MUESLI

Reference Manual

Fortran implementation

Release 2.22.1

Édouard Canot*

May 19, 2025

*IPR/CNRS, Rennes, France

Contents

Introduction

This document describes in details the MUESLI Fortran library routines.

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 Inside*

The source code of MUESLI comes with many examples which can also give some help.

MUESLI is split in two parts, which correspond to the following Fortran modules:

- Basically, FML (Fortran Muesli Library) contains all necessary materials to numerically work with a dynamic array (dynamic in size, type and structure), called `mfArray`; all available routines are described in the first section of this guide.
  To work with FML, your Fortran source code must include the statement:

  ```
  use fml
  ```

- FGL (Fortran Graphics Library) contains graphic routines which use the `mfArray` objects; these routines are described in the second section of this guide.
  To work with FGL, your Fortran source code must include the statements:

  ```
  use fml
  use fgl
  ```


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

Typographic convention

In this guide, the colored brackets ('[' and ']') indicate some optional arguments or other things; similarly, the colored pipe ('|') specifies alternates in a choice. These symbols are used only in interface definition of the routines.

Coloring is also used to differentiate source code and execution output. Lines from source code will be always displayed in this color, whereas input/output in a terminal will be always displayed in this color.

For sake of clarity, array constructors use black brackets '`[`' and '`]`' (Fortran 2003) instead of '`(/`' and '`/)`' (Fortran 95).

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 Beata Wojciechowska. It can be used under a Limited Royalty Free License (see http://www.dreamstime.com/breakfast-imagefree545439).

1 FML: Numerical Library

FML contains all routines to work with matrices and to do some linear algebra.

Routines beginning with '`mf`' are functions, whereas those beginning with '`ms`' are subroutines. Functions are used to return only one object (*e. g.* one `mfArray`); subroutines can also return many objects, but always via the `mfOut` facility.

FML also defines some new derived types: `mfArray`, `mfUnit`, `mf_Out`, `mfMatFactor`, `mfTriConnect`, `mfTetraConnect`, `mf_DE_Options`, `mf_NL_Options`.

Besides, FML defines public constant entities:

- an `integer` parameter:

  `MF_DOUBLE`

  which defines the working precision in MUESLI. In the current version, it stores the kind of double precision reals. In all this document, when Fortran numerical kind is not specified, `real` and `complex` stands for `real(kind=MF_DOUBLE)` and `complex(kind=MF_DOUBLE)`, respectively.

- some `mfArray` parameters:

  `MF_EMPTY`, `MF_COLON` (and its alias `MF_ALL`), `MF_END`, `MF_NO_ARG`

  which are, respectively, an empty `mfArray`, the '`:`' pseudo-operator (as in Fortran 90 or MATLAB, but specific to FML), an automatic index for pointing to the end of a dimension of an `mfArray` and a pseudo-keyword to tell that an argument is not present in `mfOut`.

- some `real(kind=MF_DOUBLE)` parameters:

  ```
  MF_PI
  MF_EPS
  MF_INF
  MF_NAN
  MF_E
  MF_REALMAX
  MF_REALMIN
  ```

  and some useful mathematical auxiliairy constants:

  ```
  MF_BESSEL_J0_ROOTS(:)
  MF_BESSEL_J1_ROOTS(:)
  ```

  (30 first non-zero roots for both Bessel functions J_0 and J_1, stored in increasing order)

- one `complex(kind=MF_DOUBLE)` parameter:

  `MF_I`

- and the `character` parameters:

  ```
  MF_MUESLI_VERSION
  MF_COMPILER_VERSION
  ```

  which store, respectively, the MUESLI library version and the compiler vendor and version. The boolean `mfIsVersion` routine must be used in order to compare each of these parameters with a given version number. Besides, other character functions may be used to retrieve additional information: `MF_LAPACK_VERSION()` (the Blas/Lapack library version) and `MF_COMPILATION_CONFIG()` (the configuration used to compile the whole Muesli library: *Debug* or *Optim*).

`MF_INF` and `MF_NAN` contain special IEEE values (*resp.* Infinity and Not-a-Number).

FML also defines the following global variables:

- `integer` variables:

  ```
  STDERR
  STDIN
  STDOUT
  ```

  which can be modified either directly by the user, or via the `msSetStdIO` routine.

- `logical` variables: `MF_NUMERICAL_CHECK` which allows the user to tell Muesli to do additional checks.

The available routines have been arranged into sub-parts:

Core Routines

File Input/Output

Data Analysis Functions

Operators

Elementary Math Functions

Specialized Math Functions

Elementary Matrix Manipulation Functions

Matrix Functions

Polynomial Functions

Optimization and Function Functions

Sparse Matrices.

1.1 Core Routines

mfArray	automatic and dynamic array *(derived type)*
mf	mfArray conversion
=, msAssign	mfArray assignment
msSet, mfGet	mfArray data modification and extraction
mfCount	true values count
msDisplay	mfArray pretty print
mfDisplayColumns	columns used in pretty print
msFormat	mfArray printing format
msRelease	MUESLI objects deallocation
msAutoRelease	mfArray conditional deallocation
mfOut	groups output arguments
mf_Out	set of output arguments *(derived type)*
msPointer, msFreePointer	smart pointer between f90 array and mfArray
mfNbPointers	nb of f90 pointers pointing to an mfArray
msEquiv	smart pointer between mfArray and f90 array
mf_Int_List	list of integers *(derived type)*
mf_Real_List	list of reals *(derived type)*
mfIsEmpty	mfArray empty test
mfIsEqual, mfIsNotEqual	mfArray equality test
mfIsLogical	mfArray boolean test
mfIsReal	mfArray real test
mfIsComplex	mfArray complex test
mfIsNumeric	mfArray numeric test
mfIsDense	mfArray dense storage test
mfIsSparse	mfArray sparsity test
mfIsScalar, mfIsVector, mfIsMatrix	mfArray scalar, vector and matrix test
mfIsRow, mfIsColumn	mfArray kind of vector test
mfIsPerm	mfArray permutation vector test
All, Any	test on boolean mfArray
Shape, mfShape	shape of an mfArray
Size, mfSize	size of an mfArray
mfInt	scalar integer conversion
mfDble	scalar real conversion
mfCmplx	scalar complex conversion
mfGetMsgLevel, msSetMsgLevel	message level tuning
mfGetTrbLevel, msSetTrbLevel	error traceback tuning
msSetColoredMsg	colorize Muesli messages on terminal
msPrintColoredMsg	print a colored user message on terminal
msSetTermColor	set color for printing on terminal
msMuesliTrace	helper for debugging purpose
msGetStdIO, msSetStdIO	usual logical unit inquiry and modification
msFlush	I/O flush
msPause	user pause or timing pause
msSetTermWidth, mfGetTermWidth	set and get terminal character width
msSetAutoFilling, mfGetAutoFilling	set and get out-of-range filling by msSet
msInitArgs, msFreeArgs	mfArray arguments' protection and release
msSetAsParameter	data protection
msReturnArray, mfIsTempoArray	mark and check temporary mfArray
msFlops, mfFlops	nb of floating-point operations
mfIsFlopsOk	inquire if flops is available

mfGetAutoComplex, msSetAutoComplex	conversion to complex
msEnableFPE, msDisableFPE	run-time floating-point exceptions trapping
mfGetRoundingMode, msSetRoundingMode	floating-point rounding mode
MF_NUMERICAL_CHECK	debugging additional checks
mfUnit	physical unit *(derived type)*
msUsePhysUnits	Physical units activation
msSetPhysDim	set physical unit
msSetPhysUnitAbbrev	set user physical unit abbrevation
mfHasNoPhysDim	dimensionless test
mfHaveSamePhysDim	dimension equality test
msPrepHashes	hashes print preparation
msPrintHashes	hashes print
msPostHashes	hashes print conclusion
msPrepProgress	percent progress preparation
msPrintProgress	percent progress
msPostProgress	percent progress conclusion
mfReadLine	read a line from terminal with editing and history facilities
msReadHistoryFile	read a 'readline' history file
msWriteHistoryFile	write the 'readline' history in a file
msClearHistory	clear the 'readline' history
msAddEntryInHistory	add an entry in the 'readline' history
msRemoveLastEntryInHistory	remove last entry in the 'readline' history
mfToLower	lowering string's case
mfToUpper	uppering string's case
msFindIOUnit	automatically find a free IO unit number
mfIsVersion	test on version strings
msRequMuesliVer	check for a minimum Muesli version
MF_MUESLI_VERSION	MUESLI version
MF_COMPILER_VERSION	COMPILER version
MF_COMPILATION_CONFIG	Configuration used to compile MUESLI

See also:

File Input/Output

Data Analysis Functions

Operators

Elementary Math Functions

Specialized Math Functions

Elementary Matrix Manipulation Functions

Matrix Functions

Polynomial Functions

Optimization and Function Functions

Sparse Matrices

`mfArray` automatic and dynamic array *(derived type)*

Description:

This is the fundamental derived type in MUESLI. Declaration must be made as follows:

```
type(mfArray) :: A
```

This derived type may contain real or complex numerical values, using a dense or sparse storage. All numerical values are stored in *double precision*, *i. e.* use 8 bytes of memory.

Special data types are:

- permutation vector (integer vector)
- boolean values (in dense storage only)

All possible data types are sumarized in figure 1.

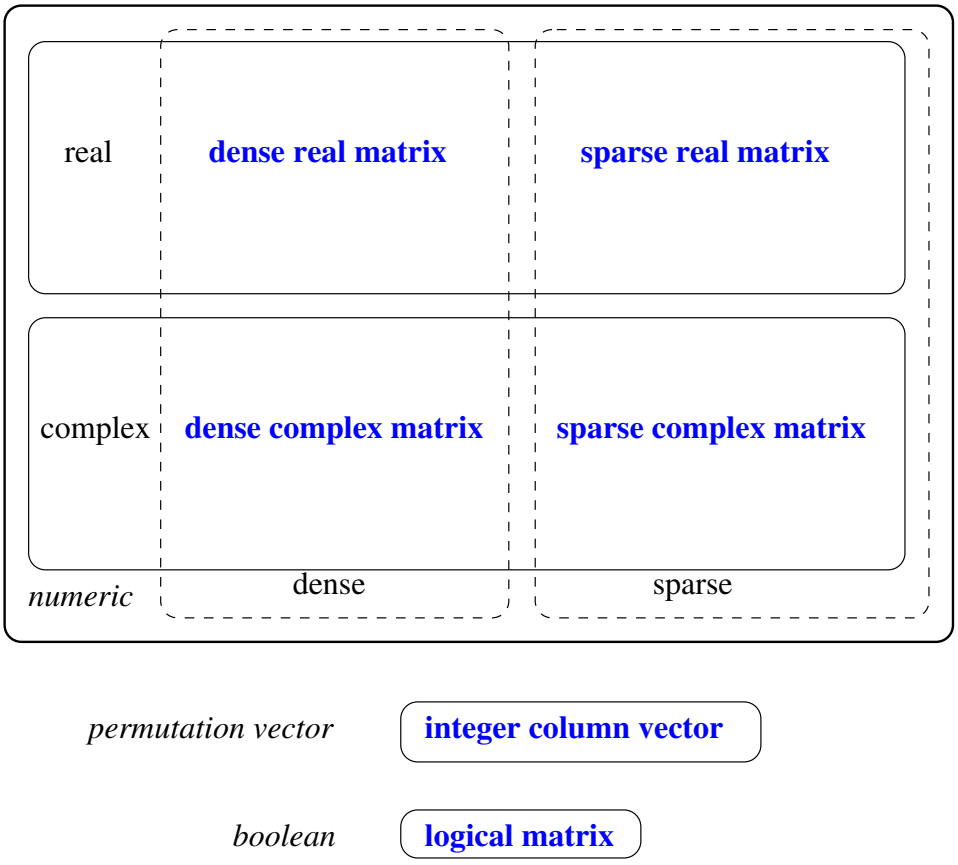


Figure 1: mfArray data types

The data type stored in a `mfArray` may be retrieved by the following inquiring functions: `mfIsReal`, `mfIsComplex`, `mfIsDense`, `mfIsSparse`, `mfIsNumeric`, `mfIsPerm`, `mfIsLogical`.

See also: `mf`, `msRelease`, `msPointer`, `msEquiv`, `msSaveAscii`

`mf` — mfArray conversion

Description:

This function initializes and returns a temporary `mfArray` from some common Fortran types: integer, real (single or double), complex (single or double), of rank 0 to 2 (scalar, vector or matrix).

For example:

```
mf( [ 1, 3, 5, 7, 11 ] )

mf( [ 1.3, 2.05, -5.1e-3 ] )
```

Whatever the argument type of `mf` (*integer*, *single precision real*, *double precision real*), all numerical numbers are stored in `mfArrays` as `double precision`.

By default, this function creates row vectors, so use the following syntax if you want a column vector (avoiding an unnecessary copy):

```
mf( [ 1, 3, 5, 7, 11 ], transpose=.true. )
```

This latter optional transposition works also for matrices. By default, the transposition is not used (see examples below).

See also: `mfArray`

Example(s):

```
real(kind=MF_DOUBLE) :: tab(2,4)
tab(:,:) = reshape( [(1.5d0*i,i=1,8)], [2,4] )
A = mf( tab )
call msDisplay( A, "A" )
At = mf( tab, transpose=.true. )
call msDisplay( At, "At" )
```

output:

```
A =

   1.5000    4.5000    7.5000   10.5000
   3.0000    6.0000    9.0000   12.0000


At =

   1.5000    3.0000
   4.5000    6.0000
   7.5000    9.0000
  10.5000   12.0000
```

= mfArray assignment

This Fortran operator has been overloaded in MUESLI to make working with `mfArray` easy.

For example, you can assign any numerical object to A, as simply as:

```
A = [ 1, 2, 3, 4, 5 ]
```

The right hand side (*RHS*) can be of type boolean, integer, real (single or double) or complex (single or double), of rank 0 to 2 (scalar, vector or matrix), or of course `mfArray`.

As the allocation of data is automatic during assignment, you should free each `mfArray` as soon as possible (via the `msRelease` routine), in order to decrease the memory usage.

Sometimes, you could be warned that an assignment is inefficient, when the *RHS* is a temporary `mfArray`. In such a case, you could use the `msAssign` routine instead.

Besides, MUESLI allows to assign a dense `mfArray A` to a boolean, an integer, a real or a complex. So, a statement like:

```
f90_obj = A
```

is valid under the following assumptions:

- shapes (of *LHS* and *RHS*) must match;
- types must be consistent (*i. e.* numeric mfArrays can be assigned to integer or real f90 arrays whereas boolean mfArrays must be assigned to logical f90 arrays — mfArrays data types are defined in figure 1).

See also: `msPointer`, `msEquiv`

`msAssign` more efficient mfArray assignment

Interface:

```
subroutine msAssign( dest, source )

   type(mfArray), intent(in) :: source
   type(mfArray)             :: dest
```

Description:

Ordinarily, you should use the simple `=` form of assignment. The `msAssign` routine is intended to make it efficiently, when the *RHS* is a temporary `mfArray`.

For example, you should write:

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

which avoids a copy of the data contained in `A + B` (an internal pointer is used instead). In comparison, the statement:

```
C = A + B
```

involves such a copy.

See also: `=`

`msSet` — mfArray data modification

Generic Calling syntax:

```
call msSet( data, x, a1 [, a2] )
```

Description:

This routine is used to modify some elements of the `mfArray x`. The indices of these elements are specified by `a1` and `a2` for array sections, or only by `a1` for scattered elements in the whole array (in this case, `a1` is a long-column-index vector).

If `a2` is not present, the `mfArray x` may be a vector, because only one index is given to specify which elements have to be changed. Note that if the index `a1` (or `a2`) is out of range, the shape of `x` is modified accordingly, *i. e.* the array is extended (see the remarks at the end of this section).

If `a1` (or `a2`) is a vector, more than one element can be modified; moreover, this vector doesn't need to contain consecutive indices.

If the modification affects more than one element, and if `data` contains only one value, this latter value is spread across all the target elements.

`data` may be of type `real`, `complex` or `mfArray`. It doesn't need to be of same data type as `x`. A conversion is then made, and concerns either `data` or `x`, according to the type.

Usually `a1` and `a2` are integers (scalar or vector), but they may be of type `mfArray` (in this latter case, they cannot be matrices).

In the special case where `a1` (or `a2`) is `MF_COLON` (or its alias `MF_ALL`), then all the specified row(s) (or column(s)) are simultaneously modified. If `data` is empty (*i. e.* equal to `MF_EMPTY`, but only for a dense `mfArray x`), then the specified row(s) or column(s) are deleted.

`a1` and/or `a2` may be also special integer sequences, written under the form

```
    i1 .to. i2 [.by. i3]
or  i1 .by. i3 .to. i2

[ ( i, i = i1, i2 [, i3] ) ]
```

(The operator named '`.step.`' is an alias for '`.by.`')

These special integer sequences can be combined as

```
    int_seq .and. int_seq
or  int_seq .but. i1
```

The step `i3` is taken equal to 1 if it is not present. Note that `a1`, `a2`, `i1` and `i2` may also be replaced by an expression using `MF_END` (*e. g.* `MF_END-1`) which is an automatic alias for the end value of the corresponding dimension. Be aware that arithmetic involving `MF_END` can use only addition or substraction.

Finally, the `mfArray x` cannot be temporary.

···/···

Remarks:

- to initialize a whole `mfArray` with 0 or 1, the following routines are available: `mfZeros` and `mfOnes`.
- when the `mfArray x` has a sparse structure, `data` may be a scalar or a vector (dense or sparse). In such a case, only whole columns of `x` may be modified. Modifying one element is however allowed, in so far as indices of this element are not out of range (see below, however).
- when the `mfArray x` has a dense structure, `data` cannot be sparse.
- not all combinaisons for the types of `a1` and `a2` are valid. Use preferently the same type for these two arguments.
- out-of-range indices are accepted, except when a long-column index is used. For the dense storage, the `mfArray` is reallocated to the right shape. A message of kind "info" is however emitted (see `msSetMsgLevel`). During the reallocation, the new elements not targeted are initialized by a predefined value (see below). For the sparse storage, the assignment is done only if the value of `nzmax` is sufficient (see `msSpReAlloc`), and, of course, no new elements are added, apart those explicitly targeted by the command.
- the predefined value used to fill the out-of-range indices is by default the *NaN* (special *IEEE* number), but this can be changed by use of the `msSetAutoFilling` routine.

See also:

`mfGet`

`mfGet` — mfArray data extraction

Generic Calling syntax:

```
mfGet( x, a1 [, a2] )
```

Description:

This routine extracts some elements (described by the indices `a1` and `a2`, or — when `x` is a matrix — only by the long-column index `a1`) of the `mfArray x` and returns them in an `mfArray`. Note that if the index `a1` (or `a2`) is out of range, an error results and an empty `mfArray` is returned.

If `a1` (or `a2`) is a vector, more than one element are extracted; moreover, this vector doesn't need to contain consecutive indices.

Usually `a1` and `a2` are integers (scalar or vector), but they may be of type real or boolean `mfArray`.

In the special case where `a1` (resp. `a2`) is `MF_COLON` (or its alias `MF_ALL`), then all the specified columns (resp. rows) are extracted.

`a1` and/or `a2` may be special integer sequences, written under the form

```
   i1 .to. i2 [.by. i3]
or i1 .by. i3 .to. i2

[ ( i, i = i1, i2 [, i3] ) ]
```

(The operator named '`.step.`' is an alias for '`.by.`')

These special integer sequences can be combined as

```
   int_seq .and. int_seq
or int_seq .but. i1
```

The step `i3` is taken equal to 1 if it is not present. Note that `a1`, `a2`, `i1` and `i2` may also be replaced by an expression using `MF_END` (*e. g.* `MF_END-1`) which is an automatic alias for the end value of the corresponding dimension. Be aware that arithmetic involving `MF_END` can use only addition or substraction.

Remarks: Sparse `mfArray`s are partially supported:

- this routine can extract a scalar value, a row or a column, or a submatrix; in this latter case, the submatrix must be contiguous;
- when extracting rows, the indices for these rows must be contiguous (this limitation doesn't hold when extracting columns);
- the above mentioned special integer sequences (built via the `.to.` and `.by.` operators) cannot be used (not yet) for indices `a1` or `a2`;
- when extracting a single row or a single column, resulted `mfArray` has a dense structure.

See also: `msSet`

`mfCount` — true values count

Interface:

```
function mfCount( A, dim ) result( out )

   type(mfArray), intent(in) :: A
   integer, intent(in), optional :: dim
   type(mfArray) :: out
```

Description:

Returns the number of true values in the columns (or rows) of a boolean `mfArray` (dense or sparse).

If `dim` is present, `A` is always considered as a matrix, and the routine processes each column (if `dim` = 1) or each row (if `dim` = 2).

If `dim` is not present, it returns the number of true values in the whole array.

When `out` is a vector, it is dense.

The behavior of this routine is similar to the *count* Fortran 90 intrinsic function.

Example(s):

```
x = mfMagic( 3 )
call msDisplay( x, "x" )
call msDisplay( mfCount(x>5.0d0), "mfCount( x > 5.0d0 )" )
call msDisplay( mfCount(x>4.0d0,dim=1), "mfCount( x > 4.0d0, dim=1 )" )
call msDisplay( mfCount(x>4.0d0,dim=2), "mfCount( x > 4.0d0, dim=2 )" )
```

output:

```
x =

    8     3     4
    1     5     9
    6     7     2


mfCount( x > 5.0d0 ) =

    4


mfCount( x > 5.0d0, dim=1 ) =

    2     1     1


mfCount( x > 4.0d0, dim=2 ) =

    1
    2
    2
```

msDisplay — mfArray pretty print

First calling syntax:

```
call msDisplay( x1 [, string1, x2, string2, ...]                    &
                [, unit ] [, head ] [, tail ] )
```

This routine displays an mfArray object (and other object types, as described below) on the screen or terminal, in a pretty form which is column oriented. Not all the decimal digits are printed: their number depends on the format used.

The object x may also be: a Fortran type, integer, real (single or double), complex (single or double), of rank 0 to 2 (scalar, vector or matrix); a MUESLI derived type, mfUnit and mf_Int_List.

Only for mfArrays:

- up to four pairs (x, string) can be simultaneously displayed.
- if the optional integer argument head (resp. tail) is present, then only the first (resp. last) rows will be printed. These two arguments can be both present. This convenience is useful to see the begin and the end of a very long column vector. For the case of a long row vector, it will display the head first values, followed by the tail last values, but only if the width of your terminal is sufficient. Internal tests are made about consistency between head, tail and the shape of the array.

For mfArrays (not boolean) and mfUnits, the optional argument unit allows the user to specify another consistent (*i. e.* of same physical dimension) unit for printing. For example, a length registered in meter (S.I. unit) may be printed in millimeter. The use of the constant parameter SI_unit enforces the routine to display, if possible, the appropriate S.I. unit.

Only for mf_Int_Lists and vectors of mf_Int_List:

- only one pair (x, string) can be displayed at a time.
- an additional optional boolean argument (compact) may be used to print the integers rows more compactly.

Second calling syntax:

```
call msDisplay( x, string, legend1 [, legend2, ..., legend9 ] )
```

This latter syntax enables the insertion of legends at the head of each column. Note the maximum number possible, 9, and also that it is restricted to real data in the x mfArray.

N.B.: When displaying an object having a great number of columns, this routine will print first the number of columns adapted to the width of your terminal. If the output is redirected to a file, you may want to change this width: then use the MF_COLUMNS environment variable (it is the terminal character width, like in the UNIX command resize, and not the matrix column's number).

See also: msFormat, mfDisplayColumns

Example(s): Many outputs from this routine are shown as examples in the *MUESLI User Guide*.

`mfDisplayColumns` columns used in pretty print

Interface:

```
function mfDisplayColumns( A, unit ) result( out )

   type(mfArray), intent(in) :: A
   type(mfUnit), intent(in), optional :: unit
   type(mfArray) :: out
```

Description:

`mfDisplayColumns` returns the number of elements printed by the routine `msDisplay` on each row.

See also: `msDisplay`

msFormat — mfArray printing format

Calling syntax:

```
call msFormat( [ mantissa, exponent ] )
```

The optional argument **mantissa** may be **"short"** (5 digits printed), **"long"** (15 digits printed) or **"hex"** (hexadecimal output). By default, the short format is used.

The optional argument **exponent** may be:

- **"auto"** (automatic format: fixed or exponential format);
- **"sci"** (scientific format: exponent is always printed);
- **"eng"** (engineer format: if the **mfArray** to be displayed is not a scalar, a scale factor is always used and its exponent is moreover a multiple of three).

By default, the automatic format is used.

In the case where this routine is called without any argument, default values for both keywords are used.

See also: msDisplay

Example(s):

```
x = .t. mfSqrt( mf([ 2, 3, 5 ]) )
call msDisplay( x, "x (short format)" )
call msFormat( "long" )
call msDisplay( x, "x (long format)" )
```

output:

```
 x (short format) =

    1.4142
    1.7321
    2.2361

 x (long format) =

   1.41421356237310
   1.73205080756888
   2.23606797749979
```

```
x = [ 1.5, 2.6E1, 3.7E2, 4.8E4 ]
call msFormat( exponent="sci" )
call msDisplay( x, "x (scientific format)" )
call msFormat( exponent="eng" )
call msDisplay( x, "x (engineer format)" )
```

output:

```
 x (scientific format) =

  1.5000E+00  2.6000E+01  3.7000E+02  4.8000E+04

 x (engineer format) =

   1.0E+03 *

    0.0015    0.0260    0.3700   48.0000
```

`msRelease` — MUESLI objects deallocation

When you have finished to work with an `mfArray`, you may deallocate its internal array by using:

```
call msRelease( x1 [, x2, x3, x4, x5, x6, x7 ] )
```

Up to seven `mfArrays` can be simultaneously released.

Remark: After execution of your program, some compilers or tools may report a warning about some memory leaks. Use of this routine for all of the automatic variables (described below) avoids such warnings.

This routine can be used to free the following derived types: `mfArray`, `mfMatFactor`, `mfTriConnect`, `mfTetraConnect`, `mf_Int_List`, `mf_NL_Options` and `mf_DE_Options` (only one variable may be freed at a time, except for an `mfArray` object).

See also: `msAutoRelease`, `msSetAsParameter`, `msExitFgl`

`msAutoRelease` — mfArray conditional deallocation

This routine is similar to `msRelease`:

```
call msAutoRelease( x1 [, x2, x3, x4, x5, x6, x7 ] )
```

but the deallocation is done only if the `mfArray` is temporary.

Is is used only in the context of a user-routine which takes arguments of type `mfArray` which can be temporary objects.

See also: `msRelease`

`mfOut` group output arguments

This function is used to group output arguments in calls of some MUESLI subroutines. For example:

```
call msLU( mfOut(L,U,p), A )
```

This facility is only intended to clearly identify the output arguments from the input ones. It is reminiscent of the Matlab syntax:

```
[L,U,p] = lu( A )
```

Arguments of `mfOut` match always the following MUESLI derived types: `mfArray` or `mfMatFactor`.

Case of optional arguments: If some of these arguments are optional, they must be called in order, not by use of keyword as in Fortran 90 syntax. Moreover, for multiple optional arguments, one of them can be omitted by using the special `mfArray MF_NO_ARG`, as in the following example:

```
call msOdeSolve( mfOut( y, status, tolout, MF_NO_ARG, solve_log),          &
                 deriv, t_span, y_0 )
```

Indeed, refering to the `msOdeSolve` routine, the `yp` and `solve_log` are both optional, and if the user wants to retrieve only the second one, he has to use `MF_NO_ARG` instead of `yp`.

Remarks: actually, the `mfOut` function returns an object of type `mf_Out`. You don't need to use this latter derived type unless you want to define yourself new routines which use `mfOut` to group output arguments.

See also: `mfArray`, `mfMatFactor`, `mf_Out`

`mf_Out` set of output arguments *(derived type)*

Description:

This is a derived type in MUESLI. You need to employ this type only if you wish to define yourself subroutines whose output arguments are grouped with the `mfOut` function.

See also: `mfArray`, `mfOut`

`msPointer` smart pointer between f90 array and mfArray

Generic Interface:

```
subroutine msPointer( A, f90_ptr )

   type(mfArray) :: A

   real(kind=MF_DOUBLE), pointer :: f90_ptr(:,:)
or complex(kind=MF_DOUBLE), pointer :: f90_ptr(:,:)
```

Description:

This routine is approximately equivalent to:

```
f90_ptr => A
```

and then, you can access the internal data of `A` via a classical Fortran array. This association is not always possible, because the types of `A` and `f90_ptr` must match. Ranks however doesn't need to match: you may have a rank-1 array pointing to a rank-2 array; in this latter case, we could see `f90_ptr` as a "long column pointer".

`A` cannot be temporary, because at any time it can be deleted, and then `f90_ptr` would become hangling.

Warning:

During all the time when `f90_ptr` points to `A`, some internal properties of `A` are locked. So, it is strongly recommended to use the routine `msFreePointer`: (i) to nullify the pointer `f90_ptr` and (ii) to unlock properties of `A`. Be aware that

```
f90_ptr => null()
```

is not sufficient (this way will not unlock the internal `mfArray` properties of `A`).

See also: `mfNbPointers`, `msFreePointer`, `msEquiv`

`msFreePointer` — smart pointer release

Generic Interface:

```
subroutine msFreePointer( A, f90_ptr )

   type(mfArray) :: A

   real(kind=MF_DOUBLE), pointer :: f90_ptr(:,:)
or complex(kind=MF_DOUBLE), pointer :: f90_ptr(:,:)
```

Description:

This routine releases the link made by `msPointer`, and unlocks the internal properties of `A`.

See also: `mfNbPointers`, `msPointer`

`mfNbPointers` nb of f90 pointers pointing to an mfArray

Generic Interface:

```
function mfNbPointers( A ) result( n )

   type(mfArray) :: A
   integer       :: n
```

Description:

This routine returns the number of Fortran 90 pointers pointing to `A`.

See also: `msPointer`, `msFreePointer`

msEquiv — smart pointer between mfArray and f90 array

Generic Interface:

```
subroutine msEquiv( f90_array, A )

   real(kind=MF_DOUBLE), target :: f90_array(:,:)
or complex(kind=MF_DOUBLE), target :: f90_array(:,:)

   type(mfArray) :: A
```

Description:

This routine is approximately equivalent to

```
A => f90_array
```

but the `mfArray A` becomes *restricted*, in the sense that you can modify the data but not the shape of your `f90_array`, nor its allocation status. As a consequence, you can use most of (but not all) MUESLI routines to work with your `f90_array`, via the `mfArray A`.

When `f90_array` is a rank-1 array (here, renamed as `f90_vector`), the `mfArray A` is by default a column vector. Use the optional argument `new_shape` to specify the new virtual dimensions of `A`, as follows:

```
call msEquiv( f90_vector, A, new_shape=[n1,n2] )
```

Of course, you must have consistance, *i. e.* the product $n_1\, n_2$ must be equal to the number of elements of `f90_vector`.

Remark:

To release the link between `A` and `f90_array`, you *must* use the `msRelease` routine. Don't write

```
A => null()
```

because usually `A` is not a pointer, it is just a derived type, *i. e.* a structure which includes the numerical data in a manner that is hidden to the user. Even if `A` has been declared as a pointer to an `mfArray`, this is not the good way to release properly `A`). If you forget to release this link, you could obtain an error when trying an assignment with `A`.

See also: `msPointer`

`mf_Int_List` — list of integers *(derived type)*

Description:

This is a derived type in MUESLI. You may employ it for storing a list of integers of arbitrary length. It's definition is:

```
type :: mf_Int_List
   integer, allocatable :: list(:)
end type mf_Int_List
```

You must allocate yourself the internal component `list`, which is an ordinary Fortran array of integers. Also, you may declare and use an array of `mf_Int_List`, which is useful to work with a set of varying length lists (see the second example below).

`msDisplay` and `msRelease` routines may be used to, respectively, print and free this derived type. These two routines can be also used with an array of `mf_Int_List`.

Example(s):

```
! type(mf_Int_List) :: int_list
allocate( int_list%list(7) )
int_list%list(:) = [ (i, i = 1, 7) ]
call msDisplay( int_list, "int_list")
! call msRelease( int_list )
```

output:

```
int_list =

 1  2  3  4  5  6  7
```

```
! type(mf_Int_List) :: int_list_vec(5)
n = 6
allocate( int_list_vec(1)%list(n) )
int_list_vec(1)%list(:) = [ (i, i = 1, n) ]
n = 15
allocate( int_list_vec(3)%list(n) )
int_list_vec(3)%list(:) = [ (i, i = 100+1, 100+n) ]
n = 9
allocate( int_list_vec(5)%list(n) )
int_list_vec(5)%list(:) = [ (i, i = 100000+1, 100000+n) ]
call msDisplay( int_list_vec, "int_list_vec")
! call msRelease( int_list_vec )
```

output:

```
int_list_vec =

 1  2  3  4  5  6

 <EMPTY>

 101  102  103  104  105  106  107  108  109  110  111  112  113  114  115

 <EMPTY>

 100001  100002  100003  100004  100005  100006  100007  100008  100009
```

`mf_Real_List` — list of reals *(derived type)*

Description:

This is a derived type in MUESLI. You may employ it for storing a list of reals of arbitrary length. It's definition is:

```
type :: mf_Real_List
   integer, allocatable :: list(:)
end type mf_Real_List
```

You must allocate yourself the internal component `list`, which is an ordinary Fortran array of reals. Also, you may declare and use an array of `mf_Real_List`, which is useful to work with a set of varying length lists (see the second example below).

`msDisplay` and `msRelease` routines may be used to, respectively, print and free this derived type. These two routines can be also used with an array of `mf_Real_List`.

Example(s):

```
! type(mf_Real_List) :: real_list
allocate( real_list%list(7) )
real_list%list(:) = [ (i+0.5d0, i = 1, 7) ]
call msDisplay( real_list, "real_list")
! call msRelease( real_list )
```

output:

```
 int_list =

  1.500  2.500  3.500  4.500  5.500  6.500  7.500
```

```
! type(mf_Real_List) :: int_real_vec(5)
n = 6
allocate( real_list_vec(1)%list(n) )
real_list_vec(1)%list(:) = [ (i+0.5d0, i = 1, n) ]
n = 11
allocate( real_list_vec(3)%list(n) )
real_list_vec(3)%list(:) = [ (i+0.5d0, i = 10+1, 10+n) ]
n = 9
allocate( real_list_vec(5)%list(n) )
real_list_vec(5)%list(:) = [ (i+0.5d0, i = 100+1, 100+n) ]
call msDisplay( real_list_vec, "int_list_vec")
! call msRelease( real_list_vec )
```

output:

```
 int_list_vec =

  1.500  2.500  3.500  4.500  5.500  6.500

  <EMPTY>

  11.500  12.500  13.500  14.500  15.500  16.500  17.500  18.500  19.500  20.500  21.500

  <EMPTY>

  101.500  102.500  103.500  104.500  105.500  106.500  107.500  108.500  109.500
```

`mfIsEmpty` **mfArray empty test**

Interface:

```
function mfIsEmpty( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Tests if the `mfArray A` is empty, *i. e.* has a null size (number of rows multiplied by nupmber of columns).

Note that an sparse matrix (non empty) may have a number of nonzero elements (see `mfNnz`) that is zero.

See also: `Shape`, `Size`

`mfIsEqual` — mfArray equality test

Interface:

```
function mfIsEqual( A1, A2 ) result( bool )

   type(mfArray), intent(in) :: A1, A2
   logical :: bool
```

Description:

Tests if two `mfArray`s are numerically equal, whatever the structure is (*i. e.* sparse or dense, real or complex). The shapes can even mismatch, and can take the zero value.

Comparison between boolean mfArray are excluded: `All` combined with '`.eqv.`' should be used instead.

Remarks: This function returns a scalar logical, whereas '`==`' returns a boolean `mfArray`.

For example:

```
mfIsEqual( A1, A2 )
```

is equivalent to:

```
All( A1 == A2 )
```

but the former statement is much more efficient for sparse `mfArray`s.

See also: `==`, `All`, `mfAll`, `.eqv.`

`mfIsNotEqual` — mfArray equality test

Interface:

```
function mfIsNotEqual( A1, A2 ) result( bool )

   type(mfArray), intent(in) :: A1, A2
   logical :: bool
```

Description:

Returns the negation of `mfIsEqual(A1,A2)`.

See also: `mfIsEqual`

`mfIsLogical` — mfArray boolean test

Interface:

```
function mfIsLogical( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if the `mfArray A` is boolean, *i. e.* if `A` results from any boolean operation ('`==`', '`>`', *etc.*) on `mfArrays`.

See also: `==`

`mfIsReal` — mfArray real test

Interface:

```
function mfIsReal( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if the `mfArray A` is numeric, but not complex. The internal structure of `A` may be dense or sparse.

See also: `mfIsNumeric`, `mfIsComplex`

`mfIsComplex` — mfArray complex test

Interface:

```
function mfIsComplex( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if the `mfArray A` is numeric and complex. The internal structure of `A` may be dense or sparse.

See also: `mfIsNumeric`, `mfIsReal`

`mfIsNumeric` — mfArray numeric test

Interface:

```
function mfIsNumeric( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if the `mfArray A` is numeric (real or complex). The internal structure of `A` may be dense or sparse.

See also: `mfIsReal`, `mfIsComplex`

`mfIsDense` — mfArray dense storage test

Interface:

```
function mfIsDense( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if internal structure of `A` is dense. The data may be boolean, real or complex.

See also: `mfIsSparse`

`mfIsSparse` — mfArray sparsity test

Interface:

```
function mfIsSparse( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if internal structure of `A` is sparse. The data may be real or complex.

See also: `mfIsDense`

`mfIsScalar` — mfArray scalar test

Interface:

```
function mfIsScalar( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if `A` is a scalar.

See also: `mfIsVector`, `mfIsMatrix`

`mfIsVector` — mfArray vector test

Interface:

```
function mfIsVector( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if `A` is a vector.

Remark: A scalar is not a vector, neither a matrix.

See also: `mfIsScalar`, `mfIsMatrix`, `mfIsRow`, `mfIsColumn`

`mfIsMatrix` — mfArray vector test

Interface:

```
function mfIsMatrix( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if `A` is a matrix.

Remark: A vector is not a matrix.

See also: `mfIsScalar`, `mfIsVector`

`mfIsRow` — mfArray kind of vector test

Interface:

```
function mfIsRow( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if `A` is a row vector.

See also: `mfIsVector`, `mfIsColumn`

`mfIsColumn` — mfArray kind of vector test

Interface:

```
function mfIsColumn( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if `A` is a column vector.

See also: `mfIsVector`, `mfIsRow`

`mfIsPerm` — mfArray permutation vector test

Interface:

```
function mfIsPerm( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if `A` has the type "permutation vector".

To make a real checking for a valid permutation, use `mfCheckPerm`.

See also: `mfIsLogical`, `mfIsNumeric`

`All` — test on boolean mfArray

Interface:

```
function All( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if all elements of the `mfArray A` are *TRUE* (therefore, `A` must be a boolean `mfArray`).

See also: `mfAll`, `Any`

`Any` — test on boolean mfArray

Interface:

```
function Any( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if at least one element of the `mfArray A` is *TRUE* (therefore, `A` must be a boolean `mfArray`).

See also: `mfAny`, `All`

`Shape, mfShape` — shape of an mfArray

First interface:

```
function Shape( A ) result( out )

   type(mfArray), intent(in) :: A
   integer :: out(2)
```

Description:

Returns the shape of the `mfArray A` (dense or sparse); the returned integer array is always of rank 2, because `A` is a matrix.

Other interface:

```
function Shape( Qhouse ) result( out )

   type(mfMatFactor), intent(in) :: Qhouse
   integer :: out(2)
```

Description:

Returns the shape of the `mfMatFactor Qhouse`, which comes from the QR decomposition of a sparse matrix; the returned integer array is always of rank 2, because an `A` is a matrix.

Remark: the `mfShape` routine is similar, but returns an `mfArray` instead.

See also: `Size`, `mfMatFactor`

`Size, mfSize` — size of an mfArray

First interface:

```
function Size( A, idim ) result( out )

   type(mfArray), intent(in) :: A
   integer, intent(in), optional :: idim
   integer :: out
```

Description:

If `idim` is not present, returns the total number of elements of the `mfArray A` (dense or sparse).

If `idim` is present (1 or 2), returns the specified dimension.

Other interface:

```
function Size( A, idim ) result( out )

   type(mfMatFactor), intent(in) :: A
   integer, intent(in), optional :: idim
   integer :: out
```

Description:

If `idim` is not present, returns the total number of elements of the `mfMatFactor A` (which comes from some decomposition of a sparse matrix).

If `idim` is present (1 or 2), returns the specified dimension.

Remarks:

- the `mfSize` routine is similar, but returns an `mfArray` instead;
- for a sparse matrix, the size returned includes the zero elements: it is the logical size, not the physical size; so, for any storage kind, the size is always the product of the two dimensions returned by `shape`. Use the `mfNnz` routine to get the number of non zeros.

See also: `Shape`

`mfInt` — scalar integer conversion

Interface:

```
function mfInt( A ) result( out )

   type(mfArray), intent(in) :: A
   integer :: out
```

Description:

This routine is a facility to convert a scalar `mfArray A` in an ordinary f90 integer.

A warning is emitted during conversion, in case of loss of precision.

See also: `mfDble`, `mfCmplx`

`mfDble` — scalar real conversion

Interface:

```
function mfDble( A ) result( out )

   type(mfArray), intent(in) :: A
   real(kind=MF_DOUBLE) :: out
```

Description:

This routine is a facility to convert a scalar `mfArray A` in an ordinary f90 real.

If `A` is complex, returns only the real part.

See also: `mfInt`, `mfCmplx`

`mfCmplx` scalar complex conversion

Interface:

```
function mfCmplx( A ) result( out )

   type(mfArray), intent(in) :: A
   complex(kind=MF_DOUBLE) :: out
```

Description:

This routine is a facility to convert a scalar `mfArray A` in a complex.

See also: `mfInt`, `mfDble`

`mfGetMsgLevel` — message level tuning

Calling syntax:

```
level = mfGetMsgLevel( )
```

returns an integer, ranged from 0 to 3, which is the current message level. See the `msSetMsgLevel` routine for a description of the message level.

See also: `mfGetTrbLevel`, `msSetTrbLevel`, `msMuesliTrace`

`msSetMsgLevel` — message level tuning

Calling syntax:

```
call msSetMsgLevel( level )
```

where `level` is an integer, ranged from 0 to 3, which specifies the message level, described as follows:

- message level = 3: all messages are printed (verbose mode)
- message level = 2: messages of kind 'ERROR' and 'Warning' are printed [default]
- message level = 1: only messages of kind 'ERROR' are printed
- message level = 0: nothing is printed (quiet mode)

There exists three kinds of messages emitted by the MUESLI library; they are tagged as "info", "Warning" and "ERROR". Only the last kind (the most important) stops temporarily the run-time execution via a "pause" statement, which requires the action of the user to resume the execution. The cause of these errors are generally due to a programming fault; this is the reason why the program stops.

As usual for *IEEE-754* arithmetic, floating-point exceptions give *Inf* and *NaN* values which quietly propagate. See `msEnableFPE` to change this default behavior.

N.B.: when message level is equal to 0, programming errors (of kind "ERROR") kill the program. This may be useful when the executable must be launched via a batch system.

See also: `mfGetMsgLevel`, `mfGetTrbLevel`, `msSetTrbLevel`, `msMuesliTrace`

`mfGetTrbLevel` — error traceback tuning

Calling syntax:

```
when = mfGetTrbLevel( )
```

returns a `character(len=4)`, which is the current error traceback policy. See the `msSetTrbLevel` routine for a description of the error traceback policy.

See also: `mfGetMsgLevel`, `msSetMsgLevel`, `msMuesliTrace`

`msSetTrbLevel` — error traceback tuning

Calling syntax:

```
call msSetTrbLevel( when | level )
```

where `when` is a `character(len=4)` which specifies the error traceback policy, described as follows:

- `when` = `"all"`: error traceback is produced for all kind of messages (*i. e.* ERRORS, Warnings and infos)
- `when` = `"auto"`: error traceback is produced only for ERRORS [default]
- `when` = `"none"`: no error traceback

An alternative is to use the integer `level` argument, which can be equal to 0, 1, 2 or 3. An message level which is greater than or equal to `level` triggers the traceback. For example, when `level` is used with the value 2, only messages of kind 'ERROR' and 'Warning' produce a traceback.

N.B.: not all compilers are able to produce an error traceback on demand. See the *MUESLI Inside* document for more information.

See also: `mfGetTrbLevel`, `mfGetMsgLevel`, `msSetMsgLevel`, `msMuesliTrace`

`msSetColoredMsg` — colorize messages on terminal

Calling syntax:

```
call msSetColoredMsg( "on" | "off" )
```

Description:

Colorizes all the Muesli messages written on the terminal. Four colours are used:

- red for the ERRORS;
- orange for the Warnings;
- yellow for the infos;
- grey-italic for the traceback.

Disabling the colors are useful before redirecting the program output to a file. Default is `"on"`.

See also: `mfGetMsgLevel`, `msSetMsgLevel`, `msMuesliTrace`

`msPrintColoredMsg` print a colored user message on terminal

Interface:

```
subroutine msPrintColoredMsg( fmt, msg, color )

   character(len=*), intent(in) :: fmt, msg, color
```

Description:

Prints the string `msg` in this `color` and using format `fmt`.

Colors are limited to the following ones: `"green"`, `"red"`, `"orange"`, `"yellow"` and `"grey"`.

See also: `msSetTermColor`

`msSetTermColor` — set color for printing on terminal

Interface:

```
subroutine msSetTermColor( color )

   character(len=*), intent(in) :: color
```

Description:

Sets the color for printing user message with `msPrintColoredMsg`.

Colors are limited to the following ones: `"green"`, `"red"`, `"orange"`, `"yellow"` and `"grey"`.

To revert to the default color, set color to `"normal"`.

`msGetStdIO` — usual logical unit inquiry

Interface:

```
subroutine msGetStdIO( stdin, stdout, stderr )

   integer, intent(out), optional :: stdin, stdout, stderr
```

Description:

Gets the Fortran unit for `stdin` (usually 5), `stdout` (usually 6) and `stderr` (usually 0).

See also: `msSetStdIO`

`msSetStdIO` — usual logical unit modification

Interface:

```
subroutine msSetStdIO( stdin, stdout, stderr )

   integer, intent(in), optional :: stdin, stdout, stderr
```

Description:

Used to change the Fortran units.

See also: `msGetStdIO`

`msFlush` — I/O flush

Interface:

```
subroutine msFlush( unit )

   integer, intent(in) :: unit
```

Description:

Flushes the I/O unit `unit`.

`msPause` — user pause or timing pause

Calling syntax:

```
call msPause( [message] [, indent] [, duration] )
```

If the optional argument `message` is present (`character(len=*)`), this routine prints the message and waits for a [RETURN] from the user.

The optional argument `indent` (a positive `integer`) allows the user to indent the prompt string

```
[Return] to continue . . .
```

in order to align the beginning of this prompt with other strings printed elsewhere.

If the optional argument `duration` is present (`real`), this routine sleeps for the specified number of seconds.

The two arguments `indent` and `duration` cannot be used together.

`msSetTermWidth` — set terminal character width

Syntax:

```
call msSetTermWidth( width | "auto" )
```

Description:

Set manually the character width of the terminal, for pretty print with the `msDisplay` routine. The argument `width` must be an integer greater than 40.

If the character string `"auto"` is used as argument, then the character width of the terminal will be determined automatically, either from the MF_COLUMNS environment variable, or from the actual value (using the `'resize'` shell command).

See also: `mfGetTermWidth`, `msDisplay`, `mfDisplayColumns`

`mfGetTermWidth` get terminal character width

Interface:

```
function mfGetTermWidth() result ( out )

   integer :: out
```

Description:

Get the character width of the terminal. The returned value is actually an internal value used for pretty print in the `msDisplay` routine. This last value is obtained from:

- first, the value set manually via the `msSetTermWidth` routine;
- then, from the MF_COLUMNS environment variable;
- last, via the 'resize' Unix command; if this latter command is not available on your system, an information message is displayed and the default value 80 is used.

See also: `msSetTermWidth`, `msDisplay`, `mfDisplayColumns`

`msSetAutoFilling` — set out-of-range filling by msSet

Syntax:

```
call msSetAutoFilling( r )
```

Description:

Set manually the value of the real used to fill the elements of an `mfArray` by `msSet` when the indices are out-of-range.

Default value is the special *IEEE* number *NaN*, which means *Not-a-Number*.

See also: `mfGetAutoFilling`

`mfGetAutoFilling` get out-of-range filling by msSet

Interface:

```
function mfGetAutoFilling() result ( r )

   real(kind=MF_DOUBLE) :: r
```

Description:

Get the value of the real used to fill the elements of an `mfArray` by `msSet` when the indices are out-of-range.

Default value is the special *IEEE* number *NaN*, which means *Not-a-Number*, but this number may be changed by use of `msSetAutoFilling`.

`msInitArgs` — mfArray arguments' protection

Interface:

```
subroutine msInitArgs( x1,                                          &
                       x2, x3, x4, x5, x6, x7 )

   type(mfArray) :: x1
   type(mfArray), optional :: x2, x3, x4, x5, x6, x7
```

Description:

Inside a user-defined routine, this routine protects some `mfArray` arguments against a (library) deallocation, in case where they are temporary.

This routine should always be paired with the other routine `msFreeArgs`.

`msFreeArgs` — mfArray arguments' release

Interface:

```
subroutine msFreeArgs( x1,                                          &
                       x2, x3, x4, x5, x6, x7 )

   type(mfArray) :: x1
   type(mfArray), optional :: x2, x3, x4, x5, x6, x7
```

Description:

Inside a user-defined routine, this routine ends the argument's protection set by `msInitArgs`.

This routine should always be paired with the other routine `msInitArgs`.

`msSetAsParameter` **data protection**

Interface:

```
subroutine msSetAsParameter( x1, x2, x3, x4, x5, x6, x7, param )

   type(mfArray)           :: x1
   type(mfArray), optional :: x2, x3, x4, x5, x6, x7
   logical                 :: param
```

Description:

When used with "`param=.true.`", protects the data (and type) inside each `mfArrays`. Lets the user creating pseudo parameters `mfArrays`, because the Fortran attribute *parameter* cannot be used in any circumstances.

Up to seven `mfArrays` can be simultaneously set.

Before freeing an `mfArray` via the `msRelease` routine, the user must apply `msSetAsParameter` with "`param=.false.`".

Of course, by default, an `mfArray` is not data-protected.

`msReturnArray` — temporary mfArray mark

Interface:

```
subroutine msReturnArray( A )

   type(mfArray) :: A
```

Description:

At the end of a user-defined function returning an `mfArray`, marks this object as temporary.

Avoids memory leaks during the execution of your program.

See also: `msInitArgs`, `msFreeArgs`, `mfIsTempoArray`

`mfIsTempoArray` — temporary mfArray check

Interface:

```
function mfIsTempoArray( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Check if the `mfArray A` is marked as temporary.

See also: `msInitArgs`, `msFreeArgs`, `msReturnArray`

`msFlops` — nb of floating-point operations

Interface:

```
subroutine msFlops( count )

   integer*8, intent(out) :: count
```

Description:

Returns the number of floating-point operations made by the processor, since the last initialization.

Initialization must be made by:

```
call msFlops( init=0 )
```

The other call:

```
call msFlops( count )
```

returns the flops count since the initialization.

`count` ≥ 0, except if the program has been linked with the dummy version of PAPI (dummy_papi.o), or if the OS doesn't support hardware counting (*e. g.* the linux kernel has not been patched for use of the PERFCTR library); in the latter case, `msFlops` always returns the value -1.

Remarks: requires a specific version of the PAPI library (cf. in papi/version)

See also: `mfFlops`, `mfIsFlopsOk`

`mfFlops` — nb of floating-point operations

Calling syntax:

```
count = mfFlops( )
```

Description:

Returns the number of floating-point operations made by the processor, since the last initialization, in the `mfArray count`.

Initialization must be made by:

```
call msFlops( init=0 )
```

`count` ≥ 0, except if the program has been linked with the dummy version of PAPI (dummy_papi.o), or if the OS doesn't support hardware counting (*e. g.* the linux kernel has not been patched for use of the PERFCTR library); in the latter case, `mfFlops` always returns the value -1.

Remarks: requires a specific version of the PAPI library (cf. in papi/version)

See also: `msFlops`, `mfIsFlopsOk`

`mfIsFlopsOk` — inquire if flops is available

Calling syntax:

```
bool = mfIsFlopsOk( )
```

Description:

Returns a `logical` revealing the availability of the `mfFlops` routine.

See also: `mfFlops`, `msFlops`

`msSetAutoComplex` set auto conversion to complex

Syntax:

```
call msSetAutoComplex( .true. | .false. )
```

Description:

By default, real `mfArrays` are automatically converted in complex when operation like $\sqrt{-1}$ are made.

When calling this routine with argument equal to `.false.`, conversion to complex is not done, then resulting to a special *IEEE* value (*Inf* or *NaN*, according to the operation used).

See also: `mfGetAutoComplex`

`mfGetAutoComplex` get auto conversion to complex

Interface:

```
function mfGetAutoComplex() result( bool )

logical, intent(in) :: bool
```

Description:

Returns a boolean specifying whether MUESLI convert real to complex for some operation (like $\sqrt{-1}$).

See also: `msSetAutoComplex`

`msEnableFPE` run-time floating-point exceptions trapping

Interface:

```
subroutine msEnableFPE( exception, full_trapping )

character(len=*), intent(in)           :: exception
logical,          intent(in), optional :: full_trapping
```

Description:

Enables, at run-time, the trapping for a given floating-point exception, *i. e.* the program stops when the specified exception(s) is (are) encountered.

`exception` may be one of: `"overflow"`, `"zero_divide"`, `"invalid"`, `"underflow"` or `"usual_exceptions"`.

`"usual_exceptions"` is a shortcut for the main three traditional exceptions: `"overflow"`, `"zero_divide"`, `"invalid"`.

`full_trapping` is an optional argument which allows the trapping inside the MUESLI library sources (default is `.false.`). Note that this features is usually reserved to the MUESLI developpers.

Remark: The call may be located anywhere is the program. Usually, it is added at the beginning of the program. However, exceptions' trapping may be added or removed later on.

See also: `msDisableFPE`

`msDisableFPE` run-time floating-point exceptions trapping

Interface:

```
subroutine msDisableFPE( exception )

character(len=*), intent(in) :: exception
```

Description:

Disables, at run-time, the trapping for a given floating-point exception, *i. e.* the program doesn't stop when the specified exception(s) is (are) encountered, but produces accordingly *Inf*s and *NaN*s.

`exception` may be one of: `"overflow"`, `"zero_divide"`, `"invalid"`, `"underflow"` or `"usual_exceptions"`.

`"usual_exceptions"` is a shortcut for the main three traditional exceptions: `"overflow"`, `"zero_divide"`, `"invalid"`.

See also: `msEnableFPE`

`mfGetRoundingMode` — floating-point rounding mode

Calling syntax:

```
rounding_mode = mfGetRoundingMode( )
```

Description:

Returns the floating-point rounding mode in a character string. The *IEEE-754* rounding mode may be equal to:

- `"to_zero"`: *IEEE-754* rounding to zero
- `"nearest"`: *IEEE-754* rounding to nearest
- `"up"`: *IEEE-754* rounding to pos. infinity
- `"down"`: *IEEE-754* rounding to neg. infinity

See also: `msSetRoundingMode`

`msSetRoundingMode` — floating-point rounding mode

Interface:

```
subroutine msSetRoundingMode( rounding_mode )

character(len=*) :: rounding_mode
```

Description:

Set the floating-point rounding mode to the specified mode. The *IEEE-754* rounding mode may be equal to:

- `"to_zero"`: *IEEE-754* rounding to zero
- `"nearest"`: *IEEE-754* rounding to nearest
- `"up"`: *IEEE-754* rounding to pos. infinity
- `"down"`: *IEEE-754* rounding to neg. infinity

Warning: The ISO/IEC TR 15580:1998(E) Technical Report, concerning the behavior of Fortran programs, specifies (Section 2.4): "In a procedure, the processor ensures that the flags for rounding have the same values on return as on entry." In the current MUESLI implementation, the rounding mode is set via a C routine, and can violate the preceding requirement; then the use of the `msSetRoundingMode` routine is left to the programmer responsability.

See also: `mfGetRoundingMode`

MF_NUMERICAL_CHECK — debugging additional checks

Interface:

```
logical :: MF_NUMERICAL_CHECK
```

Description:

This logical global variable is used by the library to do additional checks (validity of certain results, presence of *NaN* and, if required, precision of the jacobian matrix provided by the user, ...).

When Muesli is compiled in DEBUG mode (usually by the developpers of Muesli), this variable is initially set to *TRUE*. Otherwise, for ordinary users of the library (which certainly compile it in OPTIM mode), it is initially set to *FALSE*.

For debugging purpose, you can set its value temporarily to *TRUE*, especially when using the ODE/DAE solvers (msOdesolve, msDaesolve), or the minimization routines (mf/msLsqNonLin).

See also: mf_DE_Options

`mfUnit` — physical unit *(derived type)*

Description:

The Fortran derived type `mfUnit` (hidden) contains some usual physical units and constants. It can be used to initialize the physical unit of an `mfArray`, via the operators '`*`' and '`=`'.

No declarations has to be done. Instead, the program may refer to some predefined `mfUnit`s, as *e. g.*:

```
y = 0.25d0*u_kg

v = 0.10d0*c_speed_of_light
```

Remarks:

- see the *MUESLI User Guide* document for a complete list of the available physical units, physical constants and multipliers.
- physical units begin with '`u_`', multipliers begin with '`m_`', whereas physical constants begin with '`c_`'.

See also: `*`, `=`, `msUsePhysUnits`, `msSetPhysDim`, `mfHasNoPhysDim`, `mfHaveSamePhysDim`

`msUsePhysUnits` Physical units activation

Interface:

```
subroutine msUsePhysUnits( mode )

   character(len=*), intent(in) :: mode
```

Description:

Activates or de-activates the computation of physical units; `mode` must be `"on"` or `"off"`.

Default is `"off"`.

See also: `mfUnit`, `msSetPhysDim`

`msSetPhysDim` set physical unit

The first interface:

```
subroutine msSetPhysDim( A,                                          &
                         Mass, Length, Time,                         &
                         Temp, Electr_Intens, Mole, Lumin_Intens,    &
                         no_dim )

   type(mfArray), intent(in out) :: A
   real(kind=MF_DOUBLE), intent(in), optional :: Mass,               &
                                                 Length,             &
                                                 Time,               &
                                                 Temp,               &
                                                 Electr_Intens,      &
                                                 Mole,               &
                                                 Lumin_Intens
   logical, intent(in), optional :: no_dim
```

is used for setting the physical unit of the `mfArray A`, from some basic physical dimensions. `no_dim=.true.` is used to set `A` as a non-dimensional quantity.

The second interface:

```
subroutine msSetPhysDim( A, B )

   type(mfArray), intent(in out) :: A
   type(mfArray), intent(in) :: B
```

allows the user to set the physical unit of the `mfArray A`, by copying that of the `mfArray B`.

See the *MUESLI User Guide* document for some examples.

See also: `mfUnit`, `msUsePhysUnits`, `mfHasNoPhysDim`, `mfHaveSamePhysDim`

`msSetPhysUnitAbbrev` set user physical unit abbrevation

Interface:

```
subroutine msSetPhysUnitAbbrev( user_unit, abbrev )

   type(mfUnit) :: user_unit
   character(len=*) :: abbrev
```

is used for setting the abbrevation of a user physical unit.

`abbrev` is an input string limited to 12 characters.

See also: `mfUnit`, `msUsePhysUnits`

`mfHasNoPhysDim` — dimensionless test

Interface:

```
function mfHasNoPhysDim( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Tests if the `mfArray A` is dimensionless.

See also: `mfUnit`, `msUsePhysUnits`, `msSetPhysDim`, `mfHaveSamePhysDim`

`mfHaveSamePhysDim` — dimension equality test

Interface:

```
function mfHaveSamePhysDim( A, B ) result( bool )

   type(mfArray), intent(in) :: A, B
   logical :: bool
```

Description:

Tests if the `mfArrays A` and `B` have the same physical dimensions.

See also: `mfUnit`, `msUsePhysUnits`, `msSetPhysDim`, `mfHasNoPhysDim`

`msPrepHashes` — hashes print preparation

Calling syntax:

```
call msPrepHashes( start, end )
```

Description:

Prepares data before calls to `msPrintHashes`.

`start` and `end` are numerical values used to monitor the progress bar. They must be both of integer or real (double) type.

See also: `msPrintHashes`, `msPostHashes`

`msPrintHashes` **hashes print**

Calling syntax:

```
call msPrintHashes( val )
```

Description:

Makes on the terminal a progress bar by printing hashes (`#`).

Usually, the call is located inside a loop; `val` is the numerical value (either integer or real double) monitoring the progression. It's numerical extremum values have to be passed as arguments of `msPrepHashes`.

See also: `msPostHashes`, `msPrintProgress`

Example(s):

```
integer :: i_start, i_end, i
...
call msPrepHashes( i_start, i_end )
do i = i_start, i_end
   ...
   <some computation>
   ...
   call msPrintHashes( i )
end do
call msPostHashes( )
```

or:

```
double precision :: time_start, time_end, time
...
call msPrepHashes( time_start, time_end )
time = time_start
do while( time <= time_end)
   ...
   <some computation>
   ...
   call msPrintHashes( time )
   time = <new value of time, increasing but may be not linear with iterations>
end do
call msPostHashes( )
```

`msPostHashes` — hashes print conclusion

Calling syntax:

```
subroutine msPostHashes()
```

Description:

Should be added after the loop containing the call to `msPrintHashes`.

See also: `msPrepHashes`, `msPrintHashes`

`msPrepProgress` — percent progress preparation

Calling syntax:

```
call msPrepProgress( start, end [, disp_times, estimator] )
```

Description:

Prepares data before calls to `msPrintProgress`.

`start` and `end` are numerical values used to monitor the progress bar. They must be both of integer or real (double) type.

If the logical optional argument `disp_times` is set to `.false.`, time values (left time and estimated remaining time – wall clock time is used, not CPU time) are not displayed (default is *TRUE*).

The string optional argument `estimator` prescribes the method to be used to estimate the remaining time. Default is `"GLE"` (Global Linear Estimator), which corresponds to a fast method but with the assumption of a linear behavior of the cost along all the iterations. The other available method is `"LPE"` (Local Power Estimator), which involves a Least Square problem, and assumes that the behavior of the cost is under a power form; this latter method is more expensive but should work well with a larger class of problems than `"GLE"`.

See also: `msPrintProgress`, `msPostProgress`

`msPrintProgress` **percent progress**

Calling syntax:

```
call msPrintProgress( val )
```

Description:

Print on STDOUT the percentage of the work already done. According to the setting used during initialization via `msPrepProgress`, the estimated remaining time may also be displayed.

Usually, the call is located inside a loop (at the end); `val` is the numerical value (either integer or real double) monitoring the progression. It's numerical extremum values have to be passed as arguments of `msPrepProgress`.

See also: `msPostProgress`, `msPrintHashes`

Example(s):

```
integer :: i_start, i_end, i
...
call msPrepProgress( i_start, i_end )
do i = i_start, i_end
   ...
   <some computation>
   ...
   call msPrintProgress( i )
end do
call msPostProgress( )
```

or:

```
double precision :: time_start, time_end, time
...
call msPrepProgress( time_start, time_end )
time = time_start
do while( time <= time_end)
   ...
   <some computation>
   ...
   call msPrintProgress( time )
   time = <new value of time, increasing but may be not linear with iterations>
end do
call msPostProgress( )
```

`msPostProgress` — percent progress conclusion

Interface:

```
subroutine msPostProgress()
```

Description:

Should be added after the loop containing the call to `msPrintProgress`.

See also: `msPrepProgress`, `msPrintProgress`

`mfToLower` — lowering string's case

Calling syntax:

```
s2 = mfToLower( s1 )
```

Description:

Returns the input strings `s1` with each letter converted to lower case.

See also: `mfToUpper`

`mfToUpper` uppering string's case

Calling syntax:

```
s2 = mfToUpper( s1 )
```

Description:

Returns the input strings `s1` with each letter converted to upper case.

See also: `mfToLower`

`msFindIOUnit` automatically find a free IO unit number

Interface:

```
subroutine msFindIOUnit( unit )

   integer, intent(out) :: unit
```

Description:

Returns an integer IO unit which can be used for opening a file.

`mfIsVersion` — test on version strings

Calling syntax:

```
bool = mfIsVersion( v_1, op, v_2 )
```

Description:

Compares the two version strings `v_1` and `v_2` with the operator `op` (`"=="`, `">"`, `">="`, `"<"`, `"<="`). The version string is a character string, of size at most 8, matching the rule `a.b.c`, where `a`, `b` and `c` are integers ranged from 0 to 99.

This function returns a logical and is usually applied to strings as `MF_MUESLI_VERSION` or `MF_LAPACK_VERSION`.

See also: `msRequMuesliVer`

`msRequMuesliVer` check for a minimum Muesli version

Interface:

```
subroutine msRequMuesliVer( version )

   character(len=*) :: version
```

Description:

Checks that the current Muesli version used at run-time is at least equal to that provided in the `"version"` argument. This argument must be a string of the form `a.b.c`, where `a`, `b` and `c` are integers ranged from 0 to 99.

See also: `MF_MUESLI_VERSION`, `mfIsVersion`

MF_COMPILER_VERSION — compiler vendor and version

Calling syntax:

```
string = MF_COMPILER_VERSION
```

returns the compiler vendor and version used during the compilation of the MUESLI library.

See also: MF_MUESLI_VERSION, MF_COMPILATION_CONFIG, MF_LAPACK_VERSION

MF_COMPILATION_CONFIG — Configuration used to compile MUESLI

Calling syntax:

```
string = MF_COMPILATION_CONFIG()
```

returns either "Debug" or "Optim".

Note the parenthesis used after the name of the variable, because it is implemented as a function.

See also: MF_MUESLI_VERSION, MF_COMPILER_VERSION, MF_LAPACK_VERSION

MF_MUESLI_VERSION — MUESLI version

Calling syntax:

```
string = MF_MUESLI_VERSION
```

returns the MUESLI version used during the link of the executable.

The returned `string` may be used by the `mfIsVersion` boolean function.

See also: MF_COMPILER_VERSION, MF_COMPILATION_CONFIG, MF_LAPACK_VERSION

`msMuesliTrace` — helper for debugging purpose

Interface:

```
subroutine msMuesliTrace( pause )

   character(len=*), intent(in) :: pause
```

Description: Basically, this subroutine helps the user by printing, if available, a traceback of the program; optionally, it can make a pause (only if the `pause` argument is equal to `"yes"`).

Contrary to the internal traceback routine, the behavior of this routine depends only on the value of the *message level* global MUESLI property.

Remark: In order to be useful, the use of this routine requires that the Muesli library has been compiled with debugging turned *On*. This is the case if the library is used in the *DEBUG* mode. On the contrary, it suffices that the user adds the appropriate option (*e. g.* `-g` for most of compilers) for its own source files.

See also: `mfGetMsgLevel`, `msSetMsgLevel`

`mfReadLine` reads a line from terminal with editing and history facilities

Interface:

```
function mfReadLine( prompt ) result( string )

   character(len=*), intent(in), optional :: prompt
   character(len=1024) :: res
```

Description: This function prompts the string `prompt` and waits the user to input a character string, using the readline library. As a result, the user can navigate in an history, recall any previous input and modify it.

It returns the string `string`, which is automatically added in the history only if it differs from the last entry.

The history can be saved with the use of the `msWriteHistoryFile` routine.

A small example of call can be found in the *Muesli User's Guide*.

See also: `msReadHistoryFile`, `msClearHistory`, `msAddEntryInHistory`, `msRemoveLastEntryInHistory`

`msReadHistoryFile` — read a 'readline' history file

Interface:

```
subroutine msReadHistoryFile( filename )

   character(len=*), intent(in), optional :: filename
```

Description: Reads an 'history' file previously stored by the `msWriteHistoryFile` routine.

If the optional `filename` is not present, 'history' is read from the file: `/.mfreadline.history`

See also: `mfReadLine`, `msClearHistory`, `msAddEntryInHistory`, `msRemoveLastEntryInHistory`

`msWriteHistoryFile` — write a 'readline' history file

Interface:

```
subroutine msWriteHistoryFile( filename )

   character(len=*), intent(in), optional :: filename
```

Description: Writes an 'history' file in `filename`. This 'history' contains all entries read by the `mfReadLine` routine.

See also: `msReadHistoryFile`, `msClearHistory`, `msAddEntryInHistory`, `msRemoveLastEntryInHistory`

`msClearHistory` — clear the 'readline' history

Interface:

```
subroutine msClearHistory( )
```

Description: Clears all entries read by the `mfReadLine` routine.

See also: `msReadHistoryFile`, `msWriteHistoryFile`, `msAddEntryInHistory`, `msRemoveLastEntryInHistory`

`msAddEntryInHistory` add an entry in the 'readline' history

Interface:

```
subroutine msAddEntryInHistory( string )

   character(len=*) :: string
```

Description: Adds the entry specified by the `string` argument in the 'readline' history.

See also: `mfReadLine`, `msReadHistoryFile`, `msWriteHistoryFile`, `msClearHistory`, `msRemoveLastEntryInHistory`

`msRemoveLastEntryInHistory` remove last entry in the 'readline' history

Interface:

```
subroutine msRemoveLastEntryInHistory( )
```

Description: Remove last entry read by the `mfReadLine` routine.

See also: `msReadHistoryFile`, `msWriteHistoryFile`, `msClearHistory`, `msAddEntryInHistory`

1.2 File Input/Output

`msSaveAscii`	ASCII file saving
`mfLoadAscii`	ASCII file loading
`msSaveSparse`	Sparse ASCII file saving
`mfLoadSparse`	Sparse ASCII file loading
`msLoadSparse`	Sparse ASCII file (with Right-Hand-Side) loading
`msSave`	Binary file saving
`mfLoad`	Binary file loading
`mfLoadTriConnect`	Connectivity Binary file loading
`msSaveHDF5`	HDF5 file saving
`mfLoadHDF5`	HDF5 file loading
`msMedit`	graphic mfArray editor

See also:

Core Routines

Data Analysis Functions

Operators

Elementary Math Functions

Specialized Math Functions

Elementary Matrix Manipulation Functions

Matrix Functions

Polynomial Functions

Optimization and Function Functions

Sparse Matrices

`msSaveAscii` — ASCII file saving

Interface:

```
subroutine msSaveAscii( filename, A, append, format )

   character(len=*), intent(in)           :: filename
   type(mfArray),    intent(in)           :: A
   logical,          intent(in), optional :: append
   character(len=*), intent(in), optional :: format
```

Description:

Saves under ASCII form the dense `mfArray A` in the file named `filename`.

`A` is saved via a full 2D format, *i. e.* the number of lines of `filename` is exactly the number of rows of `A`. Special IEEE values (*NaN* and *Inf*s) are written also in ASCII form (using the strings `"NaN"`, `"Infinity"` and `"-Infinity"`).

If the boolean `append` is present and *TRUE* then the data is written at the end of the (previously written) file. The default is to overwrite a pre-existing file.

If the character string `format` is present, it specifies the *Fortran format* used during the write of individual numbers (for example `"ES13.6"`). It must be of course a non-empty valid format for writing numerical real values. As usual, if the format is not sufficient to write the required number of digits, stars will be printed out instead (for example, `"F3.0"` is the shortest format if your are sure that all numbers in the mfArray are integers; and even `"F2.0"` if they are all positive).

`msSaveAscii` doesn't store the internal properties of `A`, so writing a boolean `mfArray` converts it in real. Another limitation is that only the real part of complex numbers is written; to save the imaginary part, use:

```
call msSaveAscii( filename, mfImag(A) )
```

Remarks:

- if `A` contains a permutation, it will be written as a column vector, using integers (the optional argument `format` will be discarded). However, reading again the same file with mfLoadAscii will convert it into real numbers, as ordinary matrices; `msSave` is therefore recommended to save a permutation vector.
- to avoid loss of precision during the read of `filename`, a sufficient number of decimal digits is used to print each real when the optional argument `format` is not present.
- do not use the optional arguments `append` and `format` if you plan to read the file with `mfLoadAscii`.
- to save sparse matrices, use the `msSaveSparse` routine.

See also: `mfLoadAscii`, `msSave`, `msSaveHDF5`

`mfLoadAscii` ASCII file loading

Interface:

```
function mfLoadAscii( filename,                                          &
                      ieee, comment, rect, row_max, col_max ) result( out )

   character(len=*), intent(in)           :: filename
   logical,          intent(in), optional :: ieee, rect
   character(len=1), intent(in), optional :: comment
   integer,          intent(in), optional :: row_max, col_max

   type(mfArray) :: out
```

Description:

Reads data from an ASCII file named `filename` and copy it in an `mfArray`. Inside the file, numerical data are separated by one or more space(s) and tabulation(s). Usually, this file has been created by the `msSaveAscii` routine but actually, nearly all kinds of ASCII file may be read by `mfLoadAscii`: see below.

If `ieee` is present and equal to `.true.`, this routine is able to read special IEEE values, such as *NaN* and *Inf* (but this feature makes the read not very efficient for some compilers). The default behavior is to rise an error if any non numeric value is found.

If `comment` is present, all lines beginning (at any position) by this character are skipped (of course, this character must be different from: a space, a tabulation, a carriage-return, a digit, the signs + and −, the letters `I` and `N` which begin *Inf* and *NaN*); comments after data are also ignored. Empty lines are treated as commented. The default behavior assumes that the file contains only numerical values and no empty lines.

If `rect` is present and equal to `.false.`, the file read may contain data which are not under a rectangular layout. In this latter case, the routine returns a column vector `mfArray`. By default, this routine expects a rectangular layout, *i. e.* the same number of reals stored in each row.

For the standard case only (*i. e.*, `rect` is equal to `.true.`, or not present), the optional arguments `row_max` and `col_max` allow the user to read partially the data file. These two arguments are not dependent (and do not need to be both present). Note also that the whole data file may have a non-rectangular layout: it is sufficient that (`row_max`, `col_max`) covers a rectangular subpart of the data file.

Remark: DOS or UNIX text files are supported.

See also: `msSaveAscii`, `mfLoad`, `mfLoadHDF5`

`msSaveSparse` — Sparse ASCII file saving

Interface:

```
subroutine msSaveSparse( filename, A, format )

   character(len=*), intent(in) :: filename
   type(mfArray), intent(in) :: A
   character(len=3), intent(in), optional :: format
```

Description:

Saves the sparse matrix `mfArray A` in the file `filename`, using the (optionally) specified ASCII `format`:

- `"CSC"`: Compact Sparse Column (default)

  The first line contains exactly five items, which are the numbers of rows and columns (`nrow` and `ncol`), an integer (1 or 0) specifying whether the entries are row-sorted or not in each colums, the tag `"CSC"` and the type of data (`"real"` or `"complex"`);

  the following line(s) contains the integer vector of the pointers to the columns;

  the following line(s) contains the integer vector of the row indices;

  the remaining line(s) contains the floating-point values.

- `"CSR"`: Compact Sparse Row

  Very close to the previous format, except that row and column role are exchanged.

- `"HBO"`: Harwell-Boeing

  A commonly used format for sparse matrices, described at:
  http://math.nist.gov/MatrixMarket/formats.html#hb

- `"MTX"`: Matrix Market (also called "Coordinates format", or `"COO"`)

  A commonly used format for sparse matrices, described at:
  http://math.nist.gov/MatrixMarket/formats.html#MMformat

If `format` is not present, this routine uses the `"CSC"` format.

The routine always overwrites a pre-existing file.

See also: `mfLoadSparse`

`mfLoadSparse` — Sparse ASCII file loading

Interface:

```
function mfLoadSparse( filename, format, duplicated_entries ) result( out )

   character(len=*), intent(in)           :: filename
   character(len=3), intent(in), optional :: format
   character(len=*), intent(in), optional :: duplicated_entries
   type(mfArray)                          :: out
```

Description:

Reads the file `filename` and copy the sparse matrix it contains in an `mfArray`.

When reading the file, it uses the (optionally) specified ASCII `format`:

- `"CSC"`: Compact Sparse Column
- `"CSR"`: Compact Sparse Row
- `"HBO"`: Harwell-Boeing (without Right Hand Side; on the contrary, use `msLoadSparse` instead)
- `"MTX"`: Matrix Market (also called "Coordinates format", or `"COO"`)

If `format` is not present, this routine tries to auto-detect the used format.

The description of these different formats can be found at the `msSaveSparse` entry.

After reading the matrix:

- eventual duplicated entries are treated according to the optional character argument `duplicated_entries`. When this argument is present and equal to `"ignored"`, duplicated entries are ignored; when it is equal to `"added"` (default value), duplicated entries are added; when it is equal to `"replaced"`, last entries found overwrite previous one.
- entries containing a null value are removed.

See also: `msLoadSparse`, `msSaveSparse`, `mfSpImport`

`msLoadSparse` Sparse ASCII file (with Right-Hand-Side) loading

Calling syntax:

```
call msLoadSparse( mfOut(A[,RHS]), filename [, format, duplicated_entries] )
```

Description:

Similar to `mfLoadSparse` but allows the user to retrieve a Right-Hand-Side (`HBO` format only) together with the sparse matrix.

The input arguments are described in the `mfLoadSparse` description.

`A` and `RHS` are both `mfArrays` and, after reading the file, contain respectively the sparse matrix and the Right-Hand-Side (dense or sparse, according to the file content).

See also: `mfLoadSparse`, `msSaveSparse`, `mfSpImport`

msSave — Binary file saving

Interface:

```
subroutine msSave( filename, A, compressed )

   character(len=*), intent(in) :: filename

   type(mfArray), intent(in) :: A
or type(mfTriConnect), intent(in) :: A

   logical, optional :: compressed
```

Description:

Saves under a binary form the `mfArray` (or `mfTriConnect`) `A` in the file named `filename`. The MUESLI Binary Format (MBF) used is specific to the current library. Any filename extension is supported; the `".mbf"` used in the MUESLI examples just helps to remind that it is a MUESLI binary file.

If `compressed` is present and true, or if the specified filename ends with the extension `".gz"`, then `msSave` directly stores the file in the 'gzip' compressed format.

Remarks:

- to read the created file, you must use `mfLoad` (or `mfLoadTriConnect`), because a special format is used to organize data inside it. MUESLI add a special tag to help with compatibility between versions.
- `msSave` takes care of endianness of your architecture, so read/write this kind of file should be possible across big endian and little endian machines.
- `msSave` stores all the internal properties of `A`, including its dynamic type, structure and data, hidden matrix properties and physical units; so `mfLoad` (or `mfLoadTriConnect`) should restore exactly the same `mfArray` (but for sparse structure, internal arrays will have the minimal size).
- `msSave` always overwrites a pre-existing file.

See also: `msSaveAscii`, `msSaveHDF5`

`mfLoad` Binary file loading

Interface:

```
function mfLoad( filename ) result( out )

   character(len=*), intent(in) :: filename
   type(mfArray) :: out
```

Description:

Reads a matrix previously stored in the file named `filename` (MUESLI Binary Format) by the routine `msSave`, and copy it in an `mfArray`.

This routine is able to read directly a compressed gzipped file, as stored by `msSave`.

See also: `mfLoadAscii`

`mfLoadTriConnect` Connectivity Binary file loading

Interface:

```
function mfLoadTriConnect( filename ) result( out )

   character(len=*), intent(in) :: filename
   type(mfTriConnect) :: out
```

Description:

Reads a connectivity structure previously stored in the file named `filename` (MUESLI Binary Format) by the routine `msSave`, and copy it in an `mfArray`.

See also: `mfLoadAscii`

`msSaveHDF5` HDF5 file saving

Interface:

```
subroutine msSaveHDF5( filename, A,                                    &
                       name, file_access, mfarray_overwrite, status )

   character(len=*), intent(in)            :: filename
   type(mfArray),    intent(in)            :: A
   character(len=*), intent(in),  optional :: name
   character(len=*), intent(in),  optional :: file_access
   logical,          intent(in),  optional :: mfarray_overwrite
   integer,          intent(out), optional :: status
```

Description:

By default (or if `file_access` is present and equal to `"trunc"`), this routine creates a new HDF5 file (or overwrites a pre-existing one) and writes the `mfArray A` (dense or sparse) in it.

If `file_access` is present and equal to `"append"`, it tries to add in the existing HDF5 file a new group for `A`. In the case where a group with the same name already exists, the group will be overwritten only if `mfarray_overwrite` is present and equal to true; on the contrary, it will return a non zero value in the `status` flag (if this later optional argument is present).

The `mfArray A` is written as one group (named from the optional argument `name`, or `"untitled mfArray"` by default), with all its MUESLI properties, stored as group attributes.

The created file can be read in MUESLI by the routine `mfLoadHDF5`; moreover, it can be browsed by a specific viewer, as `'hdfview'` for example.

Remark: No specific extension is required for the `"filename"`, nevertheless `".h5"` or `".hdf5"` are often used.

See also: `msSave`, `msSaveAscii`

`mfLoadHDF5` HDF5 file loading

Interface:

```
function mfLoadHDF5( filename, name ) result( out )

   character(len=*), intent(in)           :: filename
   character(len=*), intent(in), optional :: name
   type(mfArray)                          :: out
```

Description:

Reads from an HDF5 file and tries to find an `mfArray` named from the optional argument `name`. If this latter in not present, it tries to get the first group which has an attribute named `MF_MUESLI_VERSION`. It doesn't enter in any group: the searched `mfArray` must be located just under the root structure (`"/"`) of the HDF5 file.

The file `filename` must have been created by `msSaveHDF5`.

See also: `mfLoad`, `mfLoadAscii`

`msMedit` graphic mfArray editor

Interface:

```
subroutine msMedit( A )

   type(mfArray), intent(in out) :: A
```

Description:

This routine allows the user to interactively modify a real `mfArray`, with a small graphical Matrix-editor.

Actually, it launch another X11 (Qt4-based) application, named '`meditor`' (spreadsheet-like), which displays a table containing initially the matrix `A`. The user can modify the values and extend the shape of the matrix. When the window is closed, new data are automatically saved in the `mfArray A`.

Special IEEE values can be entered in cells, by typing their ASCII names, *NaN* and *Inf* ('`meditor`' is not case-sensitive). Bad entries are ignored.

To extend the matrix, click anywhere outside the matrix and enter a value (empty cells of the new matrix will be filled with 0).

Remarks:

- complex numeric matrices, or containing boolean values, are not accepted;
- if `A` is a sparse matrix, it is first converted to a dense one before display, but the new modified matrix will be converted back to sparse storage;
- the run of the '`meditor`' tool blocks the calling program till it's window is closed by the user;
- the size of `A` is limited to 200×200: it can be changed in the file '`meditor.cpp`' if needed (of course, the '`meditor`' tool has to be compiled again).

1.3 Data Analysis Functions

mfMax, msMax	mfArray max
mfMin, msMin	mfArray min
mfSum	columns sum
mfExtrema	data extrema
mfProd	columns product
mfSort, msSort	columns/rows sort
mfIsSorted	vector sort test
mfSortRows	sort whole rows of a matrix
mfDiff	difference and approximate derivative
mfGradient	approximate 1D gradient
msGradient	approximate 2D gradient
mfMean	columns mean
mfMedian	columns median
mfVar	columns variance
mfStd	columns standard deviation
mfRMS	root mean square
msHist	data histogram
mfMoments	few first moments of a distribution
mfSmooth	smoothing of vector values
mfXCorr	autocorrelation of a vector data
mfXCorr2	autocorrelation of a matrix data
mfFFT, mfInvFFT	discrete Fast Fourier transformation
mfFFT2, mfInvFFT2	2D discrete Fast Fourier transformation
mfFourierCos, mfInvFourierCos	discrete Fourier cosine transformation
mfFourierSin, mfInvFourierSin	discrete Fourier sine transformation
mfFourierLeg, mfInvFourierLeg	discrete Fourier-Legendre transformation

See also:

Core Routines

File Input/Output

Operators

Elementary Math Functions

Specialized Math Functions

Elementary Matrix Manipulation Functions

Matrix Functions

Polynomial Functions

Optimization and Function Functions

Sparse Matrices

mfMax — mfArray max

`mfMax` has two forms. The first one is used to compute the maximum value of the columns (default behavior) or of the rows of an `mfArray`. The second one compares two matrices.

First interface:

```
function mfMax( A, dim, nanflag ) result( out )

   type(mfArray),    intent(in)           :: A
   integer,          intent(in), optional :: dim
   character(len=*), intent(in), optional :: nanflag
   type(mfArray)                          :: out
```

If `dim` is present, `A` is always considered as a matrix, and the returned vector contains the maximum of each column (if `dim` = 1) or the maximum of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfMax` returns the largest element; otherwise it returns the max of each columns.

If the string `nanflag` is present, it must be equal to either `"omitNaN"`, or `"includeNaN"` (practically, the argument is not case sensitive). By default, `NaN` values are discarded (see examples below), so `"omitNaN"` is implied.

For a sparse `mfArray A`, the result is exactly the same as for the full corresponding matrix. In all cases (sparse or dense), it must contain real values.

Second interface:

```
function mfMax( A, B, nanflag ) result( out )

   type(mfArray), intent(in) :: A

   type(mfArray), intent(in) :: B
or real(kind=MF_DOUBLE), intent(in) :: B

   character(len=*), intent(in), optional :: nanflag
   type(mfArray) :: out
```

When the two arguments `A` and `B` are `mfArrays` (sparse storage is allowed), they must have the same shape and must contain real values; in this case, the *max* operation is applied element-wise. Otherwise, the second argument `B` may be a scalar `real`.

The optional argument `nanflag` has the same meaning as in the first interface.

See also: mfMin, msMax, mfExtrema

Example(s):

```
x = [ 1.0d0, MF_NAN, 5.0d0, 2.0d0, 4.0d0 ]
call msDisplay( mfMax(x), "mfMax(x)" )
```

output:

```
mfMax(x) =

   5.0000
```

···/···

```
x = [ 1.0d0, MF_NAN, 5.0d0, 2.0d0, 4.0d0 ]
call msDisplay( mfMax(x,nanflag='includeNaN'), "mfMax(x,nanflag='includeNaN')" )
```

output:

```
mfMax(x,nanflag='includeNaN') =

   NaN
```

```
x = [ MF_NAN, MF_NAN ]
call msDisplay( mfMax(x), "mfMax(x)" )
```

output:

```
mfMax(x) =

   NaN
```

msMax — mfArray max

Calling syntax:

```
call msMax( mfOut(v,i), A [, dim, nanflag] )
```

Description:

Computes the maximum value of the columns (default behavior) or of the rows of the `mfArray A` (real values only) and shows where this maximum occurs.

If `dim` is present, `A` is always considered as a matrix, and the vector `v` contains the maximum of each column (if `dim` = 1) or the maximum of each row (if `dim` = 2). indices of these maxima are returned in the vector `i`.

If `dim` is not present: if `A` is a vector, `msMax` returns the largest element in `v` and the corresponding index in `i`; otherwise it returns the max of each columns in `v` and the corresponding indices in `i`.

The optional argument `nanflag` has the same meaning as in the mfMax routine.

For a sparse `mfArray A`, the result is exactly the same as for the full corresponding matrix. In all cases (sparse or dense), it must contain real values.

See also: mfOut, msMin, mfExtrema

Example(s):

```
x = [ 1.0d0, MF_NAN, 5.0d0, 2.0d0, 4.0d0 ]
call msDisplay( x, "x" )
call msMax( mfOut(a,i), x )
call msDisplay( a, "max" )
call msDisplay( i, "for index" )
```

output:

```
 x =

    1.0000       NaN     5.0000    2.0000    4.0000

 max =

     5

 for index =

     3
```

```
call msMax( mfOut(a,i), x, nanflag='includeNaN' )
call msDisplay( a, "max" )
call msDisplay( i, "for index" )
```

output:

```
 max =

     NaN

 for index =

     2
```

mfMin — mfArray min

`mfMin` has two forms. The first one is used to compute the minimum value of the columns (default behavior) or of the row of an `mfArray`. The second one compares two matrices.

First interface:

```
function mfMin( A, dim, nanflag ) result( out )

   type(mfArray),    intent(in)           :: A
   integer,          intent(in), optional :: dim
   character(len=*), intent(in), optional :: nanflag
   type(mfArray)                          :: out
```

If `dim` is present, `A` is always considered as a matrix, and the returned vector contains the minimum of each column (if `dim` = 1) or the minimum of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfMin` returns the smallest element; otherwise it returns the min of each columns.

If the string `nanflag` is present, it must be equal to either `"omitNaN"`, or `"includeNaN"` (practically, the argument is not case sensitive). By default, `NaN` values are discarded (see examples below), so `"omitNaN"` is implied.

For a sparse `mfArray A`, the result is exactly the same as for the full corresponding matrix. In all cases (sparse or dense), it must contain real values.

Second interface:

```
function mfMin( A, B, nanflag ) result( out )

   type(mfArray), intent(in) :: A

   type(mfArray), intent(in) :: B
or real(kind=MF_DOUBLE), intent(in) :: B

   character(len=*), intent(in), optional :: nanflag
   type(mfArray) :: out
```

When the two arguments `A` and `B` are `mfArrays` (sparse storage is allowed), they must have the same shape and must contain real values; in this case, the *min* operation is applied element-wise. Otherwise, the second argument `B` may be a scalar `real`.

The optional argument `nanflag` has the same meaning as in the first interface.

See also: mfMax, msMin, mfExtrema

Example(s):

```
x = [ 1.0d0, MF_NAN, 5.0d0, 2.0d0, 4.0d0 ]
call msDisplay( mfMin(x), "mfMin(x)" )
```

output:

```
mfMin(x) =

   1.0000
```

.../...

```
x = [ 1.0d0, MF_NAN, 5.0d0, 2.0d0, 4.0d0 ]
call msDisplay( mfMin(x,nanflag='includeNaN'), "mfMin(x,nanflag='includeNaN')" )
```

output:

```
mfMin(x,nanflag='includeNaN') =

    NaN
```

```
x = [ MF_NAN, MF_NAN ]
call msDisplay( mfMin(x), "mfMin(x)" )
```

output:

```
mfMin(x) =

    NaN
```

msMin — mfArray min

Calling syntax:

```
call msMin( mfOut(v,i), A [, dim, nanflag] )
```

Description:

Computes the minimum value of the columns (default behavior) or of the rows of the `mfArray A` (real values only) and shows where this minimum occurs.

If `dim` is present, `A` is always considered as a matrix, and the vector `v` contains the minimum of each column (if `dim` = 1) or the minimum of each row (if `dim` = 2). indices of these minima are returned in the vector `i`.

If `dim` is not present: if `A` is a vector, `mfMax` returns the smallest element in `v` and the corresponding index in `i`; otherwise it returns the min of each columns in `v` and the corresponding indices in `i`.

The optional argument `nanflag` has the same meaning as in the mfMin routine.

For a sparse `mfArray A`, the result is exactly the same as for the full corresponding matrix. In all cases (sparse or dense), it must contain real values.

See also: mfOut, msMax, mfExtrema

Example(s):

```
x = [ 1.0d0, MF_NAN, 5.0d0, 2.0d0, 4.0d0 ]
call msDisplay( x, "x" )
call msMin( mfOut(a,i), x )
call msDisplay( a, "min" )
call msDisplay( i, "for index" )
```

output:

```
x =

   1.0000      NaN     5.0000    2.0000    4.0000

min =

    1

for index =

    1
```

```
call msMin( mfOut(a,i), x, nanflag='includeNaN' )
call msDisplay( a, "min" )
call msDisplay( i, "for index" )
```

output:

```
min =

    NaN

for index =

    2
```

mfExtrema — data extrema

Interface:

```
function mfExtrema( A, nanflag ) result( out )

   type(mfArray),    intent(in)           :: A
   character(len=*), intent(in), optional :: nanflag
   type(mfArray)                          :: out
```

Description:

Returns in an **mfArray** the two extreme values of an **mfArray**, in ascent order, *i. e.* the couple (*min*, *max*).

The optional argument **nanflag** has the same meaning as in the mfMin and mfMax routines.

See also: msMin, msMax

Example(s):

```
x = [ 5, 9, 3, 6, 0, 2 ]
call msDisplay( x, "x" )
call msDisplay( mfExtrema(x), "extrema" )
```

output:

```
x =

    5    9    3    6    0    2

extrema =

    0    9
```

```
x = [ 5, 9, MF_NAN, 6, 0, 2 ]
call msDisplay( x, "x" )
call msDisplay( mfExtrema(x,nanflag='includeNaN'), "extrema (include NaN)" )
```

output:

```
x =

    5    9    NaN   6    0    2

extrema (include NaN) =

    NaN   NaN
```

`mfSum` — columns sum

Interface:

```
function mfSum( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Computes the sum of the columns (or rows) of an `mfArray`.

If `dim` is present, `A` is always considered as a matrix, and the output contains the sum of each column (if `dim` = 1) or the sum of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfSum` returns the sum of all elements; otherwise it returns the sum of each columns.

See also: `mfProd`

`mfProd` columns product

Interface:

```
function mfProd( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Computes the product of the columns (or rows) of an `mfArray`. The behavior of this routine is similar to the *product* Fortran 90 intrinsic function.

If `dim` is present, `A` is always considered as a matrix, and the output contains the product of each column (if `dim` = 1) or the product of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfSum` returns the product of all elements; otherwise it returns the product of each columns.

Remarks: this routine cannot be applied to a boolean `mfArray`.

Warning: this function cannot be applied to sparse matrices.

See also: `mfSum`

mfSort — columns sort

Interface:

```
function mfSort( A, dim, mode ) result( out )

   type(mfArray),    intent(in)           :: A
   integer,          intent(in), optional :: dim
   character(len=*), intent(in), optional :: mode
   type(mfArray) :: out
```

Description:

If `dim` is not present, sorts the values of the vector `A`, and returns the sorted vector in an `mfArray`.

If `dim` is present, `A` may be a matrix. In this case, the sort is applied to the columns if `dim` is equal to 1, and to the rows if `dim` is equal to 2.

Remarks:

- this routine is appropriate for 2D arrays containing unrelated data in rows. If you want to sort a particular column, keeping the coherence of the lines (*i. e.* sorting the whole lines by key), you have to write your own code, based on the `msSort` version (see an example in the *Muesli User's Guide*); this means also that it is not possible to sort one specific column (or row) while keeping the others unchanged.
- this routine use a quick sort algorithm. By default, or if the optional argument `mode` is equal to `"ascending"` values are sorted in increasing order; on the contrary, if the optional argument `mode` is equal to `"descending"` values are sorted in decreasing order.
- currently, only rank-1 `mfArrays` (*i. e.* vectors) may contain *NaN* values. These *NaN*s are moved to the tail of the sorted vector.
- see `msSort` for an 'in-place' version.

Warning: If the `mfArray A` is a matrix, it must be real; for a complex vector `mfArray`, the sort is applied first on the module values, then on the phase-angle.

See also: `msSort`, `mfSortRows`

Example(s):

```
x = mf( [ 1.0d0, MF_NAN, 5.0d0, MF_NAN, 2.0d0, 4.0d0 ] )
call msDisplay( x, "x" )
call msDisplay( mfSort(x), "mfSort(x)" )
call msDisplay( mfSort(x,'descending'), "mfSort(x,'descending')" )
```

output:

```
x =

   1.0000      NaN     5.0000      NaN     2.0000    4.0000

mfSort(x) =

   1.0000    2.0000    4.0000    5.0000      NaN       NaN

mfSort(x,'descending') =

   5.0000    4.0000    2.0000    1.0000      NaN       NaN
```

msSort — columns sort

First calling syntax:

```
call msSort ( A [, dim, mode ] )
```

The 'in-place' version of mfSort.

Second calling syntax:

```
call msSort ( mfOut(v,i), A [, dim, mode ] )
```

Similar to mfSort but the result is returned in the mfArray v; in addition, indices of the sort are stored in the vector i.

See also: mfOut, mfSortRows

Example(s):

```
x = mfMagic(3)
call msDisplay( x, "x" )
call msSort( mfOut(a,i), x, dim=1 )
call msDisplay( a, "ascending column sort" )
call msDisplay( i, "with indices" )
call msSort( mfOut(a,i), x, dim=2, mode="descending" )
call msDisplay( a, "descending row sort" )
call msDisplay( i, "with indices" )
```

output:

```
x =

    8     3     4
    1     5     9
    6     7     2

ascending column sort =

    1     3     2
    6     5     4
    8     7     9

with indices =

    2     1     3
    3     2     1
    1     3     2

descending row sort =

    8     4     3
    9     5     1
    7     6     2

with indices =

    1     3     2
    3     2     1
    2     1     3
```

`mfIsSorted` — vector sort test

Interface:

```
bool = mfIsSorted( v, mode )
```

Description:

Checks that the `mfArray` vector `v` is sorted, according the mode, which can be equal to `"ascend"`, `"descend"` or `"either"`.

If the optional argument `mode` is not present, then an ascending sort is presumed.

This function returns a logical.

Remarks:

- currently, only rank-1 `mfArrays` (*i. e.* vectors) can be checked.
- the vector `v` may contain *NaN* values. The vector is considered as sorted if all *NaN*s are located at the end.
- for a complex vector `mfArray`, sorting concerns first the module, then the phase-angle.

See also: `mfSort`

`mfSortRows` — sort whole rows of a matrix

First interface:

```
function mfSortRows( A, col, mode ) result( out )

   type(mfArray),     intent(in)           :: A
   integer,           intent(in),          :: col
   character(len=*), intent(in), optional :: mode
   type(mfArray)                           :: out
```

Description:

Sorts the rows of the matrix `A` according to the specified column given by `col`, and returns the result in an `mfArray`.

Remarks:

- this routine use a quick sort algorithm. By default, or if the optional argument `mode` is equal to `"ascending"` values are sorted in increasing order; on the contrary, if the optional argument `mode` is equal to `"descending"` values are sorted in decreasing order.
- currently, the matrix `A` should not contain *NaN* values.

Warning: The `mfArray A` must be real.

Other interface:

```
function mfSortRows( A, cols ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: cols(:)
   type(mfArray)                       :: out
```

Description:

Same as before except that the sort of the rows is made by many columns, not necessarily in order, stored in `cols` (if present). The only constraint is that the size of `cols` must less or equal the number of columns of `A`. If `cols` is not present, then the sort is done by all columns, in order.

In this new interface, the direction of the sort is specified by the sign of the elements of the `cols` argument: when a column number is positive, the sort is ascending; in the opposite, *i. e.* when it is negative, the sort is descending.

See also: `mfSort`, `msSort`

Example(s):

```
x = mfMagic(3)
call msDisplay( x, "x" )
call msDisplay( mfSortRows(x,3), "mfSortRows(x,3)" )
call msDisplay( mfSortRows(x,1,"descending"), "mfSortRows(x,1,'descending')" )
```

···/···

output:

```
 x =

     8     3     4
     1     5     9
     6     7     2

 mfSortRows(x,3) =

     6     7     2
     8     3     4
     1     5     9

 mfSortRows(x,1,'descending') =

     6     7     2
     1     5     9
     8     3     4
```

```
x = mf( [ 3, 6, 2, 5 ] ) .vc.                                    &
    mf( [ 2, 1, 4, 2 ] ) .vc.                                    &
    mf( [ 1, 0, 0, 3 ] ) .vc.                                    &
    mf( [ 2, 1, 5, 1 ] ) .vc.                                    &
    mf( [ 2, 3, 3, 1 ] )
call msDisplay(x,"x")
call msDisplay( mfSortRows(x,[3]), "sortrows(x,[3])" )
call msDisplay( mfSortRows(x,[1,-3]), "sortrows(x,[1,-3])" )
```

output:

```
 x =

     3     6     2     5
     2     1     4     2
     1     0     0     3
     2     1     5     1
     2     3     3     1

 sortrows(x,[3]) =

     1     0     0     3
     3     6     2     5
     2     3     3     1
     2     1     4     2
     2     1     5     1

 sortrows(x,[1,-3]) =

     1     0     0     3
     2     1     5     1
     2     1     4     2
     2     3     3     1
     3     6     2     5
```

msSortRows — sort whole rows of a matrix

First calling syntax:

```
call msSortRows ( A, col [, mode ] )
```

or

```
call msSortRows ( A [, cols ] )
```

The 'in-place' version of mfSortRows.

Second calling syntax:

```
call msSortRows ( mfOut(v,i), A, col [, mode ] )
```

or

```
call msSortRows ( mfOut(v,i), A [, cols ] )
```

Similar to mfSortRows but the result is returned in the mfArray v; in addition, indices of the sort are stored in the vector i.

See also: mfOut, mfSortRows

Example(s):

```
x = mf( [ 3, 6, 2, 5 ] ) .vc.                                    &
    mf( [ 2, 1, 4, 2 ] ) .vc.                                    &
    mf( [ 1, 0, 0, 3 ] ) .vc.                                    &
    mf( [ 2, 1, 5, 1 ] ) .vc.                                    &
    mf( [ 2, 3, 3, 1 ] )
call msDisplay(x,"x")
call msSortRows( mfOut(a,i), x )
call msDisplay( a,"fully row sorted", i,"with row indices" )
```

output:

```
 x =

     3     6     2     5
     2     1     4     2
     1     0     0     3
     2     1     5     1
     2     3     3     1

 fully row sorted =

     1     0     0     3
     2     1     4     2
     2     1     5     1
     2     3     3     1
     3     6     2     5

 with row indices =

     3
     2
     4
     5
     1
```

`mfDiff` — difference and approximate derivative

Interface:

```
function mfDiff( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Computes the difference of two consecutive values of the columns (or rows) of an `mfArray A`.

If `dim` is present, `A` is always considered as a matrix, and the routine computes the difference of two consecutive values for each column (if `dim` = 1) or for each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, then `mfDiff` computes the difference of two consecutive values in the natural way; otherwise the same process is applied for each columns.

Remark: this function cannot be applied to sparse matrices.

See also: `mfGradient`, `msGradient`

Example(s):

```
x = mf( [ 1, 2, 3, 2, 1 ] )
call msDisplay( x, "x" )
call msDisplay( mfDiff(x), "mfDiff(x)" )
```

output:

```
 x =

     1     2     3     2     1

 mfDiff(x) =

     1     1    -1    -1
```

`mfGradient` — approximate 1D gradient

Calling syntax:

```
D = mfGradient( F, h [, dim, location] )
```

Description:

Computes the gradient `D` of the rank-1 `mfArray F`.

If `dim` is present, `F` is always considered as a matrix, and the routine computes the gradient for each column (if `dim` = 1) or for each row (if `dim` = 2).

If `dim` is not present: if `F` is a vector, then `mfGradient` computes the gradient in the natural way; otherwise the same process is applied for each columns.

The data is supposed to be equally spaced with `h` as increment. `h` may be a double precision real or a scalar `mfArray`; it may have negative values.

If the optional argument `location` (character string) is equal to `"centered"` then the gradient is computed on an internal centered, staggered grid; otherwise it is computed at the same points as `F`.

Remark: The gradient is computed using second order formulæ.

See also: `msGradient`, `mfDiff`, `mfOut`

Example(s):

```
integer :: nx = 5
real(kind=MF_DOUBLE) :: lx = 1.0d0, hx
x = mfLinSpace(0.0d0,lx,nx)
F = x**2
call msDisplay( F, "F" )
hx = lx/(nx-1)
Fx = mfGradient( F, hx )
call msDisplay( Fx, "Fx = grad_x(F)" )
```

output:

```
 F =

    0.0000    0.0625    0.2500    0.5625    1.0000


 Fx = grad_x(F) =

    0.0000    0.5000    1.0000    1.5000    2.0000
```

```
fx = mfGradient( F, hx, "centered" )
call msDisplay( fx, "grad_x(F)" )
```

output:

```
 grad_x(z) =

    0.2500    0.7500    1.2500    1.7500
```

`msGradient` — approximate 2D gradient

Calling syntax:

```
call msGradient( mfOut( Fi, Fj ), F, hi, hj [, location] )
```

Description:

Computes the gradient of the rank-2 `mfArray F`, along the two dimensions. The data are supposed to be equally spaced in both direction, with `hi` and `hj` as increments, which may have negative values.

As a result, `Fi` is the gradient of `F` with respect to increasing indices i (*i. e.* along the rows) and `Fj` is the gradient of `F` with respect to increasing indices j (*i. e.* along the columns).

If the optional argument `location` (character string) is equal to `"centered"` then the two components of the gradient are computed on an internal centered, staggered grid; otherwise they are computed at the same points as `F`.

Remarks:

- this function cannot be applied to sparse matrices. For all cases and all points, the gradient is computed using second order formulæ.
- if the data matrix `F` is associated to coordinates X and Y, `hi` and `hj` may be chosen to give directly the two components F_x and F_y (see the *Muesli User Guide*, section entitled "Data Analysis with mfArrays").

See also: `mfGradient`, `mfDiff`, `mfOut`

Example(s):

```
integer :: nx = 5, ny = 5
real(kind=MF_DOUBLE) :: lx = 1.0d0, ly = 2.0d0, hx, hy
call msMeshGrid( mfOut(x,y), mfLinSpace(0.0d0,lx,nx),                &
                           .t. mfLinSpace(ly,0.0d0,ny) )
call msDisplay( x, "x", y, "y" )
F = (2*x+y)**2
call msDisplay( F, "F" )
hx =  lx/(nx-1)
hy = -ly/(ny-1)
call msGradient( mfOut(Fy,Fx), F, hy, hx )
call msDisplay( Fx, "Fx = grad_x(F)", Fy, "Fy = grad_y(F)" )
```

output:

```
 x =

   0.0000    0.2500    0.5000    0.7500    1.0000
   0.0000    0.2500    0.5000    0.7500    1.0000
   0.0000    0.2500    0.5000    0.7500    1.0000
   0.0000    0.2500    0.5000    0.7500    1.0000
   0.0000    0.2500    0.5000    0.7500    1.0000
```

· · · / · · ·

```
y =

   2.0000    2.0000    2.0000    2.0000    2.0000
   1.5000    1.5000    1.5000    1.5000    1.5000
   1.0000    1.0000    1.0000    1.0000    1.0000
   0.5000    0.5000    0.5000    0.5000    0.5000
   0.0000    0.0000    0.0000    0.0000    0.0000

F =

   4.0000    6.2500    9.0000   12.2500   16.0000
   2.2500    4.0000    6.2500    9.0000   12.2500
   1.0000    2.2500    4.0000    6.2500    9.0000
   0.2500    1.0000    2.2500    4.0000    6.2500
   0.0000    0.2500    1.0000    2.2500    4.0000

Fx = grad_x(F) =

    8    10    12    14    16
    6     8    10    12    14
    4     6     8    10    12
    2     4     6     8    10
    0     2     4     6     8

Fy = grad_y(F) =

    4     5     6     7     8
    3     4     5     6     7
    2     3     4     5     6
    1     2     3     4     5
    0     1     2     3     4
```

```
  call msGradient( mfOut(Fy,Fx), F, hy, hx, "centered" )
  call msDisplay( Fx, "Fx = grad_x(F)", Fy, "Fy = grad_y(F)" )
```

output:

```
Fx = grad_x(F) =

    8    10    12    14
    6     8    10    12
    4     6     8    10
    2     4     6     8

Fy = grad_y(F) =

    4     5     6     7
    3     4     5     6
    2     3     4     5
    1     2     3     4
```

`mfMean` — columns mean

Interface:

```
function mfMean( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Computes the mean of the columns (or rows) of an `mfArray`, as $\frac{1}{N}\sum_{i=1}^{N} a_i$.

If `dim` is present, `A` is always considered as a matrix, and the output contains the mean of each column (if `dim` = 1) or the mean of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfMean` returns the mean of all elements; otherwise it returns the mean of each columns.

Warning:

- this function cannot be applied to sparse matrices;
- the user must take care that the `mfArray A` doesn't contain any *NaN* values.

See also: `mfMedian`, `mfVar`, `mfStd`, `mfRMS`, `mfMoments`

`mfMedian` columns median

Interface:

```
function mfMedian( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Computes the median of the columns (or rows) of an `mfArray`. The median value of a vector is, after sorting this vector, the value located in the middle.

If `dim` is present, `A` is always considered as a matrix, and the output contains the median of each column (if `dim` = 1) or the median of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfMedian` returns the median of all elements; otherwise it returns the median of each columns.

Warning:

- this function cannot be applied to sparse matrices;
- the user must take care that the `mfArray A` doesn't contain any *NaN* values.

See also: `mfMean`, `mfVar`, `mfStd`, `mfMoments`

`mfVar` — columns variance

Interface:

```
function mfVar( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Computes the variance of the columns (or rows) of an `mfArray`, as $\frac{1}{N}\sum_{i=1}^{N}(a_i - \bar{a})^2$, with $\bar{a} = \frac{1}{N}\sum_{i=1}^{N} a_i$.

If `dim` is present, `A` is always considered as a matrix, and the output contains the variance of each column (if `dim` = 1) or the variance of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfVar` returns the variance of all elements; otherwise it returns the variance of each columns.

Warning:

- this function cannot be applied to sparse matrices;
- the user must take care that the `mfArray A` doesn't contain any *NaN* values.

See also: `mfMean`, `mfMedian`, `mfStd`, `mfMoments`

`mfStd` columns standard deviation

Interface:

```
function mfStd( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Computes the standard deviation of the columns (or rows) of an `mfArray`. The standard deviation is the square root of the variance.

If `dim` is present, `A` is always considered as a matrix, and the output contains the standard deviation of each column (if `dim` = 1) or the standard deviation of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfStd` returns the standard deviation of all elements; otherwise it returns the standard deviation of each columns.

Warning:

- this function cannot be applied to sparse matrices;
- the user must take care that the `mfArray A` doesn't contain any *NaN* values.

See also: `mfMean`, `mfRMS`, `mfMedian`, `mfVar`, `mfMoments`

`mfRMS` root mean square

Interface:

```
function mfRMS( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Computes the Root-Mean-Square of the columns (or rows) of an `mfArray`.

The RMS is defined by:

$$a_{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} a_i^2}$$

where N is the number of elements x_i. The following relation holds:

$$a_{RMS}^2 = \bar{a}^2 + \sigma_a^2$$

so the RMS of a signal is synonym to its standard deviation only when the mean is zero.

If `dim` is present, `A` is always considered as a matrix, and the output contains the mean of each column (if `dim` = 1) or the RMS of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfRMS` returns the RMS of all elements; otherwise it returns the RMS of each columns.

Warning:

- this function can be applied only to real, dense matrices;
- the user must take care that the `mfArray A` doesn't contain any *NaN* values.

See also: `mfMean`, `mfMedian`, `mfVar`, `mfStd`, `mfMoments`

`msHist` — data histogram

Calling syntax:

```
call msHist( mfOut( num [, x_bin ] ), x, x_min, x_max, n_bin )
```

computes a data histogram.

The vector `mfArray x` contains the data. The histogram is computed for data ranged from `x_min` to `x_max` using a number of bins equal to `n_bin`.

The output `mfArray num` is a vector of the bins values (length equal to `n_bin`).

If present, the output `mfArray x_bin` is a vector containing the abscissas of the bins (length equal to `n_bin`+1).

Note: to process discrete data, it is recommended to choose `x_min` as the minimum integer value minus 1/2, and `x_max` as the maximum integer value plus 1/2 (see example below).

See also: mf/msBar, mf/msPlotHist, msCumulHist, mfOut

Example(s):

```
x = .t. mf( [ 12, 11, 8, 10, 6, 8, 9, 10, 12, 9, 10, 8, 6, 10 ] )
call msHist( mfOut( y, z ), x, 6.0d0-0.5d0, 12.0d0+0.5d0, 7 )
call msDisplay( y, "y (hist)", z, "z (x_bin)" )
```

output:

```
 y (hist) =

     2
     0
     3
     2
     4
     1
     2


 z (x_bin) =

    5.5000
    6.5000
    7.5000
    8.5000
    9.5000
   10.5000
   11.5000
   12.5000
```

`mfMoments` few first moments of a distribution

Interface:

```
function mfMoments( v ) result( out )

   type(mfArray), intent(in) :: v
   type(mfArray)             :: out
```

Description:

Computes some moments of the data distribution provided in the vector `mfArray v`.

It returns a vector `mfArray` of length 6, containing in order:

- the mean value of data
- the average deviation
- the standard deviation
- the variance
- the skewness
- the kurtosis

Warning:

- this function cannot be applied to sparse matrices;
- the user must take care that the `mfArray v` doesn't contain any *NaN* values.

See also: `mfMean`, `mfMedian`, `mfVar`, `mfStd`

`mfSmooth` — smoothing of vector values

Interface:

```
function mfSmooth( v [, span ] ) result( out )

   type(mfArray), intent(in)           :: v
   integer,       intent(in), optional :: span
   type(mfArray)                       :: out
```

Description:

Applies the moving average method to the vector `mfArray v`. It is equivalent to a lowpass filter.

The width of the sliding window is `span` which must be a non negative integer. The `span` value should not be greater than the number of elements in `v`. The default value is 5.

See also: `mfSpline`

`mfXCorr` autocorrelation of a vector data

Interface:

```
function mfXCorr( v [, maxlag, scale ] ) result( out )

   type(mfArray),    intent(in)           :: v
   integer,          intent(in), optional :: maxlag
   character(len=*), intent(in), optional :: scale
   type(mfArray)                          :: out
```

Description:

Computes the autocorrelation of the real vector `mfArray v`.

The output `mfArray out` is a vector having an odd number of elements, and is symmetric with respect to its center.

Its maximum corresponds to a lag equal to zero. This maximum is equal to 1 when the optional argument `scale` is set to `"normalized"`.

The maximum lag used can be set by using the optional `maxlag` argument. By default, its value is equal to $N-1$, where N is the length of the vector `v`.

See also: `mfXCorr2`

`mfXCorr2` autocorrelation of a matrix data

Interface:

```
function mfXCorr2( A [, maxlag_r, maxlag_c, scale ] ) result( out )

   type(mfArray),    intent(in)           :: A
   integer,          intent(in), optional :: maxlag_r, maxlag_c
   character(len=*), intent(in), optional :: scale
   type(mfArray)                          :: out
```

Description:

Computes the autocorrelation of the real matrix `mfArray A`. It is similar to the `mfXCorr` routine.

The output `mfArray out` is a matrix having an odd number of rows and columns, and is symmetric with respect to its center.

Its maximum corresponds to a lag equal to zero. This maximum is equal to 1 when the optional argument `scale` is set to `"normalized"`.

The maximum lag used for the row shift (resp. the column shift) can be set by using the optional `maxlag_r` argument (resp. `maxlag_c`). By default, their value is equal to the size of thecorresponding dimension.

See also: `mfXCorr`

`mfFFT` discrete Fast Fourier transformation

Interface:

```
function mfFFT( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Applies a Fast Fourier transformation to the columns (or rows) of an `mfArray`, real or complex. The output `mfArray` is always complex.

If `dim` is present, `A` is always considered as a matrix, and the output contains the transformation coefficients of each column (if `dim` = 1) or the transformation of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfFFT` returns the transformation of all elements; otherwise it returns the transformation of each columns.

Warning: this function cannot be applied to sparse matrices.

Remarks: abscissa are supposed to be equally spaced.

See also: `mfInvFFT`, `mfFFT2`, `mfFourierCos`, `mfFourierSin`, `mfFourierLeg`

`mfInvFFT` inverse Fast Fourier transformation

Interface:

```
function mfInvFFT( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

This is the inverse tranformation of `mfFFT`. So, for any vector `v`, we have:

`v = mfInvFFT(mfFFT(v))`

See also: `mfFFT`, `mfFFT2`, `mfFourierCos`, `mfFourierSin`, `mfFourierLeg`

`mfFFT2` 2D discrete Fast Fourier transformation

Interface:

```
function mfFFT2( A ) result( out )

   type(mfArray), intent(in) :: A
   type(mfArray)             :: out
```

Description:

Applies a two dimensional Fast Fourier transformation to the `mfArray A`, real or complex. The output `mfArray` is always complex.

Warning: this function cannot be applied to sparse matrices.

Remarks: abscissa are supposed to be equally spaced for both dimensions.

See also: `mfInvFFT2`, `mfFFT`, `mfFourierCos`, `mfFourierSin`, `mfFourierLeg`

`mfInvFFT2` inverse 2D Fast Fourier transformation

Interface:

```
function mfInvFFT2( A ) result( out )

   type(mfArray), intent(in) :: A
   type(mfArray)             :: out
```

Description:

This is the inverse tranformation of `mfFFT2`. So, for any complex matrix `A`, we have:

`A = mfInvFFT(mfFFT(A))`

See also: `mfFFT2`, `mfFFT`, `mfFourierCos`, `mfFourierSin`, `mfFourierLeg`

`mfFourierCos` discrete Fourier cosine transformation

Interface:

```
function mfFourierCos( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Applies a discrete Fourier cosine transformation to the columns (or rows) of an `mfArray`.

If `dim` is present, `A` is always considered as a matrix, and the output contains the transformation coefficients of each column (if `dim` = 1) or the transformation of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfFourierCos` returns the transformation of all elements; otherwise it returns the transformation of each columns.

Warning: this function cannot be applied to sparse matrices. Moreover, the array must contain real values.

Remarks:

- abscissa are supposed to be equally spaced;
- actually, data are already considered symmetric on the interval $[-n+1, n+1]$; therefore, there doesn't exist any constraint on the value of the data vector.

See also: `mfInvFourierCos`, `mfFFT`, `mfFFT2`, `mfFourierSin`, `mfFourierLeg`

`mfInvFourierCos` inverse Fourier cosine transformation

Interface:

```
function mfInvFourierCos( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

This is the inverse tranformation of `mfFourierCos`. So, for any vector `v`, we have:

```
v = mfInvFourierCos(mfFourierCos(v))
```

See also: `mfFourierCos`, `mfFFT`, `mfFFT2`, `mfFourierSin`, `mfFourierLeg`

`mfFourierSin` — Fourier sine transformation

Interface:

```
function mfFourierSin( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Applies a discrete Fourier sine transformation to the columns (or rows) of an `mfArray`.

If `dim` is present, `A` is always considered as a matrix, and the output contains the transformation coefficients of each column (if `dim` $= 1$) or the transformation of each row (if `dim` $= 2$).

If `dim` is not present: if `A` is a vector, `mfFourierSin` returns the transformation of all elements; otherwise it returns the transformation of each columns.

Warning: this function cannot be applied to sparse matrices. Moreover, the array must contain real values.

Remarks:

- abscissa are supposed to be equally spaced;
- actually, data are already considered antisymmetric on the interval $[-n+1, n+1]$; therefore, there exists some constraints: the conditions $y(1) = y(n+1) = 0$ are required.

See also: `mfInvFourierSin`, `mfFFT`, `mfFFT2`, `mfFourierCos`, `mfFourierLeg`

`mfInvFourierSin` inverse Fourier sine transformation

Interface:

```
function mfInvFourierSin( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

This is the inverse tranformation of `mfFourierSin`. So, for any vector `v`, we have:

```
v = mfInvFourierSin(mfFourierSin(v))
```

See also: `mfFourierSin`, `mfFFT`, `mfFFT2`, `mfFourierCos`, `mfFourierLeg`

`mfFourierLeg` Fourier-Legendre transform

Interface:

```
function mfFourierLeg( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

Applies a discrete Fourier-Legendre transformation to the columns (or rows) of an `mfArray`.

This transformation is also called "discrete spherical Fourier transform".

If `dim` is present, `A` is always considered as a matrix, and the output contains the transformation coefficients of each column (if `dim` = 1) or the transformation of each row (if `dim` = 2).

If `dim` is not present: if `A` is a vector, `mfFourierLeg` returns the transformation of all elements; otherwise it returns the transformation of each columns.

Warning: this function cannot be applied to sparse matrices. Moreover, the array must contain real values.

Remarks:

- abscissa are supposed to be spanned over $[-1, 1]$ like a $\cos(\theta)$ function, with θ equally spaced;
- actually, data are already considered symmetric on the interval $[-n + 1, n + 1]$; therefore, there doesn't exist any constraint on the value of the data vector.

See also: `mfInvFourierLeg`, `mfLegendre`, `mfFFT`, `mfFFT2`, `mfFourierSin`, `mfFourierCos`

`mfInvFourierLeg` inverse Fourier-Legendre transform

Interface:

```
function mfInvFourierLeg( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

This is the inverse tranformation of `mfFourierLeg`. So, for any vector `v`, we have:

```
v = mfInvFourierLeg(mfFourierLeg(v))
```

See also: `mfFourierLeg`, `mfLegendre`, `mfFFT`, `mfFFT2`, `mfFourierSin`, `mfFourierCos`

1.4 Operators

`+`	mfArray sum
`-`	mfArray difference
`mfMul`, `.x.`	mfArray product
`mfCross`	cross-product of two vectors
`mfKron`	Kronecker tensor product
`*`	mfArray element-wise product
`/`	mfArray element-wise quotient
`**`	mfArray element-wise exponentiation
`.t.`	mfArray transposition
`.h.`	mfArray conjugate transposition
`.vc.`	vertical concatenation
`.hc.`	horizontal concatenation
`msHorizConcat`	sparse matrix horizontal concatenation
`mfColPerm`, `msColPerm`	columns permutation
`mfRowPerm`, `msRowPerm`	rows permutation
`mfInvPerm`	inverse of a permutation
`mfColScale`, `msColScale`	columns scaling
`mfRowScale`, `msRowScale`	rows scaling
`msColAutoScale`	columns auto scaling
`msRowAutoScale`	rows auto scaling
`>=`, `>`	mfArray relational operator
`<=`, `<`	mfArray relational operator
`==`, `/=`	mfArray relational operator
`mfAll`, `mfAny`	mfArray boolean operator
`mfColon`	mfArray sequence constructor
`.not.`, `.and.`, `.or.`	mfArray logical operator
`.eqv.`, `.neqv.`	mfArray logical operator
`mfIsMember`	elements members of a set
`mfIntersect`	common elements of two sets
`mfUnion`	all elements of two sets
`mfUnique`	unique elements of a set

See also:

Core Routines

File Input/Output

Data Analysis Functions

Elementary Math Functions

Specialized Math Functions

Elementary Matrix Manipulation Functions

Matrix Functions

Polynomial Functions

Optimization and Function Functions

Sparse Matrices

+ mfArray sum

Calling syntax:

```
C = A + B
```

Description:

Computes the sum of two `mfArray`s of same shape.

One of them may be a scalar of type `real` (single or double) or `complex`; in such a case, if the `mfArray` is not a scalar, then the operation is made on all elements.

Remarks: `A` and `B` may have a dense or sparse structure.

See also: -

- mfArray difference

Calling syntax:

```
C = A - B
```

Description:

Computes the difference of two `mfArray`s of same shape.

One of them may be a scalar of type `real` (single or double) or `complex`; in such a case, if the `mfArray` is not a scalar, then the operation is made on all elements.

Remarks: `A` and `B` may have a dense or sparse structure.

See also: +

`mfMul, .x.` — mfArray product

Interface:

```
function mfMul( A, B, transp ) result( out )

   type(mfArray), intent(in)           :: A, B
   integer,       intent(in), optional :: transp
   type(mfArray)                       :: out
```

`A .x. B` is a shortcut for writing: `mfMul( A, B )`

Description:

Computes the matrix product $A * B$ (which is not commutative).

When the optional argument `transp` is present, it computes $A' * B$ (transp=1) or $A * B'$ (transp=2). This latter option is valid only for dense `mfArray`s.

The shape of the two matrices must respect the classical matrix-product rule, according to the value of `transp`.

Remarks: the element-wise operation is done with the operator '`*`'.

This routine can be applied to dense or sparse `mfArray`s of any type (real or complex).

See also: `mfCross`

`mfCross` — cross-product of two vectors

Interface:

```
function mfCross( u, v ) result( out )

   type(mfArray), intent(in) :: u, v
   type(mfArray)             :: out
```

Description:

Computes the cross-product of two vectors in R^3.

See also: `mfMul`

`mfKron` — Kronecker tensor product

Interface:

```
function mfKron( A, B ) result( out )

   type(mfArray), intent(in) :: A, B
   type(mfArray)             :: out
```

Description:

Computes the Kronecker product of `A` and `B`. If the shape of `A` is (m, n) and the shape of `B` is (p, q), then the shape of the Kronecker product will be $(m\,p, n\,q)$.

The `mfArray`s `A` and `B` may be dense or sparse.

Remark: Currently, only real `mfArray`s are accepted.

See also: `mfMul`

* mfArray element-wise product

Calling syntax:

```
C = A * B
```

Description:

Computes the product (element-wise) of two `mfArray`s of same shape.

One of them may be a scalar of type `real` (single or double) or `complex`; in such a case, if the `mfArray` is not a scalar, then the operation is made on all elements.

Only for this routine, `B` may be of `type mfUnit`. This feature allows the user to change the physical unit of the `mfArray A`.

Remarks: `A` and `B` may have a dense or sparse structure.

See also: `mfMul`, `/`

/ mfArray element-wise quotient

Calling syntax:

```
C = A / B
```

Description:

Computes the division (element-wise) of two `mfArray`s of same shape.

One of them may be a scalar of type `real` (single or double) or `complex`; in such a case, if the `mfArray` is not a scalar, then the operation is made on all elements.

Remarks: `A` and `B` may have a dense or sparse structure.

See also: `mfLDiv`, `mfRDiv`, `*`

** mfArray element-wise exponentiation

Calling syntax:

```
C = A ** i
```

Description:

Computes the power (element-wise) of the `mfArray A` to the exponent `i`, which must be a scalar, of type `integer` or `real`.

Remarks: `A` may have a dense or sparse structure.

`.t.` mfArray transposition

Calling syntax:

```
C = .t. A
```

Description:

The prefixed operator '`.t.`' transposes the `mfArray A`.

A warning is emitted if this routine is applied to a complex `mfArray` (in this case, the operator '`.h.`' should be used instead).

Remarks: `A` may have a dense or sparse structure. This operator can also be applied to `integer` or `real` arrays, of rank 1 or 2 (*i. e.* vector or matrix); in all cases it returns an `mfArray`.

`.h.` mfArray conjugate transposition

Calling syntax:

```
C = .h. A
```

Description:

The prefixed operator '`.h.`' returns the conjugate transpose of the `mfArray A`.

Remarks: `A` may have a dense or sparse structure. This operator should be applied to `complex` arrays, of rank 1 or 2 (*i. e.* vector or matrix); in all cases it returns an `mfArray`. If `A` is a real `mfArray`, it is converted in a complex `mfArray`.

See also: `.t.`

`.vc.` vertical concatenation

Calling syntax:

```
C = A .vc. B
```

Description:

Vertical concatenation of `mfArrays` (dense and/or sparse).

`A` and `B` must have the same number of columns.

One of the two `mfArrays` can be empty: it enables the construction of a matrix inside a loop, from an empty one.

Remark: This operator also accepts usual Fortran 90 1-rank numeric array (*i. e.* either real or complex).

See also: `.hc.`

`.hc.` horizontal concatenation

Calling syntax:

```
C = A .hc. B
```

Description:

Horizontal concatenation of `mfArray`s (dense and/or sparse).

`A` and `B` must have the same number of rows.

One of the two `mfArray`s can be empty: it enables the construction of a matrix inside a loop, from an empty one.

Remarks:

- when working with sparse `mfArray`s, the subroutine `msHorizConcat` is much more efficient.
- this operator also accepts usual Fortran 90 1-rank numeric array (*i. e.* either real or complex).

See also: `msHorizConcat`, `.vc.`

`msHorizConcat` sparse matrix horizontal concatenation

Interface:

```
subroutine msHorizConcat( A, data )

   type(mfArray), intent(in out) :: A
   type(mfArray), intent(in)     :: data
```

Description:

This subroutine version of `operator(.hc.)` is intended to be more efficient, because: (i) modifications of `A` are made "in place" and (ii) allocation of space is anticipated and over-dimensioned.

`A` can be empty: it enables the construction of a matrix inside a loop, from an empty one.

The `mfArrays A` must be sparse; the `mfArray data` must be sparse if it is a matrix, but can be dense if it is a vector.

Restrictions: the two `mfArrays` must have the same data type.

Remarks: use `operator(.hc.)` instead of for dense matrices.

See also: `.hc.`, `.vc.`

`mf/msColPerm`, columns permutation

First calling syntax:

```
B = mfColPerm( A, p )
```

Description:

Returns the columns permutation of the `mfArray A` (may be of any type, dense or sparse), using the permutation `p`: the j^{th} column of `B` is the $p(j)^{th}$ column of `A`.

In Matlab syntax or Fortran 90 notation, it computes $B = A(:, p)$.

`p` is either an ordinary vector `mfArray`, containing the (integer) indices of the permutation, or a true permutation `mfArray` (see `mfPerm`).

The second calling syntax:

```
call msColPerm( A, p )
```

applies the same operation, but in-place.

Remark: This routine is much more efficient than computing a column's permutation by calling `mfGet(A,MF_ALL,p)`.

See also: `mf/msRowPerm`, `mf/msColScale`, `mf/msRowScale`

`mf/msRowPerm`, rows permutation

First calling syntax:

```
B = mfRowPerm( A, p )
```

Description:

Returns the rows permutation of the `mfArray A` (may be of any type, dense or sparse), using the permutation `p`: the i^{th} row of `B` is the $p(i)^{th}$ row of `A`.

In Matlab syntax or Fortran 90 notation, it computes $B = A(p, :)$.

`p` is either an ordinary vector `mfArray`, containing the (integer) indices of the permutation, or a true permutation `mfArray` (see `mfPerm`).

The second calling syntax:

```
call msRowPerm( A, p )
```

applies the same operation, but in-place.

Remark: This routine is much more efficient than computing a row's permutation by calling `mfGet(A,p,MF_ALL)`.

See also: `mf/msColPerm`, `mf/msColScale`, `mf/msRowScale`

`mfInvPerm`, inverse of a permutation

Calling syntax:

```
p_inv = mfInvPerm( p )
```

Description:

Inverts the permutation `p`.

See also: `mf/msColPerm`, `mf/msRowPerm`

`mf/msColScale`, columns scaling

First calling syntax:

```
B = mfColScale( A, s )
```

Description:

Returns the columns scaling of the `mfArray A` (must be numeric, dense or sparse), using the dense vector `s`: the j^{th} column of `B` is the j^{th} column of `A` multiplied by $s(j)$.

In Matlab syntax, it computes $B = A * diag(s)$.

Restrictions: the vector `s` must be a real `mfArray`.

The second calling syntax:

```
call msColScale( A, s )
```

applies the same operation, but in-place.

See also: `mf/msColPerm`, `mf/msRowPerm`, `mf/msRowScale`

`mf/msRowScale`, rows scaling

First calling syntax:

```
B = mfRowScale( A, s )
```

Description:

Returns the rows scaling of the `mfArray A` (must be numeric, dense or sparse), using the dense vector `s`: the i^{th} row of `B` is the i^{th} row of `A` multiplied by $s(i)$.

In Matlab syntax, it computes $B = diag(s) * A$.

Restrictions: the vector `s` must be a real `mfArray`.

The second calling syntax:

```
call msRowScale( A, s )
```

applies the same operation, but in-place.

See also: `mf/msColPerm`, `mf/msRowPerm`, `mf/msColScale`

`msColAutoScale`, columns auto scaling

Calling syntax:

```
call msColAutoScale( mfOut(s), A )
```

scales the columns of the `mfArray A` (must be numeric, dense or sparse), in such a way that the maximum magnitude of each column is equal to 1.

It ouputs the scaling row vector `s`, which is a real `mfArray`.

See also: `msRowAutoScale`, `mf/msColScale`, `mf/msRowScale`

`msRowAutoScale`, rows auto scaling

Calling syntax:

```
call msRowAutoScale( mfOut(s), A )
```

scales the rows of the `mfArray A` (must be numeric, dense or sparse), in such a way that the maximum magnitude of each row is equal to 1.

It ouputs the scaling column vector `s`, which is a real `mfArray`.

See also: `msColAutoScale`, `mf/msColScale`, `mf/msRowScale`

`>=` mfArray relational operator

Calling syntax:

```
C = A >= B
```

Description:

Returns a boolean `mfArray` from two dense `mfArray`s, by making the comparison element-wise.

`A` and `B` must have the same shape and must be, of course, of type real.

`B` may be a scalar, `real` or even `integer`.

Remarks: a boolean `mfArray` contains *TRUE* and *FALSE* values only. It can be displayed as any `mfArray` and can occur in logical operators (‘`.not.`’, ‘`.and.`’, ‘`.or.`’, *etc.*).

See also: `>`, `<=`, `<`, `==`, `/=`

> mfArray relational operator

Calling syntax:

```
C = A > B
```

Description:

Returns a boolean `mfArray` from two dense `mfArray`s, by making the comparison element-wise.

`A` and `B` must have the same shape and must be, of course, of type real.

`B` may be a scalar, `real` or even `integer`.

Remarks: a boolean `mfArray` contains *TRUE* and *FALSE* values only. It can be displayed as any `mfArray` and can occur in logical operators ('`.not.`', '`.and.`', '`.or.`', *etc.*).

See also: `>=`, `<=`, `<`, `==`, `/=`

`<=` mfArray relational operator

Calling syntax:

```
C = A <= B
```

Description:

Returns a boolean `mfArray` from two dense `mfArray`s, by making the comparison element-wise.

`A` and `B` must have the same shape and must be, of course, of type real.

`B` may be a scalar, `real` or even `integer`.

Remarks: a boolean `mfArray` contains *TRUE* and *FALSE* values only. It can be displayed as any `mfArray` and can occur in logical operators (‘`.not.`’, ‘`.and.`’, ‘`.or.`’, *etc.*).

See also: `<`, `>=`, `>`, `==`, `/=`

< mfArray relational operator

Calling syntax:

```
C = A < B
```

Description:

Returns a boolean `mfArray` from two dense `mfArray`s, by making the comparison element-wise.

`A` and `B` must have the same shape and must be, of course, of type real.

`B` may be a scalar, `real` or even `integer`.

Remarks: a boolean `mfArray` contains *TRUE* and *FALSE* values only. It can be displayed as any `mfArray` and can occur in logical operators ('`.not.`', '`.and.`', '`.or.`', *etc.*).

See also: `<=`, `>=`, `>`, `==`, `/=`

== mfArray relational operator

Calling syntax:

```
C = A == B
```

Description:

Returns a boolean `mfArray` from two dense `mfArray`s, by making the comparison element-wise.

`A` and `B` must have the same shape and must be, of course, of type real.

`B` may be a scalar, `real` or even `integer`.

Remarks: a boolean `mfArray` contains *TRUE* and *FALSE* values only. It can be displayed as any `mfArray` and can occur in logical operators ('`.not.`', '`.and.`', '`.or.`', *etc.*).

See also: `/=`, `mfIsEqual`, `<=`, `<`, `>=`, `>`

`/=` mfArray relational operator

Calling syntax:

```
C = A /= B
```

Description:

Returns a boolean `mfArray` from two dense `mfArray`s, by making the comparison element-wise.

`A` and `B` must have the same shape and must be, of course, of type real.

`B` may be a scalar, `real` or even `integer`.

Remarks: a boolean `mfArray` contains *TRUE* and *FALSE* values only. It can be displayed as any `mfArray` and can occur in logical operators ('`.not.`', '`.and.`', '`.or.`', *etc.*).

See also: `==`, `mfIsEqual`, `<=`, `<`, `>=`, `>`

`mfAll` — mfArray boolean operator

Interface:

```
function mfAll( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

If `A` is a vector `mfArray`, returns a scalar boolean `mfArray` which is *TRUE* (*i. e.* 1) if all elements are *TRUE*.

If `A` is a matrix, the routine works down the columns, returning a row vector `mfArray`.

If `dim` is present, it indicates on which dimension the routine must be applied.

See also: `All`, `mfAny`

`mfAny` — mfArray boolean operator

Interface:

```
function mfAny( A, dim ) result( out )

   type(mfArray), intent(in)           :: A
   integer,       intent(in), optional :: dim
   type(mfArray)                       :: out
```

Description:

If `A` is a vector `mfArray`, returns a scalar boolean `mfArray` which is *TRUE* (*i. e.* 1) if at least one element is *TRUE*.

If `A` is a matrix, the routine works down the columns, returning a row vector `mfArray`.

If `dim` is present, it indicates on which dimension the routine must be applied.

See also: `Any`, `mfAll`

`mfColon` — mfArray sequence constructor

Interface:

```
A = mfColon( start, end [, step, tol ] )
```

Description:

Returns an `mfArray` containing a sequence: a list of reals equally spaced, from `start` to `end`, with the specified `step`.

By default (*i. e.* when `step` is not present), increment is unity.

Arguments `start`, `end` and `step` may be of type integer or real (single or double precision), but all of same kind and precision.

Because of the rounding errors, the last number of the sequence seldom matches to the specified value of `end`, especially when `step` is not an integer. To avoid this unrequired behavior, the tolerance `tol` can be used to round the resulting numbers in the sequence to a relative precision equal to `tol`. Therefore, the tolerance `tol` should be present only in the case where the first three arguments are reals, all of them having the same precision; moreover, it should be a negative power of ten, *e. g.* 0.001.

When `step` is not present, the value of `end` must be greater than that of `start`. When `step` is present, its sign must be the same as that of `end-start`. `step` cannot have a zero value.

See also: `mfLinSpace`, `mfLogSpace`

Example(s):

```
call msDisplay( mfColon(1,15,step=3), "mfColon(1,15,step=3)" )
```

output:

```
mfColon(1,15,step=3) =

     1     4     7    10    13
```

```
call msDisplay( mfColon(1.0,3.0,step=0.3333,tol=0.001),          &
                "mfColon(1.0,3.0,step=0.3333,tol=0.001)" )
```

output:

```
mfColon(1.0,3.0,step=0.3333,tol=0.001) =

   1.0000    1.3330    1.6670    2.0000    2.3330    2.6670    3.0000
```

· · · / · · ·

```
   call msDisplay( mfColon( 0.1, 1.3, step=0.2 ),                    &
                   "mfColon( 0.1, 1.3, step=0.2 )" )
```

output:

```
 mfColon( 0.1, 1.3, step=0.2 ) =

    0.1000    0.3000    0.5000    0.7000    0.9000    1.1000
```

```
   call msDisplay( mfColon( 0.1, 1.3, step=0.2, tol=0.0001 ),        &
                   "mfColon( 0.1, 1.3, step=0.2, tol=0.0001 )" )
```

output:

```
 mfColon( 0.1, 1.3, step=0.2, tol=0.0001 ) =

    0.1000    0.3000    0.5000    0.7000    0.9000    1.1000    1.3000
```

`.not.` mfArray logical operator

Calling syntax:

```
C = .not. A
```

Description:

Takes the negation (element-wise) of a boolean `mfArray`.

Remarks: a boolean `mfArray` is of type real and contains 1 for emphTRUE and 0 for *FALSE*. It can be displayed as any `mfArray`; it is usually built from relational operators ('`==`', '`/=`', '`>`', *etc.*).

See also: `.and.`, `.or.`, `.eqv.`, `.neqv.`

`.and.` mfArray logical operator

Calling syntax:

```
C = A .and. B
```

Description:

Composes (element-wise) two boolean `mfArray`s.

Remarks: a boolean `mfArray` is of type real and contains 1 for emphTRUE and 0 for *FALSE*. It can be displayed as any `mfArray`; it is usually built from relational operators ('`==`', '`/=`', '`>`', *etc.*).

See also: `.not.`, `.or.`, `.eqv.`, `.neqv.`

`.or.` mfArray logical operator

Calling syntax:

```
C = A .or. B
```

Description:

Composes (element-wise) two boolean `mfArray`s.

Remarks: a boolean `mfArray` is of type real and contains 1 for emphTRUE and 0 for *FALSE*. It can be displayed as any `mfArray`; it is usually built from relational operators ('`==`', '`/=`', '`>`', *etc.*).

See also: `.not.`, `.and.`, `.eqv.`, `.neqv.`

`.eqv.` mfArray logical operator

Calling syntax:

```
C = A .eqv. B
```

Description:

Composes (element-wise) two boolean `mfArray`s.

Remarks: a boolean `mfArray` is of type real and contains 1 for emphTRUE and 0 for *FALSE*. It can be displayed as any `mfArray`; it is usually built from relational operators ('`==`', '`/=`', '`>`', *etc.*).

See also: `.not.`, `.and.`, `.or.`, `.neqv.`

`.neqv.` mfArray logical operator

Calling syntax:

```
C = A .neqv. B
```

Description:

Composes (element-wise) two boolean `mfArray`s.

Remarks: a boolean `mfArray` is of type real and contains 1 for emphTRUE and 0 for *FALSE*. It can be displayed as any `mfArray`; it is usually built from relational operators ('`==`', '`/=`', '`>`', *etc.*).

See also: `.not.`, `.and.`, `.or.`, `.eqv.`

`mfIsMember` — elements members of a set

Calling syntax:

```
E = mfIsMember( A, set [, tol ] )
```

Description:

Returns a boolean `mfArray` which shows which elements of the `mfArray A` are member of the `mfArray set`.

`E` has the same shape as `A`.

The `tol` optional argument is the tolerance used in the comparison test. By default, tolerance is zero, *i. e.* a strict equality is used to discard elements.

Remark: `mfArrays A` and `set` must be real with a dense structure.

See also: `mfIntersect`, `mfUnion`, `mfUnique`

Example(s):

```
a = mf([ 1, 2, 3 ]) .vc. mf([ 4, 5, 6 ])
set = [ 0, 12, 8, 6, 2, 10, 4 ]
call msDisplay( a, "a", set, "set" )
call msDisplay( mfIsMember(a,set), "mfIsMember(a,set)" )
```

output:

```
a =

    1     2     3
    4     5     6

set =

    0    12     8     6     2    10     4

mfIsMember(a,set) =

    F     T     F
    T     F     T
```

`mfIntersect` — common elements of two sets

Calling syntax:

```
E = mfIntersect( A, B [, tol ] )
```

Description:

Returns an `mfArray` containing all elements common to the `mfArrays` `A` and `B`. Duplicated elements are discarded; the result is a sorted row vector.

The `tol` optional argument is the tolerance used in the equality test to discard duplicated elements, if any. By default, tolerance is zero, *i. e.* a strict equality is used to discard elements.

Remark: `mfArrays` `A` and `B` must be both real with a dense structure.

See also: `mfIsMember`, `mfUnion`, `mfUnique`

Example(s):

```
A = reshape( [ 9, 7, 20, 4, 1, 5, 9, 7, 21, 1, 5, 25 ], [ 3, 4 ] )
B = reshape( [ 25, 15, 8, 8, 6, 5, 5, 4, 20, 10 ], [ 5, 2 ] )
call msDisplay( A, "A", B, "B" )
call msDisplay( mfIntersect(A,B), "mfIntersect(A,B)" )
```

output:

```
A =

    9     4     9     1
    7     1     7     5
   20     5    21    25

B =

   25     5
   15     5
    8     4
    8    20
    6    10

mfIntersect(A,B) =

    4     5    20    25
```

`mfUnion` — all elements of two sets

Calling syntax:

```
E = mfUnion( A, B [, tol ] )
```

Description:

Returns an `mfArray` containing all elements of to the `mfArrays` `A` and `B`. Duplicated elements are discarded; the result is a sorted row vector.

The `tol` optional argument is the tolerance used in the equality test to discard duplicated elements, if any. By default, tolerance is zero, *i. e.* a strict equality is used to discard elements.

Remark: `mfArrays` `A` and `B` must be both real with a dense structure.

See also: `mfIsMember`, `mfIntersect`, `mfUnique`

Example(s):

```
A = reshape( [ 9, 7, 20, 4, 1, 5, 9, 7, 21, 1, 5, 25 ], [ 3, 4 ] )
B = reshape( [ 25, 15, 8, 8, 6, 5, 5, 4, 20, 10 ], [ 5, 2 ] )
call msDisplay( A, "A", B, "B" )
call msDisplay( mfIntersect(A,B), "mfIntersect(A,B)" )
```

output:

```
A =

    9     4     9     1
    7     1     7     5
   20     5    21    25

B =

   25     5
   15     5
    8     4
    8    20
    6    10

mfIntersect(A,B) =

    1     4     5     6     7     8     9    10    15    20    21    25
```

`mfUnique` — unique elements of a set

Interface:

```
function mfUnique( A, order, occurrence, tol ) result( out )

   type(mfArray)                                  :: A
   character(len=*),     intent(in), optional :: order, occurrence
   real(kind=MF_DOUBLE), intent(in), optional :: tol
   type(mfArray) :: out
```

Description:

Returns the unique elements of the `mfArray A`. `A` must be a vector (either row or column).

By default, the returned elements are sorted in increasing order. However, their layout may be changed by using the following optional parameters:

- the `order` optional argument (= `"asc"`, `"des"` or `"no"`) may be used to specify the ordering.
- if `order="no"`, no ordering is applied to the resulting elements but it is required to add the other `occurrence` optional argument (value may be `"first"` or `"last"`) to specify if the selected element is the first one or the last one.

The `tol` optional argument is the tolerance used in the equality test to discard duplicated elements, if any. By default, tolerance is zero, *i. e.* a strict equality is used to discard elements.

Remark: the `mfArray A` must be real with a dense structure.

See also: `mfIsMember`, `mfUnion`, `mfIntersect`

Example(s):

```
a = [ 2, 4, 6, 8, 9, 8, 7, 1, 5, 3, 6, 2, 4 ]
call msDisplay( a, "a" )
call msDisplay( mfUnique(a), "mfUnique(a)" )
call msDisplay( mfUnique(a,order="des"), 'mfUnique(a,order="des")' )
call msDisplay( mfUnique(a,order="no",occurrence="first"),           &
                'mfUnique(a,order="no",occurrence="first")' )
call msDisplay( mfUnique(a,order="no",occurrence="last"),            &
                'mfUnique(a,order="no",occurrence="last")' )
```

output:

```
a =

    2    4    6    8    9    8    7    1    5    3    6    2    4

mfUnique(a) =

    1    2    3    4    5    6    7    8    9

mfUnique(a,order="des") =

    9    8    7    6    5    4    3    2    1
```

.../...

```
mfUnique(a,order="no",occurrence="first") =

    2     4     6     8     9     7     1     5     3

mfUnique(a,order="no",occurrence="last") =

    9     8     7     1     5     3     6     2     4
```

1.5 Elementary Math Functions

mfACos	inverse cosine
mfACosh	inverse hyperbolic cosine
mfACot	inverse cotangent
mfACoth	inverse hyperbolic cotangent
mfACsc	inverse cosecant
mfACsch	inverse hyperbolic cosecant
mfASec	inverse secant
mfASech	inverse hyperbolic secant
mfASin	inverse sine
mfASinh	inverse hyperbolic sine
mfATan	inverse tangent
mfATan2	four quadrant inverse tangent
mfATanh	inverse hyperbolic tangent
mfCos	cosine
mfCosh	hyperbolic cosine
mfCot	cotangent
mfCoth	hyperbolic cotangent
mfCsc	cosecant
mfCsch	hyperbolic cosecant
mfSec	secant
mfSech	hyperbolic secant
mfSin	sine
mfSinh	hyperbolic sine
mfTan	tangent
mfTanh	hyperbolic tangent

mfSqrt	square root
mfExp	exponential
mfExpm1	exponential minus one
mfLog	natural logarithm
mfLog1p	natural logarithm of $1 + x$
mfLog10	base 10 logarithm
mf/msLog2	base 2 logarithm
mfPow10	base 10 power
mfPow2	base 2 power
mfFun, mfFun2	general function (one or two arguments)
mfGridFun	general function (two arguments) applied to coords matrices

mfAbs	absolute value
mfAngle	phase angle
mfComplex	complex conversion
mfConj	complex conjugate
mfImag	complex imaginary part
mfReal	complex real part
mfHypot	hypotenuse
mfRound	round towards nearest integer
mfCeil	round towards plus infinity
mfFix	round towards zero
mfFloor	round towards minus infinity
mfMod	modulus after division
mfRem	remainder after division
mfSign	signum function
mfCSign	complex signum function

See also:

Core Routines

File Input/Output

Data Analysis Functions

Operators

Specialized Math Functions

Elementary Matrix Manipulation Functions

Matrix Functions

Polynomial Functions

Optimization and Function Functions

Sparse Matrices

`mfACos` — inverse cosine

Calling syntax:

```
C = mfACos( A )
```

Description:

Returns the inverse cosine (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfCos`, `mfACosh`, `mfCosh`

`mfACosh` — inverse hyperbolic cosine

Calling syntax:

```
C = mfACosh( A )
```

Description:

Returns the inverse hyperbolic cosine (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfCos`, `mfACos`, `mfCosh`

`mfACot` — inverse cotangent

Calling syntax:

```
C = mfACot( A )
```

Description:

Returns the inverse cotangent (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfCot`, `mfACoth`, `mfCoth`

`mfACoth` — inverse hyperbolic cotangent

Calling syntax:

```
C = mfACoth( A )
```

Description:

Returns the inverse hyperbolic cotangent (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfCot`, `mfACot`, `mfCoth`

`mfACsc` — inverse cosecant

Calling syntax:

```
C = mfACsc( A )
```

Description:

Returns the inverse cosecant (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfCsc`, `mfACsch`, `mfCsch`

`mfACsch` — inverse hyperbolic cosecant

Calling syntax:

```
C = mfACsch( A )
```

Description:

Returns the inverse hyperbolic cosecant (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfCsc`, `mfACsc`, `mfCsch`

`mfASec` — inverse secant

Calling syntax:

```
C = mfASec( A )
```

Description:

Returns the inverse secant (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfSec`, `mfASech`, `mfSech`

`mfASech` — inverse hyperbolic secant

Calling syntax:

```
C = mfASech( A )
```

Description:

Returns the inverse hyperbolic secant (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfSec`, `mfASec`, `mfSech`

`mfASin` — inverse sine

Calling syntax:

```
C = mfASin( A )
```

Description:

Returns the inverse sine (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfSin`, `mfASinh`, `mfSinh`

`mfASinh` — inverse hyperbolic sine

Calling syntax:

```
C = mfASinh( A )
```

Description:

Returns the inverse hyperbolic sine (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfSin`, `mfASin`, `mfSinh`

`mfATan` — inverse tangent

Calling syntax:

```
C = mfATan( A )
```

Description:

Returns the inverse tangent (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfATan2`, `mfTan`, `mfATanh`, `mfTanh`

`mfATan2` four quadrant inverse tangent

Calling syntax:

```
C = mfATan2( Y, X )
```

Description:

Returns the four quadrant inverse tangent (element-wise) of the quotient Y/X, and the result is in $(-\pi, \pi]$.

`X` and `Y` must be numerical, dense `mfArrays`.

See also: `mfATan`, `mfTan`, `mfATanh`, `mfTanh`

`mfATanh` — inverse hyperbolic tangent

Calling syntax:

```
C = mfATanh( A )
```

Description:

Returns the inverse hyperbolic tangent (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfTan`, `mfATan`, `mfTanh`

`mfCos` **cosine**

Calling syntax:

```
C = mfCos( A )
```

Description:

Returns the cosine (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfACos`, `mfACosh`, `mfCosh`

`mfCosh` — hyperbolic cosine

Calling syntax:

```
C = mfCosh( A )
```

Description:

Returns the hyperbolic cosine (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfCos`, `mfACos`, `mfACosh`

`mfCot` — cotangent

Calling syntax:

```
C = mfCot( A )
```

Description:

Returns the cotangent (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfACot`, `mfACoth`, `mfCoth`

`mfCoth` — hyperbolic cotangent

Calling syntax:

```
C = mfCoth( A )
```

Description:

Returns the hyperbolic cotangent (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfCot`, `mfACot`, `mfACoth`

`mfCsc` — cosecant

Calling syntax:

```
C = mfCsc( A )
```

Description:

Returns the cosecant (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfACsc`, `mfACsch`, `mfCsch`

`mfCsch` — hyperbolic cosecant

Calling syntax:

```
C = mfCsch( A )
```

Description:

Returns the hyperbolic cosecant (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfCsc`, `mfACsc`, `mfACsch`

`mfSec` — secant

Calling syntax:

```
C = mfSec( A )
```

Description:

Returns the secant (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfASec`, `mfASech`, `mfSech`

`mfSech` — hyperbolic secant

Calling syntax:

```
C = mfSech( A )
```

Description:

Returns the hyperbolic secant (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfASech`, `mfASec`, `mfSec`

`mfSin` — sine

Calling syntax:

```
C = mfSin( A )
```

Description:

Returns the sine (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfASin`, `mfASinh`, `mfSinh`

`mfSinh` — hyperbolic sine

Calling syntax:

```
C = mfSinh( A )
```

Description:

Returns the hyperbolic sine (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfASin`, `mfASinh`, `mfSin`

`mfTan` **tangent**

Calling syntax:

```
C = mfTan( A )
```

Description:

Returns the tangent (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfATan`, `mfATanh`, `mfTanh`

`mfTanh` — hyperbolic tangent

Calling syntax:

```
C = mfTanh( A )
```

Description:

Returns the hyperbolic tangent (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfATan`, `mfATanh`, `mfTan`

`mfSqrt` — square root

Calling syntax:

```
C = mfSqrt( A )
```

Description:

Returns the square root (element-wise) of the `mfArray A` (dense or sparse, real or complex).

The returned `mfArray` may be of type complex if any element has a negative value.

See also: `mfSqrtm`

`mfExp` **exponential**

Calling syntax:

```
C = mfExp( A )
```

Description:

Returns the exponential (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfLog`, `mfLog1p`, `mfExpm1`, `mfExpm`

`mfExpm1` — exponential minus one

Calling syntax:

```
C = mfExpm1( A )
```

Description:

Returns the value of $\exp(x)-1$, for each x element of the `mfArray A`. This operation is made element-wise.

See also: `mfExp`, `mfLog`, `mfLog1p`, `mfExpm`

`mfLog` — natural logarithm

Calling syntax:

```
C = mfLog( A )
```

Description:

Returns the natural logarithm (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfLog1p`, `mfExp`, `mfExpm1`, `mfLogm`

`mfLog1p` — **natural logarithm of** $1 + x$

Calling syntax:

```
C = mfLog1p( A )
```

Description:

Returns the value of $\log(1 + x)$, for each x element of the `mfArray A` (numerical dense only). This operation is made element-wise.

See also: `mfLog`, `mfExp`, `mfExpm1`, `mfExpm`

`mfLog10` — base 10 logarithm

Calling syntax:

```
C = mfLog10( A )
```

Description:

Returns the base 10 logarithm (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfLog`, `mf/msLog2`

`mf/msLog2` — base 2 logarithm

Calling syntax:

```
C = mfLog2( A )
```

Description:

Returns the base 2 logarithm (element-wise) of the `mfArray A` (numerical dense only).

The subroutine version returns two `mfArray`s:

```
call msLog2( mfOut(M,E), A )
```

The mantissa `M` has its element ranged in $[0.5, 1[$, whereas elements of `E` are integer exponents.

Each element of `A` verifies: $a = m2^e$

See also: `mfPow2`, `mfLog`, `mfLog10`, `mfOut`

`mfPow10` — base 10 power

Calling syntax:

```
C = mfPow10( A )
```

Description:

Returns the base 10 power (element-wise) of the `mfArray A` (numerical dense only).

See also: `mfLog10`

`mfPow2` — base 2 power

First usage:

```
C = mfPow2( A )
```

Description:

Returns the base 2 power (element-wise) of the `mfArray A` (numerical dense only).

Second usage:

```
C = mfPow2( M, E )
```

Description:

Returns numbers which write: $m2^e$, for each (real) m of `M` and each (integer) e of `E`. `M` and `E` are two `mfArray`s.

See also: `mfLog2`

`mfFun` general function (one arg.)

Usage:

```
C = mfFun( A, fun )
```

Description:

Apply the user-defined function (element-wise) `fun` to the `mfArray A` (numerical dense only).

`fun` is a real or complex function having the following interface:

```
real(kind=MF_DOUBLE) function fun( x1 )
   real(kind=MF_DOUBLE), intent(in) :: x1
end function
```

or

```
complex(kind=MF_DOUBLE) function fun( x1 )
   complex(kind=MF_DOUBLE), intent(in) :: x1
end function
```

See also: `mfFunm`, `mfFun2`

`mfFun2` — general function (two args)

Usage:

```
C = mfFun2( A, B, fun )
```

Description:

Apply the user-defined function (element-wise) `fun` to the `mfArrays` `A` and `B` (numerical dense only).

`fun` is a real or complex function having the following interface:

```
real(kind=MF_DOUBLE) function fun( x1, x2 )
   real(kind=MF_DOUBLE), intent(in) :: x1, x2
end function
```

or

```
complex(kind=MF_DOUBLE) function fun( x1, x2 )
   complex(kind=MF_DOUBLE), intent(in) :: x1, x2
end function
```

See also: `mfFun`, `mfGridFun`

`mfGridFun` general function (two args) applied to coords matrices

Usage:

```
Z = mfGridFun( v_x, v_y, fun )
```

applies the 2-argument function `fun` to the couple of coordinate matrices (X, Y) generated by the vectors `v_x` and `v_y`, without building X and Y (see `msMeshGrid` about contraints concerning `v_x` and `v_y`). It aims at economize memory for large grids.

`fun` is either a character string giving the name of a 2-arg function (as *complex*, *hypot*, *atan2*, ...), or the name of a user defined function. In this latter case, the user function must have the following interface:

```
real(kind=MF_DOUBLE) function fun( x1, x2 )
   real(kind=MF_DOUBLE), intent(in) :: x1, x2
end function
```

See also: `mfFun2`

`mfAbs` — absolute value

Calling syntax:

```
C = mfAbs( A )
```

Description:

Returns the absolute value (or the module for complex numbers) of the `mfArray A` (dense or sparse). This operation is made element-wise.

See also: `mfAngle`

`mfAngle` — phase angle

Calling syntax:

```
C = mfAngle( A )
```

Description:

Returns the phase angle (element-wise), in radians of the complex `mfArray A` (dense or sparse).

For real values, this function returns 0 or π. Moreover, it returns 0 when argument is null (real or complex), so maintains the sparsity of the `mfArray`.

See also: `mfAbs`

`mfComplex` — complex conversion

Calling syntax:

```
C = mfComplex( A [, B] )
```

Description:

Converts a real `mfArray A` (dense or sparse) in a complex one.

If the optional argument `mfArray B` is present, builds a complex `mfArray` whose real part is `A` and the imaginary part is `B`. In this latter case, `A` and `B` must be both real (dense or sparse).

See also: `mfReal`, `mfImag`

`mfConj` — complex conjugate

Calling syntax:

```
C = mfConj( A )
```

Description:

Returns the conjugate (element-wise) of the complex `mfArray A` (dense or sparse).

`A` may be real, but the returned `mfArray` is complex.

See also: `.h.`

`mfImag` — complex imaginary part

Calling syntax:

```
C = mfImag( A )
```

Description:

Returns the imaginary part (element-wise) of the complex `mfArray A` (dense or sparse).

See also: `mfReal`

`mfReal` — complex real part

Calling syntax:

```
C = mfReal( A )
```

Description:

Returns the real part (element-wise) of the complex `mfArray A` (dense or sparse).

See also: `mfImag`

`mfHypot` **hypotenuse**

Calling syntax:

```
C = mfHypot( A, B )
```

Description:

Returns the hypotenuse of a right rectangle whose sides (`mfArrays` `A` and `B`, in dense format) are provided. This operation is made element-wise.

Remarks: the computation is done without overflow or underflow.

`mfRound` — round towards nearest integer

Calling syntax:

```
C = mfRound( A )
```

Description:

Rounds towards nearest integer each element of the `mfArray A` (dense or sparse). The behavior of this routine is similar to the *anint* Fortran 90 intrinsic function.

See also: `mfCeil`, `mfFix`, `mfFloor`

`mfCeil` — round towards plus infinity

Calling syntax:

```
C = mfCeil( A )
```

Description:

Rounds towards plus infinity each element of the `mfArray A` (dense or sparse).

See also: `mfFix`, `mfFloor`, `mfRound`

`mfFix` — round towards zero

Calling syntax:

```
C = mfFix( A )
```

Description:

Rounds towards zero each element of the `mfArray A` (dense or sparse).

See also: `mfCeil`, `mfFloor`, `mfRound`

`mfFloor` — round towards minus infinity

Calling syntax:

```
C = mfFloor( A )
```

Description:

Rounds towards minus infinity each element of the `mfArray A` (dense or sparse).

See also: `mfCeil`, `mfFix`, `mfRound`

`mfMod` — modulus after division

Calling syntax:

```
C = mfMod( A, B )
```

Description:

Returns the modulus after division of each element of the `mfArray A` (numerical dense only).

N.B.: this is the Fortran 90 `modulo` function

See also: `mfRem`

`mfRem` — remainder after division

Calling syntax:

```
C = mfRem( A, B )
```

Description:

Returns the modulus after division of each element of the `mfArray A` (numerical dense only).

N.B.: this is the Fortran 90 `mod` function

See also: `mfMod`

`mfSign` — signum function

Calling syntax:

```
C = mfSign( A )
```

Description:

Applies the signum function to each element of the `mfArray A` (numerical dense only). It returns only 1 or -1.

N.B.: It differs from the Fortran 90 `sign` intrinsic function.

See also: `mfCSign`

`mfCSign` — complex signum function

Calling syntax:

```
C = mfCSign( A )
```

Description:

Applies the complex signum function to each element of the `mfArray A` (numerical dense only).

See also: `mfSign`

1.6 Specialized Math Functions

`mfErf`	Error function
`mfErfInv`	inverse Error function
`mfErfC`	Complementary Error function
`mfErfCInv`	inverse Complementary Error function
`mfErfCScaled`	Scaled Complementary Error function
`mfExpInt`	Exponential Integral function
`mfGamma`	Gamma function
`mfGammaLn`	Logarithm of the Gamma function
`mfBesselJ`	Bessel functions of the first kind
`mfBesselY`	Bessel functions of the second kind
`mfBesselI`	modified Bessel functions of the first kind
`mfBesselK`	modified Bessel functions of the second kind
`mfAiry`	Airy function
`msEllipKE`	complete elliptic integrals of first and second kinds
`mfIsPrime`	prime number test
`mfFactor`	prime numbers factorization
`msCleanPrimeNumbers`	prime numbers storage cleaning
`msRat`	rational approximation

See also:

Core Routines

File Input/Output

Data Analysis Functions

Operators

Elementary Math Functions

Elementary Matrix Manipulation Functions

Matrix Functions

Polynomial Functions

Optimization and Function Functions

Sparse Matrices

`mfErf` — Error function

Calling syntax:

```
C = mfErf( A )
```

Description:

Returns the Error function (element-wise) of the `mfArray A` (dense only, real or complex).

See also: `mfErfC`, `mfErfInv`, `mfErfCInv`, `mfErfCScaled`

`mfErfInv` — inverse Error function

Calling syntax:

```
C = mfErfInv( A )
```

Description:

Returns the inverse Error function (element-wise) of the `mfArray A` (dense only).

Restriction: `A` must be of type real.

See also: `mfErf`, `mfErfC`, `mfErfCInv`, `mfErfCScaled`

`mfErfC` — Complementary Error function

Calling syntax:

```
C = mfErfC( A )
```

Description:

Returns the Complementary Error function (element-wise) of the `mfArray A` (dense only, real or complex).

See also: `mfErf`, `mfErfInv`, `mfErfCInv`, `mfErfCScaled`

`mfErfCInv` inverse Complementary Error function

Calling syntax:

```
C = mfErfCInv( A )
```

Description:

Returns the inverse Complementary Error function (element-wise) of the `mfArray A` (dense only).

Restriction: `A` must be of type real.

See also: `mfErf`, `mfErfInv`, `mfErfC`, `mfErfCScaled`

`mfErfCScaled` — Scaled Complementary Error function

Calling syntax:

```
C = mfErfCScaled( A )
```

Description:

Returns the Scaled Complementary Error function (element-wise) of the `mfArray A` (dense only), defined by `mfErfCScaled(x) = exp(x**2) * mfErfC(x)`

Restriction: `A` must be of type real.

See also: `mfErf`, `mfErfInv`, `mfErfC`, `mfErfCInv`

`mfExpInt` Exponential Integral function

Calling syntax:

```
C = mfExpInt( A )
```

Description:

Returns the exponential integral function (element-wise) of the `mfArray A` (dense only).

This function is mathematically defined as:

$$E_1(x) = \int_x^\infty \frac{\exp(-t)}{t} dt$$

For positive value of x, it has the following properties:

$$E_1(0) = +\infty, \quad E_1(+\infty) = 0$$

Restriction: `A` must be of type real and must have a positive value. On the contrary, a warning is emitted.

`mfGamma` — Gamma function

Calling syntax:

```
C = mfGamma( A )
```

Description:

Returns the gamma function (element-wise) of the `mfArray A` (dense only).

Restriction: `A` must be of type real.

See also: `mfGammaLn`

`mfGammaLn` — Logarithm of the Gamma function

Calling syntax:

```
C = mfGammaLn( A )
```

Description:

Returns the natural logarithm of the gamma function (element-wise) of the `mfArray A` (dense only).

Restriction: `A` must contain positive real values.

See also: `mfGamma`

`mfBesselJ` Bessel functions of the first kind

Calling syntax:

```
C = mfBesselJ( alpha, A )
```

Description:

Returns the Bessel functions of the first kind $J_\alpha(x)$ (element-wise) of the `mfArray A` (dense only).

The `real alpha` order doesn't need to be an integer: both positive and negative fractional orders are possible.

Restriction: `A` must be of type real but, according to the values of α and x, a complex `mfArray` may be returned.

Remark: Some roots of $J_0(x)$ and $J_1(x)$ are stored respectively in the arrays: `MF_BESSEL_J0_ROOTS` and `MF_BESSEL_J1_ROOTS`.

See also: `mfBesselY`, `mfBesselI`, `mfBesselK`, `MF_BESSEL_J0_ROOTS`, `MF_BESSEL_J1_ROOTS`

`mfBesselY` — Bessel functions of the second kind

Calling syntax:

```
C = mfBesselY( alpha, A )
```

Description:

Returns the Bessel functions of the second kind $Y_\alpha(x)$ (element-wise) of the `mfArray A` (dense only).

The `real alpha` order doesn't need to be an integer: both positive and negative fractional orders are possible.

Restriction: `A` must be of type real but, according to the values of α and x, a complex `mfArray` may be returned.

See also: `mfBesselJ`, `mfBesselI`, `mfBesselK`

`mfBesselI` modified Bessel functions of the first kind

Calling syntax:

```
C = mfBesselI( alpha, A )
```

Description:

Returns the modified Bessel functions of the first kind $I_\alpha(x)$ (element-wise) of the `mfArray A` (dense only).

The `real alpha` order doesn't need to be an integer: both positive and negative fractional orders are possible.

Restriction: `A` must be of type real but, according to the values of α and x, a complex `mfArray` may be returned.

See also: `mfBesselJ`, `mfBesselY`, `mfBesselK`

`mfBesselK` modified Bessel functions of the second kind

Calling syntax:

```
C = mfBesselK( alpha, A )
```

Description:

Returns the modified Bessel functions of the second kind $K_\alpha(x)$ (element-wise) of the `mfArray A` (dense only).

The `real alpha` order doesn't need to be an integer: both positive and negative fractional orders are possible.

Restriction: `A` must be of type real but, according to the values of α and x, a complex `mfArray` may be returned.

See also: `mfBesselJ`, `mfBesselY`, `mfBesselI`

`mfAiry` — Airy function

Calling syntax:

```
B = mfAiry( A )
```

Description:

Returns the Airy function (element-wise) of the `mfArray A` (dense only).

Restriction: `A` must be of type real and a real `mfArray` is returned. Complex function is not yet implemented.

See also: `mfBesselJ`

`msEllipKE` complete elliptic integrals of first and second kinds

Calling syntax:

```
call msEllipKE( mfOut(K,E), A )
```

Description:

Computes the complete elliptic integrals of first and second kind $K(m)$ and $E(m)$ (element-wise) of the `mfArray A` (dense only), which are stored in the two `mfArray K` and `E`.

Each element of the array `A` is taken to be the parameter $m = k^2$, where k is the modulus.

The parameter m must be ranged in $[0, 1]$ else NaN values are returned for both integrals.

See also: `mfOut`

`mfIsPrime` — prime number test

Calling syntax:

```
B = mfIsPrime( A )
```

Description:

Tests if numbers in the `mfArray A` are prime numbers. This function returns a boolean `mfArray` of same shape as `A`. The argument `A` may be scalar, vector or matrix, and is supposed to contain only integer numbers.

Remark: Integers involved in this routine must be less than 2,147,000,000.

See also: `mfFactor`, `msCleanPrimeNumbers`

`mfFactor` — prime numbers factorization

Calling syntax:

```
B = mfFactor( A )
```

Description:

Returns the prime factors of the scalar `mfArray A`. The argument `A` is supposed to contain an integer number.

Remark: Integers involved in this routine must be less than 2,147,000,000.

See also: `mfIsPrime`, `msCleanPrimeNumbers`

Example(s):

```
x = 2**4 * 3**3 * 5**2 * 7
call msDisplay( x, "x = 2**4 * 3**3 * 5**2 * 7" )
print "(/,A)", "*** test number 109 ***"
call msDisplay( mfFactor(x), "mfFactor(x)" )
```

output:

```
x = 2**4 * 3**3 * 5**2 * 7 =

      75600

mfFactor(x) =

    2     2     2     2     3     3     3     5     5     7
```

`msCleanPrimeNumbers` — prime numbers storage cleaning

Calling syntax:

```
call msCleanPrimeNumbers( )
```

Description:

Cleans the temporary storage of prime numbers, required for `mfIsPrime` and `mfFactor` routines. According to the magnitude of the involved integers, up to 450 MB of memory can be freed.

msRat — rational approximation

Calling syntax:

```
call msRat( mfOut(N,D), A [, tol] )
```

Description:

Computes the rational approximation (element-wise) of the `mfArray A` (real dense only), and returns the matrix of numerators in the `mfArray N` and the matrix of denominators in the `mfArray D`.

Tolerance cannot be smaller than 10^{-12} (default value is 10^{-6}).

For example, with the default tolerance, $\pi \approx 355/113$. Any numerical value has a rational approximation, even $Inf = 1/0$ and $NaN = 0/0$.

See also: mfOut

Example(s):

```
x = MF_PI
call msDisplay( x, "x = MF_PI" )
call msRat( mfOut(y,z), x )
call msDisplay( y, "numerator", z, "denominator" )
```

output:

```
x = MF_PI =

   3.1416

numerator =

  355

denominator =

  113
```

```
x = mfHilb(4)
call msDisplay( x, "x = mfHilb(4)" )
call msRat( mfOut(y,z), x )
call msDisplay( y, "numerator", z, "denominator" )
```

output:

```
x = mfHilb(4)

   1.0000    0.5000    0.3333    0.2500
   0.5000    0.3333    0.2500    0.2000
   0.3333    0.2500    0.2000    0.1667
   0.2500    0.2000    0.1667    0.1429
```

.../...

```
numerator =

    1     1     1     1
    1     1     1     1
    1     1     1     1
    1     1     1     1

denominator =

    1     2     3     4
    2     3     4     5
    3     4     5     6
    4     5     6     7
```

1.7 Elementary Matrix Manipulation Functions

mfEye	identity matrix
mfLinSpace	linear spaced vector
mfLogSpace	log spaced vector
mfMagic	magic square
msMeshGrid	arrays for interpolation and 3-D plots
mfZeros	matrix of zeros
mfOnes	matrix of ones
mfRand	uniformly distributed random numbers
msRand	set the random number generator seed
mfRandN	normally distributed random numbers
mfRandPoiss	Poisson distributed random numbers
mfRepMat	replicate and tile an array
mfMerge	merge of two arrays
mfPack, mfUnpack	packing and unpacking arrays
mfCshift, mfEoshift	array shifting
mf/msDiag	diagonal matrices and diagonals of a matrix
mfBlkDiag	block diagonal concatenation
mf/msFind	find indices of nonzero elements
mfNonZeros	non-zero elements
mf/msReshape	shape modification
mfTriL	lower triangular part
mfTriU	upper triangular part
mfFlipLR	matrix flip Left-Right
mfFlipUD	matrix flip Up-Down
mfRot90	matrix rotation
mfHilb	Hilbert matrix
mfInvHilb	inverse Hilbert matrix
mfVander	Vandermonde matrix
mfCompan	Companion matrix
mfHankel	Hankel matrix
mfToeplitz	Toeplitz matrix
mfPerm	permutation vector creation
mfRandPerm	random permutation vector creation
mfCheckPerm	permutation vector checking
isfinite, mfIsFinite	finite test
isinf, mfIsInf	infinity test
isnan, mfIsNaN	Not-a-Number test
mfIsDiag	diagonal pattern test
mfIsTril	lower triangular pattern test
mfIsTriu	upper triangular pattern test

See also:

Core Routines

File Input/Output

Data Analysis Functions

Operators

Elementary Math Functions

Specialized Math Functions

Matrix Functions

`mfEye` — identity matrix

Calling syntax:

```
A = mfEye( n1 [, n2] )
```

Description:

Returns an `mfArray` containing the identity matrix.

If it is called with one integer argument, the identity matrix is square, of shape (`n1`,`n1`).

If the optional argument `n2` is present, the routine builds an identity matrix of shape (`n1`,`n2`).

See also: `mfOnes`, `mfZeros`

`mfLinSpace` — linear spaced vector, given nb of values

First calling syntax:

```
A = mfLinSpace( start, end [, nval] )
```

Description:

Returns an `mfArray` (row vector) which contains a list of reals equally spaced, from `start` to `end`, with the specified number of values `nval`. `start` and `end` can take any arbitrary finite values, even equal.

By default (*i. e.* when the integer `nval` is not present), the list contains nval = 100 reals. If `nval` is present, it must be greater than or equal to 2.

Second calling syntax:

```
A = mfLinSpace( [ start | end ], nval, step )
```

Description:

In this second syntax, only one argument among `start` and `end` can be present. The additional `step` argument specifies the step between two consecutive values; it cannot be equal to zero.

Remarks: `start`, `end` and `step` must be of type `real` (single or double precision).

See also: `mfColon`, `mfLogSpace`

Example(s):

```
call msDisplay( mfLinspace(-MF_PI, MF_PI, 5),                          &
                "mfLinspace(-MF_PI, MF_PI, 5)" )
```

output:

```
 mfLinspace(-MF_PI, MF_PI, 5) =

   -3.1416   -1.5708    0.0000    1.5708    3.1416
```

```
call msDisplay( mfLinSpace(end=1.0d0, nval=5, step=0.1d0),              &
                "mfLinSpace(end=1.0d0, nval=5, step=0.1d0)" )
```

output:

```
 mfLinSpace(end=1.0d0, nval=5, step=0.1d0) =

   0.6000    0.7000    0.8000    0.9000    1.0000
```

`mfLogSpace` — log spaced vector, given nb of values

First calling syntax:

```
A = mfLogSpace( start, end [, nval] )
```

Description:

Returns an `mfArray` (row vector) which contains a list of reals logarithmic spaced, from $10^{\texttt{start}}$ to $10^{\texttt{end}}$, with the specified number of values `nval`.

By default (*i. e.* when the integer `nval` is not present), the list contains nval = 20 reals. If `nval` is present, it must be greater than or equal to 2.

Second calling syntax:

```
A = mfLogSpace( [ start | end ], nval, step )
```

Description:

In this second syntaxe, only one argument among `start` and `end` can be present. The additional `step` argument specifies the step between the logarithm of two consecutive values; it cannot be equal to zero.

Remarks: `start`, `end` and `step` must be of type `real` (single or double precision).

See also: `mfColon`, `mfLinSpace`

Example(s):

```
call msDisplay( mfLogspace(0.0d0, 5.0d0, 6),                          &
                "mfLogspace(0.0d0, 5.0d0, 6)" )
```

output:

```
 mfLogspace(0.0d0, 5.0d0, 6) =

           1           10          100         1000        10000       100000

call msDisplay( mfLogSpace(start=0.0d0, nval=6, step=0.5d0),              &
                "mfLogSpace(start=0.0d0, nval=6, step=0.5d0)" )
```

output:

```
 mfLogSpace(start=0.0d0, nval=6, step=0.5d0) =

   1.0000    3.1623   10.0000   31.6228  100.0000  316.2278
```

mfMagic — magic square

Calling syntax:

```
A = mfMagic( n )
```

Description:

Returns an `mfArray` containing a magic square, from any positive integer `n` ≥ 3.

Example(s):

```
call msDisplay( mfMagic(3), "mfMagic(3)" )
call msDisplay( mfMagic(4), "mfMagic(4)" )
```

output:

```
mfMagic(3) =

    8     3     4
    1     5     9
    6     7     2

mfMagic(4) =

    1    12     8    13
   15     6    10     3
   14     7    11     2
    4     9     5    16
```

`msMeshGrid` — grid coordinates generation

Calling syntax:

```
call msMeshGrid( mfOut(X,Y), v_x, v_y )
```

Description:

Builds two `mfArrays` containing the (x, y) coordinates of the nodes of a rectangular, structured grid, generated from the two real vectors `v_x` and `v_y`.

Among the two input vectors (called generator vectors), one must be a row vector, and the other a column vector. Then the row vector is vertically replicated as many times as the size of the other to form one of the coordinate matrices, and conversely for the other.

The coordinate matrices `X` and `Y` are typically used to visualize a data matrix with an orientation different from that used by default. See more explanation in section *"Changing Orientation/Transposition of Matrix Data"* in the *Muesli User Guide*. They can also be used in 2D interpolation (see `mfInterp2`).

Remark: Note that most of the time, you don't need to compute explicitly the coordinate matrices `(X,Y)` since you are able to pass the generator vectors directly to graphic routines (`msPColor`, `msContour`, etc.), and also to the numerical routine `mfGridFun`.

See also: `mfOut`

Example(s):

```
v_x =     mfLinSpace( -1.0d0, 1.0d0, 5 )
v_y = .t. mfLinSpace( -1.5d0, 1.5d0, 7 )
call msMeshgrid( mfOut(X,Y), v_x, v_y )
call msDisplay( X, "X", Y, "Y" )
```

output:

```
X =

  -1.0000   -0.5000    0.0000    0.5000    1.0000
  -1.0000   -0.5000    0.0000    0.5000    1.0000
  -1.0000   -0.5000    0.0000    0.5000    1.0000
  -1.0000   -0.5000    0.0000    0.5000    1.0000
  -1.0000   -0.5000    0.0000    0.5000    1.0000
  -1.0000   -0.5000    0.0000    0.5000    1.0000
  -1.0000   -0.5000    0.0000    0.5000    1.0000

Y =

  -1.5000   -1.5000   -1.5000   -1.5000   -1.5000
  -1.0000   -1.0000   -1.0000   -1.0000   -1.0000
  -0.5000   -0.5000   -0.5000   -0.5000   -0.5000
   0.0000    0.0000    0.0000    0.0000    0.0000
   0.5000    0.5000    0.5000    0.5000    0.5000
   1.0000    1.0000    1.0000    1.0000    1.0000
   1.5000    1.5000    1.5000    1.5000    1.5000
```

`mfZeros` — matrix of zeros

Calling syntax:

```
A = mfZeros( n1 [, n2] )
```

Description:

Returns an `mfArray` containing zeros.

If it is called with one integer argument, the output matrix is square, of shape (`n1`,`n1`).

If the optional argument `n2` is present, the routine builds a matrix of shape (`n1`,`n2`).

See also: `mfEye`, `mfOnes`

`mfOnes` — matrix of ones

Calling syntax:

```
A = mfOnes( n1 [, n2] )
```

Description:

Returns an `mfArray` containing ones.

If it is called with one integer argument, the output matrix is square, of shape (`n1`,`n1`).

If the optional argument `n2` is present, the routine builds a matrix of shape (`n1`,`n2`).

See also: `mfEye`, `mfZeros`

`mfRand` — uniformly distributed random numbers

Calling syntax:

```
A = mfRand( n1 [, n2] )
```

Description:

Returns an `mfArray` containing pseudo-random real values, with a uniform distribution in $[0, 1]$.

If it is called with one integer argument, the output matrix is square, of shape (`n1`,`n1`).

If the optional argument `n2` is present, the routine builds a matrix of shape (`n1`,`n2`).

Other syntax:

```
A = mfRand( string )
```

When called with the string `"seed"`, the seed (six long integers stored as double reals) of the random number generator is returned. This value can be later used to reset the seed to a previous state with the `msRand` routine.

Remark: Generated numbers doesn't depend on the compiler used, because the *random_number* Fortran 90 intrinsic routine is not use. The RngStreams package (included in the MUESLI library) is used instead.

See also: `mfRandN`, `mfRandPoiss`

`msRand` — set the random number generator seed

Calling syntax:

```
call msRand( [ seed ] )
```

Description:

Set the random number generator seed either from given values, or from the clock.

Without argument, the seed is set from the clock.

If the optional argument `seed` is present, then this argument must be an `mfArray` whose values are used to reset the seed of the RngStreams package, included in the MUESLI library. This seed must be an array of six integer numbers (stored as double precision reals) satifying the following constraints:

- the first three values must be less than 4294967087 (and not all zero);
- the last three values must be less than 4294944443 (and not all zero).

Note that a check is done in the current routine.

See also: `mfRand`, `mfRandN`, `mfRandPoiss`

`mfRandN` — normally distributed random numbers

Calling syntax:

```
A = mfRandN( n1 [, n2] )
```

Description:

Returns an `mfArray` containing pseudo-random real values, with a normal distribution (with zero mean and unit variance).

If it is called with one integer argument, the output matrix is square, of shape (`n1`,`n1`).

If the optional argument `n2` is present, the routine builds a matrix of shape (`n1`,`n2`).

See also: `mfRand`, `mfRandPoiss`, `msRand`

`mfRandPoiss` — Poisson distributed random numbers

Calling syntax:

```
A = mfRandPoiss( mu, n1 [, n2] )
```

Description:

Returns an `mfArray` containing pseudo-random integer values, with a Poisson distribution, with mean (and variance) equal to `mu`.

The argument `mu` is a real number.

If it is called with only one size, the output matrix is square, of shape (`n1`,`n1`).

If the optional argument `n2` is present, the routine builds a matrix of shape (`n1`,`n2`).

See also: `mfRandN`, `mfRand`, `msRand`

`mfRepMat` — replicate and tile an array

Calling syntax:

```
C = mfRepMat( A, m, n )
```

Description:

This routine replicates the tile `mfArray A`, to build a bigger `mfArray`. The returned `mfArray` can be viewed as a `m`-by-`n` tiling matrix.

The behavior of this routine is similar to the *spread* Fortran 90 intrinsic function.

Example(s):

```
x = reshape( [ (i, i = 1, 6) ], [ 2, 3 ] )
call msDisplay( x, "x", mfRepmat(x,3,2), "mfRepmat(x,3,2)" )
```

output:

```
x =

    1     3     5
    2     4     6

mfRepmat(x,3,2) =

    1     3     5     1     3     5
    2     4     6     2     4     6
    1     3     5     1     3     5
    2     4     6     2     4     6
    1     3     5     1     3     5
    2     4     6     2     4     6
```

`mfMerge` — merge of two arrays

Calling syntax:

```
C = mfMerge( A, B, mask )
```

Description:

The three arguments are `mfArrays` having the same shape. This function merge the two numeric arrays `A` and `B` under the control of the boolean `mask`. Its behavior is like that of the *merge* Fortran 90 intrinsic function.

Sparse arrays are not handled.

Example(s):

```
A =  3.0d0*mfOnes(3,2)
B = -1.0d0*mfOnes(3,2)
mask = mfEye(3,2) > 0.5d0
call msDisplay( A,"A", B,"B", mask,"mask" )
call msDisplay( mfMerge(A,B,mask), "mfMerge(A,B,mask)" )
```

output:

```
A =

     3     3
     3     3
     3     3

B =

    -1    -1
    -1    -1
    -1    -1

mask =

     T     F
     F     T
     F     F

mfMerge(A,B,mask) =

     3    -1
    -1     3
    -1    -1
```

`mfPack` — packing an array

Calling syntax:

```
C = mfPack( A, mask )
```

Description:

Packs the elements of the numeric `mfArray A` in a long column vector, under the control of the boolean `mask`. The behavior of this routine is similar to the *pack* Fortran 90 intrinsic function.

Sparse arrays are not handled.

See also: mfUnpack

Example(s):

```
A = mfMagic(3)
mask = A >= 5.0d0
call msDisplay( A,"A", mask,"mask" )
Ap = mfPack( A, mask )
call msDisplay( Ap, "A packed" )
```

output:

```
A =

    8     3     4
    1     5     9
    6     7     2

mask =

    T     F     F
    F     T     T
    T     T     F

A packed =

    8     6     5     7     9
```

```
R = mfOnes(3)*MF_NAN
call msDisplay( mfUnpack(Ap,mask,field=R), "A unpacked" )
```

output:

```
A unpacked =

   8.0000       NaN       NaN
     NaN     5.0000    9.0000
   6.0000    7.0000       NaN
```

`mfUnpack` — unpacking an array

Calling syntax:

```
C = mfUnpack( A, mask, field )
```

Description:

Unpacks the elements of the numeric vector `mfArray A` under the control of the boolean `mask`. The returned `mfArray` is completed from values of `field`. `A` should have been previously processed by the `mfPack` routine.

The behavior of this routine is similar to the *unpack* Fortran 90 intrinsic function.

Sparse arrays are not handled.

See also: mfPack

Example(s):

```
A = mfMagic(3)
mask = A >= 5.0d0
call msDisplay( A,"A", mask,"mask" )
Ap = mfPack( A, mask )
call msDisplay( Ap, "A packed" )
```

output:

```
A =

    8     3     4
    1     5     9
    6     7     2

mask =

    T     F     F
    F     T     T
    T     T     F

A packed =

    8     6     5     7     9
```

```
R = mfOnes(3)*MF_NAN
call msDisplay( mfUnpack(Ap,mask,field=R), "A unpacked" )
```

output:

```
A unpacked =

  8.0000       NaN       NaN
     NaN    5.0000    9.0000
  6.0000    7.0000       NaN
```

`mfCshift` — array shifting

Calling syntax:

```
C = mfCshift( A, shift )
```

Description:

Performs a circular shift on the vector `mfArray A`. `shift` is an integer. The behavior of this routine is similar to the *cshift* Fortran 90 intrinsic function.

Sparse arrays are not handled.

See also: `mfEoshift`

Example(s):

```
A = [ 1, 2, 3, 4, 5, 6 ]
call msDisplay( A, "A" )
call msDisplay( mfCshift(A,shift=1), "mfCshift(A,shift=1)" )
```

output:

```
A =

    1     2     3     4     5     6

mfCshift(A,shift=1) =

    2     3     4     5     6     1
```

`mfEoshift` — array shifting

Calling syntax:

```
C = mfEoshift( A, shift )
```

Description:

Performs an end-off shift on the vector `mfArray A`. `shift` is an integer. The behavior of this routine is similar to the *eoshift* Fortran 90 intrinsic function.

Sparse arrays are not handled.

See also: `mfCshift`

Example(s):

```
A = [ 1, 2, 3, 4, 5, 6 ]
call msDisplay( A, "A" )
b = 99
call msDisplay( mfEoshift(A,shift=1,boundary=b),                    &
                "mfEoshift(A,shift=1,boundary=b)" )
```

output:

```
A =

    1     2     3     4     5     6

mfEoshift(A,shift=1,boundary=b) =

    2     3     4     5     6    99
```

mf/msDiag — diagonal matrices and diagonals of a matrix

The first form is:

```
C = mfDiag( A [, d] )
```

If `A` is a matrix-like `mfArray` (dense or sparse), this function extracts the main (or the `d`-) diagonal (the output is dense, whatever the storage type of `A` is).

If `A` is a vector-like `mfArray` (dense only), this function builds a dense square matrix which has `A` as main (or as `d`-) diagonal. For building a sparse matrix, use `mfSpDiags`.

The subroutine form:

```
call msDiag( A, v [, d] )
```

modifies the main (or the `d`-) diagonal of the `mfArray A` (dense or sparse) by copying the elements of the vector-like `mfArray v`. The data type of `A` may be changed because `A` and `v` can have real or complex values. The vector `v` is not modified.

Remarks: When `d` is equal to zero, it points to the main diagonal; when positive, the upper part of the matrix is concerned. If the arg. `d` is not present, the main diagonal is the default.

See also: mfBlkDiag, mfEye, mfSpDiags

Example(s):

```
v = [ 1, 2, 3 ]
C = mfDiag(v,2)
call msDisplay( C, "C = mfDiag(v,2)" )
```

output:

```
 v =

    1     2     3

 C = mfDiag(v,2) =

    0     0     1     0     0
    0     0     0     2     0
    0     0     0     0     3
    0     0     0     0     0
    0     0     0     0     0
```

```
v = [ 4, 5, 6 ]
call msDisplay( v, "v" )
call msDiag( C, v, d=-2 )
call msDisplay( C, "C" )
```

output:

```
 v =

    4     5     6

 C =

    0     0     1     0     0
    0     0     0     2     0
    4     0     0     0     3
    0     5     0     0     0
    0     0     6     0     0
```

`mfBlkDiag` — block diagonal concatenation

The first calling syntax:

```
C = mfBlkDiag( A, B )
```

gives the concatenation of the two (square) blocks diagonal `A` and `B`.

The other calling syntax:

```
C = mfBlkDiag( A, n )
```

concatenates n times (`n` being non negative) the square matrix `A` on the diagonal.

Remarks: `A` (and `B`, for the first form), may be dense or sparse, and of any type accepted by the `.vc.` or `.hc.` operators.

See also: mf/msDiag

Example(s):

```
A = mfRot90(mfEye(3,3))
call msDisplay(A,"A")
call msDisplay( mfBlkDiag(A,3), "blkdiag(A,3)" )
```

output:

```
A =

    0     0     1
    0     1     0
    1     0     0

blkdiag(A,3) =

    0     0     1     0     0     0     0     0     0
    0     1     0     0     0     0     0     0     0
    1     0     0     0     0     0     0     0     0
    0     0     0     0     0     1     0     0     0
    0     0     0     0     1     0     0     0     0
    0     0     0     1     0     0     0     0     0
    0     0     0     0     0     0     0     0     1
    0     0     0     0     0     0     0     1     0
    0     0     0     0     0     0     1     0     0
```

`mf/msFind` find indices of nonzero elements

The first form:

```
C = mfFind( A )
```

finds *non zero* (resp. *TRUE*) elements of the numerical (resp. boolean) `mfArray A` and returns a long column of their indices (called also linear indices).

Usually, this routine is applied to a boolean `mfArray`, automatically created in comparison operators; for example, `mfFind( A > 1.0d0 )` returns indices of elements which verify the specified property.

The subroutine form:

```
call msFind( mfOut(i,j[,v]), A )
```

allows the user to retrieve the matrix-indices. The two vectors returned (`i` and `j`) contain the row-index and column-index for non-zero elements.

If the optional argument `v` is present, it contains the values of corresponding non-zero elements.

Warning: Contrary to MATLAB, `mf/msFind` always return row index vectors, instead of column vectors.

See also: `mfNonZeros`, `mfOut`

Example(s):

```
A = mf( [ 9,  0, 0 ] ) .vc.                                    &
    mf( [ 0,  0, 7 ] ) .vc.                                    &
    mf( [ 0, 11, 0 ] )
call msDisplay( A, "A" )
call msFind( mfOut(i,j,v), x )
call msDisplay( i,"row indices", j,"col indices", v,"values" )
```

output:

```
 A =

    9     0     0
    0     0     7
    0    11     0

 row indices =

    1     3     2

 col indices =

    1     2     3

 values =

    9    11     7
```

`mfNonZeros` — non-zero elements

Calling syntax:

```
C = mfNonZeros( A )
```

Description:

Returns all non-zero elements of the `mfArray A` in a vector.

Works as `msFind`, except that indices (i,j) are not returned.

See also: `ms/mfFind`

mf/msReshape — shape modification

Generic interface:

```
function mfReshape( A, m, n ) result( out )

   integer, intent(in) :: A(:)
or real(kind=MF_DOUBLE), intent(in) :: A(:)
or type(mfArray) :: A

   integer, intent(in) :: m, n

   type(mfArray) :: out
```

Description:

This function extends the Fortran intrinsic *reshape*; it can be applied to a dense or sparse `mfArray`.

Remark: in the special call: `mfReshape( A, MF_EMPTY, n )` with `A` being an `mfArray` only, the first dimension, if not specified, is deduced from the total number of elements in `A` (the second dimension `n` can also be replaced by `MF_EMPTY`). Same behavior using `MF_NO_ARG` instead of `MF_EMPTY`.

Another possible form is:

```
call msReshape( A, m, n )
```

which does the same job with the `mfArray A`, except that the shape modification is made "in place". Here, `m` or `n` cannot be replaced by `MF_EMPTY` or `MF_NO_ARG`. This second routine is more efficient than the first one, especially for sparse matrices.

See also: mfRepMat

Example(s):

```
x = [ (i, i = 1, 12) ]
call msDisplay( x, "x" )
call msDisplay( mfReshape(x, 4, 3 ), "mfReshape(x, 4, 3 )" )
call msDisplay( mfReshape(x, 2, MF_EMPTY ), "mfReshape(x, 2, MF_EMPTY )" )
```

output:

```
 x =

    1     2     3     4     5     6     7     8     9    10    11    12

 mfReshape(x, 4, 3 ) =

    1     5     9
    2     6    10
    3     7    11
    4     8    12

 mfReshape(x, 2, MF_EMPTY ) =

    1     3     5     7     9    11
    2     4     6     8    10    12
```

`mfTriL` — lower triangular part

Calling syntax:

```
C = mfTriL( A [, k] )
```

Description:

This function extracts the lower triangular part of an `mfArray`, including the main diagonal.

If `k` is present, it returns the lower part of `A` up to the k-th diagonal (included). `k` must be integer; it can be negative.

See also: `mfTriU`

`mfTriU` upper triangular part

Calling syntax:

```
C = mfTriL( A [, k] )
```

Description:

This function extracts the upper triangular part of an `mfArray`, including the main diagonal.

If `k` is present, it returns the upper part of `A` down to the k-th diagonal (included). `k` must be integer; it can be negative.

See also: `mfTriL`

`mfFlipLR` — matrix flip Left-Right

Calling syntax:

```
C = mfFlipLR( A )
```

Description:

This function flips the `mfArray A` in the left-right direction.

See also: `mfFlipUD`, `mfRot90`

Example(s):

```
A = reshape( [ (i,i=1,15) ], [3,5] )
call msDisplay( A, "A" )
call msDisplay( mfFlipLR(A), "mfFlipLR(A)" )
```

output:

```
A =

     1     4     7    10    13
     2     5     8    11    14
     3     6     9    12    15

mfFlipLR(A) =

    13    10     7     4     1
    14    11     8     5     2
    15    12     9     6     3
```

`mfFlipUD` — matrix flip Up-Down

Calling syntax:

```
C = mfFlipUD( A )
```

Description:

This function flips the `mfArray A` in the up-down direction.

See also: `mfFlipLR`, `mfRot90`

Example(s):

```
A = reshape( [ (i,i=1,15) ], [3,5] )
call msDisplay( A, "A" )
call msDisplay( mfFlipUD(A), "mfFlipUD(A)" )
```

output:

```
A =

    1     4     7    10    13
    2     5     8    11    14
    3     6     9    12    15

mfFlipUD(A) =

    3     6     9    12    15
    2     5     8    11    14
    1     4     7    10    13
```

mfRot90 — matrix rotation

Calling syntax:

```
C = mfRot90( A [, k] )
```

Description:

This function rotates the **mfArray A** in the trigonometric direction, by the angle k*90 in degrees. (*i. e.* for 90°, last column becomes first row).

k may be any integer (including zero and negative value). It is an optional argument (default value is 1).

See also: mfFlipLR, mfFlipUD

Example(s):

```
A = reshape( [ (i,i=1,6) ], [3,2] )
call msDisplay( A, "A" )
call msDisplay( mfRot90(A,1),  "mfRot90(A,1)  [90° anticlockwise]" )
call msDisplay( mfRot90(A,-1), "mfRot90(A,-1) [90° clockwise]" )
```

output:

```
A =

    1     4
    2     5
    3     6

mfRot90(A,1)  [90° anticlockwise] =

    4     5     6
    1     2     3

mfRot90(A,-1) [90° clockwise] =

    3     2     1
    6     5     4
```

`mfHilb` — Hilbert matrix

Calling syntax:

```
C = mfHilb( n )
```

Description:

Builds a square `mfArray` which contains an Hilbert matrix. It is a well known example of badly conditioned matrix.

See also: `mfInvHilb`, `mfCond`

Example(s):

```
A = mfHilb(5)
call msDisplay( A, "A = mfHilb(5)" )
call msDisplay( mfCond(A), "cond(A)" )
```

output:

```
A = mfHilb(5) =

   1.0000    0.5000    0.3333    0.2500    0.2000
   0.5000    0.3333    0.2500    0.2000    0.1667
   0.3333    0.2500    0.2000    0.1667    0.1429
   0.2500    0.2000    0.1667    0.1429    0.1250
   0.2000    0.1667    0.1429    0.1250    0.1111

cond(A) =

  4.7661E+05
```

`mfInvHilb` — inverse Hilbert matrix

Calling syntax:

```
C = mfInvHilb( n )
```

Description:

Returns the inverse of an Hilbert matrix; therefore the matrix product of `mfHilb(n)` by `mfInvHilb(n)` is the identity matrix.

Remarks: this inverse is not computed by some linear algebra algorithm but by an exact formula.

See also: mfHilb

Example(s):

```
A = mfHilb(5)
B = mfInvHilb(5)
call msDisplay( B, "B = mfInvHilb(5)" )
call msDisplay( mfNorm(mfEye(5) - mfMul(A,B)), "| I - A*B |" )
```

output:

```
 B = mfInvHilb(5) =

          25        -300        1050       -1400         630
        -300        4800      -18900       26880      -12600
        1050      -18900       79380     -117600       56700
       -1400       26880     -117600      179200      -88200
         630      -12600       56700      -88200       44100

 | I - A*B | =

   5.7926E-12
```

`mfVander` — Vandermonde matrix

Calling syntax:

```
A = mfVander( v [, n ] )
```

Description:

Returns a Vandermonde matrix. The input vector `mfArray v` may be either a row or a column vector.

If the optional argument `n` is present, the returned matrix will have n columns, each one constituted by a power of the vector `v`; else the returned matrix will be square, *i. e.* will have as much columns as the size of `v`.

Example(s):

```
x = [ 1, 2, 3, 4, 5 ]
call msDisplay( x, "x" )
call msDisplay( mfVander(x,3), "A = mfVander(x,3)" )
```

output:

```
 x =

     1     2     3     4     5

 A = mfVander(x,3) =

     1     1     1
     4     2     1
     9     3     1
    16     4     1
    25     5     1
```

`mfCompan` — Companion matrix

Calling syntax:

```
A = mfCompan( v )
```

Description:

Returns the Companion matrix of a polynomial, represented by its coefficients provided in the input vector `mfArray v` (this latter array may be either a row or a column vector).

The eigenvalues of the matrix `A` are the roots of the input polynomial.

See also: `mfRoots`, `mfEig`

Example(s):

```
! polynom: (x-1)*(x-2)*(x-3)*(x-4) = x^4 - 10*x^3 + 35*x^2 - 50*x + 24,
! so the roots are: {1, 2, 3, 4}
v = [ 1.0d0, -10.0d0, 35.0d0, -50.0d0, 24.0d0 ]
call msDisplay( v, "v" )
A = mfCompan(v)
call msDisplay( A, "A = mfCompan(v)" )
call msDisplay( mfEig(A), "Eigenvalues of A" )
```

output:

```
 v =

     1   -10    35   -50    24

 A = mfCompan(v) =

    10   -35    50   -24
     1     0     0     0
     0     1     0     0
     0     0     1     0

 Eigenvalues of A =

   4.0000 + 0.0000i
   3.0000 + 0.0000i
   2.0000 + 0.0000i
   1.0000 + 0.0000i
```

`mfHankel` **mfHankel matrix**

Calling syntax:

```
A = mfHankel( C [, R ] )
```

Description:

Returns the Hankel matrix of the input vectors `mfArray C` and `R` (these latter arrays may be either a row or a column vector).

A Hankel matrix has constant anti-diagonal values.

Hankel matrices are real (because only real input vectors are supported) and symmetric (if square).

See also: `mfToeplitz`

Example(s):

```
C = [ 1.0d0, 2.0d0, 3.0d0, 4.0d0 ]
R = [ 4.0d0, 5.0d0, 6.0d0 ]
call msDisplay( C, "C", R, "R" )
call msDisplay( mfHankel(C,R), "mfHankel(C,R)" )
```

output:

```
C =

    1     2     3     4

R =

    4     5     6

mfHankel(C,R) =

    1     2     3
    2     3     4
    3     4     5
    4     5     6
```

`mfToeplitz` — Toeplitz matrix

Calling syntax:

```
A = mfToeplitz( C [, R ] )
```

Description:

Returns the Toeplitz matrix of the input vectors `mfArray C` and `R` (these latter arrays may be either a row or a column vector).

A Toeplitz matrix has constant diagonal values and is symmetric (if square).

Complex input vectors are supported. In such a case, the complex matrix is hermitian if the first element of `R` is real.

See also: `mfHankel`

Example(s):

```
C = [ 1.0d0, 2.0d0, 3.0d0, 4.0d0 ]
R = [ 1.0d0, 5.0d0, 6.0d0 ]
call msDisplay( mfToeplitz(C,R), "mfToeplitz(C,R)" )
```

output:

```
C =

    1     2     3     4

R =

    1     5     6

A = mfToeplitz(C,R) =

    1     5     6
    2     1     5
    3     2     1
    4     3     2
```

`mfPerm` — permutation vector creation

Calling syntax:

```
p = mfPerm( v )
```

Description:

Returns a permutation vector mfArray from the vector of indices `v`.

`v` may be either a Fortran 90 integer vector, or a vector-like `mfArray`. In this latter case, the `mfArray` should contains integer values; on the contrary, a warning is emitted by the routine.

See also: `mfIsPerm`, `mfRandPerm`, `mfCheckPerm`, `mfColPerm`, `mfRowPerm`, `msSaveAscii`

`mfRandPerm` random permutation vector creation

Calling syntax:

```
p = mfRandPerm( n [, k ] )
```

Description:

Returns a random permutation vector mfArray. Elements of `p` are unique integers in the range $[1, n]$.

If the second argument `k` is present, returns k unique integers in the range $[1, n]$. In this latter case, `p` is not a true permutation vector if $k \neq n$.

See also: `mfPerm`, `mfIsPerm`, `mfCheckPerm`

`mfCheckPerm` — permutation vector checking

Interface:

```
function mfCheckPerm( A ) result( bool )

   type(mfArray), intent(in) :: A
   logical :: bool
```

Description:

Returns '`.true.`' if `A` is a valid permutation vector, *i. e.* if the integers are consecutive numbers, not necessarily ordered and appearing only once in the list.

See also: `mfPerm`, `mfIsPerm`

`isfinite` — finite test

Interface:

```
function isfinite( x ) result( out )

   double precision, intent(in) :: x

   logical :: out
```

Description:

This function returns a logical value, according to the value of `x`, which must be only of type `double precision`.

Remarks: in the IEEE-754 standard, any real value is either *finite*, *infinite*, or *NaN*.

See also: `isinf`, `isnan`, `mfIsFinite`

`isinf` — infinity test

Interface:

```
function isinf( x ) result( out )

   double precision, intent(in) :: x

   logical :: out
```

Description:

This function returns a logical value, according to the value of `x`, which must be only of type `double precision`.

Remarks: in the IEEE-754 standard, any real value is either *finite*, *infinite*, or *NaN*.

See also: `isfinite`, `isnan`, `mfIsInf`

`isnan` — Not-a-Number test

Interface:

```
function isnan( x ) result( out )

   double precision, intent(in) :: x

   logical :: out
```

Description:

This function returns a logical value, according to the value of `x`, which must be only of type `double precision`.

Remarks: in the IEEE-754 standard, any real value is either *finite*, *infinite*, or *NaN*.

See also: `isfinite`, `isinf`, `mfIsNaN`

`mfIsFinite` — finite test

Calling syntax:

```
C = mfIsFinite( A )
```

Description:

This elemental function returns a boolean `mfArray`.

Remarks: in the IEEE-754 standard, any real value is either *finite*, *infinite*, or *NaN*.

See also: `isfinite`, `mfIsInf`, `mfIsNaN`

`mfIsInf` — infinity test

Calling syntax:

```
C = mfIsInf( A )
```

Description:

This elemental function returns a boolean `mfArray`.

Remarks: in the IEEE-754 standard, any real value is either *finite*, *infinite*, or *NaN*.

See also: `isinf`, `mfIsFinite`, `mfIsNaN`

`mfIsNaN` — Not-a-Number test

Calling syntax:

```
C = mfIsNaN( A )
```

Description:

This elemental function returns a boolean `mfArray`.

Remarks: in the IEEE-754 standard, any real value is either *finite*, *infinite*, or *NaN*.

See also: `isnan`, `mfIsFinite`, `mfIsInf`

`mfIsDiag` — diagonal pattern test

Calling syntax:

```
bool = mfIsDiag( A )
```

returns a *TRUE* boolean value if the `mfArray A` (sparse or dense) is a diagonal matrix.

See also: `mfIsTril`, `mfIsTriu`

`mfIsTril` — lower triangular pattern test

Calling syntax:

```
bool = mfIsTril( A )
```

returns a *TRUE* boolean value if the `mfArray A` (sparse or dense) is a lower triangular matrix.

See also: `mfIsDiag`, `mfIsTriu`

`mfIsTriu` upper triangular pattern test

Calling syntax:

```
bool = mfIsTriu( A )
```

returns a *TRUE* boolean value if the `mfArray A` (sparse or dense) is a upper triangular matrix.

See also: `mfIsDiag`, `mfIsTril`

1.8 Matrix Functions

mfDet	determinant
mfTrace	sum of diagonal elements
mfNorm	matrix or vector norm
mfRank	matrix rank
mfCond	matrix condition number
mfRCond	reciprocal condition number estimation
msLU	$L\,U$ factorization
msLDLT	$L\,D\,L^t$ factorization
mf/msChol	Cholesky factorization
msCholSpSymb	Symbolic Cholesky factorization (sparse only)
msCholSpNum	Numerical Cholesky factorization (sparse only)
mf/msBalance	scaling to improve eigenvalue accuracy
mf/msSVD	singular value decomposition
mfInv, .i.	matrix inverse
mfPseudoInv	matrix pseudo-inverse
mfQR, msQR	orthogonal-triangular factorization
mfQleft, mfQright	left or right multiplication by a Q factor (sparse only)
mf/msEig	eigenvalues and eigenvectors
mf/msHess	Hessenberg form
mf/msSchur	Schur decomposition
mfLDiv, .ix.	left matrix divide
mfRDiv, .xi.	right matrix divide
msRref	reduced row echelon form
mfNull	null space
mfOrth	matrix orthogonalization
mfExpm	matrix exponential
mfLogm	matrix logarithm
mfSqrtm	matrix square root
mfPowm	matrix power
mf/msFunm	general matrix function
mf/msEigs	few eigenvalues and eigenvectors
mf/msSVDS	few singular values and vectors
mfNormEst, mfNormEst1	norm estimations
mfCondEst	matrix condition number estimation
mfIsSymm	matrix symmetry inquiry
mfTolForSymm	tolerance for symmetry
mfIsPosDef	definite positiveness inquiry
mfIsDiagDomCol	diagonally dominant by cols inquiry
mfIsStrictDiagDomCol	strictly diagonally dominant by cols inquiry
mfIsFullRank	full rank inquiry
MF_LAPACK_VERSION	LAPACK version
msGetBlasLib	Get BLAS library type
msGetLapackLib	Get LAPACK library type
msGetArpackInfo	Get ARPACK library information
msGetSuiteSparseLib	Get SuiteSparse library version

See also:

Core Routines

File Input/Output

Data Analysis Functions

`mfDet` — determinant

Calling syntax:

```
C = mfDet( A )
```

Description:

Returns the determinant of the `mfArray A`.

`A` must be square, and cannot be sparse.

Remarks: using the determinant of a matrix to detect whether it is singular is not a good idea: use `mfCond` or `mfRCond` instead.

See also: `mfRank`

`mfTrace` — sum of diagonal elements

Calling syntax:

```
C = mfTrace( A )
```

Description:

Returns the sum of all diagonal element of the `mfArray A` (dense or sparse, real or complex).

Remarks: the trace of a square matrix is also the sum of its eigenvalues.

See also: `mfEig`

`mfNorm` — matrix or vector norm

Calling syntax:

```
C = mfNorm( A [, norm | p ] )
```

Description:

Returns the norm of the `mfArray A`, which can be a vector or a matrix (sparse or dense).

The two optional arguments (`norm` is a character string whereas `p` is an integer) cannot be both present.

- For a matrix, only `1`-, `2`-, `"inf"`- or `"fro"`-norms are supported.

  By default, returns the 2-norm (which is also the largest singular value of `A`). If `p` is present, it returns the `p`-norm of `A` (therefore, `p` can take only the values 1 or 2). If `norm` is present, it must be `inf` (infinite-norm) or `fro` (frobenius-norm).

- For a vector, only `p`- > 0 and `"inf"`- norms are supported.

  By default, returns the 2-norm. If `p` is present, it returns the `p`-norm of `A`. If `norm` is present, it must be `inf` (infinite-norm).

Note: For sparse matrices, use `mfNormEst` or `mfNormEst1` instead, which are more efficient.

See also: `mfCond`, `mfSVD`

`mfRank` — matrix rank

Calling syntax:

```
C = mfRank( A [, tol ] )
```

Description:

Computes the rank of a matrix, *i. e.* the number of linearly independent rows or columns.

If `tol`, of type `real`, is present then it returns the number of singular values greater or equal to this optional tolerance. Default tolerance is chosen as $\max(m, n)\, ||A||_2\, \epsilon$, where m and n are respectively the number of rows and number of columns of the `mfArray A`.

Remarks:

- it is internally based on the Singular Value Decomposition;
- for sparse matrices, the user is invited to use `mfSVDS`, and examine himself few singular values.

See also: `mfSVD`, `mfSVDS`, `mfIsFullRank`

`mfCond` — matrix condition number

Calling syntax:

```
C = mfCond( A )
```

Description:

Returns the condition number of the `mfArray A`. It is the ratio of the largest singular value to the smallest one.

Remarks: The condition number is always greater than 1. A value equal to infinity (resp. close to the inverse of the ϵ machine) indicates that the matrix `A` is singular (resp. nearly singular).

See also: `mfCondEst`, `mfRCond`, `mfNorm`, `mfSVD`, `mfDet`

`mfRCond` reciprocal condition number estimation

Calling syntax:

```
C = mfRCond( A )
```

Description:

Returns an estimation of the inverse of the condition number (which is called the reciprocal condition number).

Remarks: The reciprocal condition number is always between 0 and 1. A value equal to zero (resp. close to the ϵ machine) indicates a singular matrix (resp. nearly singular).

See also: `mfCond`, `mfCondEst`, `mfNorm`, `mfSVD`, `mfDet`

`msLU` LU factorization

Description:

Computes numerically the LU decomposition of an `mfArray`.

First calling syntax:

```
call msLU( mfOut( L, U [, p] ), A )
```

For dense `mfArrays`, returns the `L` (lower triangular, with a unit diagonal) and `U` (upper triangular) factors, and, optionally, the row permutation vector `p`, such that:

$$L\,U = A(p,:)$$

If the optional argument `p` is not present, the routine actually returns a row permuted L, such that:

$$L\,U = A$$

In this latter case, be aware that you will not be able to solve a linear system using only the factors L and U, because you must know the permutation p to be applied to the right hand side vector. See the *Muesli User's Guide* to an example using the LU decomposition.

Second calling syntax:

```
call msLU ( mfOut( L, U [, p, q, r] ), A )
```

For sparse `mfArrays`, the routine `msLU` behaves in a slightly different manner:

`p` and `q` are respectively row and column permutations vectors while `r` is a vector containing the row scaling of `A`. All these sparse matrices should verify the following (theoretical) relationship:

$$L\,U = P\,R\,A\,Q$$

where the matrices P, Q, R are associated to the vectors p, q, r. Practically, the RHS of the previous identity should be computed using `mfRowScale`, `mfColPerm` and `mfRowPerm`.

Moreover, the `L` factor contains a pointer to an internal structure containing all the five output matrices, so that a call to the `mfLDiv` routine is simplified.

Third calling syntax: In all cases (sparse or dense matrix `A`), the following syntax:

```
call msLU( mfOut(factors), A )
```

provides the possibility to factorize only, without copying factors in explicit `mfArrays`. In this case, the factors (accompanied by the permutations and the scaling) are internally stored (*i. e.* not available for the end-user) and can be referenced only via the `mfMatFactor` `factors`.

The `mfMatFactor` obtained is intended to be used only in `mfLDiv`.

Remark: If the matrix `A` is symmetric and positive definite, the current routine may emit a warning: indeed, a Cholesky factorization (via the `mf/msChol` routine) should be more efficient and uses less memory.

See also: `msQR`, `msRref`, `mfOut`

`msLDLT` $L\,D\,L^t$ factorization

Calling syntax:

```
call msLDLT( mfOut(L,D,P), A )
```

Description:

Computes numerically the $L\,D\,L^t$ decomposition of a real and symmetric (indefinite) `mfArray A`, dense or sparse. Returns a lower triangular `L`, a matrix `D` (block diagonal 1x1 or 2x2) and a permutation matrix `P` such that: $L\,D\,L' = P'\,A\,P$.

Remark: if `A` is moreover positive definite, then the Cholesky factorization should be more appropriate !

N.B.: For sparse matrices:

- the matrix `D` is strictly diagonal;
- be aware that the $L\,D\,L^t$ factorization can fail, especially if the original matrix has some zeros on the main diagonal (the fact that `A` is non-singular is necessary but not sufficient); in this case, use the $L\,U$ factorization instead which makes pivoting.

See also: `msLU`, `mf/msChol`, `mfOut`

`mf/msChol` Cholesky factorization

Description:

Computes numerically the Cholesky factorization of the `mfArray A`, which must be symmetric and positive definite; if this latter property doesn't hold, MUESLI emits an error and returns an empty `mfArray`.

First calling syntax:

```
U = mfChol( A )
```

For dense matrices, returns an upper triangular `mfArray U`, such that: $U'\,U = A$

Second calling syntax: For sparse matrices, the calling syntax is slightly different, because a permutation is first applied to the matrix A in order to reduce the fill-in:

```
call msChol( mfOut(L,p), A )
```

returns a lower triangular `mfArray L` and a permutation `p` such that: $L\,L' = P'\,A\,P$. The permutation `p` is always returned as a special `mfArray` (see `mfPerm`), thus the routines `mf/msColPerm` and `mf/msRowPerm` must be used to applied this permutation to the matrix A.

Third calling syntax: For sparse matrices only, there is also the possibility to factorize only, without copying the factors (`L` and `P`) in explicit `mfArrays`. In this case, the factors are internally stored (*i. e.* not available for the end-user) and can be referenced only via the `mfMatFactor factors`:

```
call msChol( mfOut(factors), A )
```

The `mfMatFactor` obtained is intended to be used only in `mfLDiv`.

See also: `msLU`, `msLDLT`, `mfOut`, `msCholSpSymb`, `msCholSpNum`

`msCholSpSymb` Symbolic Cholesky factorization (sparse only)

Description:

Computes the symbolic Cholesky factorization of the sparse `mfArray A`, which must be symmetric; if this latter property doesn't hold, MUESLI emits an error and returns an empty `mfArray`.

Interface:

```
subroutine msCholSpSymb( factor, A )
   type(mfArray),     intent(in)  :: A
   type(mfMatFactor), intent(out) :: factor
end subroutine
```

After using this routine, the usual step is to call (once or more) the `msCholSpNum` routine, before solving a linear system with `mfLDiv`.

See also: `mf/msChol`

`msCholSpNum` Numerical Cholesky factorization (sparse only)

Description:

After having called the symbolic factorisation (via the `msCholSpSymb` routine), computes the numerical Cholesky factorization of the sparse `mfArray A`, which must be positive definite; if this latter property doesn't hold, MUESLI emits an error and returns an empty `mfArray`.

Interface:

```
subroutine msCholSpNum( factor, A )
   type(mfArray),     intent(in)     :: A
   type(mfMatFactor), intent(in out) :: factor
end subroutine
```

After using this routine, the usual step should be to solve a linear system with `mfLDiv`.

See also: `mf/msChol`

`mf/msBalance` scaling to improve eigenvalue accuracy

The first form:

```
B = mfBalance( A )
```

returns an `mfArray` which is balanced, *i. e.* applies similarity transformations to `A` in order to make the rows and columns as close in norm as possible.

The second form:

```
call msBalance( mfOut(T,B), A )
```

also returns the (non singular) diagonal transformation matrix `T`, such that: $B = T^{-1}AT$

Remark: there is no permutation applied to the matrix `A`. Moreover, the matrix `A` must be square.

See also: `mfNorm`, `mfEig`, `mfOut`

mf/msSVD — singular value decomposition

Calling syntax:

```
S = mfSVD( A )
```

The function form returns only the singular values of the `mfArray A` (size m by n), in a column vector that is sorted from the greatest to the smallest; the size of this vector is $\min(m,n)$ and its type is always real, whatever the type of `A`.

The subroutine form allows the user to access to the full decomposition:

```
call msSVD( mfOut(U,S,V), A [, economy_size ] )
```

which returns three `mfArray`s such that $A = USV'$. `U` and `V` are both unitary matrices whose columns contain, respectively, the left- and right-singular vectors of `A`. In this case, `S` is returned as a rank-2 array.

If the logical optional argument `economy_size` is present and equal to *TRUE*, then `U` contains only the $\min(m,n)$ first columns of the matrix U, and `V` contains only the $\min(m,n)$ first columns of the matrix V. `S` is always a square matrix of order $\min(m,n)$.

Remarks: sparse matrices are not handled; please use `mf/msSVDS` instead.

See also: `mfNorm`, `mfRank`, `mfDet`, `mfOut`

`mfInv, .i.` **matrix inverse**

Calling syntax:

```
C = mfInv( A )
```

`.i. A` (A^{-1}) is a shortcut for writing: `mfInv( A )`

Description:

Returns the inverse of the square `mfArray A`, mathematically noted A^{-1}.

If `A` is singular (or close to, ought to the machine ϵ), it returns an infinite `mfArray` of same shape.

For singular of non square matrices, the Moore-Penrose pseudo-inverse routine `mfPseudoInv` could also be used.

Remarks: sparse matrices are not handled; it is very unusual to have to explicitly compute the inverse of a matrix. If necessary, solving the system $A\,x = e_i$ for each basis vector e_i gives each column of A^{-1}; but be aware that you will obtain a (nearly) dense matrix.

See also: `mfPseudoInv`, `mfLDiv`, `mf/msSVD`, `mfRank`

`mfPseudoInv` — matrix pseudo-inverse

Calling syntax:

```
C = mfPseudoInv( A [, tol ] )
```

Description:

Returns the Moore-Penrose pseudo-inverse of the `mfArray A`.

`A` may be non square or square singular. `mfPseudoInv` returns a matrix, based on the SVD of `A` and any singular values less than the tolerance `tol` are treated as zero. The default tolerance is $\max(m,n)\,||A||_2\,\epsilon$, where m and n are respectively the number of rows and number of columns of the `mfArray A`.

See also: `mfInv`, `mf/msSVD`, `mfRank`

`mfQR` orthogonal-triangular decomposition

Calling syntax:

```
R = mfQR( A )
```

performs a QR factorization of the matrix `A` ($m \times n$, dense or sparse `mfArray`), but returns only the `R` factor (upper triangular matrix of size $n \times n$) verifying $R' R = A' A$. This is the Q-less decomposition.

Remark for sparse matrices: The use of this routine should be restricted to small or moderate size sparse matrices because no ordering is applied to prevent the fill-in. For large sparse matrices, use the `msQR` routine instead, which has also more options.

Note: The complex type is not yet supported for sparse matrices.

See also: `msQR`, `msLU`

`msQR` orthogonal-triangular decomposition

Calling syntax:

```
call msQR( mfOut(Q,R[,p,RANK]), A [, tol] )
```

applies the orthogonal-triangular decomposition to the `mfArray A`, dense or sparse, of size $m \times n$. It returns the two `mfArrays` `Q` and `R` such that $A = Q\,R$.

Usually, `Q` is an orthogonal matrix (size $m \times m$) and `R` is a upper-triangular matrix (size $m \times n$).

If $m > n$, the economy size is always used: `Q` contains only the n first columns of the orthogonal full matrix `Q` (therefore of size $m \times n$), and `R` is the square upper part (*i. e.* the non-zero part) of the full `R` (therefore of size $n \times n$). This latter case is used for least-square problems.

Optionally, a column permutation vector p may be applied to the matrix A, such that $Q\,R = A(:,p)$:

- for the dense case, the use of this option leads to a factor R having the module of its diagonal terms in decreasing order;
- for the sparse case, the permutation is chosen to obtain a better sparsity for both Q and R factors.

If the `mfArray RANK` is present, it contains on return the numerical rank of the matrix `A`:

- for the dense case, a different method is used in order to compute safely the rank: it is the rank-revealing QR;
- for the sparse case, an estimation of the rank is returned.

If the argument `tol` (real double) is present, it specifies to which accuracy the rank is computed. Default value is the machine precision, `MF_EPS`.

Other syntax: for sparse matrices only, the additional syntax can be used:

```
call msQR( mfOut(Qhouse,R[,p,RANK]), A[, tol] )
```

which returns the Householder vectors of the Q matrix in the `mfMatFactor Qhouse` variable, and the R triangular factor in the `mfArray R`. Although `Qhouse` doesn't contain explicitly the matrix `Q`, the `Shape` routine applied to it will returned the shape of `Q`, *i. e.* (m,m). The shape of `R` is (m,n).

This second form should be used for large size matrices, `Qhouse` being much more sparse than `Q`. The user is invited to use one of the routines `mfQleft` or `mfQright` in order to perform the multiplication of Q with another vector or matrix.

The other parameters have the same meaning as in the first calling syntax.

Note: The complex type is not yet supported for sparse matrices.

See also: `mfQR`, `msLU`, `mfOut`

`mfQleft` left multiplication by a sparse Q factor

Calling syntax:

```
B = mfQleft( Qhouse, A )
```

returns the product $Q' A$.

The `mfMatFactor Qhouse` holds a sparse Q factor (obtained from a $Q R$ decomposition of a sparse matrix); the `mfArray A` may be a vector (dense) or a matrix (dense or sparse). B has the same structure (dense or sparse) than A.

By performing the operation: `Q = .t. mfQleft( Qhouse, mfSpEye(m,m) )` the standard form of the Q factor can be obtained (but be aware that `Qhouse` is often much more sparse than `Q`).

Note: The complex case is not yet implemented.

See also: `msQR`, `mfQright`

`mfQright` right multiplication by a sparse Q factor

Calling syntax:

```
B = mfQright( A, Qhouse )
```

returns the product $A\,Q$.

The `mfMatFactor Qhouse` holds a sparse Q factor (obtained from a $Q\,R$ decomposition of a sparse matrix); the `mfArray A` may be a vector (dense) or a matrix (dense or sparse). B has the same structure (dense or sparse) than A.

By performing the operation: `Q = mfQright( mfSpEye(m,m), Qhouse )` the standard form of the Q factor can be obtained (but be aware that `Qhouse` is often much more sparse than `Q`).

Note: The complex case is not yet implemented.

See also: `msQR`, `mfQleft`

`mf/msEig` eigenvalues and eigenvectors

The first form:

```
E = mfEig( A )
```

returns the eigenvalues of the `mfArray A` in the vector `mfArray E`.

The subroutine form:

```
call msEig( mfOut(V,D), A )
```

returns two `mfArrays`. `D` is a diagonal matrix containing the eigenvalues and `V` is the matrix whose columns are the corresponding eigenvectors so that: $A\,V = V\,D$ (or, equivalently, $A = V\,D\,V'$).

Remarks:

- Sparse matrices are not handled; please use `mf/msEigs` instead.
- Eigenvalues are returned in ascending order, according to the magnitude for complex ones.

See also: `mf/msEigs`, `mf/msSchur`, `mfOut`

`mf/msHess` **Hessenberg form**

The first form:

```
H = mfHess( A )
```

computes the Hessenberg form of the `mfArray A`. The returned `mfArray` has the same eigenvalues as `A` but contains zeros below the first subdiagonal.

The subroutine form:

```
call msHess ( mfOut(P,H), A )
```

returns two `mfArrays`. `H` is the Hessenberg form and `P` is a unitary matrix so that: $A = PHP'$.

Remarks: sparse matrices are not (yet) handled.

See also: `mfOut`

`mf/msSchur` Schur decomposition

Calling syntax:

```
call msSchur ( mfOut(U,T), A [, form ] )
```

produces a upper triangular Schur matrix `T` and a unitary matrix `U` so that: $A = UTU'$. `A` must be square.

For a real, dense `mfArray A` the Schur matrix `T` is generally quasi-triangular (real form by default, see below).

The optional character argument `form` may be equal to `"real"` or `"complex"`.

When processing a complex, dense `mfArray A`, or when the optional `form` string is equal to `"complex"`, the matrix `T` is complex, (strictly) upper triangular.

Other syntax:

```
T = mfSchur( A [, form ] )
```

just returns the `T` factor.

Remarks:

- in the real form, the matrix `T` contains 1-by-1 and 2-by-2 diagonal blocks: the 2-by-2 blocks correspond to complex conjugate pairs of eigenvalues of `A`.
  Each 2-by-2 block is in the form:
  $$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$
  where $b\,c < 0$. The eigenvalues of such a block are $a \pm \sqrt{b\,c}$.
- the Schur decomposition is intended to be used for non-symmetric eigenvalue problems; if the `mfArray A` is symmetric/hermitian, it is preferable to compute eigenvalues/vectors via the `mf/msEig` routines (in such a case, `V` and `D` `mfArrays` returned by `mf/msEig` correspond exactly to `U` and `T`). A warning is therefore emitted if `A` is symmetric or hermitian.
- sparse matrices are not (yet) handled.

See also: `mfOut`

`mfLDiv, .ix.` — left matrix divide

Calling syntax:

```
x = mfLDiv( A, b )
```

`A .ix. b` $(A^{-1}b)$ is a shortcut for writing: `mfLDiv( A, b )`

Description: solves a linear system of the form $A\,x = b$, *i. e.* returns $x = A^{-1}b$ without computing A^{-1}.

The `mfArray A` (of size $m \times n$) may be real or complex, sparse or dense, and doesn't need to be square; it can be even rank deficient (tests are made internally to use the most appropriate method):

- if $m > n$ the mean-square solution is returned;
- if $m < n$ the minimum 2-norm solution is returned.

This routine automatically detects if `A` is triangular and chooses the appropriate method.

Multiple right hand sides may be simultaneously solved, by concatenating all the RHS vectors in one rank-2 `mfArray b`. This feature is not available for sparse matrices A (but see below the way of solving simultaneously equations by using intermediate factors).

Note: For the sparse complex case, only square matrices A are supported.

Using this routine is always more efficient than computing the inverse of the matrix `A` first followed by the product with `b`.

Remarks:

- After doing an $L\,U$ factorization of the (dense) matrix `A`, this routine can also be called to solve the linear system $A\,x = P\,L\,U\,x = b$, by using the following syntax:

  ```
  x = mfLDiv( L, U, P, b[, option="transp" ] )
  ```

  The optional last argument allows the user to solve the system $A'\,x = b$ (*transpose* in the real case, *conjugate transpose* in the complex case).

- After doing a Cholesky factorization of the (dense) matrix `A` via `mfChol`, this routine can also be called to solve the linear system $A\,x = U'\,U\,x = b$, by using the following syntax:

  ```
  x = mfLDiv( U, b, form="cholesky" )
  ```

- Similarly,

  ```
  x = mfLDiv( L, U, b )
  ```

  is used for a sparse matrix. As noted in the `msLU` page, the `mfArray L` factor contains a pointer to an internal structure containing all necessary data concerning the $L\,U$ factorization. Note that the *RHS* `b` cannot have here multiple columns.

···/···

– For using with an mfMatFactor `factors` (from, *e. g.*, msLU or mf/msChol), the routine must be called as:

```
x = mfLDiv( factors, b[, option="transp" ] )
```

This latter interface allows the user to simultaneously solve multiple RHS. Moreover, for this case only, the RHS `b` may be dense or sparse. The optional last argument allows the user to solve the system $A' x = b$.

– When a (near) singular matrix is encountered, a *Warning* message may be emitted, according to the message level parameter (see msSetMsgLevel to change this level).

See also: mfRDiv

`mfRDiv, .xi.` — right matrix divide

Calling syntax:

```
x = mfRDiv( b, A )
```

`b .xi. A` $(b\,A^{-1})$ is a shortcut for writing: `mfRDiv( b, A )`

Description: solves linear system of the form $x\,A = b$, *i. e.* returns $x = b\,A^{-1}$ without computing A^{-1}.

The `mfArray A` (of size $m \times n$) may be real or complex, sparse or dense, and doesn't need to be square; it can be even rank deficient (tests are made internally to use the most appropriate method):

- if $n > m$ the least-square solution is returned;
- if $n < m$ the minimum 2-norm solution is returned.

This routine automatically detects if `A` is triangular and chooses the appropriate method.

Multiple right hand sides may be simultaneously solved, by concatenating all the RHS vectors in one rank-2 `mfArray b`. This feature is not available for sparse matrices A (you should use `mfLDiv` instead, by using transposition).

Note: For the sparse complex case, only square matrices A are supported.

Using this routine is always more efficient than computing the inverse of the matrix `A` first followed by the product with `b`.

Remarks:

- After doing an $L\,U$ factorization of the (dense) matrix `A`, this routine can also be called to solve the linear system $x\,A = x\,P\,L\,U x = b$, by using the following syntax:

  ```
  x = mfRDiv( b, L, U, P )
  ```

- After doing a Cholesky factorization of the (dense) matrix `A` via `mf/msChol`, this routine can also be called to solve the linear system $x\,A = x\,U'\,U = b$, by using the following syntax:

  ```
  x = mfRDiv( b, U, form="cholesky" )
  ```

- Similarly,

  ```
  x = mfRDiv( L, U, b )
  ```

  is used for a sparse matrix. As noted in the `msLU` page, the `mfArray L` factor contains a pointer to an internal structure containing all necessary data concerning the $L\,U$ factorization. Note that the *RHS* `b` cannot have here multiple columns.

- When a (near) singular matrix is encountered, a *Warning* message may be emitted, according to the message level parameter (see `msSetMsgLevel` to change this level).

See also: `mfLDiv`

`msRref` reduced row echelon form

Calling syntax:

```
call msRref ( mfOut(R,jpiv), A [, tol ] )
```

computes the reduced row echelon form of the `mfArray A`.

The optional real `tol` is the tolerance (should be positive) used to compute the rank of the matrix. This tolerance cannot be too small. Minimum is the machine precision. When it is known, use the precision of the elements of the matrix `A`.

Note that the rank provided by this method is only an approximation which may differ from that computed with other methods (*e. g.*, the Singular Value Decomposition `msSVD`). However, this approach is pedagogically interesting.

See also: `mfRank`, `mfOrth`, `mfNull`, `msQR`, `mfOut`

`mfNull` — null space

Calling syntax:

```
B = mfNull( A [, rational, tol ] )
```

returns an orthonormal basis for the null space of the `mfArray A`.

The optional logical `rational` argument specifies whether the rational basis is expected or not.

The optional real `tol` argument is the tolerance (should be positive) used to compute the rank of the matrix. As `mfNull` calls `msRref`, `tol` cannot be less than the machine ϵ.

See also: `mfRank`, `mfOrth`, `msRref`, `msQR`, `msSVD`

`mfOrth` — matrix orthogonalization

Calling syntax:

```
B = mfOrth( A )
```

returns an orthonormal basis for the range of the `mfArray A`.

See also: `mfRank`, `mfNull`, `msRref`, `msQR`, `msSVD`

`mfExpm` — matrix exponential

Calling syntax:

```
B = mfExpm( A )
```

computes the matrix exponential of the `mfArray A` (dense only).

See also: `mfExp`, `mfLogm`, `mfSqrtm`, `mfPowm`, `mfFunm`

`mfLogm` — matrix logarithm

Calling syntax:

```
B = mfLogm( A )
```

computes the matrix logarithm of the `mfArray A` (dense only).

See also: `mfLog`, `mfExpm`, `mfSqrtm`, `mfPowm`, `mfFunm`

`mfSqrtm` — matrix square root

Calling syntax:

```
B = mfSqrtm( A )
```

computes the matrix square root of the `mfArray A` (dense only).

The property $BB = A$ doesn't always hold, because A may not have a square root.

See also: `mfSqrt`, `mfExpm`, `mfLogm`, `mfPowm`, `mfFunm`

`mfPowm` — matrix power

Calling syntax:

```
B = mfPowm( A, e )
```

computes the matrix power of the `mfArray A`, *i. e.* A^e.

If `e` is integer, then `A` may have a sparse or a dense structure. Any positive integer is accepted.

If `e` is real, then `A` must have a dense structure. Any real value for `e` is accepted.

See also: `mfPow10`, `mfExpm`, `mfLogm`, `mfSqrtm`, `mfFunm`

`mf/msFunm` general matrix function

Calling syntax:

```
F = mfFunm( A, fun )
```

applies the matrix function `fun` to the `mfArray A` (dense only).

`fun` is either a character string giving the name of a usual function (as *sin*, *cos*, *asin*, ...), or the name of a user function. In this latter case, the user function must have the following interface:

```
function fun(z1) result(z2)
   complex(kind=MF_DOUBLE) :: z1, z2
end function
```

The subroutine form, which has the calling syntax:

```
call msFunm( mfOut(F,e), A, fun )
```

returns also an estimation of the error in the scalar `mfArray e`. Note that in the case of the exponential function (`fun` set to `"exp"`), the `mfExpm` is directly called, so that the error estimation will be not available (the `mfArray e` will be empty).

Remark: Use `mfFun` to apply the function `fun` in a element-wise manner.

See also: `mfExpm`, `mfLogm`, `mfSqrtm`, `mfPowm`, `mfOut`

`mf/msEigs` few eigenvalues and eigenvectors

The first form:

```
E = mfEigs( A, k [, which|sigma] [, tol, ncv] )
```

returns few eigenvalues of the matrix `A` using ARPACK in the vector `mfArray E`. `A` may be real/complex, sparse/dense.

The number of requested eignevalues `k` must be ranged in $[2, n-2]$.

`which` is a `character(len=2)` optional argument, can take the values `"LM"` or `"SM"`, depending on whether the Largest or the Smaller Magnitude eigenvalues/vectors are requested. In addition, `which` can take the following values:

- for real, symmetric matrices:

  `"LA"` = Largest Algebraic, with an algebraic selection;

  `"SA"` = Smallest Algebraic, with an algebraic selection;

  `"BE"` = Both Ends, with an algebraic selection (half from the largest ones and half from the smallest ones.

- for real, unsymmetric matrices:

  `"LR"` = Largest Real part, with an *algebraic* selection;

  `"SR"` = Smallest Real part, with an *algebraic* selection;

  `"LI"` = Largest Imaginary part, with a *magnitude*[1] selection;

  `"SI"` = Smallest Imaginary part, with a *magnitude*[1] selection;

- for complex matrices:

  `"LR"` = Largest Real part, with an *algebraic* selection;

  `"SR"` = Smallest Real part, with an *algebraic* selection;

  `"LI"` = Largest Imaginary part, with a *algebraic* selection;

  `"SI"` = Smallest Imaginary part, with a *algebraic* selection;

Eigenvalues are generally returned in a ascending order, excepted for the `"LM"` and `"LA"` cases, for which they are returned in a descending order.

`sigma` is a `complex` optional argument. If present, `mfEigs` tries to find the `k` eigenvalues closest to the complex `sigma`.

If `tol` is present, it indicates the tolerance of the computed eigenvalues. By default, the machine epsilon is taken.

If `ncv` is present, it indicates how many Arnoldi vectors are generated (following the ARPACK documentation, the minimum value for `ncv` is 2*k+1 and the maximum value is the order n of the matrix; a warning is emitted if these conditions are not fulfilled).

If `ncv` is not present, the `mfEigs` routine tries larger and larger values of `ncv`, until (hopefully) the convergence of all requested eigenvalues.

In both cases (`ncv` present or not present), a warning is issued if not all eigenvalues converged. Use the subroutine form, explained below, to obtain a quiet behavior.

· · · / . . .

[1] in order to keep adjacent the complex conjugate pairs.

The subroutine form:

```
call msEigs( mfOut(V,E,flag), A, k [, which|sigma] [, tol, ncv] )
```

returns the requested eigenvalues in the vector `mfArray E`, the corresponding eigenvectors in the columns of the `mfArray V`, and the convergence of the iterative process. If the boolean `mfArray flag` is *TRUE* then all the eigenvalues converged; otherwise not all converged. Note that non-converged eigenvalues/eigenvectors are set to NaN.

The remaining arguments are processed as described above.

See also: `mf/msEig`, `mf/msSVDS`, `mfOut`

`mf/msSVDS` few singular values

The first form:

```
S = mfSVDS( A, k [, which, tol, ncv ] )
```

returns few singular values of the matrix `A` using ARPACK in the vector `mfArray S`. `A` may be real/complex, sparse/dense; it doesn't need to be square.

`k`, the number of required singular values, cannot be larger than $\min(m,n)-2$. If so, *NaN* values are used to fill the vector `S`.

`which` is a `character(len=2)` optional argument, which must be equal to `"LM"`, `"SM"` or `"BE"`. If present, `mfSVDS` tries to find the `k` singular values of largest magnitude (`"LM"`), or smallest magnitude (`"SM"`). Only for real matrices, `"BE"` allows to compute the both-end singular values. Default value is `"LM"`.

If `tol` is present, it indicates the tolerance of the computed singular values. By default, the machine epsilon is taken.

If `ncv` is present, it indicates how many Arnoldi vectors are generated (following the ARPACK documentation, the minimum value for `ncv` is 2*`k`+1 and the maximum value is $\min(m,n)$ of the matrix; a warning is emitted if these conditions are not fulfilled).

If `ncv` is not present, the `mfSVDS` routine tries larger and larger values of `ncv`, until (hopefully) the convergence of all requested singular values.

In both cases (`ncv` present or not present), a warning is issued if not all singular values converged. Use the subroutine form, explained below, to obtain a quiet behavior.

The subroutine form:

```
call msSVDS( mfOut(S,flag), A, k [, which, tol, ncv ] )
```

returns the requested singular values in the vector `mfArray S` and the convergence of the iterative process. If the boolean `mfArray flag` is *TRUE* then all the singular values converged; otherwise not all converged. The remaining arguments are processed as described above.

See also: `mf/msSVD`, `mf/msEigs`, `mfOut`

`mfNormEst`, `mfNormEst1` — norm estimations

Syntax:

```
C = mfNormEst( A [, tol ] )
```

Description:

Returns an estimation of the 2-norm of a rank-2 `mfArray` (dense or sparse, real or complex, square or non-square).

The optional argument allows the user to choose the tolerance used in the iterative algorithm. By default, tolerance is 10^{-6}.

Syntax:

```
C = mfNormEst1( A )
```

returns an estimation of the 1-norm of a square matrix (sparse or dense).

See also: `mfNorm`

`mfCondEst` — matrix condition number estimation

Calling syntax:

```
C = mfCondEst( A )
```

Description:

Returns an estimation of the condition number of the rank-2 `mfArray A`.

Remarks: The condition number is always greater than 1. A value equal to infinity (resp. close to the inverse of the ϵ machine) indicates that the matrix `A` is singular (resp. nearly singular).

See also: `mfCond`, `mfRCond`

`mfIsSymm` — matrix symmetry inquiry

Calling syntax:

```
bool = mfIsSymm( A [, option="pattern", tol ] )
```

returns a boolean value according to the symmetry property of the `mfArray A`. If the matrix is complex, this property means hermitian.

For a sparse `mfArray A`, if the optional argument `option` is present and equal to `"pattern"`, then the routine checks only for the symmetry of the sparse structure.

If the optional tolerance `tol` is present, the symmetry test (for the real case) is made according to the rule:

$$||A - A'||_1 < 2\,\mathrm{tol}\,||\mathrm{A}||_1$$

Default tolerance is `MF_EPS`.

See also: `mfIsPosDef`, `mfTolForSymm`

`mfTolForSymm` — tolerance for symmetry

Calling syntax:

```
tol = mfTolForSymm( A )
```

returns the tolerance for which the `mfArray A` should be symmetric.

See also: `mfIsSymm`

`mfIsPosDef` definite positiveness inquiry

Calling syntax:

```
bool = mfIsPosDef( A )
```

returns a boolean value according to the definite positiveness property of the `mfArray A`.

See also: `mfIsSymm`

`mfIsDiagDomCol` — diagonally dominant by cols inquiry

Calling syntax:

```
bool = mfIsDiagDomCol( A )
```

returns a boolean value according to the "diagonally dominant by cols" property of the `mfArray A`.

See also: `mfIsStrictDiagDomCol`

`mfIsStrictDiagDomCol` strictly diagonally dominant by cols inquiry

Calling syntax:

```
bool = mfIsStrictDiagDomCol( A )
```

returns a boolean value according to the "strictly diagonally dominant by cols" property of the `mfArray A`.

See also: `mfIsDiagDomCol`

`mfIsFullRank` — full rank inquiry

Calling syntax:

```
bool = mfIsFullRank( A [, tol ] )
```

returns '`.true.`' if the `mfArray A` is full rank, *i. e.* if its rank is equal to the minimum of its dimensions.

If `tol`, of type `real`, is present then this value is used as tolerance in the rank computation. Default tolerance is chosen as $\max(m,n)||A||\epsilon$.

See also: `mfRank`

MF_LAPACK_VERSION — LAPACK version

Calling syntax:

```
string = MF_LAPACK_VERSION()
```

returns the LAPACK version used during the link of the executable.

Note the parenthesis used after the name of the variable, because it is implemented as a function.

The returned `string` may be used by the `mfIsVersion` boolean function.

See also: `MF_COMPILER_VERSION`, `MF_MUESLI_VERSION`, `msGetBlasLib`, `msGetLapackLib`, `msGetArpackInfo`, `msGetSuiteSparseLib`

`msGetBlasLib` — Get BLAS library type

Calling syntax:

```
call msGetBlasLib( string )
```

returns the BLAS library type used during the link of the executable, that is, either *Reference* for most compilers or a specific package for others (*e. g. MKL* for the INTEL compiler suite; *OpenBLAS* or *ATLAS* for the GNU compiler suite).

`string` is a character string of length at least 128.

The current routine is able to detect the dynamic switch between BLAS implementations like *OpenBLAS* and *ATLAS*.

See also: `MF_COMPILER_VERSION`, `MF_MUESLI_VERSION`, `MF_LAPACK_VERSION`, `msGetLapackLib`, `msGetArpackInfo`, `msGetSuiteSparseLib`

`msGetLapackLib` Get LAPACK library type

Calling syntax:

```
call msGetLapackLib( string )
```

returns the LAPACK library type used during the link of the executable, that is, either *Reference* for most compilers or a specific package for others (*e. g. MKL* for the INTEL compiler suite or *ATLAS* for the GNU compiler suite).

`string` is a character string of length at least 128.

The current routine is able to detect the dynamic switch between BLAS implementations like *Reference* and *ATLAS*.

See also: `MF_COMPILER_VERSION`, `MF_MUESLI_VERSION`, `MF_LAPACK_VERSION`, `msGetBlasLib`, `msGetArpackInfo`, `msGetSuiteSparseLib`

msGetArpackInfo — Get ARPACK library information

Calling syntax:

```
call msGetArpackInfo( version[, info] )
```

`version` is a character string of length at least 24, containing on return the version of the library. The `"NG"` letters indicates the *new generation* version, which can be found at: http://forge.scilab.org/index.php/p/arpack-ng/.

`info` is an optional character string of length at least 128, containing on return additional information, especially various non official fixes applied to the library.

See also: MF_COMPILER_VERSION, MF_MUESLI_VERSION, MF_LAPACK_VERSION, msGetBlasLib, msGetLapackLib, msGetSuiteSparseLib

`msGetSuiteSparseLib` Get SuiteSparse library version

Calling syntax:

```
call msGetSuiteSparseLib( version )
```

`version` is a character string of length at least 8, containing on return the version of the library.

The returned `string` may be used by the `mfIsVersion` boolean function.

See also: `MF_COMPILER_VERSION`, `MF_MUESLI_VERSION`, `MF_LAPACK_VERSION`, `msGetBlasLib`, `msGetLapackLib`, `msGetArpackInfo`

1.9 Polynomial Functions

mfPolyVal	polynomial evaluation
mfPolyFit, msPolyFit	polynomial fitting
mfFunFit, msFunFit	nonlinear function fitting
mfRoots	roots of a polynomial
mfPoly	polynomial with specified roots
mfSpline, msSpline	cubic spline interpolation or smoothing
mfLegendre	Legendre polynomials
mfPPVal	piecewise polynomial evaluation
mfPPDer	piecewise polynomial derivative
mfInterp1	1D interpolation
mfInterp2	regular 2D interpolation
mfPSLG	Planar Straight Line Graph *(derived type)*
msPrintPSLG	Planar Straight Line Graph display
mfDelaunay	Delaunay 2D tessellation (triangulation)
mfTriConnect	2D triangular connectivity *(derived type)*
msBuildTriConnect	2D triangular connectivity initialization
msUpdateTriConnect	2D triangular connectivity update
msExtractTriConnect	get components from a mfTriConnect structure
msCheckDomainConvexity	Check convexity of a 2D triangulation
msPrintTriConnect	2D triangular connectivity display
msTriNodeNeighbors	2D triangulation node neighbors
mfTriSearch, mfNodeSearch	2D triangulation search
mfVoronoi	2D Voronoi diagram from a set of points
mfVoronoiStruct	2D Voronoi structure *(derived type)*
msPrintVoronoi	2D Voronoi structure display
mfGridData	irregular 2D interpolation
mfDelaunay3D	Delaunay 3D tessellation (tetrahedralization)
msEndDelaunay3D	Internal free storage after Delaunay 3D
mfTetraConnect	3D tetrahedral connectivity *(derived type)*
msBuildTetraConnect	3D tetrahedral connectivity initialization
msExtractTetraConnect	get components from a mfTetraConnect structure
msPrintTetraConnect	3D tetrahedral connectivity display
msDel3DNodeNeighbors	3D tetrahedralization node neighbors
mfTetraSearch, mfNodeSearch3D	3D tetrahedralization search
mfGridData3D	irregular 3D interpolation

See also:

Core Routines

File Input/Output

Data Analysis Functions

Operators

Elementary Math Functions

Specialized Math Functions

Elementary Matrix Manipulation Functions

Matrix Functions

Optimization and Function Functions

Sparse Matrices

`mfPolyVal` **polynomial evaluation**

Interface:

```
function mfPolyVal( p, x ) result( out )

   type(mfArray), intent(in) :: p, x
   type(mfArray) :: out
```

Description:

Evaluates polynomial `p` at points `x`.

The `mfArray p` is a vector of length k+1 whose elements are the coefficients of the polynomial in descending powers.

$$out = p_1\, x^k + p_2\, x^{k-1} + ... + p_k\, x + p_{k+1}$$

See also: `mfPPVal`, `mfPolyFit`

`mfPolyFit` **polynomial fitting**

Interface:

```
function mfPolyFit( x, y, n ) result( p )

   type(mfArray), intent(in) :: x, y
   integer, intent(in) :: n
   type(mfArray) :: p
```

Description:

Fit polynomial to real data.

The `mfArray` `x` and `y` must be column vectors of same shape. They are coordinates of the data. `n` is the degree of the polynomial to be obtained, so $n+1$ real coefficients are returned in the vector `p`. These coefficients are sorted for descending powers.

See also: `msPolyFit`, `mfPolyVal`, `mfFunFit`

`msPolyFit` — polynomial fitting

Calling syntax:

```
call msPolyFit( mfOut(p, normr[, r2]), x, y, n )
```

Description:

Similar to `mfPolyFit`, but also returns (in `mfArrays`) the norm-2 of the residuals (in `normr`) and optionally the correlation coefficient r^2_{corr} (in `r2`).

Nota: The correlation coefficient of the approximation is computed as follows:

$$r^2_{\text{corr}} = 1 - \frac{\texttt{normr}^2}{M\ \sigma^2}$$

where σ^2 is the standard deviation (as computed by `mfVar(y)`) and M is the number of data points in `x` and `y`.

See also: `mfPolyVal`, `msFunFit`, `mfOut`

`mfFunFit` — nonlinear function fitting

Interface:

```
function mfFunFit( x, y, fun, p0, n, tol ) result( out )

   type(mfArray), intent(in) :: x, y, p0
   interface
      function fun( x, p, n ) result( res )
         import :: MF_DOUBLE
         real(kind=MF_DOUBLE), intent(in) :: x
         real(kind=MF_DOUBLE), intent(in) :: p(n)
         integer,              intent(in) :: n
         real(kind=MF_DOUBLE) :: res
      end function fun
   end interface
   integer,              intent(in) :: n
   real(kind=MF_DOUBLE), intent(in), optional :: tol

   type(mfArray) :: out
```

Description:

Fit arbitrary (nonlinear) function to real data.

Data is stored in the `mfArrays` `x` and `y`, which must be column vectors of same shape. There are m couples (x, y).

`fun` is the real function which tries to model the data. This function is parameterized by `n` variables stored in the `mfArray p`.

At input, `p0` contains an initial guess for the values of the `n` unknown parameters. Final values are stored in the `mfArray out`.

`tol` is the tolerance for the convergence. Its default value is 10^{-6}.

Remarks:

- the problem is formulated as a non-linear least-square problem, solved by package *MINPACK*;
- therefore, m (the number of couples (x, y)) must be bigger than the number of parameters `n`.

See also: `msFunFit`, `mfPolyFit`, `mfLsqNonLin`

`msFunFit` — nonlinear function fitting

Calling syntax:

```
call msFunFit( mfOut(p[, r2]), x, y, fun, p0, n [, tol ] )
```

Description:

Similar to `mfFunFit`, but in addition it returns the correlation coefficient r^2_{corr} (in the `mfArray r2`).

For this subroutine, the output of the desired `n` variables is stored in the `mfArray p`.

Remark: Please note that the correlation coefficient r^2_{corr} is ranged in the interval $[-1, +1]$. A value close to $+1$ indicates a perfect correlation while a null value indicates no correlation. A negative value means that the relationship between original data and fitted data is very bad most of time (practically, this should indicates that your function `fun` is not adapted to the data).

See also: `mfOut`

`mfRoots` — roots of a polynomial

Calling syntax:

```
r = mfRoots( p )
```

Description:

Returns the roots of the polynomial whose coefficients are provided in the `mfArray p` (row or column vector).

This polynomial writes:

$$P(x) = p_1\, x^{n-1} + p_2\, x^{n-2} + \ \ldots\ + p_{n-1}\, x + p_n$$

where $[p_1, p_2, \ldots, p_n]$ are the n components of the input vector `p`.

Even if the roots are real, the returned `mfArray` is complex. The roots are not sorted.

Remarks:

- `p` must be of type real.
- The `mfPoly` function is the inverse of the `mfRoots` function.

See also: `mfCompan`, `mfEig`, `mfFZero`

`mfPoly` — polynomial with specified roots

Calling syntax:

```
p = mfPoly( r )
```

Description:

Returns the coefficients of the polynomial whose roots are provided in the `mfArray r` (row or column vector).

If n is the size of `p` then the output polynomial reads:

$$P(x) = p_1\, x^{n-1} + p_2\, x^{n-2} + \ \ldots\ + p_{n-1}\, x + p_n$$

Remarks:

- The specified roots in `r` may be complex.
- The `mfRoots` function is the inverse of the `mfPoly` function.

See also: `mfCompan`, `mfEig`

`mfSpline` — cubic spline interpolation

Interface:

```
function mfSpline( x, y,                                                    &
                   xi,                                                      &
                   BC_type_1, BC_val_1, BC_type_2, BC_val_2,                &
                   periodic )                                               &
result( y_sec )

   type(mfArray),         intent(in)           :: x, y
   type(mfArray),         intent(in), optional :: xi
   integer,               intent(in), optional :: BC_type_1
   real(kind=MF_DOUBLE), intent(in), optional :: BC_val_1
   integer,               intent(in), optional :: BC_type_2
   real(kind=MF_DOUBLE), intent(in), optional :: BC_val_2
   logical,               intent(in), optional :: periodic

   type(mfArray) :: y_sec
```

Description:

If `xi` is not present, this function returns the coefficients of the spline (piecewise cubic polynomial form) which interpolates the values `y` at the abscissas `x`. The returned coefficients in `y_sec` (second derivative at abscissas) are intended to be used with the `mfPPVal` function.

If `xi` is present, `mfSpline` returns only the interpolated values at the entries contained in the `mfArray` `xi`.

By default, the spline form is natural, which means that the first and the last coefficients are null (*i. e.* curvature is null at both ends). Optionally, boundary conditions of various (and mixed) types may be specified at the ends of the interval:

`BC_type_1` $= 1$ implies that $y' =$ `BC_val_1` at first end;

`BC_type_1` $= 2$ implies that $y'' =$ `BC_val_1` at first end;

`BC_type_2` $= 1$ implies that $y' =$ `BC_val_2` at second end;

`BC_type_2` $= 2$ implies that $y'' =$ `BC_val_2` at second end;

If `periodic` is present and equal to *TRUE*, then the value of `y`, along with the first and second derivatives are equal at the two ends of the interval. Default is *FALSE*.

Remarks:

- `x` and `y` must be vectors having the same shape, but `xi` may have any shape.
- Abscissas `x` doesn't need to be equally spaced; `x` must contain at least 2 points.
- `x`, `y` and `xi` may be temporary mfArrays.
- interpolated values, at abscissas different than the nodes in `x` may be obtained by using the `mfPPVal` function.
- smoothing of data may be obtained by using the `msSpline` routine.

`msSpline` cubic spline smoothing

Interface:

```
call msSpline( mfOut(y_smooth, y_sec), x, y, weights                    &
               [, BC_type_1, BC_val_1, BC_type_2, BC_val_2] )
```

Description:

Similar to the `mfSpline` function (in particular the last four optional arguments), but apply a smoothing to the data, instead of a strict interpolation. The new (smoothed) values of `y` are stored in the `y_smooth mfArray`, and the cubic spline coefficients are stored in the `y_sec mfArray`.

Remarks:

- The `weights mfArray` contains the weight of the smoothing, to be applied to each value of the `y mfArray`; it may be a scalar: in such a case, all weights are equal. Weights must be strictly positive reals from 0 to infinity. If some weights are less or equal to zero, this will lead to an error.
- When choosing infinite weights, this routine gives the same results than the `mfSpline` function, *i. e.* a strict interpolation of the data; on the other end, choosing very small weights leads to the least-square linear regression of all the data points.
- Typically, the values of the weights vary according to the powers of ten. If the abscissas are not equally spaced, you may choose to scale your weights with h^3, where h is the interval between the `x` values.

See also: `mfOut`

`mfLegendre` — Legendre polynomials

Calling syntax:

```
C = mfLegendre( n, A )
```

Description:

Returns the evaluation of the `n`-th Legendre polynomial (degree n) of each element of the `mfArray A`.

Restriction: `A` must be of type real.

`mfPPVal` piecewise polynomial evaluation

Interface:

```
function mfPPVal( x, y, p, xi, extrapol ) result( yi )

   type(mfArray),                  intent(in)           :: p, xi
  (logical|real(kind=MF_DOUBLE)), intent(in), optional :: extrapol
   type(mfArray)                                       :: yi
```

Description:

Computes the interpolated values at the entries `xi`.

The way this function treats out-of-range input values in `xi` is similar to that of `mfInterp1` (see the remarks below).

`p` is the vector of Spline coefficients – of the curve (`x`,`y`) – returned by `mfSpline`.

Remarks:

- `x`, `y`, `p` and `xi` may be temporary mfArrays.
- `xi` mfArray may have any shape.
- By default, an element of `yi` is set to `MF_NAN` if the corresponding value in `xi` is outside the range of `x`. The optional argument `extrapol` may be used to override this behaviour: a real value allows the use of a constant for out-of-range inputs. More interestingly, a logical value can be used to compute an extrapolated value by using the interpolation law of the nearest valid interval.

See also: `mfPolyVal`, `mfPPDer`

`mfPPDer` piecewise polynomial derivative

Interface:

```
function mfPPDer( x, y, p, xi, extrapol ) result( yi )

   type(mfArray),                  intent(in)           :: p, xi
  (logical|real(kind=MF_DOUBLE)), intent(in), optional :: extrapol
   type(mfArray)                                       :: yi
```

Description:

Similar to `mfPPVal` but returns the derivative instead of the function value.

`p` is the vector of Spline coefficients – of the curve (`x`,`y`) – returned by `mfSpline`.

`mfInterp1` 1D interpolation

Interface:

```
function mfInterp1( x, y, xi, order, extrapol ) result( yi )

   type(mfArray),                 intent(in)           :: x, y, xi
   integer,                       intent(in), optional :: order
  (logical|real(kind=MF_DOUBLE)), intent(in), optional :: extrapol
   type(mfArray)                                       :: yi
```

Description:

Interpolates data `y`, which are defined along a vector `x`, at the entry(ies) specified by `xi`.

`x` and `y` must be vectors with the same shape; `x` must have strictly monotonous values.

`xi` may be a scalar or a vector; `yi`, which contains interpolated data, will have the same shape as `xi`.

Remarks:

- By default a *linear interpolation* (2-point stencil) is used (`order=1`); set the optional argument `order` to 2 (resp. 3) in order to use a *quadratic* (resp. *cubic*) *interpolation* using a 3-point (resp. 4-point) stencil. A *nearest interpolation* (1-point stencil) is also possible: set `order` to 0.
- By default, an element of `yi` is set to `MF_NAN` if the corresponding value in `xi` is outside the range of `x`. The optional argument `extrapol` may be used to override this behaviour: a real value allows the use of a constant for out-of-range inputs. More interestingly, a logical value can be used to compute an extrapolated value by using the interpolation law of the nearest valid interval.

See also: `mfSpline`, `mfInterp2`, `mfGridData`, `msMeshGrid`

Example(s):

```
x = mfLinspace( 0.0d0, 1.0d0, 5 )
call msDisplay( x, "x" )
print *, "y = x**2"
y = x**2
xi = mfLinspace( 0.05d0, 0.95d0, 4 )
call msDisplay( xi, "xi" )
yi = mfInterp1( x, y, xi, order=2 )
call msDisplay( yi, "interpolated data at many points" )
```

output:

```
x =

   0.0000    0.2500    0.5000    0.7500    1.0000

y = x**2

xi =

   0.0500    0.3500    0.6500    0.9500

interpolated data at many points =

   0.0025    0.1225    0.4225    0.9025
```

···/···

```
  xi = mfLinspace( -0.15d0, 1.15d0, 7 )
  call msDisplay( xi, "xi" )
  yi = mfInterp1( x, y, xi, order=2 )
  call msDisplay( yi, "interpolated data at many points (2 NaNs expected)" )
```

output:

```
 xi =

   -0.1500    0.0667    0.2833    0.5000    0.7167    0.9333    1.1500

 interpolated data at many points (2 NaNs expected) =

      NaN     0.0044    0.0803    0.2500    0.5136    0.8711       NaN
```

```
  yi = mfInterp1( x, y, xi, order=2, extrapol=MF_INF )
  call msDisplay( yi, "interpolated data at many points (2 Infs expected)" )
```

output:

```
 interpolated data at many points (2 Infs expected) =

      Inf     0.0044    0.0803    0.2500    0.5136    0.8711       Inf
```

```
  yi = mfInterp1( x, y, xi, order=2, extrapol=.true. )
  call msDisplay( yi, "interpolated data at many points (using extrapolation)" )
```

output:

```
 interpolated data at many points (using extrapolation) =

    0.0225    0.0044    0.0803    0.2500    0.5136    0.8711    1.3225
```

`mfInterp2` — regular 2D interpolation

Interface:

```
function mfInterp2( x, y, z, xi, yi, order, extrapol ) result( zi )

   type(mfArray),                 intent(in)           :: x, y, z, xi, yi
   integer,                       intent(in), optional :: order
  (logical|real(kind=MF_DOUBLE)), intent(in), optional :: extrapol
   type(mfArray)                                       :: zi
```

Description:

Interpolates data `z`, which are defined on a rectangular, horizontal grid `(x,y)`, at the entry(ies) specified by `xi` and `yi`.

`x`, `y` and `z` must be matrices with the same shape; `x`, `y` must have strictly monotonous values. Usually, `x` and `y` are built with the routine `msMeshGrid`.

`(xi,yi)` also must have the same shape (scalars, vectors, or matrices); `zi`, which contains interpolated data, will have the same shape as `xi` and `yi`.

Remarks:

- By default a *bilinear interpolation* (4-point stencil) is used (`order=1`); set the optional argument `order` to 2 (resp. 3) in order to use a *biquadratic* (resp. *bicubic*) *interpolation* using a 9-point (resp. 16-point) stencil. A *nearest interpolation* (1-point stencil) is also possible: set `order` to 0.
- Note that the method used is valid only for rectangular, horizontal grid. For a general, irregular grid, see `mfGridData`.
- By default, an element of `zi` is set to `MF_NAN` if the corresponding value in `(xi,yi)` is outside the range of `x` and `y`. The optional argument `extrapol` may be used to override this behaviour: a real value allows the use of a constant for out-of-range inputs. More interestingly, a logical value can be used to compute an extrapolated value by using the interpolation law of the nearest valid interval.

See also: `mfInterp1`, `msMeshGrid`

`mfPSLG` Planar Straight Line Graph *(derived type)*

Description:

This derived type encapsulates internal arrays describing a 2D domain definition, using polygonal shapes.

Declaration is made as follows:

```
type(mfPSLG) :: PSLG_domain
```

Contrary to other Muesli derived types, all internal fields are public, letting the user to allocate/deallocate and set the data himself. The complete definition of this kind of object is:

```
type :: mfPSLG

   real(kind=MF_DOUBLE), allocatable :: n_xy(:,:), holes_xy(:,:)

   integer,              allocatable :: edge_n(:,:)

end type
```

`n_xy` contains the coordinates of the nodes; must be of size $(nn, 2)$, with $nn \geq 3$.

`edge_n` contains the definition of each edge segment, by its two endpoints; ; must be of size $(ne, 2)$, with $ne \geq 0$.

`holes_xy` contains the definition of holes; they are points having two coordinates; must be of size $(nh, 2)$, with $nh \geq 0$. A hole should be, ideally, located inside a closed boundary of edges or, alternatively, outside the domain but inside the convex hull (in this later case, its role is to allow concavities). The hole points must not be located exactly on any edge.

Remarks:

- This structure can be used directly by `msDelaunay` to triangulate the corresponding domain.
- See examples of PSLG domains in the *Muesli User's Guide*.
- A variable of this type may be freed, at the end of its use, by the `msRelease` routine.

See also: `msPrintPSLG`, `msPlotPSLG`

`msPrintPSLG` Display of a Planar Straight Line Graph

Calling syntax:

```
call msPrintPSLG( PSLG_domain [, short_info ] )
```

Description:

For a small number of nodes (typically less than few dozens), this routine print on the screen the definition of a Planar Straight Line Graph, which is used to represent a domain geometry.

The input argument `PSLG_domain` must be of type `mfPSLG`.

When the boolean optional argument `short_info` is present and equal to *TRUE* then the routine just print the number of items, *i. e.* nodes, edges and holes. Therefore, it is the only way to inspect the structure when it is big.

See also: `msPlotPSLG`

`mfDelaunay` Delaunay 2D tessellation (triangulation)

First calling syntax:

```
tri = mfDelaunay( x_in, y_in )
```

Description:

Builds a Delaunay 2D triangulation from a list of nodes whose coordinates are given in `x_in` and `y_in` (these two `mfArray`s must be vectors having the same shape).

The returned `mfArray` contains indices for the triangles: each row of `tri` provides three integers which describes a triangle (direct orientation, when travelling from first node to the second and the third ones).

Second calling syntax:

```
call msDelaunay( mfOut( x, y, tri ), x_in, y_in,                    &
                 [ theta_min, area_max ] )
```

Description:

Builds a constrained conforming Delaunay 2D triangulation (as previously, from a given list of nodes whose coordinates are stored in `x_in` and `y_in`), giving a quality mesh by adding new nodes where it is required. Two different criteria are used:

- A minimal angle `theta_min`, to avoid triangles with too small angles. This angle, in degree, must be less than 28.6 (otherwise the algorithm may fail to converge – anyway, the routine will emit an error). The recommanded value ranges from 20 to 25.
- A maximum area `area_max`, to avoid big triangles.

These two optional arguments are `real`. At least one is required.

Third calling syntax:

```
call msDelaunay( mfOut( x, y, tri ), PSLG_domain,                    &
                 [ theta_min, area_max ] )
```

Description:

Same as the above interface, but take a PSLG domain on input (see `mfPSLG`), instead of a list of nodes. PSLG means *Planar Straight Line Graph* and, very briefly, can be viewed as a polygonal description of the boundaries of a domain; it is intended to store the description of a plane domain to be meshed.

See also: `msBuildTriConnect`, `mfNodeSearch`, `mfTriSearch`, `mfDelaunay3D`

`mfTriConnect` 2D triangular connectivity *(derived type)*

Description:

This derived type encapsulates internal arrays describing the connectivity of a 2D triangulation. It must be initialized with the `msBuildTriConnect` routine.

Declaration is made as follows:

```
type(mfTriConnect) :: tri_connect
```

Note that this structure embeds not only the connectivity tables, but also all nodes coordinates.

See `msExtractTriConnect` for more information about the content of this structure.

A variable of this type must be freed, at the end of its use, by the `msRelease` routine.

See also: `msPrintTriConnect`, `msTriMesh`

`msBuildTriConnect` 2D triangular connectivity initialization

Interface:

```
subroutine msBuildTriConnect( x, y, tri, tri_connect,                    &
                              check_tri_orient,                          &
                              tri_renum, equil_face_orient )

   type(mfArray),      intent(in)           :: x, y
   type(mfArray),      intent(in out)       :: tri
   type(mfTriConnect), intent(out)          :: tri_connect
   logical,            intent(in), optional :: check_tri_orient
   logical,            intent(in), optional :: tri_renum, equil_face_orient
```

Description:

This routine initializes the internal structures of the 2D triangulation connectivity `tri_connect`. See `mfTriConnect`.

Arguments are `mfArray`s describing (i) the coordinates of all nodes (in `x` and `y`) and (ii) the definition of the triangles in terms of nodes (in `tri`).

The indices in the `mfArray tri` describe the 2D triangulation; they may come from the `mfDelaunay` routine applied to the coordinates `x` and `y`.

The optional logical argument `check_tri_orient` is used to check that all triangles, described in the `mfArray tri`, have a direct orientation; this is required if you plan to apply, later on, the `msCheckDomainConvexity` routine. Default is to apply such a check. As this check is costly, you should omit it only if you are sure on the constraint about triangles' orientation, for example when `tri` results from the use of the `mfDelaunay` routine. When used, this check may change some indices in the `tri mfArray`.

The optional logical argument `tri_renum` is used to apply a renumbering of the triangles, in such a way that they are, wherever possible, geometrically contiguous. In this case, the table of triangles `tri` can be modified (row ordering). Default is to not apply such a renumbering.

The optional logical argument `equil_face_orient` is used to modify the orientation of triangles' faces, in such a way that they are, wherever possible, equilibrated at each node. Default is to not modify the faces' orientation.

Remarks:

- After use, the `tri_connect` object should be freed via a call to the routine `msRelease`.
- The nodes coordinates are embedded in the `tri_connect` structure. Therefore, you should NOT change the nodes coordinates after calling the current routine: indeed, while the connectivity remains the same, the convexity (which is a geometrical property embedded in the structure, see `msExtractTriConnect`) may be affected. Otherwise, especially if you DO modify the coordinates of the boundary nodes, you will have to compute again the connectivity.

See also: `mfTriSearch`, `mfNodeSearch`, `msPrintTriConnect`

`msUpdateTriConnect` 2D triangular connectivity update

Interface:

```
subroutine msUpdateTriConnect( x, y, tri_connect )

   type(mfArray),                      :: x, y
   type(mfTriConnect), intent(in out) :: tri_connect
```

Description:

Update the nodes' position in an `mfTriConnect` structure.

Arguments `x` and `y` are `mfArray`s containing the new (x, y) position of the nodes in the 2D triangulation.

See also: `msBuildTriConnect`

`msExtractTriConnect` get components from a mfTriConnect structure

Interface:

```
subroutine msExtractTriConnect( tri_connect, tri_f, face_n,                  &
                                face_tri, convexity,                         &
                                boundary_nodes, boundary_faces )

   type(mfTriConnect),             intent(in)            :: tri_connect
   integer, allocatable,           intent(out), optional :: tri_f(:,:),       &
                                                            face_n(:,:),      &
                                                            face_tri(:,:)
   integer,                        intent(out), optional :: convexity
   type(mf_Int_List), allocatable, intent(out), optional :: boundary_nodes(:)
   type(mf_Int_List), allocatable, intent(out), optional :: boundary_faces(:)
```

Description:

Usually, the 2D mesh connectivity is embedded (with hidden components) in a `mfTriConnect` structure. If the user want to know/use the detailed links between nodes, triangles and faces, he has to extract himself the corresponding connectivity tables.

The input 2D connectivity `tri_connect` must have been initialized by the `msBuildTriConnect` routine.

According to the optional argument presence, this routine returns the following integer tables, containing positive integers (when nothing else is specified):

- `tri_f`, of dimension (`nt`, 3), which contains the three faces of a given triangle; `nt` is the total number of triangles.
- `face_n`, of dimension (`nf`, 2), which contains the two nodes which define a given face; `nf` is the total number of faces.
- `face_tri`, of dimension (`nf`, 2), which contains the two triangles separated by a given face; `nf` is the total number of faces. In the case where the given face is on the boundary domain, only the first triangle number is valid; the second one is set to a value which is negative or zero (see below).
- `convexity`, an integer variable whose value may be -1 (*UNKNOWN*), 0 (*FALSE*) or 1 (*TRUE*). It is recommended to ask for its value when interpreting the value of the second column of the `face_tri` table. When the domain convexity is *UNKNOWN* or *TRUE*, the second element of the `face_tri` table is always zero for a boundary face. When the domain convexity is *FALSE*, the value zero for a boundary face indicates the convex exterior part, the value -1 indicates the non-convex part of the exterior boundary, whereas other negative values indicates an interior bondary (*i. e.* a hole; add one to this number to obtain the numbering of the hole).
  Note that the order of the interior boundaries' numbering is determined by the smallest value of the nodes for each of these internal boundaries.
  Note also that the convexity can be explicitely determined by use of the `msCheckDomainConvexity` routine.
- `boundary_nodes`, a structure of type `mf_Int_List`, containing the indices of the boundary nodes, stored in several lists, if appropriate (*i. e.*, if internal boundaries exist, in addition to the external one). This implies computing the `convexity` of the domain (see above).
- `boundary_faces`, a structure of type `mf_Int_List`, containing the indices of the boundary nodes, stored in several lists, if appropriate (*i. e.*, if internal boundaries exist, in addition to the external one). This implies computing the `convexity` of the domain (see above).

· · · / · · ·

Remarks:

- the connectivity doesn't depend on the exact location of the nodes, but the convexity does.
- the number of faces (`nf`) can be retrieve by asking the shape of the `face_n` array.
- the allocatable arrays (`tri_f`, `face_n` and others) are allocated by the routine itself; in the case where one of them is already allocated, it will be deallocated and reallocated if needed.

See also: `mfTriConnect`, `msBuildTriConnect`

`msCheckDomainConvexity` Check convexity of a 2D triangulation

Calling syntax:

```
call msCheckDomainConvexity( tri_connect )
```

Description:

Computes the convexity of a 2D triangulation, given as argument in the `tri_connect` connectivity.

After use of this routine, the `convexity` component of the connectivity may be only *TRUE* or *FALSE*, not *UNKNOWN* (see `msExtractTriConnect`).

See also: `mfTriConnect`, `msBuildTriConnect`, `msPrintTriConnect`

`msPrintTriConnect` Display of a 2D triangular connectivity

Calling syntax:

```
call msPrintTriConnect( tri_connect [, short_info ] )
```

Description:

For a small number of nodes (typically less than few dozens), this routine print on the screen the connectivity tables of a triangular mesh (the connectivity could be computed by the `msBuildTriConnect` routine, or by other ways).

The input argument `tri_connect` must be of type `mfTriConnect`.

When the boolean optional argument `short_info` is present and equal to *TRUE* then the routine just print the number of items, *i. e.* nodes, triangles and faces. Therefore, it is a simple way to inspect the structure when it is big. Another way is to extract some components of `tri_connect` by using the `msExtractTriConnect` routine.

See also: `msTriMesh`

`msTriNodeNeighbors` 2D triangulation node neighbors

Interface:

```
subroutine msTriNodeNeighbors( tri_connect [, connected_nodes | connected_faces ] )

   type(mfTriConnect), intent(in)                          :: tri_connect
   type(mf_Int_List),  intent(out), allocatable, optional :: connected_nodes(:)
   type(mf_Int_List),  intent(out), allocatable, optional :: connected_faces(:)
```

Description:

This routine computes the list of the neighbors (either nodes, stored in `connected_nodes`, or faces, stored in `connected_faces`) for each node belonging to the triangulation given in `tri_connect`.

The ouput lists, `connected_nodes` and `connected_faces` are both vectors of type `mf_Int_List`, a special structure which can contain a variable number of elements. Be aware that the allocation is done by the current routine, not by the user. The length of these vectors is, of course, equal to the number of nodes of the triangulation.

The input mesh connectivity `tri_connect` must have been built before via a call to `msBuildTriConnect`.

`mfTriSearch` 2D triangulation search (for triangle)

Generic Interface:

```
function mfTriSearch( tri_connect, x, y, strict ) result( num )

   type(mfTriConnect), intent(in) :: tri_connect

   type(mfArray), intent(in) :: x, y
or real(kind=MF_DOUBLE), intent(in) :: x, y

   logical, intent(in), optional :: strict

   integer :: num
```

Description:

Searches in a Delaunay triangulation the triangle which encloses the point `(x,y)`.

The input mesh connectivity `tri_connect` must have been built before via a call to `msBuildTriConnect`.

For a *general* search (*i. e.* when the optional argument `strict` is *FALSE*) the routine always returns a valid triangle index which is the enclosing triangle to the targeted point, or the nearest triangle if the targeted point is outside the meshed domain.

On the contrary, for a *strict* search (*i. e.* when the optional argument `strict` is *TRUE*, which is the default), if the search successes, then `num` contains the (non zero) index of the triangle; else it contains a negative or null value, which is related to concavities in the meshed domain: 0 means a targeted point outside the domain, whereas $-k$ means a point located in the hole number k (a warning is emitted).

Remark: The belonging of a point to a triangle is not strict: this means that the same point can belong to several adjacent triangles. In particular, a node of the triangulation belongs to all triangles who share it.

See also: `mfNodeSearch`

`mfNodeSearch` 2D triangulation search (for node)

Interface:

```
function mfNodeSearch( tri_connect, x, y, strict ) result( num )

   type(mfTriConnect), intent(in) :: tri_connect

   type(mfArray), intent(in) :: x, y
or real(kind=MF_DOUBLE), intent(in) :: x, y

   logical, intent(in), optional :: strict

   integer :: num
```

Description:

Searches in a Delaunay triangulation the nearest node to the point `(x,y)`.

The input mesh connectivity `tri_connect` must have been built before via a call to `msBuildTriConnect`.

For a *general* search (*i. e.* when the optional argument `strict` is *FALSE*) the routine always returns a valid node number which is the nearest node to the targeted point, even if this latter point is outside the meshed domain.

On the contrary, for a *strict* search (*i. e.* when the optional argument `strict` is *TRUE*, which is the default), if the search successes, then `num` contains the (non zero) index of the node; else it contains a negative or null value, which is related to concavities in the meshed domain: 0 means a targeted point outside the domain, whereas $-k$ means a point located in the hole number k (a warning is emitted).

See also: `mfTriSearch`

`mfVoronoi` 2D Voronoi diagram from a set of points

First calling syntax:

```
voronoi = mfVoronoi( x_in, y_in, what )
```

Description:

Builds a 2D Voronoi diagram from a list of nodes whose coordinates are given in `x_in` and `y_in` (these two `mfArrays` must be vectors having the same shape).

The returned structure `voronoi` (`mfVoronoiStruct` derived type) contains optionally the vertices and the neighbors.

More specifically, the returned structure have a content which depends on the value argument `what` (character string):

- if `what` is equal to `"vertices"`, then `voronoi` contains only the coordinates of the vertices defining each Voronoi cell, and the list of corresponding indices of these vertices for each input points;
- if `what` is equal to `"neighbors"`, then `voronoi` contains only the list of neighbors for each input points.
- if `what` is equal to `"both"`, the two previous components are stored in the `voronoi` structure.

Remark: Note that the returned structure contains also the input points, making it an autonomous structure.

See also: `msPrintVoronoi`, `mfPlotVoronoi`, `mfDelaunay`

`mfVoronoiStruct` 2D Voronoi structure emph(derived type)

Description:

This derived type encapsulates internal arrays describing a 2D Voronoi diagram. It is created by the `mfVoronoi` routine.

Declaration is made as follows:

```
type(mfVoronoiStruct) :: voronoi
```

The content of this structure, useful to the user, is:

```
type :: mfVoronoiStruct

   real(kind=MF_DOUBLE), pointer :: n_xy(:,:)

   real(kind=MF_DOUBLE), pointer :: v_xy(:,:)

   type(mf_Int_List), pointer :: vertices(:)

   type(mf_Int_List), pointer :: neighbors(:)

end type
```

`n_xy` contains the coordinates of the input points; its size is $(nn, 2)$, where nn is the number of points.

`v_xy` contains the position of the vertices; its size is $(nv, 2)$, where nv is the number of (unique) vertices.

`vertices` is a vector of lists (size nn); each list contains integer indices of the vertices, for each Voronoi cell centered on a point.

`neighbors` is a vector of lists (size nn); each list contains integer indices of the neighbors, for each Voronoi cell.

Remarks:

- It is of course not recommended to set or change yourself the components of this structure.
- A variable of this type may be freed, at the end of its use, by the `msRelease` routine.

See also: `msPrintVoronoi`, `mfPlotVoronoi`, `mf_Int_List`

`msPrintVoronoi` 2D Voronoi structure display

Calling syntax:

```
call msPrintVoronoi( voronoi )
```

Description:

For a small number of nodes (typically less than few dozens), this routine print on the screen a Voronoi diagram (this structure must be computed by the `mfVoronoi` routine).

The input argument `voronoi` must be of type `mfVoronoiStruct`.

See also: `mfVoronoiStruct`, `mfPlotVoronoi`

`mfGridData` irregular 2D interpolation

Interface:

```
function mfGridData( x, y, f, xi, yi ) result( fi )

   type(mfArray), intent(in) :: x, y, f, xi, yi
   type(mfArray) :: fi
```

Description:

Interpolates data `f`, which are defined on an irregular grid `(x,y)`, at the entries specified by `xi` and `yi`.

`x`, `y` and `f` must be vectors with the same shape.

`(xi,yi)` also must have the same shape (scalars, vectors, or matrices); `fi`, which contains interpolated data, will have the same shape as them.

`fi` is set to `MF_NAN` if `(xi,yi)` is outside the convex hull of `x` and `y`.

Remarks: a linear interpolation is used.

See also: `mfInterp2`, `mfGridData3D`

`mfDelaunay3D` Delaunay 3D tessellation (tetrahedralization)

Interface:

```
function mfDelaunay3D( x, y, z ) result( tetra )

   type(mfArray), intent(in) :: x, y, z
   type(mfArray) :: tetra
```

or

```
function mfDelaunay3D( coords ) result( tetra )

   type(mfArray), intent(in) :: coords
   type(mfArray) :: tetra
```

Description:

Builds a Delaunay 3D tetrahedralization from a list of nodes whose coordinates are given in `x`, `y` and `z` (these three `mfArray`s must be vectors having the same shape). In the second possible interface, `coords` must have three columns.

The returned `mfArray` contains indices for the tetrahedra: each row of `tetra` provides four integers which describes a tetrahedron.

All tetrahedra have a direct orientation: nodes 1, 2 and 3 is a direct triangle if node 4 is above them.

See also: `msBuildTetraConnect`, `mfDelaunay`

`msEndDelaunay3D` Internal free storage after Delaunay 3D

Calling syntax:

```
call msEndDelaunay3D( )
```

Description:

After using the `mfDelaunay3D` (and perhaps also `msBuildTetraConnect`) routine(s), it is recommended to call `msEndDelaunay3D` in order to clean the auxiliary memory used to build the tetrahedralization.

Currently, the Muesli library is not designed to handle the cleaning of intricated 3D connectivities creation. In other words, you should follow each `mfDelaunay3D` call by (if nedeed) a `msBuildTetraConnect` call, and last by a `mfDelaunay3D` call, otherwise you will be faced with some memory leaks.

`mfTetraConnect` 3D tetrahedral connectivity *(derived type)*

Description:

This derived type encapsulates internal arrays describing the connectivity of a 3D tetrahedralization. It must be initialized with the `msBuildTetraConnect` routine.

Declaration is made as follows:

```
type(mfTetraConnect) :: tetra_connect
```

A variable of this type must be freed, at the end of its use, by the `msRelease` routine.

See also: `msPrintTetraConnect`, `mfDelaunay3D`

`msBuildTetraConnect` 3D tetrahedral connectivity initialization

Interface:

```
subroutine msBuildTetraConnect( x, y, z, tetra, tetra_connect )

   type(mfArray), intent(in) :: x, y, z, tetra
   type(mfTetraConnect), intent(out) :: tetra_connect
```

or

```
subroutine msBuildTetraConnect( coords, tetra, tetra_connect )

   type(mfArray), intent(in) :: coords, tetra
   type(mfTetraConnect), intent(out) :: tetra_connect
```

Description:

This routine initializes the internal structures of a 3D tetrahedralization connectivity.

The indices in the `mfArray tetra` describe the 3D tetrahedralization; they MUST come from the `mfDelaunay3D` routine applied to the coordinates `x`, `y` and `z`.

Arguments are `mfArrays` describing (i) the coordinates of all nodes (in `x`, `y` and `z`) and (ii) the definition of the triangles in terms of nodes (in `tetra`).

The second possible interface allows the user to group all the columns vector coordinates in one matrix having a coherent shape.

After use, the `tetra_connect` object should be freed via a call to the routine `msRelease`.

Remarks: This routine may lead to an error if internal information have been erased by using `msEndDelaunay3D`. As a consequence, you should call this latter cleaning routine after getting the `mfTetraConnect` connectivity.

See also: `mfTetraSearch`, `mfNodeSearch3D`

`msExtractTetraConnect` get components from a mfTetraConnect structure

Interface:

```
subroutine msExtractTetraConnect( tetra_connect, tetra_f, face_n, face_tetra )

   type(mfTetraConnect), intent(in)            :: tetra_connect
   integer, allocatable, intent(out), optional :: tetra_f(:,:),                &
                                                  face_n(:,:),                 &
                                                  face_tetra(:,:)
```

Description:

Usually, the 3D mesh connectivity is embedded (with hidden components) in a `mfTetraConnect` structure. If the user want to know/use the detailed links between nodes, tetrahedra and faces, he has to extract himself the corresponding connectivity tables.

The input 3D connectivity `tetra_connect` must have been initialized by the `msBuildTetraConnect` routine.

According to the optional argument presence, this routine returns the following integer tables, containing positive integers (when nothing else is specified):

- `tetra_f`, of dimension (`nt`, 4), which contains the four faces of a given tetrahedra; `nt` is the total number of tetrahedra.
- `face_n`, of dimension (`nf`, 3), which contains the three nodes which define a given face; `nf` is the total number of faces.
- `face_tetra`, of dimension (`nf`, 2), which contains the two tetrahedra separated by a given face; `nf` is the total number of faces. In the case where the given face is on the boundary domain, only the first triangle number is valid; the second one is set to a value which is zero.

Remarks:

- the number of faces (`nf`) can be retrieve by asking the shape of the `face_n` array.
- the allocatable arrays (`tretra_f`, `face_n` and `face_tetra`) are allocated by the routine itself; in the case where one of them is already allocated, it will be deallocated and reallocated if needed.

See also: `mfTetraConnect`, `msBuildTetraConnect`

`msPrintTetraConnect` Display of a 3D tetrahedral connectivity

Calling syntax:

```
call msPrintTetraConnect( tetra_connect [, short_info ] )
```

Description:

For a few numbers of nodes (typically less than few dozens), this routine print on the screen the connectivity tables of a tetrahedral mesh (the connectivity could be computed by the `msBuildTetraConnect` routine, or by other ways).

The input argument `tetra_connect` must be of type `mfTetraConnect`.

When the boolean optional argument `short_info` is present and equal to *TRUE* then the routine just print the number of items, *i. e.* nodes, tetrahedra and faces. Therefore, it is a simple way to inspect the structure when it is big. Another way is to extract some components of `tetra_connect` by using the `msExtractTetraConnect` routine.

`msDel3DNodeNeighbors` 3D tetrahedralization node neighbors

Interface:

```
subroutine msDel3DNodeNeighbors( tetra, node_neighbors )

   type(mfArray),     intent(in)  :: tetra
   type(mf_Int_List), intent(out) :: node_neighbors(:)
```

Description:

This routine computes the list of the neighbors of all nodes in the allocatable array `node_neighbors` (of type `mf_Int_List`). Be aware that the allocation is done by the current routine, not by the user.

The indices in the `mfArray tetra` describe the 3D tetrahedralization; they may come from the `mfDelaunay3D` routine applied to the coordinates (`x`, `y`, `z`) of a set of nodes.

See also: `msBuildTetraConnect`

`mfTetraSearch` 3D tetrahedralization search (for tetrahedron)

Generic Interface:

```
function mfTetraSearch( tetra_connect, x, y, z ) result( num )

   type(mfTetraConnect), intent(in) :: tetra_connect

   type(mfArray), intent(in) :: x, y, z
or real(kind=MF_DOUBLE), intent(in) :: x, y, z

   integer :: num
```

Description:

Searches in a Delaunay tetrahedralization the tetrahedron which encloses the point `(x,y,z)`.

`tetra_connect` must be built before via a call to `msBuildTetraConnect`.

If the search successes, then `num` contains the (non zero) index of the tetrahedron, else it contains 0.

Remark: The belonging of a point to a tetrahedron is not strict: this means that the same point can belong to several adjacent tetrahedra. In particular, a node belongs to all tetrahedra who share it.

See also: `mfNodeSearch3D`, `msBuildTetraConnect`

`mfNodeSearch3D` 3D tetrahedralization search (for node)

Interface:

```
function mfNodeSearch3D( tetra_connect, x, y, z ) result( num )

   type(mfTetraConnect), intent(in) :: tetra_connect

   type(mfArray), intent(in) :: x, y, z
or real(kind=MF_DOUBLE), intent(in) :: x, y, z

   integer :: num
```

Description:

Searches in a Delaunay tetrahedralization the nearest node to the point `(x,y,z)`.

`tetra_connect` must be built before via a call to `msBuildTetraConnect`.

If the search successes, then `num` contains the (non zero) index of the node, else it contains 0. If `num` is 0, this means that the point is outside the convex hull (a Warning is emitted).

See also: `mfTetraSearch`, `msBuildTetraConnect`

`mfGridData3D` irregular 3D interpolation

Interface:

```
function mfGridData3D( x, y, z, f, xi, yi, zi ) result( fi )

   type(mfArray), intent(in) :: x, y, z, f, xi, yi, zi
   type(mfArray) :: fi
```

Description:

Interpolates data `f`, which are defined on an irregular 3D grid `(x,y,z)`, at the entries specified by `xi`, `yi` and `zi`.

`x`, `y`, `z` and `f` must be vectors with the same shape.

`(xi,yi,zi)` also must have the same shape (scalars, vectors, or matrices); `fi`, which contains interpolated data, will have the same shape as them.

`fi` is set to `MF_NAN` if `(xi,yi,zi)` is outside the convex hull of `x`, `y` and `z`.

Remarks: a linear interpolation is used.

See also: `mfGridData`

1.10 Optimization and Function Functions

`mf/msFZero`	single-variable non-linear zero finding
`mf/msFSolve`	multiple-variable non-linear zero finding
`mf/msLsqNonLin`	non-linear least-square minimization
`mfTrapz`	numerical evaluation of integrals by trapezoidal method
`mfCumTrapz`	cumulative integrals by trapezoidal method
`mfSimpson`	numerical evaluation of integrals by Simpson method
`mfQuad`, `msQuad`	numerical evaluation of integrals (adaptive method)
`mfDblQuad`, `msDblQuad`	numerical evaluation of double integrals (adaptive method)
`mfOdeSolve`, `msOdeSolve`	integrator of ODE (explicit) systems
`mfDaeSolve`, `msDaeSolve`	integrator of DAE (implicit) systems
`mf_DE_Options`	options for differential solvers *(derived type)*
`mf_NL_Options`	options for non-linear solvers *(derived type)*

See also:

Core Routines

File Input/Output

Data Analysis Functions

Operators

Elementary Math Functions

Specialized Math Functions

Elementary Matrix Manipulation Functions

Matrix Functions

Polynomial Functions

Sparse Matrices

`mf/msFZero` single-variable non-linear zero finding

Interface:

```
function mfFZero( fun, x0, tol ) result( out )

   interface
      real(kind=MF_DOUBLE) function fun( x )
         real(kind=MF_DOUBLE), intent(in) :: x
      end function
   end interface

   type(mfArray), intent(in)      :: x0
   real(kind=MF_DOUBLE), optional :: tol

   type(mfArray) :: out
```

Description:

Search for a solution of the non-linear equation $\mathrm{fun}(x) = 0$, via the Dekker algorithm (combination of bisection, linear and quadratic interpolation).

`fun` is a user-supplied function which has the prescribed interface above.

`x0` must be a real vector `mfArray` containing exactly two values; it defines the interval for searching x. These two values must be set such that the sign of $\mathrm{fun}(x)$ differs.

The optional argument `tol` specifies the relative tolerance for the stopping criterium. It should be a non negative and not too small real value; default value is $2\,\epsilon$.

Remarks:

- the convergence is usually fast and the Dekker method theoretically never fail, providing the initial interval given in `x0` is correct;
- `fun` has an imposed list of arguments; the user may use module's data in order to exchange other information between his program and this function. Therefore, except for very simple cases, this user's routine must be located inside a module, `USE`d also by the user's program.

The subroutine form:

```
call msFZero( mfOut(x,fval[,status]), fun, x0 [, tol ] )
```

allows the user to access to other information: `fval` is the function value associated to the zero `x`, and a `status` (optional) which has the following values:

`status`	*explanation*
0	Normal termination of the algorithm.
-1	Bad initialization: `fun`, applied to the two elements of `x0`, must have opposite sign.
-2	Bad argument value: `x0` has not enough components.

Remark: Adding the optional argument `status` allows the user to launch its program in a batch mode: indeed, errors in the current routine will not stop the program (it is the responsability of the programmer to test the value of the `status` variable before using the result). Otherwise (*i. e.* when the `status` optional argument is not used), an error will stop the program and, in debug mode, a traceback of this error will be displayed.

See also: `mfRoots`, `mfFSolve`, `mfOut`

`mfFSolve` multiple-variable non-linear zero finding

Interface:

```
function mfFSolve( fcn, n, x0, options, jac, sparse ) result( out )

   interface
      subroutine fcn( n, x, fvec, flag )
         integer,              intent(in) :: n
         real(kind=MF_DOUBLE), intent(in) :: x(n)
         real(kind=MF_DOUBLE)             :: fvec(n)
         integer                          :: flag
      end subroutine
   end interface

   type(mfArray), intent(in)      :: x0
   type(mf_NL_Options), intent(in), optional :: options
   external,                        optional :: jac
   logical,             intent(in), optional :: sparse

   type(mfArray) :: out
```

Description:

Search for a solution of `n` non-linear equations with `n` unknowns:

$$F_i(x_1, x_2, \ldots, x_n) = 0, \quad i = 1, \ldots, \mathtt{n}$$

described in the subroutine `fcn`. This user-supplied subroutine has the interface prescribed above and computes `fvec(i)` $= F_i(x_1, x_2, \ldots, x_n),\ i = 1, \ldots,$ `n`.

`flag` is actually an in-out argument of `fcn`: on entry, its value is always zero. The user can stop the iteration process, by setting it to any negative integer.

`x0` is an initial guess for the solution x.

The `mfArray options%tol` contains the tolerance(s) for the convergence:

- when it is a scalar, both the function tolerance $FTOL$ and the parameter tolerance $XTOL$ are set to the corresponding value;
- when it is a vector of two elements, the values are used to set the two tolerances: $FTOL$ and $XTOL$, respectively.

When the `mfArray options%tol` is empty (*i. e.* not set by the user), $FTOL$ and $XTOL$ are both set to 10^{-9}.

See `mf_NL_Options` for other options.

Optionally, the user may provide the computation of the jacobian via the subroutine `jac`. Its interface must be as follows:

```
   interface
      subroutine jac( n, x, jacobian )
         integer,              intent(in) :: n
         real(kind=MF_DOUBLE), intent(in) :: x(n)
         real(kind=MF_DOUBLE)             :: jacobian(n,n)
      end subroutine jac
   end interface
```

· · · / · · ·

The jacobian matrix is defined as $J = \dfrac{\partial F}{\partial x}$, therefore, the rank-2 array `jacobian` must be filled as follows: `jacobian(i,j)` $= J_{i,j} = \dfrac{\partial F_i}{\partial x_j}$. Only non-zero values of J need to be defined.

For the sparse case, the optional argument `sparse` must be set to `.true.` and the `jac` routine must be defined as:

```
interface
   subroutine jac( n, x, job, pd, ipd, jpd, nnz )
      integer,              intent(in) :: n, job
      real(kind=MF_DOUBLE), intent(in) :: x(n)
      real(kind=MF_DOUBLE)             :: pd(*)
      integer,                         :: ipd(*), jpd(*), nnz
   end subroutine
end interface
```

The `pd,ipd,jpd` f90 arrays describe the CSC (Compact Sparse Column) representation of the sparse jacobian matrix, as follows:

- `pd(1:nnz)` contains the non-zero matrix entries;
- `ipd(1:nnz)` contains the row indices;
- `jpd(1:ncol+1)` is the pointer to the beginning of the columns, in arrays `(pd,ipd)`.

`pd,ipd` must contain all diagonal terms, even if they are null. Moreover, row indices must be sorted in ascending order. Lastly, this routine must include a mechanism such that only the value of `nnz` is computed when `job=0`. You must also use the convention: `jpd(ncol+1)=nnz+1`.

Remarks:

- the problem is solved by the package *MINPACK*;
- the algorithm may fail (it is not unusual to have no solution for the non-linear problem – the returned value will be therefore a vector of *NaN*s); in such a case, using the `msFSolve` subroutine helps in giving some information about the reason of the failure.
- `fcn` and `jac` have imposed lists of arguments; as for `mfLsqNonLin`, `mfOdeSolve` and `mfDaeSolve` the user may use module's data in order to exchange other information between his program and these routines (especially for checking the number of equations).Therefore, except for very simple cases, these two user's routines must be located inside a module, `USE`d also by the user's program.

See also: `mfFZero`, `mfLsqNonLin`

`msFSolve` multiple-variable non-linear zero finding

Calling syntax:

```
call msFSolve( mfOut( x[, fvec, status ] ), fcn, n, x0 [, options, jac] )
```

Description:

Similar to `mfFSolve`, but also returns the vector function (in the `mfArray fvec`) and, optionally, a `status` which has the following values:

`status`	*explanation*
1	Normal termination of the algorithm. The stopping criterion about `xtol` has been reached first.
2	Normal termination of the algorithm. The stopping criterion about `ftol` has been reached first.
-1	Bad initialization for `x0` (size or type).
-2	Bad argument value: `tol` is not strictly positive.
-3	Bad termination: number of calls of `fcn` has reached or exceeded `max_iter*(n+1)` (if `jac` is not present) or `max_iter` (if `jac` is present).
-4	Bad termination: `tol` is too small. No further improvement in the approximate solution `x` is possible.
-5	Bad termination: iteration is not making good progress.
-10	The user has stopped the algorithm via the argument `flag` of the `fcn` subroutine.
-14	The user-supplied `fcn` routine returned invalid entries (*NaN* value).

For this subroutine, the solution is stored in the `mfArray x`.

If the `status` value is not positive (*i. e.* when algorithm fails), both `x` and `fvec` are vectors of *NaN*s.

Remark: Adding the optional argument `status` allows the user to launch his program in a batch mode: indeed, errors in the current routine will not stop the program (It is the responsability of the programmer to test the value of the `status` variable before using the result). Otherwise (*i. e.* when the `status` optional argument is not used), an error will stop the program and, in debug mode, a traceback of this error will be displayed.

See also: `mfOut`

`mfLsqNonLin` nonlinear least-square minimization

Interface:

```
function mfLsqNonLin( m, fcn, p0, n, options, jac )  result( out )

   integer,       intent(in)  :: m, n
   type(mfArray), intent(in)  :: p0

   interface
      subroutine fcn( m, n, p, fvec, flag )
         integer,              intent(in) :: m, n
         real(kind=MF_DOUBLE), intent(in) :: p(n)
         real(kind=MF_DOUBLE)             :: fvec(m)
         integer                          :: flag
      end subroutine fcn
   end interface

   type(mf_NL_Options), optional :: options
   external,            optional :: jac

   type(mfArray) :: out
```

Description:

Minimize the sum of the squares of `m` nonlinear functions F_i depending on `n` parameters.

`fcn` is the user-supplied subroutine which calculates the functions; it has the interface prescribed above and computes `fvec(i)` $= F_i(p)$, $i = 1, \ldots,$ `m`, where p represents the vector of the `n` parameters.

`flag` is actually an in-out argument of `fcn`: on entry, its value is always zero. The user can stop the iteration process, by setting it to any negative integer. See just below for constrained minimization.

Constraints of type *BOX* are possible, for each parameter(see `mf_NL_Options`).

At input, `p0` contains an initial guess for the values of the `n` unknown parameters. Final values are stored in the `mfArray out`.

All the following options are stored in the structure `options` (see `mf_NL_Options`).

The `mfArray options%tol` contains the tolerance(s) for the convergence:

- when it is a scalar, both the function tolerance *FTOL* and the parameter tolerance *XTOL* are set to the corresponding value, while the orthogonality tolerance *GTOL* (it measures the orthogonality between the function vector and the columns of the jacobian) is set to zero;
- when it is a vector of three elements, the values are used to set the three tolerances: *FTOL*, *XTOL* and *GTOL*, respectively.

When the `mfArray options%tol` is empty (*i. e.* not set by the user), *FTOL* and *XTOL* are both set to 10^{-6} while *GTOL* is set to zero.

`options%max_iter` is the maximum number of iterations. Its default value is 50.

$\cdots/\ldots$

`options%epsfcn` is the approximation error of the functions F_i. Its default value is zero. It is used only to compute the step for the finite differences computation of the Jacobian; this argument will not be referenced if `jac` is present. In the case of an approximate computation in the `fcn` routine (*e. g.* an integration or whatelse), the value of `epsfcn` may be critical to obtain good convergence; it must be chosen carefully; see `mf_NL_Options` for additional information.

The logical `options%print` can be used to print some information during the iterative process. Default is *FALSE*.

Optionally, the user may provide the computation of the jacobian via the subroutine `jac`. Its interface must be as follows:

```
interface
   subroutine jac( m, n, p, jacobian )
      integer,               intent(in) :: m, n
      real(kind=MF_DOUBLE), intent(in) :: p(n)
      real(kind=MF_DOUBLE)             :: jacobian(m,n)
   end subroutine jac
end interface
```

The jacobian matrix is defined as $J = \dfrac{\partial F}{\partial p}$, and the rank-2 array `jacobian` must be filled as follows: `jacobian(i,j)` $= J_{i,j} = \dfrac{\partial F_i}{\partial p_j}$. Only non-zero values of J need to be defined.

The integer `options%check_jac` allows the user to numerically check (via finite differences) the components of the jacobian each time it is called. A quick global check is done by setting this argument to 1 while a complete (but more expensive) check is done by setting it to 2. Default is 0, *i. e.* no check.

Lastly, the integer `print_check_jac` allows the printing (either on the screen if the value is 1, or in files if the value is 2) of the results of the previous jacobian check. Default is 0, *i. e.* no check.

Remarks:

- the problem is solved by the LMA method (Levenberg-Marquardt Algorithm) of the *MINPACK* package;
- `m` must be bigger than the number of parameters `n`.
- `fcn` and `jac` have imposed lists of arguments; as for `mfFSolve`, `mfOdeSolve` and `mfDaeSolve` the user may use module's data in order to exchange other information between his program and these routines (especially for checking the number of equations). Therefore, except for very simple cases, these two user's routines must be located inside a module, `USE`d also by the user's program.
- the initial step bound for the parameters' vector is based on a factor whose default value is 100. Sometimes, we may want to change this factor to a smaller value, especially if you use the $log()$ function to ensure the positiveness of your unknowns; in such a case, factor $= log(100)$ is more appropriate, by setting `options%init_step_bound_factor` to this value.

See also: `mfFunFit`, `msLsqNonLin`

`msLsqNonLin` nonlinear least-square minimization

Calling syntax:

```
call msLsqNonLin( mfOut( p, resnorm[, status, res_log, p_log, ident ] ),     &
                  m, fcn, p0, n[, options, jac ] )
```

Description:

Similar to `mfLsqNonLin`, but also returns the *2-norm* residue (in the `mfArray resnorm`) and, optionally, a `status` which has the following values:

`status`	*explanation*
1	Normal termination of the algorithm. Both actual and predicted relative reductions in the sum of squares of the functions `fvec` are at most `ftol`.
2	Normal termination of the algorithm. Relative error between two consecutive iterates of the parameters' vector is at most `xtol`.
3	Normal termination of the algorithm. Conditions for status = 1 and status = 2 both hold.
-1	Improper input parameters.
-4	`fvec` is orthogonal to the columns of the jacobian. This means that the gradient is approximatively null (with respect to `gtol`): it may correspond to a minimum, or a maximum, or a saddle point, so the algorithm cannot continue.
-5	number of calls of `fcn` has reached `max_iter*(n+1)` (if `jac` is not present) or `max_iter` (if `jac` is present).
-6	`tol` is too small. No further reduction in the sum of squares is possible.
-7	`tol` is too small. No further improvement in the approximate solution `p` is possible.
-8	`fvec` is orthogonal to the columns of the jacobian to machine precision. This means that the gradient is approximatively null: it may correspond to a minimum, or a maximum, or a saddle point, so the algorithm cannot continue. Incidently, `gtol` is too small.
-9	The algorithm has been stopped because a constraints' interval for one of the unknown parameters is too small, which makes the finite-differences approximation of the jacobian impossible.
-10	The algorithm has been stopped via the argument `flag` of the `fcn` subroutine.
-14	The user-supplied `fcn` routine returned invalid entries (*NaN* value).

For this subroutine, the output of the desired `n` variables is stored in the `mfArray p`.

`res_log` and `p_log` are mfArrays which contain the history of the norm of the residue and the parameters, along the iterations. Each row of these matrices corresponds to an iteration.

`ident` is an mfArray which contains the identifiability of each parameter. This vector has values ranged between 0 and 1, and computed internally by the ratio $|R_i| \,/\, |R_1|$, where $R_i, i = 1 \ldots$ `n`, are the diagonal values of the R matrix obtained after a $Q\,R$ decomposition of the jacobian (sensitivity matrix). The elements of `ident` have been reordered to match the original order of the parameters. A value equal or close to 0 means that the corresponding parameter is not identifiable. Actually, the user should compare the values of `ident` to the numerical precision of the `fvec` functions.

$\cdots/\ldots$

Remark: Adding the optional argument `status` allows the user to launch his program in a batch mode: indeed, errors in the current routine will not stop the program (It is the responsability of the programmer to test the value of the `status` variable before using the result). Otherwise (*i. e.* when the `status` optional argument is not used), an error will stop the program and, in debug mode, a traceback of this error will be displayed.

See also: `mfOut`

`mfTrapz` integration by trapezoidal method

First calling syntax:

```
res = mfTrapz( y )
```

Description:

Performs the numerical integration of the data `y`, assuming constant unit spacing for the abscissas, using the trapezoidal rule. If the spacing is different from unity, the result `res` must be multiplied by the effective spacing.

Second calling syntax:

```
res = mfTrapz( x, y )
```

Description:

Performs the numerical integration of the data `y`, spread against the abscissas `x`.

Remark:

- `x` and `y` must be real vector `mfArrays`.
- the method used is of order 2, *i. e.* the resulting error (when the exact result is known or can be estimated) should be divided by 4 when the number of values in `x` and `y` is doubled. If this behavior is not reached, this may indicate that the integral is not convergent.

See also: `mfSimpson`, `mfQuad`, `mfCumTrapz`

`mfCumTrapz` cumulative integration by trapezoidal method

First calling syntax:

```
res = mfCumTrapz( y )
```

Description:

Performs the numerical cumulative integration of the data `y`, assuming constant unit spacing for the abscissas, using the trapezoidal rule. If the spacing is different from unity, the result `res` must be multiplied by the effective spacing.

Second calling syntax:

```
res = mfCumTrapz( x, y )
```

Description:

Performs the numerical cumulative integration of the data `y`, spread against the abscissas `x`.

Remark:

- `x` and `y` must be real vector `mfArrays`.
- the method used is of order 2, *i. e.* the resulting error (when the exact result is known or can be estimated) should be divided by 4 when the number of values in `x` and `y` is doubled. If this behavior is not reached, this may indicate that the integral is not convergent, at least at some abscissas.

See also: `mfTrapz`

`mfSimpson` integration by Simpson method

Calling syntax:

```
res = mfSimpson( y )
```

Description:

Performs the numerical integration of the data `y`, assuming constant unit spacing for the abscissas, using the Simpson rule. If the spacing is different from unity, the result `res` must be multiplied by the effective spacing.

Second calling syntax:

```
res = mfSimpson( x, y )
```

Description:

Performs the numerical integration of the data `y`, spread against the abscissas `x`.

Remarks:

- `x` and `y` must be real vector `mfArrays`.
- the number of points must be odd.
- the method used is of order 4, *i. e.* the resulting error (when the exact result is known or can be estimated) should be divided by 16 when the number of values in `x` and `y` is doubled. If this behavior is not reached, this may indicate that the integral is not convergent.

See also: `mfTrapz`, `mfQuad`

`mfQuad` numerical evaluation of integrals

Interface:

```
function mfQuad( fun, a, b, abs_tol, rel_tol, sing ) result( out )

   interface
      real(kind=MF_DOUBLE) function fun( x )
         real(kind=MF_DOUBLE), intent(in) :: x
      end function
   end interface

   real(kind=MF_DOUBLE), intent(in)           :: a, b
   real(kind=MF_DOUBLE), intent(in), optional :: abs_tol, rel_tol
   logical,              intent(in), optional :: sing
   type(mfArray) :: out
```

Description:

Performs the numerical computation of the integral of the function `fun`(x) over the interval [`a`, `b`].

The optional arguments `abs_tol` and `rel_tol` allows the user to give the absolute and/or the relative precision of the result. Default tolerance is 1×10^{-12}.

The second optional argument `sing` should be used when the function `fun`(x) has a singularity at $x =$`a` or/and at $x =$`b`. '`sing=.true.`' leads to use another quadrature method, which usually involves a fewer number of function evaluation. This optional argument cannot be used when the interval range is infinite.

Remarks:

- `out` is set to `MF_NAN` if required precision cannot be achieved; an error is emitted by the library. Call the subroutine version if you don't want such a behavior.
- `a` and/or `b` may be infinite values. In such a case, an appropriate routine is called; a correct result is expected as long as the integral is convergent.

See also: `msQuad`, `mfTrapz`, `mfDblQuad`, `msDblQuad`

`msQuad` numerical evaluation of integrals

Calling syntax:

```
call msQuad( mfOut( q, abserr[, status, neval] ),                    &
             fun, a, b [, abs_tol, rel_tol, sing] )
```

Description:

The behavior of this routine is similar to those of `mfQuad`.

The subroutine call allows to get more information returned by the integrator:

- `q` contains the numerical result
- `abserr` contains an estimation of the absolute error
- `status` (optional) contains the returned error code:

  `status` = 0: normal and reliable termination of the routine. It is assumed that the requested accuracy has been achieved.

  `status` < 0: abnormal termination of the routine. The estimates for `q` and `abserr` are less reliable. It is assumed that the requested accuracy has not been achieved. Only a warning is emitted by the library. (Exact meaning of the error can be found `below`)
- `neval` (optional) contains the effective number of function evaluation

All these variable must be of `type(mfArray)`.

Remark: Adding the optional argument `status` allows the user to launch its program in a batch mode: indeed, errors in the current routine will not stop the program (It is the responsability of the programmer to test the value of the `status` variable before using the result). Otherwise (*i. e.* when the `status` optional argument is not used), an error will stop the program and, in debug mode, a traceback of this error will be displayed.

See also: `mfQuad`, `mfOut`, `mfDblQuad`, `msDblQuad`

`status`	*explanation*
−1	Maximum number of subdivisions allowed has been achieved.
−2	The occurrence of roundoff errors is detected, which prevents the requested tolerance from being achieved. The error may be under-estimated.
−3	Extremely bad integrand behaviour occurs at some points of the integration interval.
−4	The algorithm does not converge. Roundoff error is detected in the extrapolation table. It is presumed that the requested tolerance cannot be achieved, and that the returned result is the best which can be obtained.
−5	The integral is probably divergent, or slowly convergent. It must be noted that divergence can occur with any other value of `status`.
−6	Invalid input (this error should have been trapped by the routine `msQuad` itself).

`mfDblQuad` numerical evaluation of double integrals

Interface:

```
function mfDblQuad( fun, xa, xb, ya, yb, abs_tol, rel_tol ) result( out )

   interface
      real(kind=MF_DOUBLE) function fun( x, y )
         real(kind=MF_DOUBLE), intent(in) :: x, y
      end function
   end interface

   real(kind=MF_DOUBLE), intent(in)           :: xa, xb, ya, yb
   real(kind=MF_DOUBLE), intent(in), optional :: abs_tol, rel_tol
   type(mfArray) :: out
```

Other calling syntaxes:

```
out = mfDblQuad( fun, fun_xa, fun_xb, ya, yb, abs_tol, rel_tol )
```

or

```
out = mfDblQuad( fun, xa, xb, fun_ya, fun_yb, fun_y=.true., abs_tol, rel_tol )
```

where `fun_xa`, `fun_xb`, `fun_ya` and `fun_ya` are all double precision real function of one real argument. By using these latter syntaxes, the user is able to integrate over non rectangular domains. Note that in the second case, the additional argument `fun_y` is mandatory (no other way to disambiguate a unique interface for differente routines).

Description:

Performs the numerical computation of the double integral of the function `fun`(x, y) over the interval [`xa`, `xb`]×[`ya`, `yb`].

The optional arguments `abs_tol` and `rel_tol` allows the user to give the absolute and/or the relative precision of the result. Default tolerance is 1×10^{-12}.

Remarks: `out` is set to `MF_NAN` if required precision cannot be achieved; an error is emitted by the library. Call the subroutine version if you don't want such a behavior.

See also: `mfQuad`, `msQuad`

`msDblQuad` numerical evaluation of double integrals

Calling syntax:

```
call msDblQuad( mfOut( q, abserr[, status, neval] ),                    &
                fun, xa, xb, ya, yb [, abs_tol, rel_tol] )
```

Description:

The behavior of this routine is similar to those of `mfDblQuad` (including the fact to do integration on non rectangular domains).

The subroutine call allows to get more information returned by the integrator:

- `q` contains the numerical result
- `abserr` contains an estimation of the absolute error
- `status` (optional) contains the returned error code:

  `status` = 0: normal and reliable termination of the routine. It is assumed that the requested accuracy has been achieved.

  `status` < 0: abnormal termination of the routine. The estimates for `q` and `abserr` are less reliable. It is assumed that the requested accuracy has not been achieved. Only a warning is emitted by the library. (Exact meaning of the error can be found `below`)
- `neval` (optional) contains the effective number of function evaluation

All these variable must be of `type(mfArray)`.

Remark: Adding the optional argument `status` allows the user to launch its program in a batch mode: indeed, errors in the current routine will not stop the program (It is the responsability of the programmer to test the value of the `status` variable before using the result). Otherwise (*i. e.* when the `status` optional argument is not used), an error will stop the program and, in debug mode, a traceback of this error will be displayed.

See also: `mfDblQuad`, `mfOut`

`status`	*explanation*
−1	Maximum number of subdivisions allowed has been achieved.
−2	The occurrence of roundoff errors is detected, which prevents the requested tolerance from being achieved. The error may be under-estimated.
−3	Extremely bad integrand behaviour occurs at some points of the integration interval.
−4	The algorithm does not converge. Roundoff error is detected in the extrapolation table. It is presumed that the requested tolerance cannot be achieved, and that the returned result is the best which can be obtained.
−5	The integral is probably divergent, or slowly convergent. It must be noted that divergence can occur with any other value of `status`.
−6	Invalid input (this error should have been trapped by the routine `msDblQuad` itself).

`mfOdeSolve` integrator of ODE (explicit) systems

Main interface:

```
function mfOdeSolve( deriv, t_span, y_0, options, jac, sparse ) result( y )

   interface
      subroutine deriv( t, y, yprime, flag )
         real(kind=MF_DOUBLE), intent(in)     :: t, y(*)
         real(kind=MF_DOUBLE), intent(out)    :: yprime(*)
         integer,              intent(in out) :: flag
      end subroutine
   end interface

   type(mfArray),       intent(in)           :: t_span, y_0
   type(mf_DE_Options), intent(in), optional :: options
   external,                        optional :: jac
   logical,             intent(in), optional :: sparse

   type(mfArray) :: y
```

Description:

Performs the numerical integration of a system of (explicit[1]) ordinary differential equations (ODEs) of first order, which is written under the form:

$$y' = F(t, y)$$

described via the subroutine `deriv` (the variables `t`, `y` and `yprime` represent respectively t, y and y'). Usually, the integration is done between time t_0 and t_{end} (extremal values of `t_span`, which is a column vector `mfArray` of at least two components) or, in the continuation mode[2], up to the new values specified in `t_span` (in this case, `t_span` may contain only one value).

The output of `mfOdeSolve` is the row vector `y` which represents the solution y at time t_{end}. If the vector `t_span` contains more than two values (or more than one value in the continuation mode), the routine returns in `y` the intermediate values of y, as specified by all time values present in `t_span`; in this case, `y` is a rank-2 `mfArray` with as many rows as components in `t_span`.

The integration uses the prescribed initial condition stored in the row vector `y_0`. Note that this `mfArray` is not referenced in the continuation mode.

The last argument `flag` of `deriv` is always set to zero (internally by the solver) on input, and should be altered (inside the routine `deriv`) only in some special situations:

- set `flag` to -1 for an *Illegal Condition*; the value of y' at this time may not be available. The routine will try to continue the integration (reducing the time step) as long as possible without getting the illegal condition. [You can put this assignment anywhere in the `deriv` routine.]
- set `flag` to -2 for an *Emergency Exit*; the value of y' at this time may not be available. The routine will return control immediately to the calling program. [You can put this assignment anywhere in the `deriv` routine.]
- set `flag` to 3 for an *End Condition*; the value of y' at this time must be always available. The routine will adapt its time step to finish exactly at the current time given by the argument `t` of `deriv`. [WARNING: you **must put this assignment** at the end of the `deriv` routine, to insure that all the derivatives have been computed.]

· · · / · · ·

[1] For implicit systems, the `mfDaeSolve` routine should be used instead.
[2] Currently, only the *BDF* integrator has this feature.

The previous three situations are called *stopping conditions*. Note that when a *stopping condition* is found in the user subroutine `resid`, then the vector `t_span` (which contains the value of t_{end}) is modified. Note also that the three *stopping conditions* doesn't have the same numerical cost; their cost is mentioned in the following table:

Flag to used in `resid`	**Stopping condition**	**Numerical cost**
-1	*Illegal Condition*	expensive
-2	*Emergency Exit*	very cheap
3	*End Condition*	cheap

All following options are stored in the structure `options` (see `mf_DE_Options`).

The boolean `mfArray options%continuation` allows the user to continue the integration to a new value of t_{end}, without specify a new t_0. Thanks to specific calls to `mf/msOdeSolve`, the user is even able to save internal data of the integrator, to stop the program, and to launch again the integrator after the reload of saved data. See the next section *Other calling syntaxes*.

The `mfArray options%tol` allows the user to give the precision of the result, and must contain strictly positive values. Default values are 10^{-3} for the relative tolerance and 10^{-6} for the absolute tolerance; this `mfArray` may be of rank 0, 1 or 2:

- if it is a scalar, it specifies the relative tolerance for all the components of the solution; the absolute tolerance is fixed to 10^{-6}.
- if it is a vector, it must have two elements which specifies the relative and the absolute tolerance, respectively.
- if it is a matrix, it must have two columns (corresponding to the relative and absolute tolerance) and a number of rows equal to the number of equations.

The `mfArray options%y_ind_out` allows the user to get only a subpart of the `mfArray y`; if present, it must contain integer indices of the wanted components of the `y` vector.

The string `options%method` specifies the method to be used. Possible choices are `"RKF"` (*Runge-Kutta Fehlberg*), `"ABM"` (*Adams-Bashforth-Moulton*), or `"BDF"` (*Backward Differentiation Formula*). By default, the *Runge-Kutta Fehlberg* method is selected.

When the `"BDF"` method is used, the user may provide the `jac` routine to compute the jacobian matrix (otherwise, it will be numerically evaluated by finite differences). Moreover, this jacobian matrix may be defined in dense, band or sparse format. For the first two formats, the interface of the `jac` routine is:

```
interface
   subroutine jac( t, y, jacobian, ldjac )
      real(kind=MF_DOUBLE), intent(in)  :: t, y(*)
      integer,              intent(in)  :: ldjac
      real(kind=MF_DOUBLE), intent(out) :: jacobian(ldjac,*)
   end subroutine
end interface
```

The jacobian matrix is defined as $J = \dfrac{\partial F}{\partial y}$, therefore, the rank-2 array `jacobian` must be filled as follows: `jacobian(i,j)` $= J_{i,j} = \dfrac{\partial F_i}{\partial y_j}$.

Please note that, for the dense format, only non-zero values need to be defined.

If the jacobian is symmetric and positive definite (abbreviated as SPD), you should set the `jac_symm_pos_def` option (see `mf_DE_Options`) in order to economize storage and obtain a better performance for its factorization; in such a case, you must give only the upper part of the jacobian.

$\cdots/\ldots$

Furthermore, when the jacobian J is banded, the `mfArray options%band` may be used to economize both memory and CPU time. It must be a vector of exactly two elements: ML and MU which are the widths of the lower and upper parts of the band, respectively, with the main diagonal being excluded (*e. g.* for a tridiagonal system, both elements are equal to 1).

In this case, routine `jac` must store the elements $J_{i,j}$ in the matrix `jacobian` as follows:

```
irow = i - j + ML + MU + 1
jacobian(irow,j) =  J_{i,j}
```

Actually, `ldjac` is equal to `2*ML+MU+1`: indeed, an extra storage is needed for the matrix factorisation. For an SPD matrix, simply take `ML` equal to zero in the previous formulae.

For the sparse case, the optional argument `sparse` must be set to `.true.` and the `jac` routine must be defined as:

```
interface
   subroutine jac( t, y, nrow, job, pd, ipd, jpd, nnz )
      real(kind=MF_DOUBLE), intent(in)     :: t, y(*)
      integer,              intent(in)     :: nrow, job
      real(kind=MF_DOUBLE), intent(out)    :: pd(*)
      integer,              intent(out)    :: ipd(*), jpd(*)
      integer,              intent(in out) :: nnz
   end subroutine
end interface
```

The `pd,ipd,jpd` f90 arrays describe the CSC (Compact Sparse Column) representation of the sparse jacobian matrix, as follows:

- `pd(1:nnz)` contains the non-zero matrix entries;
- `ipd(1:nnz)` contains the row indices;
- `jpd(1:ncol+1)` is the pointer to the beginning of the columns, in arrays `(pd,ipd)`.

`pd,ipd` must contain all diagonal terms, even if they are null. Moreover, row indices must be sorted in ascending order. Lastly, this routine must include a mechanism such that only the value of `nnz` is computed when `job=0`. You must also use the convention: `jpd(ncol+1)=nnz+1`.

Be aware that the sparse structure (that is to say, the indices' vectors `ipd` and `jpd`) must remain the same during all the integration process. Otherwise, use the option `spjac_const_struct` of the `mf_DE_Options` derived type. Moreover, you can even force to reuse the same sparse structure if you make many calls to `mf/msOdeSolve` during, *e. g.*, an iterative procedure: see the option `reuse_spjac_struct` of the `mf_DE_Options` derived type.

Remarks:

- on output, `y` may contain *NaN* values for some reasons. In such a case, use the `msOdeSolve` routine to further investigate the problem.
- `deriv` and `jac` have imposed lists of arguments; the user may use module's data in order to exchange other information between his program and these routines (especially for checking the number of equations). Therefore, except for very simple cases, these two user's routines must be located inside a module, `USE`d also by the user's program.
- for debugging purposes, the user may inspect the values of y and y' during the integration process. See explanations about the fields `monitor_y_ind`, `monitor_yp_ind` and `monitor_pause` of the `mf_DE_Options` derived type.
- the *stopping conditions* are not (yet) available for the `"ABM"` method.

· · · / . . .

Other calling syntaxes

```
call msOdeSolve( A, "save" [, jac_sparse] )
```

By using this call, the user is able to save all internal data of the integrator in the `mfArray A`, and then continue the integration later on. The optional argument `jac_sparse`, which is the name of the routine computing the sparse Jacobian matrix, is required when the Jacobian has a sparse structure.

```
call msOdeSolve( A, "restore" [, jac_sparse] )
```

This call is required to restore all internal data of the integrator, before a new integration sequence. The optional argument `jac_sparse`, which is the name of the routine computing the sparse Jacobian matrix, is required when the Jacobian has a sparse structure.

```
call msOdeSolve( "finalyze" )
```

This call is useful to avoid memory leaks at the end of the integration process.

See also: `mfDaeSolve`

msOdeSolve — integrator of ODE (explicit) systems

Calling syntax:

```
call msOdeSolve( mfOut( y, status[, tolout, yp, solve_log, t_log, order_log] ),   &
                 deriv, t_span, y_0[, options, jac, sparse] )
```

Description:

The behavior of this routine is similar to those of mfOdeSolve, but it allows to get more information returned by the integrator (via mfArrays which must be enclosed to the mfOut function):

- y contains the numerical result.
- status contains a returned error code:

  status > 0: normal and reliable termination of the routine. It is assumed that the requested accuracy has been achieved. (Exact meaning of the status can be found below)

  status < 0: abnormal termination of the routine. The estimates for y is less reliable. It is assumed that the requested accuracy has not been achieved. Moreover, in some circumstances, a message is printed; in such cases, the user cannot use numerical values in y and yp and therefore an appropriate decision must be taken. (Exact meaning of the error can be found below)
- tolout is different from tol only if status $= -2$: in such a case, it contains an appropriate value for continuing the integration.
- yp contains an estimation of the derivative at time t_{end} (or at all intermediate values defined in t_span); moreover, the role of the option options%y_ind_out applies also to yp.
- solve_log contains additional information about the integration process. It is a vector of 4 or 6 elements, according to the method used. These elements are:

  nb_step is the total number of steps used

  dt_min the smallest step used

  dt_max the largest step used

  nb_deriv is the number of calls of the deriv routine

  nb_jac is the number of evaluation of the jacobian matrix (only for the "BDF" method)

  nb_solve is the number of times the linear system is solved using the jacobian matrix (only for the "BDF" method)
- t_log contains the whole history of the times used during the integration. This can be useful to detect where a critical behavior is located.
- order_log contains the whole history of the order used during the integration. The first value, not defined, is set to NaN (Not-a-Number special IEEE floating-point value); as a consequence, this mfArray has the same size as t_log.

Among these returned variables, only the first two are required, while the others are optional. Moreover, it is forbidden to use the same mfArray in place of two or more actual arguments, even empty.

Remarks:

- The presence of the argument status allows the user to launch its program in a batch mode: indeed, errors in the current routine will not stop the program (it is the responsability of the programmer to test the value of the status variable before using the result). If the mfOdeSolve routine was used instead, an error would stop the program and, in debug mode, a traceback of this error would be displayed.

· · · / . . .

– The special calling syntaxes used to save/restore the internal data (in case of continuation in overlay situations), or to finalize the integrator, have been presented at the end of the previous entry.

`status`	*explanation*
0	Normal termination of the solver.
4	Normal termination of the solver, but a large amount of work has been expended.
5	Normal termination of the solver, but the *RKF* integrator has been used very inefficiently because the natural step size has been restricted by too frequent output.
12	Normal termination of the solver, but an *End Condition* was found (`flag`, last argument in `deriv`, was equal to 3 and the solver has been called to finish at the new current point).
−2	The error tolerances are too stringent.
−3	The local error test cannot be satisfied because you specified a zero value for the absolute error in `tol` and the corresponding computed solution component is zero. Thus, a pure relative error test is impossible for this component.
−4	The problem appears to be stiff (*RKF* and *ABM* integrators only).
−6	The *BDF* integrator had repeated convergence test failures on the last attempted step.
−7	The *BDF* integrator had repeated error test failures on the last attempted step.
−10	Abnormal termination of the solver, due to an *Illegal Condition* (`flag`, last argument in `deriv`, was equal to −1 and the solver did its best to not obtain −1).
−11	Abnormal termination of the solver, due to an *Emergency exit* (`flag`, last argument in `deriv`, was equal to −2 and control has been returned to the calling program).
−14	The user-supplied `deriv` routine returned invalid entries (*NaN* value).
−20	Allocation error because the maximum of available memory has been reached. An explanation is printed.
−33	Invalid argument.
−34	Invalid implementation of the user-defined sparse jacobian.
−40	On calling, the number of output argument(s) is incorrect.
−100	The routine wasn't able to start, nor choose an initial timestep (`flag`, last argument in `deriv`, was different from 0).

See also: `mfOdeSolve`, `msDaeSolve`, `mfOut`, `mf_DE_Options`

`mfDaeSolve` integrator of DAE (implicit) systems

Interface:

```
function mfDaeSolve( resid, t_span, y_0, yp_0, options, jac, sparse ) result( y )

   interface
      subroutine resid( t, y, yprime, delta, flag )
         real(kind=MF_DOUBLE), intent(in)     :: t, y(*), yprime(*)
         real(kind=MF_DOUBLE), intent(out)    :: delta(*)
         integer,              intent(in out) :: flag
      end subroutine
   end interface

   type(mfArray),       intent(in out)           :: t_span
   type(mfArray),       intent(in)               :: y_0, yp_0
   type(mf_DE_options), intent(in),     optional :: options
   external,                            optional :: jac
   logical,             intent(in),     optional :: sparse

   type(mfArray) :: y
```

Description:

Performs the numerical integration of a system of (fully implicit [3]) differential algebraic equations (DAEs) of first order (and differential index 1 only[4]), which can be written in the general form:

$$R(t, y, y') = 0$$

described via the user subroutine `resid`, which must define the components $R_i(t, y, y')$ and store the values in the `delta` array (the variables `t`, `y` and `yprime` represent respectively t, y and y'). Usually, the integration is done between time t_0 and t_{end} (extremal values of `t_span`, which is a column vector `mfArray` of at least two components) or, in the continuation mode, up to the new values specified in `t_span` (in this case, `t_span` may contain only one value).

The output of `mfDaeSolve` is the row vector `y` which represents the solution y at time t_{end}. If the vector `t_span` contains more than two values, the routine returns in `y` the intermediate values of y, as specified by all time values present in `t_span`; in this case, `y` is a rank-2 `mfArray` with as many rows as components in `t_span`.

The integration uses the prescribed initial conditions stored in the row vectors `y_0` and `yp_0`. By default, it is assumed that the data `y_0` and `yp_0` are consistent (note that a consistency test is performed; if it fails, an error is returned), *i. e.* verify the equation

$$R(t_0, y_0, y'_0) = 0;$$

otherwise, the user must set the `options%IC_known` argument to `.false.`, so that the initial conditions will be computed by the solver, taking `y_0` and `yp_0` as initial guesses; `yp_0` may be empty but `y_0` must always be allocated and must have its size equal to the number of equations (see `mf_DE_Options` for more details; in particular, the user must indicate which components of these two vectors should be held fixed).

· · · / · · ·

[3] Not just linearly implicit, which would have a mass matrix. For explicit systems, the `mfOdeSolve` routine should be used instead.

[4] For higher-index problems, the current algorithm may or may not work!

The last argument `flag` of `resid` is always set to zero (internally by the solver) on input, and should be altered (inside the routine `resid`) only in some special situations:

- set `flag` to -1 for an *Illegal Condition*; the value of y' at this time may not be available. The routine will try to continue the integration (reducing the time step) as long as possible without getting the illegal condition. [You can put this assignment anywhere in the `resid` routine.]
- set `flag` to -2 for an *Emergency Exit*; the value of y' at this time may not be available. The routine will return control immediately to the calling program. [You can put this assignment anywhere in the `resid` routine.]
- set `flag` to 3 for an *End Condition*; the value of y' at this time must be always available. The routine will adapt its time step to finish exactly at the current time given by the argument `t` of `resid`. [WARNING: you **must put this assignment** at the end of the `resid` routine, to insure that all the residues `delta` have been computed.]

The previous three situations are called *stopping conditions*. Note that when a *stopping condition* is found in the user subroutine `resid`, then the vector `t_span` (which contains the value of t_{end}) is modified. Note also that the three *stopping conditions* doesn't have the same numerical cost; their cost is mentioned in the following table:

Flag to used in `resid`	**Stopping condition**	**Numerical cost**
-1	*Illegal Condition*	expensive
-2	*Emergency Exit*	very cheap
3	*End Condition*	cheap

All following options are stored in the structure `options` (see `mf_DE_Options`).

The boolean `mfArray options%continuation` allows the user to continue the integration to a new value of t_{end}, without specify a new t_0. Thanks to specific calls to `mf/msDaeSolve`, the user is even able to save internal data of the integrator, to stop the program, and to launch again the integrator after the reload of saved data. See the next section *Other calling syntaxes*.

The `mfArray options%tol` allows the user to give the precision of the result, and must contain non negative values. Default values are 10^{-3} for the relative tolerance and 10^{-6} for the absolute tolerance; this `mfArray` may be of rank 0, 1 or 2:

- if it is a scalar, it specifies the relative tolerance for all the component of the solution; the absolute tolerance is fixed to 10^{-6}.
- if it is a vector, it must have two elements which specifies the relative and the absolute tolerance, respectively.
- if it is a matrix, it must have two columns (corresponding to the relative and absolute tolerance) and a number of rows equal to the number of equations.

The `mfArray options%non_neg` allows the user to apply non-negativity constraints to a subset (or all the set) of the components of the solution `y`. It must be a vector describing the indices of such constraints.

The `mfArray options%y_ind_out` allows the user to get only a subpart of the `mfArray y`; if present, it must contain integer indices of the wanted components of the `y` vector.

The user may provide the `jac` routine to compute the jacobian matrix (otherwise, it will be numerically evaluated by finite differences). This matrix may be defined in dense, band or sparse format. For the first two formats, the interface of the `jac` routine is:

· · · / . . .

```
interface
   subroutine jac( t, y, yprime, jacobian, cj, ldjac )
      real(kind=MF_DOUBLE), intent(in)  :: t, y(*), yprime(*), cj
      integer,              intent(in)  :: ldjac
      real(kind=MF_DOUBLE), intent(out) :: jacobian(ldjac,*)
   end subroutine
end interface
```

The generalized jacobian[5] $\widetilde{J}$ must be defined as:

$$\widetilde{J} = \frac{\partial R}{\partial y} + c_J \frac{\partial R}{\partial y'}$$

where c_J is a scalar computed by the solver; therefore the numerical values of $\widetilde{J}$ must be stored in the `jacobian` rank-2 array as follows: `jacobian(i,j)` $= \widetilde{J}_{i,j} = \frac{\partial R_i}{\partial y_j} + c_J \frac{\partial R_i}{\partial y'_j}$.

Please note that, for the dense format, only non-zero values of $\widetilde{J}$ need to be defined.

If the jacobian is symmetric and positive definite (abbreviated as SPD), you should set the `jac_symm_pos_def` option (see `mf_DE_Options`) in order to economize storage and obtain a better performance for its factorization; in such a case, you must give only the upper part of the jacobian.

Furthermore, the jacobian $\widetilde{J}$ may be banded, in order to economize both memory and CPU time. In such a case, the `mfArray options%band` must be set; it must contain a vector of exactly two elements: ML and MU which are the widths of the lower and upper parts of the band, respectively, with the main diagonal being excluded (*e. g.* for a tridiagonal system, both elements are equal to 1). In this case, routine `jac` must store the elements $\widetilde{J}_{i,j}$ in the matrix `jacobian` as follows:

```
irow = i - j + ML + MU + 1
jacobian(irow,j) =  J̃_{i,j}
```

Actually, `ldjac` is equal to `2*ML+MU+1`: indeed, an extra storage is needed for the matrix factorisation. For an SPD matrix, simply take `ML` equal to zero in the previous formulae.

For the sparse case, the optional argument `sparse` must be set to `.true.` and the `jac` routine must be defined as:

```
interface
   subroutine jac( t, y, yprime, cj, nrow, job, pd, ipd, jpd, nnz )
      real(kind=MF_DOUBLE), intent(in)     :: t, y(*), yprime(*), cj
      integer,              intent(in)     :: nrow, job
      real(kind=MF_DOUBLE), intent(out)    :: pd(*)
      integer,              intent(out)    :: ipd(*), jpd(*)
      integer,              intent(in out) :: nnz
   end subroutine
end interface
```

c_J having the same meaning as before.

· · · / · · ·

[5] It is so called because it is the linear combination of two jacobian. Actually, it is close to the iteration matrix of the non-linear solver. See also this remark.

The `pd,ipd,jpd` f90 arrays describe the CSC (Compact Sparse Column) representation of the sparse jacobian matrix, as follows:

- `pd(1:nnz)` contains the non-zero matrix entries;
- `ipd(1:nnz)` contains the row indices;
- `jpd(1:ncol+1)` is the pointer to the beginning of the columns, in arrays `(pd,ipd)` (actually, a jacobian matrix is always square, hence `ncol`=`nrow`).

Row indices must be sorted in ascending order. Moreover, this routine must include a mechanism such that only the value of `nnz` is computed when `job`=0; therefore the argument `nnz` has generally an `intent(y)` for the first call and an `intent(in)` for subsequent calls.

Be aware that the sparse structure (that is to say, the indices' vectors `ipd` and `jpd`) must remain the same during all the integration process. Otherwise, use the option `spjac_const_struct` of the `mf_DE_Options` derived type. Moreover, you can even force to reuse the same sparse structure if you make many calls to `mf/msOdeSolve` during, *e. g.*, an iterative procedure: see the option `reuse_spjac_struct` of the `mf_DE_Options` derived type.

Remarks:

- on output, `y` may contain *NaN* values for some reasons. In such a case, use the `msDaeSolve` routine to further investigate the problem;
- `resid` and `jac` have imposed lists of arguments; the user may use module's data in order to exchange other information between his program and these routines (especially for checking the number of equations). Therefore, except for very simple cases, these two user's routines must be located inside a module, `USE`d also by the user's program.
- for debugging purposes, the user may inspect the values of y and y' during the integration process. See explanations about the fields `monitor_y_ind`, `monitor_yp_ind` and `monitor_pause` of the `mf_DE_Options` derived type.

Other calling syntaxes

```
call msDaeSolve( A, "save" [, jac_sparse] )
```

By using this call, the user is able to save all internal data of the integrator in the `mfArray A`, and then continue the integration later on. The optional argument `jac_sparse`, which is the name of the routine computing the sparse Jacobian matrix, is required when the Jacobian has a sparse structure.

```
call msDaeSolve( A, "restore" [, jac_sparse] )
```

This call is required to restore all internal data of the integrator, before a new integration sequence. The optional argument `jac_sparse`, which is the name of the routine computing the sparse Jacobian matrix, is required when the Jacobian has a sparse structure.

```
call msDaeSolve( "finalyze" )
```

This call is useful to avoid memory leaks at the end of the integration process.

See also: `mfOdeSolve`

msDaeSolve — integrator of DAE (implicit) systems

Calling syntax:

```
call msDaeSolve( mfOut( y, status                                                    &
                        [, tolout, yp, solve_log, init_log, t_log, order_log] ),  &
                 resid, t_span, y_0, yp_0[, options, jac, sparse] )
```

Description:

The behavior of this routine is similar to those of mfDaeSolve, but it allows to get more information returned by the integrator (via mfArrays which must be enclosed by the mfOut function):

- y contains the numerical result
- status contains a returned error code:

  status > 0: normal and reliable termination of the routine. It is assumed that the requested accuracy has been achieved. (Exact meaning of the status can be found below)

  status < 0: abnormal termination of the routine. The estimates for y is less reliable. It is assumed that the requested accuracy has not been achieved. Moreover, in some circumstances, a message is printed; in such cases, the user cannot use numerical values in y and yp and therefore an appropriate decision must be taken. (Exact meaning of the error can be found below)
- tolout is different from tol in some few cases. First, if status $= -2$, it contains an appropriate value for continuing the integration. Second, if status $= -101$, it contains the absolute value of the residual vector $R(t_0, y_0, y'_0)$ (initial data consistency).
- yp contains an estimation of the derivative at time t_{end} (or at all intermediate values defined in t_span); moreover, the role of the optional argument y_ind_out applies to yp.
- solve_log contains additional information about the integration process. It is a vector of 6 elements, which are:

  nb_step is the total number of steps used;

  dt_min the smallest step used;

  dt_max the largest step used;

  nb_resid is the number of calls of the resid routine;

  nb_jac is the number of evaluation of the jacobian matrix;

  nb_solve is the number of solving the linear system using the jacobian matrix.
- init_log contains specific information about the computation of consistent initial conditions (when this has been required). It is a vector of 4 elements, which are:

  nb_resid_0 is the number of calls of the resid routine;

  nb_jac_0 is the number of evaluation of the jacobian matrix;

  nb_solve_0 is the number of solving the linear system using the jacobian matrix;

  cpu_time_0 is the CPU time spent in this initialization part.
- t_log contains the whole history of the times used during the integration. This can be useful to detect where a critical behavior is located.
- order_log contains the whole history of the order used during the integration. The first value, not defined, is set to NaN (Not-a-Number special IEEE floating-point value); as a consequence, this mfArray has the same size as t_log.

.../...

Among these returned variables, only the first two are required, while the others are optional. Moreover, it is forbidden to use the same `mfArray` in place of two or more actual arguments, even empty.

Remark: The presence of the argument `status` allows the user to launch its program in a batch mode: indeed, errors in the current routine will not stop the program (it is the responsability of the programmer to test the value of the `status` variable before using the result). If the `mfDaeSolve` routine was used instead, an error would stop the program and, in debug mode, a traceback of this error would be displayed.

`status`	*explanation*
0	Normal termination of the solver.
4	Normal termination of the solver, but a large amount of work has been expended.
12	Normal termination of the solver, but an *End Condition* was found (`flag`, last argument in `resid`, was equal to 3 and the solver has been called to finish at the new current point).
−2	The error tolerances are too stringent.
−3	The local error test cannot be satisfied because you specified a zero value for the absolute error in `tol` and the corresponding computed solution component is zero. Thus, a pure relative error test is impossible for this component.
−6	The method used had repeated error test failures on the last attempted step.
−7	The corrector could not converge.
−8	The matrix of partial derivatives is singular.
−9	The corrector could not converge: there were repeated error test failures in this step.
−10	Abnormal termination of the solver, due to an *Illegal Condition* (`flag`, last argument in `resid`, was equal to −1 and the solver did its best to not obtain −1).
−11	Abnormal termination of the solver, due to an *Emergency Exit* (`flag`, last argument in `resid`, was equal to −2 and control has been returned to the calling program).
−13	Some indices for the non-negativity constraints stored in the `non_neg` array of the `mf_DE_Options` are not in the range `[1,Neq]` (where `Neq` is the number of equations).
−14	The user-supplied `resid` routine returned invalid entries (*NaN* value).
−15	The user-supplied `jac` routine (either dense or sparse) returned invalid entries (*NaN* value).
−20	Allocation error because the maximum of available memory has been reached. An explanation is printed.
−33	Invalid argument.
−34	Invalid implementation of the user-defined sparse jacobian.
−40	On calling, the number of output argument(s) is incorrect.
−100	The routine wasn't able to start, neither choose an initial timestep, nor check initial conditions for consistency (`flag`, last argument in `resid`, was different from 0).
−101	Initial conditions `y_0` and `yp_0` are not consistent.
−102	`msDaeSolve` failed to compute the initial vector `yp_0`.

See also: `msOdeSolve`, `mf_DE_Options`

`mf_NL_Options` options for non-linear solvers *(derived type)*

Description:

This derived type gathers options used by the two routines `mfFSolve` and `mfLsqNonLin`, and is defined as follows:

```
type :: mf_NL_Options

   type(mfArray)        :: tol
   integer              :: max_iter               = 50
   type(mfArray)        :: epsfcn
   real(kind=MF_DOUBLE) :: init_step_bound_factor = 1.0d2
   logical              :: print                  = .false.,            &
                           print_using_transf     = .false.,            &
                           reuse_spjac_struct     = .false.
   type(func_ptr), allocatable :: f_inv_transf(:)
   integer              :: check_jac              = 0,                  &
                           print_check_jac        = 0
   logical              :: box_constrained        = .false.
   type(mfArray)        :: lower_bounds,                                &
                           upper_bounds
   real(kind=MF_DOUBLE) :: sing_jac_tol           = epsilon(1.0d0)
   logical              :: print_sing_val         = .false.

end type mf_NL_Options
```

The following fields may be used in both two routines:

- `tol` specifies the tolerance to be used. See the appropriate routine to know the usage of this option.
- `max_iter` specifies the approximated maximum number of iterations.
- `epsfcn` specifies the optimum small increment (in a relative sense) used for computing the derivatives (the elements of the jacobian matrix) by finite differences. Therefore, its value is not used when the jacobian matrix is provided by the user. It may be a scalar or a vector; in the latter case (but only for `LsqNonLin`), each components of `epsfcn` corresponds to an unknown.
  The value(s) of `epsfcn` must be carefully chosen; there doesn't exist a universal constant because the optimum value depends on the algorithm in the user-supplied `fcn` routine. Its role is basically to equalize the numerical error which comes from the scheme and the roundoff error inherent to the floating-point model. If `fcn` is coded as a mathematical function, it can be set to zero (because internally, the finite-difference routine will choose $\sqrt{\epsilon}$, where ϵ is the machine precision; this is the default). If `fcn` uses an interpolation over a given interval Δx (*e. g.* from a table lookup) then the `epsfcn` should be related to some power of this Δx value, according the accuracy of the interpolation scheme.
- `init_step_bound_factor` specifies the initial step bound factor. The default value of 100 is recommended in most cases. However it can be changed for another value as low as 0.1. For example, if you use the $log()$ function to ensure the positiveness of your unknowns, you should choose $log(100)$ instead.
- `print` is used to print on screen the following information during the iterative process: the iteration number, the value of each unknown parameter and the norm of the residue. If some transformation is used between the physical value of the parameters and the actual value in the algorithm, it may be convenient to print the physical value of each parameter; to do that, the user must set the `print_using_transf` field to `.true.` and associate the function pointers stored in the field `f_inv_transf` to appropriate user-defined inverse functions. The last pointer array must be allocated by the user. See the *Muesli User's Guide* for more details.

· · · / · · ·

– `reuse_spjac_struct` must be set to `.true.` only when you are sure that the sparse structure of the jacobian is not modified during many different calls of the FSolve solver: this can occur inside a loop when using the same set of equations. Be aware that this option can only set to `.true.` *after* the first call of the solver; on the contrary, an error will arise. Note that this option is used only for the sparse case.

– for debugging purposes (but only when `MF_NUMERICAL_CHECK` is *TRUE*), `check_jac` can be used to make a check of the user jacobian (dense only):

1. when `check_jac` is set to 1, a global and quick check is done;
2. when `check_jac` is set to 2, a complete (but more expensive) check is done.

In both cases, a message is displayed according to the errors obtained concerning the accuracy of the jacobian. *Note that this is an estimation because (i) discrepancies found may be strongly affected by both floating-point roundoff errors and badly scaled variables or equations, and (ii) relative errors are based on a sort of norm of the jacobian; when this latter is large, some terms in the error matrix may remain small.*

`print_check_jac` can be used to print the results of the jacobian check:

1. when `print_check_jac` is set to 1, information is displayed on the screen;
2. when `print_check_jac` is set to 2, information is stored in files (one different file for each jacobian call).

In both cases, (i) the given information depends on the level of the check and (ii) the output is done at each call of the `jac` routine, regardless of the severity of the discrepancies. See the *Muesli User's Guide* where a detailed example of use can be found.

– (for `LsqNonLin` only) `box_constrained` can be used to constrain the parameters to be inside a box. Set it to `.true.` and specify the box bounds as follows:

1. `lower_bounds` is a vector containing the lower bound for each parameter. The length of this `mfArray` vector must be equal to the number of unknown parameters; use `-MF_INF` to release the constraint.
2. `upper_bounds` is a vector containing the upper bound for each parameter. The length of this `mfArray` vector must be equal to the number of unknown parameters; use `+MF_INF` to release the constraint.

These constraints of type box are dynamic, which means that during the iterations, the user can change the value of the bounds (*e. g.* in the `fcn` routine).

– Only when `MF_NUMERICAL_CHECK` is *TRUE*, `sing_jac_tol` allows the user to know whether his system of equations is singular or not. It concerns only the `LsqNonLin` routine. Normally the jacobian is not singular, which means that all the parameters are independant; on the contrary, the routine displays the linear relation between the parameters, by showing the vector of the basis of the nullspace. Note that the nullspace basis is always computed from a local point-of-view (linearized equations); if this basis changes along the iteration, this means that the relationship between the parameters is nonlinear. The default value of this tolerance is the machine ϵ. Set `sing_jac_tol` to zero to avoid the computation of the nullspace.

– Only when `MF_NUMERICAL_CHECK` is *TRUE*, `print_sing_val` allows the user to print the singular values of the jacobian matrix. Default is *FALSE*.

All fields of this derived type are `public`, so the user can access each component via the '`%`' Fortran 90 selector.

Lastly, the user can use the `msRelease` routine to free all fields of this structure (and to re-initialize the components between two different calls to any of these integrators).

See also: `mf_DE_Options`

mf_DE_Options — options for differential solvers *(derived type)*

Description:

This derived type gathers options used by the two integrators mfOdeSolve and mfDaeSolve, and is defined as follows:

```
type :: mf_DE_Options

   logical              :: continuation = .false.
   character(len=3)     :: method = ""
   type(mfArray)        :: tol, y_ind_out, band
   type(mfArray)        :: non_neg, y0_ind, yp0_ind
   logical              :: IC_known            = .true.,                &
                           print_progress      = .false.,               &
                           disp_times          = .false.,               &
                           jac_symm_pos_def    = .false.,               &
                           spjac_const_struct  = .true.,                &
                           reuse_spjac_struct  = .false.,               &
                           jac_investig        = .false.,               &
                           rational_null_basis = .false.,               &
                           pseudo_inverse      = .false.
   real(kind=MF_DOUBLE) :: jac_rcond_min       = MF_EPS

   logical              :: save_sing_jac       = .false.
   type(mfArray)        :: monitor_y_ind,                               &
                           monitor_yp_ind
   logical              :: monitor_pause       = .false.
   integer              :: check_jac           = 0,                     &
                           print_check_jac     = 0

   type(mf_DE_Named_Group), allocatable :: named_eqn(:),                &
                                           named_var(:)

end type
```

The following fields may be used in both integrators:

- `continuation` specifies that the integrator is not restarted. Using such a feature to continue the integration leads to much better performance than restarting the integrator.
- `tol` specifies the tolerance in many ways (relative or absolute, as a whole or by component).
- `y_ind_out` allows the user to get only a subpart of the result.
- `band` specifies that the jacobian is stored in a band format.
- `print_progress` specifies that the percentage of work already done is printed in the terminal. This options should not be used when using the main program in a batch mode, redirecting output in a file.
- `disp_times` specifies that time values will also be displayed – wall clock time is used, not CPU time.

  (See msPrepProgress and msPrintProgress for other information concerning the last two options.)

· · · / · · ·

- `jac_symm_pos_def` must be set to `.true.` only when you are sure that the jacobian is symmetric and positive definite; in such a case, you must only provide the upper part of the jacobian: this will economize some storage. A Cholesky factorization will be used (instead of a general LU one) and, especially for sparse, large matrices, this will be more efficient.
- `spjac_const_struct` must be set to `.false.` only when you are sure that the sparse structure of the jacobian is modified along all the time integration (this is not common!); note that this option is used only for the sparse case.
- `reuse_spjac_struct` must be set to `.true.` only when you are sure that the sparse structure of the jacobian is not modified during many different calls of the DAE or ODE solvers: this can occur during an iterative procedure using the same mesh. Be aware that this option can only set to `.true.` *after* the first call of the solver; on the contrary, an error will arise. Note that this option is used only for the sparse case.
- for debugging the DAE integration (but only if `jac_investig` is *TRUE*), the user may set a threshold value (not greater than unity) in `jac_rcond_min` in order to detect whether the jacobian is singular or not[6]. When (an estimation of) the reciprocal of condition number is less than or equal to the value of `jac_rcond_min`, then the jacobian is considered as singular: a warning message is then emitted. Moreover, for the dense case only:

  1. additional information is provided to the user, about the null space of the transpose of the jacobian matrix: this may help the user in understanding why the jacobian is singular.
  2. the jacobian and the nullspace of its transpose can be saved on disk according to the value of the boolean `save_sing_jac` (the filenames are respectively '*jacobian.dat*' and '*nullspace.dat*').

  The default value of `jac_rcond_min` is the machine epsilon and, if set by the user, it cannot be smaller than this default value. The nullspace basis may be returned in the form of a rational basis (set `rational_null_basis` to `.true.`) but this should be useful for small problem only (few number of equations) or for pedagogical reasons. The `pseudo_inverse` component is reserved for a future usage.
- for debugging purposes again, the user can tell the appropriate integrator to store all intermediate values (*note:* at the internal time steps!), by specifying a number of indices in `monitor_y_ind`, and/or indices in `monitor_yp_ind`. In such a case, the selected values of the vector y (resp. y') are copied in the file '*odesolve_y.out*' or '*daesolve_y.out*' (resp. '*odesolve_yp.out*' or '*daesolve_yp.out*'); each line contains the current time (independant variable) in the first column, then the selected values of the dependant variables. The boolean `monitor_pause` allows the user to pause after each successful internal step of the ODE/DAE solver (the default behavior is not to make such an interruption). Moreover, the data files are flushed continuously, so you can use an external graphic tool to plot the data which are monitored.
- recall that the use of `MF_NUMERICAL_CHECK` is recommended, because it allows internal routines to check the validity of the equations or of the jacobian elements, especially non finite values (Infinities, NaNs).

· · · / · · ·

[6] For the ODE solver, the jacobian provided by the user may be singular, it doesn't matter for the integration process. See the remark at the end of this entry.

– for debugging purposes (but only when MF_NUMERICAL_CHECK is *TRUE*), `check_jac` can be used to make a check of the user jacobian (dense only):

  1. when `check_jac` is set to 1, a global and quick check is done;
  2. when `check_jac` is set to 2, a complete (but more expensive) check is done.

  In both cases, a message is displayed according to the errors obtained concerning the accuracy of the jacobian. *Note that this is an estimation because (i) discrepancies found may be strongly affected by both floating-point roundoff errors and badly scaled variables or equations, and (ii) relative errors are based on a sort of norm of the jacobian; when this latter is large, some terms in the error matrix may remain small.*

  `print_check_jac` can be used to print the results of the jacobian check:

  1. when `print_check_jac` is set to 1, information is displayed on the screen;
  2. when `print_check_jac` is set to 2, information is stored in an overwritten file (a unique file along all the integration process). The library makes as many pauses as needed to let the user to inspect this file.

  In both cases, (i) the given information depends on the level of the check and (ii) the output is displayed at each call of the `jac` routine, regardless of the severity of the discrepancies. See the *Muesli User's Guide* where a detailed example of use can be found.

– the `named_eqn` and `named_var` fields allow the user to name his equations and variables by grouping them under useful, physical names. The derived type `mf_DE_named_group` is defined as follows:

```
type :: mf_DE_Named_Group
   character(len=132) :: name
   integer            :: begin, last
end type
```

  The user must allocate himself the arrays `named_eqn(:)` and `named_var(:)`, then correctly initializes the three internal fields: `name` (the name of the group), `begin` and `last` (the indices of equation in `deriv` or `resid` which define the beginning and the end of the each named group). See the *Muesli User's Guide* where an example of use can be found.

The field `method` is specific to the `mfOdeSolve` integrator. It may contains `"RKF"` (*Runge-Kutta Fehlberg*, which is the default method), `"ABM"` (*Adams-Bashforth-Moulton*), or `"BDF"` (*Backward Differentiation Formula*).

The remaining are specific to the `mfDaeSolve` integrator:

– `non_neg` allows the user to constrain some components of the solution to remain non negative.

– `IC_known` indicates whether initial conditions are known (and then supposed to be consistent) or not (see also just below).

– in the case where initial conditions have to be computed (*i. e.* when `IC_known` is equal to `.false.`) and only when the jacobian matrix is provided, the user may indicate which components of the vectors `y_0` and `yp_0` are prescribed in, respectively, `y0_ind` and `yp0_ind` (at most N components may be prescribed, where N is the number of the system's equations).

  If used, these two vectors must have a length equal to N; free components are selected by putting a zero at appropriate locations, whereas prescribed components may be selected by putting any non-zero value.

  The user may also leave the fields empty, which is equivalent to specify that no component at all are prescribed. Values present in the two `mfArrays` `y_0` and `yp_0` are then considered as initial guesses.

  It is worth mention that initialization may fail in the case where the user prescribed too many components. A very common error is to prescribed the N components of `y_0` by setting all com-

ponents of `y0_ind` to one, for a DAE systems which have N_a algebraic equations; in such a case, all components referencing to the algebraic equations should be left free, and the user can only prescribed the $N - N_a$ components of `y_0` corresponding to the differential equations.

All fields of this derived type are `public`, so the user can access each component via the '`%`' Fortran 90 selector.

Lastly, the user can use the `msRelease` routine to free all fields of this structure (and to re-initialize the components between two different calls to any of these integrators).

Remark: The jacobian matrix of the ODE solver may be singular, because the iteration matrix M writes:

$$M = I + \Delta t\, J = I + \Delta t\, \frac{\partial F}{\partial y}$$

It can be seen easily that for a sufficiently small time-step Δt, M is not singular so there is no constraint over J: it could be even the null matrix! Otherwise, the iteration matrix M used by the DAE solver is close to the following generalized jacobian:

$$M \approx \widetilde{J} = \frac{\partial R}{\partial y} + c_J \frac{\partial R}{\partial y'}$$

where c_J is related to the inverse of the time step Δt. In this latter case, $\widetilde{J}$ cannot be singular. (The two functions $F(t, y)$ and $R(t, y, y')$ have been defined respectively at the `mfOdeSolve` and `mfDaeSolve` entries.)

See also: `mf_NL_Options`

1.11 Sparse Matrices

`mfSparse`	sparse matrix conversion
`mfFull`	dense matrix conversion
`mfSpAlloc`, `msSpReAlloc`	sparse matrix space allocation
`mfNnz`	number of nonzero matrix elements
`mfNzMax`	sparse matrix internal size
`mfNcolMax`	sparse matrix internal size
`mfSpEye`	sparse identity matrix
`mfSpOnes`	replace nonzero elements with ones
`mfSpRand`	sparse uniformly distributed random matrix
`mfSpRandN`	sparse normally distributed random matrix
`mfSpDiags`	sparse matrix formed from diagonals
`mfSpCut`	cut small elements
`mfSpImport`, `msSpExport`	sparse matrix conversion from/to f90 arrays
`mfIsRowSorted`	row sorted inquiry
`msRowSort`	sparse matrix row sort
`msGetAutoRowSorted`	row sorting policy
`msSetAutoRowSorted`	row sorting policy
`mfMatFactor`	handle to internal matrix factors *(derived type)*
`msFreeMatFactor`	deallocation of internal matrix factors

See also:

Core Routines

File Input/Output

Data Analysis Functions

Operators

Elementary Math Functions

Specialized Math Functions

Elementary Matrix Manipulation Functions

Matrix Functions

Polynomial Functions

Optimization and Function Functions

`mfSparse` — sparse matrix conversion

Calling syntax:

```
B = mfSparse( A )
```

converts the `mfArray A` (which should be dense) into a sparse `mfArray`.

To create a sparse matrix from scratch, use the `mfSpAlloc` routine instead; from existing contents, use `mfSpImport`.

See also: `mfFull`, `msSpExport`

`mfFull` — dense matrix conversion

Calling syntax:

```
B = mfFull( A )
```

converts the `mfArray A` (which should be sparse) into a dense `mfArray`.

See also: `mfSparse`

`mfSpAlloc` sparse matrix space allocation

Interface:

```
function mfSpAlloc( m, n, nzmax, ncolmax, kind ) result( out )

   integer,          intent(in)           :: m, n
   integer,          intent(in), optional :: nzmax, ncolmax
   character(len=*), intent(in), optional :: kind
   type(mfArray) :: out
```

Description:
This routine is used to create new space for a sparse `mfArray` of size `m` by `n` (logical dimensions) which will contain at most `nzmax` (if present) non-zero values. If `nzmax` is not present, it is set to zero.

If `ncolmax` is present, it reserves space for a greater number of columns. The value of `ncolmax` cannot be less than `n`; it defaults to `n`. This argument is usually employed with the routine `msHorizConcat`.

`kind` may be equal to `"real"` or `"complex"`; it defaults to `"real"`.

See also: `msSpReAlloc`, `msHorizConcat`

`msSpReAlloc` sparse matrix space re-allocation

Interface:

```
subroutine msSpReAlloc( A, nzmax, ncolmax )

   type(mfArray), intent(in out)          :: A
   integer,       intent(in),    optional :: nzmax
   integer,       intent(in),    optional :: ncolmax
```

Description:
This routine changes the space for elements in the sparse `mfArray A`. At least one argument, among the two optional ones `nzmax` and `ncolmax`, must be present.

The argument `nzmax` corresponds to the maximum number of non zero entries in the sparse structure. It can be increased to any value, but you will get an error if the value of `nzmax` is too small.

The argument `ncolmax` corresponds to the maximum number of the columns. Here again, you will get an error if the value of `ncolmax` is too small. Be careful when changing its value, because the logical size of the matrix is determine by `ncolmax`.

Usually the two arguments are used to *increase* the corresponding value. It may be dangerous to try to *decrease* their value without knowledge about the internal sparse structure. See below to decrease as much as possible the internal size of the structure automatically, without loss of data.

Other calling syntax:
To compact an `mfArray` to its minimal room space (both in terms of the maximum number of non zero entries `nzmax`, and maximum number of columns `ncolmax`), use the following special calling sequence:

```
call msSpReAlloc( A, "minimal" )
```

Note that this will remove also any (unwanted) zeros actually stored.

See also: `mfSpAlloc`

`mfNnz` number of nonzero matrix elements

Interface:

```
function mfNnz( A ) result( out )

   type(mfArray), intent(in) :: A
   integer :: out
```

Description:
Returns the number of non-zero element of the sparse `mfArray A`.

See also: `mfNzMax`

`mfNzMax` — sparse matrix internal size

Interface:

```
function mfNzMax( A ) result( out )

   type(mfArray), intent(in) :: A
   integer :: out
```

Description:
Returns the maximum number of non-zero element of the sparse `mfArray A`.

See also: `mfNnz`, `mfNcolMax`

`mfNcolMax` — sparse matrix internal size

Interface:

```
function mfNcolMax( A ) result( out )

   type(mfArray), intent(in) :: A
   integer :: out
```

Description:
Returns the maximum number of columns of the sparse `mfArray A`.

See also: `mfNnz`, `mfNzMax`

`mfSpEye` — sparse identity matrix

Interface:

```
function mfSpEye( m, n, kind ) result( out )

   integer,          intent(in)           :: m
   integer,          intent(in), optional :: n
   character(len=*), intent(in), optional :: kind
   type(mfArray) :: out
```

Description:
This routine creates a sparse `mfArray` of size `m` by `n`, having ones on its main diagonal.

The optional argument `kind` allows the user to choose a complex array (if `kind` is equal to `"complex"`); by default, a real matrix is returned.

See also: `mfSpDiags`, `mfSpOnes`

`mfSpOnes` — replace nonzero elements with ones

Calling syntax:

```
B = mfSpOnes( A )
```

Description:
This routine creates a sparse matrix by replacing all non-zero values of the `mfArray A` by one.

Be aware that this routine is *NOT* the sparse equivalent of `mfOnes`.

See also: `mfSpEye`

`mfSpRand` — sparse uniformly distributed random matrix

First calling syntax:

```
B = mfSpRand( A )
```

Description:
This routine creates a sparse matrix by replacing all non-zero values of the `mfArray A` (which may be dense or sparse) by a uniformly distributed random value. Therefore, the `mfArray B` has the same sparsity than `A`.

Second calling syntax:

```
B = mfSpRand( m, n, density )
```

Description:
creates a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` uniformly distributed nonzero entries (the real `density` must be in [0,1]).

See also: `mfSpEye`, `mfSpOnes`, `mfSpRandN`

`mfSpRandN` sparse normally distributed random matrix

First calling syntax:

```
B = mfSpRandN( A )
```

Description:
This routine creates a sparse matrix by replacing all non-zero values of the `mfArray A` (which may be dense or sparse) by a normally distributed random value. Therefore, the `mfArray B` has the same sparsity than `A`.

Second calling syntax:

```
B = mfSpRandN( m, n, density )
```

Description:
creates a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` normally distributed nonzero entries (the real `density` must be in [0,1]).

See also: `mfSpEye`, `mfSpOnes`, `mfSpRand`

`mfSpDiags` — sparse matrix formed from diagonals

Interface:

```
function mfSpDiags( m, n, v, d ) result( out )

   integer, intent(in) :: m, n
   type(mfArray) :: v
   integer, intent(in), optional :: d

   type(mfArray) :: out
```

Description:
Creates a sparse matrix (of size `m` by `n`) whose `d`-diagonal is the input vector `v` (dense).

The vector `v` may have a length greater than those of the `d`-diagonal (in such a case, subsequent elements will not be referenced); it can be also a scalar (in such a case, the scalar value is spread over the diagonal).

Remarks: When `d` is equal to zero, it points to the main diagonal; when positive, the upper part of the matrix is concerned. If the arg. `d` is not present, the main diagonal is the default.

Simplified interface:

```
function mfSpDiags( v ) result( out )

   type(mfArray) :: v

   type(mfArray) :: out
```

Description:
When only one argument is used (it must be a vector `mfArray A`), then the routine outputs a diagonal square sparse matrix whose dimensions are deduced from the length of the vector `d`.

Other interface:

```
function mfSpDiags( m, n, A, d ) result( out )

   integer, intent(in) :: m, n
   type(mfArray) :: A
   integer, intent(in) :: d(:)

   type(mfArray) :: out
```

Description:
In this second interface the input `mfArray A` may be a rank-2 array. So, each column of `A` is used to fill a diagonal of the returned sparse matrix, whose index is specified in the vector `d`.

If `A` is a row vector then each constructed diagonal will have the same elements.

See also: `mfSpEye`, `mf/msDiag`

`mfSpCut` — cut small elements

Calling syntax:

```
B = mfSpCut( A, threshold )
```

Description:
This routine removes all elements of the `mfArray A` (sparse, real or complex) whose magnitude is less or equal than the prescribed `threshold` (positive real arg.).

`mfSpImport` sparse matrix conversion from f90 arrays

Calling syntax:

```
A = mfSpImport( i, j, d[, m, n, nzmax, format, duplicated_entries, pattern ] )
```

creates the sparse `mfArray A` from a sparse structure stored in classical f90 arrays; `i` and `j` are integer f90 vectors whereas `d` is a real or a complex f90 vector.

The vector `d` may have a null size, in which case the allocation of data space is made by this routine but filled with *NaN* value (initialization may occur later on).

If present, the character string `format` specifies the sparse format; it may be equal to `"COO"` (*COOrdinates*), `"CSC"` (*Compact Sparse Columns*) or `"CSR"` (*Compact Sparse Rows*). By default, `format="COO"` is assumed.

If present, `m` and `n` define the (logical) shape of the sparse `mfArray`; these shape must be large enough to contain all the input indices specified in `i` and `j`. By default, the logical shape is deduced as follows:

- if the format is `"COO"`, then the matrix shape is deduced from the maximum value of arrays `i` and `j`. It follows than these two arrays are inspected.
- if the format is `"CSC"` (resp. `"CSR"`), the number of columns (resp. rows) comes from the size of the `j` array (resp. of the `i` array). Then the other dimension is deduced by inspecting the appropriate part of the other integer array; this is because the long index array may be overdimensioned.

If present, `nzmax` allows to reserve more place than the actual size.

During importing f90 arrays:

- eventual duplicated entries are treated according to the optional character argument `duplicated_entries`. When this argument is present and equal to `"ignored"`, duplicated entries are ignored; when it is equal to `"added"` (default value), duplicated entries are added; when it is equal to `"replaced"`, last entries found overwrite previous ones.
- entries containing a null value are removed.

If present, the character string `pattern` indicates that the input matrix has a specific pattern, and consequently that only a part of the values may be present in the vector `d`:

- `"symm"`: the input matrix is symmetric (real or complex) and, possibly, only the lower or the upper part is provided by the user.
- `"skew"`: the input matrix is skew-symmetric (real or complex) and, possibly, only the lower or the upper part is provided by the user. No diagonal element can be found in the entries.
- `"herm"`: the input matrix is hermitian (complex) and, possibly, only the lower or the upper part is provided by the user. For a real matrix, it is equivalent to `"symm"`.

Note that a check about the pattern is done after importing the input values; in other words, giving a random matrix will raise an error.

See also: `msSpExport`, `mfSparse`, `mfLoadSparse`, `msSaveSparse`

`msSpExport` sparse matrix conversion to f90 arrays

Calling syntax:

```
call msSpExport( A, i, j, d [, format ] )
```

copies the sparse `mfArray A` to a sparse structure using classical f90 arrays.

`i` and `j` are integer f90 vectors. `d` is a real or a complex f90 vector.

If present, the character string `format` specifies the sparse format; it may be equal to `"COO"` (*COOrdinates*), `"CSC"` (*Compact Sparse Columns*) or `"CSR"` (*Compact Sparse Rows*). By default, `format="COO"` is assumed.

See also: `mfSpImport`, `mfSparse`, `mfLoadSparse`, `msSaveSparse`

`mfIsRowSorted` — row sorted inquiry

Interface:

```
function mfIsRowSorted( A ) result( bool )

   type(mfArray), intent(in out) :: A
   logical :: bool
```

Description:
Returns '`.true.`' if the columns of the sparse `mfArray A` are "row sorted" and contain no duplicated entries.

See also: `msRowSort`, `msGetAutoRowSorted`, `msSetAutoRowSorted`

`msRowSort` — sparse matrix row sort

Calling syntax:

```
call msRowSort( A [, struct_only ] )
```

Description:
Sorts all columns of the sparse `mfArray A` in increasing order, and remove all duplicated entries.

If the logical optional argument `struct_only` is present and equal to '`.true.`' then the data array is discarded.

See also: `mfIsRowSorted`, `msGetAutoRowSorted`, `msSetAutoRowSorted`

`msGetAutoRowSorted` — row sorting policy

Interface:

```
subroutine msGetAutoRowSorted( auto_row_sorted )

   logical, intent(out) :: auto_row_sorted
```

Description:
If the returned logical `auto_row_sorted` is *TRUE*, then all sparse matrices are row sorted when needed.

See also: `msRowSort`, `mfIsRowSorted`, `msSetAutoRowSorted`

`msSetAutoRowSorted` — row sorting policy

Interface:

```
subroutine msSetAutoRowSorted( auto_row_sorted )

   logical, intent(in) :: auto_row_sorted
```

Description:
Only if `auto_row_sorted` is *TRUE*, then all sparse matrices will be row sorted when needed. Therefore, this routine can change this row sorting policy.

By default, `auto_row_sorted` is *TRUE*.

See also: `msRowSort`, `mfIsRowSorted`, `msGetAutoRowSorted`

`mfMatFactor` handle to internal matrix factors *(derived type)*

Description:

This derived type is a handle to (sparse or dense) factors computed by some factorization routine (*e. g.* `msLU`, `msQR` or `msChol`). Declaration is made as follows:

```
type(mfMatFactor) :: factor
```

The user cannot manipulates the contents of this derived type: it is indeed intended to be used only by other MUESLI routines.

The user may release this structure by use of the `msFreeMatFactor` routine (but use of `msRelease` is equivalent).

`msFreeMatFactor` deallocation of internal matrix factors

Calling syntax:

```
call msFreeMatFactor( factor )
```

Description:
Deallocate the `mfMatFactor` `factor`.

Remark: The routine `msRelease` may also be used in an equivalent way.

2 FGL: Graphical Library

FGL contains all routines to make figures and plots (only in 2D).

Routines beginning with '`ms`' are subroutines, whereas those beginning with '`mf`' are functions. Usually, the two forms of the same routine do the same job, but the latter returns an identification (actually, a handle to the graphic object).

The available routines have been grouped into sub-parts:

2.1 Global graphic settings

`msSetBackgroundColor`	background color setting
`msSetColorOverflowPolicy`	set color overflow policy
`mfGetColorOverflowPolicy`	get color overflow policy
`msExitFgl`	FGL closing
`mfGetX11Device`, `msSetX11Device`	X11 device status
`mfGetX11ColorDepth`	get X11 screen color depth
`mfGetDefaultCapStyle`, `msSetDefaultCapStyle`	default cap style for lines
`mfGetDefaultJoinStyle`, `msSetDefaultJoinStyle`	default join style for lines
`mfGetCharEncoding`, `msSetCharEncoding`	character encoding

See also:

Window's and figure's management

Figure properties

Figure annotation – Low level graphic object's manipulation

High level plotting routines

Interactive routines

`msSetBackgroundColor` — background color setting

Interface:

```
subroutine msSetBackgroundColor( color_string )

   character(len=*), intent(in) :: color_string
```

Description:
Set the background color. The only two possibilities are `"white"` (the default) and `"black"`.

This routine must be used at the beginning of the graphic part. The background color must be the same for all the opened windows; it cannot be changed during the execution, until *FGL* graphic part is closed (`msExitFgl`) and automatically reinitialized by opening any figure.

Remark: When the background has been selected as black, printing leads to different behavior for `EPS` and `PDF` formats (see `msPrint`).

`mfGetColorOverflowPolicy` get color overflow policy

Interface:

```
function mfGetColorOverflowPolicy() result( policy )

   character(len=10) :: policy(2)
```

Description:
Returns two strings. The first one (resp. second one) refers to color overflow in the low part (resp. high part) of the colormap.

The color overflow policy may be `"signaled"` or `"truncated"`. When it is `"signaled"`, a contrasted color is used to signal the colormap overflow. When it is `"truncated"`, the lowest color (or highest color) is used, giving a smooth aspect to the colors.

Default behavior is `"signaled"` for both ends.

See also: `msSetColorOverflowPolicy`

`msSetColorOverflowPolicy` set color overflow policy

Interface:

```
subroutine msSetColorOverflowPolicy( low, high )

   character(len=*), optional :: low, high
```

Description:
This routine is used to monitor the colormap overflow. `low` and `high` are two strings which must contain `"signaled"` or `"truncated"`.

When it is `"signaled"`, a contrasted color is used to signal the colormap overflow. When it is `"truncated"`, the lowest color (or highest color) is used, giving a smooth aspect to the colors.

Default behavior is `"signaled"` for both ends.

Remark: if none of the two arguments is present, then nothing is changed.

See also: `mfGetColorOverflowPolicy`

`mfGetX11Device` — Get X11 device status

Interface:

```
status = mfGetX11Device( )
```

Description:
Returns the status of the X11 device, which can be `"on"` (the default) or `"off"`.

See also: `msSetX11Device`

`msSetX11Device` — Set X11 device status

Interface:

```
subroutine msSetX11Device( flag )

   character(len=*), intent(in) :: flag
```

Description:
During execution, as MUESLI is able to produce printed images (EPS or PDF) in an independant way, the X11 device can be de-activated, by setting the `flag` parameter to `"off"`. It can be useful for batch executions.

Remarks:

- the X11 device is `"on"` by default;
- it can be enabled/disabled by the MFPLOT_X11_DEVICE environment variable, which can take the value 0 or 1.

See also: `mfGetX11Device`

`mfGetX11ColorDepth` get color depth

Interface:

```
function mfGetX11ColorDepth() result( depth )

   integer :: depth
```

Description:
Returns the color depth currently used by the X11 server. A figure must have been previously opened.

See also: `mf/msColormapSize`, `mf/msColormap`, `msColorbar`

`msExitFgl` — FGL closing

Calling syntax:

```
call msExitFgl()
```

This routine is intended to be used at the end of your graphical program. It leaves a prompt to the user and waits for an answer ('`y`' or '`Y`'), after warning the user by a text like:

```
***
End of MUESLI plotting:
- all figures are going to be closed.
- graphic memory will be properly cleaned.
Do you really close graphics ? [y|Y]
```

It is a useful command to avoid that all graphic windows disappear; as mentioned, it also cleans the FGL internal memory (all graphic objects); moreover, X11 windows position and size are saved in a small database.

Remarks: If you want that your program terminates without a user action, you can redirect the standard input (via '`<`' in a shell command) and read a file containing the '`y`' symbol. Another possibility is to use the '`yes`' unix command which outputs repeatedly the character '`y`' to your program via a unix pipe, *e. g.*:

```
$ yes | a.out
```

See also: `msRelease`

`mfGetDefaultCapStyle` — Get default cap style for lines

Calling syntax:

```
cap_style = mfGetDefaultCapStyle( )
```

Description:
Returns the current default cap style, which may be 0 (`"CapButt"`), 1 (`"CapRound"`) or 2 (`"CapProjecting"`).

The signification of these three attributes may be found easily on the web.

See also: `msSetGrObj`, `msSetDefaultCapStyle`, `msSetDefaultJoinStyle`

`msSetDefaultCapStyle` — Set default cap style for lines

Interface:

```
subroutine msSetDefaultCapStyle( cap_style )

   character(len=*), intent(in) :: cap_style
```

Description:
Change the global value for the default cap style. The argument `cap_style` may be set to `"CapButt"`, `"CapRound"` or `"CapProjecting"`.

The signification of these three attributes may be found easily on the web.

Note also that this graphic attribute may be change for a particular object, via the `msSetGrObj` routine.

See also: `mfGetDefaultCapStyle`, `mfGetDefaultJoinStyle`

`mfGetDefaultJoinStyle` — Get default join style for lines

Calling syntax:

```
join_style = mfGetDefaultJoinStyle( )
```

Description:
Returns the current default join style, which may be 0 (`"JoinMiter"`), 1 (`"JoinRound"`) or 2 (`"JoinBevel"`).

The signification of these three attributes may be found easily on the web.

See also: `msSetGrObj`, `msSetDefaultJoinStyle`, `msSetDefaultCapStyle`

`msSetDefaultJoinStyle` — Set default join style for lines

Interface:

```
subroutine msSetDefaultJoinStyle( join_style )

   character(len=*), intent(in) :: join_style
```

Description:
Change the global value for the default join style. The argument `join_style` may be set to `"JoinMiter"`, `"JoinRound"` or `"JoinBevel"`.

The signification of these three attributes may be found easily on the web.

Note also that this graphic attribute may be change for a particular object, via the `msSetGrObj` routine.

See also: `mfGetDefaultJoinStyle`, `mfGetDefaultCapStyle`

`mfGetCharEncoding` — Get character encoding

Interface:

```
encoding = mfGetCharEncoding( )
```

Description:
Returns the current character encoding, which is `"Latin"` (the default) or `"UTF-8"`.

This is used in the following graphic routines, which display character strings in a figure: `mf/msText`, `msTitle`, `msXLabel`, `msYLabel`, `mf/msFigure` and `msLegend`.

See also: `msSetCharEncoding`

`msSetCharEncoding` — Set character encoding

Interface:

```
subroutine msSetCharEncoding( encoding )

   character(len=*), intent(in) :: encoding
```

Description:
When processing character's strings (*e. g.* in the `msTitle` routine), MUESLI accepts two different character encodings. The charset may be `"Latin"` (default one) or `"UTF-8"`.

So, the `encoding` argument can take only the following values:

- `"Latin"` (or `"ISO-8859-1"` which is an alias)
- `"UTF-8"`

Warning: Not all the characters defined in the `"UTF-8"` encoding are valid. Actually, only a subpart of the 256 first characters (*i. e.* mostly of the `"Latin"` encoded characters, but not all) are correctly processed.

See also: `mfGetCharEncoding`

2.2 Window's and figure's management

`mf/msFigure`	figure creation
`msResizeWindow`	window resizing
`mfGetWinId`	current figure identification
`msClose`	figure deletion
`mfGetColorScheme`, `msSetColorScheme`	color scheme
`mfGetColorInd`, `msSetColorInd`	index in color scheme
`msPrint`	file printing (EPS and PDF)
`msSetPdfOC`	PDF Optional Contents management

See also:

Global graphic settings

Figure properties

Figure annotation – Low level graphic object's manipulation

High level plotting routines

Interactive routines

`mf/msFigure` figure creation

Calling syntax:

```
call msFigure( [win_id] [, position] [, size] [, title] )
```

where arguments are all optional.

This routine opens a new window or selects (and therefore makes active) a previously opened one.

The `win_id` argument (integer) indicates the identification of the window; if not present, a sequential integer from 1 is used.

The `position` argument is used to impose the position of the window on the screen. It must be a Fortran array of two integer values specifying the position of the newly created window; the position in pixels of the top-left corner of the window is taken from the top-left corner of the whole screen (virtual screen in case of many monitors), counting positively downward for the y value. Default position is approximately $(0, 0)$ for the first window, and shifted both in x- and y-position, in order to be not completely overlapped (for the windows below others, only their title is visible).

The `size` argument is used to impose the size of the window on the screen. It must be a Fortran array of two integer values specifying the size of the newly created window; the two values indicates the width and height in pixels of the window. Default size is 800 by 600.

The `title` argument (character string, of maximum length 80) specifies the title of the figure. This title is visible in the X11 window title; it applies also for the printed EPS and PDF files.

Remarks:

- the default size of X11 windows can be modified via the environment variables MFPLOT_X11_WIDTH and MFPLOT_X11_HEIGHT.
- the window title always contains `win_id` and the name of the executable program.
- ordinarily, if you have no figure opened, most of graphical routines automatically open a new one.
- maximum number of concurrent figures is 64. However, `win_id` values don't need to be contiguous, and may be even greater than 64.

The other form:

```
win_id = mfFigure()
```

creates a new figure and returns its (integer) window identification.

Remarks: use `mfGetWinId` to get the window identification of the current active figure.

See also: `msClose`, `msClf`

`msResizeWindow` — window resizing

Calling syntax:

```
call msResizeWindow( [width, height] )
```

Description:
If the optional arguments `width` and `height` are both present, then the window is resized by the library.

On the contrary, *i. e.* when the optional arguments are not present, this routine becomes interactive, prints a prompt on the terminal, and wait the resizing of the window by the mouse.

See also: `msFigure`, `msClf`

`mfGetWinId` — current figure handle

Calling syntax:

```
win_id = mfGetWinId( )
```

Description:
Returns the window identification (integer) of the current active figure.

See also: `msFigure`

`msClose` — figure deletion

Calling syntax:

```
call msClose( [win_id] )
```

Description:
Closes the specified window, or the current window if the optional argument `win_id` is not present.

Note that after closing a window, the current selected window is undefined, even if other figures yet exist. It is required to select another existing window if you wish to plot inside, else a new figure will be automatically created.

See also: `msFigure`, `msClf`

`mfGetColorScheme` — Get color scheme

Calling syntax:

```
color_scheme = mfGetColorScheme( )
```

Description:
Returns the color scheme attached to the selected window, which is an integer ranged in [1-4]. Default is 3.

There are four different color schemes, used for recycling colors when you plot several lines on the same figure. They are described in the *Muesli User's Guide* (cf. § 5.2 Color management).

See also: `msSetColorScheme`

`msSetColorScheme` — Set color scheme

Interface:

```
subroutine msSetColorScheme( color_scheme )

   integer, intent(in) :: color_scheme
```

Description:
Set the color scheme for the selected window, via an integer argument ranged in [1-4].

There are four different color schemes, used for recycling colors when you plot several lines on the same figure. They are described in the *Muesli User's Guide* (cf. § 5.2 Color management).

The default color scheme is the third one, *i. e.* the new colors in Matlab (from release R2014b).

See also: `mfGetColorScheme`

`mfGetColorInd` — Get color scheme index

Calling syntax:

```
ind = mfGetColorInd( )
```

Description:
Returns the color scheme index attached to the selected window.

This is the index of the next color to be used.

See also: `msSetColorInd`

`msSetColorInd` — Set color scheme index

Interface:

```
subroutine msSetColorInd( ind )

   integer, intent(in) :: ind
```

Description:
Set the color scheme index for the selected window, and for the current color scheme.

Usually, when plotting many curves on the same figure, colors are cycled inside the colortable of the selected color scheme. This routine allows to select a specific index, for example to avoid the cycling in the colors.

See also: `mfGetColorInd`

`msPrint` — file printing

Calling syntax:

```
call msPrint( filename [, file_format ] )
```

Description:
Prints the current figure in a file named `filename`.

Currently, `file_format` may be only `"EPS"` or `"PDF"`. If this optional argument is not present, MUESLI tries to deduce the device type from the extension of the provided `filename`. By default, an EPS file is created.

For EPS (resp. PDF) files, and for debugging purpose, comments can be inserted in the generated file by setting the environment variable MFPLOT_EPS_COMMENTS (resp. MFPLOT_PDF_COMMENTS) to 1 (default is no comments). Be aware that this will increase the size of the file.

Note, if you have selected a black background (see `msSetBackgroundColor`), that the EPS only will have also a black background. For the PDF files have always a white background.

Remarks:

- EPS and PDF files intensively make use of language-based shading. This means that all coloured gradients (from `msPatch` or `msPColor` for example) are treated by using appropriate PostScript or PDF commands, and leads to smaller files.
- Of course, printed files may be generated without any X11 capability. This is useful under Windows (where there is no X11 access) or for batch processes.
- For PDF files only, optional contents (also known as layers) may be inserted. See `msSetPdfOC`.

`msSetPdfOC` **PDF Optional Contents management**

The first interface is:

```
subroutine msSetPdfOC( handle,                                          &
                       name, mutex, persistent, super_group )

   integer,          intent(in)           :: handle
   character(len=*), intent(in), optional :: name, mutex, super_group
   logical,                      optional :: persistent
```

It is used to tag a graphic object of the current window as *Optional Content* in the PDF. Doing so, and once the *Layers* of your PDF viewer is opened, you should be able to choose to display or not this graphic object (which appears under the `name`) on the screen.

The graphic object, referenced by its `handle`, may be visible or not. The initial state of visibility in the PDF layers will be consistent with the visibility in the X11 window at the time of printing, except when the `persistent` argument is used: in such a case, only the graphic object tagged as *persistent* will be initially visible in the *mutex* group.

If `name` is not present, then it will be assigned the corresponding graphic object number (as *"handle i"*).

The `mutex` optional argument *(MUTually EXclusive)* is used to group some optional contents in a *Radio-Button group*: at most one of these optional contents may be visible in the PDF (but that does not prevent the existence of a state where none of these are visible — see below the section concerning `persistent` if you want that exactly one content must be visible at a time). The `mutex` name is the name which will appear in the "layers" panel of the PDF viewer, which enclosed all different names. Same thing for the `super_group` optional argument, except that the groups inside are not in mutually exclusion.

When using the `mutex` feature, replacing '`name`=*name*' by '`persistent=.true.`' allows that one object is always visible, and it is the concerned object by default. A typical application is to group text annotations in the figure in mutually exclusive groups, one for a language; each group may have its own `name`, for example, equal to *Français*, *Deutsch*, etc. The `mutex` name could be then, *Other languages (default: English)*. See the `Labels_and_Languages_test.f90` program under `src/test/fgl`.

The other interface:

```
subroutine msSetPdfOC( handle,                                          &
                       merge, name, mutex, persistent, super_group )

   integer,          intent(in)           :: handle(:)
   logical,                      optional :: merge, persistent
   character(len=*), intent(in), optional :: name, mutex, super_group
```

concerns a vector of `handle`, and can be used to the following aims:

- when the two arguments `merge` and `name` are *both present*, it merges a group of graphic objects under the same `name` as optional contents; as an example of use, see the `PlotCubicBezier_test` program under `src/test/fgl`.
- when the two arguments `merge` and `name` are *both not present*, it gets a different name for each graphic objects of the group, from the legend of the figure. Of course, such a legend should exist; on the contrary, a generic name (as *"handle i"*) will be used instead. As an example of use, see the figure 4 of the `Legend_test` program under `src/test/fgl`.

The `mutex` optional argument has the same meaning as in the first interface. Its use implies the merge of all optional contents specified by the array of `handles`. Same thing for `super_group`.

···/···

Constraints about the presence of optional arguments:

- `mutex` and `super_group` arguments cannot be both present.
- `persistent` and `name` arguments cannot be both present.
- in a given group of optional contents, the `persistent` argument can be used only one time.
- currently (but this may change in a future version), there are only one Radio-Button group and only one Super-group for each figure.

Remarks:

- this routine must be called before creating the PDF file via `msPrint`.
- you are allowed to first call the second interface, using an array of handle, getting the names from the legend and then, call the first interface to modify the name of one specific graphic object.
- the PDF Optional Contents will be ignored if you print in the EPS format.

2.3 Figure properties

mf/msAxis	figure axes setting
msAxisFontSize	axis font size setting
msAxisLineWidth	axis line width setting
msCharInPixels	character size policy
mf/msCAxis	get or set color axis
msClf	clear current figure
msColorbar	color bar setting
mf/msColormap	get or set colormap
mf/msColormapSize	get or set colormap size
msDrawBox	redraw the box in current figure
msDrawGrid	if needed, redraw the grid in current figure
msRedrawFigure	redraw the current figure
msGetX11Pixmap	X11 pixmap saving
msGrid	grid setting
msHold	plot command superposition
msRemoveClipBox	remove a clipping box inside axes
msSetClipBox	define a clipping box inside axes
msShading	color shading mode
mfGetXAxisTicksNb	get ticks number of the X-axis
msSetXAxisUserLabels	set customized ticks labels of the X-axis
mf/msTitle	title setting
msTitleFontSize	title font size setting
mf/msXLabel	x-label setting
mf/msYLabel	y-label setting
msLabelFontSize	descriptive labels font size setting
msAxisLabelFormat	kind of labelling for Axis
msSetWinProp	window properties setting
msLegend	add legends to curves, in one frame
mfLegend	add legends to curves, possibly in many frames

See also:

Global graphic settings

Window's and figure's management

Figure annotation – Low level graphic object's manipulation

High level plotting routines

Interactive routines

`mf/msAxis` figure axes setting

The *function form*

```
function mfAxis( [ axis ] ) result( mf_range )

   type(mfArray)               :: mf_range
   character(len=*), optional  :: axis
```

returns the current axis values of the selected figure, always in the order: *left*, *right*, *bottom*, *top*, whatever the mode (`"xy"` or `"ij"` – see below) is. This array is a vector of length 4 or 2, containing extremal values of the x- and/or y-axis, according to the presence of the optional argument `axis` which must be equal to the single character `"x"` or `"y"`.

The *subroutine form* has the following calling syntax:

```
call msAxis( mode | dp_range | mf_range [, axis ] )
```

If the character string `mode` is present, it may have the following values:

- `"on"` / `"off"`: axes are drawn (default) or hidden (and so for X label and Y label).
- `"auto"`: axes' ranges are computed from the actual numerical values of data to be plotted, giving *smart* extremal values; this is the default. Note that graphic data involved do not include arrows or texts, which are considered as annotations.
- `"tight"`: axes' ranges are computed from actual values of data (see just above what is considered as "data"), strictly keeping their extremal values.
- `"manual"`: axes' ranges are not automatic; the current axis values are frozen, whatever the following plots. This mode is also set when a range is explicitly used (see below). Only another call using the `"auto"` or `"tight"` argument can unlock it.
- `"equal"`: x- and y- axis are drawn with the same scale; the default behaviour is `"unequal"`. Note that the *equal* mode is possible only when axes have the same scaling (both linear or both logarithmic).
- `"xy"` / `"ij"`: the origin of the x- and y-axis is at the bottom left corner of the figure (default location for most plotting commands) or at the top left corner (when the data is interpreted as a matrix, which is the case for `msSpy`, `msImage` and `msPColor` in the absence of (X,Y) coordinates). In the `"xy"` mode, the x-axis labels are written at the bottom side of the box, whereas they are written at the top one in the other mode.
- `"inverted"`: in this case, the second optional argument `axis` must be present and equal to `"x"` or `"y"`. Corresponding extremal values of this axis are then swapped together. This doesn't imply that the axis becomes `"manual"`. There is no `"normal"` state, *i. e.* two successive calls with the `"inverted"` argument is equivalent to do nothing; in other words, `"inverted"` is not a property but an action to realize.
- `"linlin"` / `"loglog"`: x- and y-axis are both linear (default) or logarithmic (base 10).
- `"loglin"` / `"linlog"`: as specified, one of the two axes is linear whereas the other is logarithmic (base 10); Note that if the *equal* mode is set, it will be disabled.

If the `real dp_range`, or `mfArray mf_range`, is present (containing 4 values), it is used to initialize the axes' range; the *manual* mode is automatically set. The additional optional argument `axis` can be used to set only the x- or the y-axis: in this case, the length of the range vector must be equal to 2, instead of 4, and the `axis` argument set to the single character `"x"` or `"y"`.

The `axis` optional argument can also be used when the following modes are set: *auto*, *tight* and *manual*.

Axes may be inverted (*e. g.*, for the x-axis, the maximum value on the left and the minimum value on the right); to do this, simply give a range in decreasing order.

See also: `msAxisFontSize`, `msAxisLineWidth`, `msLabelFontSize`, `msGrid`

`msAxisFontSize` — axis font size setting

Interface:

```
subroutine msAxisFontSize( size_factor )

   real(kind=MF_DOUBLE), intent(in) :: size_factor
```

Description:
This routine is used to change the font size of the axis numbering.

`size_factor` must be ranged in $[0.4, 2.5]$. Default size factor is unity. Actual font size also depends on the `char_height_pixel` mode (see the `msCharInPixels` routine).

See also: `msAxis`, `msTitleFontSize`, `msLabelFontSize`, `msAxisLineWidth`

`msAxisLineWidth` — axis line width setting

Interface:

```
subroutine msAxisLineWidth( width )

   real(kind=MF_DOUBLE), intent(in) :: width
```

Description:
This routine is used to change the line width of the axis and the associated numbering.

Default width is unity.

Remark: If you increase the value of `width`, you will remark that the effective line width in pixels is approximately half of this latter value. Actually, the pixel width is a discontinuous function of `width`, especially at small values. The corresponding behaviour is however continuous for the drawing generated in EPS and PDF files.

See also: `msAxis`, `msAxisFontSize`, `msLabelFontSize`

`msCharInPixels` — character size policy

Interface:

```
subroutine msCharInPixels( mode )

   character(len=*), intent(in) :: mode
```

Description:
If `mode` is equal to `"off"`, the character size is proportional to the window size (default behavior).

If `mode` is equal to `"on"`, the character size is absolute. Unity is about 12 pixels.

See also: `msAxisFontSize`, `msText`

`mf/msCAxis` get or set color axis

The first form:

```
function mfCAxis() result ( out )

   type(mfArray) :: out
```

returns the color axis of the current figure. It is a 2-value vector which contains the min and max values used in the colorbar. The default value (if not set by the user) is the range `[0,1]`.

The subroutine form:

```
call msCAxis( col_range | mf_range | string )
```

is used to set the color axis. One argument is required, it is either a `real` 2-value vector, an `mfArray` or a character string.

When the argument contains numerical values, it specifies the color range. Be aware that the two values must not be too close together: in such a case, the routine enlarges slightly the given interval and emits a warning. Once the color axis has been defined, it will remain the same until the next call.

When the argument is a string, it must be equal to `"reset"`: the color axis takes the initial `[0,1]` range, whereas it becomes extensible again, when used repeatedly (by successive calls of `msContour`, for example).

Remark: The two input numerical values defining the color axis must be ordered: first is the minimum, second is the maximum.

See also: `msColorbar`, `msColormap`, `msColormapSize`, `msPColor`, `msContour`, `msPatch`

`msClf` clear current figure

Calling syntax:

```
call msClf()
```

Description:
Clears the current figure, but don't close the corresponding window.

See also: `msFigure`, `msClose`, `msCla`

`msColorbar` color bar setting

Calling syntax:

```
call msColorbar( mode [, position ] [, label ] )
```

Description:
If `mode` is equal to `"on"`, adds a colorbar to the current figure.

Its position can be specified by an optional argument, `position`, which can be `"vert"` (*i. e.* right side) or `"horiz"` (*i. e.* bottom side); by default the position is automatically chosen.

The optional argument `label` can be used for the labelling of the color bar.

The three arguments are of type `character(len=*)`.

Remarks:

- In some situations, in particular when axis scaling are equal (see `msAxis`) and one figure side is much smaller than the other, it is recommended to left the routine choose itself the position; otherwise the colorbar may have a too short length.
- It is recommended to call this routine after drawing all objects.

See also: `msCAxis`, `msColormap`, `msColormapSize`

`mf/msColormap` — get or set colormap

The function form:

```
colormap = mfColormap()
```

returns the current colormap in an `mfArray` (rank-2 array of 3 columns) which contains the Red-Green-Blue components of each color. These components are reals, ranged from 0.0 to 1.0.

The other function form:

```
bool = mfColormap( "init_status" )
```

returns a logical value according to the initialization status of the colormap.

The subroutine form:

```
call msColormap( name [, "inverted"] )
```

or

```
call msColormap( colormap [, int_max] )
```

is used to set the colormap.

The user can choose a predefined colormap, via the `character name` argument: `"rainbow"` (or its alias `"jet"`), `"parula"`, `"hot"`, `"bluered"`, `"fusion"`, `"flag"` or `"grey"` (a grey scale). Number of colors is 256, but it can be changed via `msColormapSize`. You can see their color representation in the *Muesli User's Guide*. If the optional argument `"inverted"` is added, then the colormap is inverted.

User-defined colormap can also be registered, via the `mfArray colormap`. This array must have 3 columns and a number of rows up to 4096.

By default, `colormap` contains real color components, ranged from 0.0 to 1.0. If the optional argument `int_max` is present, it indicates that entries are integers whose max value is provided (*e. g.* 255 for 256 values for each R,G,B component, giving a subset among 16 millions of colors. Integer color components always start from 0).

Remarks:

- Before using this routine, a window must be opened via `msFigure`.
- There is no default colormap. The user must select one of them before using colors.

See also: `msCAxis`, `msColorbar`

`mf/msColormapSize` — get or set colormap size

The function form:

```
size = mfColormapSize()
```

returns the size (integer) of the current colormap.

The subroutine forms:

```
call msColormapSize( size )
```

is used to set the size of the colormap. A maximum of 4096 colors is allowed; note however that in most of case, 256 colors are sufficient to eliminate color artifacts on the screen.

```
call msColormapSize( "auto" )
```

sets the size of the colormap to its default size, *i. e.* 256.

See also: `mf/msColormap`, `msColorbar`

`msDrawBox` redraw the box in the current figure

Calling syntax:

```
call msDrawBox()
```

Redraws the framed box in the current figure.

This routine is useful after drawing some plots which partially erase the ticks and marks of the box. It doesn't redraw the numerical label neither the optional grid (which should remain in the background).

See also: `msRedrawFigure`, `msDrawGrid`

`msDrawGrid` if needed, redraw the grid in the current figure

Calling syntax:

```
call msDrawGrid()
```

Redraws the grid in the current figure, if needed.

This routine is useful during animation: if the grid has been set with the `msGrid` routine, the current routine must be called in the iteration loop, before any plotting with FGL.

See the *Muesli User's Guide* about a typical sequence of Muesli routines to include in an animation loop.

See also: `msRedrawFigure`, `msDrawBox`

`msRedrawFigure` — redraw the current figure

Calling syntax:

```
call msRedrawFigure( )
```

redraws the current figure.

Typically, this routine should be used after any graphic object change, modified by the routine `msSetGrObj`, in order to make visible the modification on the screen.

See also: `msDrawBox`, `msDrawGrid`, `msResizeFigure`

`msGetX11Pixmap` X11 pixmap saving

Calling syntax:

```
call msGetX11Pixmap( A1, A2, A3 )
```

Description:
This routine returns in three `mfArrays`, the color components of the X11 pixmap drawn on the screen.

If the background color is black, then it gets the R,G,B planes.
If the background color is white, then it gets the C,M,Y planes.

Remark: the `mfArray` arguments should not be pointed by another ordinary Fortran pointer (see `msPointer`); on the contrary, a Warning is emitted.

See also: `msSetBackgroundColor`

`msGrid` — grid setting

Calling syntax:

```
call msGrid( mode [, minor ] )
```

The string `mode` may have the following values:

- `"on"`: a background grey dashed grid is drawn for major axis ticks; (default)
- `"off"`: grid is hidden;

The boolean optional `minor` can be set to `.true.` to get minor lines (aligned on minor ticks). By default, these minor dotted lines are not drawn.

See also: `msAxis`

`msHold` — plot command superposition

Calling syntax:

```
call msHold( "on" | "off" )
```

Description:
Usually, each new plot command erase the previous one. If you want to keep the previous graphic objects drawn in the figure, you must call this routine with `"on"`.

Use `"off"` to come back to the default behavior.

Remark: to avoid unwanted behavior, place this command after the first plot (`msPlot` or whatever).

`msRemoveClipBox` — remove a clipping box inside axes

Interface:

```
subroutine msRemoveClipBox( )
```

Description:
Remove the clipping box set by the `msSetClipBox` routine.

See also: `msSetClipBox`

`msSetClipBox` define a clipping box inside axes

Interface:

```
subroutine msSetClipBox( dp_range )

   real(kind=MF_DOUBLE), intent(in) :: dp_range(:)
```

Description:
Define a clipping box (inside current axes) for subsequent plotting commands.

See also: `msRemoveClipBox`

`msShading` color shading mode

Calling syntax:

```
call msShading( mode )
```

where `mode` is a character string which can be `"flat"` or `"interp"`.

Description:
Sets the shading mode used when drawing patches:

- `"flat"` implies that only one color is used;
- `"interp"` leads to interpolated colors between the vertices of the polygonal shape.

`"flat"` is the default mode.

Note that the `"interp"` mode doesn't lead to bigger files when printing in PDF. Only for EPS, the use of transparency leads to the inclusion of bitmap images.

The shading has no effect on the display of the colorbar. Indeed, the colormap may have sharp gradients or even jumps in colors, or may use of a very small number of colors (*e. g.* 8 or 16): that must appear clearly in the colorbar.

See also: `mf/msPatch`, `mf/msPColor`, `msColorbar`

`mfGetXAxisTicksNb` get ticks number of the X-axis

Calling syntax:

```
nb = mfGetXAxisTicksNb( "major" | "all" )
```

returns the number of ticks of the X-axis (linear scaling only). This can be the number of major ticks (the default, or if `"major"` is used as argument) or all ticks (both major and minor) when `"all"` is used as argument.

Usually, this routine is used in conjunction with a call to `msSetXAxisUserLabels`.

`msSetXAxisUserLabels` set customized ticks labels of the X-axis

First calling syntax:

```
call msSetXAxisUserLabels( labels )
```

This call must occur only after the function `mfGetXAxisTicksNb`, which returns the number N of ticks of the (linear) X-axis.

`labels` must be an array (of size N) of character strings. The ticks labels (major or all, according the previous call) are then replaced by each string in `labels`. Only the first 6 characters are used for each label (truncation beyond).

Remark: Using customized labels can be useful for a Bar plot (see `msBar`), either because numerics are not appropriate, or because you want to use a logarithmic scaling when it is not allowed.

Other calling syntax:

```
call msSetXAxisUserLabels( .false. )
```

This later call is used to manually disable the replacement of the labels. Note that this replacement is also automatically disabled when the range of the X-axis is modified, either by the user (explicit call to `msAxis`) or by a plot of other graphic objects which would modify this range.

`mf/msTitle` **title setting**

Interface:

```
subroutine msTitle( title )

   character(len=*), intent(in) :: title
```

Description:
Adds a title to the current figure. Not drawn if *axis* is *off*.

Maximum length of the title is 128 characters.

This routine should be called after the other data plotting routines.

Escaped sequences may be use in the string (see `msText`) to change the font, the style, the position (sub- or superscript), etc. Note also that the global character size may be changed by using `msTitleFontSize`.

Remark: the function `mfTitle` has the same argument list; it returns the (integer) handle of the created graphic objet; this is useful to create many strings in different languages, and set them as optional content for PDF creation (see `msSetPdfOC`).

See also: `msXLabel`, `msYLabel`, `msAxis`

`msTitleFontSize` — title font size setting

Interface:

```
subroutine msTitleFontSize( size_factor )

   real(kind=MF_DOUBLE), intent(in) :: size_factor
```

Description:
This routine is used to change the font size of the title.

`size_factor` must be ranged in $[0.5, 3.5]$. Default size factor is 2. Actual font size also depends on the `char_height_pixel` mode (see the `msCharInPixels` routine).

See also: `msAxis`, `msAxisFontSize`, `msTitle`, `msLabelFontSize`

`mf/msXLabel` x-label setting

Interface:

```
subroutine msXLabel( xlabel )

   character(len=*), intent(in) :: xlabel
```

Description:
Adds a label to the x-axis of the current figure. Not drawn if *axis* is *off*.

Maximum length of this label is 96 characters.

This routine should be called after the other data plotting routines.

Escaped sequences may be use in the string (see `msText`) to change the font, the style, the position (sub- or superscript), etc. Note also that the global character size may be changed by using `msLabelFontSize`.

Remark: the function `mfXLabel` has the same argument list; it returns the (integer) handle of the created graphic objet; this is useful to create many strings in different languages, and set them as optional content for PDF creation (see `msSetPdfOC`).

See also: `msTitle`, `msYLabel`, `msAxis`

`mf/msYLabel` y-label setting

Interface:

```
subroutine msYLabel( ylabel )

   character(len=*), intent(in) :: ylabel
```

Description:
Adds a label to the y-axis of the current figure. Not drawn if *axis* is *off*.

Maximum length of this label is 96 characters.

This routine should be called after the other data plotting routines.

Escaped sequences may be use in the string (see `msText`) to change the font, the style, the position (sub- or superscript), etc. Note also that the global character size may be changed by using `msLabelFontSize`.

Remark: the function `mfYLabel` has the same argument list; it returns the (integer) handle of the created graphic objet; this is useful to create many strings in different languages, and set them as optional content for PDF creation (see `msSetPdfOC`).

See also: `msTitle`, `msXLabel`, `msAxis`

`msLabelFontSize` descriptive labels font size setting

Interface:

```
subroutine msLabelFontSize( size_factor )

   real(kind=MF_DOUBLE), intent(in) :: size_factor
```

Description:
This routine is used to change the font size of the axis labels.

`size_factor` must be ranged in $[0.4, 2.5]$. Default size factor is unity. Actual font size also depends on the `char_height_pixel` mode (see the `msCharInPixels` routine).

See also: `msAxis`, `msAxisFontSize`, `msTitleFontSize`, `msXLabel`, `msYLabel`

`msAxisLabelFormat` — kind of labelling for Axis

Interface:

```
subroutine msAxisLabelFormat( x_axis_mode, y_axis_mode )

   character(len=*), intent(in), optional :: x_axis_mode, y_axis_mode
```

Description:
This routine is used to set the format of labelling axis.

Default mode is `"std"`, *i. e.* numeric numbers without unit. If set to `"time"` then numbers are supposed to be seconds and are written under the format (DD) HH MM SS.S.

See also: `msAxisFontSize`

`msSetWinProp` — window properties setting

Interface:

```
call msSetWinProp( property, data )
```

Description: Used to modify a specified property of the current window.

`property` may take the following values:

- `"axis_font_size"`, `"axis_line_width"`, `"label_font_size"` or `"title_font_size"`:
  → `data` of type `real`
- `"xlabel"`, `"ylabel"` or `"title"`:
  → `data` of type `character(len=*)`

Remark: Contrary to the specific following routines: `msAxisFontSize`, `msAxisLineWidth`, `msLabelFontSize`, `msTitleFontSize`, `msXLabel`, `msYLabel`, `msTitle`, the redraw of the figure is not done; therefore, the user must call the `msRedrawFigure` to see the change effects.

`msLegend` — add legends to curves, in one frame

First interface:

```
subroutine msLegend( legend_1 [, legend_2, legend_3, ... ]                &
                     [, location | position ] )

   character(len=*), intent(in)           :: legend_1
   character(len=*), intent(in), optional :: legend_2, ..., legend_13
   character(len=2), intent(in), optional :: location
   type(mfArray),    intent(in), optional :: position
```

Second interface:

```
subroutine msLegend( legend_1, handle_1 [, legend_2, handle_2, ... ]       &
                     [, location | position ] )

   character(len=*), intent(in)           :: legend_1
   integer,          intent(in)           :: handle_1
   character(len=*), intent(in), optional :: legend_2, ..., legend_13
   integer,          intent(in), optional :: handle_2, ..., handle_13
   character(len=2), intent(in), optional :: location
   type(mfArray),    intent(in), optional :: position
```

Description:
Adds legends to the curves plotted in the figure. Up to 13 legends (noted above as `legend_1`, `legend_2`, ...) in one frame are supported, each containing a string of at most 80 characters.

The handles of the curves may be provided (see the second interface), making possible to legend only a subset of the curves. Of course, all handled must be valid and must be integers returned by *Muesli* graphic functions. Note that the two calling syntaxes above cannot be mixed.

Numbered legends and handles must be used in increasing order, otherwise some legends may not appear.

The optional argument `location` (not to be used in conjonction with `position`) can be used to specify in which corner of the figure the legend is displayed. Possible values are `"TL"` (top-left), `"TR"` (top-right), `"BL"` (bottom-left), `"BR"` (bottom-right) or `"outside"` (see below). Default is top-left corner.

The optional argument `position` (not to be used in conjonction with `location`) specify the position of the legend's frame, *i. e.* the coordinates of the top-left angle of the frame. The user is even allowed to locate the legend "outside" of the axes, by specifying any value of the coordinates. The legend frame may or may not be entirely visible in the *X11* window, but it should appear in the EPS and PDF files. This is useful in case of a very high legend. A better way, however, is to use `location="outside"`; in such a case, the legend will be drawn in a small, additional window (but the legend will appear in the same EPS or PDF after printing).

Legends are grouped in a unique frame which can be moved inside or outside axes via the `msMoveLegend` routine. See `mfLegend` to create many frames. A legend frame cannot be removed without knowing its handle, *i. e.* via `mfLegend`.

Remarks:

– Usually, the number of string arguments (or pairs of string and associated handle) specifying the legends is equal to the current number of curves. However, it is possible to specify less legends than the actual number of curves: a simple warning will be emitted during the call. Similarly, specifying a number of legends larger than the number of current curves leads to a simple warning, and additional legends will be discarded.

· · · / · · ·

– For bar plot, the different items in the legend correspond to the data series plotted; this works only for *grouped* or *stacked bars* (see `mf/msBar`). Currently, the second interface above is not compatible with bar plots.

Third interface:

```
subroutine msLegend( legend_array [, handle_array ]                    &
                     [, location | position ] )

   character(len=*), intent(in)           :: legend_array(:)
   integer,          intent(in), optional :: handle_array(:)
   character(len=2), intent(in), optional :: location
   type(mfArray),    intent(in), optional :: position
```

Description:
Allows the user to group all the legends (and, optionally, the associated handles) in one array of character strings.

`mfLegend` add legends to curves, possibly in many frames

Description:
As opposed to most of graphic routines, the function version of the `Legend` family behaves differently.

It has the same interfaces as `msLegend` and, as usual, returns the (integer) handle of the graphic objet; moreover, it can be called many times for the same figure, making it useful to create legends in different languages, and to set them as optional content for PDF creation (see `msSetPdfOC`).

2.4 Figure annotation – Low level graphic object's manipulation

`mf/msArrow`	arrow drawing
`mf/msText`	text display
`mfGetAllGrObj`	get all graphic objects
`mfGetTypeGrObj`	get the type of a graphic object
`mfSelectTypeGrObj`	select graphic objects by type
`msSetGrObj`	graphic object setting
`msRemoveGrObj`	graphic object deletion

See also:

Global graphic settings

Window's and figure's management

Figure properties

High level plotting routines

Interactive routines

`mf/msArrow` — arrow drawing

The first form:

```
call msArrow( x_start, y_start, x_end, y_end                                &
              [, color] [, linewidth] [, headsize] [, clipping ] )
```

draws an arrow from the position (`x_start`, `y_start`) to the position (`x_end`, `y_end`). These coordinates must be of type `real`.

The optional argument `color` have the same meaning as in the routine `mf/msPlot`.

Optional arguments `linewidth` and `headsize` (real numbers) can be used to specify the line width and the head size of the arrow. Default values are unity.

If the optional boolean argument `clipping` is set to *FALSE*, then the arrow can be displayed out of the viewport (default is TRUE, *i. e.* the arrow is clipped at the viewport; this default behavior is recommended when it is an annotation at some location inside the viewport and you might want to pan or zoom inside the axes).

The second form:

```
call msArrow( x, y, angle                                                   &
              [, color] [, headsize] )
```

draws only an arrow head, given a position (x, y) and an angle in radian (all are real variables).

The remaining arguments have the same meanings as above.

Remarks:

- the function `mfArrow` has the same argument list; it returns the (integer) handle of the created graphic object.
- in future, the four first arguments will be able to take also the type `mfArray`, in order to display many arrow at a time.

See also: `mf/msText`

`mf/msText` — text display

Interface:

```
call msText( x, y, text                                                    &
             [, angle] [, just] [, just_vert] [, pix_voffset]              &
             [, color] [, bg] [, height] [, clipping] [, xbox, ybox] )
```

Description:
Writes the string `text` at the position (`x`, `y`). These coordinates may be of type `real` or `mfArray`. Many escape codes allow the use of different kinds of fonts, the use of subscript or superscript, greek letters, *etc.* See the *Muesli User's Guide* at section 5.5 for more information.

If the real argument `angle` is present then text is written with this angle (in degrees).

The real argument `just` specifies the justification of the string in comparison with the position (-1 for left (default), 0 for centered, $+1$ for right — other real values ranged from -1 to $+1$ are permitted, giving intermediate justifications). Similarly, the `real just_vert` argument concerns the vertical justification (default is -1, *i. e.* the baseline of the glyph located at `y` coordinate, and 0 to obtain the topline of the glyph at `y`; use -0.5 for a vertically centered glyph). Concerning only the vertical justification, the optional boolean argument `pix_voffset` can be used (under X11) to move vertically the string box by a 2 pixel shift — this is especially useful when the bottom or the top of the string box is against a drawn line (default is `pix_voffset=.false.`).

If the optional argument `color` is present, it specifies the color of the text displayed (see `msSetGrObj`). Default color is black when the figure background is white (see `msSetBackgroundColor`).

If the optional argument `bg` is present, it specifies the color of the background (see `msSetGrObj`). Default is transparent.

The optional real argument `height` specifies the height of the characters drawn (default is 1). This height may be relative (the default, character size is proportional to the window size) or absolute (character height is in pixels, unity is about 12 pixels), according to the mode defined in the routine `msCharInPixels`.

If the optional boolean argument `clipping` is set to *FALSE*, then `text` can be displayed out of the viewport (default is TRUE, *i. e.* the text is clipped at the viewport; this default behavior is recommended when `text` is an annotation at some location inside the viewport and you might want to pan or zoom inside the axes).

The optional output arguments `xbox` and `ybox` allows the user to retrieve the value of the rectangle frame of the string displayed (4 values for both x and y, because the string may be inclined). These arguments must be real arrays of length 4.

Remarks:

- usually, the current routine is used to annotate a graphic, so it should be employed after any use of `mf/msPlot`; on the contrary, the `msHold` routine must be added before the plot command, else all texts from `mf/msText` will be erased.
- the function `mfText` has the same argument list; it returns the (integer) handle of the created graphic object.

See also: `mf/msArrow`, `msSetCharEncoding`

`mfGetAllGrObj` — get all graphic objects

Calling syntaxe:

```
hdle_vec = mfGetAllGrObj( )
```

Description: Returns a vector of handle for all graphic objects in the current window.

The `hdle_vec` variable, declared by the user, must be an integer vector. For an easy programming use, it should be allocatable: due to a feature of modern Fortran, it will be allocate with the appropriate size during the assignment; moreover, there is no need to deallocate it between calls.

See also: `mfGetTypeGrObj`, `msSetGrObj`

`mfGetTypeGrObj` get the type of a graphic object

Calling syntaxe:

```
type = mfGetTypeGrObj( handle )
```

Description: Returns the type of a graphic object, identified by its handle, as a character string.

The type may be one of the followings:

- `"line"`, `"point"` or `"line+point"`, typically created by `mf/msPlot`;
- `"image"`, created by `mf/msImage`;
- `"text"`, created by `mf/msText`;
- `"polygon"`, created by `mf/msPatch`;
- `"pcolor"`, created by `mf/msPColor`;
- `"contour"`, created by `mf/msContour`;
- `"quiver"`, created by `mf/msQuiver`;
- `"streamline"`, created by `mf/msStreamline` or `mf/msTriStreamline`;
- `"arrow"` or `"arrow_head_only"`, created by `mf/msArrow`;
- `"quadr_bezier"`, created by `mf/msPlotQuadrBezier`;
- `"cubic_bezier"`, created by `mf/msPlotCubicBezier` or `mf/msPlotCubicSpline`;
- `"histogram"`, created by `mf/msPlotHist`;
- `"errorbar_x_line"`, `"errorbar_y_line"`, `"errorbar_xy_line"`, `"errorbar_x_pt"`, `"errorbar_y_pt"`, `"errorbar_xy_pt"`, `"errorbar_x_line+pt"`, `"errorbar_y_line+pt"` or `"errorbar_xy_line+pt"`, created by `mf/msErrorBar`;
- `"pcolor_spy"`, `"pcolor_spy_sparse"`, `"pcolor_spy_sparse_2"`, `"plot_spy_sparse"`, created by `mf/msSpy`;
- `"legend"`, created by `mf/msLegend`;
- `"tri_fill"`, created by `mf/msTriFill`;
- `"trimesh_by_fac"`, `"trimesh_by_tri"` or `"mesh_boundary_unstruct"`, created by `mf/msTriMesh`;
- `"tri_pcolor"`, created by `mf/msTriPColor`;
- `"tri_quiver"`, created by `mf/msTriQuiver`;
- `"tri_contour"`, created by `mf/msTriContour`;
- `"xlabel"`, created by `mf/msXLabel`;
- `"ylabel"`, created by `mf/msYLabel`;
- `"title"`, created by `mf/msTitle`;
- `"set_clip_box"`, created by `msSetClipBox`;
- `"remove_clip_box"`, created by `msRemoveClipBox`.

See also: `mfGetAllGrObj`, `msSetGrObj`

`mfSelectTypeGrObj` — select graphic objects by type

Calling syntaxe:

```
hdle_vec = mfSelectTypeGrObj( handles, grobj_type )
```

Description: Returns a vector of handles for the graphic objects which match the type specified as argument, among a list of input handles.

`handles` must be a integer vector, containing a set of valid graphic object handles.

`grobj_type` is a character string, containing the type of the graphic object, as returned by `mfGetTypeGrObj`.

The `hdle_vec` variable, declared by the user, must be an integer vector. For an easy programming use, it should be allocatable: due to a feature of modern Fortran, it will be allocate with the appropriate size during the assignment; moreover, there is no need to deallocate it between calls.

See also: `mfGetAllGrObj`, `msSetGrObj`

msSetGrObj — graphic object setting

First interface:

```
call msSetGrObj( handle, property, data )
```

Description: Changes one property of the graphic object identified by its `handle`. This handle may be a scalar or vector `integer`, or even a vector of handles stored in an `mfArray`.

The type of `data` depends on the target `property`, as detailed below.

`property` may take the following values:

- `"visible"`: `data` may be any string, but the object will be displayed only if `data` is `"on"`; an exception occurs if the object is tagged as *Optional Content* for the PDF driver (see msSetPdfOC).
- `"linestyle"`: `data` must be a string among `"-"`, `"--"`, `"-."` (or `".-"` as an alias) or `":"` (or `".."` as an alias), for continuous, dashed, dashed-dotted, and dotted, respectively.
- `"linewidth"`: `data` must be of type `real`, positive (default value is 1).
- `"cap_style"`: `data` must be a string among `"CapButt"`, `"CapRound"` or `"CapProjecting"` (but not case sensitive). Signification of these three keywords may be found easily on the web and concerns all devices (*X11*, *EPS* and *PDF*). Default is `"CapRound"`.
- `"join_style"`: `data` must be a string among `"JoinMiter"`, `"JoinRound"` or `"JoinBevel"` (but not case sensitive). Signification of these three keywords may be found easily on the web and concerns all devices (*X11*, *EPS* and *PDF*). Default is `"JoinRound"`.
- `"marker"`: `data` may be a single character among `"."`, `"+"`, `"*"`, `"o"`, `"x"`, `"s"`, `"^"` or `"d"` (for Dot, Plus, Asterisk, Circle, X, Square, TriangleUp and Diamond, respectively), or an escaped sequence as `"\M01"` to `"\M26"` for using the complete set of markers, described in Fig. 27 of the *Muesli User's Guide*. Negative codes `"\M-14"` to `"\M-21"` tell Muesli to draw the 8 filled markers with a small white border around them. For a more comprehensive reference, the markers may be also specified by an escaped text sequence, as `"\CircleFilled"` (the character case doesn't matter; it can be adapted for a better reading).
- `"markersize"`: `data` must be of type `real`, positive (default value is 1).
- `"color"`: `data` may be a single character, among `"w"`, `"k"`, `"r"`, `"g"`, `"b"`, `"c"`, `"m"` or `"y"` (for white, black, red, green, blue, cyan, magenta and yellow, respectively), or an escaped sequence of the form `"\C`*nn*`"` (*e. g.* `"\C01"`, `"\C02"`, ...) for using the corresponding color of the current color scheme (see msSetColorScheme).
- `"col_name"`: `data` may be any string which represents a valid colorname from the X11 RGB database.
- `"col_rgb"`: `data` must be a vector of type `real` and size 3.
- `"text"`: `data` may be any string.
- `"position"`: `data` must be a string among `"first"`, `"last"`, `"up"` or `"down"`. This affects the position in the stack, *i. e.* the drawing order.
- `"opacity"`: `data` must be of type `real`, ranged from 0.0 (full transparency) to 1.0 (full opacity).
- `"coordinates"`: (for text string) `data` must be of type `real` and size 2.
- `"relat_coords"`: (for a legend frame created by mfLegend) `data` must be of type `real` and size 2, and is understood as relative coordinates (left and bottom sides correspond to 0, right and top sides correspond to 1).
- `"EPS_user_comment"` (not case sensitive): `data` can be any character string. This text will be add only in the EPS file in order to find easily the PostScript commands related to the graphic object.

···/···

Changing the coordinates of the points in a polyline needs to change both x and y coordinates together by using the *Other interface*:

```
call msSetGrObj( handle, "x", x, "y", y )
```

Remarks:

- the user must call the `msRedrawFigure` to see the change effects.
- properties are not all editable for a given graphic object. The following table describes what can be modified for all types of graphic objects.

See also: `msSetWinProp`, `msRemoveGrObj`

`msRemoveGrObj` graphic object deletion

Calling syntax:

```
call msRemoveGrObj( handle [, redraw ] )
```

Description:
Removes the graphic object in the current figure, specified by its handle, and redraws the figure.

`handle` may be a scalar or a vector of handle (always integer); it can be also an `mfArray` containing integer values of handles.

If the optional argument `redraw` is used and set to *FALSE*, the figure is not redrawn. This could be justified when you have to remove a lot of graphic objects. Don't forget to specify `redraw` = *TRUE* on the last call, or call the `msRedrawFigure` to redraw the whole figure.

Remark: After the grobj removing, the handles are set to zero. In the case of an `mfArray`, all elements are set to zero but the `mfArray` itself is not released.

See also: `msSetWinProp`, `msSetGrObj`

2.5 High level plotting routines - 2D

mf/msPlot	data plot (using straight segments)
mf/msErrorBar	data plot with error bars
mf/msPlotQuadrBezier	data plot (using quadratic curved segments)
mf/msPlotCubicBezier	data plot (using cubic curved segments)
mf/msPlotCubicSpline	data plot (using cubic curved segments)
mf/msBar	bar data plot
mf/msPColor	pseudo-color plot
mf/msContour	data contouring
mf/msContourF	data contouring with filled regions
mf/msQuiver	vector field plot
mf/msStreamline	vector field streamline
mf/msPatch	graphic patch
mf/msPlotHist	data histogram
msCumulHist	cumulative data histogram
msImRead, msImWrite	image read and write
mf/msImage	image display
mf/msPlotPSLG	plot a PSLG domain
mf/msTriMesh	plot a triangulation
mf/msPlotVoronoi	plot a Voronoi diagram
mf/msTriPColor	pseudo-color on a triangulation
mf/msTriFill	coloring on a triangulation
mf/msTriContour	data contouring on triangles
mf/msTriContourF	data contouring on triangles with filled regions
mf/msTriQuiver	vector field plot on a triangulation
mf/msTriStreamline	vector field streamline on a triangulation
mf/msSpy	sparsity pattern visualization

See also:

Global graphic settings

Window's and figure's management

Figure properties

Figure annotation – Low level graphic object's manipulation

Interactive routines

`mf/msPlot` data plot (using straight segments)

Calling syntax:

```
call msPlot( x [, y]                                                    &
             [, linespec] [, color] [, linewidth] [, markersize]        &
             [, dashes_inverted] )
```

Description:
Plots the vector `mfArray y` versus the vector `mfArray x`.

If `y` is not present then `x` is plotted against its integer indices. One exception is when `x` is complex: in such a case, `msPlot(x)` is equivalent to `msPlot(real(x),imag(x))`, so the drawing is done in the complex plane.

The optional argument `linespec` is a string which contains a multiple attribute vector (color, linestyle, marker) as in MATLAB. See `msSetGrObj` for further information.

The optional argument `color` may be a character string (one-letter color code, escaped sequence, color-name from the RGB database) or a vector of type `real` and length 3 containing the RGB components of a color.

The optional argument `linewidth` must be a positive number of type `real` (default is 1). Under *X11* the unit is pixel, therefore specifying a value less than 1 makes no difference; however, in *EPS* and *PDF*, this will lead to very thin lines.

The optional argument `markersize` must be a positive number of type `real`; it is used only for marker symbols. If this argument is not present, then the marker size is automatically chosen to be coherent with the line width value.

The optional argument `dashes_inverted` must be a boolean; it is used only for dashed lines: it shifts the dashes in such a way that two identical dashed lines using different colors are both visible, when this argument is used for the second curve.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

Log representation of the data is made by using the `msAxis` routine.

Remarks:

- the drawing is always clipped at the viewport.
- the function `mfPlot` has the same argument list; it returns the (integer) handle of the created graphic object.
- when calling many time this routine in the same figure, without providing any color information, there is an automatic color selection, cycling the colors from the current colortable (for more information about colortables – or color schemes – see the *Muesli User's Guide*).
- by default, the axis range presents its minimum on the left and its maximum on the right. To invert the axis, use the `msAxis` routine.

See also: `mf/msBar`, `mf/msPlotQuadrBezier`, `mf/msPlotCubicBezier`, `mf/msPlotCubicSpline`

mf/msErrorBar — data plot with error bars

Calling syntax:

```
call msErrorBar( [ x, ] y [, x_err] [, y_err]                              &
                 [, linespec] [, color] [, linewidth] [, markersize] )
```

Description:
Plots the `mfArray y` versus the `mfArray x`, using line specification given in `linespec` and adding error bars at each point.

The argument `x` is optional (the x-axis will use a vector of sequential integers from 1). `x` and `y` must be vectors (row or col) of same length, not matrices.

The optional argument `linespec` (character string)) has exactly the same meaning as in the msPlot routines. In particular, when not specified, colors will be recycled from the current colortable (for more information about colortables – or color schemes – see the *Muesli User's Guide*).

The optional arguments `linewidth`, `color` and `markersize` have exactly the same meaning as in the msPlot routines.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the msHold routine.

Remark:

- the drawing is always clipped at the viewport.
- the function `mfErrorBar` has the same argument list; it returns the (integer) handle of the created graphic object.
- currently, axis must be both linear.
- due to the simple implementation and the number of optional arguments, it is required, most of times, to use the name of each argument as keywords (on the contrary, errors will occur at compilation and even at run-time). The examples below show typical calls.

See also: mf/msPlot, msAxis

Example(s):

For adding error bars only along the y-axis, the first call:

```
call msErrorBar( mf([2.00d0,1.50d0,3.00d0,2.50d0]),                        &
                 mf([0.10d0,0.15d0,0.20d0,0.25d0]) )
```

will give at run-time:

```
(MUESLI msErrorBar:) ERROR: bad optional arg combinaison (see doc) !
```

because the first actual argument is associated to `x` and the second one to `y`.

The second one:

```
call msErrorBar(      mf([2.00d0,1.50d0,3.00d0,2.50d0]),                   &
                y_err=mf([0.10d0,0.15d0,0.20d0,0.25d0]) )
```

will not compile:

```
Compilation Error: Missing actual argument for argument 'y'
```

Only this third call is correct:

```
call msErrorBar( y=mf([2.00d0,1.50d0,3.00d0,2.50d0]),                      &
                 y_err=mf([0.10d0,0.15d0,0.20d0,0.25d0]) )
```

`mf/msPlotQuadrBezier` data plot (using quadratic curved segments)

Interface:

```
call msPlotQuadrBezier( x, y                                          &
                        [, linespec] [, color] [, linewidth]          &
                        [, dashes_inverted] )
```

Description:
Draws a quadratic Bézier curve from the control points whose coordinates are given in the `mfArrays` `x` and `y`. The drawn object is a quadratic piecewise polynomial parametric curve. The curve is continuous (but doesn't have necessarily continuous derivatives); therefore, the total number of points in the `x` and `y` `mfArrays` must be of the form: $2k + 1$, where k is the number of segments.

Optional arguments `linespec`, `color`, `linewidth` and `dashes_inverted` have the same meaning as in the routine `msPlot`.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

Remarks:

- the drawing is always clipped at the viewport.
- the marker choice, when specified, is discarded.
- the function `mfPlotQuadrBezier` has the same argument list; it returns the (integer) handle of the created graphic object.
- when not specified, colors will be recycled from the current colortable (for more information about colortables – or color schemes – see the *Muesli User's Guide*).

See also: `mf/msPlotCubicBezier`, `mf/msPlotCubicSpline`

mf/msPlotCubicBezier — data plot (using cubic curved segments)

Interface:

```
call msPlotCubicBezier( x, y                                          &
                        [, linespec] [, color] [, linewidth]          &
                        [, dashes_inverted] )
```

Description:
Draws a cubic Bézier curve from the control points whose coordinates are given in the `mfArrays` `x` and `y`. The drawn object is a cubic piecewise polynomial parametric curve. The curve is continuous (but doesn't have necessarily continuous derivatives); therefore, the total number of points in the `x` and `y` `mfArrays` must be of the form: $3k + 1$, where k is the number of segments.

Optional arguments `linespec`, `color`, `linewidth` and `dashes_inverted` have the same meaning as in the routine `msPlot`.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

Remarks:

- the drawing is always clipped at the viewport.
- the marker choice, when specified, is discarded.
- the function `mfPlotCubicBezier` has the same argument list; it returns the (integer) handle of the created graphic object.
- when not specified, colors will be recycled from the current colortable (for more information about colortables – or color schemes – see the *Muesli User's Guide*).

See also: `mf/msPlotQuadrBezier`, `mf/msPlotCubicSpline`

mf/msPlotCubicSpline — data plot (using cubic curved segments)

Interface:

```
call msPlotCubicSpline( abs_curv, x, y, wx, wy,                          &
                        [, linespec] [, color] [, linewidth]             &
                        [, dashes_inverted] )
```

Description:
Draws a cubic Spline curve from given points whose coordinates are given in the `mfArrays` `x` and `y`. The drawn object is a cubic piecewise polynomial curve parametrized by the vector `abs_curv`. It is C^2 continuous at segment boundaries (*i. e.* both the tangent vector and the curvature are continuous). The two vectors `wx` and `wy` must have been computed before the call by use of the routine `mfSpline`.

Optional arguments `linespec`, `color`, `linewidth` and `dashes_inverted` have the same meaning as in the routine `msPlot`.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

Remarks:

- the drawing is always clipped at the viewport.
- the marker choice, when specified, is discarded.
- the function `mfPlotCubicSpline` has the same argument list; it returns the (integer) handle of the created graphic object.
- when not specified, colors will be recycled from the current colortable (for more information about colortables – or color schemes – see the *Muesli User's Guide*).

See also: `mf/msPlotQuadrBezier`, `mf/msPlotCubicBezier`

mf/msBar — bar data plot

Calling syntax:

```
call msBar( [ x, ] y                                                    &
            [, color] [, width] [, baseline] [, style] )
```

Description:
Plots the `mfArray y` versus the `mfArray x`, using vertical bars. The argument `x` is optional (the x-axis will use a vector of sequential integers from 1). When `y` is a vector (row or col) then `x` must have the same shape as `y`; however, when `y` is a matrix (processed by columns), then `x` must be a column vector of same length as that of `y` columns.

The optional argument `color` specifies the color used to fill the bars; the color convention is the same as in the `msPlot` routine. See also `msSetGrObj` for further information. It may have many types and shapes:

- it may be a single character string, containing a one-letter color code or a color name from the RGB database; it may be also a vector of N character strings, N being the number of columns of `y`.
- it may be a vector of type `real` with 3 RGB components (describing one color); or a matrix of dimension $(3, N)$, N being the number of columns of `y`.

The optional argument `"width"` must be a positive number of type `real` ranged in $[0, 1]$. It is the relative width of the bars (default is 0.8).

The optional argument `"baseline"` must be of type `real`. Its role depends on whether the Y-axis is linear or logarithmic:

- in the case of a linear Y-axis, `baseline` can take any value (default value is zero). The routine plots a thin line to visualize the baseline.
- in the case of a logarithmic Y-axis, `baseline` is the minimum value for data (no default value). Negative values, or those which are less than `baseline` are discarded.

The optional argument `style` works only for matrix data. When it is equal to `"grouped"` (default value), *grouped bars* are used; when it is equal to `"stacked"`, *stacked bars* are used.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

Remarks:

- the function `mfBar` has the same argument list; it returns the (integer) handle of the created graphic object.
- only the Y-axis can be of logarithmic type; in this case, of course, all values in `y` must be positive.
- when not specified, colors will be recycled from the current colortable (for more information about colortables – or color schemes – see the *Muesli User's Guide*).

See also: `mf/msPlot`, `msAxis`

`mf/msPColor` graphic pseudo-color

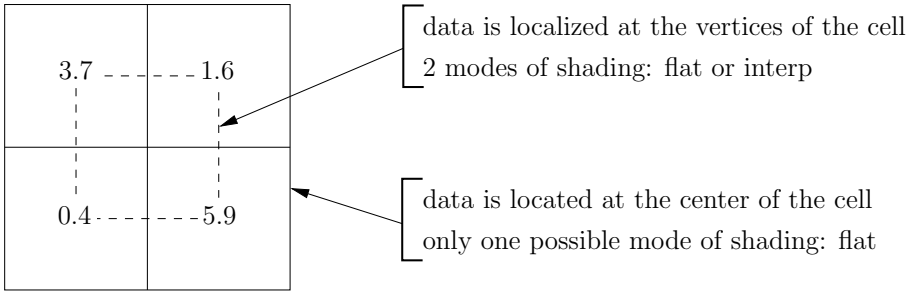
Calling syntax:

```
call msPColor( [ X, Y, ] Z [, data_centering, view]                    &
               [, grid, grid_color, grid_step] [, lighting] )
```

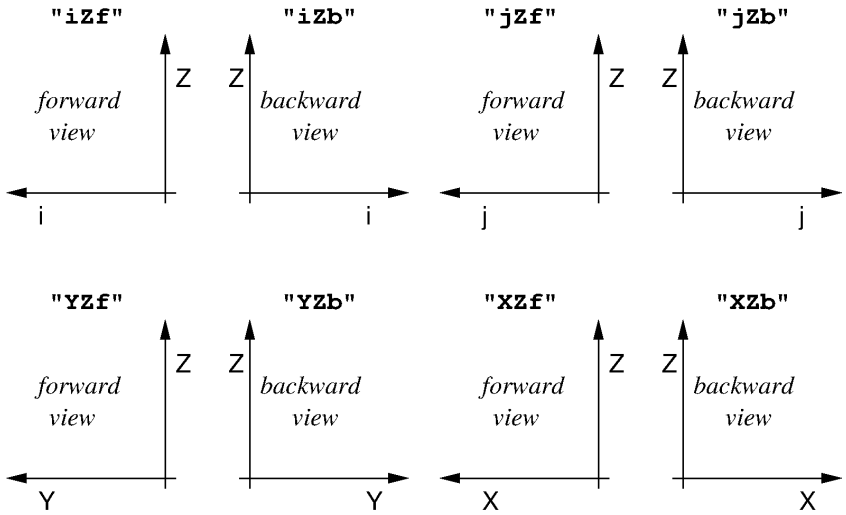
Draws a pseudo color plot of the data contained in the `Z mfArray`.

The (i, j) indices are mapped to the axis $(-Y, X)$, so data visualisation is coherent with the matrix layout and with other routines working with matrix data (`msContour`, `msQuiver`, `msStreamline`). If this orientation is not appropriate, use the coordinates (`mfArrays X` and `Y`), as described below.

The optional `data_centering` argument specifies how the data stored in the matrix are graphically displayed; it is a string which must be equal to `"vertex"` (default value — (i, j) points to the vertex of a cell) or `"cell"` ((i, j) points to the center of a cell). In the cell centered model, the shading of the color is only flat, whatever the value specified by the use of `msShading`. This is summarized in the following figure:



The optional `view` argument is a character string specifying the type of view for displaying the data in the matrix `Z`: the default view is the usual *top view*, noted `"ji"` or `"XY"`, according to the presence of the arguments `X, Y`. When these latter arguments are not present, `view` may be equal to `"iZf"`, `"iZb"`, `"jZf"` or `"jZf"`, specifying a *side view* as noted by the letters `i`, `j` or `Z` (always the vertical axis); the third letter is a modifier, indicating a *forward* or a *backward view*. In the other case, *i. e.* when the arguments `X, Y` are presents, the corresponding strings are `"YZf"`, `"YZb"`, `"XZf"` and `"XZf"`. The different *side views* are summarized below:



Note that, in projection mode, only the default data centering (*i. e.* `data_centering="vertex"`) is accepted.

· · · / · · ·

A thin line may be drawn to show the boundary of each quadrilateral cell by using the optional argument `grid` equal to *TRUE*. The default is to not show these boundaries. The color of this thin line may be specified in the optional argument `grid_color`, with the same convention as in the `msPlot`, routine; by default, it is the foreground color. The optional argument `grid_step` allows to plot only some lines of the grid, indicating the step to be used along the X or the Y coordinate; it must be a vector of two integers. This is particularly useful when the size of the data is large.

The optional argument `lighting` (boolean variable) changes the use of colors: instead of selecting shades from the `Z` values, they come from the lighting of the surface. In this case, it is recommended to select the `grey` colormap from the `msColormap` routine. Note also that this option is applicable only when `X` and `X` coordinates are provided.

When arguments `X` and `Y` are also provided, then their values are used as coordinates for the data points in `Z`:

- usually, `X` and `Y` are matrices and their shape must be the same as that of the data matrix `Z`. Use of these coordinates is required when, *e. g.*, you want to swap the indices i and j, or apply a transformation (linear or not) between the (i, j) indices and the (x, y) coordinates; these coordinate matrices may be created by the `msMeshGrid` routine.
- to economize memory, it is possible to pass only vectors: in this case, `X` and `Y`, provided as rank-1 `mfArrays`, must obey to the same constraint as edicted for the generator vectors (see again `msMeshGrid`).

Be aware that `X` and `Y` are *not used* to defined the axes of the plot: to do so, you must use the `msAxis` routine with appropriate argument(s).

The color axis may be set explicitly by the user via the `msCAxis` routine; otherwise it will be chosen automatically by the routine itself. In the color axis manual mode, data should be theoretically ranged entirely inside the color axis; on the contrary, the resulting behavior depends on the shading value (cf. `msShading`). When shading is equal to `"flat"`, a quiet behavior is obtained and the colormap overflow can be monitored by use of the `msSetColorOverflowPolicy` routine; when shading is equal to `"interp"`, a warning is emitted by the library because strange colors can be drawn. This latter case should be avoided if possible.

Note that a colormap must have been defined before the call, (via `msColormap`); otherwise, a warning is emitted and strange or unexpected results can occur.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

Remarks:

- The drawing is always clipped at the viewport.
- The function `mfPColor` has the same argument list; it returns the (integer) handle of the created graphic object.
- For making a pseudo color plot on a triangle (or a triangular patch), use the `msTriPColor` routine.
- `NaN` values are not (yet) allowed: an error is returned by the routine.

See also: `msPatch`, `msContour`, `msImage`, `msQuiver`

mf/msContour — data contouring

Calling syntax:

```
call msContour( [ mfOut(C), X, Y,] Z                                    &
                [, nb_levels, levels, order, linespec, linewidth,       &
                   labels, labelscolor, labelsize ] )
```

Description:
Draws a contour plot of the data contained in the `mfArray Z`. The plot is done in the current figure. As opposed as `msPColor` routine, `msContour` accepts only vertex-centered data.

It doesn't fill the regions between the contours with colors; for such a feature, see the `mf/msContourF` routine.

(i, j) indices of `Z` are mapped to the axis $(-Y, X)$, so data visualisation is coherent with the matrix layout and with other routines working with matrix data (`msPColor`, `msQuiver`, `msStreamline`). If this orientation is not appropriate, use the coordinates (`mfArrays X` and `Y`), as described below.

When arguments `X` and `Y` are also provided, then their values are used as coordinates for the data points in `Z`:

- usually, `X` and `Y` are matrices and their shape must be the same as that of the data matrix `Z`. Use of these coordinates is required when, *e. g.*, you want to swap the indices i and j, or apply a transformation (linear or not) between the (i, j) indices and the (x, y) coordinates; these coordinate matrices may be created by the `msMeshGrid` routine.
- to economize memory, it is possible to pass only vectors: in this case, `X` and `Y`, provided as rank-1 `mfArrays`, must obey to the same constraint as edicted for the generator vectors (see again `msMeshGrid`).

If the optional `C mfArray` is given in front of `(X,Y)` coordinate matrices, then it will contain on output a two-row matrix storing the contour lines. Each contiguous drawing segment contains the value of the contour, the number of (x, y) drawing pairs, and the pairs themselves. The segment are appended end-to-end as

```
C = [ level_1, x_1, x_2, ..., level_2, x_1, x_2, ...;
      pairs_1, y_1, y_2, ..., pairs_2, y_1, y_2, ...]
```

Be aware that these point coordinates must be interpreted in different ways, according the value of `order` (see below).

The optional (integer) argument `nb_levels` specifies the number of contours to be drawned; it cannot be used together with `levels`.

The optional argument `levels`, which must be a rank-1 `mfArray`, is used to specify the level values; it cannot be used together with `nb_levels`.

By default, the number of level curves is equal to 9, and the level values are equally spaced within the *min* and *max* values of the data, or within thoses of the Color Axis if it has been set before (see `msCAxis`); this can be changed by using the optional arguments `nb_levels` and `levels` (see above). A warning is emitted if a level is outside the Color Axis, excepted when the color is prescribed in `linespec`.

By default, the level curves are labelled (use `labels=.false.` to remove the labels) and the contour curves are colored using the current colormap (use a color inside `linespec` to specify the color of all curves). Labels are displayed in the foreground color (*e. g.* black on white) by default; the user may choose another color via the `labelscolor` optional argument. Moreover the label size may be changed by using the optional `labelsize` real argument, which lets the user to specify a percentage of the standard size (default is `labelsize=1.0d0`).

· · · / · · ·

The optional argument `order` is an integer (1 or 2) specifying the quality of the approximation of the contour lines. Default is `order` = 1: in this case, segments are straight lines. If `order` is set to 2 then a bilinear interpolation is used inside the cells, and segments are constituted by quadratic Bezier arcs. Note that in all cases, the approximation provides only continuity at the cell boundaries, therefore the tangent lines may appear as broken. Note also that the default case (`order` = 1) is well adapted for most situations; only for situations where you have a small number of cells, or if you want to strongly zoom inside a contour map, you would choose `order` = 2.

Optional arguments `linespec` (character string) and `linewidth` (real value) allow the user to change the appearance of the contour curves. Possible values for these arguments are described in `msSetGrObj`, under the `"color"`, `"linestyle"` and `"linewidth"` entries. If a color has been specified, then all contour lines will be drawn with the same color. Be aware, however, that the choice of markers is disabled.

Note that a colormap (see `msColormap`) must have been defined before the call if a specific color is not present in `linespec`; otherwise, a warning is emitted and strange or unexpected results can occur.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

Remarks:

- The drawing is always clipped at the viewport.
- The function `mfContour` has the same argument list; it returns the (integer) handle of the created graphic object.
- For making contours on a triangle (or a triangular patch), use the `msTriContour` routine.
- `NaN` values are not (yet) allowed: an error is returned by the routine.

See also: `mf/msContourF`, `mf/msImage`, `mf/msQuiver`

mf/msContourF — data contouring with filled regions

Calling syntax:

```
call msContourF( [ mfOut(C), X, Y,] Z                                    &
                 [, nb_levels, levels, linewidth,                        &
                    labels, labelscolor, labelsize ] )
```

Description:
Similar to the msContour routine, except that the regions between the contours are filled with appropriate colors.

The whole region whose values are greater than the level is filled with the corresponding color. As a consequence, the region less than all specified levels is left in white. To avoid white regions, it is sufficient to add a level less than or equal the smaller value in the Z array.

Be aware however that some arguments of msContour are not available here.

See also: mf/msTriContourF

`mf/msQuiver` vector field plot

Calling syntax:

```
call msQuiver( [ X, Y,] u, v                                                    &
               [, data_centering] [, color] [, arrow_length] [, step]           &
               [, linewidth] [, arrow_head] )
```

Draws the vector field whose 2D components are stored in the `u` and `v` `mfArrays`, located at position (`X`,`Y`) or whows location is deduced from the layout of `u` and `v` when `X` and `Y` are not present. The plot is done in the current figure.

(i, j) indices are mapped to the axis $(-Y, X)$, so data visualisation is coherent with the matrix layout and with other routines working with matrix data (`msContour`, `msPColor`, `msStreamline`). If this orientation is not appropriate, use the coordinates (`mfArrays` `X` and `Y`), as described below.

When arguments `X` and `Y` are also provided, then their values are used as coordinates for the data points in `u` and `v`:

- usually, `X` and `Y` are matrices and their shape must be the same as those of the data matrices `u` and `v`. Use of these coordinates is required when, *e. g.*, you want to swap the indices i and j, or apply a transformation (linear or not) between the (i, j) indices and the (x, y) coordinates; these coordinate matrices may be created by the `msMeshGrid` routine.
- to economize memory, it is possible to pass only vectors: in this case, `X` and `Y`, provided as rank-1 `mfArrays`, must obey to the same constraint as edicted for the generator vectors (see again `msMeshGrid`).

Be aware that `X` and `Y` are *not used* to defined the axes of the plot: to do so, you must use the `msAxis` routine with appropriate argument(s).

`u` and `v` must be rank-2 (*i. e.* matrices) `mfArrays` of same shape. They represent respectively the X and Y components of the vector field to be drawn. If the orientation or the scaling is not appropriate, use the second form of call described below, by adding coordinates.

The optional `data_centering` argument specifies how the data are stored in the matrix; it is a string which must be equal to `"vertex"` (default value) or `"cell"`.

The optional `color` argument specifies the color used for drawing the arrows: it is a one-letter color symbol or an escaped sequence, as described in the `msSetGrObj` routine.

The optional `arrow_length` argument (real) specifies the maximum length of the arrow field as a multiplicative factor applied to the mesh size; by default its value is 1 so that each arrow is inside the rectangular cell where its center is located. A value much less that 1 is useless, whereas a big value is not recommended.

The optional `step` argument (integer) the step in the loops over the (i, j) indices to display the arrow; the default value is 1. It is useful to avoid a too big number of arrows for high resolution meshes.

The optional `linewidth` argument (real) specifies the line width, as a relative factor; by default its value is 1. The line width is constant and is the same for all arrows, whatever their length is.

The optional `arrow_head` argument (real) specifies the size of the arrow head, based on the character height; by default its value is 1.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

. . . / . . .

Remarks:

- The drawing is always clipped at the viewport.
- An arrow is not drawn if any component of the corresponding vector (`u`, `v`) is equal to *NaN*.
- The function `mfQuiver` has the same argument list; it returns the (integer) handle of the created graphic object.
- For drawing a 2D vector field on a triangular mesh, use the `msTriQuiver` routine.
- When specifying a color, the routine `msColormap` should be used before for definition of the colormap.

See also: `mf/msContour`, `mf/msPColor`

`mf/msStreamline` vector field streamline

Calling syntax:

```
call msStreamline( [ X, Y,] u, v, start [, direction, n_arrow]                &
                   [, color] [, linestyle] [, linewidth] [, arrow_head]       &
                   [, npt_max] [, curv_tol] [, stop_zone] )
```

Based on a velocity vector field whose x- and y-components are respectively `u` and `v` (`mfArrays`), draws streamlines in the current figure. There is no restriction about the velocity field: it may have a non-zero divergence, or a non-zero curl.

The streamlines are specified by their starting points (2D-point coordinates stored in the `start mfArray`, which may have any number of lines, but exactly two columns). The integration always begins at the starting point, in two possible direction: forward and/or backward (see below the explanation about the `direction` optional argument). Usually, the end-points of a streamline are located at the boundary of the domain, but in some rare cases (velocity field having a non zero divergence) they can end inside the domain, at a stationnary point.

(i, j) indices are mapped to the axis $(-Y, X)$, so data visualisation is coherent with the matrix layout and with other routines working with matrix data (`msContour`, `msPColor`, `msQuiver`). If this orientation is not appropriate, use the coordinates (`mfArrays` `X` and `Y`), as described below.

When arguments `X` and `Y` are also provided, then their values are used as coordinates for the data points in `u` and `v`:

- usually, `X` and `Y` are matrices and their shape must be the same as those of the data matrices `u` and `v`. Use of these coordinates is required when, *e. g.*, you want to swap the indices i and j, or apply a transformation (linear or not) between the (i, j) indices and the (x, y) coordinates; these coordinate matrices may be created by the `msMeshGrid` routine.
- to economize memory, it is possible to pass only vectors: in this case, `X` and `Y`, provided as rank-1 `mfArrays`, must obey to the same constraint as edicted for the generator vectors (see again `msMeshGrid`).

Be aware that `X` and `Y` are *not used* to defined the axes of the plot: to do so, you must use the `msAxis` routine with appropriate argument(s).

`u` and `v` must be rank-2 (*i. e.* matrices) `mfArrays` of same shape. They represent respectively the X and Y components of the vector field to be drawn. If the orientation or the scaling is not appropriate, use the second form of call described below, by adding coordinates.

The optional argument `direction` (character string) specifies whether the integration must be done following the flow direction (`"forward"`), against the flow direction (`"backward"`) or in both direction (`"both"`); this latter case is the default.

The optional `n_arrow` argument specifies the number of arrow heads to be drawn at equally curvilinear abscissa along each streamline. It must be an integer greater or equal zero (default value is 3); set it to zero if you want to suppress the arrow heads.

The optional `color` argument specifies the color used for drawing the streamlines; it has the same meaning as in the routine `msPlot`.

The optional `linestyle` argument (character string) specifies the line style, in the same way as `msPlot`; default line style is *continuous*.

The optional `linewidth` argument (real) specifies the line width, as a relative factor; by default its value is 1.

The optional `arrow_head` argument (real) specifies the size of the arrow head; by default its value is 1.

· · · / · · ·

The optional `npt_max` argument is the maximum number of points for each half-streamline (indeed, you can ask for a forward integration, a backward one or both – see above); its default value is 1000.

The optional `curv_tol` argument concerns the integration process and more specifically the tolerance for high curvature detection; it is relative to the mesh size; its default value is 0.5.

The optional `stop_zone` argument indicates to the integration process that some regions of the (x, y) plane are forbidden. For example, this is useful to avoid entering in a region near a singularity of the velocity vector field, which would lead to an excessive increase in the number of points. `stop_zone` is a user-provided subroutine, having the following interface:

```
logical function stop_zone( x, y )
   real(kind=MF_DOUBLE), intent(in) :: x, y
end function
```

This function is called for each new point and the integration process will stop as soon as a *TRUE* value is encountered.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

Remarks:

- The drawing is always clipped at the viewport.
- The function `mfStreamline` has the same argument list; it returns the vector of the (integer) handles of the created graphic objects. Be aware that even for a single streamline, the returned handle is a vector (of size 1).
- For drawing 2D streamlines on a triangular mesh, use the `msTriStreamline` routine.
- `NaN` values are not allowed: an error is returned by the routine. If such values are present in the vector field (*i. e.* in `u` or `v`) you can try to define a `stop_zone` user-function to avoid an error.

See also: `mf/msQuiver`

mf/msPatch — graphic patch

First interface:

```
call msPatch( x, y, c [, opacity] [, grid, grid_color] )
```

Description:
This routine draws one colored polygonal shape from the **mfArrays** `x`, `y` and `c`. These vectors contain respectively the coordinates of each vertex and the attached color. The polygonal shape doesn't need to be closed, *e. g.* a triangle may be defined only using three points.

The drawing is clipped at the viewport.

Before use, the color axis must be explicitly set by the user via the `msCAxis` routine. Theoretically, data should be ranged entirely inside the color axis; on the contrary, the resulting behavior depends on the shading value (cf. `msShading`). When shading is equal to `"flat"`, a quiet behavior is obtained and the colormap overflow can be monitored by use of the `msSetColorOverflowPolicy` routine; when shading is equal to `"interp"`, a warning is emitted by the library because strange colors can be drawn. This latter case should be avoided if possible.

If the optional argument `opacity` is present, transparency can be used: `opacity` must be a real number between 0 (full transparency, invisible object) and 1 (full opacity).

A thin line may be drawn to show the boundary of the polygonal cell by using the optional argument `grid` equal to *TRUE*. The default is to not show the boundary. The color of this thin line may be specified in the optional argument `grid_color`; by default, it is the foreground color.

A colormap must have been defined before the call, (via `msColormap`); otherwise, a warning is emitted and strange or unexpected results can occur.

Second interface:

```
call msPatch( x, y [, color] [, opacity] [, grid, grid_color] [, clipping] )
```

Description:
This second form fills the shape with a flat color (if specified), avoiding the use of an array for specifying the color. The `color` argument has the same meaning as in the routine `msPlot`.

`grid` and `grid_color` are described above for the first interface.

The optional argument `clipping` is a boolean (default value is *TRUE*). When set to *FALSE*, it allows the drawing of a polygonal colored shape (only using flat color) outside the viewport. This facility is especially useful when the user wants to update a text string outside the viewport: due to the use of antialiased text drawing, he must erase before the ancient text with a rectangular patch of appropriate color.

Remarks:

- the function `mfPatch` has the same argument list; it returns the (integer) handle of the created graphic object.
- this routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects of the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

See also: `mf/msPColor`

`mf/msPlotHist` — data histogram

Calling syntax:

```
handle = mfPlotHist( x, x_min, x_max, n_bin                                    &
                     [, color] [, filled] )
```

returns the (integer) handle of the newly created graphic object.

The vector `mfArray x` contains the data. The histogram is computed for data ranged from `x_min` to `x_max` using a number of bins equal to `n_bin`.

If present, `color` specifies the color for the bar plot. It may be a character string containing the usual color code (see `msPlot`), a real triplet containing the RGB values or a string containing a color name from the RGB database. If the color is left unspecified, then it is taken from the current colortable (for more information about colortables – or color schemes – see the *Muesli User's Guide*).

If present, the logical argument `filled` specifies that bars are filled with the color used (default is FALSE).

The subroutine form:

```
call msPlotHist( mfOut( num [, x_bin] ), x, x_min, x_max, n_bin                &
                 [, color] [, filled] )
```

allows the user to keep some internal data.

The optional arguments `color` (& Co.) and `filled` have the same meaning as above.

The output `mfArray num` is the vector of the bins values.

If present, the `mfArray x_bin` is the vector containing the abscissas of the bins.

Remarks:

- The drawing is always clipped at the viewport.
- This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.
- This routine may be called many times in the same figure with different data: histograms will be superposed one above the other, showing cumulative statistics. When not specified, colors will be recycled from the current colortable. Note that if your data is stored inside a two-dimensional array, then it may be easier to call the `msCumulHist` routine instead.

See also: `msHist`, `mfOut`

`msCumulHist` cumulative data histogram

Interface:

```
subroutine msCumulHist( X, x_min, x_max, n_bin, x_nb )

   type(mfArray), intent(in) :: X
   real(kind=MF_DOUBLE), intent(in) :: x_min, x_max
   integer, intent(in) :: n_bin
   type(mfArray), optional :: x_nb
```

Description:
Cumulative Histogram. Works on columns of the `mfArray X`.

If `X` has NCOL columns, draws first an histogram (via `msHist`) from the data of all columns, then draw a second histogram from the data of the NCOL-1 first columns, and so on. Each histogram is drawn in a different color, via the colors of the current colortable (for more information about colortables – or color schemes – see the *Muesli User's Guide*); if NCOL is greater than the maximum, following colors are cycled in the same set.

Columns of `X` may have been initialized from data which contain different number of element. In such a case, the optional argument `x_nb` is a vector which contains the useful number of elements in each column.

Remarks:

- The drawing is always clipped at the viewport.
- This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

See also: `msHist`, `mf/msPlotHist`

msImRead — image read

Interface:

```
call msImRead( mfOut(A,cmap), filename [, fmt, indexed] )
```

Description:
Reads the image file `filename`: the colormap is extracted from the image and stored in the `mfArray cmap` and the pixels values are stored in the `mfArray A`.

The image can be displayed by using first `msColormap` and then `msImage`.

The optional argument `fmt`, if present, is a string which must contain the image format. Common accepted values for `fmt` are: `"TIFF"`, `"JPEG"`, `"PNG"`, `"XPM"`, ... Actually, in the current version of MUESLI (but this should change in future), the file `image` is always converted in the XPM format via the '`convert`' command of '*ImageMagick*'. So, a great number of image formats is supported and, moreover, the argument `fmt` is not yet used for reading.

There are two options to fill the `mfArray A`, *i. e.* to give numerical values to pixels (the optional logical argument `indexed` is by default false):

1. real valued pixels – each pixel of the array contains a numerical real value (ranged in [0.,1.]), and a linear mapping is used between the whole range of these values and the index range of the colormap. This first way is rather oriented to the numerical processing of the pixels value, and is well adapted to image obtained via the pseudo colors from any numerical matrix.
   For example, you may apply some filters or compute the gradient of the pixels values and you can visualize the image in different ways by changing the colormap, which is always a continuum of colors.
2. indexed pixels – each pixel of the array contains the index of the colormap.
   This second way is more adapted to digital photos, or to color bitmaps which contain transparency (as for some PNG images – see below).
   You should not modify the whole colormap, else you may obtain some strange results when you visualize the image. In contrast, you can easily remove or add some colors in the colormap.

Remarks about transparent pixels:
If the original image contains transparent pixels ("NONE" color in an XPM file, or transparency in PNG files), then it cannot be opened using the default indexing scheme: the current routine will give an error. You can open it only via the "indexed pixels" scheme. Moreover, these transparent pixels will be colored in medium gray by the `msImage` routine.

Limitations:
Currently, these limitations are linked to the XPM format:

- original XPM images having more than two characters per color are not supported (to check yourself, open the XPM file with any text editor: the fourth integer number of the third line must be equal to 1 or 2). Moreover, they must have less than 8281 colors.
- for original XPM images, they must have maximum 12000 pixels in width if colors are stored in one char (*i. e.* when the number of colors is small, less than 91), or 6000 pixels in width if colors are stored in two chars (*i. e.* when the number of colors is greater than 91).
- for images converted in XPM by the 'convert' tool of '*ImageMagick*', the number of colors is always less than 256. So, the quality of true-color images is pretty degraded and, at least up to the version 6.9.3 of '*ImageMagick*', the picture is darker.

· · · / · · ·

Other interface:

```
call msImRead( mfOut(infos), filename [, fmt], only_infos=.true. )
```

allows the user to obtain some information about the image (currently, only the width and the height in pixels unit), which are returned in the `mfArray infos`.

See also: `msImWrite`, `mfOut`

`msImWrite` image write

Interface:

```
call msImWrite( A, cmap, filename [, fmt, indexed] )
```

Description:
Writes the `mfArray A` to the image file `filename`, using the colormap `cmap`.

The optional argument `fmt`, if present, is a string which must contain the image format. Common accepted values for `fmt` are: `"TIFF"`, `"JPEG"`, `"PNG"`, `"XPM"`, ... Actually, in the current version of MUESLI (but this should change in future), the file `image` is always converted in the XPM format via the '`convert`' command of '*ImageMagick*'. So, a great number of image formats is supported.

The optional logical argument `indexed` (its default value is false) has the same meaning as those explained in the `msImRead` routine. It concerns obviously the mfArray `A`, not the image stored in the image file `filename`.

Remarks about transparency:

- if the mfArray `image` contains transparent pixels, consider using an appropriate image format (XPM or PNG) with supports this transparency.
- transparency can be set in your image by two ways:
  1. for the indexed pixels indexing scheme, set a color (in `cmap`) to the triplet (*NaN*, *NaN*, *NaN*).
  2. for the real valued pixels indexing scheme, set an element of the image mfArray `A` to *NaN*.

Limitations:

- XPM images must have less than 8281 colors in their colormap.
- XPM images must have maximum 12000 pixels in width if colors are stored in one char (*i. e.* when the number of colors is small, less than 91), or 6000 pixels in width if colors are stored in two chars (*i. e.* when the number of colors is greater than 91).

See also: `msImRead`, `mf/msImage`

mf/msImage — image display

Interface:

```
call msImage( image [, angle] [, flip] [, indexed] )
```

Description:
Displays in the current figure the `mfArray image`, associated with the appropriate colormap previously set by msColormap.

(i, j) indices of the `image` are mapped to the axis $(-Y, X)$, so data visualisation is coherent with Contour and PColor.

The optional real argument `angle` is used to define the orientation in degrees of the image (default is 0 degree; any real values are accepted).

The optional argument `flip` is a string can take the following values: `"NONE"`, `"HORIZ"` (or mirror, left/right swap), `"VERT"` (or upside/downside swap); default value is `"NONE"`.

The optional logical argument `indexed` (its default value is *FALSE*) has the same meaning as those explained in the msImRead routine. It tells the current routine that the `image` mfArray contains integer indices of colors and not real values.

This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the msHold routine.

Remarks:

- The drawing is always clipped at the viewport.
- The function `mfImage` has the same argument list; it returns the (integer) handle of the created graphic object.

See also: mf/msContour, mf/msPColor

`mf/msPlotPSLG` — plot a PSLG domain

Description:
This routine plots a PSLG domain and, optionally, shows the numbering of nodes, edges and holes.

Calling syntax:

```
call msPlotPSLG( PSLG_domain [, color] [, linewidth]                    &
                 [, nod_num] [, edg_num] [, hol_num] )
```

`PSLG_domain`, of type `mfPSLG`, represents the 2D domain definition. It must be built by the user.

The optional `color` argument specifies the color used for drawing the cells: it may be a character string or a vector of type `real` and length 3 (see `mf/msPlot`).

The optional `linewidth` argument (real) specifies the line width, as a relative factor; by default its value is 1.

The three last optional arguments `nod_num`, `edg_num` and `hol_num`, all booleans, are used to display respectively nodes', edges' and holes' numbering of the PSLG. By default, numbering is not displayed.

Be aware that only nodes involved in segments are drawn; unused nodes are therefore ignored. To see all registered nodes, use `msPrintPSLG`.

Remarks:

- The drawing is always clipped at the viewport.
- This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.
- The function `mfPlotPSLG` has the same argument list; it returns the (integer) handle of the created graphic object.

`mf/msTriMesh` — plot a triangulation

Description:
This routine plots a triangular mesh and, optionally, shows the numbering of triangles, nodes and faces.

First calling syntax:

```
call msTriMesh( x, y, tri [, color] [, linewidth] [, height]                &
                [, tri_num] [, nod_num] [, boundary_only] )
```

The `mfArrays` `x` and `y` are vectors of same length which describe the position (x, y) of the nodes. The `mfArray` `tri` contains the triangulation (*i. e.* the triangles' indices, as returned by, *e. g.*, the routine `mfDelaunay`).

The optional `color` argument specifies the color used for drawing the cells: it has the same meaning as in other plotting routine involving colors (see the `msSetGrObj` routine).

The optional `linewidth` argument (real) specifies the line width, as a relative factor; by default its value is 1.

The optional `height` argument (real) specifies, for the numbering of triangles, nodes and faces, the character height, as a relative factor; by default its value is 1.

The two last optional arguments `tri_num` and `nod_num`, both booleans, are used to display triangles' and nodes' numbering of the triangular mesh. By default, numbering is not displayed.

When the optional boolean argument `boundary_only` is used with the value *TRUE* then only the boundary of the mesh is drawn. In this case, the triangles' numbers are never displayed, whatever the optional argument `tri_num` is.

Second calling syntax:

```
call msTriMesh( tri_connect, ...  [, fac_num] )
```

The first argument is `tri_connect` of type `mfTriConnect`, the mesh connectivity (see `msBuildTriConnect`); it replaces the three first `mfArrays` in the first calling syntax.

Same optional arguments as in the first calling syntax may be added. Note however that an additional optional argument is possible in this second case: the boolean `fac_num` concerns the numbering of the triangle faces.

This second use of `msTriMesh` uses a fast algorithm to draw the mesh; the printed files (both EPS and PDF) are also smaller in size.

Remarks:

- The drawing is always clipped at the viewport.
- This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.
- The function `mfTriMesh` has the same argument list; it returns the (integer) handle of the created graphic object.

See also: `msPrintTriConnect`

`mf/msPlotVoronoi` plot a Voronoi diagram

Description:
This routine plots a Voronoi diagram and, optionally, shows the numbering of points and vertices.

Calling syntax:

```
call msPlotVoronoi( voronoi [, color] [, linewidth] [, height]                &
                    [, nod_num] [, vert_num] )
```

The only mandatory argument is `voronoi`, a structure of type `mfVoronoiStruct`, created by the `mfVoronoi` routine.

The optional `color` argument specifies the color used for drawing the cells: it has the same meaning as in other plotting routine involving colors (see the `msSetGrObj` routine).

The optional `linewidth` argument (real) specifies the line width, as a relative factor; by default its value is 1.

The optional `height` argument (real) specifies, for the numbering of nodes and vertices, the character height, as a relative factor; by default its value is 1.

The two last optional arguments `nod_num` and `vert_num`, both booleans, are used to display nodes' and vertices' numbering. By default, numbering is not displayed.

Remarks:

- The drawing is always clipped at the viewport.
- This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.
- The function `mfPlotVoronoi` has the same argument list; it returns the (integer) handle of the created graphic object.

See also: `msPrintVoronoi`

`mf/msTriPColor` pseudo-color on a triangulation

Calling syntax:

```
call msTriPColor( x, y, z, tri )
```

Description:
This routine is equivalent to the routine `msPColor`, but for data which are spread on a triangulation.

The `mfArrays` `x`, `y` and `z` are vectors of same length which describes position (x, y) of the vertices and the corresponding function value.

The `mfArray tri` contains the triangulation (*i. e.* the triangle's indices, as returned by, *e. g.* the routine `mfDelaunay`).

Remarks:

- The drawing is always clipped at the viewport.
- Use `msTriMesh` to show the grid or the boundary of the triangular mesh.
- The function `mfTriPColor` has the same argument list; it returns the (integer) handle of the created graphic object.
- This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.
- `NaN` values are not (yet) allowed: an error is returned by the routine.

See also: `mf/msTriContour`, `mf/msTriFill`, `msPatch`

`mf/msTriFill` coloring on a triangulation

Calling syntax:

```
call msTriFill( x, y, val, tri )
```

Description:
This routine is equivalent to the routine `msPatch` (when using flat colors), but for data which are spread on a triangulation.

The `mfArrays` `x` and `y` are vectors of same length which describes position (x, y) of the nodes.

The value to be colored is stored inside the vector `mfArray val`. This latter vector must have a length equal to the number of triangles. The `mfArray val` may contain *NaN* values: in such a case the corresponding triangles are not drawn.

The `mfArray tri` contains the triangulation (*i. e.* the triangle's indices, as returned by, *e. g.* the routine `mfDelaunay`).

Remarks:

- The drawing is always clipped at the viewport.
- Use `msTriMesh` to show the grid or the boundary of the triangular mesh.
- The function `mfTriFill` has the same argument list; it returns the (integer) handle of the created graphic object.
- This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

See also: `mf/msTriContour`, `mf/msTriPColor`

`mf/msTriContour` data contouring on a triangulation

Description:
This routine is equivalent to the routine `msContour`, but for data which are spread on a triangular grid.

It doesn't fill the regions between the contours with colors; for such a feature, see the `mf/msTriContourF` routine.

First calling syntax:

```
call msTriContour( [ mfOut(C),] X, Y, Z, tri                              &
                   [, nb_levels, levels, linespec, linewidth,             &
                      labels, labelscolor, labelsize ] )
```

The `mfArrays` `X`, `Y` and `Z` are vectors of same length which describes position (x, y) of the nodes and the corresponding function value z.

The `mfArray tri` is the triangulation (*e. g.* returned by the routine `mfDelaunay`).

If the optional `C mfArray` is given in front of `(X,Y)` coordinate matrices, then it will contain on output a two-row matrix storing the contour lines. Each contiguous drawing segment contains the value of the contour, the number of (x, y) drawing pairs, and the pairs themselves. The segment are appended end-to-end as

```
C = [ level_1, x_1, x_2, ..., level_2, x_1, x_2, ...;
      pairs_1, y_1, y_2, ..., pairs_2, y_1, y_2, ...]
```

By default, the number of level curves is equal to 9, and the level values are equally spaced within the *min* and *max* values of the Color Axis (see `msCAxis`); this can be changed by using the optional arguments `nb_levels` and `levels` (see below).

The optional (integer) argument `nb_levels` specifies the number of contours to be drawned; it cannot be used together with `levels`.

The optional argument `levels`, which must be a rank-1 `mfArray`, is used to specify the level values.

The optional arguments `linespec` (character string) and `linewidth` allow the user to change the appearance of the contour curves. The possible values for these arguments are described in `msSetGrObj`, under the `"color"`, `"linestyle"` and `"linewidth"` entries.

By default, the level curves are labelled (use `labels=.false.` to remove the labels) and the contour curves are colored using the current colormap (use a color inside `linespec` to specify the color of all curves). Labels are displayed in the foreground color (*e. g.* black on white) by default; the user may choose another color via the `labelscolor` optional argument. Moreover the label size may be changed by using the optional `labelsize` real argument, which lets the user to specify a percentage of the standard size (default is `labelsize=1.0d0`).

Second calling syntax:

```
call msTriContour( [ mfOut(C),] Z, tri_connect, [ ... ] )
```

Here, `tri_connect` (of type `mfTriConnect`) is the mesh connectivity (see `msBuildTriConnect`); it replaces the three `mfArrays` `X`, `Y` and `tri` in the first calling syntax. The other optional arguments are the same.

···/···

Remarks:

- The triangular mesh used doesn't need to be convex, and may also have holes. See `mfPSLG` to create triangulation with holes.
- The drawing is always clipped at the viewport.
- Use `msTriMesh` to show the grid or the boundary of the triangular mesh.
- The function `mfTriContour` has the same argument list; it returns the (integer) handle of the created graphic object.
- This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.
- *NaN* values are allowed: all triangles involved by these special non finite values are simply ignored.

See also: `mf/msContourF`, `mf/msTriPColor`, `mf/msTriFill`

mf/msTriContourF — data contouring on triangles with filled regions

First calling syntax:

```
call msTriContourF( [ mfOut(C),] X, Y, Z, tri                              &
                    [, nb_levels, levels, linewidth,                       &
                       labels, labelscolor, labelsize ] )
```

Second calling syntax:

```
call msTriContourF( [ mfOut(C),] Z, tri_connect, [ ... ] )
```

Description:
Similar to the msTriContour routine, except that the regions between the contours are filled with appropriate colors.

The whole region whose values are greater than a specified level is filled with the corresponding color. As a consequence, the region less than all specified levels is left in white. To avoid white regions, it is sufficient to add a level less than (or equal to) the smaller value in the Z array.

Be aware however that some arguments of msTriContour are not available here.

Lastly, be warned that the function version (*i. e.* mfTriContourF) returns a vector of handle(s), even if the number of handles is one. Indeed, under some circumstances, the whole triangular mesh may be splitted in many independant zones when the number of NaN values is great enough.

See also: mf/msContourF

`mf/msTriQuiver` vector field plot on a triangulation

Calling syntax:

```
call msTriQuiver( x, y, u, v                                              &
                  [, color ] [, arrow_length ] [, arrow_head ] )
```

This routine is equivalent to the routine `msQuiver`, but for data which are spread on a triangulation.

`x`, `y`, `u` and `v` must be rank-1 (*i. e.* vectors) `mfArrays` having the same number of rows. They represent respectively the X and Y coordinates of the points where the components (u, v) of the vector field have to be drawn.

The optional `color` argument specifies the color used for drawing the arrows: it is a one-letter color symbol or an escaped sequence, as described in the `msSetGrObj` routine.

The optional `arrow_length` argument (real) specifies the length of the arrow as a multiplicative factor; by default its value is 1 so that the biggest arrow has approximately a length equal to 1/10th the full axis range.

The optional `arrow_head` argument (real) specifies the size of the arrow head; by default its value is 1.

Remarks:

- The drawing is always clipped at the viewport.
- An arrow is not drawn if any component of the corresponding vector (`u`, `v`) is equal to *NaN*.
- The function `mfTriQuiver` has the same argument list; it returns the (integer) handle of the created graphic object.
- This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the `Hold` property must be set to `"on"` via the `msHold` routine.

See also: `mf/msTriPColor`, `mf/msTriStreamline`

mf/msTriStreamline — vector field streamline on a triangulation

Calling syntax:

```
call msTriStreamline( x, y, u, v, start [, tri | tri_connect]                &
                      [, direction, n_arrow] [, color] [, linestyle]          &
                      [, linewidth, arrow_head] [, npt_max] [, curv_tol]      &
                      [, stop_zone] )
```

This routine is similar to the routine msStreamline, but for data which are spread on a triangulation. In particular, the arguments direction, n_arrow, color, linewidth, arrow_head, npt_max and curv_tol have exactly the same meanings.

The optional arguments tri and tri_connect (see msBuildTriConnect) describe respectively the triangulation of the mesh and its connectivity. If they are both absent, then the triangulation and its connectivity are done by the routine itself. If the user already knows the connectivity, it is preferable to pass it to the current routine, to avoid a redundant computation. On the other hand, if the user knows only the triangulation, it should pass it as argument, and the current routine will compute only the connectivity.

Remarks:

- The drawing is always clipped at the viewport.
- The function mfTriStreamline has the same argument list; it returns the vector of the (integer) handles of the created graphic objects. Be aware that even for a single streamline, the returned handle is a vector (of size 1).
- This routine, as most of high level ones, first erase the current plot; in order to plot multiple graphics objects on the figure, the Hold property must be set to "on" via the msHold routine.
- NaN values are not allowed: an error is returned by the routine. If such values are present in the vector field (*i. e.* in u or v) you can try to define a stop_zone user-function to avoid an error.

See also: mf/msTriQuiver

`mf/msSpy` sparsity pattern visualization

Calling syntax:

```
call msSpy( A [, symbolspec, bitmap, continuous,                    &
            color_scale, show_nnz, nz_threshold ] )
```

draws non-zero elements of the `mfArray A` (usually sparse).

If the optional argument `symbolspec` (string) is present, it must contain the specification for a symbol and/or a color, as described in msSetGrObj at `marker` and `color` entries. By default, a star (`"*"`) is used, but note that it is well adapted only for small size matrices: for big matrices, a dot (`"."`) may be prefered.

If the optional argument `continuous` (logical) is present, the matrix `A` is drawn using a color/grey scale for the absolute value of each element, as in msPColor. If you intend to spy a dense matrix, the `continuous` argument must be used. Default value is `.false.`
In the `continuous` case:

- if the optional argument `color_scale` (string) is present, it must be equal to `"lin"` or `"log"`. The latter case specifies the use of a logarithmic color/grey scale in the mapping between the value of the matrix elements and the color/grey index (default mapping is linear);
- if the optional argument `nz_threshold` (real) is present, it specifies the threshold value under which an element is considered as zero and, thus, not displayed. The default value is zero for the linear scale and $\max(|A_{i,j}|) \times \epsilon$ for the logarithmic scale. The threshold for small values is used for defining the color map range.
  By setting `nz_threshold` to the special value -1, the threshold will be set to the minimum value of the elements of $|A|$.

If the optional argument `show_nnz` (logical) is present, the effective number of non-zero element is shown in the X-label. It's default value is '`.true.`'.

If the optional argument `bitmap` (character string) is present, it must contains the bitmap size (in pixels) which will be used for printing. The string reads under the form `"WIDTHxHEIGHT"` (*e. g.* `"3000x3000"`). This option is useful only for large, sparse matrices.

Remarks:

- The drawing is always clipped at the viewport.
- The function `mfSpy` has the same argument list; it returns the (integer) handle of the created graphic object.
- The `continuous` and `symbolspec` options are exclusive.
- The grey scale mentionned above is the default colormap when none was selected by the user. Please use the msColormap routine to select another one, prior to the `msSpy` call.

2.6 Interactive routines

Caution: the routines available in this chapter require the selection of the *X11* driver (which is usually selected by default). In batch mode, where the user selects the *NULL* driver (see `msSetX11Device`), these routines will not be available, and an error will occur if the user tries to call any of them.

A figure must be also opened (see `mf/msFigure`).

If you have multiple workspaces, switching from one workspace to another may breaks the interaction of the mouse with your Muesli program; therefore, it is recommended to stay in the same workspace during all the duration of the interactive process.

Some routines (`msPan`, `msZoom` and `msPanAndZoom`) accept the use of an additional keyboard key (usually the CTRL one) to modify the action. In such a case, it is required that the figure's window have the focus; on the contrary, the use of the key do nothing.

`mfGinput`, `msGinput`	graphic input
`mfGinputCustom`	graphic input using custom cursors with alternative
`mfGinputRect`	rectangular graphic selection
`mfGetModKeys`	modifier keys state
`msPan`	manual scrolling
`msZoom`	figure zoom
`msPanAndZoom`	combine scrolling and zooming
`msMoveLegend`	adjust legends frame in axes
`msMoveGrObj`	move graphic object
`msAnimation`	screen animation mode
`msShowNow`	screen update during an animation
`msCla`	clear current axes
`msSetGBuffer`	graphic buffer policy
`msBBuf`, `msEBuf`	begin and end of graphic buffering
`msDefineCustomCursors`	define custom cursors for use with `mfGinputCustom`

See also:

Global graphic settings

Window's and figure's management

Figure properties

Figure annotation – Low level graphic object's manipulation

High level plotting routines

`mfGinput` graphic input

Interface:

```
function mfGinput( event, key ) result ( out )

   logical, intent(in), optional :: event, key
   type(mfArray)                 :: out
```

Description:
This function is an interactive facility to get the pointer coordinates when it is inside the current figure.

Usually, this function is called without the optional `event` and `key` argument (by default, `event` is *TRUE*): the library waits for a mouse click or a key pressed, and the cursor show a cross shape in the current active window.

The returned `mfArray` contains the coordinates (x, y) of the selected point. The point can be chosen by any mouse button (which can be also emulated via the `L`, `M` or `R` key). Be aware that the current routine implements a two-event procedure and therefore the location returned is that of the pointer for the second event! (*mouse up* in the usual case).

For other key pressed, the routine `mfGinput` terminates and leaves the returned `mfArray` empty.

Remarks:

- this routine is usually employed for graphic interaction. If the user doesn't have a mouse, the pointer can be moved via the keyboard's arrows (the `shift key` accelerates the move by a factor of 10); moreover each mouse button can be emulated by using an equivalent sequence (`L`, `M` or `R` followed by any other key).
- see the `msGinput` routine for more features.

When called with the `event` argument equal to *FALSE*, the library doesn't wait for a user event: it returns immediately the pointer coordinates in the `mfArray out`. A third element is added to `out`, indicating whether the pointer is inside the window (value 1) or outside (value 0); the user must use this flag to take any decision in its own program.

Moreover, if the `key` is present and equal to *TRUE*, then a fourth element is returned in `out`: the ASCII code of the key pressed recently.

See also: `mfGinputRect`

`msGinput` graphic input

Calling syntax:

```
call msGinput( mfOut( coords, keycode [, color] )                    &
               [, whole_keyboard, rect_size, rect_inside_axes] )
```

Description:
This function is an interactive facility to get the pointer position inside the current figure. See `mfGinput` for a simpler use.

In case of multiple opened graphic windows, the cursor has a cross shape in the current active one instead of the classical pointer.

The returned `mfArray coords` contains the coordinates (x, y) of the selected point. The point can be chosen either by the mouse (default behavior), or by a key pressed (see below).

The returned `mfArray keycode` contains an equivalent code for the event detected:

- If the optional argument `whole_keyboard` is *FALSE* (default behavior), then `keycode` contains the button number (left=1, middle=2, right=3), even if the equivalent keys (`L`, `M` or `R`) have been used; Note that `mfArray coords` and `keycode` will be empty if any key different from `L`, `M` or `R` is pressed.
- On the contrary, `keycode` contains the value of the key pressed (actually `ichar(key_pressed)`, where `ichar` is the standard Fortran function which returns the ASCII code of the letter). `coords` will be empty only if the ESCAPE key is pressed. Only when using keyboard, the mouse wheel can be detected. The values returned are `ichar(8)=56` for wheel up, and `ichar(2)=50` for wheel down (think that on a numeric keypad, 8 is for arrow up, and 2 is for arrow down).

If present, the returned `mfArray color` contains the RGB triple of the colored pixel just under the pointer.

Usually (*i. e.* when the optional argument `rect_size` is not present), the cursor shown on the screen has the shape of a big crosshair, whose length is that of the selected window; on the contrary, if the `mfArray rect_size` is present and not empty, it must contain the size (both width and height) of a rectangle which follows the pointer during its move. This rectangle is always centered around the pointer position. Moreover, if the `logical` optional argument `rect_inside_axes` is present and equal to *TRUE*, the rectangle drawn is constrained to be entirely inside the axes.

See also: `mfGinputRect`

`mfGinputCustom` graphic input using custom cursors with alternative

Description:

The first *Calling syntax* has no argument:

```
out = mfGinputCustom( )
```

Three elements are returned in the `mfArray out`: the coordinates (x, y) of the pointer when the mouse has been clicked, and the case used (1 or 2)

1. without the Control key pressed, displaying the first custom cursor;
2. with the Control key pressed, displaying the second custom cursor.

The other *Calling syntax* implements a magnetic grid and has one argument:

```
out = mfGinputCustom( magnetic_grid_rule )
```

The `magnetic_grid_rule` subroutine, provided by the user, must have the following interface:

```
subroutine magnetic_grid_rule( icase, x1, y1, x2, y2, valid )
   integer,              intent(in)  :: icase
   real(kind=MF_DOUBLE), intent(in)  :: x1, y1
   real(kind=MF_DOUBLE), intent(out) :: x2, y2
   logical,              intent(out) :: valid
end subroutine
```

where `x1, y1, x2, y2` are world-coordinates. The `(x1,y1)` is the pointer position in the figure, and the `(x2,y2)`, computed by the user, is where the user-defined cursor is displayed (`icase` is 1 or 2, according the status of the Control Key, pressed or not, as mentioned above). If, for some reasons, no `(x2,y2)` point can be computed, then the routine must set the last argument (`valid`) to *FALSE* and in this case the cursor will show its standard shape (*i. e.* the left arrow).

In this second calling syntax, four elements are returned in the `mfArray out`:

1. x-coordinate of the clicked location.
2. y-coordinate of the clicked location.
3. an integer indicating whether the Control Key has been pressed or not.
4. an integer giving the validity of the position; if the validity is zero, the position should be ignored.

Note: Before use, the `msDefineCustomCursors` routine must be called to define the two cursor shapes.

`mfGinputRect` — rectangular graphic selection

Interface:

```
function mfGinputRect() result ( out )

   type(mfArray) :: out
```

Description:
This function is an interactive facility to get the size and position of a rectangle drawn inside the current figure.

The returned `mfArray` contains the bounding box of the rectangle. This rectangle must be selected by the left mouse button only; for other key pressed, the routine `mfGinputRect` terminates and leaves the returned `mfArray` empty.

In case of multiple opened graphic windows, the cursor has a cross shape in the current active one, instead of the classical pointer.

See also: `mfGinput`

`mfGetModKeys` — modifier keys state

Interface:

```
function mfGetModKeys( ) result ( out )

   logical :: out(2)
```

Description:
This function checks the state of the keyboard for the **Shift** and the **Control** keys.

It returns an array of two booleans: the **first** one (resp. the **second** one) tells if one of the **Shift** keys (resp. the **Control** keys) was down when this routine was called.

`msPan` — manual scrolling

Calling syntax:

```
call msPan( )
```

Description:
Activates the 'pan' mode for scrolling in the current figure. It ends when the [ESCAPE] key is pressed.

Scrolling is monitored via the mouse (or the equivalent keyboard key):

- click-and-drag with the *left* button ('L' key) scrolls through the figure to a new view;
- *middle*-click does nothing;
- *right*-click ('R' key) resets the view to the initial one; if the [CTRL] key is pressed during the click (or the '6' key is pressed) then it centers the view with respect to the bounding box of all graphic objects.

Remark:
Scrolling in the current figure should be used before putting annotations on the figure, otherwise, parts of graphic objects drawn outside the axes will not be moved with this routine; however, a call to `msRedrawFigure` will fix this problem.

See also: `msZoom`, `msPanAndZoom`

`msZoom` — figure zoom

Calling syntax:

```
call msZoom( )
```

Description:
Activates the 'zoom' mode in the current figure. It ends when the [ESCAPE] key is pressed.

The zoom is monitored via the mouse:

- the *left* button is used to "zoom in" at the pointer location:
  - a single click is used to enlarge the current view with a factor of 1.41 (or 1.19 if the [CTRL] key is pressed during the click);
  - a click-and-drag zooms to the new selected area;
- a *middle*-click is used to "zoom out" at the pointer location (by a factor of 1.41 — or 1.19 if the [CTRL] key is pressed during the click);
- a *right*-click resets the view to the initial one; if the [CTRL] key is pressed during the click, then it sets the view to an area which shows all the graphic objects.

See also: `msPan`, `msPanAndZoom`

`msPanAndZoom` — combine scrolling and zooming

Calling syntax:

```
call msPanAndZoom( )
```

Description:
Activates both the 'pan' mode for scrolling and the 'zoom' mode for zooming in the current figure. It ends when the [ESCAPE] key is pressed.

Scrolling is monitored via the three mouse buttons:

- click-and-drag with the left button scrolls the figure to a new view;
- middle-click does nothing;
- right-click resets the view to the initial one; if the [CTRL] key is pressed during the click, then it sets the view to an area which shows all the graphic objects.

Zooming is monitored via the mouse wheel by a factor of 1.41 (or 1.19 if the [CTRL] key is pressed during the click). Note that the figure remains centered during this zooming process.

Remark:
Scrolling in the current figure should be used before putting annotations on the figure, otherwise, parts of graphic objects drawn outside the axes will not be moved with this routine; however, a call to `msRedrawFigure` will fix this problem.

See also: `msPan`, `msZoom`

`msMoveLegend` — adjust legends frame in axes

Calling syntax:

```
call msMoveLegend( [ handle ] )
```

Description:
This interactive routine allows the user to move the legend frame inside the figure via the mouse.

When a unique legends frame has been created by `msLegend`, the routine has no argument.

Use the optional argument to move a legends frame created by `mfLegend` and referenced by its `handle`.

Note that the legend frame may be located outside the axes; this may be useful in the case where a great number of curves are plotted in the figure. During the interactive move on the screen, the legend frame may appear truncated by the window boundary under *X11*; however, printing in EPS or PDF should give correct results.

The use of this routine could follow `msPan`, `msZoom` or `msPanAndZoom` in order to optimize its position.

`msMoveGrObj` — move graphic object

Interface:

```
subroutine msMoveGrObj( handle, prompt )

   integer,          intent(in)           :: handle
   character(len=*), intent(in), optional :: prompt
```

Description:
Allows the user to move some types of graphic object, by interaction with the mouse (left button click only). It ends when the [ESCAPE] key is pressed. Supported graphic object types are: *text* (generated by `mfText`), *arrow* (generated by `mfArrow`) and *polygon* (generated by `mfPatch` using a flat color).

When this routine is called, it waits for a *mouse down* event (first part of a click) while the pointer have the shape of an opened hand. To grab a *text* or *polygon* object, you can click elsewhere inside its bounding box (displayed using a gray animated dashed line); for an *arrow*, you have three hotspots (sensitive area): one is the arrow base, the second is the middle and the third is the arrow head.

When a *mouse down* event is detected, the shape of the pointer becomes a closed hand and the graphic object (referenced by its `handle`) can be moved in the figure. Note that a *text* or *polygon* object can be only translated, without changing its orientation; however, for an *arrow* and according to the selected hotspot, you can change both its direction (when you have clicked on its base or its head) and its location (when you have clicked in its middle).

When a *mouse up* event is detected (end of a click), the pointer shape changes back to an opened hand.

The `prompt` argument lets you chosing an appropriate sentence on the screen. By default, the prompt is adapted to the type of graphic object:

text: `-> Grab and move the text string surrounded by a dashed line...`

arrow: `-> Grab and move the arrow wearing three hotspots...`

polygon: `-> Grab and move the polygon surrounded by a dashed line...`

Remarks:

- Note that the graphic objects may be clipped at the axes framebox. If you move a graphic object out of the axes, it may no longer be visible; however, the dashed line(s) around it will be always visible and will be an appreciable help.
- An example of use of the `msMoveGrObj` routine can be found in the test program named `MoveGrObj_test` inside the `tests/fgl` folder.

See also: `msPan`, `msZoom`, `msMoveLegend`

`msAnimation` screen animation mode

Calling syntax:

```
call msAnimation( "on" | "off" )
```

Description:
The aim of this routine is to avoid the flickering between frames when you want to show quickly (usually in a `do` loop) a succession of graphic objects.

Note that, at the end of the animation, you will not be able to print the figure in EPS or PDF files (graphic objects are not saved into memory), unless you run yourself the same graphic commands out of the animation.

Before setting animation to `"on"`, a window must be opened (with `msFigure`). Moreover axis must be defined as constant, by calling the `msAxis` routine before.

After drawing each frame, the only way to show the display of the graphic object in the figure is to call the `msShowNow` routine. The *MUESLI User Guide* presents a detailed example of a typical sequence of such an animation.

At the end of the animation sequence, the user should set animation to `"off"`.

Remarks:
During an animation:

- the "hold" property cannot be changed by the user. Therefore, the `msHold` routine shouldn't be used.
- all graphic objects are volatile; as a consequence the handle returned by all plotting functions beginning by `mf` is equal to zero. Therefore, you cannot use any routine requiring a valid handle.
- don't use `msClf` (which does a complete erasing of the figure) between frames: in such a case, this will annihilate a large part of the benefit of the current routine. Use `msCla` instead.
- a varying text string may be displayed outside the axes, and erased between frames with the use of a white rectangle over it (see `mf/msPatch`).
- if the `msPColor` routine is used, then the optional arguments `X` and `Y` must be used.

`msShowNow` screen update during an animation

Calling syntax:

```
call msShowNow( )
```

Description:
This routine may be called only during an animation (see `msAnimation`). It forces the update of the screen to show the new display of the current frame.

`msCla` — clear current axes

Calling syntax:

```
call msCla()
```

Description:
Clears the current axes, *i. e.* fills the rectangle inside the axes using the background color.

A typical usage concerns the animation of any graphic objects (curve, symbol, text, ...): at the beginning of a loop, the axes should be erased with `msCla` (quick way) and then graphic objects are drawn. Avoid to use `msClf` (heavy way) because this latter routine do a complete reset of the figure (among other things, the axes range is lost).

See also: `msFigure`, `msClose`

`msSetGBuffer` graphic buffer policy

Interface:

```
subroutine msSetGBuffer( flag )

   character(len=*), intent(in) :: flag
```

`flag` may be equal to `"on"` or `"off"`.

Description:
Activates the graphic buffer (default is `"on"`). Usually increases the display speed.

The graphic buffer may be turned `"off"` for debugging purpose, for example.

See also: `msBBuf`, `msEBuf`

`msBBuf` — begin of graphic buffering

Calling syntax:

```
call msBBuf( )
```

Description:
Begins to draw in the graphic buffer, instead of directly on the screen.

Remarks:

- this routine is active only if the graphic buffer is not turned off by the `msSetGBuffer` routine;
- actually, many internal low-level graphic routines are already buffered.

See also: `msSetGBuffer`, `msEBuf`

`msEBuf` end of graphic buffering

Calling syntax:

```
call msEBuf( )
```

Description:
Ends to draw in the graphics buffer.

`msBBuf` and `msEBuf` calls should always be paired.

See also: `msSetGBuffer`, `msBBuf`

`msDefineCustomCursors` define custom cursors for use with `mfGinputCustom`

Interface:

```
subroutine msDefineCustomCursors( cursor_1, cursor_1_mask, color_1,    &
                                  cursor_2, cursor_2_mask, color_2 )

   character(len=*), intent(in) :: cursor_1, cursor_1_mask, color_1,   &
                                   cursor_2, cursor_2_mask, color_2
```

Description:
Register two customized cursors, provided by the user in the XBM format. The arguments `cursor_*` are the filenames of these images. Each cursor is defined by two bitmap images: one for the cursor shape itself and the other for the mask. A color must be also specified for each cursor.

A size of 16×16 pixels is recommended. The cursor main image must define the Hot Spot position, for example

```
#define _x_hot 8
#define _y_hot 8
```

Last, each color is a colorname from the RGB X11 database.

See also: `mfGinputCustom`

Index

1 - Derived Types

2 - Parameters and Global Variables

3 - Operators

4 - FML Routines

5 - FGL Routines