

MUESLI

Inside

Fortran implementation

Release 2.23.5

Édouard CANOT*

May 4, 2026



*IPR/CNRS, Rennes, France

Contents

1	Introduction	3
2	Implementation of the numerical part: FML	4
2.1	Structure of FML	4
2.2	The <code>mod_core</code> module	4
2.2.1	The <code>mfArray</code> derived type	5
2.2.2	About assignments between <code>mfArrays</code>	7
2.2.3	Index sequences for <code>msSet</code> and <code>mfGet</code>	8
2.2.4	The <code>mfMatFactor</code> derived type	9
2.3	The <code>mod_sparse</code> module	9
2.4	The <code>mod_elmat</code> module	9
2.5	The <code>mod_fileio</code> module	9
2.6	The <code>mod_elfun</code> module	10
2.7	The <code>mod_ops</code> module	10
2.8	The <code>mod_datafun</code> module	10
2.9	The <code>mod_specfun</code> module	10
2.10	The <code>mod_matfun</code> module	10
2.10.1	Implementation of some specific routines	10
2.11	The <code>mod_polyfun</code> module	10
2.11.1	2D triangulation	10
2.12	The <code>mod_funfun</code> module	10
2.13	Overall modules' graph	10
3	Coupling the Muesli library with a graphic tool	13
3.1	The example of <code>meditor</code>	13
4	Implementation of the graphical part: FGL	14
4.1	Overall modules' graph	14
4.2	Predefined colors in MFLOT	14
4.3	Image indexing schemes	15
4.4	The available devices and their driver	16
4.5	Managing the graphic memory	16
4.6	Fonts used in FGL	17
5	Source organization and Compilation tools	18
5.1	On demand traceback	18
5.2	Debugging tools	18
5.3	Modified compilers	19
A	MBF format	20
B	HDF5 format	21

1 Introduction

This document describes the implementation of the MUESLI Fortran library.

MUESLI is freely available at the following address: <https://perso.univ-rennes1.fr/edouard.canot/muesli>

More information can be found in the following documents:

- *MUESLI Installation Guide*
- *MUESLI User's Guide*
- *MUESLI Reference Manual*

Copyright © 2003-2026, Édouard CANOT, IPR/CNRS, Rennes, France.

Bugs reports or comments: <mailto:Edouard.Canot@univ-rennes.fr>

About the name

Muesli: loose mixture of mainly rolled oats and often also wheat flakes, together with various pieces of dried fruit, nuts, and seeds. There are many varieties, some of which also contain honey powder, spices, or chocolate. (from <https://en.wikipedia.org/wiki/Muesli>)

Credits

Cover photograph: the photo on the cover is copyrighted by Vania Dobrinova. It can be used under a Limited Royalty Free License (see <http://www.dreamstime.com/watermelon-imagefree2893158>).

2 Implementation of the numerical part: FML

2.1 Structure of FML

From the user point-of-view, the FML part of MUESLI is seen as one big Fortran module. However, FML has been split in several components for the main following reason: during the development, we must compile FML a great number of times – even after a small change (without changing any routine API) the whole module compilation takes some time.

At the beginning of the Muesli library design, the *submodule* feature of Fortran 2008 was not available¹. Ok we would have decomposed FML in ordinary modules but we found another difficulty about module access of private components. Actually, the current solution is to compile twice each (sub)module²; a first time (in direct inclusion order) with all (or selected) components declared as public, and keeping the object files created; a second time (in reverse inclusion order) with some components declared as private, without creating the object files, *i. e.* via a syntax-only way, leading to a quicker second compilation. This works well³.

So, FML is the set⁴ of the following modules:

- `mod_core` (Core routines)
- `mod_sparse` (Sparse Matrices)
- `mod_elmat` (Elementary Matrix Manipulation Functions)
- `mod_fileio` (File Input/Output)
- `mod_elfun` (Elementary Math Functions)
- `mod_ops` (Operators)
- `mod_datafun` (Data Analysis Functions)
- `mod_specfun` (Specialized Math Functions)
- `mod_matfun` (Matrix Functions)
- `mod_polyfun` (Polynomial Functions)
- `mod_funfun` (Optimization and Function Functions)

The previous modules correspond to the sub-parts described in the *MUESLI Reference Manual* (section 1 about FML). Lastly, the `fml` module `uses` (*i. e.* includes) all of them.

2.2 The `mod_core` module

This basic module (file '`mod_core.F90`') constitutes the core of MUESLI and includes other (sub)modules⁴:

- `mod_mfdebug` (for debug purpose)
- `mod_mem_debug` (for memory debug purpose)
- `mod_mfaux` (auxiliary routines)
- `mod_prop` (matrix properties)
- `rational_numbers` (rational numbers, used in physical units)
- `mod_mfarray` (fundamental derived type of MUESLI)
- `mod_physunits` (physical units)
- `mod_ieee` (IEEE special routines)

¹on september 2016, the *submodule* feature is supported by few compilers: CRAY, IBM, INTEL and GNU; but it was not the case in the past, during the development of the version 2 of Muesli.

²hereafter, we introduce the notation “(sub)module” to distinguish from the true Fortran 2008 “submodule”.

³a switch to use the true *submodule* feature would imply a complete refactoring of the library; it is possible that the next major version of Muesli does use *submodules*.

⁴these modules are listed in USE order.

The following section will focus on the `mfArray` derived type and assignment associated with its use. Beside that, the `mod_core` module defines, among other things, conversion between ordinary Fortran scalar/vector/array variables to `mfArray` (`mf()` routine), display of `mfArrays`, routines to set and/or get element(s) or array section inside an `mfArray`.

2.2.1 The `mfArray` derived type

The `mfArray` derived type contains all stuff to deal with automatic (*i.e.* dynamic in size, type and structure) numerical arrays.

It is defined inside the `mod_mfarray` module and is designed as follows:

```

type mfArray

    integer(kind=kind_1) :: data_type = MF_DT_EMPTY ! current data type
    integer               :: shape(2) = [ 0, 0 ] ! matrix dimensions (row,col)

    ! dense matrix
    real(kind=MF_DOUBLE),    pointer :: double(:, :) => null() ! type #1
    complex(kind=MF_DOUBLE), pointer :: cmplx(:, :) => null() ! type #2
    ! booleans are defined as 1 and 0 in the 'double' field ! type #3

    ! sparse matrix in CSC format (Compressed Sparse Column)
    real(kind=MF_DOUBLE),    pointer :: a(:) => null() ! type #4
    complex(kind=MF_DOUBLE), pointer :: z(:) => null() ! type #5
    integer,                  pointer :: i(:) => null(), &
                                   j(:) => null()

    ! in each compressed column, rows are not necessarily sorted
    integer(kind=kind_1) :: row_sorted = UNKNOWN ! may be also FALSE or TRUE
    ! ptr to numeric structure of UMFPACK
    integer(kind=MF_ADDRESS), pointer :: umf4_ptr_numeric => null()

    ! matrix pattern and properties
    type(properties) :: prop

    integer(kind=kind_1) :: level_locked = 0 ! used by 'msPointer'
    logical :: crc_stored = .false.
    integer :: crc = 0 ! a 32-bit checksum is stored to verify that data has
                      ! not been modified during the use of 'msPointer'

    ! about the value returned by a function.
    logical :: status_temporary = .false.

    ! protection against deallocation during nested calls
    integer(kind=kind_1) :: level_protected = 0

    ! restricted mode (used by 'msEquiv')
    ! if mfArray points to a Fortran (static) array
    logical :: status_restricted = .false.

    ! parameter mode (data are protected, except matrix properties)
    logical :: parameter = .false.

    ! physical unit
    type(rational) :: units(num_base_units)

end type mfArray

```

The first field, `data_type`, describes the data type and takes its value from the following parameters, defined in the same module (`mod_mfarray`):

```

integer, parameter :: MF_DT_EMPTY   = 0,
MF_DT_DBLE         = 1,
MF_DT_CMPLX        = 2,
MF_DT_BOOL         = 3,
MF_DT_SP_DBLE      = 4,
MF_DT_SP_CMPLX     = 5,
MF_DT_PERM_VEC     = 6

```

The second field, `shape(2)`, is an integer vector storing the two dimensions of the matrix, *i. e.* the number of rows and the number of columns.

Then, we find several arrays (data containers) for both the dense and the sparse cases, and some matrix properties. Data containers are all implemented as pointers to array, and not as allocatable arrays, because we want to keep the possibility to points to (user's) ordinary Fortran arrays via the `msEquiv` routine. Before copying data, memory must be of course allocated for these pointers and this is done by internal routines; moreover, this memory must be freed at one time or another to avoid memory leaks. In this document, we will employ “allocate an `mfArray`” to mean that some memory is allocated for one of the data containers and “release an `mfArray`” to mean that all memory used by this `mfArray` is freed (or deallocated).

Let's now enumerate all the possibilities for the data types which can be contained in an `mfArray`:

1. the first data type is DBLE, *i. e.* real numeric. Only the 2D array `double` field is used to store the matrix. This matrix have mathematical properties which are also stored inside `mfArray`. These properties are described below.
2. the second data type is the CMPLX one, *i. e.* complex numeric. The values of the complex matrix are stored in the `cmplx` field. As for the previous data type, CMPLX has properties.
3. the third data type is BOOL, *i. e.* boolean or, in terms of Fortran language, logical. Booleans are implemented as double: a value equal to 1 is equivalent to TRUE, and a value equal to 0 is equivalent to FALSE.
4. the fourth data type is SP_DBLE, *i. e.* real sparse matrix. Numeric entries are stored in the `a` rank-1 field, whereas the indices (for the CSC format) are stored in the integer vectors `i` and `j`.
5. the fifth data type is SP_CMPLX, *i. e.* complex sparse matrix. Numeric values are stored in the complex rank-1 field `z`, whereas the indices are stored in `i` and `j`.
6. the sixth data type is PERM_VEC, *i. e.* a permutation vector. It is stored in the rank-1 integer array `i`.

The matrix pattern and properties are packed in the following derived type (declared in the `mod_prop` (sub)module):

```

type properties
  ! structure pattern (independant of matrix values)
  integer(kind=kind_1) :: tril = UNKNOWN ! lower triangular
  integer(kind=kind_1) :: triu = UNKNOWN ! upper triangular
  ! matrix properties (depend on values)
  integer(kind=kind_1) :: symm = UNKNOWN ! symmetric/hermitian
  integer(kind=kind_1) :: posd = UNKNOWN ! positive definite
end type properties

```

Different internal flags are fundamental:

- `level_locked` protects the `mfArray` against any resize or reshape when it is pointed by an ordinary Fortran array: it is incremented (resp. decremented) at each call of the `msPointer` (resp. `msFreePointer`) routine.

In connection to this flag, another one, `crc_stored`, indicates that a checksum is computed (to check if the user has modified the data) to ensure that the matrix properties remain valid⁵. For performance reasons, it may be preferable to not find again the matrix properties: therefore, in case where the data has been modified, all the matrix properties are set to `UNKNOWN`.

- `status_temporary` ensures that the `mfArray` object returned by a function will be released as soon as possible after its use. It can be set by an ordinary user via the `msReturnArray` routine, as explained in the *MUESLI User Guide*, section 4.3.5; it does not need to be unset.
- `level_protected` protects the `mfArray` data returned by a function when nested calls are used. It should be incremented (resp. decremented) only via the `msInitArgs` (resp. `msFreeArgs`) routine.
- `status_restricted` is set by the `msEquiv` routine, when the `mfArray` “points” to a Fortran array; in this latter case, the `mfArray` cannot change its own shape and then works in a “restricted” mode.

Another flag (`parameter`) has been added to simulate data protection, because it is not possible to declare a `type(mfArray)` variable with the `parameter` attribute.

Lastly, the field `units` holds the physical unit of the `mfArray`, in terms of an array for the base units.

2.2.2 About assignments between `mfArrays`

Most of time, assignment in MUESLI uses the ‘=’ symbol, thanks to the overload of the corresponding operator. So, a statement of the form

`A = B`

is processed in a way which depends on the type of `A` and `B`. Many cases occur:

`(mfArray) A = (mfArray) B`

- the `mfArray A` is “released”;
- data of `B` is copied to `A`
(when `B` is a temporary object, the user is informed of a loss of performance, according to the message level, because the more efficient `msAssign` routine could be used instead of ‘=’, see below);
- the `mfArray B` is “released” if it is temporary and not protected (via `msInitArgs`).

`(mfArray) A = (Fortran) B`

- the `mfArray A` is “released”;
- data of `B` (integer / logical / real / double precision / complex / double complex; any rank in 0-2) is copied to `A`.

`(Fortran) A = (mfArray) B`

- the `Fortran A` may be integer / logical / real / double precision / complex / double complex, and of any rank in 0-2;
- the `mfArray B` (which must have the same type and shape as `A`) is copied to `A`;
- the `mfArray B` is “released” if it is temporary and not protected (via `msInitArgs`).

⁵This 32-bit CRC may be time-consuming for large arrays. For this reason, an additional optional argument, not documented in the *MUESLI Reference Manual*, allows internal calls from the Muesli library to avoid this computation (see `msPointer` in the index); we are also looking for another way to protect the memory, without the computation of a checksum.

Increasing performance

The case described above:

```
(mfArray) A = (tempo mfArray) B
```

implies a true copy between A and B; when B is a temporary object and the user wants to increase performance, he can write the statement `A = B` as:

```
call msAssign( A, B )
```

Indeed, the `msAssign` routine transfers the data of B (via pointers) to A without any copy.

Be careful when using the previous `msAssign` call inside a routine, when B is an argument of this routine:

- the ‘tempo’ flag of B is removed, but nothing else. Thus, if the user has access to B (*e.g.* from inside a routine, when B is an argument which is a temporary `mfArray`), he can yet use B; but he should not modify B, because A would be modified as well!
- similarly, when B comes from nested calls, you should not modified A, because B also would be modified. You can test the value of `B%level_protected` to take the decision of either use `call msAssign(A,B)` or use `A=B`.

2.2.3 Index sequences for `msSet` and `mfGet`

Index sequences are defined via the following derived type

```
type seq_def

  integer :: start_1 = 1
  integer :: end_1   = 0
  integer :: step_1  = 1
  logical :: start_1_EndIndex = .false.
  logical :: end_1_EndIndex = .false.

  integer :: start_2 = 1
  integer :: end_2   = 0
  integer :: step_2  = 1
  logical :: start_2_EndIndex = .false.
  logical :: end_2_EndIndex = .false.

  integer :: but = 0
  logical :: but_present = .false.
  logical :: but_EndIndex = .false.

end type seq_def
```

Index sequences dealing with constant (predictable) integers are processed by routines stored in `fml_core/seq_def.inc`. Others, *i.e.* those implying `MF_END`, are processed by routines stored in `fml_core/seq_def_EndIndex.inc`, using negative integers to be relative to the end of the array. In all cases, the index sequence is built by the routine `build_int_seq` located in `fml_core/seq_def.inc`. `build_int_seq` is always called by `msSet` or `mfGet` in such a way that the actual value of `MF_END` is known for each dimension.

The following predefined operators: ‘`.from.`’, ‘`.to.`’ and ‘`.by.`’ are described in the *Muesli Reference Manual* and in the *Muesli User’s Guide*.

For compatibility reason, the operator name ‘`.step.`’ is kept as an alias of ‘`.by.`’.

The operators ‘`.but.`’ and ‘`.and.`’ have been introduced recently (as of version 2.12.10) in *muesli* in order to add smart features in writing index sequences:

- ‘`.but.`’ remove one (and only one) integer from an index sequence;

- ‘.and.’ combine two index sequences at a time.

Each of the preceding two operators can be used only one time per sequence, because the derived type `seq_def` includes only two parts.

2.2.4 The `mfMatFactor` derived type

```

type mfMatFactor

    integer :: data_type = MF_DT_EMPTY

    ! dense factor
    type(mfArray), pointer :: mf_ptr_1 => null() ! L
    type(mfArray), pointer :: mf_ptr_2 => null() ! U
    type(mfArray), pointer :: mf_ptr_3 => null() ! P

    ! sparse factor
    integer :: order ! nb of rows of the matrix

    ! 1 (LU UMFPACK): L and U factors      ptr_1
    ! 2 (LL' CHOLMOD): only L factor      ptr_1, ptr_2
    ! 3 (QR SPQR):    only Q Householder ptr_1, ptr_2, ptr_3, ptr_4
    integer :: package = 0

    ! contains addresses of C structures
    integer(kind=MF_ADDRESS), pointer :: ptr_1 => null(), &
                                         ptr_2 => null(), &
                                         ptr_3 => null(), &
                                         ptr_4 => null()

    ! used only for package = 3
    integer :: shape(2) ! shape of Q

    ! physical unit
    type(rational) :: units(num_base_units)

end type

```

This derived type is used to store either some factors (obtained from few matrix decomposition of the `mod_matfun` module), either in dense or in sparse format. It is closely related to the use of the *SuiteSparse* numerical library.

2.3 The `mod_sparse` module

Most of routines of this module come from the *SPARSKIT* library, but they have been modified or improved.

2.4 The `mod_elmat` module

2.5 The `mod_fileio` module

The `mod_fileio` module contains routines performing Input/Output operation involving `mfArrays`.

It includes the following (sub)modules:

- `f90_gzlib` (a f90 wrapper of zlib)
- `iso_varying_string` (ISO/IEC 1539-2:2000 – varying-length strings for Fortran 95)

This module contains routines which are able to dump `mfArrays` to ASCII files, binary MBF files (a binary format specific to MUESLI) and HDF5 files.

MBF format and HDF5 organisation are respectively described in appendices [A](#) and [B](#).

2.6 The `mod_elfun` module

The `mod_elfun` module contains routines performing elementary functions.

It includes the following (sub)module:

- `mod_mf_gsl` (a very small subset of the GSL library, translated in Fortran 90)

2.7 The `mod_ops` module

2.8 The `mod_datafun` module

2.9 The `mod_specfun` module

2.10 The `mod_matfun` module

2.10.1 Implementation of some specific routines

- `mfFunm/msFunm` uses the Parlett’s method, as in MATLAB®-6.5.2
The user function, as described in the *MUESLI Reference Manual*, must apply to complex numbers.
- `mfExpn` (the matrix exponential) uses the Padé approximation, as in MATLAB®-6.5.2
- `mfLogm`, `mfPowm` and `mfSqrtm` use the Parlett’s method, as in MATLAB®-6.5.2

2.11 The `mod_polyfun` module

The `mod_polyfun` module contains routines performing polynomial operations.

It includes the following (sub)modules:

- `bezier` (f90 implementation of Bézier curves)
- `splines` (f90 implementation of Splines interpolation methods)

2.11.1 2D triangulation

A 2D triangulation is usually made by the `msDelaunay` routine. It provides a direct triangles’ orientation (i.e. anti-clockwise). This is mentioned by the documentation.

The `msBuildTriConnect` routine, which compute the mesh connectivity, allows an undefined orientation of all triangles. However, when checking the convexity of the mesh’s nodes, we are required to employ a direct orientation for all the triangles. This is why a check has been added inside the `msBuildTriConnect` routine.

2.12 The `mod_funfun` module

The `mod_funfun` module contains routines performing quadratures, optimization and ODE/DAE integrations. It is based on the Fortran 77 SLATEC package and the MinPACK library, which have been both modified.

It includes the following (sub)module:

- `ddeb2` (sparse solver access for Jacobian)

2.13 Overall modules’ graph

Dashed lines indicates a partial *use* of a module, whereas continuous ones indicates a full *use*.

Figure [1](#) shows the whole dependency graph of the FML’s modules.

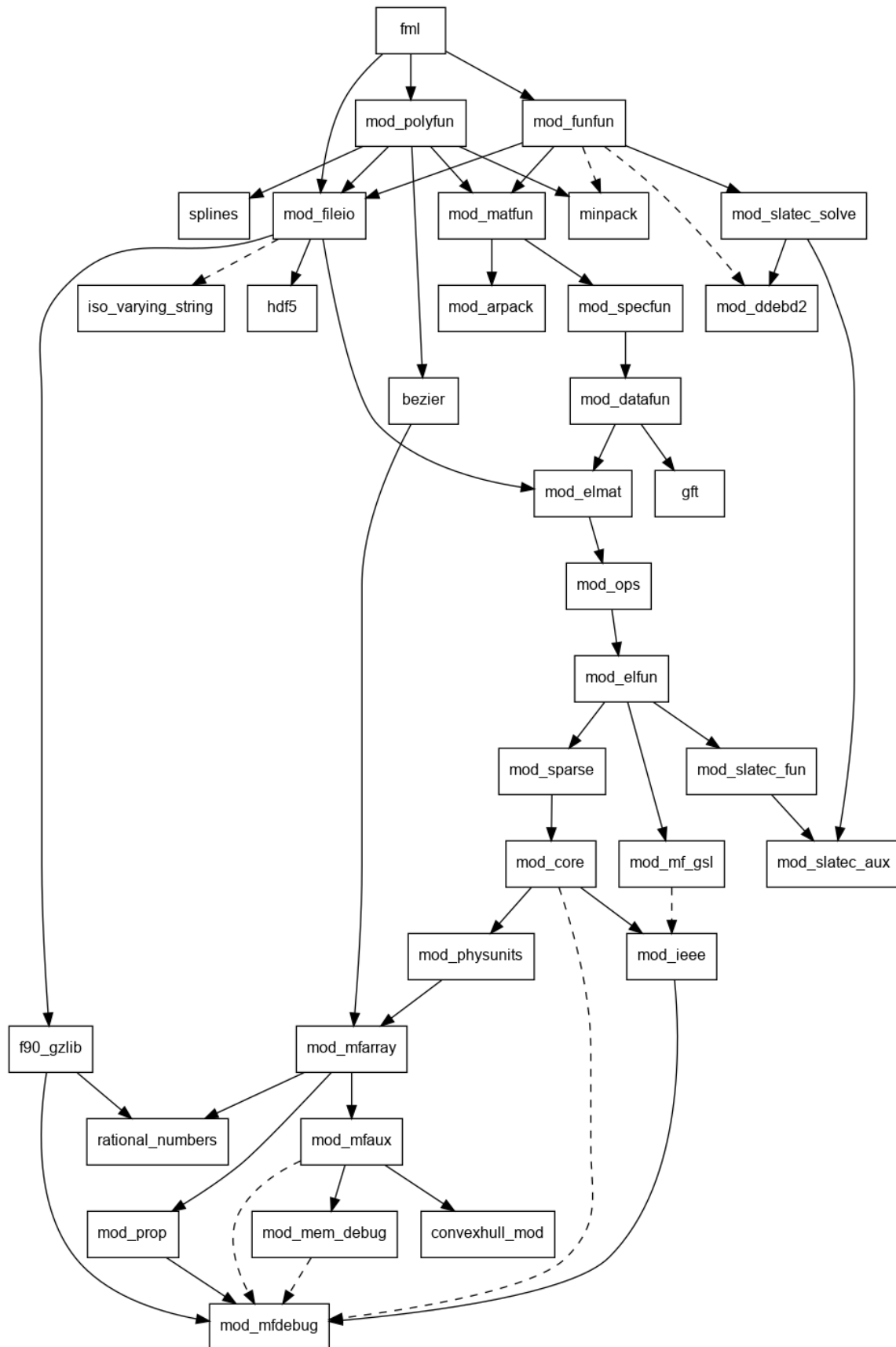


Figure 1: FML modules' graph

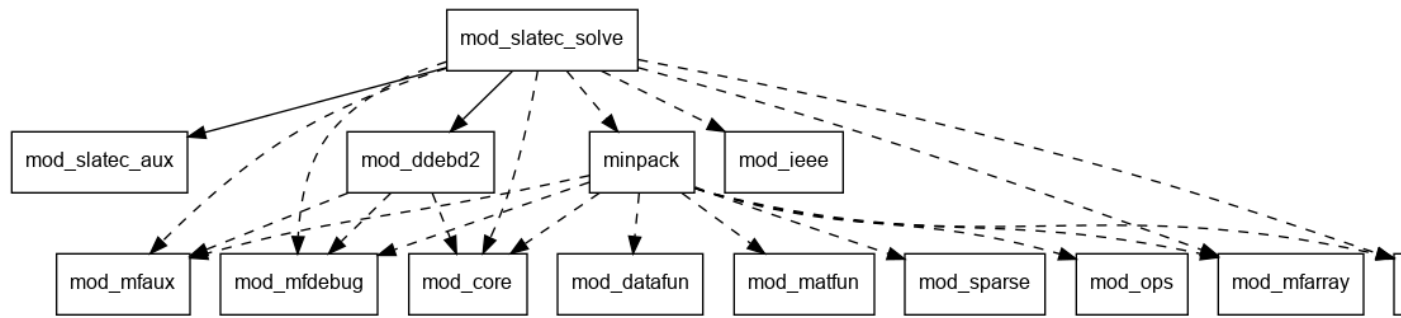


Figure 2: SLATEC modules' graph

Figure 2 shows the whole dependency graph of the pseudo SLATEC's modules. Note that SLATEC is not a true module: it contains only the `ddebd2` module and a collection of files; but these latter files uses many modules.

3 Coupling the Muesli library with a graphic tool

A basic coupling may be implemented by use of the `system` command under Unix. Data are exchanged between the library and the external tool via temporary files (or pipes, which are pseudo files, created by `mkfifo` under linux).

3.1 The example of `meditor`

`meditor` is a small Qt based application, launched by the `msMedit` Muesli routine. The source code of this latter routine is located in the `mod.fileio` module.

Basically, `msMedit` takes one `mfArray` argument (see the *Muesli Reference Manual*) and apply the following steps:

- copy the shape (m,n) of the real (dense or sparse) numerical matrix in the `/tmp/meditor.$(PID).in` file;
- append to the same previous file all the elements of the matrix, under a rectangular layout;
- launch the `meditor` tool via the `system` Unix command, with appropriate arguments (see below);
- once the `meditor` tool is closed by the user, read the output file `/tmp/meditor.$(PID).out` and modify the initial `mfArray`.

The command used to launch `meditor` by the Fortran `msMedit` routine is something like:

```
system("unset SESSION_MANAGER; NO_AT_BRIDGE=1 LC_NUMERIC=en_US.utf-8 meditor
      < /tmp/meditor.19834.in > /tmp/meditor.19834.out")
```

(the previous command should read on one line)

Some remarks:

- ‘`unset SESSION_MANAGER`’ is added to avoid the following warning: “*Qt: Session management error: Could not open network socket*”.
- ‘`NO_AT_BRIDGE=1`’ is added to avoid a long warning message about the *Qt inter-process communicator D-Bus*.
- ‘`LC_NUMERIC=en_US.utf-8`’ is added to be sure that the decimal separator is the dot ‘.’ and not the comma ‘,’ like in french country (type the ‘`locale`’ command in your shell terminal).

Note that the temporary files `/tmp/meditor.*` are not protected: even they are deleted after their use, they can be read by any user on the machine.

Currently, the `meditor` tool can be compiled by the Qt-4 library.



Figure 3: FGL modules' graph

4 Implementation of the graphical part: FGL

The graphics part is based on the PGPLOT library: <http://www.astro.caltech.edu/~tjp/pgplot/> which has been highly modified and extended. Therefore, it has been renamed MFPLLOT.

4.1 Overall modules' graph

Figure 3 show the whole dependency graph of the FGL's modules.

4.2 Predefined colors in MFPLLOT

There are 42 predefined colors in MFPLLOT. The following table shows the sixteen firsts, which were in the old PGPLOT package:

num.	(R,G,B) values	Color name (or description)
0	(0.0,0.0,0.0)	black
1	(1.0,1.0,1.0)	white
2	(1.0,0.0,0.0)	red
3	(0.0,1.0,0.0)	green
4	(0.0,0.0,1.0)	blue
5	(0.0,1.0,1.0)	cyan
6	(1.0,0.0,1.0)	magenta
7	(1.0,1.0,0.0)	yellow
8	0.003922×3	quasi black
9	0.1647×3	very dark gray
10	0.3333×3	dark gray
11	0.5×3	medium gray
12	0.6667×3	light gray
13	0.8314×3	very light gray
14	0.9961×3	quasi white
15	1.0×3	white
...

The eight first colors (0-7) may be selected by one letter, as in Matlab (see the `msPlot` routine): "k", "w", "r", "g", "b", "c", "m", "y".

The next eight colors are for internal usage. An ordinary user cannot select them. Perhaps the color index 15, which is duplicated (same color as index 1), may disappear in future...

Other colors can be selected by their name, via different ways (see the `msSetGrObj` routine).

Three colortables are present in MFPLLOT. There are:

- the old Matlab colortable of 7 colors (R2014a and earlier);
- the new Matlab colortable of 7 colors (starting in R2014b);
- the Breeze colortable of 12 colors (from LibreOffice Draw).

The selection of the colortable (for cycling in colors) may be changed via two ways:

1. via the `msSetColorScheme` routine, for the selected figure. Therefore, one colortable is attached to each figure.
2. via the environment variable `MFPLLOT_COLORS_SCHEME`, which change the default colortable for all following figures.

In both cases, the two variables can take their value among 1 to 4. Default value is 3, *i.e.* the new Matlab colors.

4.3 Image indexing schemes

An image is defined by both an array of pixel value (hereafter called the image array) and a colormap.

Two image indexing schemes are available for the image array:

1. the real valued pixels scheme: (fractional) real values are stored in an `mfArray`; these real values are ranged in $[r_{min}, r_{max}]$. When reading an image file, the latter range is equal to $[0.0, 1.0]$, but this is not mandatory (*i.e.* the user can define as an image any rank-2 array of numerical values). Transparent pixels can be defined by setting some elements to the `NaN` value.
2. the indexed pixels scheme: integer values are stored in an `mfArray` (as numerical values); these indices are ranged in $[1, N]$, where N is the number of colors in the colormap. All indices of the image `mfArray` must have finite values.
Transparent pixels can be defined by setting a color in the colormap to the triplet special value

$(R,G,B) = (-1, -1, -1)$.

Transparent pixels are colored in light gray (predefined color number 14) by the `msImage` routine.

See also information at the `msImRead` and `msImWrite` entries in the *MUESLI Reference Manual*.

4.4 The available devices and their driver

Four different devices are available:

- *X11* for plotting on the screen (opening many windows at a time); Muesli uses the *libXft* library to display antialiased characters.
- *NULL* for computing in a batch mode (it is, of course, a non-interactive device, but it is useful to print figures); currently, not all `opcode` are implemented. Moreover, it has no graphic memory to manage transparency for EPS printing (see, however, the features of *PDF*).
- *EPS* for printing the current figure in an EPS (Encapsulated PostScript File) image.
- *PDF* for printing the current figure in a PDF image. The driver supports natively transparency. It supports also *Optional Contents*, also known as layers.

The corresponding source files can be found in `src/fgl/mfplot/src/drivers`.

All plotting commands are firstly sent either to the *X11* driver, either to the *NULL* driver (in this case, you see nothing). Printing a figure in EPS or PDF format redraw all the graphic commands via the appropriate driver. The resulting image does not depend on the first selected device (*X11* or *NULL*).

Lastly, EPS and PDF images can be built anonymous: the information fields Title, Creator, CreationDate and Author are present or not, according the value of the environment variables `MFLOT_EPS_ANONYMOUS` and `MFLOT_PDF_ANONYMOUS`.

4.5 Managing the graphic memory

In order to be able to redraw all the commands (either after a window resize, or during printing in a file), it is necessary to keep all graphic objects in memory. These *grobjs* are stored in a double linked lists, one for each window.

Each figure (or window) have a set of own properties, grouped in a derived type named `mf_win_info` (see the `mod_win_db` module). Two components, of type pointer, are used to access the above mentioned linked list:

- `grobj_head` is the head of the list of *grobjs*,
- `grobj_tail` is its tail.

Each *grobj* is of a derived type (`grobj_struct`) which contains all the necessary information (see the `mod_win_db` module). It is important to note that this *grobj* structure is embedded in another derived type, named `grobj_elem`:

```
type :: grobj_elem ! used in a double-linked list
  type(grobj_struct)      :: struct
  type(grobj_elem), pointer :: prev => null()
  type(grobj_elem), pointer :: next => null()
end type
```

The two pointer components, named `prev` and `next` are used to link all *grobjs* together, and `grobj_head` points to the first *grobj* (via its `next` component), while `grobj_tail` points to the last *grobj* (via its `prev` component), as depicted in Figure 4.

At the creation of a *grobj*, the code to insert this new element at the end of the linked list is:

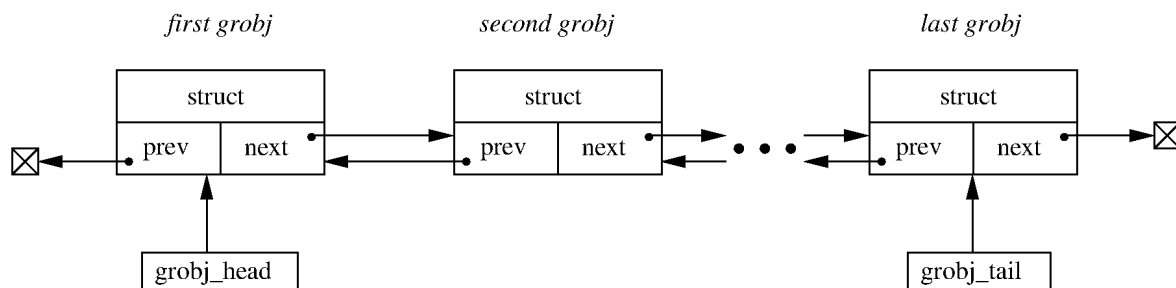


Figure 4: Double linked list of Graphic Objects

```

if( associated(win%grobj_tail) ) then
  grobj%prev => win%grobj_tail
  win%grobj_tail%next => grobj
  win%grobj_tail => grobj
else ! list was empty
  win%grobj_head => grobj
  win%grobj_tail => grobj
end if

```

A unique handle is attributed to each *grobj*. At the user level, it is a big integer, which comes from a short index (named hereafter *hdl*) encoded with the window index. Making a large, non predictable integer avoid the user to manipulate himself *grobjs* by guessing their handle indices.

A dynamic array (*i.e.*, a pointer array, which can be deallocated and reallocated) is used to keep a link (via a pointer) to each *grobj*. The routines:

- `mf_win_handles_init`
- `mf_win_handles_increase_size`
- `mf_win_get_free_handle`

are used to, respectively, allocate the pointer array `win%handles(:)` to a small size, increase its size by doubling the number of its elements, and get an index of a free position (after, *e.g.*, removing a *grobj*).

4.6 Fonts used in FGL

FGL use mainly PostScript Type 1 fonts to display characters on screen, in an X11 window, via the *libXft* library; this latter library produces antialiased characters on the screen.

When printing in *EPS* or *PDF*, FGL uses PostScript fonts. Six of them come from the Adobe standard fonts (Helvetica, Helvetica-Bold, Times, Times-Bold, Times-Italic, Times-Bold-Italic — or their free equivalent provided by all linux distributions) plus a subset of a script font. This latter font (English157BT-Regular) is embedded in the printed files. Marker symbols use a special small PostScript font, created by the author of the Muesli library, named CeMaSP (which means “Centered Markers for Scientific Plot”).

All binary objects (images, fonts) are converted in human readable formats before inclusion in *EPS* or *PDF*. These formats accept the `AsciiHEX` and the `Ascii85` encoding. Doing so, it is easy to modify by hand the printed figures, either to fix something, or for tracking bugs of the FGL drivers.

5 Source organization and Compilation tools

- All the source files are under control revision using the GIT system: <http://git-scm.com/>.
- Compilation is done via GNU-Makefiles. According to the compiler used, many compilation options are proposed during the ‘make’ of the libraries; they are explained via a ‘make help’ call.
- Some special tools have been carefully designed and implemented to avoid the well-known *Compilation Cascade Problem* when recompiling the sources after some modifications.
- Two system cache programs, ‘ccache’ and ‘f90ccache’, are used. The first one has been updated in order to take into account compilers other than GNU-GCC; the second one has been designed (from ideas of ‘ccache’) to keep in cache both object files and precompiled module information files. See: <https://perso.univ-rennes1.fr/edouard.canot/ccache> and <https://perso.univ-rennes1.fr/edouard.canot/f90ccache>.
- For debugging purpose, the setting of some environment variables may be useful to check things inside the created EPS and PDF files:

`MFPLLOT_EPS_COMMENTS` = 0 or 1. If set to 1, comments are included in the EPS file at appropriate location. These comments describe the top-level FGL command used by the user to plot something. Default is 0, *i.e.* not including comments.

`MFPLLOT_PDF_COMMENTS` = 0 or 1. If set to 1, comments are included in the PDF file at appropriate location. These comments describe the top-level FGL command used by the user to plot something. Default is 0, *i.e.* not including comments.

`MFPLLOT_DEFLATE_A85` = 0 or 1. If set to 1, the binary streams used in EPS and PDF files are reencoded using ASCII85, which is more compact than HEXA (used when it is set to 0). Default is 1, *i.e.* use of ASCII85 encoding.

`MFPLLOT_DEBUG` = 0 or 1. If set to 1, some information may be written on the screen (it depends on the current programming status); if set to 2, the X11 drawing buffer (see the `pgbbuf/pggebuf` pair) is disabled⁶. Default is 0, *i.e.* use of drawing buffer and no debug information write.

5.1 On demand traceback

The *MUESLI Reference Manual* describes the `msSetTrbLevel` which allows the user to get the call stack with relevant information (source filename and line number) each time a message is issued from the *MUESLI* library. However, not all the compilers are able to produce this traceback on demand:

- The INTEL ifort compiler provides the `tracebackqq` routine (in the `ifport` module).
- From release 4.8 of GNU GCC, the `backtrace()` subroutine is callable from the user program.

5.2 Debugging tools

A number of tools are used during the development of the libraries FML and FGL (only under a Linux system, not under MinGW-64/Windows):

- `valgrind` (<http://valgrind.org/>) is a very useful tool to detect not only memory leaks (*i.e.* `mfArray` not cleaned), but also use of uninitialized variables. In the latter case, the tool is able to provide additional information (in DEBUG mode) about source lines’ number.
- Enabling *Floating-Point Exceptions* is also useful to detect an erroneous implementation of some routines. This enabling can be done at the top level by using the official MUESLI `msEnableFPE` routine, or inside the code by using direct calls to the appropriate system routines. Be aware that *FPE* trapping is disabled inside most of MUESLI routines; therefore, adding the argument `full.trapping=.true.` is necessary to inspect the behavior of all routines. Pay attention to the following: if the *Lapack* `ieeeck` function is called, then an *FPE* will be raised, leading to a run-time stop. The cure is to link the executable with another version of `ieeeck` which always return 1.

⁶But this facility is restricted to the DEBUG mode of Muesli, not the OPTIM mode.

A couple of internal routines (`mf_save_and_disable_fpe` and `mf_restore_fpe`) are available to the developers to deal with test or internal routines which work with *NaNs*.

5.3 Modified compilers

The GNU Fortran compiler can be modified to clean the output of the traceback (only the file `libgfortran/runtime/backtrace.c`). The traceback feature is systematically used in case of error, but the user may be lost when facing with a long chain of call, most of them located in the system library or the Muesli library itself. The above-mentioned clean concerns the removing of FML routines, but keep the user routines. This can be done on all GNU Fortran compilers from version 4.

A MBF format

The MBF format (Muesli Binary Format) is used to store data in binary files. Whereas an ASCII dump is possible, most of information is lost (in particular the matrix properties); in case of loading, Muesli has to find again these properties, sometime after time-consuming operations.

The MUESLI routines `msSave` and `mfLoad` must be used to deal with these binary files. Besides, the provided `mbfread` and `mbfwrite` may be used under MATLAB to access or create these files.

The proposed extension for this kind of file is “.mbf”, but it is not strictly required.

Each MBF file contain one `mfArray`, in gzipped/not gzipped state, in little/big endian format, of real/complex type with sparse structure as well as dense structure.

MUESLI stores all the internal fields of the `mfArray`, including its dynamic type, structure, matrix properties and physical units.

Most of items are stored using the Fortran-way, *i. e.* framed by the size (in bytes) of the data, using 4 bytes. Therefore, it is *de facto* limited to array of size up to 2 GB (to be verified... and how write bigger file?)...

An MBF file contains:

- the string "MF-2.5 ECanot CNRS " (without the double quotes), using 20 bytes; this string is written byte after byte, so it is always stored in the right way, irrespective of the endianness;
- the string "1234", using 4 bytes; as opposed to the previous string, it is stored as "1234" for little endian, and as "4321" for big endian;
- the data type of the `mfArray`, as described in section 2.2.1, stored as integer using 4 bytes;
- the shape of the `mfArray`, stored as a couple of 4-byte integers;
- the double precision elements of the `mfArray`, each stored as 8-byte reals (complex values using twice this size);
- the properties of the matrix.

B HDF5 format

Many `mfArrays` can be stored in a single HDF5 file, but no hierarchy can be defined: all the `mfArrays` are located at the “root” of the HDF5 file.

Each `mfArray` is defined as an HDF5 “group”; numeric data is stored in a dataset (actually, one dataset for each array or vector defining the matrix) whereas the matrix properties (see the `prop` field of the `mfArray` derived type) are all defined as attributes of the group.

Fortran defines the storage of numeric data in a 2D array by column, whereas C defined this storage by row. For this reason, matrices appear as transposed if they are loaded by a C library. This is the case of the *hdfview* tool.