UMFPACK User Guide

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

VERSION 5.7.4, Feb 1, 2016

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-04-003 (revised)

Availability: http://www.suitesparse.com

Acknowledgments:

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

UMFPACK[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 two strategies for pre-ordering the rows and columns: *unsymmetric* 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. Let d be the number of nonzero entries on the diagonal of $\mathbf{S}$. Let $\mathbf{S}$ be ν-by-ν. Rule 3: $(\sigma_1 \geq 0.5) \wedge (d \geq 0.9\nu) \rightarrow$ symmetric. The matrix has a nearly symmetric nonzero pattern (50% or more), and a mostly-zero-free diagonal (90% or more nonzero).
- Rule 4: Otherwise, the unsymmetric strategy is used.

Each strategy is described below:

- *unsymmetric*: The column pre-ordering of $\mathbf{S}$ is computed by a modified version of COLAMD [8, 9]. The method finds a symmetric permutation $\mathbf{Q}$ of the matrix $\mathbf{S}^\mathsf{T}\mathbf{S}$ (without forming $\mathbf{S}^\mathsf{T}\mathbf{S}$ explicitly). This is a good choice for $\mathbf{Q}$, since the Cholesky factors of $(\mathbf{SQ})^\mathsf{T}(\mathbf{SQ})$ are an upper bound (in terms of nonzero pattern) of the factor $\mathbf{U}$ for the unsymmetric LU factorization ($\mathbf{PSQ} = \mathbf{LU}$) 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 L and 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

[1] Pronounced with two syllables: umph-pack

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.

- *symmetric*: The column ordering is computed from AMD [1, 2], applied to the pattern of $\mathbf{S} + \mathbf{S}^\mathsf{T}$ followed by a post-ordering of the supernodal elimination tree of $\mathbf{S} + \mathbf{S}^\mathsf{T}$. 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 strategy, and their automatic selection, are new to Version 4.1. Version 4.0 only used the unsymmetric strategy. Versions 4.1 through 5.3 included a *2-by-2* ordering strategy, but this option has been disabled in Version 5.4.

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 strategy 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 [5, 4], available at http://www.suitesparse.com.

2 Availability

In addition to appearing as a Collected Algorithm of the ACM, UMFPACK is available at http://www.suitesparse.com. It is included as a built-in routine in MATLAB. Version 4.0 (in MATLAB 6.5) does not have the symmetric strategy and it takes less advantage of the level-3 BLAS [11, 12, 27, 25]. Versions 5.x through v4.1 tend 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.suitesparse.com. 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 as MA38 (functionally equivalent to Version 2.2.1) in the Harwell Subroutine Library.

NOTE: you must use the correct version of AMD with UMFPACK; using an old version of AMD with a newer version of UMFPACK can fail.

3 Primary changes from prior versions

A detailed list of changes is in the `ChangeLog` file.

3.1 Version 5.7.3

Renamed the MATLAB interface back to `umfpack` from `umpfack2`, since it no longer conflicts with any built-in MATLAB function of the same name. See the comments for Version 5.0.3 below.

3.2 Version 5.7.0

Replaced memory manager functions and printf with functions in `SuiteSparse_config`.

3.3 Version 5.6.0

Replaced `UFconfig` with `SuiteSparse_config`, for `SuiteSparse_timer`. The `UF_long` type is replaced with `SuiteSparse_long` to avoid potential name conflicts with the `UF_` prefix, but the former is still available for user programs. User programs may safely `#undef` the `UF_long` macro and use `SuiteSparse_long` instead. In future versions, `UF_long` will be removed completely from `SuiteSparse_config.h`.

3.4 Version 5.5.0

Added more user ordering options (interface to CHOLMOD and METIS orderings, and the ability to pass in a user ordering function). Minor change to how the 64-bit BLAS is used. Added an option to disable the search for singletons.

3.5 Version 5.4.0

Disabled the 2-by-2 ordering strategy. Bug fix for `umfpack_make.m` for Windows.

3.6 Version 5.3.0

A bug fix for structurally singular matrices, and a compiler workaround for gcc versions 4.2.(3 and 4).

3.7 Version 5.2.0

Change of license from GNU Lesser GPL to the GNU GPL.

3.8 Version 5.1.0

Port of MATLAB interface to 64-bit MATLAB.

3.9 Version 5.0.3

Renamed the MATLAB function to `umfpack2`, so as not to confict with itself (the MATLAB built-in version of UMFPACK).

3.10 Version 5.0

Changed `long` to `UF_long`, controlled by the `UFconfig.h` file. A `UF_long` is normally just `long`, except on the Windows 64 (WIN64) platform. In that case, it becomes `__int64`.

3.11 Version 4.6

Added additional options to `umf_solve.c`.

3.12 Version 4.5

Added function pointers for malloc, calloc, realloc, free, printf, hypot, and complex divisiion, so that these functions can be redefined at run-time. Added a version number so you can determine the version of UMFPACK at run time or compile time. UMFPACK requires AMD v2.0 or later.

3.13 Version 4.4

Bug fix in strategy selection in `umfpack_*_qsymbolic`. Added packed complex case for all complex input/output arguments. Added `umfpack_get_determinant`. Added minimal support for Microsoft Visual Studio (the `umf_multicompile.c` file).

3.14 Version 4.3.1

Minor bug fix in the forward/backsolve. This bug had the effect of turning off iterative refinement when solving $\mathbf{A}^\mathsf{T}\mathbf{x} = \mathbf{b}$ after factorizing $\mathbf{A}$. UMFPACK mexFunction now factorizes $\mathbf{A}^\mathsf{T}$ in its forward-slash operation.

3.15 Version 4.3

No changes are visible to the C or MATLAB user, except the presence of one new control parameter in the `Control` array, and three new statistics in the `Info` array. The primary change is the addition of an (optional) drop tolerance.

3.16 Version 4.1

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.

10. You can compute an incomplete LU factorization, by dropping small entries from **L** and **U**. By default, no nonzero entry is dropped, no matter how small in absolute value. This feature is new to Version 4.3.

Table 1: Using UMFPACK's MATLAB interface

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

Solve $\mathbf{Ax} = \mathbf{b}$ using a different row and column preordering (symmetric strategy).	`S = spones (A) ;`
	`Q = symamd (S+S') ;`
	`Control = umfpack ;`
	`Control.strategy = 'symmetric' ;`
	`x = umfpack (A,Q,'\',b,Control) ;`
		`spparms ('autommd',0) ;`
		`S = spones (A) ;`
		`Q = symamd (S+S') ;`
		`x = A (Q,Q) \ b (Q) ;`
		`x (Q) = x ;`
		`spparms ('autommd',1) ;`

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

Scale and factorize $\mathbf{A}$, then solve $\mathbf{Ax} = \mathbf{b}$.	`[L,U,P,Q,R] = umfpack (A) ;`
	`c = P * (R \ b) ;`
	`x = Q * (U \ (L \ c)) ;`
		`[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)) ;`

4 Using UMFPACK in MATLAB

The easiest way to use UMFPACK is within MATLAB. Version 4.3 is a built-in routine in MATLAB 7.0.4, 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 compile both the UMFPACK and AMD mexFunctions, just type `umfpack_make` in MATLAB, while in the `UMFPACK/MATLAB` directory.

See Section 8 for more details on how to install UMFPACK. Once installed, the UMFPACK mexFunction can analyze, factor, and solve linear systems. Table 1 summarizes some of the more common uses of 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.

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

Note: in MATLAB 6.5 or later, 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 `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 or later, which uses UMFPACK 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*A` is equal to `L*U`. UMFPACK behaves differently. By default, it scales the rows of `A` and reorders the columns of `A` prior to factorization, so that `L*U` is equal to `P*(R\A)*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 structural rank `m` and $m < n$, then `umfpack` might not find a factor `U` with a structurally zero-free diagonal. Unless the matrix ill-conditioned or poorly scaled, factorizing `A'` in this case will guarantee that both factors will have zero-free diagonals (in the structural sense; they may be numerically zero). Note that there is no guarantee as to the size of the diagonal entries of `U`; UMFPACK does not do a rank-revealing factorization. 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.

To compute the determinant with UMFPACK:

```
d = umfpack (A, 'det') ;
[d e] = umfpack (A, 'det') ;
```

The first case is identical to MATLAB's `det`. The second case returns the determinant in the form $d \times 10^e$, which avoids overflow if e is large.

5 Using UMFPACK in a C program

The C-callable UMFPACK library consists of 32 user-callable routines and one include file. All but three of the routines come in four versions, with different sizes of integers and for real or complex floating-point numbers:

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

where `*` denotes the specific name of one of the 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 `SuiteSparse_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 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/SuiteSparse_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 `SuiteSparse_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 the `SuiteSparse_config/SuiteSparse_config.mk` file.

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 ("packed" complex form).

New to Version 4.4 is the option of providing input/output arguments in packed complex form.

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 10:

- `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 **U** will have zeros on the diagonal if **A** is singular; this produces a warning, but the factorization is still valid.

- `umfpack_*_solve`:

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

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

The complex versions add another array for the imaginary part:

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

Alternatively, if `Az` is `NULL`, the real part of the kth entry is located in `Ax[2*k]` and the imaginary part is located in `Ax[2*k+1]`, and the `Ax` array is of size `2*nz`.

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

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`.

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 $\mathbf{x} = [1\,2\,3\,4\,5]^{\mathsf{T}}$. The program uses default control settings and does not return any statistics about the ordering, factorization, or solution (`Control` and `Info` are both `(double *) NULL`). 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 11:

- `umfpack_*_defaults`:

  Sets the default control parameters in the `Control` array. These can then be modified as desired before passing the array to the other UMFPACK routines. Control parameters are summarized in Section 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, than 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. The default is `UMFPACK_STRATEGY_AUTO`, in which UMFPACK selects the strategy by itself. `UMFPACK_STRATEGY_UNSYMMETRIC` gives the unsymmetric strategy, which is to use a column pre-ordering (such as COLAMD) and to give no preference to the diagonal during partial pivoting. `UMFPACK_STRATEGY_SYMMETRIC` gives the symmetric strategy, which is to use a symmetric row and column ordering (such as AMD) and to give strong preference to the diagonal during partial pivoting.

The parameter `Control [UMFPACK_ORDERING]` defines what ordering method UMFPACK should use. The options are:

- `UMFPACK_ORDERING_CHOLMOD` (0). This is the method used by CHOLMOD. It first tries AMD or COLAMD (depending on what strategy is used). If that method gives low fill-in, it is used without trying METIS at all. Otherwise METIS is tried (on $\mathbf{A}^T\mathbf{A}$ for the unsymmetric strategy, or $\mathbf{A} + \mathbf{A}^T$ for the symmetric strategy), and the ordering (AMD/COLAMD or METIS) giving the lowest fill-in is used.
- `UMFPACK_ORDERING_DEFAULT`. This is the same as `UMFPACK_ORDERING_AMD`.
- `UMFPACK_ORDERING_AMD` (1). This is the default. AMD is used for the symmetric strategy, on the pattern of $\mathbf{A} + \mathbf{A}^T$. COLAMD is used on $\mathbf{A}$ for the unsymmetric strategy.
- `UMFPACK_ORDERING_GIVEN` (2). This is assumed if a permutation is provided to `umfpack_*_qsymbolic`.
- `UMFPACK_ORDERING_METIS` (3). Use METIS (on $\mathbf{A}^T\mathbf{A}$ for the unsymmetric strategy, or $\mathbf{A} + \mathbf{A}^T$ for the symmetric strategy).

- `UMFPACK_ORDERING_BEST` (4). Try three methods and take the best. The three methods are AMD/COLAMD, METIS, and NESDIS (CHOLMOD's nested dissection ordering, based on METIS and CCAMD/CCOLAMD). This results the highest analysis time, but the lowest numerical factorization time.
- `UMFPACK_ORDERING_NONE` (5). The matrix is factorized as-is, except that singletons are still removed.
- `UMFPACK_ORDERING_USER` (6). Use the user-ordering function passed to `umfpack_*_fsymbolic`. Refer to `UMFPACK/Source/umf_cholmod.c` for an example.

To disable the singleton filter, set `Control [UMFPACK_SINGLETONS]` to 0. Disabling this search for singletons can slow UMFPACK down quite a bit for some matrices, but it does ensure that $\mathbf{L}$ is well-conditioned and that any ill-conditioning of $\mathbf{A}$ is captured only in $\mathbf{U}$.

- `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 $\mathbf{A}^{\mathsf{T}}\mathbf{A}$ would be suitable for UMFPACK's unsymmetric strategy. A partitioning of $\mathbf{A} + \mathbf{A}^{\mathsf{T}}$ would be suitable for UMFPACK's symmetric strategy.

- `umfpack_*_fsymbolic`:

  An alternative to `umfpack_*_symbolic`. Allows the user to pass a pointer to a function, which is called to compute the ordering on the matrix (or on a submatrix with singletons removed, if any exist).

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

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

For the `SuiteSparse_long` version:

```
SuiteSparse_long Ti [nz] ;
SuiteSparse_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 = `Ti[k]`, j = `Tj[k]`, and a_{ij} = `Tx[k]`. For the complex versions, `Tx[k]` is the real part of a_{ij} and `Tz[k]` is the imaginary part. The triplets can be in any order in the `Ti`, `Tj`, and `Tx` arrays (and `Tz` for the complex versions), and duplicate entries may exist. If `Tz` is NULL, then the array `Tx` becomes of size `2*nz`, and the real and imaginary parts of the k-th triplet are located in `Tx[2*k]` and `Tx[2*k+1]`, respectively. 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 `R = A(P,Q)'` (linear algebraic transpose, using the complex conjugate) or `R = A(P,Q).'` (the array transpose) in MATLAB, except for the presence of numerically zero entries.

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

- `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 $\mathbf{A}^\mathsf{T}\mathbf{x} = \mathbf{b}$, since `umfpack_*_solve` applies the scale factors for those systems.

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 12 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]. The CHOLMOD provides many of these matrix operations, which can then be used in conjunction with UMFPACK. See my web page for details.

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. An additional routine is provided that computes the determinant. They are fully described in Section 13:

- `umfpack_*_get_lunz`:

  Returns the number of nonzeros in **L** and **U**.

- `umfpack_*_get_numeric`:

  Copies **L**, **U**, **P**, **Q**, and **R** from the `Numeric` object into arrays provided by the user. The matrix **L** is returned in compressed row form (with the column indices in each row sorted in ascending order). The matrix **U** is returned in compressed column form (with sorted columns). There are no explicit zero entries in **L** and **U**, but such entries may exist in the `Numeric` object. The permutations **P** and **Q** are represented as permutation vectors, where `P[k] = i` means that row `i` of the original matrix is the the `k`-th row of **PAQ**, and where `Q[k] = j` means that column `j` of the original matrix is the `k`-th column of **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 preordering, supernodal column elimination tree, and information about each frontal matrix) into arrays provided by the user.

- `umfpack_*_get_determinant`:

  Computes the determinant from the diagonal of **U** and the permutations **P** and **Q**. This is mostly of theoretical interest. It is not a good test to determine if your matrix is singular or not.

- `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. None of them do any computation, except for `umfpack_*_get_determinant`.

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 14:

- `umfpack_*_report_status`:

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

- `umfpack_*_report_info`:

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

- `umfpack_*_report_control`:

  Prints the `Control` settings.

- `umfpack_*_report_matrix`:

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

- `umfpack_*_report_triplet`:

  Verifies and prints a matrix in triplet form.

- `umfpack_*_report_symbolic`:

  Verifies and prints a `Symbolic` object.

- `umfpack_*_report_numeric`:

  Verifies and prints a `Numeric` object.

- `umfpack_*_report_perm`:

  Verifies and prints a permutation vector.

- `umfpack_*_report_vector`:

  Verifies and prints a real or complex vector.

The `umfpack_*_report_*` routines behave slightly differently when compiled into the C-callable UMFPACK library than when used in the MATLAB mexFunction. MATLAB stores its sparse matrices using the same compressed column data structure discussed above, where row and column indices of an m-by-n matrix are in the range 0 to $m-1$ or $n-1$, respectively[2] It prints them as if they are in the range 1 to m or n. The UMFPACK mexFunction behaves the same way.

[2]Complex matrices in MATLAB use the split array form, with one `double` array for the real part and another array for the imaginary part. UMFPACK supports that format, as well as the packed complex format (new to Version 4.4).

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

Three timing routines are provided in UMFPACK Version 4.1 and following, umfpack_tic, umfpack_toc, and umfpack_timer. These three routines are the only user-callable routine that is identical in all four int/SuiteSparse_long, real/complex versions (there is no umfpack_di_timer routine, for example).

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. These defaults provide nearly optimal performance (both speed, memory usage, and numerical accuracy) for a wide range of matrices from real applications.

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 may increase in future versions, since all 20 entries are in use.

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

Table 2: UMFPACK Control parameters

MATLAB struct	ANSI C	default	description
prl	Control[UMFPACK_PRL]	1	printing level
-	Control[UMFPACK_DENSE_ROW]	0.2	dense row parameter
-	Control[UMFPACK_DENSE_COL]	0.2	dense column parameter
tol	Control[UMFPACK_PIVOT_TOLERANCE]	0.1	partial pivoting tolerance
-	Control[UMFPACK_BLOCK_SIZE]	32	BLAS block size
strategy	Control[UMFPACK_STRATEGY]	0 (auto)	select strategy
ordering	Control[UMFPACK_ORDERING]	1 (AMD)	select ordering
-	Control[UMFPACK_ALLOC_INIT]	0.7	initial memory allocation
irstep	Control[UMFPACK_IRSTEP]	2	max iter. refinement steps
-	Control[UMFPACK_FIXQ]	0 (auto)	fix or modify Q
-	Control[UMFPACK_AMD_DENSE]	10	AMD dense row/col param.
symtol	Control[UMFPACK_SYM_PIVOT_TOLERANCE]	0.001	for diagonal entries
scale	Control[UMFPACK_SCALE]	1 (sum)	row scaling (none, sum, max)
-	Control[UMFPACK_FRONT_ALLOC_INIT]	0.5	frontal matrix allocation ratio
-	Control[UMFPACK_DROPTOL]	0	drop tolerance
-	Control[UMFPACK_AGGRESSIVE]	1 (yes)	aggressive absorption
singletons	Control[UMFPACK_SINGLETONS]	1 (enable)	enable singleton filter

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 α_r =`Control [UMFPACK_DENSE_ROW]`, α_c =`Control [UMFPACK_DENSE_COL]`, and α =`Control [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 $\mathbf{S} + \mathbf{S}^\mathsf{T}$ 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_*_fsymbolic`, `umfpack_*_numeric`, `umfpack_*_solve`, and the `umfpack_*_report_*` routines, in Sections 10 through 14, 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. 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. 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_WARNING_determinant_underflow, (2): The determinant is nonzero, but smaller in magnitude than the smallest positive floating-point number.
- UMFPACK_WARNING_determinant_overflow, (3): The determinant is larger in magnitude than the largest positive floating-point number (IEEE Inf).
- 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, for example.[3]. 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]` $\leq$ `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 `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).

[3]Exception: v4.3, v4.3.1, and v4.4 use identical data structures for the `Numeric` and `Symbolic` objects

Some routines take a triplet-form input, with arguments `nz`, `Ti`, and `Tj`. This error code is returned if `nz` is less than zero, if any row index in `Ti` is outside the range 0 to `n_col-1`, or if any column index in `Tj` 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_ordering_failed`, (-18): The ordering method failed.
- `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/Include/amd_internal.h` and adding the statement `#undef NDEBUG`, 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 (DrTimothyAldenDavis@gmail.com) 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`, `umfpack_*_qsymbolic` or `umfpack_*_fsymbolic`. 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.suitesparse.com. 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)	$\mathbf{A}^{\mathsf{H}}\mathbf{x} = \mathbf{b}$
`UMFPACK_Aat`	(2)	$\mathbf{A}^{\mathsf{T}}\mathbf{x} = \mathbf{b}$
`UMFPACK_Pt_L`	(3)	$\mathbf{P}^{\mathsf{T}}\mathbf{Lx} = \mathbf{b}$
`UMFPACK_L`	(4)	$\mathbf{Lx} = \mathbf{b}$
`UMFPACK_Lt_P`	(5)	$\mathbf{L}^{\mathsf{H}}\mathbf{Px} = \mathbf{b}$
`UMFPACK_Lat_P`	(6)	$\mathbf{L}^{\mathsf{T}}\mathbf{Px} = \mathbf{b}$
`UMFPACK_Lt`	(7)	$\mathbf{L}^{\mathsf{H}}\mathbf{x} = \mathbf{b}$
`UMFPACK_Lat`	(8)	$\mathbf{L}^{\mathsf{T}}\mathbf{x} = \mathbf{b}$
`UMFPACK_U_Qt`	(9)	$\mathbf{UQ}^{\mathsf{T}}\mathbf{x} = \mathbf{b}$
`UMFPACK_U`	(10)	$\mathbf{Ux} = \mathbf{b}$
`UMFPACK_Q_Ut`	(11)	$\mathbf{QU}^{\mathsf{H}}\mathbf{x} = \mathbf{b}$
`UMFPACK_Q_Uat`	(12)	$\mathbf{QU}^{\mathsf{T}}\mathbf{x} = \mathbf{b}$
`UMFPACK_Ut`	(13)	$\mathbf{U}^{\mathsf{H}}\mathbf{x} = \mathbf{b}$
`UMFPACK_Uat`	(14)	$\mathbf{U}^{\mathsf{T}}\mathbf{x} = \mathbf{b}$

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, `SuiteSparse_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 `SuiteSparse_long`. Similarly, the complex, `SuiteSparse_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 `SuiteSparse_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. [4] Valid values are listed in Table 3. The notation $\mathbf{A}^{\mathsf{H}}$ refers to the matrix transpose, which is the complex conjugate transpose for complex matrices (`A'` in MATLAB). The array transpose is $\mathbf{A}^{\mathsf{T}}$, which is `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] ;
```

[4] Integer values for `sys` are used instead of strings (as in LINPACK and LAPACK) to avoid C-to-Fortran portability issues.

```
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_fsymbolic (m, n, Ap, Ai, Ax, &user_ordering, user_params, &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], Mx [1], Ex [1] ;
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) ;
status = umfapck_di_get_determinant (Mx, Ex, Numeric, Info) ;
```

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) ;
```

The arrays Ax, Bx, and Xx double in size if any imaginary argument (Az, Xz, or Bz) is NULL.

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_fsymbolic (m, n, Ap, Ai, Ax, Az, &user_ordering, user_params, &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) ;
```

The arrays Tx, Rx, Yx, and Zx double in size if any imaginary argument (Tz, Rz, Yz, or Zz) is NULL.

6.9 Getting the contents of opaque objects: complex/int

```
double Lz [lnz], Uz [unz], Dx [min (m,n)], Dz [min (m,n)], Mz [1] ;

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) ;
status = umfapck_zi_get_determinant (Mx, Mz, Ex, Numeric, Info) ;
```

The arrays `Lx`, `Ux`, `Dx`, and `Mx` double in size if any imaginary argument (`Lz`, `Uz`, `Dz`, or `Mz`) is `NULL`.

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) ;
```

The arrays `Ax`, `Rx`, `Tx`, and `Xx` double in size if any imaginary argument (`Az`, `Rz`, `Tz`, or `Xz`) is `NULL`.

6.11 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) ;
```

6.12 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; `SuiteSparse_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) ;
```

7 Using UMFPACK in a Fortran program

UMFPACK 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 8 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 10 through 15. 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_f77zwrapper.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)
```

The Fortran interface does not support the packed complex case.

8 Installation

8.1 Installing the C library

The following discussion assumes you have the `make` program, either in Unix, or in Windows with Cygwin[5]. 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 and AMD to use UMFPACK. The `UMFPACK` and `AMD` subdirectories must be placed side-by-side within the same parent directory. AMD is a stand-alone package that is required by UMFPACK. UMFPACK can be compiled without the BLAS [11, 12, 27, 25], but your performance will be much less than what it should be.

UMFPACK also requires CHOLMOD, CCAMD, CCOLAMD, COLAMD, and metis-5.1.0 by default. You can remove this dependency by compiling with `-DNCHOLMOD`. Add this to the `UMFPACK_CONFIG` definition in `SuiteSparse_config/SuiteSparse_config.mk`.

System-dependent configurations are in the `SuiteSparse_config/SuiteSparse_config.mk` file. The default settings will work on most systems, except that UMFPACK will be compiled so that it does not use the BLAS. Sample configurations are provided for Linux, Mac, Sun Solaris, SGI IRIX, IBM AIX, and the DEC/Compaq Alpha. The Makefile file will automatically detect what system you have and will set the compile parameters accordingly.

To compile both packages, go to the `UMFPACK` directory and type `make`. This will compile the static libraries (`AMD/Lib/libamd.a` and `UMFPACK/Lib/libumfpack.a`), and the shared libraries (`*.so` on Linux/Unix or `*.dylib` on the Mac). 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.

To install into /usr/local/lib and /usr/local/include, do `make install`. To uninstall from there, do `make uninstall`. For more options, see the `SuiteSparse/README.txt` file.

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

If you compile UMFPACK and AMD and then later change the `SuiteSparse_config/SuiteSparse_config` file then you should type `make purge` and then `make` to recompile.

Here are a few of the various parameters that you can control in your `SuiteSparse_config/SuiteSparse_co` file. To list them all, do `make config`.

- `CC =` your C compiler, such as `cc`.
- `RANLIB =` your system's `ranlib` program, if needed.

[5] www.cygwin.com

- `CFLAGS =` optimization flags, such as `-O`.
- `UMFPACK_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.
- `F77 =` the command to compile a Fortran program (optional).
- `F77FLAGS =` the Fortran compiler flags (optional).
- `F77LIB =` the Fortran libraries (optional).

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

- `-DNBLAS` if you do not have any BLAS at all.
- `-DNSUNPERF` if you are on Solaris but do not have the Sun Performance Library (for the BLAS).
- `-DLONGBLAS` if your BLAS takes non-`int` integer arguments.
- `-DBLAS_INT =` the integer used by the BLAS.
- `-DBLAS_NO_UNDERSCORE` for controlling how C calls the Fortran BLAS. This is set automatically for Windows, Sun Solaris, SGI Irix, Red Hat Linux, Compaq Alpha, and AIX (the IBM RS 6000).
- `-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 $\mathbf{A}$ are scaled. If `-DNRECIPROCAL` is defined (or if any scale factor is less than 10^{-12}), entries in the rows of $\mathbf{A}$ 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 $\mathbf{A}$ are applied.
- `-DNO_DIVIDE_BY_ZERO` controls how UMFPACK treats zeros on the diagonal of $\mathbf{U}$, for a singular matrix $\mathbf{A}$. If defined, then no division by zero is performed (a zero entry on the diagonal of $\mathbf{U}$ is treated as if it were equal to one). By default, UMFPACK will divide by zero.

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.

When you compile your program that uses the C-callable UMFPACK library, you need to link your program with all libraries: -lumfpack -lamd -lcholmod -lcolamd -lccolamd -lcamd -lmetis. If you don't compile UMFPACK to use METIS, then you can All libraries are placed in `SuiteSparse/lib` and all include files are placed in `SuiteSparse/include`.

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 do `make install`, then the compiler will know where to find all of the SuiteSparse libraries. Otherwise, add `-L/home/myself/SuiteSparse/lib` and `-I/home/myself/SuiteSparse/include` to your compiler flags, if your copy of SuiteSparse is located in `/home/myself/SuiteSparse`.

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.

The `umf_multicompile.c` file has been added to assist in the compilation of UMFPACK in Microsoft Visual Studio, for Windows.

8.2 Installing the MATLAB interface

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).

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\libmwlapack.lib` file into the `<matlab>\extern\lib\win32\lcc\` directory. Next, type `mex -setup` at the MATLAB prompt, and ask MATLAB to select the `lcc` compiler. MATLAB 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.

8.3 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.

In particular, I was not able to get C and Fortran to work together on the Mac (Snow Leopard).

8.4 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.

9 Future work

Here are a few features that are not in the current version of UMFPACK, 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. Remove the restriction that the column-oriented form be given with sorted columns. This has already been done in AMD Version 2.0.

2. 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].

3. 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]). See also CSparse and CXSparse for software for handling sparse right-hand sides.

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

5. When using iterative refinement, the residual $\mathbf{Ax} - \mathbf{b}$ could be returned by `umfpack_solve`.

6. The solve routines could handle multiple right-hand sides, and sparse right-hand sides. See `umfpack_solve` for the MATLAB version of this feature. See also CSparse and CXSparse for software for handling sparse right-hand sides.

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. KLU includes this feature. See also `cs_dmperm` in CSparse and CXSparse.

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

10. 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 strategy, 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. See `cs_counts` in CSparse and CXSparse, and `cholmod_analyze` in CHOLMOD.

11. Other basic sparse matrix operations, such as sparse matrix multiplication, could be included.

12. A more complete Fortran interface.

13. A C++ interface.

14. A parallel version using MPI. This would require a large amount of effort.

10 The primary UMFPACK routines

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

10.1 umfpack_*_symbolic

10.2 umfpack_*_numeric

10.3 umfpack_*_solve

10.4 umfpack_*_free_symbolic

10.5 umfpack_*_free_numeric

11 Alternative routines

11.1 umfpack_*_defaults

11.2 umfpack_*_qsymbolic and umfpack_*_fsymbolic

11.3 umfpack_*_wsolve

12 Matrix manipulation routines

12.1 umfpack_*_col_to_triplet

12.2 umfpack_*_triplet_to_col

12.3 umfpack_*_transpose

12.4 umfpack_*_scale

13 Getting the contents of opaque objects

13.1 umfpack_*_get_lunz

13.2 umfpack_*_get_numeric

13.3 umfpack_*_get_symbolic

13.4 umfpack_*_save_numeric

13.5 umfpack_*_load_numeric

13.6 umfpack_*_save_symbolic

13.7 umfpack_*_load_symbolic

13.8 umfpack_*_get_determinant

14 Reporting routines

14.1 umfpack_*_report_status

14.2 umfpack_*_report_control

14.3 umfpack_*_report_info

14.4 umfpack_*_report_matrix

14.5 umfpack_*_report_numeric

14.6 umfpack_*_report_perm

14.7 umfpack_*_report_symbolic

14.8 umfpack_*_report_triplet

14.9 umfpack_*_report_vector

15 Utility routines

15.1 umfpack_timer

15.2 umfpack_tic and umfpack_toc

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 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):381–388, 2004.

[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 832: UMFPACK, an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, 2004.

[5] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):165–195, 2004.

[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 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):377–380, 2004.

[9] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):353–376, 2004.

[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] 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.