mUMFPACK Documentation

Release 1.0

lh

April 29, 2014

CHAPTER

ONE

FORTRAN AND C

1.1 Fortran 77

For many years, Fortran users that had to link their codes with C libraries (and vice versa) were pushed to dive into too specific tasks. Different compilers did the name decoration, truncation or letter-case changes of procedure names in different ways, sometimes poorly documented, and this Fortran-C linking barrier was often too high to let the user even see other barriers behind like datatype mapping and argument passing. Fortran 90/95 did not improve anything in this regard.

For those reasons, Fortran users had the only choice to access the UMFPACK C library via calls to the **C wrappers** `umf4*` included in the UMFPACK package (`umf4_f77wrapper.c` in `UMFPACK/Demo`). The C wrappers provided only a subset of UMFPACK's capabilities and have been considered highly non-portable. Consequently, a Fortran user had to call other interface than that described in the UMFPACK User Guide, could not pass data in the Fortran native complex type (the UMFPACK's packed complex format) and had not available UMFPACK's symbolic constants. Difficulties to link together Fortran user codes and the UMFPACK C library on some platforms and with certain compilers were perhaps most painful.

1.2 Fortran 2003

The standardized **interoperability with C** has been introduced in Fortran 2003 and is implemented apparently in all major Fortran compilers (cf. Fortran 2003 status). Using the intrinsic `iso_c_binding` module, it is possible to declare interoperable types, variables, pointers, arrays and structures and handle argument passing between C and Fortran program units in both directions. In particular, codes compiled with Fortran compilers can be linked with compiled C libraries on a given platform regardless of specific compilers. However, it is necessary to describe explicitly the C library headers by **interface blocks** on the Fortran side. Moreover, direct calls to such interfaces may sometimes appear bizzare to Fortran users and often it is desirable to add **Fortran wrappers** around those interfaces.

CHAPTER

TWO

MUMFPACK MODULE

The `mUMFPACK` module provides a Fortran-2003 implementation of the **full UMFPACK interface** closely corresponding to that defined in the UMFPACK User Guide. The legacy Fortran-77 interface is implemented as well, this time using portable Fortran wrappers. The `mUMFPACK` module allows to pass data in double precision real (`di` routines) and complex types, the latter as a doublet of two arrays with real and imaginary components (`zi` routines) or as a Fortran native complex array (`ci` routines).

To use the `mUMFPACK` module, insert the `use mUMFPACK` statement into a header of a Fortran unit that makes UMFPACK calls:

```
program main
use mUMFPACK
implicit none
```

and compile and link your project with the source file `umfpack.f90` and the UMFPACK C library, e.g.:

```
# Linux Ubuntu 12.04
f90 umfpack.f90 main.f90 -lumfpack
# Windows 7
f90 umfpack.f90 main.f90 -Lc:\lib -lumfpack -lamd -lcholmod -lcolamd -lsuitesparseconfig -lblas
```

Here, `f90` stands for any `iso-c-binding` capable Fortran compiler. Successful tests were run with GNU Fortran, Intel Fortran, PGI Fortran and even g95 against the Linux Ubuntu 64bit `suitesparse` package and the Windows 32bit UMFPACK compiled by the MinGW GNU C.

The UMFPACK Fortran interface comes in five flavours described below:

- the **C-original form**, `umfpack_*`,
- the **subroutine form**, `s_umfpack_*`,
- the low-level **C-interface form**, `c_umfpack_*`,
- the high-level **operator** `.umfpack.`,
- the basic legacy form, `umf4*`.

Note that each form of the interface is illustrated in examples packaged with the `mUMFPACK` module, and three complete examples are also inserted below.

2.1 C-original form (umfpack_*)

This form offers Fortran functions and subroutines with the **same names and arguments** as in the **C version** described in the UMFPACK User Guide. Actually, these are all Fortran wrappers around the low-level C interface blocks. The primary intent of the wrappers is to allow for **optional arguments** that are necessary to mimic the null values of some arguments on the C side (`Control`, `Info` and some others). Moreover, the wrappers make the `Symbolic` and

`Numeric` pointers optional as well, and can use **private module pointers** instead of external variables. Finally, the wrappers can be referenced by **generic names** overloaded for real and complex input matrices; the generic names do not contain infixes `di`, `zi` and `ci`. Any of the following calls is thus possible:

```
! optional arguments
status=umfpack_di_symbolic(n,n,Ap,Ai,Ax,Symbolic,Control,Info)
status=umfpack_di_symbolic(n,n,Ap,Ai,Ax,Symbolic)
status=umfpack_di_symbolic(n,n,Ap,Ai,Ax)
! generic (overloaded) procedure
status=umfpack_symbolic(n,n,Ap,Ai,Ax)
```

If positional association of arguments cannot be used, **keyword association** is necessary. All formal arguments have the same names as in the original C functions:

```
! positional association of arguments
status=umfpack_di_numeric(Ap,Ai,Ax,Symbolic,Numeric,Control,Info)
! keyword association
status=umfpack_di_numeric(Ap,Ai,Ax,Control=Control,Info=Info)
```

C functions with a `void` return value correspond to Fortran **subroutines**:

```
call umfpack_di_free_symbolic(Symbolic)
call umfpack_di_free_symbolic
```

where the second statement frees the memory accessible by the private module pointer.

The `Symbolic` and `Numeric` variables are **C pointers** and should be declared as those:

```
! C pointers
type(c_ptr) Symbolic,Numeric
```

The `Control` and `Info` variables should be declared as **zero-based arrays** to allow straightforward use of the **C symbolic constants**. Note here that the triplet `Ap`, `Ai`, `Az` is optional in the call to `umfpack_solve` when the number of iterative refinements is zero and that keyword association is needed when the triplet is not present:

```
! accessing Control and Info elements by C symbolic constants
real(8) Control(0:UMFPACK_CONTROL-1),Info(0:UMFPACK_INFO-1)
Control(UMFPACK_IRSTEP)=0
status=umfpack_solve(UMFPACK_A,X=x,B=b,Control=Control,Info=Info)
print *,Info(UMFPACK_SOLVE_TIME)
```

For **complex data**, the Fortran interface accepts `zi` routines for both real component doublets and native complex data, and also defines `ci` routines for the latter. Generic names without any infix are possible as well:

```
! complex data as real component doublets or native complex type
real(8) Ax(nnz),Az(nnz),Xx(n),Xz(n),Bx(n),Bz(n)
complex(8) Ac(nnz),Xc(n),Bc(n)
status=umfpack_zi_solve(sys,Ap,Ai,Ax,Az,Xx,Xz,Bx,Bz)
status=umfpack_zi_solve(sys,Ap,Ai,Ac,Xc,Bc)
status=umfpack_ci_solve(sys,Ap,Ai,Ac,Xc,Bc)
status=umfpack_solve(sys,Ap,Ai,Ac,Xc,Bc)
```

Certain routines could not be overloaded as the specific routines are not distinguishable by the argument types. In such cases, routines with generic names still exist but have one extra argument `version` that is optional with the default value of `"di"`:

```
! default same as "di"
call umfpack_free_symbolic(Symbolic)
! for private module pointer and complex data
call umfpack_free_symbolic(version="zi")
```

Finally, let us put it all together to make the Fortran version of the `umfpack_simple.c` example from the `UMFPACK\Demo` directory:

```
! a complete example for the C-original interface
! ( 2  3  0  0  0 ) ( 1 ) = (  8 )
! ( 3  0  4  0  6 ) ( 2 ) = ( 45 )
! ( 0 -1 -3  2  0 ) ( 3 ) = ( -3 )
! ( 0  0  1  0  0 ) ( 4 ) = (  3 )
! ( 0  4  2  0  1 ) ( 5 ) = ( 19 )
program umfpack_simple
use mUMFPACK
implicit none
integer,parameter :: n=5,nnz=12
integer :: Ap(0:n)=[0,2,5,9,10,12]
integer :: Ai(nnz)=[0, 1, 0,  2, 4, 1,  2, 3, 4, 2, 1, 4]
real(8) :: Ax(nnz)=[2.,3.,3.,-1.,4.,4.,-3.,1.,2.,2.,6.,1.]
real(8) :: b(n)=[8.,45.,-3.,3.,19.],x(n)
integer :: status,i
type(c_ptr) :: Symbolic,Numeric
status=umfpack_di_symbolic(n,n,Ap,Ai,Ax,Symbolic)
status=umfpack_di_numeric(Ap,Ai,Ax,Symbolic,Numeric)
call umfpack_di_free_symbolic(Symbolic)
status=umfpack_di_solve(UMFPACK_A,Ap,Ai,Ax,x,b,Numeric)
call umfpack_di_free_numeric(Numeric)
do i=lbound(x,1),ubound(x,1)
  print '(a,i0,a,f0.10)',"x(",i,") = ",x(i)
enddo
end program
```

2.2 Subroutine form (s_umfpack_*)

To those Fortran users that prefer **subroutine calls** to a library, the subroutine form of the interface with the same features as the C-original form is made ready. All subroutines begin with the `s_` prefix, including those that are subroutines in the C-original form already. The `status` return value of the corresponding functions is still available as the last optional argument:

```
! UMFPACK via subroutine calls
integer,parameter :: n=5,nnz=12
integer Ap(0:n),Ai(nnz)
real(8) Ax(nnz),b(n),x(n)
integer status
call s_umfpack_symbolic(n,n,Ap,Ai,Ax,status=status)
call s_umfpack_numeric(Ap,Ai,Ax)
call s_umfpack_free_symbolic
call s_umfpack_solve(UMFPACK_A,Ap,Ai,Ax,x,b)
call s_umfpack_free_numeric
```

2.3 Low-level C-interface (c_umfpack_*)

Rarely, if ever, it might be useful to call UMFPACK C routines directly, without Fortran wrappers. Such calls have assigned names of the corresponding C routines with the `c_` prefixed. All arguments are mandatory. Integer scalars are passed by value. All non-scalar arguments are declared as C pointers on the Fortran side to allow for passing `null` pointers to the C side. It obliges the user to use the Fortran function `c_loc` frequently to get C pointers of the actual arguments. Targets of the `c_loc` functions must be declared with the `target` attribute and must be addressable:

```
! UMFPACK via direct C calls
integer,parameter :: nx=10,ny=20,n=nx*ny,nnz=1000
integer,target :: Ap(0:n),Ai(nnz) ! arguments of c_loc must be targets or pointers
real(8),target :: Ax(nnz),b(nx,ny),x(nx,ny)
type(c_ptr) :: Symbolic,Numeric
status=c_umfpack_di_solve(UMFPACK_A,c_loc(Ap),c_loc(Ai),c_loc(Ax),c_loc(x),c_loc(b), &
       Numeric,c_null_ptr,c_null_ptr)
```

Note that `x` and `b` are declared as 2D matrices here that may fit a problem geometry better in some setups. In such a case, either explicit reshape of the matrices into vectors is necessary before calling the standard `umfpack_solve` wrapper, or a direct call to `c_umfpack_di_solve` like here may be of benefit. Note also that these routines are not overloaded, the `di` and `zi` infixes cannot be omitted and the `ci` infix is not available.

2.4 High-level operator (.umfpack.)

A new Fortran operator, **.umfpack.**, is defined to perform a typical sequence of UMFPACK calls, i.e., symbolic and numeric factorization, a solver call for one right-hand side and deallocation of memory. The operator is overloaded for real and complex data and, moreover, for input arrays given in the (default) **CSC** and (non-default) **CSR formats** (compressed sparse columns and rows, respectively).

Because Fortran operators can be defined for at most two arguments, the array triplet `Ap`, `Ai`, `Ax` (or a quadruplet in the `zi` case) has to be encapsulated into a structure that becomes the first argument, and the right-hand side vector is the other one (a doublet in the `zi` case). Several defined data types are made ready for that. These types differentiate among real and complex data (`di`/`zi`/`ci`), sparse matrix formats (`CSC`/`CSR`) and between a storage scheme used for the components – types `t*` use **allocatable arrays** and types `p*` **pointer arrays**:

double	complex	complex	description of the derived type
`tCSC_di`	`tCSC_zi`	`tCSC_ci`	CSC-formatted allocatable components
`tCSR_di`	`tCSR_zi`	`tCSR_ci`	CSR-formatted allocatable components
`pCSC_di`	`pCSC_zi`	`pCSC_ci`	CSC-formatted pointer components
`pCSR_di`	`pCSR_zi`	`pCSR_ci`	CSR-formatted pointer components

A user should declare a triplet structure, initialize it using a structure constructor and invoke the `.umfpack.` operator for the triplet and a right-hand side. An example for real CSC-formatted data and the triplet with **allocatable components**:

```
! .umfpack. operator for a triplet with allocatable components
integer Ap(0:n),Ai(nnz)
real(8) Ax(nnz),b(n),x(n)
type(tCSC_di) A     ! a structure with allocatable components
A=tCSC_di(Ap,Ai,Ax) ! a standard structure constructor
x=A .umfpack. b
print *,x
```

In this approach, extra memory is automatically allocated for data arrays that are stored in structure components duplicitly. If it is undesirable, a user may store data directly to the triplet structure, or use the other approach with **pointer components**:

```
! .umfpack. operator for a triplet with pointer components and a standard structure constructor
integer Ap(0:n),Ai(nnz)
real(8) Ax(nnz),b(n),x(n)
target Ap,Ai,Ax     ! explicit target attributes
type(pCSC_di) A     ! a structure with pointer components
A=pCSC_di(Ap,Ai,Ax) ! a standard structure constructor
x=A .umfpack. b
print *,x
```

The triplet components here are pointer arrays associated with the target arrays and no extra memory is allocated. However, a user is required to declare explicitly the target attribute for the target arrays, and the target arrays must be addressable (not, say, the function results). As always with pointers, this approach is more sensitive to coding impurities, and it seems that some compilers are not as reliable here as with the allocatable arrays. Perhaps a more robust alternative to the latter case is based on **defined structure constructors** `pCSC` and `pCSR`, both overloaded for real and complex data. Note that it is not required to declare target attributes for the arguments of a defined structure constructor:

```
! .umfpack. operator for a triplet with pointer components and a defined structure constructor
integer Ap(0:n),Ai(nnz)
real(8) Ax(nnz),b(n),x(n)
type(pCSC_di) A  ! a structure with pointer components
A=pCSC(Ap,Ai,Ax) ! a defined structure constructor
x=A .umfpack. b
print *,x
```

Finally, the previous snippet is integrated into another complete version of the `umfpack_simple` example:

```
! a complete example for the .umfpack. operator
program umfpack_simple
use mUMFPACK
implicit none
integer,parameter :: n=5,nnz=12
integer :: Ap(0:n)=[0,2,5,9,10,12]
integer :: Ai(nnz)=[0, 1, 0,  2, 4, 1,  2, 3, 4, 2, 1, 4]
real(8) :: Ax(nnz)=[2.,3.,3.,-1.,4.,4.,-3.,1.,2.,2.,6.,1.]
real(8) :: b(n)=[8.,45.,-3.,3.,19.],x(n)
integer :: i
type(pCSC_di) :: A
A=pCSC(Ap,Ai,Ax)
x=A .umfpack. b
do i=lbound(x,1),ubound(x,1)
  print '(a,i0,a,f0.10)',"x(",i,") = ",x(i)
enddo
end program
```

2.5 Basic F77 interface (umf4*)

Since early times, UMFPACK C functions have been available to Fortran users through C wrappers. While it was straightforward to compile and link all together on some platforms, it was difficult on others. For that reason, the `mUMFPACK` module implements the basic legacy Fortran interface as well. The C wrappers are not needed anymore, the work is done by portable **Fortran wrappers**.

As an added value, `mUMFPACK` also contains wrappers to handle Fortran **native complex arrays** (UMFPACK packed complex type). Moreover, all specific routines (`umf4*`/`umf4z*`/`umf4c*`) are overloaded by the **generic names** `umf4*`. In other words, the wrappers for real arguments apparently accept complex data as well. In contrast to the `umfpack_*` wrappers, all arguments of the `umf4*` wrappers are mandatory:

```
! umf4* wrappers overloaded for real and complex input arrays
real(8) Ax(nnz),Az(nnz)
complex(8) Ac(nnz)
call umf4sym(n,n,Ap,Ai,Ax,symbolic,control,info)    ! real arguments
call umf4sym(n,n,Ap,Ai,Ax,Az,symbolic,control,info) ! complex arguments as real doublets
call umf4sym(n,n,Ap,Ai,Ac,symbolic,control,info)    ! native complex arguments
```

The issue is how to declare the `symbolic` and `numeric` **C pointers** in `umf4*` wrappers. The natural way is `type(c_ptr)`, but `integer` type is also accepted to keep backward compatibility with the C wrappers. Note

however, that while `integer(4)` is probably enough on 32bit systems, `integer(8)` is needed on 64bit systems. Fortran C-like symbolic constant `c_intptr_t` should return the appropriate value. It is thus advisable, but still optional, to specify the legacy `integer` pointers as `integer(c_intptr_t)`. Some compilers (g95, in particular) have been found to work correctly with the `type(c_ptr)` declaration only.

```
! umf4* wrappers overloaded for type(c_ptr) and integer(c_intptr_t) C pointers
type(c_ptr) symbolic,numeric
integer(c_intptr_t) symbolic,numeric
```

The implicit lower bound of array indices is 1 in Fortran, but **zero-based arrays** can be explicitly declared. The latter option is advisable for `umf4*` wrappers as the C symbolic constants can then be used without `+1` corrections:

```
! zero-based arrays
real(8) control(0:UMFPACK_CONTROL-1),info(0:UMFPACK_INFO-1)
print *,info(UMFPACK_SOLVE_TIME)

! one-based arrays
real(8) control(UMFPACK_CONTROL),info(UMFPACK_INFO)
print *,info(UMFPACK_SOLVE_TIME+1) ! correction +1 necessary
```

Finally, the third complete `umfpack_simple` example follows:

```
! a complete example for the basic legacy Fortran interface
program umfpack_simple
use mUMFPACK
implicit none
integer,parameter :: n=5,nnz=12
integer :: Ap(0:n)=[0,2,5,9,10,12]
integer :: Ai(nnz)=[0, 1, 0,  2, 4, 1,  2, 3, 4, 2, 1, 4]
real(8) :: Ax(nnz)=[2.,3.,3.,-1.,4.,4.,-3.,1.,2.,2.,6.,1.]
real(8) :: b(n)=[8.,45.,-3.,3.,19.],x(n)
integer(c_intptr_t) :: symbolic,numeric
! type(c_ptr) :: symbolic,numeric     ! equivalent alternative declaration
real(8) :: control(0:UMFPACK_CONTROL-1),info(0:UMFPACK_INFO-1)
integer :: i
call umf4def(control)
call umf4sym(n,n,Ap,Ai,Ax,symbolic,control,info)
call umf4num(Ap,Ai,Ax,symbolic,numeric,control,info)
call umf4fsym(symbolic)
call umf4solr(UMFPACK_A,Ap,Ai,Ax,x,b,numeric,control,info)
call umf4fnum(numeric)
do i=lbound(x,1),ubound(x,1)
  print '(a,i0,a,f0.10)',"x(",i,") = ",x(i)
enddo
```

CHAPTER

THREE

PACKAGE CONTENT

The source file `umfpack.f90` that contains the `mUMFPACK` module is packaged with 12 Fortran examples (7 variants of `umfpack_simple.c` and 5 variants of `umf4hb.f`/`umf4zhb.f`), the C original `umfpack_simple.c`, `README` files (html, pdf, txt), `Makefile` and `Make.inc`, two alternative compilation scripts and two data files for the `umf4hb` examples.

The `umfpack_simple` examples illustrate various forms of the UMFPACK Fortran interface: from basic legacy `umf4*` calls through C-original calls `umfpack_*` to the `.umfpack.` operator. The `umf4hb` example presents a minimal amount of code patches necessary to move from the legacy C wrappers to the `mUMFPACK`'s Fortran wrappers:

- insertion of the `use mUMFPACK` statement
- (optional) refinement of declarations of `Symbolic` and `Numeric` variables to `type(c_ptr)` or `integer(c_intptr_t)`. The change would make the declarations portable to both 32bit and 64bit systems.

Download the tgz or zip archives. Compile `umfpack.f90` with your project, or just run test examples. For examples, first edit `Make.inc` to select a subset of compiler-related variables. Second, run `make` to compile all examples. Third, run `make run` and check output of `umfpack_simple` and `umf4hb`/`umf4zhb`. The UMFPACK C library with required dependencies (BLAS) should be installed independently.

- `make`, `make all`, `make Fortran`, `make F`, `make simple hb` compile all Fortran examples
- `make C` compiles the `umfpack_simple.c` example
- `make simple` compiles `umfpack_simple`-based examples
- `make hb` compiles `umf4hb`-based examples
- `make run` or `make run_simple run_hb` start test runs
- `make clean` removes everything created during compilation

Note that a port of `make` for Windows is available in the MSYS package, a supplement of MinGW. This port can interpret a Linux `Makefile` in Windows without any change. Current versions of `make` by equation.com (a provider of MinGW GNU C/Fortran) or `nmake` by Microsoft do not reach this ideal that easy.

CHAPTER

FOUR

CHANGELOG

Version 1.0 (April 28, 2014)

Initial release of the `mUMFPACK` module. Tested with UMFPACK version 5.6.2. Implemented all primary routines and other routines necessary to reimplement the legacy UMFPACK Fortran interface. Supported `di`, `zi` and packed complex `ci` versions of the routines. Provided C symbolic constants. 5 flavours of the Fortran interface: Fortran wrappers that mimic the C interface (`umfpack_*`), the same wrappers as subroutines (`s_umfpack_*`), interface blocks to the C routines (`c_umfpack_*`), a defined operator (`.umfpack.`) and legacy Fortran-77 wrappers (`umf4*`). All routines overloaded to allow omission of `di`/`zi`/`ci` infixes. Many arguments declared as optional, including the `Symbolic` and `Numeric` pointers that may be silently provided as private module variables. The `.umfpack.` operator accepts CSC- or CSR-formatted array triplets with allocatable or pointer components.

Tested on the Linux Ubuntu 12.04 64bit and Windows 7 32bit systems with GNU Fortran (4.4.3, 4.6.3, 4.8.2), Intel Fortran (11.1, 13.1, 14.0), Portland Fortran (14.2) and g95 (0.93, 0.94) against the Linux Ubuntu 64bit `suitesparse` package and the Windows 32bit UMFPACK compiled by equation.com GNU C. Attached 7 versions of the `umfpack_simple` example and 5 versions of the `umf4hb`/`umf4zhb` example. Provided `Makefile` with `Make.inc` and Sphinx-formatted `README`.

Interfaces implemented in the `mUMFPACK` module (`di`/`zi`/`ci` routines):

- primary routines (UMFPACK User Guide, sections 6.1, 6.6): `symbolic`, `numeric`, `solve`, `free_symbolic`, `free_numeric`
- alternative routines (6.2, 6.7): `defaults`
- matrix manipulation routines (6.3, 6.8): `scale`
- getting the content (6.4, 6.9): `load_numeric`, `save_numeric`, `load_symbolic`, `save_symbolic`
- reporting routines (6.5, 6.10): `report_control`, `report_info`
- legacy Fortran-77 routines (7): `umf4def`, `umf4zdef`, `umf4sym`, `umf4zsym` etc. (all routines)

To-do list

Interfaces not yet implemented in the `mUMFPACK` module:

- alternative routines (6.2, 6.7): `qsymbolic`, `fsymbolic`, `wsolve`
- matrix manipulation routines (6.3, 6.8): `col_to_triplet`, `triplet_to_col`, `transpose`
- getting the content (6.4, 6.9): `get_lunz`, `get_numeric`, `get_symbolic`, `get_determinant`
- reporting routines (6.5, 6.10): `report_status`, `report_matrix`, `report_numeric`, `report_perm`, `report_symbolic`, `report_triplet`, `report_vector`
- utility routines (6.11): `timer`, `tic`, `toc`
- AMD ordering routines (6.12): `amd_defaults`, `amd_order`, `amd_control`, `amd_info`
- any routine with `integer(8)` arguments (`dl`/`zl`/`cl` routines)

CHAPTER

FIVE

DOWNLOAD

Both packages have the same content, have been tested on Linux and Windows platforms and are expected to be usable on others.

- mUMFPACK.tgz
- mUMFPACK.zip