]: CPU time, in seconds
```

References

- [1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Applic.*, 17(4):886–905, 1996.
- [2] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 8xx: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 2003 (under submission). Also TR-03-010 at www.cise.ufl.edu/tech-reports.
- [3] 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.
- [4] T. A. Davis. Algorithm 8xx: UMFPACK V4.1, an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 2003 (under submission). Also TR-03-007 at www.cise.ufl.edu/tech-reports.
- [5] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 2003 (under submission). Also TR-03-006 at www.cise.ufl.edu/tech-reports.
- [6] 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.
- [7] 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.
- [8] 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.*).
- [9] 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.*).
- [10] T. A. Davis and W. W. Hager. Modifying a sparse Cholesky factorization. *SIAM J. Matrix Anal. Applic.*, 20(3):606–627, 1999.
- [11] 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.
- [12] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level-3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.
- [13] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Comm. ACM*, 30:403–407, 1987. www.netlib.org.
- [14] I. S. Duff. Algorithm 575: Permutations for a zero-free diagonal. *ACM Trans. Math. Softw.*, 7:387–390, 1981.
- [15] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users’ guide for the harwell-boeing sparse matrix test collection. Technical report, AERE Harwell Laboratory, United Kingdom Atomic Energy Authority, 1987.

- [16] I. S. Duff and J. K. Reid. Algorithm 529: Permutations to block triangular form. *ACM Trans. Math. Softw.*, 4(2):189–192, 1978.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.*, 9:862–874, 1988.
- [25] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication, FLAME working note 9. Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences, Nov. 2002.
- [26] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.
- [27] S. I. Larimore. An approximate minimum degree column ordering algorithm. Technical Report TR-98-016, Univ. of Florida, CISE Dept., Gainesville, FL, 1998. www.cise.ufl.edu/tech-reports.
- [28] 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.