

# MUESLI

## *User's Guide*

### *Fortran implementation*

Release 2.23.5

Édouard CANOT\*

May 4, 2026



---

\*IPR/CNRS, Rennes, France

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>What is MUESLI ?</b>	<b>5</b>
2.1	MUESLI and other Products	6
<b>3</b>	<b>Getting Started</b>	<b>7</b>
3.1	Accessing the <code>muesli-config</code> Tool	7
3.2	Using <code>muesli-config</code>	7
3.3	Compiling and Linking MUESLI Programs	8
3.4	Graphical Usage	9
3.5	Using Makefiles	9
3.6	Setting the Environment Variables	9
3.7	Using Preprocessing	10
3.7.1	General Case	10
3.7.2	Compiler Dependant Functions	10
3.8	Command Line Argument	10
3.9	Private protection of the derived types in Muesli	10
<b>4</b>	<b>FML (Numerical Library)</b>	<b>11</b>
4.1	First seeds of MUESLI	11
4.1.1	The Fundamental Derived Type: <code>mfArray</code>	11
4.1.2	Creating and Setting <code>mfArrays</code>	11
4.1.3	Displaying <code>mfArrays</code>	12
4.1.4	Parameters and Global Variables	14
4.1.5	Information returned at Run-Time by MUESLI	15
4.1.6	Programmer's Facilities in MUESLI	15
4.2	Working with <code>mfArrays</code>	16
4.2.1	Setting and Getting Elements in an <code>mfArray</code>	16
4.2.2	Concatenating and Replicating <code>mfArrays</code>	20
4.2.3	Doing Operations with <code>mfArrays</code>	22
4.2.4	Boolean <code>mfArrays</code>	24
4.2.5	Doing Linear Algebra	27
4.2.6	Data Analysis with <code>mfArrays</code>	33
4.2.7	Domain triangulation	41
4.2.8	Other Numerical Tasks	43
4.2.9	Saving/Reading <code>mfArrays</code> to/from Disk	47
4.2.10	Sparse <code>mfArrays</code>	47
4.3	Advanced topics	55
4.3.1	Matrix Properties	55
4.3.2	Physical Units in <code>mfArrays</code>	56
4.3.3	Performance Issues	60
4.3.4	Managing Memory	61
4.3.5	Creating your own <code>mf-</code> or <code>ms-</code> Routines	62
4.3.6	Numerical Precision	63
4.3.7	Debugging in MUESLI	63
4.3.8	Exchanging Data with MATLAB	74
4.4	Equivalence between MUESLI Routines, MATLAB Commands and <code>f90</code> Intrinsic Functions	75
4.4.1	from MUESLI Routines	75
4.4.2	from MATLAB Commands	76
4.4.3	from <code>f90</code> Intrinsic Functions	77
<b>5</b>	<b>FGL (Graphical Library)</b>	<b>78</b>
5.1	Creating and Numbering Figures	78
5.2	How axes may be defined in your figure?	78
5.3	Color Management	79
5.4	High-level Plotting Routines	80
5.5	Changing Orientation/Transposition of Matrix Data	98
5.5.1	Matrix Orientation in Practice	98

5.6	Interactive Routines	99
5.7	Annotating a Figure	100
5.8	Text, Symbols and Fonts	103
5.8.1	About the Character Sets used	103
5.8.2	Greek Alphabet and other Symbols	103
5.8.3	Fonts available	103
5.9	Printing Figures	107
5.10	Low-level Graphic Object's Manipulation	107
5.11	Making Animations	109
<b>Index</b>		<b>111</b>

# 1 Introduction

This document describes the MUESLI Fortran library.

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

More information can be found in the following documents:

- *MUESLI Installation Guide*
- *MUESLI Reference Manual*
- *MUESLI Inside*

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

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

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

Thanks to the MUESLI users: Samih ZEIN, Rabé BADÉ, Mohamad MUHIEDDINE, Eduardo MENDES, Flávio AQUINO, Saulo Benchimol BASTOS, Mohamed Lamine SAHARI, Merline Flore DJOUWE, Salwa MANSOUR, Paweł BIERNAT, Jerzy WRÓBEL, Antti KEMPPINEN, Corentin BEAUCÉ, Anthony RUIZ, Kamel SADOK, Saeid BAHRAMI, Robinson OUTEROVITCH, Sidaty CHEIKH SIDI ELY, Mateus da SILVA TEIXEIRA, Vikas SHARMA, Nicolas CHAPUIS, Rawad WAKIM, Maxime BOILEAU.

## Typographic convention

In this guide, coloring is 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**.

MUESLI routine names displayed in **this color** are included in the final index.

For sake of clarity, array constructors use 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 Niderlander. It can be used under a Limited Royalty Free License (see <http://www.dreamstime.com/muesli-and-candied-fruits-imagefree4006423>).

## 2 What is MUESLI ?

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

- **FML** (Fortran Muesli Library) contains all necessary materials to numerically work with a dynamic array (dynamic in size, type and structure), called **mfArray**.

To work with FML, your Fortran source code must include the statement:

```
use fml
```

- **FGL** (Fortran Graphics Library) contains graphics routines which use the **mfArray** objects.

To work with FGL, your Fortran source code must include the statements:

```
use fml
use fgl
```

Nearly all of MUESLI routine names begin either by the two characters "**ms**" (for Muesli Subroutine), or by "**mf**" (for Muesli Function).

In the case where one of your symbols conflicts<sup>1</sup> with those of the MUESLI library, you can rename the MUESLI symbol by the following example:

```
use fml, mfsin_muesli => mfsin
```

In such a case, the MUESLI function **mfSin** will be yet available by using the **mfsin\_muesli** symbol. Of course, you can also change your own **mfsin** symbol.

---

<sup>1</sup>This is detected at compile-time, generally by a message like:  
Error: Symbol **mfsin** conflicts with symbol from module **mod\_mfaux...**

## 2.1 MUESLI and other Products

Actually, MUESLI is trying to be a near-clone of MATFOR®, especially of its philosophy. Note that most of routine names are derived from the ones of MATLAB. It's approach is also close to the MATRAN's one but, in our humble opinion, MUESLI is easier to use.

### MATLAB®

Matlab is an interactive software for algorithm development, data visualization, data analysis, and numeric computation. Add-on toolboxes extend the Matlab environment to solve particular classes of problems. Its syntax (vector- and matrix-oriented) makes it a powerfull high-level language.

Matlab is a commercial product: <http://www.mathworks.com/products/matlab>

### OCTAVE

Octave is a high-level language, primarily intended for numerical computations. It provides a convenient command line interface for solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with Matlab.

Octave is a free software: <http://www.octave.org>

### SCILAB

Scilab is a scientific software for numerical computations providing a powerful open computing environment for engineering and scientific applications. Scilab includes hundreds of mathematical functions with the possibility to add interactively programs from various languages. It has sophisticated data structures, an interpreter and a high-level programming language.

Scilab is a free software: <http://www.scilab.org>

### MATRAN

Matran is a Fortran 95 wrapper that implements matrix operations and computes matrix decompositions using Lapack and the Blas. It provides the flavor and convenience of coding in matrix oriented systems like Matlab, Octave, etc. By using routines from Lapack and the Blas, Matran allows the user to obtain the computational benefits of these packages with minimal fuss and bother.

Matran is a free software: <http://www.cs.umd.edu/~stewart/matran/Matran.html>

### MATFOR®

Matfor is a set of Fortran 95 libraries that enhances your program with dynamic visualization capabilities, shortens your numerical codes and speeds up your development process. By adding a few lines of Matfor codes to your Fortran program, you can easily visualize your computing results and perform real-time animations.

Matfor is a commercial product: <http://www.ancad.com/overview.html>

### 3 Getting Started

The aim of this section is to verify that MUESLI is properly installed and works well.

#### 3.1 Accessing the `muesli-config` Tool

First of all, you must verify that the MUESLI configuration tool works on your system. In a terminal, type the following command:

```
$ muesli-config
```

You should obtain something like:

```
MUESLI configuration tool
MUESLI is a numerical and graphical library with a Fortran API.
Copyright (C) 2003-2026, Édouard CANOT, IPR/CNRS, Rennes, France.
https://perso.univ-rennes1.fr/edouard.canot/muesli
```

```
Usage: muesli-config [OPTION]
```

Known values for OPTION are:

```
--version          library version
--check-version ver_min [ver_max]
                    check library version:
                    ver_min <= version [ <= ver_max ]
--build-opt        options used during the build
--f90compiler       Fortran compiler name
--f90flags          Fortran pre-processor and compiler flags
--ccompiler         C compiler name
--cflags            C compiler flags
--libs              library linking information
--libs-static       static link (if available)
--blas-lib          BLAS library
--blas-implen       BLAS implementation currently in use
--lapack-lib         LAPACK library
--lapack-implen     LAPACK implementation currently in use
--graphic-part      Availability of FGL
--prefix            MUESLI installation prefix
--info              all information (library + compiler)
--help              displays this help and exit
```

If you obtain instead:

```
bash: muesli-config: command not found
```

then, you should add the location of the `muesli-config` script in your `PATH` environment variable, for example (for bash shells):

```
$ export PATH=<path to muesli-config>:$PATH
```

If you are not sure that MUESLI has been installed on your system, you can also install it anywhere inside your own home directory. See the *MUESLI Installation Guide* for doing this.

#### 3.2 Using `muesli-config`

As you can see when you invoke it, `muesli-config` has many options giving some information about the MUESLI configuration. Only one option must be used at a time, but one of them (`--all`) displays all known information:

```
$ muesli-config --info
```

returns for example:

```

library source path:
/home/ecanot/tools/fortran/f90-modules/MUESLI/muesli/GNU_GFC

library installation path:
/home/ecanot/lib/muesli

library documentation path:
/home/ecanot/share/doc/muesli

library release:
2.23.0_2025-12-26

build date:
wen. 26 dec 2025 07:56:59 CEST

compiler version:
GNU Fortran (GCC) 13.4.0
Copyright (C) 2023 Free Software Foundation, Inc.

make options:
make CFG=debug PROF=no

```

All this information is important in the case where you find a bug or an incorrect behavior of the MUESLI library.

The next subsection shows you how to take advantage of the `muesli-config` script to easily compile and link your programs.

### 3.3 Compiling and Linking MUESLI Programs

Three options of the `muesli-config` script are useful: `--f90compiler`, `--f90flags` and `--libs`.

The first one, `--f90compiler`, gives the name of the Fortran 90 compiler used to build the installed version of MUESLI. For compatibility reasons with the precompiled modules of MUESLI, the same compiler (and even sometimes the same version) must be used to generate MUESLI programs. For example:

```

$ muesli-config --f90compiler
gfortran

```

The second one, `--f90flags`, is related to the compilation options you should use. It returns paths for module's inclusion or other specific flags. For example:

```

$ muesli-config --f90flags
-D_GNU_GFC -I/home/ecanot/lib/muesli -I. -fno-range-check -w -O3 -funroll-loops
-ffree-line-length-none -fPIC

```

A simple way to include the returned strings in the command line is to use backquotes:

```

$ `muesli-config --f90compiler` `muesli-config --f90flags` -c myprog.f90

```

this will produce the object file `myprog.o`. It is strictly equivalent to:

```

$ gfortran -D_GNU_GFC -I/home/ecanot/lib/muesli -I. -fno-range-check -w -O3
-funroll-loops -ffree-line-length-none -fPIC -fopenmp -c myprog.f90

```

The last option, `--libs`, returns all the required linking flags and libraries. For example:

```

$ muesli-config --libs
-Wl,-rpath,/home/ecanot/lib/muesli -L/home/ecanot/lib/muesli -lfgl -lfml -lmfp
lot -llapack -lblas -lX11 -lXRander -lXrand -lz -lpthread -lreadline -lhistory
/home/ecanot/lib/ muesli/dummy_papi.o -lstdc++

```

Therefore, to build your executable `myprog`, you should use the following shorter command:



```
$ 'muesli-config --f90compiler' -o myprog myprog.o 'muesli-config --libs'
```

### 3.4 Graphical Usage

**CAVEAT:** As already explained in the *Muesli Installation Guide*, the graphical part of Muesli may or may not work on [Wayland](#). If your experienced some unexpected, not documented errors using Muesli on Wayland, you should switch to [Xorg](#). Usually, you can switch from one protocol to the other during the login process.

### 3.5 Using Makefiles

Using a Makefile is an easy way to produce your executables. This means that you will avoid to repeatedly type appropriate commands in the terminal.

For example, in order to build the `myprog` executable from the `myprog.f90` source file, you should copy the following command lines in a file named `Makefile` (actually a GNU-Makefile):

```
CFLAGS =

myprog: myprog.o
    'muesli-config --f90compiler' -o $@ $< 'muesli-config --libs'

.SUFFIXES:
.SUFFIXES: .f90 .o

%.o: %.f90
    'muesli-config --f90compiler' 'muesli-config --f90flags' $(CFLAGS) -c $<

clean:
    rm -f *.o myprog
```

(don't forget to insert the `<TAB>` character instead of the eight spaces at the beginning of some lines; moreover be aware of the backquotes used around the `muesli-config` calls).

The `CFLAGS` variable can be used to set additional flag, for example the `'-g'` one which tell to the compiler to add symbols and information in order to debug your program. It is just an example because, actually, this option is perhaps already included in the list of options used by the `muesli-config` script (it should be the case if you use the `DEBUG` version of the Muesli library).

Then, type the `make` command to process the Makefile.

### 3.6 Setting the Environment Variables

In order to have a ready-to-use library, some environment variables must be set correctly: `PATH` and, optionally, `LD_LIBRARY_PATH`.

The `PATH` environment variable contains a list of directories containing executable binaries (*e.g.*, the compiler name, the `muesli-config` script, etc). Type:

```
$ echo $PATH
```

to verify.

If the installation of MUESLI is in the `OPTIM` mode (actually, it is certainly the case), you will have to include also in the `LD_LIBRARY_PATH` variable the path of the HDF5 shared library (only if you have configured Muesli to use this latter library).

If, at execution, the following error message is displayed:

```
$ myprog
./myprog: error while loading shared libraries: libhdf5.so: cannot open shared
object file: No such file or directory
```

you must register the path of `libhdf5.so` by setting:

```
$ export LD_LIBRARY_PATH=<path to libhdf5.so>:$LD_LIBRARY_PATH
```

In the case where you obtain problems with other shared libraries, you have to build a static executable by typing (but this static build is not always available, especially if you install Muesli from a binary package):

```
$ 'muesli-config --f90compiler' -o myprog myprog.o 'muesli-config --libs-static'
```

## 3.7 Using Preprocessing

### 3.7.1 General Case

For many reasons, you may need to use preprocessing in your program. You can do it via, as usual, the sharp character symbol (#) in the first column of your source files.

For example, you can introduce the definition of any preprocessing symbol in your Makefile by using the universal '-D' flag. This must be added in the automatic rule command which transform the source file (%.F90) into object file (%.o), or added in the CFLAGS variable above.

In any case, don't forget that your source file must end with the '.F90' extension, and not '.f90'.

### 3.7.2 Compiler Dependant Functions

In the case where you have to use some compiler dependant functions, be aware that the following preprocessing symbols are already defined, only one for each compiler used:

```
_GNU_GFC
_INTEL_IFC
```

## 3.8 Command Line Argument

The two following usual forms are possible:

- the 'iarg()' and 'getarg()' F77 compatibility functions <sup>2</sup>;
- the command\_argument\_count() and get\_command\_argument() F2003 functions.

## 3.9 Private protection of the derived types in Muesli

Be aware that most of internal components of Muesli derived types (mfArray, mfMatFactor, mfTriConnect, mfTetraConnect, mf\_DE\_Named\_Group) are hidden to the user by a special (non-standard) mechanism. You cannot access to these internal components from your program. Typically, the following error describes such a situation:

```
$ 'muesli-config --f90compiler' 'muesli-config --f90flags' -c myprog.f90

myprog.f90:134:59:

 134 |     print *, "Domain convexity: ", tri_connect%convex_domain
      |                                           1
Error: 'convex_domain' at (1) is not a member of the 'mftriconnect' structure;
did you mean '-convex_domain'?
```

In the example shown above, you cannot access to the `convex_domain` component of the `mfTriConnect` structure because it has been hidden.

---

<sup>2</sup>declare 'iargc' as an integer, without the 'external' attribute; this is the recommended rule valid for all the compilers supported in MUESLI.

## 4 FML (Numerical Library)

### Typographic note

Knowing that Fortran is not a case sensitive programming language, we use lower and upper case names only for the sake of clarity.

### 4.1 First seeds of MUESLI

#### 4.1.1 The Fundamental Derived Type: `mfArray`

An `mfArray` object is an automatic and dynamic 2D array. This means that at run time, its shape, type and structure can change automatically without the need to do it explicitly. The program deduces them from the context.

The `mfArray` `A` is first declared as follows:

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

We must see now how to initialize it and how to work with it.

In all examples presented in this document, except when it is specified, all the variables are of type `mfArray`.

#### 4.1.2 Creating and Setting `mfArrays`

An `mfArray` can be assigned a scalar:

```
A = 7
```

a vector:

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

or a matrix:

```
A = reshape( [ 1, 2, 3, 4, 5, 6 ], [ 2, 3 ] )
```

which has 2 rows and 3 columns. The array which is contained in `A` has exactly the same shape than the Fortran array in the *RHS* of the previous assignment. Moreover, the elements are stored in the same order; after all, it is just a special Fortran array.

Many functions exist to help you in creating classical and well known `mfArrays`. A random array is a typical example:

```
A = mfRand( 2, 3 )
```

Even if your `mfArray` `A` is a scalar or a vector, it will be stored internally as a rank-2 array; therefore you must specify its two dimensions. So, for example, to create a vector of 5 random values, it will writes:

```
A = mfRand( 1, 5 )
```

for a row vector, and

```
A = mfRand( 5, 1 )
```

for a column vector.

Other MUESLI routines are available to build identity matrices (`mfEye`), zeros matrices (`mfZeros`), ones matrices (`mfOnes`), integers sequences (`mfColon`), equally spaced vectors (`mfLinSpace`), etc. In the *MUESLI Reference Manual*, they are listed into the subsection named "Elementary Matrix Manipulation Functions".

Note also that the `mf()` routine converts many Fortran objects into `mfArrays`. The use of this latter routine is sometimes useful when you must put an `mfArray` argument to a routine.

Finally, a graphical tool should also be mentioned: 'meditor' is a small Qt-based application<sup>3</sup>, like a

---

<sup>3</sup>perhaps this application is not available on your system.

spreadsheet, which can be launched by the `msMedit` routine. This tool could be helpful to interactive programs (see however some limitations in the *MUESLI Reference Manual*); it could also be used as a model to develop other interactive tools based on the Qt library (see the document *MUESLI Inside*).

► Example #4-1:

```
A = mfEye( 6 ) ! Identity matrix of order 6
call msMedit( A ) ! launch a graphic tool
call msDisplay( A, "A" )
```

	1	2	3	4	5	6	7	8
1	1	0	0	0	-1	0		
2	0	1	0	0	0	0		
3	0	0	1	0	0	0		
4	0	0	0	1	0	0		
5	0	0	0	0	1	0		
6	-1	0	0	0	NaN	1		
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								

▷ Output of ex. #4-1: (after some modifications on the screen)

```
A =

  1.0000    0.0000    0.0000    0.0000   -1.0000    0.0000
  0.0000    1.0000    0.0000    0.0000    0.0000    0.0000
  0.0000    0.0000    1.0000    0.0000    0.0000    0.0000
  0.0000    0.0000    0.0000    1.0000    0.0000    0.0000
  0.0000    0.0000    0.0000    0.0000    1.0000    0.0000
 -1.0000    0.0000    0.0000    0.0000     NaN     1.0000
```

### 4.1.3 Displaying mfArrays

For many reasons, calling the `msDisplay` subroutine is the only way to print `mfArrays` on the screen. Indeed, the `mfArray` derived type contains private components hence only a part of the whole structure is relevant to print. Moreover, printing a big array on the screen must be done by a specific routine, because information should be clearly presented.

Anyway, a try to directly print an `mfArray`, as *e.g.*:

```
A = mfRand( 2, 3 )
print *, "A = ", A
```

will ordinarily lead to an error at compile time (GNU fortran compiler):

```
print *, "A = ", A
1
Error: Data transfer element at (1) cannot have POINTER components
unless it is processed by a defined input/output procedure
```

`msDisplay` is a sophisticated routine which takes care of, among other things, the number of columns of

your terminal<sup>4</sup>. It prints the array in a pretty form which is column-oriented, like in MATLAB.

Here are some examples of `mfArrays` displayed.

► Example #4-2:

```
A = mfRand( 2, 3 )
call msDisplay( A, "A" ) ! default : only 5 digits are printed
```

▷ Output of ex. #4-2:

```
A =
0.9839    0.2753    0.8098
0.7000    0.6611    0.9100
```

► Example #4-3:

```
call msFormat("long") ! changes the default : 15 digits are printed
A = mfRandN(5,5) ! generates random numbers with normal distribution
call msDisplay(A, "A") ! fits the number of cols to the terminal width
```

▷ Output of ex. #4-3:

```
A =
Columns 1 through 4
-1.25110306696529    0.89704366553922   -1.92831962464287   -0.15523126577206
 2.17898954410140    1.57784530974119    0.20224358652870   -0.70089607528083
 1.36288491055927    0.80171278372511    0.72856226839328    0.36313288326467
-0.67409792738829    2.29746275397409    0.53498742614739    0.03939436198377
-0.32474732088845   -1.88330423996211   -0.94996868780511   -0.67354050025227

Column 5
 0.91939656133688
-0.15673634460146
 0.71741927783678
-0.07448835251978
 2.31970783552094
```

► Example #4-4:

```
call msFormat("short") ! returns to the default behavior
A = mfRandN(5,1)*1.0e+5
call msDisplay(A, "A") ! factorizes a common exponent
```

▷ Output of ex. #4-4:

<sup>4</sup>if the current display doesn't fit your terminal, you can force the number of columns by setting manually the environment variable `MF_COLUMNS`. See also the description of the `msSetTermCol()` routine in the *MUESLI Reference Manual*

```
A =

1.0E+05 *

-1.2511
0.8970
-1.9283
-0.1552
0.9194
```

Two interesting options available in `msDisplay` are `head` and `tail`. They show only few elements of the `mfArray`, avoiding a print of a large number of elements.

► Example #4-5:

```
A = mfLinSpace(0.0d0,1.0d0,1001) ! long row vector
A = .t. A                        ! long col vector
call msDisplay(A, "A", head=5, tail=5)
```

▷ Output of ex. #4-5:

```
A =

0.0000
0.0010
0.0020
0.0030
0.0040
... (991 lines skipped)
0.9960
0.9970
0.9980
0.9990
1.0000
```

#### 4.1.4 Parameters and Global Variables

A few number of parameters and constants are predefined in MUESLI, in order to allow the user to quickly (or easily) set `mfArrays` to certain values.

These parameters include mathematical usual real constants (`MF_PI`  $\approx 3.14159$ , `MF_E`  $\approx 2.71828$ ) as well as special IEEE value (`MF_INF` for infinity and `MF_NAN` for Not-a-Number).

The complex constant `MF_I` (pure imaginary complex number) is also useful to build complex `mfArrays`.

► Example #4-6:

```
A = .t. mf( [ 1, 2, 3, 4, 5 ] ) * MF_I
call msDisplay(A,"A")
```

▷ Output of ex. #4-6:

```
A =

0.0000 + 1.0000i
0.0000 + 2.0000i
0.0000 + 3.0000i
0.0000 + 4.0000i
0.0000 + 5.0000i
```

Section 1 in the *MUESLI Reference Manual* describes other real parameters. On the other hand, only three (special) `mfArray` parameters are defined in MUESLI: `MF_EMPTY`, `MF_COLON` (or its alias `MF_ALL`) and `MF_END`.

At the user level, it is not possible to declare other `mfArray` parameters, because the `mfArray` derived type contains private components. However, the `msSetAsParameter` allow a protection of the data; this routine must be used explicitly, once the `mfArray` object is initialized.

► Example #4-7:

```
permeability = 1.0d-12           ! initialization
call msSetAsParameter( permeability, param=.true. ) ! protecting the mfArray

permeability = 0                 ! should rise an error
```

▷ Output of ex. #4-7:

```
(MUESLI assignment(=):) ERROR: cannot change LHS !
                           it is a protected array (pseudo-parameter).

traceback requested from MUESLI :
...
```

Lastly, `STDIN`, `STDOUT` and `STDERR` are global variables which can be modified by the user.

#### 4.1.5 Information returned at Run-Time by MUESLI

As opposed to a classical work with standard Fortran 90 variables, when working with the MUESLI library, the compiler doesn't know the actual type of an `mfArray` because it is a dynamic object.

So, the major part of the errors you obtain will arise at run-time and not at compile-time. When a MUESLI routine encounters an error (or even an unexpected situation), it always prints a message which should be self-explanatory for the user.

MUESLI messages are of three kinds: *info* (just give some information), *warning* (expect some strange results) or *error* (something wrong has happened). MUESLI rarely stops the program execution, except for internal errors (bugs). However, for *error* kind messages, MUESLI waits for an answer from the keyboard to resume.

The default behavior can be changed by the user, at any location in his program, via the `msSetMsgLevel` subroutine (see the *MUESLI Reference Manual*). In particular, the quiet mode is useful when using a batch system: in case of error, the program stops without waiting for a user answer.

#### 4.1.6 Programmer's Facilities in MUESLI

When your program executes a long task, it can be useful to see the progress report. In the case where a costly part is located inside a loop, the `msPrintHashes` routine displays a line of hashes, progressively during the loop execution. Another routine, `msPrintProgress`, is more sophisticated since it prints the percentage of the work done and an estimation of the remaining time.

In other circumstances, when the program prompts the user for the name of a file or whatever, the user will find useful to use the `mfReadLine` routine. This routine and others (`msReadHistoryFile`, `msWriteHistoryFile`) brings to the user the ability to navigate (via the keyboard arrows) in an history and to edit the line. A small example of use of this facility is presented below:

► Example #4-8:

```

! We want to ask the user some data (here some sensors' indices):
! 1) load the history (file located in the same folder)
call msReadHistoryFile( ".inverse_pb_local.history" )
! 2) ask the user to enter some data (the only argument is the prompt
!     which will be presented in the terminal; 'string' will be
!     automatically added to the history)
string = mfReadLine( " Sensors are: " )
! 3) read in 'string' in order to initialize the 'ind' vector
read(string,*) ind(:)
! 4) if all is ok, save the new history (usually in the same file)
call msWriteHistoryFile( ".inverse_pb_local.history" )

```

## 4.2 Working with mfArrays

### 4.2.1 Setting and Getting Elements in an mfArray

Two routines are used to set and get elements of an `mfArray`: `msSet` and `mfGet`. Both can be used with a single element, an array section, or even whole row(s) and whole column(s). The position of the elements is given as two lists of integers which are the row and column indices.

► Example #4-9:

```

A = reshape( [ (i, i = 1, 9) ], [ 3, 3 ] )
call msDisplay(A,"A")
call msSet(0.5d0,A,2,2) ! changes one element
call msDisplay(A,"msSet(0.5d0,A,2,2), A")

```

▷ Output of ex. #4-9:

```

A =

   1   4   7
   2   5   8
   3   6   9

msSet(0.5d0,A,2,2), A =

 1.0000  4.0000  7.0000
 2.0000  0.5000  8.0000
 3.0000  6.0000  9.0000

```

► Example #4-10:

```

A = reshape( [ (i, i = 1, 9) ], [ 3, 3 ] )
call msDisplay(A,"A")
call msSet(0.25d0,A,[2,3],[2,3]) ! changes an array section
call msDisplay(A,"msSet(0.25d0,A,[2,3],[2,3]), A")

```

▷ Output of ex. #4-10:



```

A =

    1    4    7
    2    5    8
    3    6    9

msSet(0.25d0,A,[2,3],[2,3]), A =

    1.0000    4.0000    7.0000
    2.0000    0.2500    0.2500
    3.0000    0.2500    0.2500

```

► Example #4-11:

```

A = reshape( [ (i+0.5, i = 1, 9) ], [ 3, 3 ] )
call msDisplay(A,"A")
call msSet(0.0d0,A,MF_ALL,[3]) ! changes a whole column
call msDisplay(A,"msSet(0.0d0,A,MF_ALL,[3]), A")

```

▷ Output of ex. #4-11:

```

A =

    1.5000    4.5000    7.5000
    2.5000    5.5000    8.5000
    3.5000    6.5000    9.5000

msSet(0.0d0,A,MF_ALL,[3]), A =

    1.5000    4.5000    0.0000
    2.5000    5.5000    0.0000
    3.5000    6.5000    0.0000

```

► Example #4-12:

```

A = reshape( [ (i+0.5, i = 1, 9) ], [ 3, 3 ] )
call msDisplay(A,"A")
call msSet(MF_EMPTY,A,[2],MF_ALL) ! suppresses a whole row
call msDisplay(A,"msSet(MF_EMPTY,A,[2],MF_ALL), A")

```

▷ Output of ex. #4-12:

```

A =

    1.5000    4.5000    7.5000
    2.5000    5.5000    8.5000
    3.5000    6.5000    9.5000

msSet(MF_EMPTY,A,[2],MF_ALL), A =

    1.5000    4.5000    7.5000
    3.5000    6.5000    9.5000

```

► Example #4-13:

```

A = [ (i, i = 1, 12) ]
call msDisplay(A,"A")
call msDisplay(mfGet(A,3),"mfGet(A,3)") ! gets one element in a vector

```

▷ Output of ex. #4-13:

```
A =
      1      2      3      4      5      6      7      8      9     10     11     12

mfGet(A,3) =
      3
```

Index sequences can be easily created by using the following special feature of Muesli: `i .to. j .by. k` or `i .to. j .step. k` creates a index sequence, where `i` is the start index, `j` is the end index, and `k` is the step (only this latter integer may be negative). Moreover, `i` or `j` can be replaced by the parameter `MF_END` (or any expression using addition or subtraction involving it), which means the last element of the corresponding dimension.

► Example #4-14:

```
A = reshape( [ (i, i = 1, 25) ], [ 5, 5 ] )
call msDisplay(A,"A")
! gets an array section (non contiguous)
call msDisplay(mfGet(A, MF_ALL, 1 .to. 5 .by. 2 ),           &
               "mfGet(A, MF_ALL, 1 .to. 5 .by. 2 )")
! gets only the last part of the last line
call msDisplay(mfGet(A, MF_END, MF_END-2 .to. MF_END ),     &
               "mfGet(A, MF_END, MF_END-2 .to. MF_END )")
```

▷ Output of ex. #4-14:

```
A =
      1      6      11      16      21
      2      7      12      17      22
      3      8      13      18      23
      4      9      14      19      24
      5     10      15      20      25

mfGet(A,MF_ALL, 1 .to. 5 .by. 2 ) =
      1     11     21
      2     12     22
      3     13     23
      4     14     24
      5     15     25

mfGet(A, MF_END, MF_END-2 .to. MF_END ) =
     15     20     25
```

More things may be done using index sequences; the `' .but. '` operator is used to exclude one integer from a sequence:

► Example #4-15:

```
v = [ (i, i = 1, 9) ]
call msDisplay(v,"v")
! gets a non contiguous section
call msDisplay(mfGet(v, MF_ALL .but. 5 ),           &
               "mfGet(v, MF_ALL .but. 5 )")
```

▷ Output of ex. #4-15:

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

mfGet(v, MF_ALL .but. 5 ) =
      1      2      3      4      6      7      8      9
```

Lastly, the ‘.and.’ operator may be used to combine two index sequences:

► Example #4-16:

```
v = [ (i, i = 1, 9) ]
call msDisplay(v,"v")
! combining two index sequences (parenthesis around each sequence is
! recommended)
call msDisplay(mfGet(v, (2 .to. 4) .and. (7 .to. MF_END) ),      &
               "mfGet(v, (2 .to. 4) .and. (7 .to. MF_END) )")
```

▷ Output of ex. #4-16:

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

mfGet(v, (2 .to. 4) .and. (7 .to. MF_END) ) =
      2      3      4      7      8      9
```

Under certain circumstances, you may find that the routine `msPointer` could be more useful (see the examples below). However, you must know exactly the current data type of the `mfArray`. Moreover, it locks some internal properties about the matrix, therefore you must not forget to call `msFreePointer` instead of a simple nullify of the pointer; this latter routine remove the pointer link but also unlocks the internal properties.

► Example #4-17:

```
! real(kind=MF_DOUBLE), pointer :: f90_ptr(:, :)

A = [ 1, 2, 3 ]
call msDisplay(A,"A")
call msPointer(A,f90_ptr) ! special use of: f90_ptr => A
f90_ptr(1,2) = 777
print *, "'msPointer(A,f90_ptr); f90_ptr(1,2) = 777'"
call msDisplay(A,"A")
call msFreePointer(A,f90_ptr) ! implies: f90_ptr => null()
```

▷ Output of ex. #4-17:

```
A =
      1      2      3

'msPointer(A,f90_ptr); f90_ptr(1,2) = 777'

A =
      1    777      3
```

## ► Example #4-18:

```
! real(kind=MF_DOUBLE), pointer :: f90_ptr_vec(:)

A = mfEye(3)
call msDisplay(A,"A")
call msPointer(A,f90_ptr_vec) ! special use of: f90_ptr_vec => A
f90_ptr_vec(2) = 7
print *, "'msPointer(A,f90_ptr_vec); f90_ptr_vec(2) = 7'"
call msDisplay(f90_ptr_vec,"f90_ptr_vec")
call msFreePointer(A,f90_ptr_vec) ! implies: f90_ptr_vec => null()
```

## ▷ Output of ex. #4-18:

```
A =

  1    0    0
  0    1    0
  0    0    1

'msPointer(A,f90_ptr_vec); f90_ptr_vec(2) = 7'

f90_ptr_vec =

  1    7    0    0    1    0    0    0    1
```

Lastly, `msSet` accepts out-of-range indices for element(s) to be modified: the matrix shape is modified accordingly, with filling by a predefined value (`NaN` by default) which can be changed by use of the `msSetAutoFilling` routine.

## ► Example #4-19:

```
A = reshape( [ (i, i = 1, 9) ], [ 3, 3 ] )
call msDisplay(A,"A")
call msSet(0.75d0,A,4,5) ! out-of-range indices => reshapes the matrix
call msDisplay(A,"msSet(0.75d0,A,4,5), A")
```

## ▷ Output of ex. #4-19:

```
A =

  1    4    7
  2    5    8
  3    6    9

msSet(0.75d0,A,4,5), A =

  1.0000    4.0000    7.0000    NaN    NaN
  2.0000    5.0000    8.0000    NaN    NaN
  3.0000    6.0000    9.0000    NaN    NaN
  NaN      NaN      NaN      NaN    0.7500
```

### 4.2.2 Concatenating and Replicating mfArrays

`mfArrays` concatenation is obtained via two operators (`.hc.` and `.vc.`) which place two matrices side by side, either horizontally or vertically.

## ► Example #4-20:

```

A = transpose( reshape( [ (-i, i = 1, 6) ], [ 2, 3 ] ) )
call msDisplay(A,"A")
B = reshape( [ (i, i = 1, 15) ], [ 3, 5 ] )
call msDisplay(B,"B")
z = A .hc. B ! A or B could be empty
call msDisplay(z,"A .hc. B")

```

▷ Output of ex. #4-20:

```

A =

  -1  -2
  -3  -4
  -5  -6

B =

   1   4   7  10  13
   2   5   8  11  14
   3   6   9  12  15

A .hc. B =

  -1  -2   1   4   7  10  13
  -3  -4   2   5   8  11  14
  -5  -6   3   6   9  12  15

```

► Example #4-21:

```

A = mfColon( 1, 10 ) ! integers sequence from 1 to 10
A = .t. mfReshape( A, 5, 2 ) ! '.t.' operator takes the transpose
call msDisplay(A,"A")
A = A .vc. [ 11.0d0, 12.0d0, 13.d0, 14.d0, 15.d0 ]
call msDisplay(A,"A .vc. [ 11.0d0, 12.0d0, 13.d0, 14.d0, 15.d0 ]")

```

▷ Output of ex. #4-21:

```

A =

   1   2   3   4   5
   6   7   8   9  10

A .vc. [ 11.0d0, 12.0d0, 13.d0, 14.d0, 15.d0 ] =

   1   2   3   4   5
   6   7   8   9  10
  11  12  13  14  15

```

Another way to construct a bigger matrix is to use `mfRepMat` which applies concatenation of the same tile-matrix in both direction.

► Example #4-22:

```

A = reshape( [ (i, i = 1, 4) ], [ 2, 2 ] ) ! this is the tile
call msDisplay(A,"A",mfRepMat(A,2,4),"mfRepMat(A,2,4)")

```

▷ Output of ex. #4-22:

```

A =
  1  3
  2  4

mfRepmat(A,2,4) =
  1  3  1  3  1  3  1  3
  2  4  2  4  2  4  2  4
  1  3  1  3  1  3  1  3
  2  4  2  4  2  4  2  4

```

### 4.2.3 Doing Operations with mfArrays

Usual arithmetic operation (+, −, \*, /) can be applied to **mfArrays**, because the corresponding Fortran intrinsic operators have been overloaded by specific routines. More generally, nearly all the operations and functions available in Fortran 90 can be applied to **mfArrays** (however sometimes under a different command name, see the section 4.4).

The general rule could be the following one: try first to use the function you want (of course, prefixed by "mf"). If you obtain an error during the compilation of your program, then see the *MUESLI Reference Manual* which describes all available MUESLI routines. If your function doesn't exist, you can in a last resort implement it by reading the guidelines in the section 4.3.5.

Most of the mathematical functions work element-wise. Using a Fortran 90 specific term, MUESLI functions are *elemental*. This is the case for the multiplication (\*), the exponential function (**mfExp**), the square-root function (**mfSqrt**), the sine function (**mfSin**), etc.

MUESLI contains a number of specialized mathematical functions: **mfErf** for the Error Function, **mfGamma** for the  $\Gamma$  function; it also contains routines about Bessel functions (**mfBesselI**, **mfBesselJ**, **mfBesselK**, **mfBesselY**) and Complete Elliptic Integrals of first and second kinds (**msEllipKE**).

We cannot show an example of each MUESLI function. In the MUESLI source package, there is a folder named *tests* which contains many program tests. Please refer to these source files to understand how the routines work. Only few examples are presented here.

► Example #4-23:

```

A = mfOnes(3) - mfEye(3)*MF_I ! mixing real and complex kinds
call msDisplay(A,"A")

```

▷ Output of ex. #4-23:

```

A =
  1.0000 - 1.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
  1.0000 + 0.0000i   1.0000 - 1.0000i   1.0000 + 0.0000i
  1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 - 1.0000i

```

► Example #4-24:

```

A = mfMagic(3)
call msDisplay(A,"A")
A = A * A ! element-wise operation
call msDisplay(A,"A * A")

```

▷ Output of ex. #4-24:

```

A =

      8      1      6
      3      5      7
      4      9      2

A * A =

     64      1     36
      9     25     49
     16     81      4

```

► Example #4-25:

```

A = mfOnes(2,3)*3.0d0
call msDisplay(A,"A")
call msDisplay(A/2.0d0,"A/2.0d0" )

```

▷ Output of ex. #4-25:

```

A =

      3      3      3
      3      3      3

A/2.0d0 =

     1.5000     1.5000     1.5000
     1.5000     1.5000     1.5000

```

► Example #4-26:

```

x = mfLinSpace( -1.5d0, 1.5d0, 4 )
call msDisplay(x,"x",mfASin(x),"mfASin(x)") ! result will be complex

```

▷ Output of ex. #4-26:

```

x =

    -1.5000    -0.5000     0.5000     1.5000

mfASin(x) =

    -1.5708 + 0.9624i   -0.5236 + 0.0000i    0.5236 + 0.0000i    1.5708 - 0.9624i

```

Looking for matrix-functions, use `mfMul` for the (non commutative) multiplication. This routine is the equivalent of the `matmul` function in Fortran 90. On the other hand, the `dotproduct` Fortran 90 function doesn't need to exist in MUESLI: if you have two column vectors, use the transpose of the first vector multiplied by the second vector.

► Example #4-27:

```

A = mfMagic(5)
x = .t. [ 0, 0, 0, 1, 0 ] ! creation of a column vector
call msDisplay(A,"A",x,"x")
call msDisplay(mfMul(A,x),"mfMul(A,x)") ! equiv. to the 'matmul' f90 intrinsic

```

▷ Output of ex. #4-27:

```

A =

  17   24    1    8   15
  23    5    7   14   16
   4    6   13   20   22
  10   12   19   21    3
  11   18   25    2    9

x =

  0
  0
  0
  1
  0

mfMul(A,x) =

  8
 14
 20
 21
  2

```

► Example #4-28:

```

x = [ 1, 2, 3, 4, 5 ] ! creation of two vectors
y = [ 5, 4, 3, 2, 1 ]
call msDisplay(x,"x",y,"y")
! equiv. to the 'dotproduct' f90 intrinsic
call msDisplay(mfMul(x,.t.y),"dot product = mfMul(x,.t.y)")

```

▷ Output of ex. #4-28:

```

x =

  1    2    3    4    5

y =

  5    4    3    2    1

dot product = mfMul(x,.t.y) =

 35

```

#### 4.2.4 Boolean mfArrays

Boolean **mfArrays** are automatically created when using comparison operators (**==**, **/=**, **>**, etc.). Syntactically, the same symbols than in Fortran 90 can be used, due to operator overloads. A similar situation arises with logical operators (**.not.**, **.and.**, **.or.**, etc.), and also for logical reduction functions **All** and **Any**.

Arithmetic on boolean **mfArrays** is not permitted, but the matrix structure can be modified by the following routines: **mfReshape**, **mfRepMat**, **mfDiag** (only diagonal extraction from a matrix), **mfFlipLR**, **mfFlipUD**, **mfRot90** and the operators **.vc.**, **.hc.**, **.t.**.



## ► Example #4-29:

```

A = [ 1, 2, 3 ]
B = [ 3, 2, 1 ]
call msDisplay( A, "A", B, "B" )
call msDisplay( A==B, "A==B" ) ! boolean mfArray displayed (F=false, T=true)

```

## ▷ Output of ex. #4-29:

```

A =
  1    2    3
B =
  3    2    1
A==B =
  F    T    F

```

## ► Example #4-30:

```

A = [ 1, 2, 3 ]
B = [ 3, 2, 1 ]
call msDisplay( A, "A", B, "B" )
call msDisplay( A>=B, "A>=B" ) ! boolean mfArray displayed

```

## ▷ Output of ex. #4-30:

```

A =
  1    2    3
B =
  3    2    1
A>=B =
  F    T    T

```

## ► Example #4-31:

```

A = reshape( [ (i, i = 1, 9) ], [ 3, 3 ] )
call msDisplay( A, "A" )
! by default, 'mfAll' works on columns
call msDisplay( mfAll(A>5.0d0), "mfAll(A>5.0d0)" )
! below, 'mfAll' works on rows since the first dimension is specified
call msDisplay( mfAll(A>5.0d0,1), "mfAll(A>5.0d0,1)" )

```

## ▷ Output of ex. #4-31:

```

A =

    1    4    7
    2    5    8
    3    6    9

mfAll(A>5.0d0) =

    F    F    T

mfAll(A>5.0d0,1) =

    F
    F
    F

```

To use boolean `mfArrays` in tests, you should apply `All` or `Any` to them.

► Example #4-32:

```

A = reshape( [ (i, i = 1, 9) ], [ 3, 3 ] )
call msDisplay( A, "A" )
print *, "all(A>5.0d0) : ", all(A>5.0d0) ! logical result
if( any(A>=9.0d0) ) then
  print *, "at least one element of 'A' is greater or equal 9"
else
  print *, "no element of 'A' is greater or equal 9"
end if

```

▷ Output of ex. #4-32:

```

A =

    1    4    7
    2    5    8
    3    6    9

all(A>5.0d0) : F
at least one element of 'A' is greater or equal 9

```

You can also assign classical Fortran logical arrays (rank 0, 1 or 2) to `mfArrays`.

► Example #4-33:

```

v = [ .true., .false., .true. ] ! creating a boolean vector
call msDisplay( v, "v" )

```

▷ Output of ex. #4-33:

```

v =

    T    F    T

```

Lastly, boolean `mfArrays` cannot have physical units (see section 4.3.2).

### 4.2.5 Doing Linear Algebra

There are a number of routines for doing linear algebra. Actually, they form an important part of the numerical part of MUESLI. Most of them are high-level routines which prepare data for a call to specific Lapack routines. So, solving a linear system of equations (**mfLDiv** or **mfRDiv**), factorizing a matrix in the product  $LU$  (**msLU**),  $QR$  (**msQR**) or getting its singular values (**msSVD**) can be realized via a single call, like in Matlab, with a very light and intuitive interface.

Only a subset of these routines is presented here; refer to the section “Matrix Functions” in the *MUESLI Reference Manual* for the complete list of available routines.

**mfLDiv** — the left division for matrices — is used to solve a linear system like  $Ax = b$ . It returns the vector solution  $x = A^{-1}b$  without computing the inverse of the matrix  $A$ . Many internal tests are made to calling the most appropriate Lapack routine. The matrix  $A$  doesn't need to have the same number of row and column: when facing with an overdetermined system, **mfLDiv** returns the least square solution; when facing with an underdetermined system, **mfLDiv** returns one possible solution.

► Example #4-34:

```
n = 3
A = mfRand(n) + real(n)*mfEye(n)
b = .t. mf( [ (i, i=1,n) ] )
call msDisplay(A,"A",b,"b")
x = mfLDiv(A,b) ! solve the linear system : A.x = b    (x = A\b in Matlab)
call msDisplay(x,"x")
call msDisplay(mfNorm(mfMul(A,x)-b),"residue |A*x-b|")
```

▷ Output of ex. #4-34:

```
A =
    3.1270    0.8258    0.4808
    0.3185    3.2216    0.3556
    0.3092    0.5334    3.1360

b =
    1
    2
    3

x =
    0.0496
    0.5206
    0.8632

residue |A*x-b| =
    4.4409E-016
```

The previous routine always returns a solution, even if the matrix  $A$  is singular. In such a case, it firstly compute the pseudo-inverse of  $A$  and then multiply it by  $b$ . Here is an example:

► Example #4-35:

```

n = 3 ! below, 'A' is a well-known example of singular matrix
A = mf( reshape( [ (i, i=1,n*n) ], [n,n] ) )
b = mfMul( A, mfOnes(n,1) ) ! taking 'b' in the image
call msDisplay(A,"A",b,"b")
x = mfLDiv(A,b) ! solution 'x' is not unique
call msDisplay(x,"x")
call msDisplay(mfNorm(mfMul(A,x)-b),"residue |A*x-b|")

```

▷ Output of ex. #4-35:

```

A =

    1    4    7
    2    5    8
    3    6    9

b =

    12
    15
    18

(MUESLI mfLDiv:) Warning: A is not invertible !
                    Moore-Penrose pseudo-inverse will be used to compute
                    the solution.

x =

    1.0000
    1.0000
    1.0000

residue |A*x-b| =

    3.9721E-15

```

In a similar way, the `mfRDiv` routine can be used to solve systems like  $x A = b$  or  $A' x = b'$ ; in the latter case, simply call the `mfRDiv` routine then transpose the returned vector, since it is equivalent to  $x' A = b$ .

In the case where you want an orthonormal basis of a subspace, you can use the `mfOrth` routine:

► Example #4-36:

```

n = 3 ! below, 'A' is a well-known example of singular matrix
A = mf( reshape( [ (i, i=1,n*n) ], [n,n] ) )
call msDisplay(A,"A",mfRank(A),"rank(A)")
call msDisplay(mfOrth(A),"orth(A)")

```

▷ Output of ex. #4-36:

```

A =

    1    4    7
    2    5    8
    3    6    9

rank(A) =

    2

orth(A) =

   -0.4797    0.7767
   -0.5724    0.0757
   -0.6651   -0.6253

```

In the previous example, the orthonormal basis of  $A$  contains only two vectors because the rank of  $A$  is equal to 2.

If you want to test whether a vector  $v$  belongs or not to a given subspace, there exist at least two possibilities. The simplest way uses the rank of the matrix formed by the concatenation of the subspace and  $v$ :

► Example #4-37:

```

n = 3 ! below, 'A' is a well-known example of singular matrix
A = mf( reshape( [ (i, i=1,n*n) ], [n,n] ) )
call msDisplay(A,"A",mfRank(A),"rank(A)")

v1 = .t. mf( [ 5, 4, 3 ] )
call msDisplay(v1,"v1")
B = A .hc. v1
call msDisplay(B,"B",mfRank(B),"rank(B)")

v2 = .t. mf( [ 5, 4, 5 ] )
call msDisplay(v2,"v2")
B = A .hc. v2
call msDisplay(B,"B",mfRank(B),"rank(B)")

```

▷ Output of ex. #4-37:

```

A =

    1    4    7
    2    5    8
    3    6    9

rank(A) =

    2

v1 =

    5
    4
    3

B =

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

rank(B) =

    2

v2 =

    5
    4
    5

B =

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

rank(B) =

    3

```

In the previous example, the results show that  $v_1$  belongs to the image of the linear application  $A$  because the rank doesn't increase. On the contrary,  $v_2$  does belong to this image.

The other way is to use the basis of the image, computed by `mfOrth`, and to compute the residue of a linear system:

► Example #4-38:

```

Z = mfOrth(A)
call msDisplay( v1 - mfMul( Z, mfLDiv(Z,v1) ), "residue of (Z*x = v1)")
call msDisplay( v2 - mfMul( Z, mfLDiv(Z,v2) ), "residue of (Z*x = v2)")

```

▷ Output of ex. #4-38:

```

residue of (Z*x = v1) =

    1.0E-014 *

    -0.1776
    -0.2665
     0.0444

residue of (Z*x = v2) =

     0.3333
    -0.6667
     0.3333

```

The advantage of using the residue is that the result is a real number and not an integer.

The LU decomposition factorizes a matrix  $A$  in two factors  $L$  and  $U$  such that  $LU = A(p,:)$ . Both factors are triangular matrices.

► Example #4-39:

```

A = mf( reshape( [ 1, 7, 1, 2, 8, 2, 3, 9, 4 ], [ 3, 3 ] ) )
call msDisplay(A,"A")
call msLU( mfOut(L,U,p), A ) ! output args are always enclosed in mfOut()
call msDisplay(L,"L",U,"U",p,"p")
call msDisplay( mfNorm(mfMul(L,U)-mfRowPerm(A,p)), "|L*U - A(p,:)|" )

```

▷ Output of ex. #4-39:

```

A =

    1     2     3
    7     8     9
    1     2     4

L =

    1.0000    0.0000    0.0000
    0.1429    1.0000    0.0000
    0.1429    1.0000    1.0000

U =

    7.0000    8.0000    9.0000
    0.0000    0.8571    1.7143
    0.0000    0.0000    1.0000

p =

    2
    1
    3

|L*U - A(p,:)| =

    0

```

After a LU decomposition, you can use the two factors  $L$  and  $U$ , together with the permutation  $p$ , to solve a linear system of the form  $Ax = b$ .

► Example #4-40:

```

A = mf( reshape( [ 1, 7, 1, 2, 8, 2, 3, 9, 4 ], [ 3, 3 ] ) )
call msDisplay(A,"A")
b = .t. mf([ 6, 24, 7 ]) ! this is the right hand side
call msDisplay(b,"b")
call msLU( mfOut(L,U,p), A )
b = mfRowPerm(b,p) ! don't forget to apply the permutation to the vector b!
! solving A*x = b ! solution should be [1, 1, 1]'
x = mfLDiv(L,b) ! first left division
x = mfLDiv(U,x) ! second left division
call msDisplay( x, "solution vector x = A\b" )

```

▷ Output of ex. #4-40:



```

A =

    1    2    3
    7    8    9
    1    2    4

b =

    6
   24
    7

solution vector x = A\b =

    1
    1
    1

```

The singular value decomposition of a matrix  $A$  is obtained by calling the `msSvd` routine. This subroutine returns the three matrices  $U, S, V$  such that  $A = USV'$ . Another function `mfSvd` only returns the singular values:

► Example #4-41:

```

A = .t. mf( reshape( [ (i,i=1,9) ], [ 3, 3 ] ) )
call msDisplay(A,"A")
! a singular matrix have at least one singular value equal to zero
call msDisplay( mfSvd(A), "S" )

```

▷ Output of ex. #4-41:

```

A =

    1    2    3
    4    5    6
    7    8    9

S =

  16.8481
   1.0684
   0.0000

```

Other examples of linear algebra routines are `msChol`, `mfInv`, `mfNorm`, `mfCond` and `mfRank`.

#### 4.2.6 Data Analysis with mfArrays

The following routines, whose names are evident to understand, are available for working with `mfArrays`: `mfMin`, `mfMax`, `mfSum`, `mfProd`, `mfSort`, `mfFind`, `mfMean`, `mfMedian`, `mfVar`, `mfStd`, `mfRMS` and `msHist`. You can also apply the Fourier transform to your data in many variants: `mfFFT` (Fast Fourier Transform), `mfFourierCos` (Fourier-cosine), `mfFourierSin` (Fourier-sine), `mfFourierLeg` (Fourier-Legendre) and their inverse (resp. `mfInvFFT`, `mfInvFourierCos`, `mfInvFourierSin` and `mfInvFourierLeg`). Some of these routines have also a subroutine version, which allows generally a more sophisticated task. See the examples below.

## ► Example #4-42:

```

A = mfMagic(3)
call msDisplay(A,"A = mfMagic(3)")
call msDisplay(mfMin(A,1),"col min") ! returns a row vector
call msDisplay(mfMin(A,2),"row min") ! returns a column vector
! the scalar '4.0d0' is spread to the same shape than 'A'
call msDisplay(mfMin(A,4.0d0),"mfMin(A,4.0d0)")

```

## ▷ Output of ex. #4-42:

```

A = mfMagic(3) =

      8      1      6
      3      5      7
      4      9      2

col min =

      3      1      2

row min =

      1
      3
      2

mfMin(A,4.0d0) =

      4      1      4
      3      4      4
      4      4      2

```

## ► Example #4-43:

```

A = mfMagic(3)
call msDisplay(A,"A = mfMagic(3)")
! in a MUESLI subroutine call, 'mfOut' always encloses the output arguments
call msMax( mfOut(x,i), A, 1 ) ! works on columns
call msDisplay(x,"col max")
call msDisplay(i,"for index")

```

## ▷ Output of ex. #4-43:

```

A = mfMagic(3) =

      8      1      6
      3      5      7
      4      9      2

col max =

      8      9      7

for index =

      1      3      2

```

► Example #4-44:

```

A = mfMagic(3)
call msDisplay(A,"A = mfMagic(3)")
call msDisplay(mfSum(A,1),"col sum")
call msDisplay(mfSum(A,2),"row sum")
call msDisplay(mfSum(mfDiag(A)),"diag. sum")

```

▷ Output of ex. #4-44:

```

A = mfMagic(3) =

      8      1      6
      3      5      7
      4      9      2

col sum =

     15     15     15

row sum =

     15
     15
     15

diag. sum =

     15

```

► Example #4-45:

```

A = mfMagic(3)
call msDisplay(A,"A = mfMagic(3)")
call msDisplay(mfSort(A,1),"mfSort(A,1)") ! works on columns

```

▷ Output of ex. #4-45:

```
A = mfMagic(3) =
```

```

      8      1      6
      3      5      7
      4      9      2

```

```
mfSort(A,1) =
```

```

      3      1      2
      4      5      6
      8      9      7

```

► Example #4-46:

```

A = mf( [ 2, 0, 0 ] ) .vc. &
      mf( [ 0, 0, 1 ] ) .vc. &
      mf( [ 0, 3, 0 ] )
call msDisplay(A,"A")
! 'mfFind' returns indices of non-zero elements
long_ind = mfFind( A>1.0d0 ) ! works on a boolean mfArray
call msDisplay(long_ind,"(long column indices) mfFind( A>1.0d0 )")

```

▷ Output of ex. #4-46:

```
A =
```

```

      2      0      0
      0      0      1
      0      3      0

```

```
(long column indices) mfFind( A>1.0d0 ) =
```

```

      1      6

```

You can also use the **msFind** routine to directly get the  $(i,j)$  indices instead of long column indices.

MUESLI also provides some polynomial routines essentially to make interpolation. Firstly, cubic spline interpolation is available, via **mfSpline** and **mfPPVal**. This is a 1-D interpolation.

► Example #4-47:

```

x = [ 0.0d0, 0.5d0, 1.0d0 ]
y = [ 0.0d0, 1.0d0, 0.0d0 ]
call msDisplay(x,"x",y,"y")
pp = mfSpline( x, y )
! 'pp' actually contains the second derivatives of 'y' with respect to 'x'
call msDisplay( pp, "pp = mfSpline( x, y )" )

```

▷ Output of ex. #4-47:

```

x =
    0.0000    0.5000    1.0000

y =
    0     1     0

pp = mfSpline( x, y ) =
    0   -12    0

```

► Example #4-48:

```

xx = mfLinspace( -0.25d0, 1.25d0, 7 )
call msDisplay(xx, "xx")
! 'x', 'y' and 'pp' comes from the previous example
yy = mfPPVal( x, y, pp, xx )
call msDisplay( yy, "yy = mfPPVal( x, y, pp, xx )" )

```

▷ Output of ex. #4-48:

```

xx =
   -0.2500    0.0000    0.2500    0.5000    0.7500    1.0000    1.2500

yy = mfPPVal( x, y, pp, xx ) =
   -0.6875    0.0000    0.6875    1.0000    0.6875    0.0000   -0.6875

```

Secondly, there are also 2-D interpolation routines, either for a regular grid (**mfInterp2**) or for an irregular grid (**mfGridData**).

For **mfInterp2**, the interpolation method chosen by default is bilinear with respect to the two coordinates ( $x, y$ ). For example, coordinates and data are stored in rank-2 **mfArrays** **x**, **y**, **z**. **x** and **y** can be obtained from the **mfMeshGrid** routine. The output may be either a single point, either a vector of points, either a matrix of points according the shape of the **mfArrays** **xi** and **yi**.

► Example #4-49:

```

a = mfLinspace( 0.0d0, 1.0d0, 4 ); b = .t. mfLinspace( 0.0d0, 1.0d0, 3 )
call msMeshgrid( mfOut(x,y), a, b )
z = x
xi = mfColon( -0.1d0, 1.1001d0, step=0.2d0 ); yi = xi ! vector inputs
call msDisplay( x, "x", y, "y", xi, "xi", yi, "yi" )
zi = mfInterp2( x, y, z, xi, yi ) ! output is a vector
print *, "NaN are obtained for 'out-of-range' input points."
call msDisplay( zi, "interpolated data at points on a line" )

```

▷ Output of ex. #4-49:

```

x =
    0.0000    0.3333    0.6667    1.0000
    0.0000    0.3333    0.6667    1.0000
    0.0000    0.3333    0.6667    1.0000

y =
    0.0000    0.0000    0.0000    0.0000
    0.5000    0.5000    0.5000    0.5000
    1.0000    1.0000    1.0000    1.0000

xi =
   -0.1000    0.1000    0.3000    0.5000    0.7000    0.9000    1.1000

yi =
   -0.1000    0.1000    0.3000    0.5000    0.7000    0.9000    1.1000

NaN are obtained for 'out-of-range' input points.

interpolated data at points on a line =
    NaN    0.1000    0.3000    0.5000    0.7000    0.9000    NaN

```

**mfGridData** considers three vectors of type **mfArray**, which contain respectively the coordinates of irregular points and the attached values. It interpolates with a linear scheme.

► Example #4-50:

```

x = [ -0.15d0,  1.12d0, 1.10d0, -0.12d0, 0.5d0 ]
y = [ -0.11d0, -0.13d0, 1.14d0,  1.13d0, 0.5d0 ]
z = x
call msDisplay( .t.((x .vc. y) .vc. z), "nodes and values" )
xi = 0.1234d0 ! scalar input
yi = 0.9876d0
call msDisplay( xi, "xi", yi, "yi" )
zi = mfGridData( x, y, z, xi, yi ) ! output is a scalar
call msDisplay( zi, "interpolated data at a point" )

```

▷ Output of ex. #4-50:

```

nodes and values =

-0.1500  -0.1100  -0.1500
 1.1200  -0.1300   1.1200
 1.1000   1.1400   1.1000
-0.1200   1.1300  -0.1200
 0.5000   0.5000   0.5000

xi =

 0.1234

yi =

 0.9876

interpolated data at a point =

 0.1234

```

**mfGradient** is used to compute the gradient of a data vector. See the examples below.

► Example #4-51:

```

! Simple call to compute the gradient of a data vector: derivative is
! done with respect to increasing index if the step h is positive.
F = [ 0.0d0, 0.0625d0, 0.25d0, 0.5625d0, 1.0d0 ]
call msDisplay( F, "F" )
Fi = mfGradient( F, h=0.25d0 )
call msDisplay( Fi, "Fi = grad(F)" )

```

▷ Output of ex. #4-51:

```

F =

 0.0000   0.0625   0.2500   0.5625   1.0000

Fi = grad(F) =

 0.0000   0.5000   1.0000   1.5000   2.0000

```

The user is allowed to specify a negative value for the step  $h$ , especially when data is associated to a coordinate vector.

► Example #4-52:

```
! By taking a negative step 'h', the computation leads directly to the
! derivative with respect to 'x'.
F = [ 0.0d0, 0.5d0, 1.0d0, 1.5d0, 2.0d0 ]
x = [ 4.0d0, 3.0d0, 2.0d0, 1.0d0, 0.0d0 ]
call msDisplay( x, "x", F, "F" )
h = mfGet(x,2) - mfGet(x,1) ! automatic sign of h, taking into account
                             ! the ordering of the coordinate 'x'

Fx = mfGradient( F, h )
call msDisplay( h, "h", Fx, "Fx = grad_x(F)" )
```

▷ Output of ex. #4-52:

```
x =
      4      3      2      1      0

F =
      0.0000      0.5000      1.0000      1.5000      2.0000

h =
      -1

Fx = grad_x(F) =
      -0.5000      -0.5000      -0.5000      -0.5000      -0.5000
```

The **msGradient** subroutine must be used for 2D data matrices, because it outputs two **mfArrays**; but be aware that the gradient is computed with respect to increasing indices (see the *Muesli Reference Manual*). Therefore, when the data is associated to coordinates, a careful examination of the generator vectors leads to different formula for the determination of the step  $h$  according to the expected matrix orientation.

Referring to the section 5.5.1, *Matrix Orientation in Practice*, we consider the two main different cases:

1.  $v_x$  is a row vector and  $v_y$  is a column vector.  
Set  $h_i$  to  $v_y(2) - v_y(1)$ , and  $h_j$  to  $v_x(2) - v_x(1)$ , then get  $F_x = Fj$ , and  $F_y = Fi$ .
2.  $v_x$  is a column vector and  $v_y$  is a row vector.  
Set  $h_i$  to  $v_x(2) - v_x(1)$ , and  $h_j$  to  $v_y(2) - v_y(1)$ , then get  $F_x = Fi$ , and  $F_y = Fj$ .

Lastly, fitting (both for polynomial or any user-supplied function) may be easily done by using the routines **msPolyFit** and **msFunFit**.



### 4.2.7 Domain triangulation

Domain triangulation is often used in physics simulations before solving PDEs (Partial Differential Equations). The Muesli library contains a number of routines to triangulate a given domain, build a Delaunay triangulation from pre-existing nodes (`mfDelaunay`), and to search for nodes or triangles inside this triangulation (`mfNodeSearch`, `mfTriSearch`).

A computational domain is defined by use of the PL domain (Piecewise Linear domain) concept. In short, the user defines the boundary by line segments and, optionally, holes inside the domain. After that, the PL domain is used to generate a Delaunay triangulation.

The derived type `mfPLdomain` contains the different PL domain elements. An example follows:

► Example #4-53:

```
! L-shaped domain with three holes
allocate( PL_domain%n_xy(18,2) )
PL_domain%n_xy(:,1) = [ 0.0d0, 2.0d0, 2.0d0, 1.0d0, 1.0d0, 0.0d0, &
                        0.2d0, 0.8d0, 0.8d0, 0.2d0, &
                        0.2d0, 0.8d0, 0.8d0, 0.2d0, &
                        1.2d0, 1.8d0, 1.8d0, 1.2d0 ] ! x
PL_domain%n_xy(:,2) = [ 0.0d0, 0.0d0, 1.0d0, 1.0d0, 2.0d0, 2.0d0, &
                        1.2d0, 1.2d0, 1.8d0, 1.8d0, &
                        0.2d0, 0.2d0, 0.8d0, 0.8d0, &
                        0.2d0, 0.2d0, 0.8d0, 0.8d0 ] ! y

allocate( PL_domain%edge_n(18,2) )
PL_domain%edge_n(:,1) = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, &
                          13, 14, 15, 16, 17, 18 ]
PL_domain%edge_n(:,2) = [ 2, 3, 4, 5, 6, 1, 8, 9, 10, 7, 12, 13, &
                          14, 11, 16, 17, 18, 15 ]

allocate( PL_domain%holes_xy(4,2) )
PL_domain%holes_xy(1,:) = [ 1.25d0, 1.25d0 ]
PL_domain%holes_xy(2,:) = [ 0.50d0, 1.50d0 ]
PL_domain%holes_xy(3,:) = [ 0.50d0, 0.50d0 ]
PL_domain%holes_xy(4,:) = [ 1.50d0, 0.50d0 ]

call msPrintPLdomain( PL_domain )
```

The resulting structure may be checked (when the number of points is not too high) by the `msPrintPLdomain` routine:

▷ Output of ex. #4-53: (first part)

Planar Straight Line Graph

nn: number of nodes = 18  
 ne: number of edges = 18  
 nh: number of holes = 4

n\_xy(nn,2): nodes by 2D-coordinates

node	coordinates	
1	0.0000E+00	0.0000E+00
2	2.0000E+00	0.0000E+00
3	2.0000E+00	1.0000E+00
4	1.0000E+00	1.0000E+00
5	1.0000E+00	2.0000E+00
6	0.0000E+00	2.0000E+00
7	2.0000E-01	1.2000E+00
8	8.0000E-01	1.2000E+00
9	8.0000E-01	1.8000E+00
10	2.0000E-01	1.8000E+00
11	2.0000E-01	2.0000E-01
12	8.0000E-01	2.0000E-01
13	8.0000E-01	8.0000E-01
14	2.0000E-01	8.0000E-01
15	1.2000E+00	2.0000E-01
16	1.8000E+00	2.0000E-01
17	1.8000E+00	8.0000E-01
18	1.2000E+00	8.0000E-01

edge\_n(ne,2): edges by nodes

edge	nodes	
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	1
7	7	8
8	8	9
9	9	10
10	10	7
11	11	12
12	12	13
13	13	14
14	14	11
15	15	16
16	16	17
17	17	18
18	18	15

▷ Output of ex. #4-53: (second part)

```
holes_xy(nn,2): holes by 2D-coordinates
```

node	coordinates	
1	1.2500E+00	1.2500E+00
2	5.0000E-01	1.5000E+00
3	5.0000E-01	5.0000E-01
4	1.5000E+00	5.0000E-01

See the following images (21) in the next chapter for the resulting display of the previous PL domain by `msPlotPLdomain`.

#### 4.2.8 Other Numerical Tasks

With MUESLI, you can easily perform zero finding of functions (`mfFZero` for the single-variable case; `mfFSolve` for the multiple-variable case), nonlinear least-square minimization (`mfLsqNonLin`), integration of data (*i. e.* function integration or quadrature schemes) and integration of systems of ODEs and DAEs (*resp.* Ordinary Differential Equations and Differential Algebraic Equations).

The `mfFZero` routine searches by bisection the zero of a function given by the user:

► Example #4-54:

```
! definition of a user-function (actually in an external module)
function fun_sqr_minus_onehalf( x ) result( res )
  real(kind=MF_DOUBLE), intent(in) :: x
  real(kind=MF_DOUBLE) :: res
  res = x**2 - 0.5d0
end function fun_sqr_minus_onehalf
!-----
res = mfFZero( fun_sqr_minus_onehalf, x0=mf([ 0, 1 ]) )
call msDisplay( res, "numerical result" )
```

▷ Output of ex. #4-54:

```
numerical result =

0.7071
```

The `mfQuad` routine computes the numerical integration on a function, which must be supplied by the user. Let's compute, for example, the quantity:

$$\int_0^1 \sqrt{x} \, dx$$

► Example #4-55:

```

! definition of a user-function (actually in an external module)
function fun_sqrt( x ) result( res )
  real(kind=MF_DOUBLE), intent(in) :: x
  real(kind=MF_DOUBLE) :: res
  res = sqrt( x )
end function
!-----
res = mfQuad( fun_sqrt, 0.0d0, 1.0d0 ) ! integration of 'sqrt(x)' over [0,1]
call msDisplay( res, "numerical result" )

```

▷ Output of ex. #4-55:

```

numerical result =

0.6667

```

Therefore

$$\int_0^1 \sqrt{x} dx \approx 0.6667$$

On the other hand, `msQuad` allows the user to get more information returned by the integrator. Beside, MUESLI offers two other basic integrators: `mfTrapz` and `mfSimpson`.

The `mfLsqNonLin` routine find the parameters which minimize a function, written as the sum of the square of a vector's components. Let's consider the following module:

► Example #4-56:

```

module optim_funs
  use fml
  implicit none
contains
  subroutine exp_min( 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
    !---
    integer :: i
    if( p(1) < 0.0d0 ) then
      flag = +1 ! out-of-range condition to avoid an invalid result in sqrt
      return
    end if
    do i = 1, m
      fvec(i) = 2.0d0*i + i*(1.0d0-p(1)) + sqrt(i*p(1)) + exp(-i*p(2))
    end do
  end subroutine exp_min
end module optim_funs

```

The previous module defines the function to be minimized, `exp_min`, with respect to two parameters `p(1)` and `p(2)`. Note the presence of the `sqrt` function which is not defined for a negative parameter `p(1)`; this is why a flag equal to +1 has been returned when `p(1)` is found to be out-of-range (in other words, a constraint on `p(1)` is included in `fcn`). The call to the `mfLsqNonLin` routine could be written as simply as:

## ► Example #4-57:

```
! type(mfArray) :: p, z
m = 10
n = 2
z = mfLsqNonLin( m, exp_min, p, n )
call msDisplay( z, "solution" )
```

## ▷ Output of ex. #4-57:

```
(MUESLI mfLsqNonLin:) Warning: initial guess 'p0' is empty: trying to use
                        a zero vector!
(MUESLI mfLsqNonLin:) Warning: non zero flag at exit of fcn when evaluating
                        the jacobian ! (Return from fdjac2)
(MUESLI mfLsqNonLin:) ERROR: [minpack]/lmdif routine failed.
                        for your information: info = -9
```

However, as seen above, the previous call will returned an error ( $-9$ , which means that a constraint about an interval for one of the parameters cannot be satisfied, according the table of errors of `mfLsqNonLin` in the *Muesli Reference Manual*). The next call will lead to a solution, because the first component `p(1)` of the initial guess is not too close to the critical limit of definition of the `sqrt` function.

## ► Example #4-58:

```
m = 10
n = 2
p = [ 0.5d0, -0.5d0 ] ! initial guess
z = mfLsqNonLin( m, exp_min, p, n )
call msDisplay( z, "solution" )
```

## ▷ Output of ex. #4-58:

```
solution =

4.6201   -0.2304
```

The `mf0deSolve` routine solves an ODE system described by a user subroutine. A second order differential equation has been chosen for this purpose:

$$\frac{\partial^2 y(x)}{\partial x^2} = -y(x), \quad x > 0; \quad y(0) = 0, \quad y'(0) = 1$$

which must be decomposed into the following system of two first order differential equations:

$$\begin{aligned} y_1'(x) &= y_2(x) \\ y_2'(x) &= -y_1(x) \end{aligned}$$

Then, this system is in turn described in the `deriv` following subroutine. Below, the independent variable  $x$  is noted  $t$  like in the documentation.

## ► Example #4-59:

```

! definition of a user-subroutine (actually in an external module)
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
  yprime(1) = y(2)
  yprime(2) = -y(1)
end subroutine
!-----
! type(mfArray) :: t_0, t_end, t_span, y_0, y
print *, "integration of {y'' = - y} over [0,pi]:"
print *, " [ solution is: y(t) = sin(t) ]"
t_0 = 0.0d0
t_end = MF_PI
t_span = .t. mfLinSpace( t_0, t_end, 10 ) ! 10 values required for output
y_0 = [ sin(t_0), cos(t_0) ]
y = mfOdeSolve( deriv, t_span, y_0 )
call msDisplay( y, "numerical result: y(1), y(2)" ) ! Runge-Kutta Fehlberg

```

▷ Output of ex. #4-59:

```

integration of {y'' = - y} over [0,pi]:
 [ solution is: y(t) = sin(x) ]

numerical result: y(1), y(2) =

    0.0000    1.0000
    0.3420    0.9397
    0.6428    0.7660
    0.8660    0.5000
    0.9848    0.1737
    0.9848   -0.1736
    0.8660   -0.5000
    0.6428   -0.7661
    0.3420   -0.9397
    0.0000   -1.0000

```

By default, The *Runge-Kutta Fehlberg* method is used. It is suitable for most of ODE systems, except for stiff systems. Other available methods are *Adams-Bashforth-Moulton* (non-stiff systems) and *Backward Differentiation Formula* (stiff systems). Again, `msOdeSolve` allows the user to get more information returned by the ODE solver.

For implicit ODE systems (or DAE systems), you can use the `mfDaeSolve` routine, which has an interface similar to `mfOdeSolve`. But be aware that the current implementation can deal with DAE systems of differential-index 1 only (but this can change in the future... – note also, as stated in the *Reference Guide*, that solving higher-index systems may or may not work).

ODE and DAE systems are very common today; in particular, most of them arise from applying the method of lines to PDE systems. In this context, reading the book of Schiesser and Griffiths, 2009<sup>5</sup>, may be extremely beneficial for every physicist programmer (specifically, the examples provided with the book can be transposed easily using the MUESLI routines).

Some of the ODE or DAE systems are stiff. The user must then choose a BDF scheme which requires the Jacobian of the equations. This Jacobian can of course be approximated by solvers (via Finite

<sup>5</sup>Schiesser W. E., and Griffiths G. W., 2009. *A Compendium of Partial Differential Equation Models – Method of Lines Analysis with Matlab*, Cambridge University Press

Differences) but sometimes it is better to calculate it analytically, for example using a computer algebra system such as Maple and Maxima.

#### 4.2.9 Saving/Reading mfArrays to/from Disk

Saving data in a file is important for real applications. MUESLI provides several ways to save your **mfArrays**.

The most simple way is to save one **mfArray** in an ASCII file: **msSaveAscii** copies only the real part, and formats it under an array form. It is somewhat slow and not very efficient, because ASCII data leads to large file size. However, ASCII files can be read by any application on any platform, therefore it is a great advantage. The companion routine **mfLoadAscii**, reads an ASCII file (usually organized like a matrix but it is not strictly required), and stores it in an **mfArray**.

If you are looking for smaller file size, another way is to write data in a binary format. **msSave** copies an **mfArray** in a binary file using a special format (specific to the MUESLI library), because it stores also other information about the type of data, its structure and some matrix properties. As a consequence, the created file may be read only by the **msLoad** routine. The proposed extension for this kind of file is **.mbf** (MUESLI Binary File), but it is not strictly required because the binary file contains a special tag to be automatically recognized. Lastly, **msSave** can even write directly in a gzipped binary file: just provide a name which ends with **.mbf.gz**!

MUESLI provides also two routines, **msSaveHDF5** and **mfLoadHDF5**, to write and read **mfArrays** in HDF5 files. Many **mfArrays** can be stored in a single HDF5 file but currently in a flat way, not yet in a hierarchical way as possible in the HDF5 format.

See also in section 4.2.10 about sparse matrices to know how to save them in files, because MUESLI proposes other specific routines.

#### 4.2.10 Sparse mfArrays

Sparse structure for arrays are used when the number of non-zero elements is small in comparison with the total number of elements. This technique allows to reduce both memory space and floating-point operations. MUESLI uses internally the CSC format (Compact Sparse Columns), and most of routines involving sparse **mfArrays** are of course adapted to this special storage; on the other hand, MUESLI is able to read (from files) and import (from ordinary f90 arrays) other sparse format, such as COO (Coordinates) and CSR (Compact Sparse Row).

Sparse **mfArrays** doesn't arise automatically: the programmer has to create initially sparse arrays or has to convert explicitly dense **mfArrays** to sparse ones. Beside the two basic routines: **mfSpAlloc** (which create a new sparse **mfArray**) and **mfSparse** (which convert a dense **mfArray**), other specific routines are useful: **mfSpEye** (creates a sparse identity matrix) and **mfSpDiags** (create a sparse diagonal matrix from a vector).

► Example #4-60:

```
A = mfSpEye( 5 )
call msDisplay(A, "mfSpEye( 5 )")
```

▷ Output of ex. #4-60:

```
mfSpEye( 5 ) =

  real sparse matrix of size : 5, 5
  nz : 5, nzmax : 5
  (1,1)  1.0000E+00
  (2,2)  1.0000E+00
  (3,3)  1.0000E+00
  (4,4)  1.0000E+00
  (5,5)  1.0000E+00
```

To build a new sparse `mfArray`, you can also aggregate previously existing `mfArrays` using the concatenation operators `.vc.` and `.hc.` or, better, the routine `msHorizConcat`, as in the following example:

► Example #4-61:

```
! building a big (nrow,ncol) sparse mfArray from its columns
A = mfSpAlloc( nrow, 0 ) ! A is sparse but has zero columns
do i = 1, ncol
  ! <filling the dense column vector vec_col ...>
  call msHorizConcat( A, vec_col ) ! 'in place' operation
end do
```

If you don't have easily access to columns of the sparse matrix you want to build but only the rows, you can perform the same thing but in a transpose mode (because `msVertConcat` doesn't exist):

► Example #4-62:

```
! building a big (nrow,ncol) sparse mfArray from its rows
At = mfSpAlloc( ncol, 0 ) ! A-transposed is sparse but has zero columns
do i = 1, nrow
  ! <filling the dense row vector vec_row ...>
  call msHorizConcat( At, vec_row ) ! 'in place' operation
end do
! lastly, transpose the mfArray
A = .t. At
```

When working with sparse matrices, the routine `msDisplay` prints the actual size of the array, the number of non-zero values and the maximum number of elements. This last value can be modified (up or down) via the routine `msSpReAlloc`. You can also pack your sparse matrix to its minimum size via the same routine, by using the "minimal" argument (see the *MUESLI Reference Manual* document).

► Example #4-63:

```
v = .t. [ 1.0d0, 2.0d0, 3.0d0 ]
call msDisplay(v,"v")
B = mfSpDiags( 3, 5, v, 0 ) ! last arg. : zero-diagonal is the main diagonal
call msDisplay(B,"B = mfSpDiags( 3, 5, v, 0 )")
```

▷ Output of ex. #4-63:

```
v =

  1
  2
  3

B = mfSpDiags( 3, 5, v, 0 ) =

real sparse matrix of size : 3, 5
nz : 3, nzmax : 3
(1,1)    1.0000E+00
(2,2)    2.0000E+00
(3,3)    3.0000E+00
```



## ► Example #4-64:

```

A = mfSpAlloc(10,10,nzmax=30) ! used to reserve storage place in advance
print *, "A = mfSpAlloc(10,10,nzmax=30)"
call msDisplay(mfShape(A), "mfShape(A)")
call msDisplay(mfSize(A), "mfSize(A)")
call msDisplay(mfNzmax(A), "mfNzmax(A)")
call msDisplay(mfNnz(A), "mfNnz(A)")

```

## ▷ Output of ex. #4-64:

```

A = mfSpAlloc(10,10,nzmax=30)

mfShape(A) =

    10    10

mfSize(A) =

    100

mfNzmax(A) =

    30

mfNnz(A) =

    0

```

## ► Example #4-65:

```

A = mfEye(5,5)
call msSet( -1.0d0, A, 1, 5 )
call msSet( -1.0d0, A, 5, 1 )
! A has only 7 non zero element over 25
B = mfSparse(A)
call msDisplay(B, "B = mfSparse(A)")

```

## ▷ Output of ex. #4-65:

```

B = mfSparse(A) =

    real sparse matrix of size : 5, 5
    nz : 7, nzmax : 7
    (1,1)    1.0000E+00
    (5,1)   -1.0000E+00
    (2,2)    1.0000E+00
    (3,3)    1.0000E+00
    (4,4)    1.0000E+00
    (1,5)   -1.0000E+00
    (5,5)    1.0000E+00

```

MUESLI also provides exchange routine: `mfSpImport` and `msSpExport` which support many format (COO, CSC and CSR).

► Example #4-66:

```
! integer, allocatable :: ir(:), jc(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

allocate( ir(3), jc(3), val(3) )
ir(:) = [ 2, 3, 4 ]
jc(:) = [ 2, 4, 3 ]
val(:) = [ 1.0d0, 2.0d0, 3.0d0 ]
A = mfSpImport(ir,jc,val) ! default: COO format is used
call msDisplay(A, "A = mfSpImport(ir,jc,val)")
```

▷ Output of ex. #4-66:

```
A = mfSpImport(ir,jc,val) =

  real sparse matrix of size : 4, 4
  nz : 3, nzmax : 3
  (2,2)    1.0000E+00
  (4,3)    3.0000E+00
  (3,4)    2.0000E+00
```

► Example #4-67:

```
! integer, allocatable :: ptr(:), j(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

A = mf( [ 0, 3, 4, 0, 0 ] ) .vc. &
    mf( [ 1, 0, 0, 6, 0 ] ) .vc. &
    mf( [ 0, 0, 0, 0, 7 ] ) .vc. &
    mf( [ 2, 0, 5, 0, 0 ] )

call msDisplay(A,"A")
A = mfSparse(A)

nrow = mfGet( mfShape(A), 1 )
nnz = mfNnz(A)
allocate( ptr(nrow+1), j(nnz), val(nnz) )
call msSpExport( A, ptr, j, val, format="CSR" )
print *, "exporting 'A' to CSR format:"
print *, " ptr = ", ptr
print *, " j   = ", j
print *, " val = ", val
```

▷ Output of ex. #4-67:

```

A =

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

exporting 'A' to CSR format:
ptr = 1 3 5 6 8
j   = 2 3 1 4 5 1 3
val = 3. 4. 1. 6. 7. 2. 5.

```

To modify an element of a sparse `mfArray` you can use, as for a dense `mfArray`, the routine `msSet`. However, it must be emphasized that modifying a single value (especially when the entry doesn't exist) is very inefficient; the following strategy should then be used instead (using `mfGet` and `msSet`): get the corresponding column, change the value (or better, update all values which must be changed) and put back the modified column in the `mfArray` `x`. Indeed, modifying a whole column is more efficient than modifying a whole row.

Additionally, `mfFull` is used to convert from sparse to dense.

Nearly all arithmetic operations and most of routines can be applied transparently to sparse `mfArrays`. In the contrary, you will be warned by a specific message. For example, solving a sparse linear system of equation is done by the usual way (as already seen in section 4.2.5):

► Example #4-68:

```

! integer, allocatable :: ir(:), jc(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

allocate( ir(12), jc(12), val(12) )
ir(:) = [ 1, 2, 1, 3, 5, 2, 3, 4, 5, 3, 2, 5 ]
jc(:) = [ 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 5, 5 ]
val(:) = [ 2.0d0, 3.0d0, 3.0d0, -1.0d0, 4.0d0, 4.0d0, -3.0d0,      &
          1.0d0, 2.0d0, 2.0d0, 6.0d0, 1.0d0 ]

A = mfSpImport(ir,jc,val)
call msDisplay(mfFull(A), "Sparse matrix A")

b = mfMul( A, .t. mf( [ 1, 2, 3, 4, 5 ] ) )
call msDisplay(b, "b [RHS]")

x = mfLDiv( A, b ) ! for sparse systems, UMFPack library is used
call msDisplay(x, "x [sol.]")

```

▷ Output of ex. #4-68:

```

Sparse matrix A =

    2    3    0    0    0
    3    0    4    0    6
    0   -1   -3    2    0
    0    0    1    0    0
    0    4    2    0    1

b [RHS] =

    8
   45
   -3
    3
   19

x [sol.] =

    1.0000
    2.0000
    3.0000
    4.0000
    5.0000

```

Generally operations between mixed storage structures are not allowed (you must use the same storage structure), except for the matrix-vector product, which can be sparse/dense, and few others.

If you have any doubt about the sparsity of an `mfArray`, use the inquiry logical functions `mfIsSparse` and `mfIsDense`.

► Example #4-69:

```

A = mfSpEye(1)
print *, "mfIsSparse(mfSpEye(1)) = ", mfIsSparse(A)

```

▷ Output of ex. #4-69:

```
mfIsSparse(mfSpEye(1)) =  T
```

Sparse matrices have generally a big number of rows and columns. So, when computing eigenvalues or singular values, we like to find not all values, but only some of them. This can be done with `mfEigs` and `mfSVDS` (if you obtain some convergence problems, you should use the `'ncv='` option):

► Example #4-70:

```

! integer, allocatable :: ir(:), jc(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

allocate( ir(9), jc(9), val(9) )
ir(:) = [ 1, 1, 2, 2, 3, 4, 4, 5, 5 ]
jc(:) = [ 1, 5, 2, 4, 3, 2, 4, 1, 5 ]
val(:) = [ 2.0d0, 1.0d0, 1.0d0, -3.0d0, 1.0d0, -3.0d0, 1.0d0, 1.0d0, 2.0d0 ]

A = mfSpImport(ir,jc,val)
call msDisplay(mfFull(A), "Sparse matrix A")

d = mfEigs( A, 3 ) ! ARPACK is used (which ? Largest Magnitude by default)
call msDisplay(d, "Largest eigenvalues from mfEigs()")

```

▷ Output of ex. #4-70:

```

Sparse matrix A =

      2      0      0      0      1
      0      1      0     -3      0
      0      0      1      0      0
      0     -3      0      1      0
      1      0      0      0      2

Largest eigenvalues from mfEigs() =

      4.0000
      3.0000
     -2.0000

```

► Example #4-71:

```

! integer, allocatable :: ir(:), jc(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

allocate( ir(9), jc(9), val(9) )
ir(:) = [ 1, 1, 2, 2, 3, 4, 4, 5, 5 ]
jc(:) = [ 1, 5, 2, 4, 3, 2, 4, 1, 5 ]
val(:) = [ 2.0d0, -1.0d0, 1.0d0, -3.0d0, 1.0d0, 3.0d0, 1.0d0, 1.0d0, 2.0d0 ]

A = mfSpImport(ir,jc,val)
call msDisplay(mfFull(A), "Sparse matrix A")

d = mfSVDS( A, 3, which="SM" ) ! which ? Smallest Magnitude
call msDisplay(d, "Smallest singular values from mfSVDS()")

```

▷ Output of ex. #4-71:

Sparse matrix A =

```

  2    0    0    0   -1
  0    1    0   -3    0
  0    0    1    0    0
  0    3    0    1    0
  1    0    0    0    2

```

Smallest singular values from mfSVDS() =

```

1.0000
2.2361
2.2361

```

To perform IO operations, both `msSave` and `mfLoad` (described in section 4.2.9) support sparse `mfArrays`. Moreover, for ASCII format, there are `msSaveSparse` and `mfLoadSparse`, using different ASCII formatting:

- "CSC" for "Compact Sparse Column" (default)
- "CSR" for "Compact Sparse Row"
- "HBO" for "Harwell-Boeing"
- "MTX" for "Matrix Market"

The first two formats "CSC" or "CSR", correspond to a write of a first line of 5 objects, then the write of the three vectors: pointer to indices, indices and values. See the following example:

► Example #4-72:

```

! integer, allocatable :: ir(:), jc(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

allocate( ir(9), jc(9), val(9) )
ir(:) = [ 5, 1, 1, 2, 2, 3, 4, 4, 5 ]
jc(:) = [ 1, 1, 5, 2, 4, 3, 2, 4, 5 ]
val(:) = [ 8.0d0, 1.0d0, 2.0d0, 3.0d0, 4.0d0, 5.0d0, 6.0d0, 7.0d0, 9.0d0 ]

A = mfSpImport(ir,jc,val)
call msDisplay(mfFull(A),"Sparse Matrix A")
call msSaveSparse( "A.csc", A, format="CSC" )

```

▷ Output of ex. #4-72:

Sparse Matrix A =

```

  1    0    0    0    2
  0    3    0    4    0
  0    0    5    0    0
  0    6    0    7    0
  8    0    0    0    9

```

Then, the file "A.csc" contains the following data:

```

5 5 1 csc real
1 3 5 6 8 10
1 5 2 4 3 2 4 1 5
1.000000000000000E+00 8.000000000000000E+00 3.000000000000000E+00
6.000000000000000E+00 5.000000000000000E+00 4.000000000000000E+00
7.000000000000000E+00 2.000000000000000E+00 9.000000000000000E+00

```

The first line of the previous file contains the shape of the sparse matrix (5,5), followed by the "row sorted" property (here 1, because this property is true), then a keyword describing the compact format used ("csc" or "csr") and last the type of the data ("real" or "complex").

Matrix Market format ("MTX") uses the coordinates format ("COO").

Harwell-Boeing format ("HBO") is very similar to "CSC" or "CSR" format, except that the header is different and it is more compact because there is no blanks between indices and numbers; as a consequence, this format is not always suitable for a human read.

The internal sparse format used (CSC format) doesn't need the elements to be row sorted in each column. However, some sparse algorithms are more efficient when columns of sparse matrices are row sorted. By default, MUESLI automatically sorts the rows when needed; use `msSetAutoRowSorted` to change this default behavior. See also `mfIsRowSorted` and `msRowSort` in the *MUESLI Reference Manual*.

## 4.3 Advanced topics

### 4.3.1 Matrix Properties

We have seen in section 4.2.5 that some high-level linear algebra routines check for some properties of an `mfArray` before applying the specific Lapack routine. These properties are the matrix symmetry and its definite positiveness (which are theoretically not dependent). Sometimes, the user may find useful to get these properties, which are stored in the internal structure of the `mfArray`. To do that, two inquiry functions are available: `mfIsSymm` and `mfIsPosDef`.

► Example #4-73:

```

A = mfHilb(5) ! Hilbert matrices are always symmetric
print *, "Is A symmetric ? ", mfIsSymm( A )
B = mfMagic(5) ! Magic squares are never symmetric
print *, "Is B symmetric ? ", mfIsSymm( B )

```

▷ Output of ex. #4-73:

```

Is A symmetric ?  T
Is B symmetric ?  F

```

► Example #4-74:

```

A = mfHilb(5) ! Hilbert matrices are always positive definite
print *, "Is A positive definite ? ", mfIsPosDef( A )
B = mfOnes(5)
print *, "Is B positive definite ? ", mfIsPosDef( B )

```

▷ Output of ex. #4-74:

```
Is A positive definite ?  T
Is B positive definite ?  F
```

Sparse `mfArrays` have an additional property (see section 4.2.10).

#### 4.3.2 Physical Units in `mfArrays`

When doing some computations, a physicist manipulates only numerical quantities and loses the physical units. Most of the time, he has prepared his equations before and knows exactly what he has coded. Unfortunately, if his equations are not physically consistent, this will not be detected at run-time and the results will be wrong.

MUESLI lets the user to assign each `mfArray` (not of type boolean) a physical unit. These physical units are composed together during run-time execution and any physical inconsistency will be detected and signaled as soon as it arises.

By default, MUESLI doesn't take physical units into consideration. The user must call the `msUsePhysUnits` routine to make them active.

The `msDisplay` routine takes care of the physical unit, if present, and prints inside brackets the decomposition of this unit into the seven following fundamental dimensions (all in capital letters):

- M (mass)
- L (length)
- T (time)
- $\theta$  (thermodynamic temperature)
- I (electric current)
- N (amount of substance)
- J (luminous intensity)

► Example #4-75:

```
call msUsePhysUnits( "on" ) ! default is "off"
x = 2.0d0
call msSetPhysDim( x, Mass=1.0d0/2, Length=-7.0d0/2, Time=1.0d0/3 )
call msDisplay( x, "x" )
```

▷ Output of ex. #4-75:

```
x =
[ M^(1/2) L^(-7/2) T^(1/3) ]

      2
```

Predefined common units are useful to set physical unit in an easier way:

► Example #4-76:



```

! type(mfUnit) :: u_Ly
call msUsePhysUnits( "on" )
y = 0.25d0*u_kg ! 'u_kg' is a predefined unit
call msDisplay( y, "y" )

u_Ly = u_calorie / u_cm**2 / u_min ! new user-defined unit
call msSetPhysUnitAbbrev( u_Ly, "Langley" )

x = 1.95d0 * u_Ly
call msDisplay( x, "x", unit=u_Ly )
call msDisplay( x, "", unit=SI_unit )

```

▷ Output of ex. #4-76:

```

y =
[ M ]

    0.2500

x =
[ M T^-3 ] (Langley)

    1.9500

=
[ M T^-3 ] (S.I.)

    1.3585E+03

```

As seen in the previous example, you can combine predefined units in order to build new ones. You can even attach the abbreviation of your choice to the new user-defined `mfUnit`. When combining `mfUnits`, the following operators can be used: `*`, `/` and `**` (integer or real exponent). When using the exponentiation operator (`**`) with a real exponent, be aware that the exponent will be converted to a 2-bytes based rational number: this means that `0.33333d0` will be replaced by `1/3` but that `0.3333d0` will be replaced by `3333/10000`; in this later example, the best practice is to employ `(1.0d0/3.0d0)` or `(1.0d0/3)` as exponent.

Hereafter is the list of all predefined variables (`units`, `multipliers`, `constants` and conversion `factors`):

- predefined units begin with the `u_` prefix:

```

u_kg = u_kilogram : kilogram (mass)
u_m = u_meter : meter (length)
u_s = u_second : second (time)
u_K = u_kelvin : kelvin (temperature)
u_A = u_ampere : ampere (electric current)
u_mol = u_mole : mole (amount of substance)
u_cd = u_candela : candela (luminous intensity)

u_rad : radian (rad)
u_steradian : steradian (sr)
u_Hertz : hertz (Hz)
u_Newton : newton (N)
u_Pascal : pascal (Pa)
u_Joule : joule (J)
u_Watt : watt (W)

```

```

u_Coulomb : coulomb (C)
u_Volt : volt (V)
u_Farad : farad (F)
u_Ohm : ohm ( $\Omega$ )
u_Siemens : Siemens (S)
u_Weber : weber (Wb)
u_Tesla : tesla (T)
u_Henry : henry (H)
u_degree_Celsius : degree Celsius ( $^{\circ}\text{C}$ )
u_lumen : lumen (lm)
u_lux : lux (lx)
u_Becquerel : becquerel (Bq)
u_gray : gray (Gy)
u_sievert : sievert (Sv)
u_katal : katal (kat)

u_mg : milligram (mg)
u_cg : centigram (cg)
u_g : gram (g)
u_angstrom : angström ( $\text{\AA}$ )
u_micron : micrometer ( $\mu\text{m}$ )
u_mm : millimeter (mm)
u_cm : centimeter (cm)
u_km : kilometer (km)
u_astronomical_unit : astronomical unit (au)
u_min = u_minute : minute of time (min)
u_hr = u_hour : hour (h)
u_day : day (d)
u_liter : liter (L)
u_millimeter_Hg : millimeter Hg (mm Hg)
u_bar : bar (bar)
u_atm : atmosphere (atm)
u_psi : pound per square inch (psi)
u_calory : calory (cal)
u_BTU : british thermal unit (BTU)
u_Poiseuille : poiseuille (Pl)

```

– multipliers begin with the `m_` prefix:

```

m_yotta =  $10^{24}$ 
m_zetta =  $10^{21}$ 
m_exa =  $10^{18}$ 
m_peta =  $10^{15}$ 
m_tera =  $10^{12}$ 
m_giga =  $10^9$ 
m_mega =  $10^6$ 
m_kilo =  $10^3$ 
m_hecto =  $10^2$ 
m_deca =  $10^1$ 
m_deci =  $10^{-1}$ 
m_cent =  $10^{-2}$ 
m_milli =  $10^{-3}$ 
m_micro =  $10^{-6}$ 
m_nano =  $10^{-9}$ 
m_pico =  $10^{-12}$ 
m_femto =  $10^{-15}$ 
m_atto =  $10^{-18}$ 
mzepto =  $10^{-21}$ 

```

`m_yocto = 10-24`

- physical constants begin with the `c_` prefix:

`c_speed_of_light` : speed of light [ $LT^{-1}$ ]  
`c_Planck` : Planck constant [ $ML^2T^{-1}$ ]  
`c_Avogadro` : Avogadro number [ $N^{-1}$ ]  
`c_universal_gas` : universal gas constant [ $ML^2T^{-2}\Theta^{-1}N^{-1}$ ]  
`c_Boltzmann` : Boltzmann constant [ $ML^2T^{-2}\Theta^{-1}$ ]  
`c_electron_charge` : electron charge [ $TI$ ]  
`c_gravity` : gravity constant [ $M^{-1}L^3T^{-2}$ ]  
`c_gravity_accel` : gravity acceleration [ $LT^{-2}$ ]

- conversion factors begin with the `f_` prefix:

`f_kelvin_conversion` : temperature of triple point of water [ $\Theta$ ]

When printing a physical quantity, you may change its displayed unit. Note that to avoid confusion, the name of the required unit is added in parenthesis. If you don't know the S.I. name of a quantity, you can let MUESLI try to find it, by using '`unit=SI_unit`' as additional argument to `msDisplay`

► Example #4-77:

```
! call msUsePhysUnits( "on" )
L = u_m ! length
call msDisplay( L, "length", unit=u_cm ) ! should be 100 centimeters
F = u_Newton ! force
call msDisplay( F, "force", unit=SI_unit )
A = L*L ! area
call msDisplay( A, "area", unit=SI_unit )
P = F/A ! pressure
call msDisplay( P, "pressure", unit=SI_unit )
```

▷ Output of ex. #4-77:

```
length =
[ L ] (cm)

100

force =
[ M L T^-2 ] (N)

1

area =
[ L^2 ] (m^2)

1

pressure =
[ M L^-1 T^-2 ] (Pa)

1
```

Finally, you can check whether a quantity is dimensionless, or whether two quantities share the same

physical dimensions:

► Example #4-78:

```
x = 2.0d0 * u_m
y = 3.5d0 * u_cm
print *, "Do 'x' and 'y' share the same phys. dim. ? ", mfHaveSamePhysDim(x,y)
z = x/y
print *, "Is 'z' dimensionless ? ", mfHasNoPhysDim(z)
```

▷ Output of ex. #4-78:

```
Do 'x' and 'y' share the same phys. dim. ? T
Is 'z' dimensionless ? T
```

### 4.3.3 Performance Issues

#### Avoiding copy of a temporary array

When you use, for example, the following form of assignment:

```
A = mfRand(N,N)
```

MUESLI first creates a temporary array storing the random values and then, copy this array into A. You can be warned of this copy by setting the message level to an appropriate level, *i. e.* inserting the following in your source code:

```
call msSetMsgLevel(3)
```

In such a case, you will see as output:

```
(MUESLI assignment(=):) info: you are using the simple form of
assignment 'a=b', but with the tempo RHS 'b'.
You could use the other form : 'call msAssign(a,b)',
which only points to data, and therefore is much more
efficient !
```

(you cannot simply locate the concerned statement, because the routine `msSetTrbLevel` doesn't apply to the *info* level). To improve performance, you may use the subroutine form of the assignment:

```
call msAssign( A, mfRand(N,N) )
```

which avoid such a copy. More explanation can be found in the *MUESLI inside* document.

#### Counting Floating-Point Operations

Counting Floating-Point Operations (or, in short, flops) is possible in MUESLI only via hardware counters. Therefore, you must have some software installed to access these counters. On linux, at least for the 2.4 and 2.6 series, the kernel must be patched with 'perfctr' and the PAPI library must be installed.

If your system contains the PAPI library, don't forget to tell to MUESLI its path by updating the corresponding variable in the MUESLI config file. See section 3 (Installation / Papi installation) in the *MUESLI Installation Guide*.

At any place in your program, you must initialize the counter, by the `msFlops` routine, then calling it again to obtain a flops count. Typically, flops counts are used to measure performance of numerical algorithms because they lead to more accurate results than using the CPU time, especially for short computations.

► Example #4-79:

```

! integer*8 :: flops
print *, "one FP operation :"
call msFlops(init=0) ! flops initialization
x = 1.0d0 + 1.0d0
call msFlops(count=flops)
print *, "flops = ", flops

```

▷ Output of ex. #4-79:

```

one FP operation :
flops = 1

```

► Example #4-80:

```

! integer*8 :: flops
print *, "matrix product 100x100 by 100x100 :"
x = mfOnes(100,100)
y = mfOnes(100,100)
call msFlops(init=0) ! flops initialization
u = mfMul( x, y )
call msFlops(count=flops)
print *, "flops = ", flops

```

▷ Output of ex. #4-80:

```

matrix product 100x100 by 100x100 :
flops = 2000000

```

If you obtain errors in getting flops (typically, all calls to `msFlops` returns the '-1' value), see again the *MUESLI Installation Guide*.

#### 4.3.4 Managing Memory

The fundamental derived type used in this library, the `type(mfArray)`, consists actually of several storage arrays declared as pointer<sup>6</sup>. Deallocation of pointer objects is not automatic, so it is under the responsibility of the programmer. Most of time, these deallocations are made by the library itself, however it remains some cases where it must be made inside the user program. For this reason, it is a good practice to deallocate an `mfArray` as soon as it becomes unused.

The deallocation of all internal storage arrays in an `mfArray` is made by using the `msRelease` routine:

► Example #4-81:

```

A = mfRand(100,100)
! ... (computation)
call msRelease( A )

```

If you don't actually do this deallocation, you could obtain a warning message at the end of your program execution (according your compiler). Further information about this topic can be found in the *MUESLI inside* document.

---

<sup>6</sup>see the *MUESLI inside* document

### 4.3.5 Creating your own mf- or ms- Routines

You can create your own routines to manipulate `mfArray`s but you should be aware of some important things:

- you should firstly remember that MUESLI can automatically deallocate temporary objects, so you must protect the `mfArray` arguments of your routine. This can be realized by using `msInitArgs`. You must also unprotect them at the end of the routine by using the `msFreeArgs` routine.
- secondly, if your routine is a function that returns an `mfArray`, this latter object must be tagged as temporary by `msReturnArray`.

The following example show a function which returns the symmetric part of a matrix.

► Example #4-82:

```

function sym_part( A ) result( out )
  type(mfArray) :: A
  type(mfArray) :: out
  ! returns the symmetric part of the mfArray 'A'
  integer :: s(2), i, j, n
  type(mfArray) :: alpha, beta

  call msInitArgs( A ) ! protects the input argument
  !-----
  ! input matrix must not be empty
  if( mflIsEmpty(A) ) then
    print *, "sym_part: empty matrix."; go to 99
  end if

  ! input matrix must be square
  s = Shape(A)
  if( s(1) /= s(2) ) then
    print *, "sym_part: non square matrix."; go to 99
  end if

  n = s(1)
  if( n == 1 ) then
    ! trivial case
    out = A
  else
    out = mfZeros(n)
    do i = 1, n
      do j = 1, i-1
        alpha = mfGet(A,i,j)
        beta  = mfGet(A,j,i)
        call msSet( 0.5d0*(alpha+beta), out, i, j )
        call msSet( 0.5d0*(alpha+beta), out, j, i )
      end do
      ! diagonal
      call msSet( mfGet(A,i,i), out, i, i )
    end do
  end if

  call msRelease( alpha, beta )
  !-----
99 continue
  call msFreeArgs( A )      ! unprotects the input argument ...
  call msAutoRelease( A )   ! ... and deallocate it if needed
  call msReturnArray( out ) ! marks the output as temporary

end function sym_part

```

### 4.3.6 Numerical Precision

All internal computations are done in double precision (see the *IEEE-754* standard).

### 4.3.7 Debugging in MUESLI

The following routines may help in getting information about the internal behavior of the library: `mfGetMsgLevl`, `msSetMsgLevl`, `mfGetTrbLevel`, `msSetTrbLevel`. Information about these routines can be found in the *MUESLI Reference Manual* document. See also the *MUESLI Inside* document.

## Locating your run-time errors...

Because MUESLI uses automatic structures in the `mfArray` derived type, you will face, sooner or later, run-time errors – note that they cannot be detected at compile-time. Though locating these errors is possible by using a debugger (for example, GNU `gdb`), it may lead to problems because each Fortran 90 optional argument has a null address when they are not used, and this sometimes hangs the debugger...

Since the beginning of its design, the library MUESLI shows you the stack of called routines when such errors arise. This is the “backtrace” feature. This facility may even be used by the user himself, by calling the `msMuesliTrace()` routine (see more information in the *MUESLI Reference Manual* document).

## Detecting Floating-Point instabilities

The `msSetRoundingMode` routine may be used to change the rounding mode used by the processor to perform floating-point operations. From a global point of view, it can be judicious to change the rounding mode if you suspect some instabilities in your code. If numerical results change a lot between runs using different rounding modes, you will be able to conclude to a floating-point instability.

However, you will not be able to find the exact location of such floating-point instabilities with MUESLI: specialized libraries, like *CADNA*<sup>7</sup>, are devoted to this kind of difficult problem.

## Detecting *Inf*s and *NaN*s

Nowadays, like default other programs' behaviour, MUESLI operates in full IEEE-754 mode, *i. e.* doesn't stop on floating-point exceptions but produces *Inf*s and *NaN*s (resp. Infinities and Not-a-Number). If you are interested in finding which statement creates these special IEEE values, you must use the following routine: `msEnableFPE`, which is usually called at the beginning of your program.

However, *Inf*s and *NaN*s are sometimes expected (*e. g.*, *NaN*s may represent holes in data whereas *Inf* is the valid and expected result of  $\tan(\pi/2)$ ); so you may want to prevent the exception trapping for certain parts of your code, by using the other routine: `msDisableFPE`.

► Example #4-83:

```
!real :: tab(1000)
!type(mfArray) :: x, y

call msEnableFPE( "usual_exceptions" )

... ! some computation

call msDisableFPE( "usual_exceptions" )
x = 1.0d0/0.0d0 ! no FP exception because trapping is disabled by the
               ! previous statement
call msEnableFPE( "usual_exceptions" )

x = 0.0d0
y = 1/x ! no FP exception because the current division becomes a MUESLI
        ! operation (operator overload), for which FP exceptions are
        ! usually internally disabled.

print *, "using uninitialized data..."
tab(:) = acos(tab(:)) ! FP exception because tab(:) is not initialized and
                     ! is not an mfArray. One can be pretty sure that at
                     ! least one value of tab(:) is out of [-1,1]
```

▷ Output of ex. #4-83: (when using the GNU Fortran compiler)

<sup>7</sup><http://www-anp.lip6.fr/cadna/>



```
using uninitialized data...
```

```
Program received signal SIGFPE: Floating-point exception - erroneous arithmetic
operation.
```

As shown in the previous example, all internal routines are protected against FP exceptions, so it is not always easy to detect programmer's defects. Moreover, you could wonder what is the behaviour of MUESLI when using inverse trigonometric functions in your program: by default, when it is possible, a complex value is returned like in the following example (but this default behaviour can be changed, via the `msSetAutoComplex` routine):

► Example #4-84:

```
!type(mfArray) :: x

x = 1.5d0
call msDisplay( x, "x" )

print *, "Auto-Complex conversion = ", mfGetAutoComplex()

call msDisplay( mfACos(x), "mfACos(x)" )

call msSetAutoComplex( .false. )
print *, "Auto-Complex conversion = ", mfGetAutoComplex()

call msDisplay( mfACos(x), "mfACos(x)" )

call msEnableFPE( "usual_exceptions" )

call msDisplay( mfACos(x), "mfACos(x)" )
```

▷ Output of ex. #4-84: (when using the GNU Fortran compiler)

```
x =
    1.5000

Auto-Complex conversion =  T

mfACos(x) =
    0.0000 + 0.9624i

Auto-Complex conversion =  F

mfACos(x) =
    NaN

Program received signal SIGFPE: Floating-point exception - erroneous arithmetic
operation.

Backtrace for this error:
...
```



## Numerical checks

The logical global variable `MF_NUMERICAL_CHECK` can be set by the user. It is useful for the ODE/DAE solvers environment (see below).

One interesting feature is that, when this global variable is set to `.true.`, the library can check the validity of a user-defined jacobian (see the appropriate option in the `mf.DE.Options` entry of the *Muesli Reference Manual*). Moreover, the corresponding line number of the jacobian is displayed, and, if the “named group” feature is used, it is easier for the user to find his error in the system of equations. Hereafter is displayed an example of such a case.

► Example #4-85:

```
! defining 7 groups of equations (these are physical groups)
allocate( options%named_eqn(7) )
options%named_eqn(1)%name = "Mass conservation of water vapor"
options%named_eqn(1)%begin = 2
options%named_eqn(1)%last = N-1
options%named_eqn(2)%name = "Energy conservation"
options%named_eqn(2)%begin = 2 + N
options%named_eqn(2)%last = N-1 + N
options%named_eqn(3)%name = "State law for water vapor"
options%named_eqn(3)%begin = 1 + 2*N
options%named_eqn(3)%last = N + 2*N
options%named_eqn(4)%name = "Boundary Condition #1 (at top)"
options%named_eqn(4)%begin = 1
options%named_eqn(4)%last = 1
options%named_eqn(5)%name = "Boundary Condition #2 (at top)"
options%named_eqn(5)%begin = 1 + N
options%named_eqn(5)%last = 1 + N
options%named_eqn(6)%name = "Boundary Condition #3 (at bottom)"
options%named_eqn(6)%begin = N
options%named_eqn(6)%last = N
options%named_eqn(7)%name = "Boundary Condition #4 (at bottom)"
options%named_eqn(7)%begin = N + N
options%named_eqn(7)%last = N + N
!-----
! defining now 3 differents variables (optional)
j = 0
allocate( options%named_var(3) )
j = j + 1
options%named_var(j)%name = "Temperature (temp)"
options%named_var(j)%begin = 1
options%named_var(j)%last = N
j = j + 1
options%named_var(j)%name = "Gas pressure (p_g)"
options%named_var(j)%begin = 1 + N
options%named_var(j)%last = N + N
j = j + 1
options%named_var(j)%name = "Gas density (rho_g)"
options%named_var(j)%begin = 1 + 2*N
options%named_var(j)%last = N + 2*N
```

In this example, a DAE system is solved using the `msDaeSolve` routine (after applying the method of line to a PDAE system): `N` is the total number of discretized equations. Three main physical equations are used, plus four boundary conditions; so the total number of equations group is 7. Besides, there 3 (groups of) variables.

In case where the main system of equations (`deriv` or `resid` routine) contains a bug (*e.g.*, a floating-point

exception arise somewhere), the library may output a message like the following:

▷ Output of ex. #4-85:

```
(MUESLI Daesolve:) [BDF/ddastp] ERROR
      A NaN value has been found after calling the
      user-supplied RESID routine.
      This occured in delta(i) for i = 49

      Named equation is: Energy conservation
      Equation number is: 6
```

which means clearly that the “Energy conservation” equation has a problem, especially at node number 6.

Another possible output is the following, when the provided jacobian is not exact, with respect to the definition of the equations:

▷ Output of ex. #4-85:

```
(MUESLI DaeSolve:) Warning: full check of the user jacobian:
      max. relat. error is:  5.489E-03
      and occurs for: named equation
      'Energy conservation' (num: 1)
                      named variable
      'Gas pressure (p_g)' (num: 1)
```

### Using specialized options for some solvers

When using the non-linear solvers (`mfFSolve` and `mfLsqNonLin`) or differential integrators using implicit schemes (`mfDaeSolve` on one hand, `mfOdeSolve` with the BDF method on the other hand), the numerical computation is often much more efficient when the jacobian is provided by the user, and not approximated internally via numerical finite differences. Of course, convergence is related to the quality of the user-supplied jacobian routine, so special attention must be paid when writing the jacobian routine, especially when it is done by hand.

Muesli offers tools for checking the correctness of the jacobian. This can be achieved by using the `check_jac` option (in the `mf.NL.Options` and `mf.DE.Options` derived types).

► Example #4-86: Vector function for which we want to find two parameters  $\{p(1), p(2)\}$  such that the sum of the square of all components is minimum.

```
subroutine exp_min( m, n, p, fvec, iflag )
  integer,          intent(in) :: m, n
  real(kind=MF_DOUBLE), intent(in) :: p(n)
  real(kind=MF_DOUBLE)          :: fvec(m)
  integer           :: iflag
  !---
  integer :: i
  do i = 1, m
    fvec(i) = 2.0d0 + 2.0d0*i - exp(i*p(1)) - exp(i*p(2))
  end do
end subroutine exp_min
```

► The jacobian of the previous vector-function is defined here. Note the explicit modification made for one component: we will try to find automatically this error.

```

subroutine exp_min_jac( m, n, p, fjac )
  integer,          intent(in) :: m, n
  real(kind=MF_DOUBLE), intent(in) :: p(n)
  real(kind=MF_DOUBLE)          :: fjac(m,n)
  !---
  integer :: i
  do i = 1, m
    fjac(i,1) = - i*exp(i*p(1))
    fjac(i,2) = - i*exp(i*p(2))
  end do
  !=====
  fjac(5,1) = fjac(5,1)*1.01d0 ! introduce an explicit error
  !=====
end subroutine exp_min_jac

```

► The minimization is done via the `mfLsqNonLin` routine, asking the algorithm to do a quick check of the jacobian at each iteration.

```

m = 10
n = 2
p = [ 0.3d0, 0.4d0 ]
call msRelease( options_NL )
options_NL%tol = 1.0d-3
options_NL%check_jac = 1
options_NL%print_check_jac = 1
z = mfLsqNonLin( m, exp_min, p, n, options_NL, jac=exp_min_jac )

```

▷ Output of ex. #4-86:

```

...

(MUESLI LsqNonLin:) Warning: iteration: 009

quality =

  1.0000
  0.9869
  0.9883
  0.9599
  0.2939
  0.9379
  0.9301
  0.9237
  0.9173
  0.9108

(MUESLI LsqNonLin:) Warning: user jacobian appears as
                           probably *** incorrect ***!

(MUESLI LsqNonLin:) Warning: quick check of the user jacobian:
                           min. of quality vector is: 0.294
                           and occurs for equation: 5

```

In the previous output, we see that the `LsqNonLin` algorithm detect a strong discrepancy in the fifth equation. We have to do a full check to find the exact location of this error in the jacobian, as shown below.

► Now, we ask `LsqNonLin` routine to do a full check of the jacobian at each iteration (Ex. #4-87).

```
m = 10
n = 2
p = [ 0.3d0, 0.4d0 ]
call msRelease( options_NL )
options_NL%tol = 1.0d-3
options_NL%check_jac = 2
options_NL%print_check_jac = 1
z = mflsqNonLin( m, exp_min, p, n, options_NL, jac=exp_min_jac )
```

► Output of ex. #4-87: With the full check, the exact location of the error is found.

```
(MUESLI LsqNonLin:) ERROR: iteration: 001

errors in the jacobian =

    0.0000    0.0000
    0.0000    0.0000
    0.0000    0.0000
    0.0000    0.0000
    0.2241    0.0000
    0.0000    0.0000
    0.0000    0.0000
    0.0000    0.0000
    0.0000    0.0000
    0.0000    0.0000

(MUESLI LsqNonLin:) ERROR: user jacobian appears as pretty *** wrong ***!
                      (max. of relative error is greater than 10 %)

(MUESLI LsqNonLin:) ERROR: full check of the user jacobian:
                      max. relat. error is:  2.241E-01
                      and occurs at (row,col): 5  1
```

Be aware that it is not unusual to get strange results for these checks, especially if the system of equations is badly scaled or if floating-point rounding errors occur. The following example deals with such a case.

► Example #4-88: Define a DAE system of three equations.

```

! real(kind=MF_DOUBLE) :: r3_a = -0.5d0, r3_b = 5.0d0, r3_c = -5.0d0

subroutine resid_3( 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
  delta(1) = r3_a*y(1) + r3_b*y(2)*y(3)
  delta(2) = -delta(1) + r3_c*y(2)**2 - yprime(2)
  delta(1) = delta(1) - yprime(1)
  delta(3) = y(1) + y(2) + y(3) - 1.0d0
end subroutine resid_3

subroutine jac_resid_3( t, y, yprime, jacobian, cj, nrow )
  real(kind=MF_DOUBLE), intent(in) :: t, y(*), yprime(*), cj
  integer, intent(in) :: nrow
  real(kind=MF_DOUBLE), intent(out) :: jacobian(nrow,*)
  jacobian(1,1) = r3_a - cj
  jacobian(2,1) = -r3_a
  jacobian(3,1) = 1.0d0
  jacobian(1,2) = r3_b*y(3)
  jacobian(2,2) = -r3_b*y(3) + 2*r3_c*y(2) - cj
  jacobian(3,2) = 1.0d0
  jacobian(1,3) = r3_b*y(2)
  jacobian(2,3) = -r3_b*y(2)
  jacobian(3,3) = 1.0d0
end subroutine jac_resid_3

```

► The DAE solver is called as follows:

```

t_span = .t. mf( [ 0.0d0, 0.5d0, 1.0d0, 2.0d0, 5.0d0 ] )
y_0 = [ 1.0d0, 0.0d0, 0.0d0 ]
yp_0 = MF_EMPTY
options_DE%IC_known = .false.
options_DE%check_jac = 1
options_DE%print_check_jac = 1
y = mfDaeSolve( resid_3, t_span, y_0, yp_0, options_DE, jac=jac_resid_3 )

```

▷ Output of ex. #4-88: During integration, the output doesn't present any warning (but quality of the jacobian is not perfect!):

```

...

quality =

  0.9470
  0.9347
  1.0000

quality =

  0.9574
  0.9354
  1.0000

```

▷ Because the jacobian is mathematically correct, the full check (using `check_jac = 2` in the `options_DE` structure) doesn't detect any error:

```
...

errors in the jacobian =

1.0E-007 *

0.0000    0.0012    0.0031
0.0000    0.2446    0.0031
0.0000    0.0000    0.0000

errors in the jacobian =

1.0E-007 *

0.0000    0.0020    0.0016
0.0000    0.2327    0.0016
0.0000    0.0000    0.0000
```

However, if we change the value of the three parameters `[r3_a, r3_b, r3_c]` to their original value (the DAE equations actually describe a stiff chemical system), *i.e.* `[r3_a=-0.04d0, r3_b=1.0d4, r3_c=-3.0d7]`, then both the quick check and the full check claim that the jacobian is inexact:

▷ Output of ex. #4-88: The new DAE system is very stiff (parameters `[r3_a, r3_b, r3_c]` has been changed back to their original values). Case of a quick check which give a wrong result, due to rounding-off errors.

```
(MUESLI DaeSolve:) ERROR: time: 0.000000E+00

quality =

0.3101
0.0000
1.0000

(MUESLI DaeSolve:) ERROR: user jacobian appears as
                        certainly *** wrong ***!

(MUESLI DaeSolve:) ERROR: quick check of the user jacobian:
                        min. of quality vector is: 0.000
                        and occurs for equation: 2
```

▷ Output of ex. #4-88: The new DAE system is very stiff (parameters `[r3_a, r3_b, r3_c]` has been changed back to their original values). Case of a full check which give a wrong result, due to rounding-off errors.



```

(MUESLI DaeSolve:) ERROR: time:  0.000000E+00

errors in the jacobian =

    0.0000    0.0000    0.0000
    0.0000    0.4470    0.0000
    0.0000    0.0000    0.0000

(MUESLI DaeSolve:) ERROR: user jacobian appears as pretty *** wrong ***!
                        (max. of relative error is greater than 10 %)

(MUESLI DaeSolve:) ERROR: full check of the user jacobian:
                        max. relat. error is:  4.470E-01
                        and occurs at (row,col): 2  2

```

For the DAE solver (`mf/msDaeSolve`), the jacobian must not be singular. The `jac_rcond_min` option of the `mf.DE.Options` derived type allows the user to choose the threshold of the singularity. Moreover, using the `jac_investig` option in the same structure allows the user to know the nullspace of the jacobian when it is singular; this is useful to discover, for example, that two rows of the jacobian are exactly the same (this can come from an error while coding the jacobian or from a badly formed set of DAE equations). See the following example.

- Example #4-89: This DAE system contains two identical equations

```

subroutine resid_singular( 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
  delta(1) = y(1) + y(3)
  delta(2) = y(2) - yprime(3)
  delta(3) = y(3) - yprime(4)
  ! this equation 4 is exactly the same as equation 2.
  delta(4) = delta(2)
end subroutine resid_singular

```

- We solve it using the following calls with the `jac_investig` option

```

t_span = .t. mf( [ 0.0d0, 1.0d0, 2.0d0 ] )
y_0 = [ 0.0d0, 0.0d0, 0.0d0, 0.0d0 ]
options_DE%IC_known = .false. ! Initial Conditions are not known
options_DE%jac_investig = .true.
options_DE%save_sing_jac = .true.
y = mfDaeSolve( resid_singular, t_span, y_0, yp_0, options_DE )

```

- ▷ Output of ex. #4-89:

```
(MUESLI mfDaeSolve:) Error: the DAE solver encountered the following problem:
                        'Jacobian is singular during the integration phase'

(MUESLI:) processing the Jacobian matrix...

Jacobian matrix has been saved in 'jacobian.dat'.

rational basis of the null space =

    0
   -1
    0
    1

dim(nullspace) =

    1

Above can be found an approximation of the null space. It may help you
detecting wrong equations in your system. Each column is a null vector of the
linear application and it shows, via its components, the dependencies between
the current equations coded in the 'resid()' routine.
(note 1: the transpose of the jacobian matrix has been processed)
(note 2: if the nullspace appears as empty, try another smaller value for
        'rcond_jac_min')

A rational basis of the null space has been saved in 'nullspace.dat'.
```

▷ We can easily verify that the entries of the approximated jacobian matrix, saved in the file `jacobian.dat`, are all correct with respect to the differential equations in the `resid_singular` routine:

```
1.00E+000  0.00E+000  1.00E+000  0.00E+000
0.00E+000  1.00E+000 -4.00E+003  0.00E+000
0.00E+000  0.00E+000  1.00E+000 -4.00E+003
0.00E+000  1.00E+000 -4.00E+003  0.00E+000
```

Coming back to the output of `mfDaeSolve` which contains the main components of the null space, it clearly shows that equations 2 and 4 are the same.

Note that this investigation is done only in the `DEBUG` mode and that it may be very expensive, according to the size of the jacobian matrix.

#### 4.3.8 Exchanging Data with MATLAB

MUESLI provides a way to build MATLAB mex functions. These executables (seen as new commands under MATLAB), `mbfread` and `mbfwrite`, are used to read and write MBF files, *i. e.* binary files created by the `msSave` routine (cf. section 4.2.9), in gzipped/not gzipped state, in little/big endian format, of real/complex type and for sparse structure as well as dense structure.

The use of these MATLAB commands are not described in the *MUESLI Reference Manual*. However, you can type `'help mbfread'` and `'help mbfwrite'` under MATLAB to get a short usage description.

Matrix properties (nor physical properties) are not read in MATLAB. On the other hand, the `mbfwrite` mex function writes a valid MBF file, always setting `UNKNOWN` to the two matrix properties and always returning a dimensionless `mfArray`.

Please see the *MUESLI Installation Guide* document for building these mex functions.

## 4.4 Equivalence between MUESLI Routines, MATLAB Commands and f90 Intrinsic Functions

The usual mathematical functions ( $\sin$ ,  $\cos$ ,  $\dots$ ,  $\log$ ,  $\exp$ ,  $\sqrt{\phantom{x}}$ ,  $\dots$ ) are not included in the following tables. Note that in MUESLI the following functions:  $\arccos$ ,  $\arcsin$ ,  $\arctan$ ,  $\cosh$ ,  $\sinh$ ,  $\tan$ ,  $\tanh$  have been extended to support full range of real and complex values, including special IEEE-754 values ( $Inf$  and  $NaN$ ).

### 4.4.1 from MUESLI Routines

MUESLI	MATLAB	Fortran 90
<b>MF_EPS</b>	eps	epsilon
<b>MF_REALMAX</b>	realmax	huge
<b>MF_REALMIN</b>	realmin	tiny
<b>operator(.t.)</b>	'	transpose
<b>mfAll</b>	all	all
<b>mfAny</b>	any	any
<b>mfCeil</b>	ceil	ceiling
<b>mfComplex</b>	complex	complex
<b>mfConj</b>	conj	conjg
<b>mfCount</b>	nnz (1) or sum	count
<b>mfCshift</b>	—	cshift
<b>mfEoshift</b>	—	eoshift
<b>mfFix</b>	fix	—
<b>mfFlipLR</b>	fliplr	—
<b>mfFlipUD</b>	flipud	—
<b>mfFloor</b>	floor	floor
<b>mfImag</b>	imag	aimag
<b>mfMul</b>	*	matmul
<b>mfMax</b>	max	maxval
<b>mfMerge</b>	—	merge
<b>mfMin</b>	min	minval
<b>mfPack</b>	—	pack
<b>mfProd</b>	prod	product
<b>mfRand</b>	rand	random_number
<b>mfRot90</b>	rot90	—
<b>mfRound</b>	round	anint
<b>mfReal</b>	real	real
<b>mfReshape</b>	reshape	reshape
<b>mfShape</b>	size	shape
<b>mfSize</b>	prod and size (2)	size
<b>mfRepMat</b>	repmat	spread
<b>mfSum</b>	sum	sum
<b>mfUnpack</b>	—	unpack

notes:

- (1) **nnz** operates globally on the array whereas **count** and **mfCount** may operate by columns.
- (2) under MATLAB, the total size of an array **A** (*i. e.* the number of its elements) writes **prod(size(A))**.

## 4.4.2 from MATLAB Commands

MATLAB	MUESLI	Fortran 90
*	<a href="#">mfMul</a>	matmul
'	<a href="#">operator(.t.)</a>	transpose
all	<a href="#">mfAll</a>	all
any	<a href="#">mfAny</a>	any
ceil	<a href="#">mfCeil</a>	ceiling
complex	<a href="#">mfComplex</a>	complex
conj	<a href="#">mfConj</a>	conjg
eps	<a href="#">MF_EPS</a>	epsilon
fix	<a href="#">mfFix</a>	—
fliplr	<a href="#">mfFlipLR</a>	—
flipud	<a href="#">mfFlipUD</a>	—
floor	<a href="#">mfFloor</a>	floor
imag	<a href="#">mfImag</a>	aimag
max	<a href="#">mfMax</a>	maxval
min	<a href="#">mfMin</a>	minval
nnz	<a href="#">mfNnz</a>	count and size
prod	<a href="#">mfProd</a>	product
rand	<a href="#">mfRand</a>	random_number
real	<a href="#">mfReal</a>	real
realmax	<a href="#">MF_REALMAX</a>	huge
realmin	<a href="#">MF_REALMIN</a>	tiny
repmat	<a href="#">mfRepMat</a>	spread
reshape	<a href="#">mfReshape</a>	reshape
rot90	<a href="#">mfRot90</a>	—
round	<a href="#">mfRound</a>	aint
size	<a href="#">mfShape</a>	shape
sum	<a href="#">mfSum</a>	sum

## 4.4.3 from f90 Intrinsic Functions

Fortran 90	MUESLI	MATLAB
aimag	<code>mfImag</code>	imag
anint	<code>mfRound</code>	round
all	<code>mfAll</code>	all
any	<code>mfAny</code>	any
ceiling	<code>mfCeil</code>	ceil
complex	<code>mfComplex</code>	complex
conjg	<code>mfConj</code>	conj
count	<code>mfCount</code>	nnz (1)
cshift	<code>mfCshift</code>	—
dot_product	<code>mfMul</code> (3)	* and ' (3)
eoshift	<code>mfEoshift</code>	—
epsilon	<code>MF_EPS</code>	eps
floor	<code>mfFloor</code>	floor
huge	<code>MF_REALMAX</code>	realmax
matmul	<code>mfMul</code>	*
maxloc	<code>mfMax</code> (4)	max (4)
maxval	<code>mfMax</code>	max
merge	<code>mfMerge</code>	—
minloc	<code>mfMin</code> (4)	min (4)
minval	<code>mfMin</code>	min
pack	<code>mfPack</code>	—
product	<code>mfProd</code>	prod
random_number	<code>mfRand</code>	rand
random_seed	<code>mfRand</code> (4)	rand (4)
real	<code>mfReal</code>	real
reshape	<code>mfReshape</code>	reshape
shape	<code>mfShape</code>	size
size	<code>mfSize</code>	prod and size (2)
spread	<code>mfRepMat</code>	repmat
sum	<code>mfSum</code>	sum
tiny	<code>MF_REALMIN</code>	realmin
transpose	<code>operator(.t.)</code>	'
unpack	<code>mfUnpack</code>	—

notes:

- (1) `nnz` operates globally on the array whereas `count` and `mfCount` may operate by columns.
- (2) under MATLAB, the total size of an array `A` (*i.e.* the number of its elements) writes `prod(size(A))`.
- (3) if `a` and `b` are column vectors, then `dot_product(a,b)` writes `mfMul(.t.a,b)` with MUESLI and `a'*b` under MATLAB.
- (4) an additional argument must be used.

## 5 FGL (Graphical Library)

FGL is a library which depends on FML. Most of FGL routines work with `mfArrays`. Usually, the two forms of the same routine (*i. e.*, the subroutine, beginning with 'ms', and the function, beginning with 'mf') do the same job, but the latter returns an identifier (most of time one graphic handle, sometimes a vector of many graphic handles). Each registered graphic object has a handle, which is used to access to this graphic object (*e. g.*, to remove it or to change some of its attributes).

It is recommended to use the routine `msExitFgl` at the end of the graphic part of your program; indeed, not only this routine waits for a user answer – avoiding the close of all opened Muesli figures – but it also cleans the graphic internal memory.

### 5.1 Creating and Numbering Figures

A simple call to `msFigure` opens a window in which all the following graphic commands will be displayed. Two optional arguments let you to set (i) the number of this figure (in the case where you have multiple figures) and (ii) the position and the size of the window.

Even if it is not specified at the creation, each figure has a number which can be later retrieved by the function `mfFigure`. This number is required to make active or to close a window.

Maximum number of figures opened at the same time is 48.

► Example #5-0:

```
call msFigure(5) ! opens a window for the FGL figure #5
! ... next graphic commands are output in FGL figure #5
geometry = [ 50, 50, 900, 650 ] ! prepares for next figure
call msFigure(3,geometry) ! opens a window for the FGL figure #3, with
                           ! position = (50,50) and size = 900 by 650 pixels
! ... next graphic commands are output in FGL figure #3
call msFigure(5)
! ... next graphic commands are output in FGL figure #5
call msClose(5)

! hereafter, only figure #3 is opened
call msExitFgl() ! figure #3 is visible until the answer 'y'
                  ! is typed by the user
```

By default, the background color of graphic windows is white (it is more consistent with the EPS and PDF files printed from the figures). However, the user can change it to the black color, by using the `msSetBackgroundColor` routine.

### 5.2 How axes may be defined in your figure?

The presence of axes in your figure is very important because it helps to understand how the information can be read inside it. This is important of course for an ordinary plot showing the relationship between *x*- and *y*-values, but even for the display of an image to understand why the image is orientated as it is. As for most of (if not all) plotting software, the *x*-axis is horizontal, whereas the *y* one is vertical. But there are further things after this common description.

The only entry point is the `msAxis` routine. The use of the subroutine form allows to define some properties via different keywords.

The corresponding page in the *Muesli Reference Manual* contains all the details to work with the `msAxis` routine, but we prefer to give here some complementary aspect, like its main philosophy.

Contrary to Matlab (or other software), all properties may be defined before plotting something. By default, axes properties are set to:

- "on": axes are visible. The contrary is "off".

- "auto": axes' ranges are chosen to give smart extremal values. The contraries are "tight" and "manual".
- "equal": axes have the same scaling. The contrary is "unequal".
- "xy": the  $y$ -axis is upward directed. The contrary is "ij".
- "lin": the corresponding axis is linearly graduated. The other possibility is "log" (*i. e.*, logarithmic).

The previously default properties concern most of plotting commands, or most of situations. Some exceptions follow:

- for the `msSpy` routine (visualization of the pattern of a sparse matrix), the "ij" property is set.
- for the `msPcolor` and `msImage` routines (resp. rendering colors from a matrix), the "tight" and "ij" properties are set.
- for any plot using the "linlog" or "loglin" axis scaling, the "unequal" property is set.
- for any plot using the "ij" property set, then the "linlin" axis scaling is chosen.

If any property is not suitable to the user then he/she can change this property, but after the plotting command. Likewise, the user must change these properties before executing different plotting commands, if they are not appropriate.

### 5.3 Color Management

For displaying distinct graphic objects of a figure (some lines for example), FGL needs to use different colors. These colors are grouped in colortables, each on them called a color scheme.

FGL uses one of four different color schemes:

1. the old PGPLOT colors: these are 6 saturated colors not always appropriate concerning the contrast of a colored object on the white background (*e. g.*, the yellow color is terrible!). See figure 1.

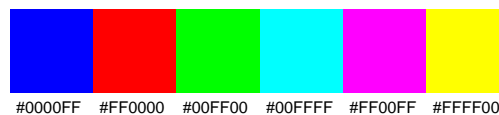


Figure 1: Display of the color scheme number 1 (old PGPLOT colors)

2. the old Matlab colors: 7 colors, similar to the above ones, but darker; they were in use in Matlab up to the R2014a release. See figure 2.

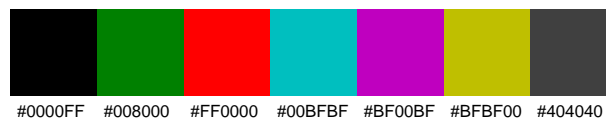


Figure 2: Display of the color scheme number 2 (old Matlab colors)

3. the new Matlab colors: 7 colors, totally different from the previous one, and having a good contrast between them; they are used from the release R2014b. See figure 3.



Figure 3: Display of the color scheme number 3 (new Matlab colors)

4. the Breeze colortable: 12 colors from LibreOffice. See figure 4.



Figure 4: Display of the color scheme number 4 (Breeze colors – LibreOffice)

The current color scheme may be changed (resp. retrieved) by use of the routine `msSetColorScheme` (resp. `mfGetColorScheme`). Equivalently, the environment variable `MFPLLOT_COLOR_SCHEME` can be used. Default color scheme is number 3, *i. e.* the new Matlab color scheme).

The colors are selected automatically by the library, but the user may select explicitly one of them, by using the special `\Cnn` escaped sequence in the `col` or `linespec` argument of many plotting routines (see *e.g.* `msPlot` in the *Muesli Reference Manual*). If the number specified by the code `nn` exceeds the size of the current scheme, then a cycling among colors is applied.

## 5.4 High-level Plotting Routines

The main routine which makes simple plots, constituted by linear segments, is `msPlot`: it plots a vector `mfArray` against another vector `mfArray` and have a similar interface than that of the ‘`plot`’ Matlab command. Logarithmic and semi-logarithmic plots are obtained by specifying an appropriate option in `msAxis`.

- Example #5-1: (see output plot in figure 5)

```
call msFigure(1)

x = mfLinSpace( 0.0d0, 1.0d0, 8 )
y = [ 0., 1., 0., 1., 0., 1., 0., 1. ]

call msPlot( x, y )
call msXLabel( "X-axis" ); call msYLabel( "Y-axis" ); call msTitle( "Title" )
```

While the previous plot routines draw linear segment between data points, two additional routines are available to draw curved lines. The first one, `msPlotCubicBezier` takes the data vector as control points for a piecewise cubic Bézier curve.

(By the way `msHold` is used to plot a new graphic object without erasing the current figure.)

- Example #5-2: (see output plot in figure 6)



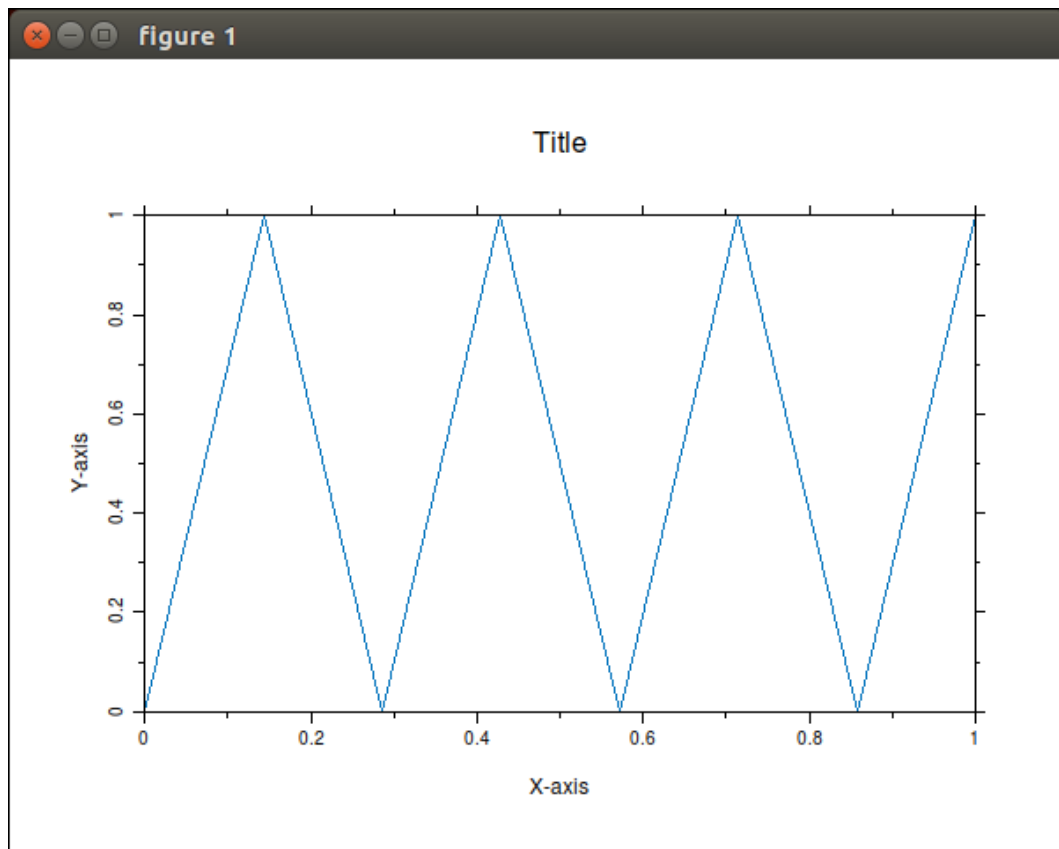


Figure 5: output of ex. #5-1: msPlot use

```

! integer :: n_seg, i, ioff

call msFigure(1)

call msAxis( [ -0.1d0, 1.1d0, -0.1d0, 1.1d0 ] )
call msAxis( "equal" )

x = [ 0., 0.3333, 0.3333, 0.6667, 0.6667, 1., 1. ]
y = [ 0., 0., 1., 1., 0., 0., 1. ]

call msPlotCubicBezier( x, y ) ! default color is blue

! control points drawing (in red)
call msHold( "on" )
call msPlot( x, y, "r+" )
n_seg = (Size(x)-1)/3
do i = 1, n_seg
  ioff = 3*(i-1)
  call msPlot( mfGet(x,ioff+1 .to. ioff+2),      &
               mfGet(y,ioff+1 .to. ioff+2), "r--" )
  call msPlot( mfGet(x,ioff+3 .to. ioff+4),      &
               mfGet(y,ioff+3 .to. ioff+4), "r--" )
end do
! set X-, Y-label and title

```

The second one, `msPlotCubicSpline` interpolates through the data points, and draw a piecewise

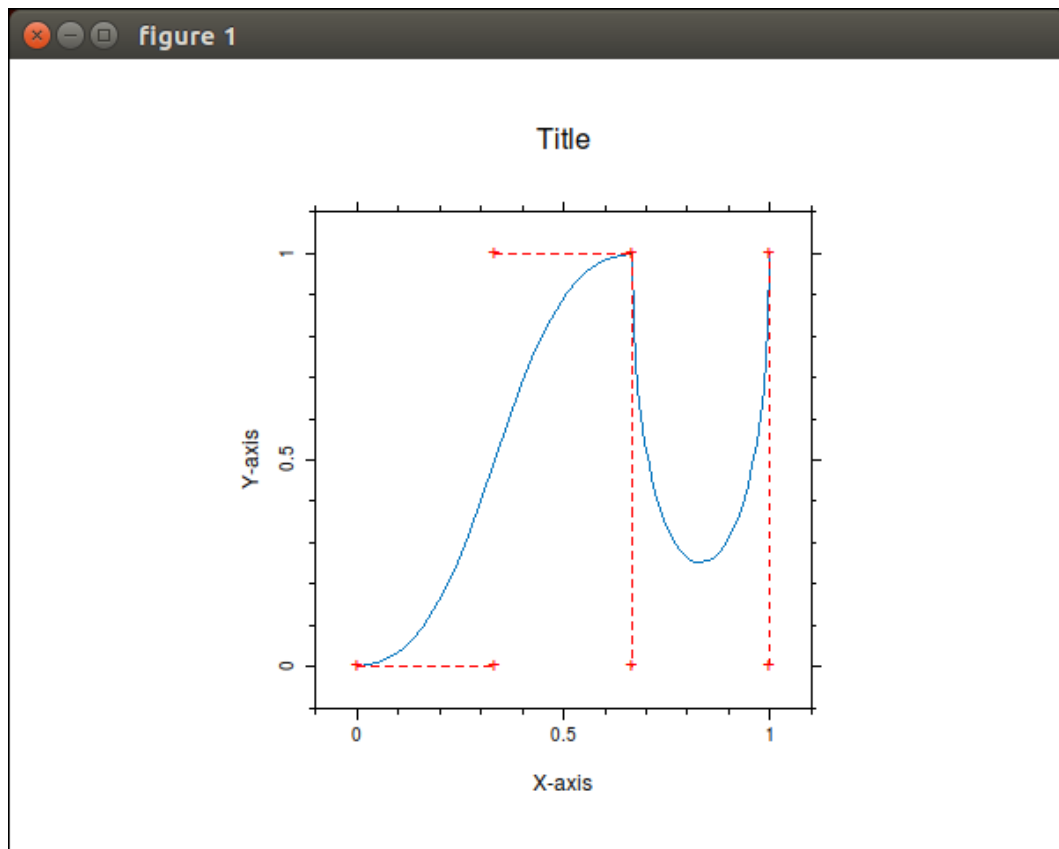


Figure 6: output of ex. #5-2: msPlotCubicBezier use

parametrized cubic spline curve. However, some additional data must have been previously computed by using the routine `mfSpline`.

► Example #5-3: (see output plot in figure 7)

```
call msFigure(1)

abs_curv = [ 0., 1., 2., 3., 4., 5., 6. ]
x = [ 0., 0.3333, 0.3333, 0.6667, 0.6667, 1., 1. ]
y = [ 0., 0., 1., 1., 0., 0., 1. ]

wx = mfSpline( abs_curv, x )
wy = mfSpline( abs_curv, y )

call msPlotCubicSpline( abs_curv, x, y, wx, wy )
call msHold( "on" )
call msPlot( x, y, "ro" ) ! red 'o' symbols show the data points
! set X-, Y-label and title
```

Two routines work with 2D data. `msContour` draws contours of data contained in an `mfArray`, whereas `msPColor` draws a pseudo-color plot of the same data. Be aware that a colormap must be defined before (via the routine `msColorMap`).

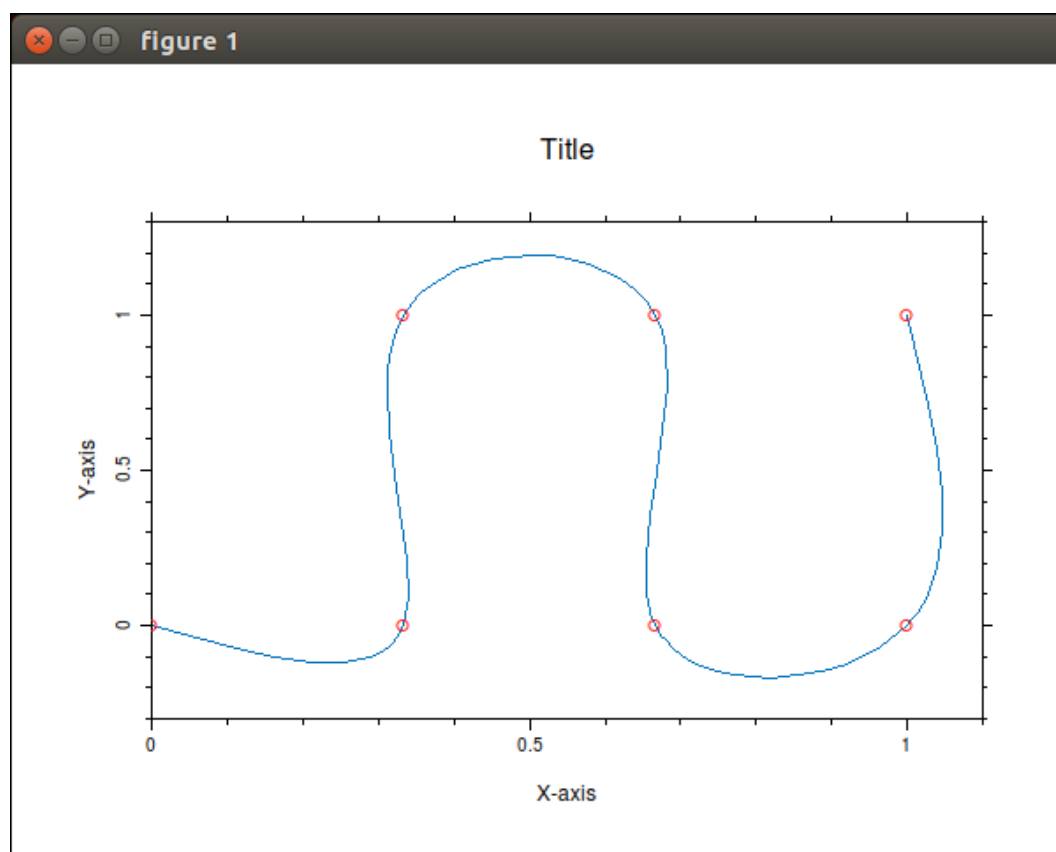


Figure 7: output of ex. #5-3: msPlotCubicSpline use

The seven colormaps available in Muesli are shown below in figures 8-14.

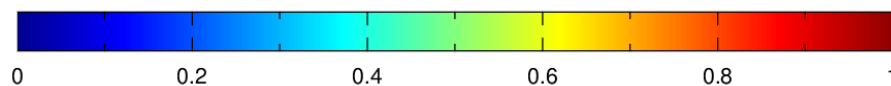


Figure 8: The “Rainbow” colormap (default)



Figure 9: The “Parula” colormap

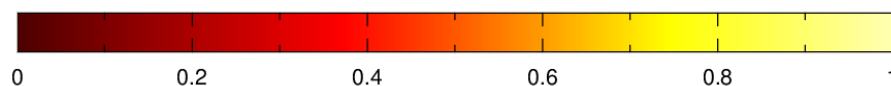


Figure 10: The “Hot” colormap



Figure 11: The “Bluered” colormap

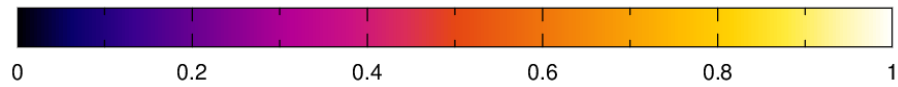


Figure 12: The "Fusion" colormap

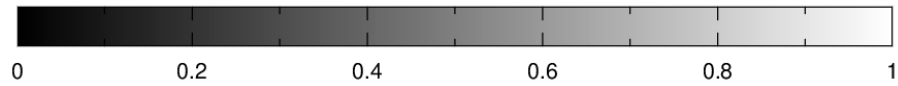


Figure 13: The "Grey" colormap



Figure 14: The "Flag" colormap

► Example #5-4: (see output plot in figure 15)

```
call msFigure(1)

call msColormap( "rainbow" )
call msAxis( "equal" )

call msMeshGrid( mfOut(x,y), mfLinspace(0.0d0, 5.0d0, 20),      &
                 .t. mfLinspace(0.0d0, 10.0d0, 40) )
r = mfSqrt(x**2 + y **2)
z = mfSin( r ) / r
call msSet( 1.0d0, z, 1, 1 ) ! avoids NaN value : sin(0)/0 = 1

call msCAxis( [ -0.3d0, +1.0d0 ] ) ! defines the color axis

call msContour( mfGet(x,1,MF_ALL), mfGet(y,MF_ALL,1), z,      &
               labels=.false. )
call msColorbar( "on" )
! set X-, Y-label and title
```

► Example #5-5: (see output plot in figure 16)

```
call msFigure(1)

call msColormap( "rainbow" )
call msAxis( "equal" )

call msMeshGrid( mfOut(x,y), mfLinspace(0.0d0, 5.0d0, 20),      &
                 .t. mfLinspace(0.0d0, 10.0d0, 40) )
r = mfSqrt(x**2 + y **2)
z = mfSin( r ) / r
call msSet( 1.0d0, z, 1, 1 ) ! avoids NaN value : sin(0)/0 = 1

call msCAxis( [ -0.3d0, +1.0d0 ] ) ! defines the color axis

call msContourF( mfGet(x,1,MF_ALL), mfGet(y,MF_ALL,1), z,      &
               labels=.false. )
call msColorbar( "on" )
! set X-, Y-label and title
```

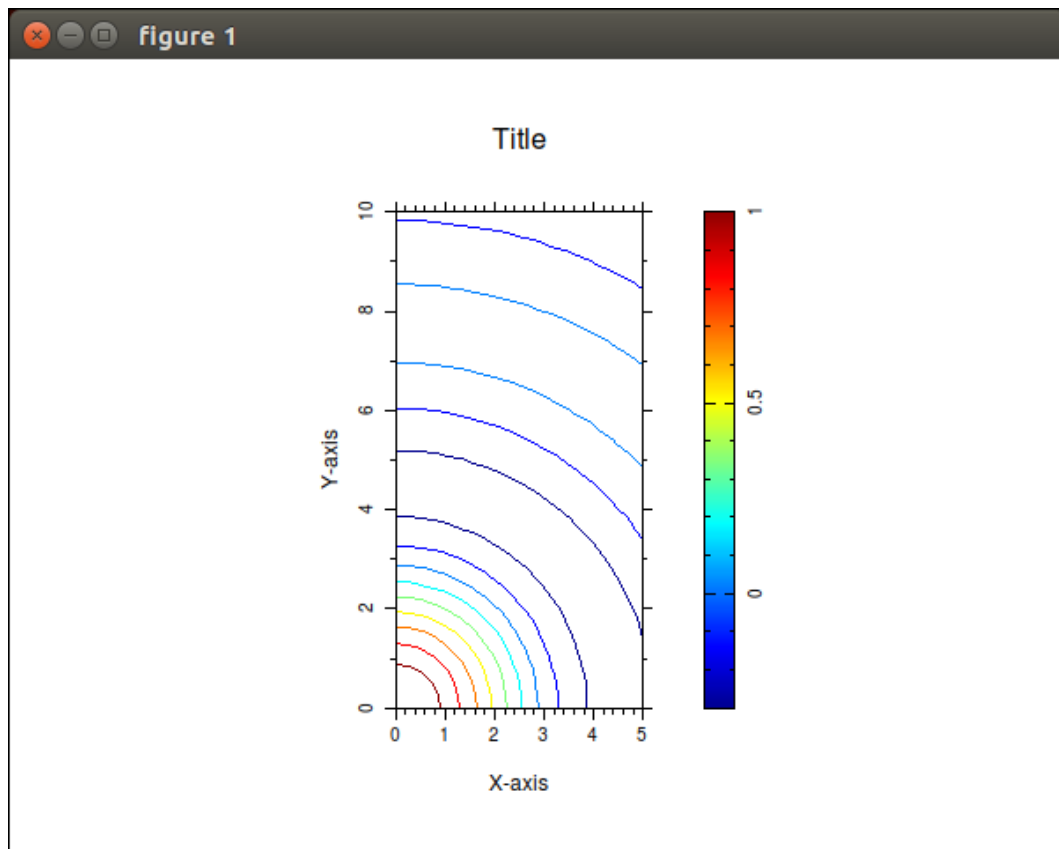


Figure 15: output of ex. #5-4: msContour use

► Example #5-6: (see output plot in figure 17)

```
call msFigure(1)

call msColormap( "rainbow" )
call msAxis( "equal" )

z = mfRandN( 10 )
call msCAxis( [ -3.0d0, +3.0d0 ] )

call msPcolor( z ) ! default: shading is "flat"

call msColorbar( "on" )
! set X-, Y-label and title
```

Note that for `mfArrays` with great size (*i.e.* giving small cells) the *flat* shading is sufficient and much more efficient.

► Example #5-7: (see output plot in figure 18)

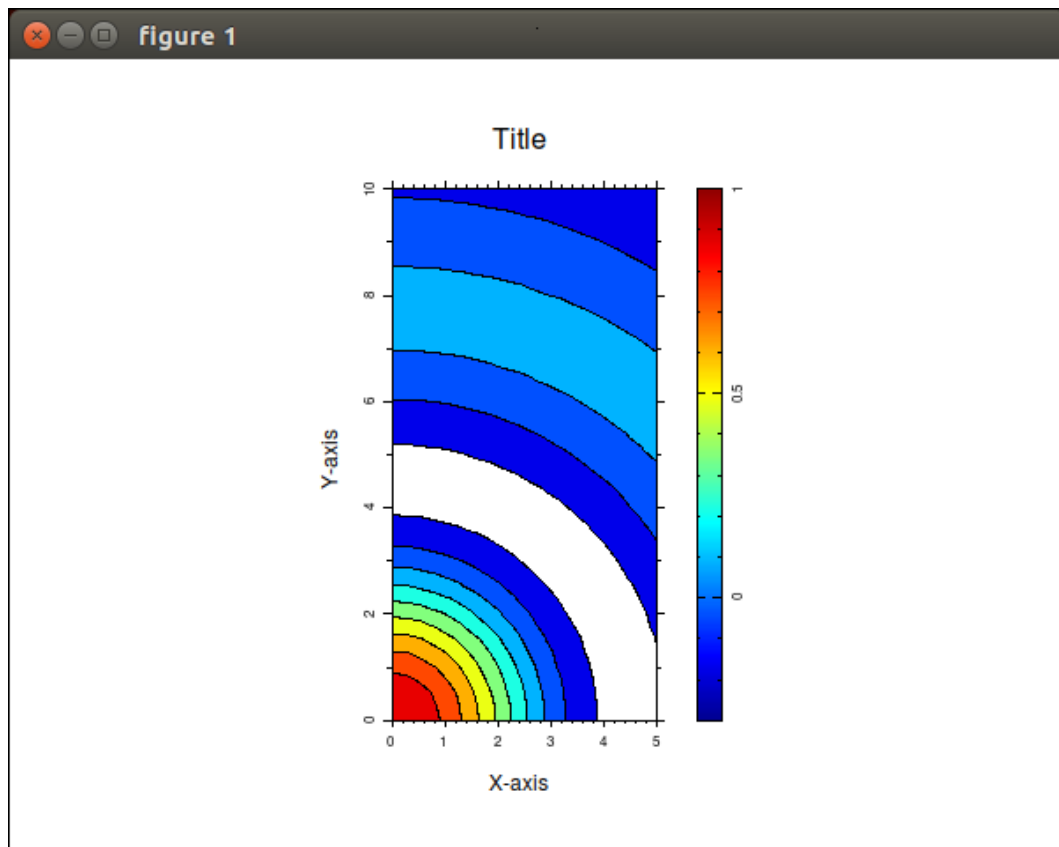
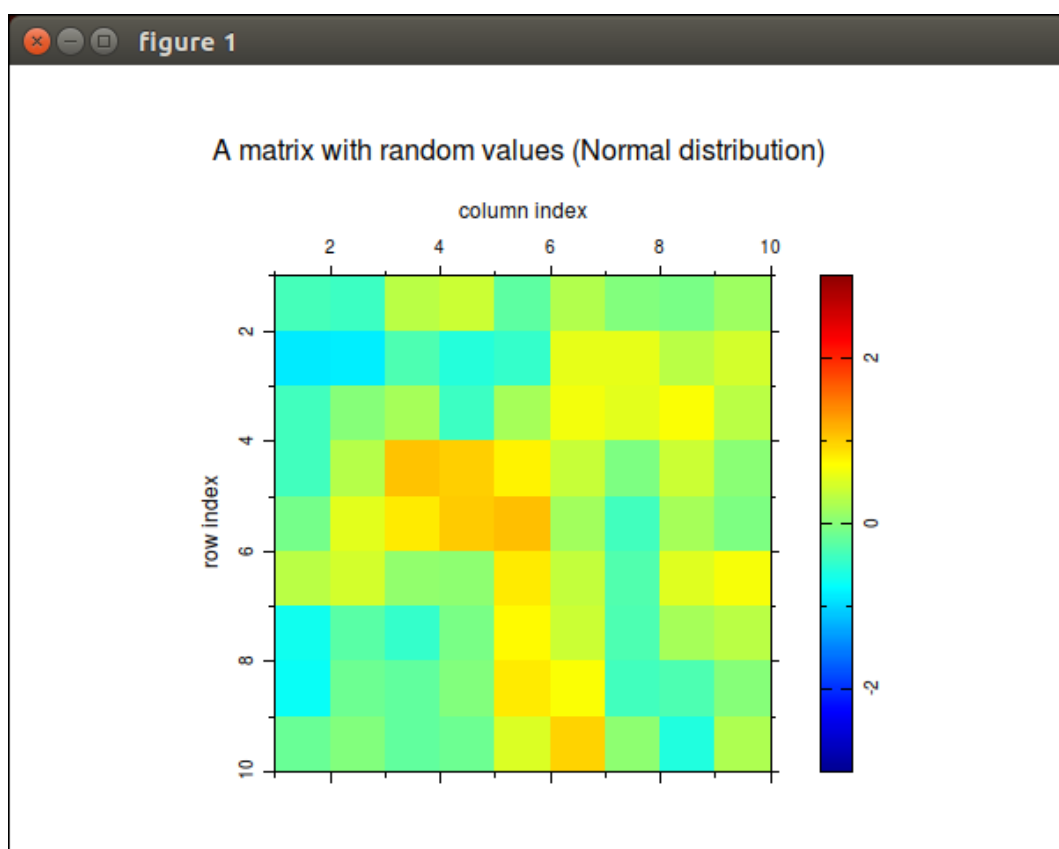


Figure 16: output of ex. #5-5: msContour use

Figure 17: output of ex. #5-6: msPColor use (default shading, *i. e.* flat)

```

call msFigure(1)

call msColormap( "rainbow" )
call msAxis( "equal" )

x = mfRandN( 10 )
call msCAxis( [ -3.0d0, +3.0d0 ] )

call msShading( "interp" )
call msPcolor( x )

call msColorbar( "on" )
! set X-, Y-label and title

```

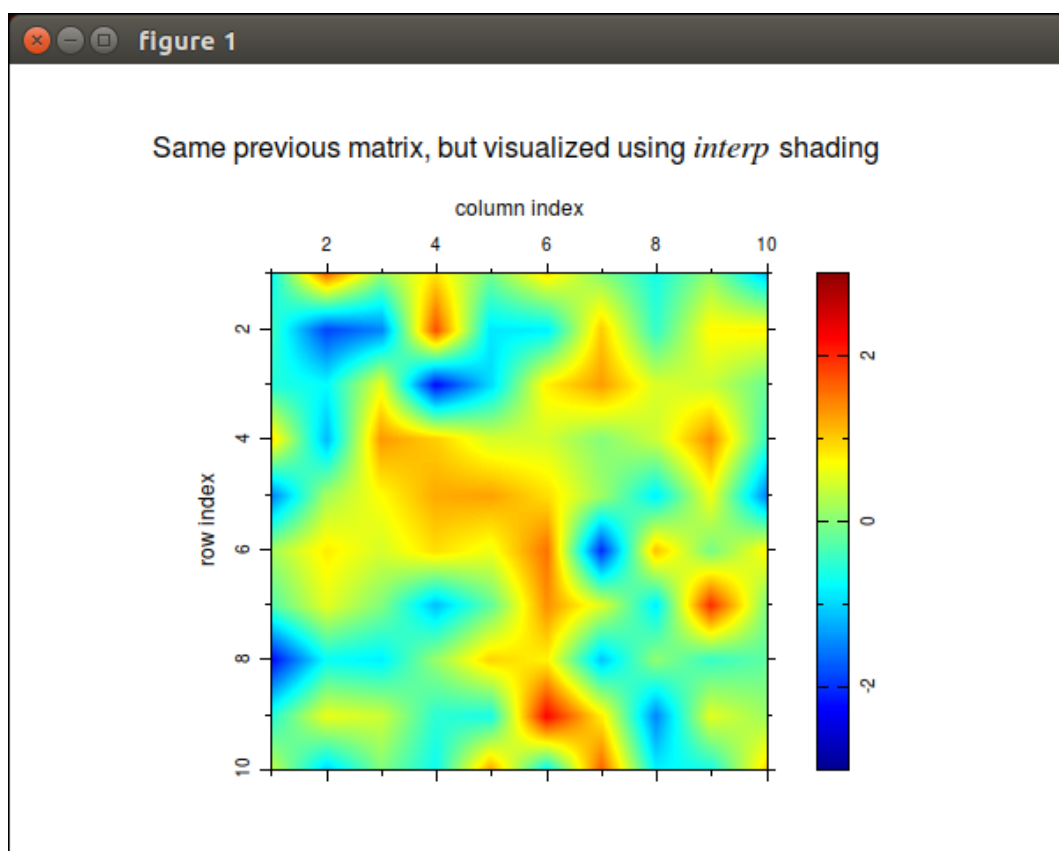


Figure 18: output of ex. #5-7: msPColor use (*interp* shading)

If you want to process 2D vector fields, you can use `msQuiver` and `msStreamline`. The first routine plots an arrow field, whereas the second one compute the streamline attached to a starting point (a streamline is a continuous line which is tangential to a given vector field).

- Example #5-8: (see output plot in figure 19)

```

call msFigure(4)

call msMeshGrid( mfOut(x,y), mfColon(0.0d0,2.0d0,0.2d0),           &
                 .t. mfColon(1.8d0,0.0d0,-0.2d0) )

z = x*mfExp(-x**2 - y**2)
call msGradient( mfOut(dx,dy), z, 0.2d0, 0.2d0 )

call msAxis( "equal" )
call msAxis( [ -0.1d0, 2.1d0, -0.1d0, 1.9d0 ] )
call msPlot( mf([0.,2.,2.,0.,0.]), mf([0.,0.,1.8,1.8,0.]), color="grey" )
call msHold( "on" )

call msColormap( "rainbow" )
call msCAxis( mfExtrema(z) )
call msContour( x, y, z )
call msColorbar( "on", position="vert" )

! reducing matrices (x,y) to vectors
x = mfGet( x, 1, MF_ALL )
y = mfGet( y, MF_ALL, 1 )
call msQuiver( x, y, dx, dy )
! set X-, Y-label and title

```

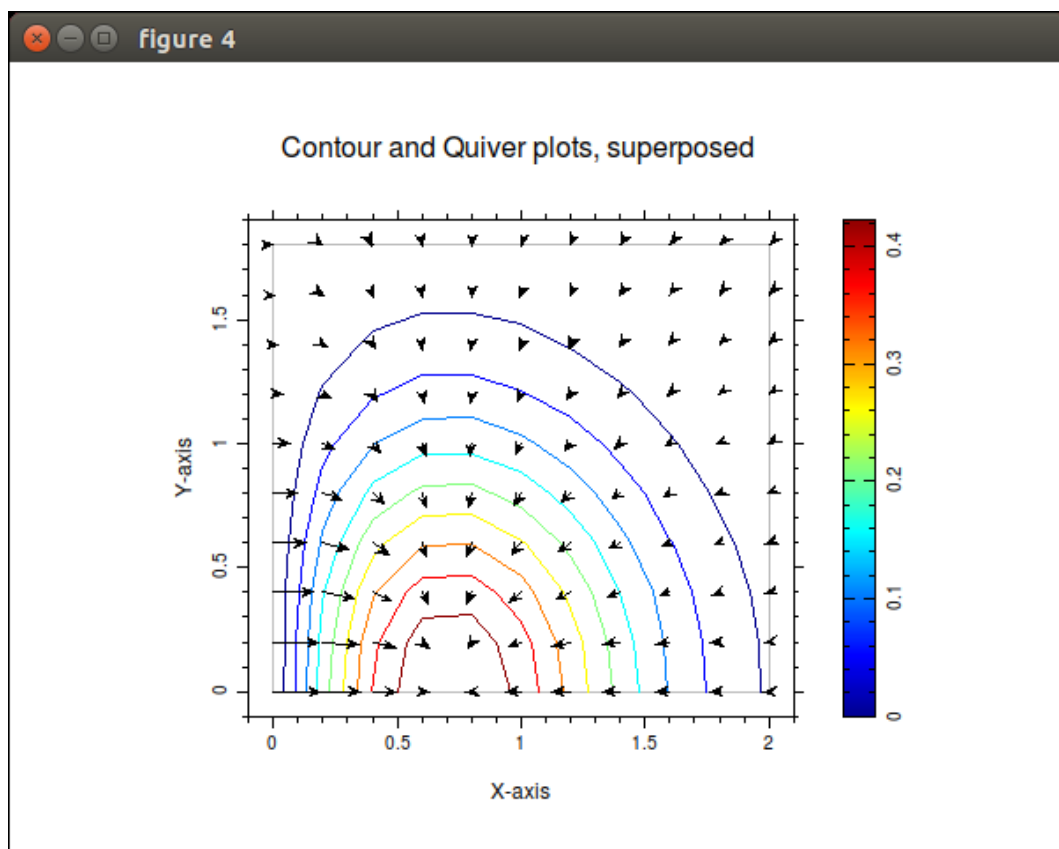


Figure 19: output of ex. #5-8: msQuiver use (superposed to the result of msContour)

► Example #5-9: (see output plot in figure 20)



```

call msFigure(5)

! Data contains a 2D vector field on a 5x5 square grid (initialization
! not shown). The N starting points are stored in the mfArray 'start'.

! drawing of the rectangular mesh
do k = 1, N
  call msPlot( x, mfGet(y,k)*mfOnes(1,N), color=[0.7d0,0.7d0,0.7d0] )
end do
do k = 1, N
  call msPlot( mfGet(x,k)*mfOnes(1,N), y, color=[0.7d0,0.7d0,0.7d0] )
end do
call msQuiver( x, y, ux, uy, arrow_head=1.0d0 )
call msPlot( mfGet(start,MF_ALL,1), mfGet(start,MF_ALL,2), "ob" )
call msStreamline( x, y, ux, uy, start, color="r", direction="forward" )
! set X-, Y-label and title

```

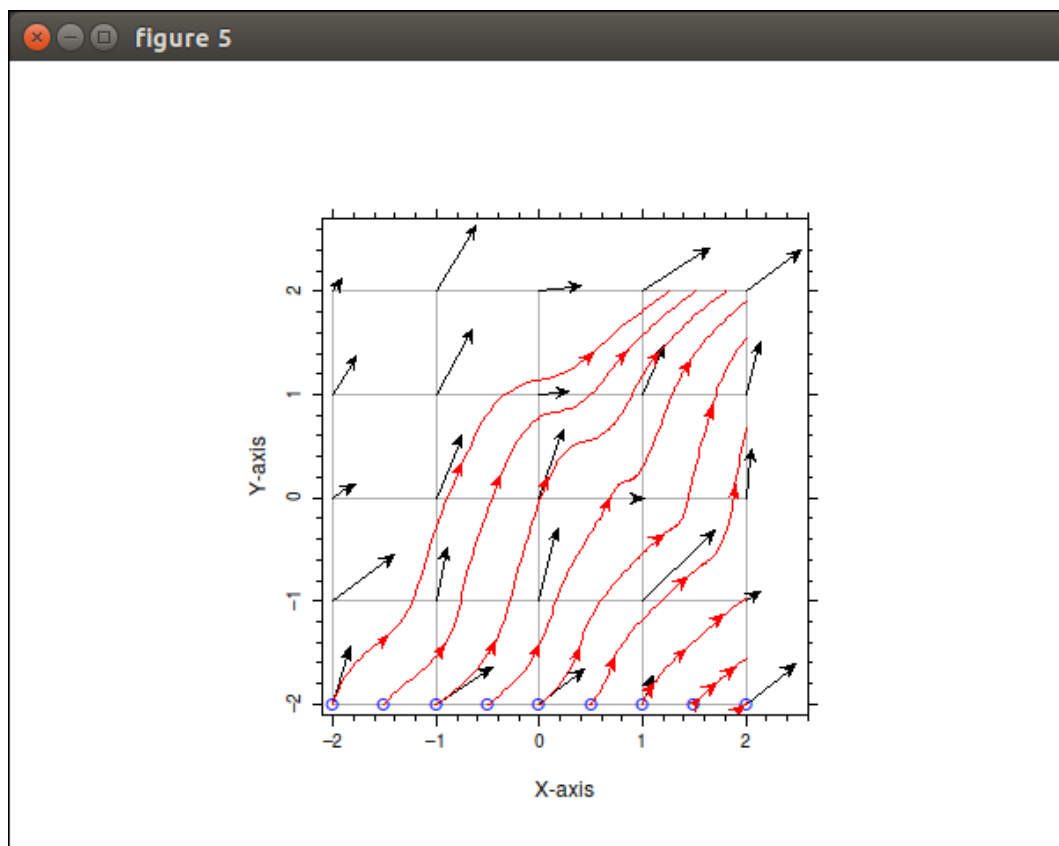


Figure 20: output of ex. #5-9: msStreamline use (superposed to the result of msQuiver)

The `msPlotPLdomain` is used to display a PL domain structure. The following example just plot the PL domain defined in example #4-53.

► Example #5-10: (see output plot in figure 21)

```
call msFigure(1,size=[600,600])
call msCharInPixels( "on" )
call msAxisFontSize( char_height )
call msAxis( [ -0.1d0, 2.1d0, -0.1d0, 2.1d0] )
call msAxis( "equal" )

call msPlotPLdomain( PL_domain, color=[0.75d0,0.75d0,0.75d0],      &
                    linewidth=4.5d0,                               &
                    nod_num=.true., edg_num=.true., hol_num=.true. )
```

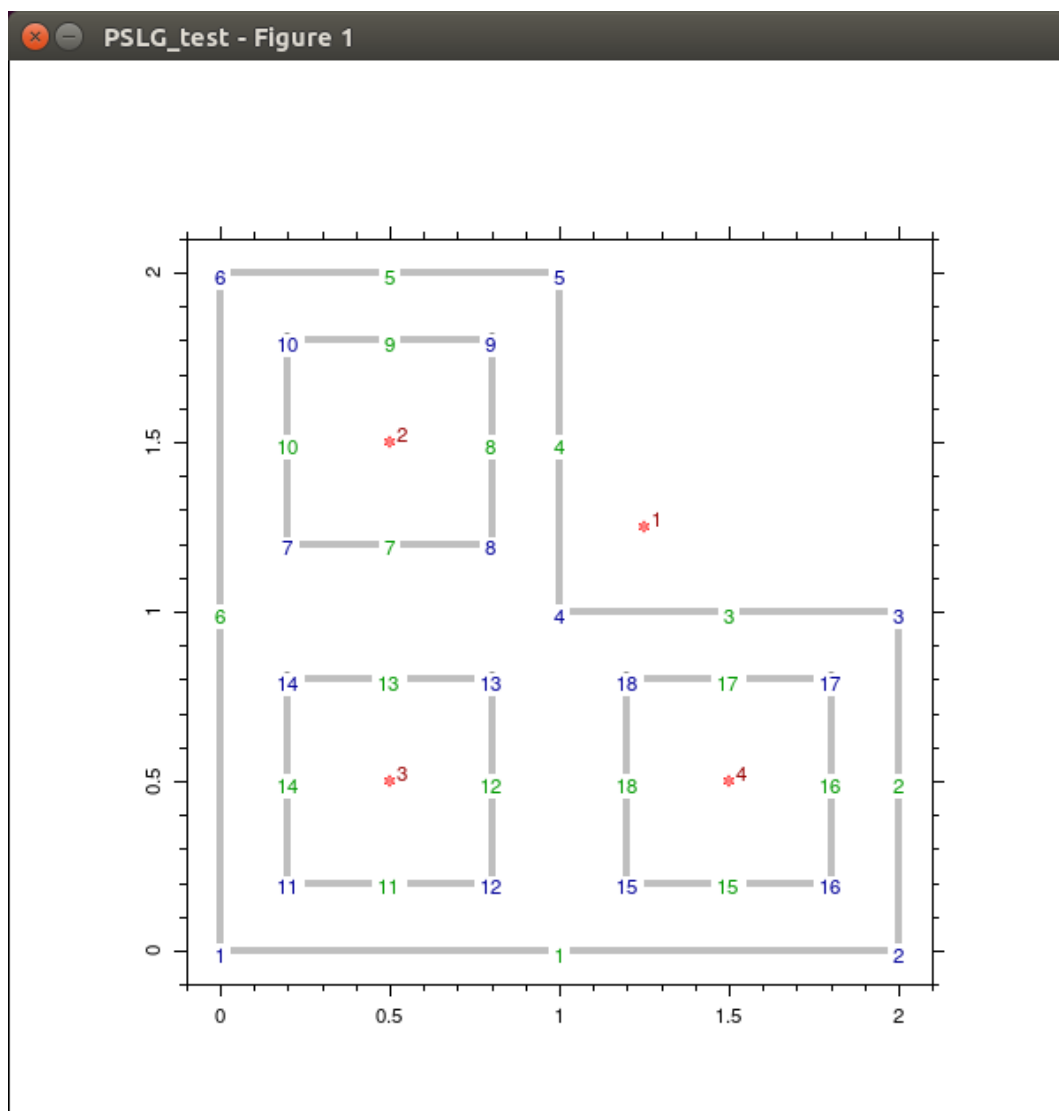


Figure 21: output of ex. #5-10: `msPlotPLdomain` use  
(nodes are drawn in blue; boundary segments in green; holes in red)

The triangulation of the same PL domain is made as follows:

► Example #5-11: (see output plot in figure 22)

```
call msFigure(2,size=[600,600])
call msCharInPixels( "on" )
call msAxisFontSize( char_height )
call msAxis( [ -0.1d0, 2.1d0, -0.1d0, 2.1d0] )
call msAxis( "equal" )

call msDelaunay( mfOut(x,y,tri), PL_domain, theta_min=28.6d0,      &
                  area_max=0.1d0 )

call msBuildTriConnect( x, y, tri, tri_connect )

call msTriMesh( tri_connect, color=[0.75d0,0.75d0,0.75d0],      &
                nod_num=.true., tri_num=.true., fac_num=.true. )
```

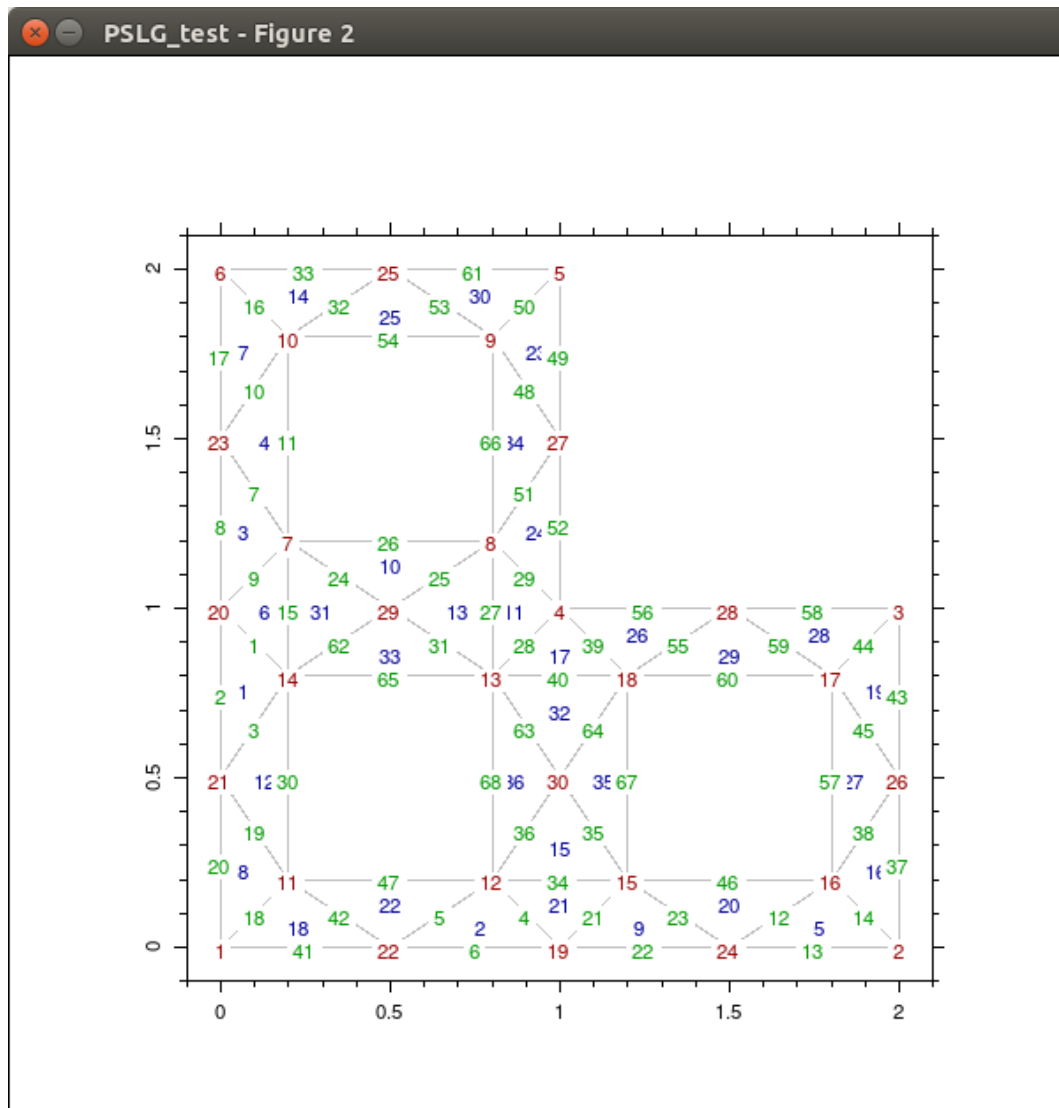


Figure 22: output of ex. #5-11: msTriMesh use, after a Delaunay triangulation (triangles are drawn in blue; nodes in red; faces in green)

The four previous routines (`msPcolor`, `msContour`, `msContourF`, `msQuiver` and `msStreamline`) work only for 2D regular data. Other routines are available when data are spread on an irregular 2D grid: `msTriPcolor`, `msTriContour`, `msTriContourF`, `msTriQuiver` and `msTriStreamline`.

► Example #5-12: (see output plot in figure 23)

```
call msFigure(1)

call msColormap( "rainbow" )
call msAxis( "equal" )

x = [ 0.05d0, 1.05d0, 1.0d0, -0.05d0, 0.5d0 ]
y = [ 0.0d0, -0.05d0, 1.0d0, 0.95d0, 0.5d0 ]
z = x + y
call msDisplay( .t.(x .vc. y .vc. z), "nodes (x, y, value)" )

tri = mfDelaunay( x, y )
call msDisplay( tri, "triangles" )

call msCAxis( [ 0.0d0, 2.0d0 ] )

call msTriMesh( x, y, tri, color=[0.75d0,0.75d0,0.75d0] )
call msAxis( [ -0.1d0, 1.1d0, -0.1d0, 1.1d0 ] )
call msHold( "on" )

call msTriContour( x, y, z, tri, linewidth=2.0d0 )
call msColorbar( "on" )
```

► Output of ex. #5-12:

```
nodes (x, y, value) =

  0.0500    0.0000    0.0500
  1.0500   -0.0500    1.0000
  1.0000    1.0000    2.0000
 -0.0500    0.9500    0.9000
  0.5000    0.5000    1.0000

triangles =

  4      1      5
  5      2      3
  2      5      1
  5      3      4
```

► Example #5-13: (see output plot in figure 24)

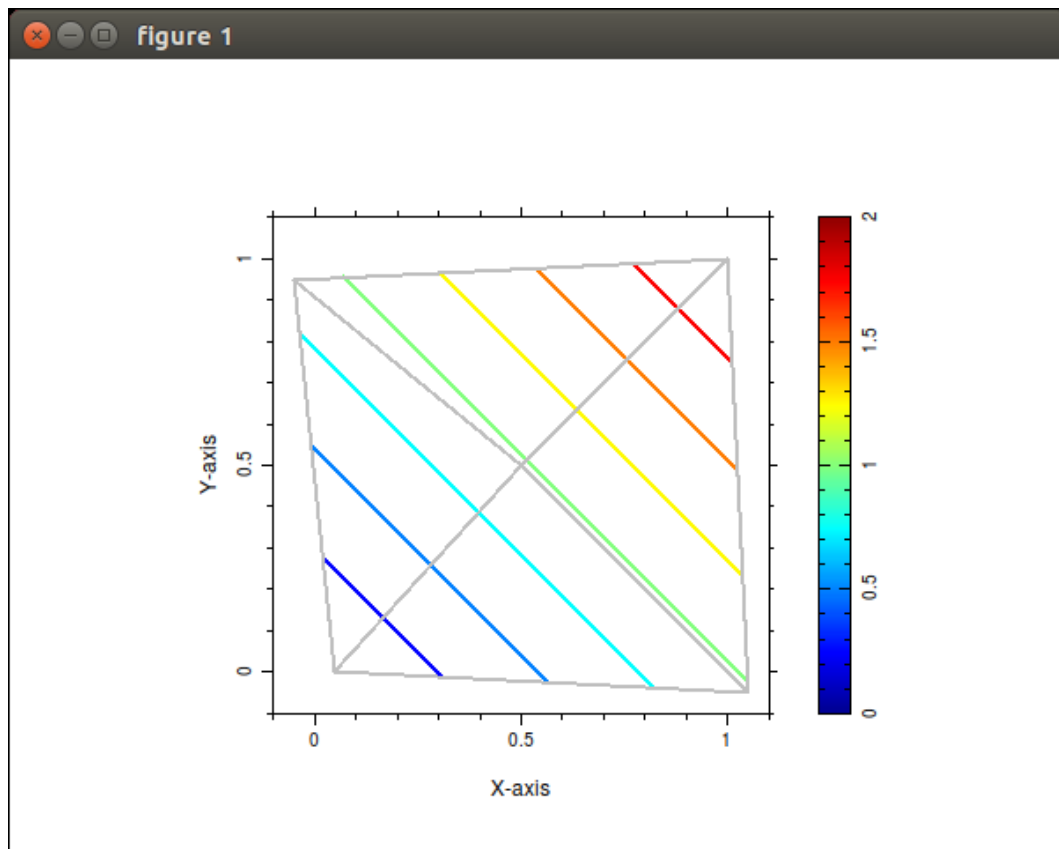


Figure 23: output of ex. #5-12: msTriContour use

```

call msFigure(1)

call msColormap( "rainbow" )
call msAxis( "equal" )

x = [ 0.05d0, 1.05d0, 1.0d0, -0.05d0, 0.5d0 ]
y = [ 0.0d0, -0.05d0, 1.0d0, 0.95d0, 0.5d0 ]
z = x + y
call msDisplay( .t.(x .vc. y .vc. z), "nodes (x, y, value)" )

tri = mfDelaunay( x, y )
call msDisplay( tri, "triangles" )

call msAxis( [ -0.1d0, 1.1d0, -0.1d0, 1.1d0 ] )
call msCAxis( [ 0.0d0, 2.0d0 ] )
call msShading( "interp" ) ! default is flat

call msTriPcolor( x, y, z, tri )

call msHold( "on" )
call msTriMesh( x, y, tri, color=[0.75d0,0.75d0,0.75d0] )
call msColorbar( "on" )

```

▷ Output of ex. #5-13:

```

nodes (x, y, value) =

    0.0500    0.0000    0.0500
    1.0500   -0.0500    1.0000
    1.0000    1.0000    2.0000
   -0.0500    0.9500    0.9000
    0.5000    0.5000    1.0000

triangles =

    4     1     5
    5     2     3
    2     5     1
    5     3     4

```

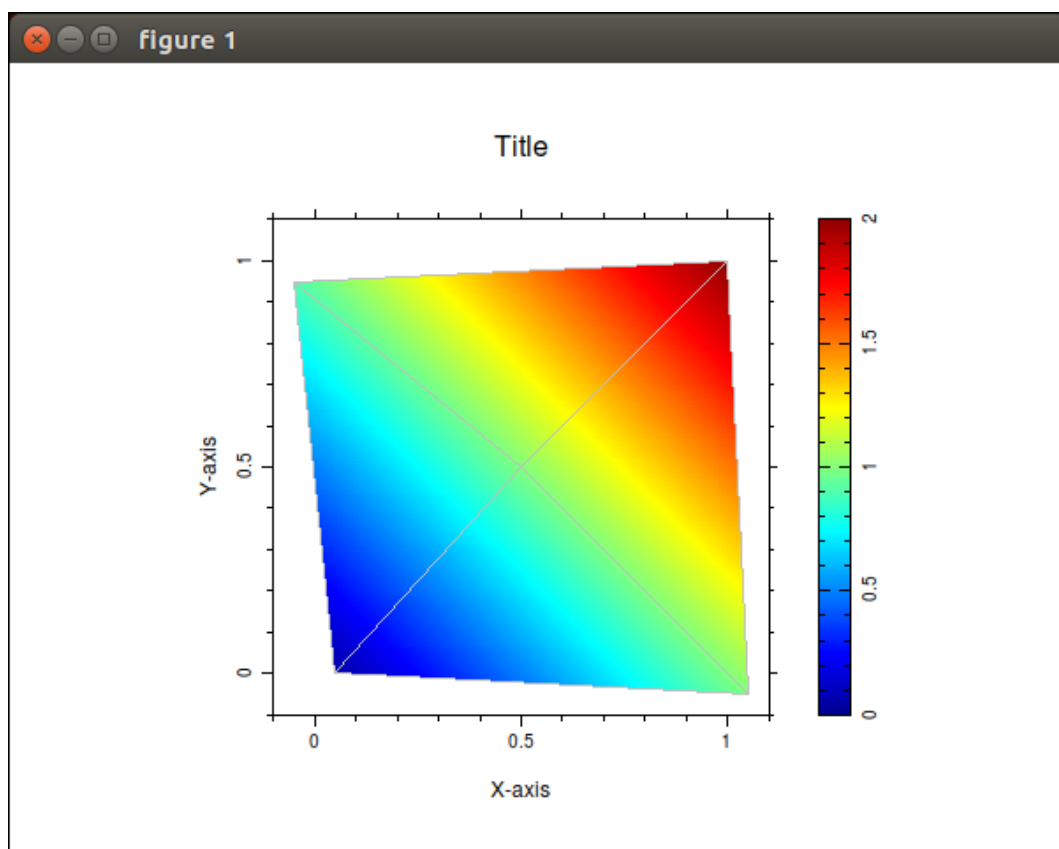


Figure 24: output of ex. #5-13: msTriPColor use

In order to display bitmap images with `msImage`, you must first load the image by using `mfImRead`.

► Example #5-14: (see output plot in figure 25)

```
! character(len=80) :: image_name
! type(mfArray) :: image, cmap

call msFigure(1)
call msAxis( "equal" )

image_name = "images/XPM_image.xpm"
call msImRead( mfOut(image,cmap), trim(image_name) )
call msColormap( cmap )

call msImage( image )
call msTitle( trim(image_name) )
```

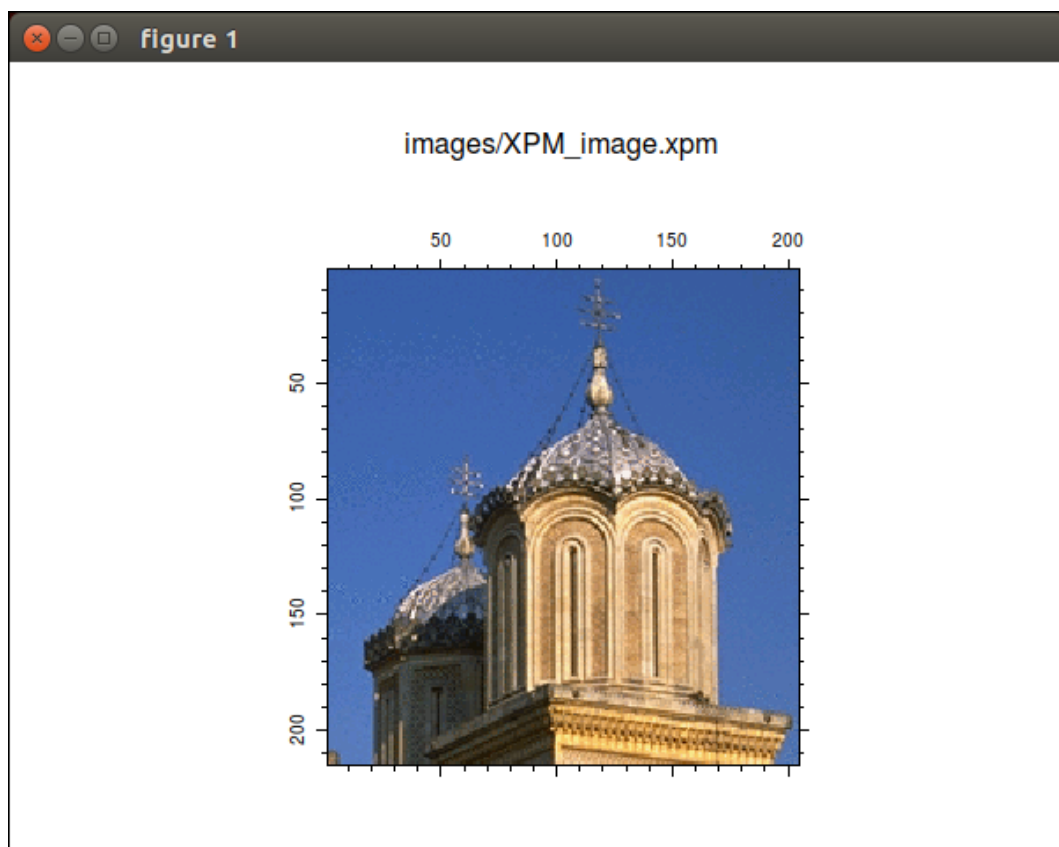


Figure 25: output of ex. #5-14: `msImage` use

Transparency can be set in your image by two ways:

1. for the real valued pixels storing scheme, set a color (in the colormap `cmap`) to the triplet (NaN, NaN, NaN).
2. for the indexed pixels storing scheme, set an element of the `mfArray image` to NaN.

`msPlotHist` can be used to build and plot histograms.

► Example #5-15: (see output plot in figure 26)

```
! integer :: n_max, n, nbin
! real(kind=MF_DOUBLE) :: hope

n_max = 1.0e7
n = 1.0e6
nbin = 60
hope = dble(n)/dble(nbin)

call msFigure(2)

x = mfRandN( n, 1 )
call msPlotHist( mfOut(), x, -3.0d0, 3.0d0, nbin )

call msHold( "on" )

x = mfLinspace( -3.0d0, 3.0d0, 500 )
y = 0.10*n/sqrt(2.0d0*MF_PI)*mfExp(-0.5d0*x**2)
call msPlot( x, y, "--r" )

call msXLabel("Value"); call msYLabel("N")
call msTitle( "Normal Distribution" )
```

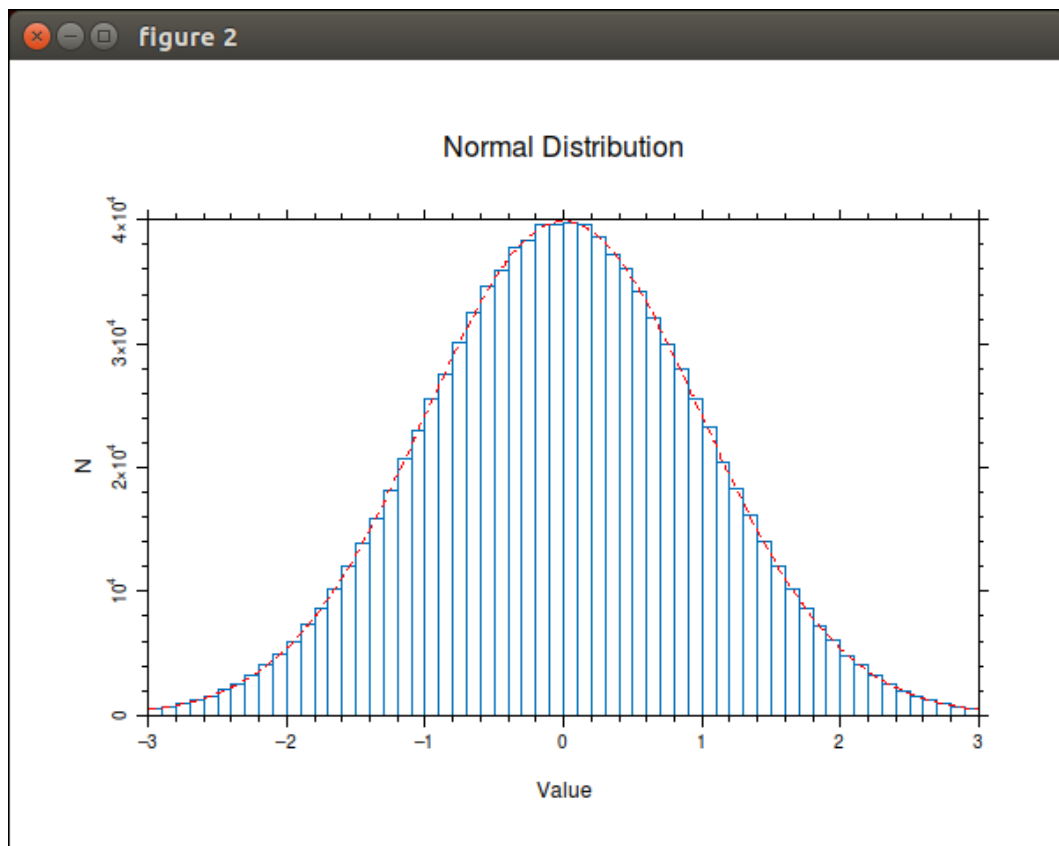


Figure 26: output of ex. #5-15: `msPlotHist` use



`msSpy` is exclusively used to plot the pattern of a sparse `mfArray`.

► Example #5-16: (see output plot in figure 27)

```
A = mfLoadSparse( "data/C.csc", format="CSC" )  
call msSpy( A )  
! set title
```

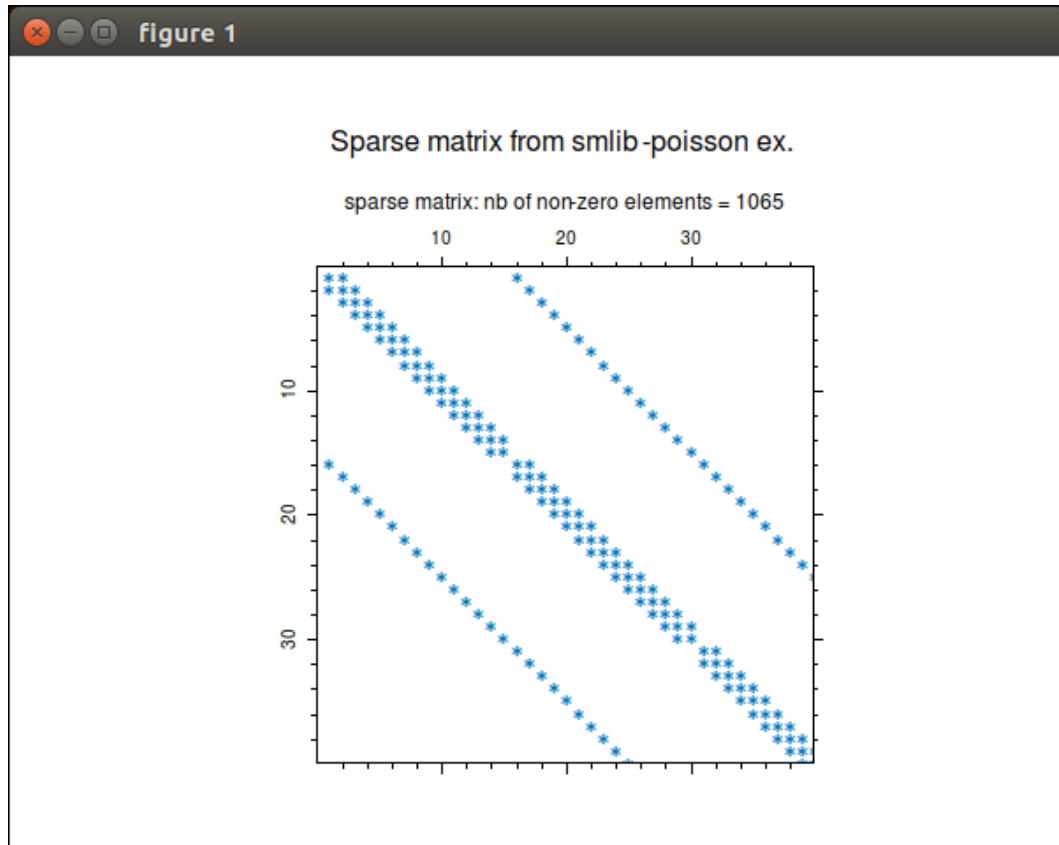


Figure 27: output of ex. #5-16: `msSpy` use tp visualize a 225x225 sparse matrix (zoom)

## 5.5 Changing Orientation/Transposition of Matrix Data

Consider some 2D numerical data that you want to visualize with some Muesli routines as `msPColor` or `msContour`. The numerical data is most of times stored in an `mfArray`, that is a 2D matrix. You can use the above-mentioned routines directly on the `mfArray`, without providing explicit coordinates (`X`, `Y`): by default, the  $(i, j)$  matrix indices will be mapped to the axes  $(-Y, X)$  of the figure, so data visualization will be coherent with the matrix layout. If this orientation is not appropriate, you have two different ways:

1. use the `.t.` transposition operator, and/or the transformation routines `mfFlipLR`, `mfFlipUD` and `mfRot90`, to modify the matrix data itself;
2. or, provide explicit coordinates (`X`, `Y`) of each data point, *i. e.* use matrix coordinates.

In what follows, we explain in details the second way, and more specifically how to generate the matrix coordinates via the `msMeshGrid` routine. Note that the same considerations apply to treat vector fields (couples of 2D numerical data) that can be visualized by `msQuiver` or `msStreamline`.

Of course, if you want to have a non-linear mapping between the matrix indices  $(i, j)$  and the coordinates of the axes  $(X, Y)$ , the only possibility is to use matrix coordinates. The usual procedure is (a) to generate the rectangular matrices (`X`, `Y`) by calling the `msMeshGrid` routine and (b) to apply some function to these coordinates by use of `mfGridFun`.

### 5.5.1 Matrix Orientation in Practice

Let's consider that a matrix data  $M$ . Without loss of generality, we can restrict it to the  $2 \times 2$  matrix, which can also be represented by pseudo-colors (as would the `msPColor` routine):

$$M = \begin{bmatrix} R & G \\ B & Y \end{bmatrix} = \begin{array}{|c|c|} \hline \text{red} & \text{green} \\ \hline \text{blue} & \text{yellow} \\ \hline \end{array}$$

There are only 8 different representations of the matrix  $M$  (keeping the same relative position between the elements), 4 of the original matrix, and 4 others from its transpose. They are summarized in the following table:

The different matrix coordinates ( $X, Y$ ) shown in Table 1 can be easily obtained by using the `msMeshGrid` routine with the generator vectors given in the last column.

For example: most of the time, you want to visualize a matrix with the same orientation as shown by a text editor, *i. e.* the first line of the matrix at the top of the figure. Suppose that the coordinates of the data points extend to  $[X_{min}, \dots, X_{max}]$  for  $X$  (columns) and to  $[Y_{min}, \dots, Y_{max}]$  for  $Y$  (rows), then the following command should be used to generate the appropriate coordinates `X` and `Y`:

```
v_x =      mfLinSpace(x_min,x_max,N)
v_y = .t. mfLinSpace(y_max,y_min,N)
call msMeshGrid( mfOut(X,Y), v_x, v_y )
```

where `N` is the number of coordinates points for each direction. Note that the second generator vector is inverted, as specified in the first row of Table 1. Then you can visualize the pseudo-colors of your matrix by the call

```
call msPColor( X, Y, Z )
```

or even, more simply, by using the generator vectors (this avoids the call of `msMeshGrid` above)

```
call msPColor( v_x, v_y, Z )
```

assuming that your matrix data is stored in the `mfArray` `Z`.









Representation	Comment	Matrix Coordinates	Generator vectors
	original matrix	$X = \begin{bmatrix} 0 & a \\ 0 & a \end{bmatrix} \quad Y = \begin{bmatrix} b & b \\ 0 & 0 \end{bmatrix}$	$v_x = [0, a] \quad v_y = \begin{bmatrix} b \\ 0 \end{bmatrix}$
	orig. after a 90° rot.	$X = \begin{bmatrix} 0 & 0 \\ a & a \end{bmatrix} \quad Y = \begin{bmatrix} 0 & b \\ 0 & b \end{bmatrix}$	$v_x = \begin{bmatrix} 0 \\ a \end{bmatrix} \quad v_y = [0, b]$
	orig. after a 180° rot.	$X = \begin{bmatrix} a & 0 \\ a & 0 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & 0 \\ b & b \end{bmatrix}$	$v_x = [a, 0] \quad v_y = \begin{bmatrix} 0 \\ b \end{bmatrix}$
	orig. after a 270° rot.	$X = \begin{bmatrix} a & a \\ 0 & 0 \end{bmatrix} \quad Y = \begin{bmatrix} b & 0 \\ b & 0 \end{bmatrix}$	$v_x = \begin{bmatrix} a \\ 0 \end{bmatrix} \quad v_y = [b, 0]$
	transposed matrix	$X = \begin{bmatrix} 0 & 0 \\ a & a \end{bmatrix} \quad Y = \begin{bmatrix} b & 0 \\ b & 0 \end{bmatrix}$	$v_x = \begin{bmatrix} 0 \\ a \end{bmatrix} \quad v_y = [b, 0]$
	transp. after a 90° rot.	$X = \begin{bmatrix} 0 & a \\ 0 & a \end{bmatrix} \quad Y = \begin{bmatrix} 0 & 0 \\ b & b \end{bmatrix}$	$v_x = [0, a] \quad v_y = \begin{bmatrix} 0 \\ b \end{bmatrix}$
	transp. after a 180° rot.	$X = \begin{bmatrix} a & a \\ 0 & 0 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & b \\ 0 & b \end{bmatrix}$	$v_x = \begin{bmatrix} a \\ 0 \end{bmatrix} \quad v_y = [0, b]$
	transp. after a 270° rot.	$X = \begin{bmatrix} a & 0 \\ a & 0 \end{bmatrix} \quad Y = \begin{bmatrix} b & b \\ 0 & 0 \end{bmatrix}$	$v_x = [a, 0] \quad v_y = \begin{bmatrix} b \\ 0 \end{bmatrix}$

Table 1: The eight possible representations for the same matrix data  $M$ . Here the intervals  $[0, a]$  and  $[0, b]$  used in the coordinates are for the sake of simplicity; in reality, they should be replaced by  $[X_{min}, \dots, X_{max}]$  for  $X$ , and by  $[Y_{min}, \dots, Y_{max}]$  for  $Y$

## 5.6 Interactive Routines

Interactive routines are those which require the user intervention:

- **msPan** translates the axes of the current figure;
- **msZoom** changes the axes scaling of the current figure;
- **msPanAndZoom** does both scrolling and zooming in the current figure;
- **mfGinput** returns in an **mfArray** the coordinates of the clicked point; optionally, you can get which mouse button has been used, or which keyboard key has been pressed; the other form **msGinput** has even more capabilities;
- **mfGinputRect** returns in an **mfArray** the bounding box of the drawn rectangle;
- **msMoveLegend** moves the legends frame inside the axes;
- **msMoveGrObj** allows the user to move some types of graphic objects with the mouse.

In a standard usage, the mouse is used to move the pointer and its five buttons may be detected. When the mouse is not present, the user can use some letters of the keyboard to emulate the mouse:

- capital **L** for the left button;
- capital **M** for the middle button;
- capital **R** for the right button;
- **2** for scroll up (with the wheel);
- **8** for scroll down (with the wheel).

Moreover, be aware that a click of the mouse leads to two different X11 events: a *MouseDown* event followed by a *MouseUp* one. Knowing that, you will have to type two keys to simulate a click: for example, a left-button click is emulated by typing **L** followed by any key. On the contrary, the mouse wheel generates only one X11 event, like ordinary keys of the keyboard.

To help you to take appropriate actions, as usual, the pointer changes its shape on the screen. Table 2 shows the different pointer shapes.








Pointer shape	Name	Remark and action
	left arrow	normal cursor (nothing to do)
	crosshair	select a particular point to get its coords, or begin/end a rectangular selection
	resize	wait for a window resizing
	watch	in progress operation (most of time: redrawing the figure)
	opened hand	ready to grab an graphic object or the axes zone in order to move it
	closed hand	ready to move (an object is currently grabbed)
	zoom	ready to select a zoomed zone

Table 2: The different shapes of the pointer, which depends on the context and the interactive routine used.

As noted in the *Muesli Reference Manual* (at the page describing the `msMoveGrObj` routine), switching from one workspace to another may break the interaction between the mouse and your Muesli program.

## 5.7 Annotating a Figure

You may add some additional decorations to a figure, with the use of `msText`, `msXLabel`, `msYLabel` and `msTitle`. Here is an example.

- Example #5-17: (see output plot in figure 28)

```

! character(len=72) :: string

call msFigure(1)

! before annotate, the axes must exist
call msAxis( [ -0.05d0, 1.05d0, -0.05d0, 1.05d0] )

string = "color = 'indian red', inclined at 30 degrees"
call msText( 0.0d0, 0.0d0, trim(string), angle=30.0d0,           &
             height=2.0d0, color="indian red" )
call msHold( "on" )
call msText( 0.1d0, 0.85d0, "RGB=[0.2,0.3,0.7]", angle=-20.0d0, &
             height=2.0d0, color=[0.2d0,0.3d0,0.7d0] )

call msTitle( "The title" )
call msXLabel( "X-axis" )
call msYLabel( "Y-axis" )

```

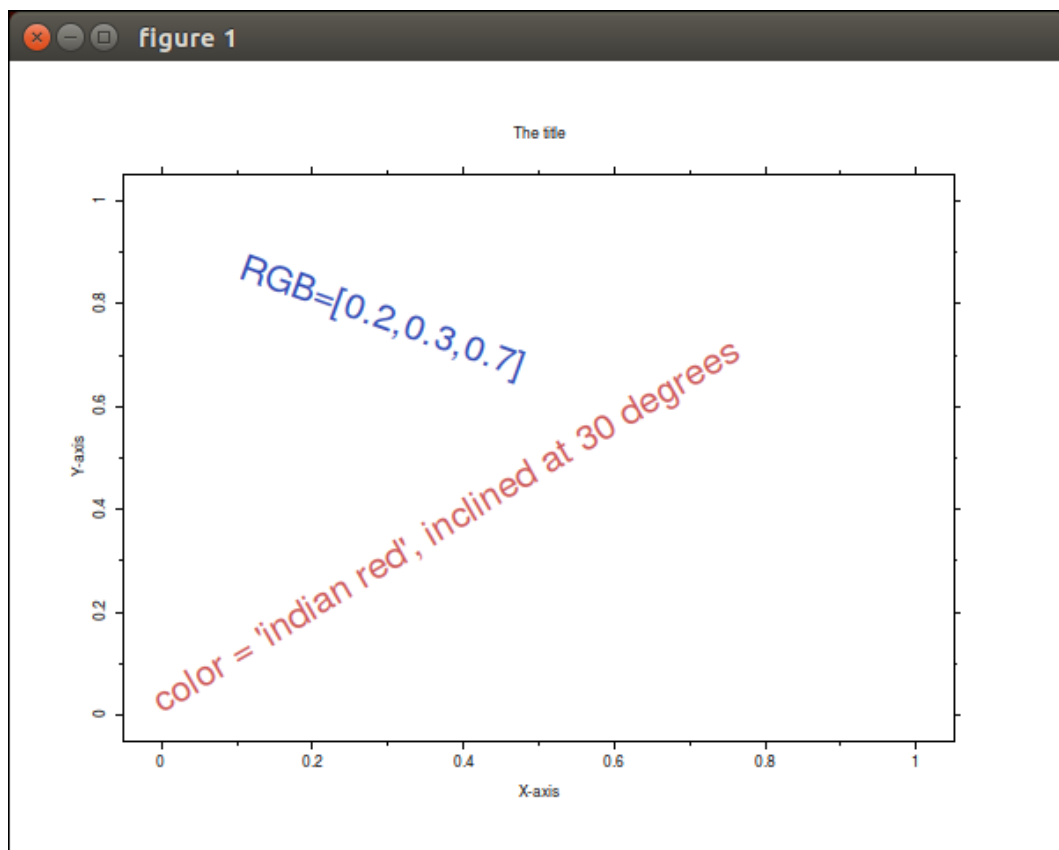


Figure 28: output of ex. #5-17: annotations on a figure

If you have multiple curves plotted in the same figure, you can also use the `msLegend` to add some legend to the curves; up to six curves are supported. An example can be found below.

- Example #5-18: (see output plot in figure 29)

```

call msFigure(1)
call msAxis( [ -0.1d0, 1.1d0, -0.1d0, 1.1d0 ] )
call msHold( "on" )

x = [ 0., 0.333, 0.333, 0.666, 0.666, 1. ]
y = [ 0., 0., 0.6, 0.6, 0.1, 0.1 ]
call msPlot( x, y, "b" )

x = [ 0., 0.25, 0.25, 0.5, 0.5, 1. ]
y = [ 0.5, 0.5, 0.1, 0.1, 0.25, 0.25 ]
call msPlot( x, y, "r--" )

x = [ 0., 0.15, 0.15, 0.75, 0.75, 1. ]
y = [ 0.3, 0.3, 0.2, 0.2, 0.4, 0.4 ]
call msPlot( x, y, "mo-" )

x = mflinSpace( 0.0d0, 1.0d0, 25 )
y = 0.333d0 + 0.333d0*mfSin( 2*MF_PI*x )
call msPlot( x, y, "k+" )

call msText( 0.05d0, 0.05d0, "Hello", height=2.0d0 )

call msLegend( "blue polyline", "red dashed line",
               "magenta marked line", "black crossed sine" )

```

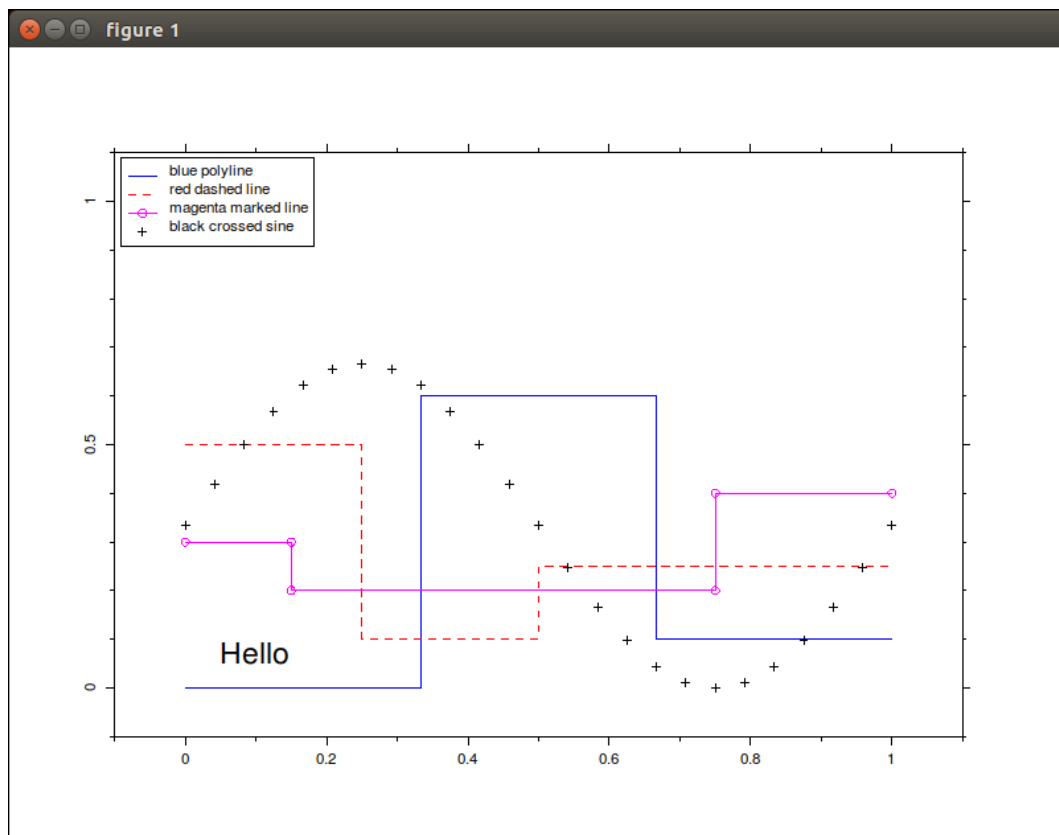


Figure 29: output of ex. #5-18: displaying legends to curves

Once drawn on the figure, this legends frame can be interactively moved via the `msMoveLegend`. By default, the legends frame is drawn near the top left corner.

Finally, `msClf` clears the current figure but doesn't close its window.

## 5.8 Text, Symbols and Fonts

In this section, we describe how characters can be displayed in a figure.

Letter's and symbol's display is essentially done by the following routines, described in the previous section: `msText`, `msXLabel`, `msYLabel` and `msTitle`. They all accept a character string as argument.

### 5.8.1 About the Character Sets used

The abovementioned character string contains usually ASCII encoded characters (ASCII set comprises 128 characters in all: most of them are glyphs, but there are also 32 non printable characters).

But the string argument of the text routines supports also 8-bits encoded characters, namely the ISO-8859-1 – or Latin-1 – character set (called for short “charset”). It adds, to the 128 ASCII letters, other symbols and accentuated letters, mainly for european languages. By default, MUESLI suppose that the strings are in this charset; it also accepts UTF-8 character string (some symbols are coded on two or three characters), but you must explicitly tell this to MUESLI by calling the `msSetCharEncoding` routine. *However, only glyphs which correspond to Latin-1 characters will be actually displayed.*

► Example #5-19: (see output plot in figure 30)

```
! assuming that this code is stored in an ISO-8859-1 encoded file.
call msFigure(1)
call msAxis( "off" )

call msText( 0.0d0, 0.6d0,                                     &
             "É Ê Ç À Û Ü Î Ï Ò          é è ç à ù û ü î ï ô", height=2.0d0 )

call msText( 0.0d0, 0.4d0,                                     &
             "Â Ã Ä Å Æ Ñ Ò Ö Ø Ý      â ã ä å æ ñ ò ö ø ý", height=2.0d0 )
```

### 5.8.2 Greek Alphabet and other Symbols

Greek letters are available either via the short escaped sequence: `\gl`, where *l* is a latin letter, or via the more explicit `\(greek letter name)` escaped sequence inside the character string, as in the table 3.

Commonly used symbols are summarized in the table 4.

26 graph markers are also available via two possible escaped sequences:

1. a short way: `\Mnn` (*i. e.* `\M01` to `\M26`);
2. using the explicit long names: `\Star`, `\Circle`, `\TriangleUp`, `\SquareFilled`, `\DiamondFilled`, etc. See the complete list of the names in the legend of figure 31.

The graph markers *must be called* from the `msPlot` command, *not* from the `msText` command. They are drawn on figure 31. The 8 filled markers (14 to 21) may be printed with a small white border around them, useful when a number of markers overwrite themselves: use a negative number (*i. e.* `-14` to `-21`).

Lastly, concerning the layout of the characters, you may refer to the table 5 for producing superscript, subscript and superposition of characters (note that `\u` and `\d` should always be used in pairs) and also fine tuning for spaces between characters.

### 5.8.3 Fonts available

There are four different fonts; they are:

- normal, abbreviated by the letter **n** (or **N** for bold); this is the default font. It is a sans-serif font (like *Helvetica*).
- roman, abbreviated by the letter **r** (or **R** for bold). It is a serif font, like *Times*.

Greek letter	explicit escaped sequence	short escaped sequence
$\alpha$	<code>\(alpha)</code>	<code>\ga</code>
$\beta$	<code>\(beta)</code>	<code>\gb</code>
$\gamma$	<code>\(gamma)</code>	<code>\gg</code>
$\delta$	<code>\(delta)</code>	<code>\gd</code>
$\epsilon$	<code>\(epsilon)</code>	<code>\ge</code>
$\zeta$	<code>\(zeta)</code>	<code>\gz</code>
$\eta$	<code>\(eta)</code>	<code>\gh</code>
$\theta$	<code>\(theta)</code>	<code>\gq</code>
$\iota$	<code>\(iota)</code>	<code>\gi</code>
$\kappa$	<code>\(kappa)</code>	<code>\gk</code>
$\lambda$	<code>\(lambda)</code>	<code>\gl</code>
$\mu$	<code>\(mu)</code>	<code>\gm</code>
$\nu$	<code>\(nu)</code>	<code>\gn</code>
$\xi$	<code>\(xi)</code>	<code>\gx</code>
$\omicron$	<code>\(omicron)</code>	<code>\go</code>
$\pi$	<code>\(pi)</code>	<code>\gp</code>
$\rho$	<code>\(rho)</code>	<code>\gr</code>
$\sigma$	<code>\(sigma)</code>	<code>\gs</code>
$\tau$	<code>\(tau)</code>	<code>\gt</code>
$\upsilon$	<code>\(upsilon)</code>	<code>\gu</code>
$\phi$	<code>\(phi)</code>	<code>\gf</code>
$\chi$	<code>\(chi)</code>	<code>\gc</code>
$\psi$	<code>\(psi)</code>	<code>\gy</code>
$\omega$	<code>\(omega)</code>	<code>\gw</code>
$\vartheta$	<code>\(varthetaeta)</code>	—
$\varpi$	<code>\(varpi)</code>	<code>\gv</code>
$\varsigma$	<code>\(varsigma)</code>	—
$\varphi$	<code>\(varphi)</code>	<code>\gj</code>

Table 3: Escaped sequences for greek letters – Uppercase glyphs are obtained in changing the first letter in capital (variants have the same capital letter as the basic ones).

Symbol	Description	escaped sequence
Å	Ångström	<code>\A</code>
×	multiply sign	<code>\x</code>
—	minus sign	<code>\-</code>
·	centered dot	<code>\.</code>
œ	oe ligature	<code>\oe</code>
Œ	OE ligature	<code>\OE</code>
<sup>1</sup>	1 as superscript	<code>\1</code>
<sup>2</sup>	2 as superscript	<code>\2</code>
<sup>3</sup>	3 as superscript	<code>\3</code>
\	backslash	<code>\\</code>

Table 4: Escaped sequences for commonly used symbols (some symbols are only valid for some fonts! – see section 5.8.3)



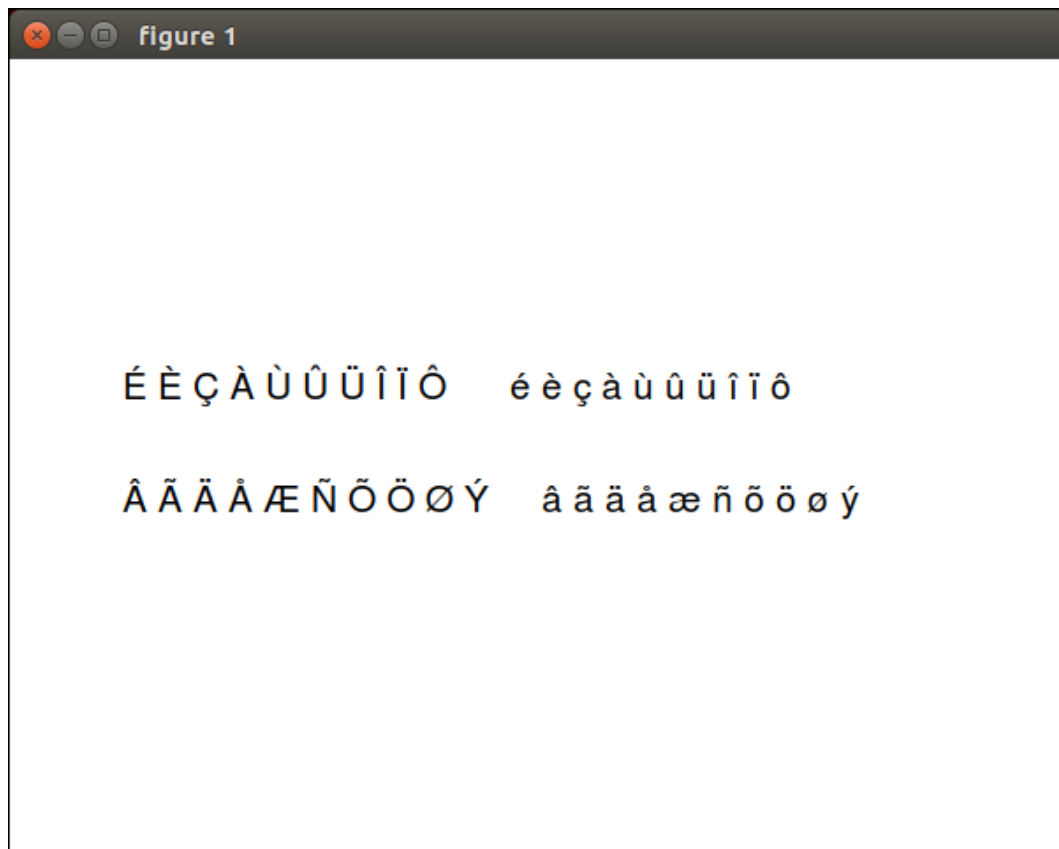


Figure 30: output of ex. #5-19: Latin-1 character string

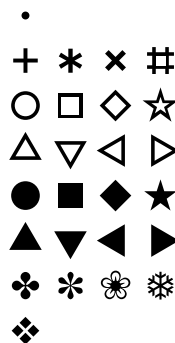


Figure 31: The 26 graph markers available: Dot (M01), Plus (M02), Asterisk (M03), X (M04), Hash (M05), Circle (M06), Square (M07), Diamond (M08), Star (M09), TriangleUp (M10), TriangleDown (M11), TriangleLeft (M12), TriangleRight (M13), CircleFilled (M14), SquareFilled (M15), DiamondFilled (M16), StarFilled (M17), TriangleUpFilled (M18), TriangleDownFilled (M19), TriangleLeftFilled (M20), TriangleRightFilled (M21), HeavyFourBalloon (M22), HeavyTearDrop (M23), WhiteFlorette (M24), Snowflake (M25), BlackDiamondMinusWhiteX (M26).

- italic, abbreviated by the letter **i** (or **I** for bold) (like *Times-Italic*).
- script, abbreviated by the letter **s**. It actually looks close to the *Old English* script font. Be aware that this font embed only the 24 letters of the alphabet (either in the lowercase or the uppercase) and the space character. It is mainly used for mathematical indices.

You are able to switch from one font to another one in the same character string, by using the escaped sequences: `\fn`, `\fN`, `\fr`, `\fR`, `\fi`, `\fI` and `\fs` (be aware that the interpreter of the escaped sequences is case sensitive).

► Example #5-20: (see output plot in figure 32)

Layout	escaped sequence
up one level (superscript)	\u
down one level (subscript)	\d
backspace (superposition)	\b
standard space	\s1
thin space (half of std sp.)	\s2
very thin space (half of thin sp.)	\s3

Table 5: Layout escaped sequences

```

call msFigure(1)
call msAxis( "off" )

call msText( 0.0d0, 0.8d0,                                &
             "\fn A B C D a b c d   \fN A B C D a b c d", height=2.5d0 )
call msText( 0.0d0, 0.6d0,                                &
             "\fr A B C D a b c d   \fR A B C D a b c d", height=2.5d0 )
call msText( 0.0d0, 0.4d0,                                &
             "\fi A B C D a b c d   \fI A B C D a b c d", height=2.5d0 )
call msText( 0.0d0, 0.2d0,                                &
             "\fs A B C D a b c d", height=2.5d0 )

```

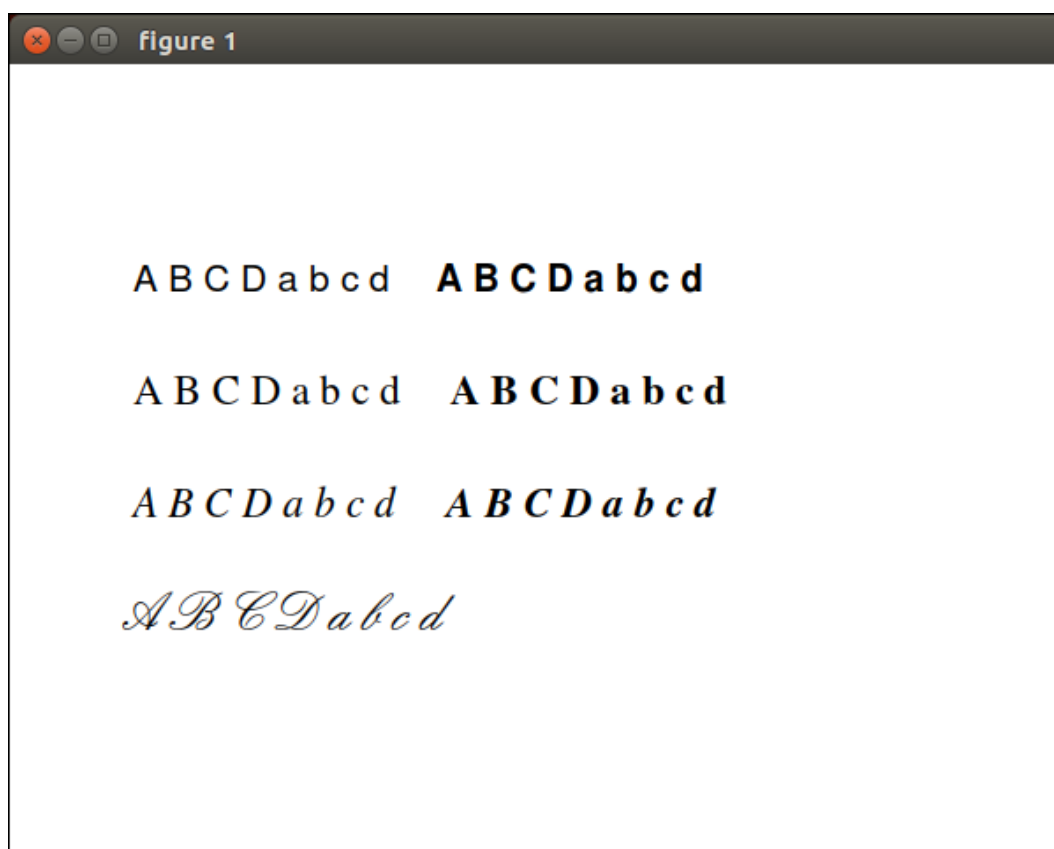


Figure 32: output of ex. #5-20: using different character fonts  
 (from left to right and top to bottom, normal font, normal bold, roman font, roman bold font, roman italic font, roman bold italic font and script font).

Greek letters are exactly the same in all these fonts.

## 5.9 Printing Figures

When you print the current figure (`msPrint` routine), all the graphic objects are drawn once again in another device. Only two devices (or formats) are available, EPS and PDF, which are both vectorized formats. Metadata are inserted in these files (creator, creation date, ...) <sup>8</sup>. Many applications can convert these EPS files in another vectorized format (like SVG) or in bitmap format (like PNG).

The EPS (Encapsulated PostScript) format corresponds to a vectorized description of the figure. So, it can be magnified at any level, without altering the quality of the graphics. Moreover, bitmap objects, like all 'images' are compressed with zlib, providing small size EPS files.

The PDF (Portable Document Format) format is another vector oriented painting model. It has all the advantages of the EPS format but supports native gradients and transparency plus the ability of managing optional contents. See the `msSetPdfOC` routine in the *Muesli Reference Manual*.

By default, all graphic commands are drawn in an X11 window. If your system has no graphic capabilities (or if you intend to execute your program as a batch), you can de-activate the X11 device by calling the routine `msX11Device` with the argument "off", or by setting the environment variable `MFPLLOT_X11_DEVICE` to 0. In such a case, you can nevertheless print them in files, because graphic commands are always kept in memory.

## 5.10 Low-level Graphic Object's Manipulation

While it can be viewed as a high-level routine, the `msPatch` routine is used to draw one polygonal colored face. We prefer to describe it in this sub-section.

`msPatch`, unlike `msPColor`, accept an optional argument about transparency (actually, opacity) of the drawn object.

► Example #5-21: (see output plot in figure 33)

---

<sup>8</sup>As mentioned in the *Muesli Installation Guide*, *Okular* is a better PDF viewer than *Evince*; indeed this latter one doesn't support yet native gradient nor native transparency! (at least until the version 40.1, 2020)

```

call msFigure(1)

call msAxis( [ 0.1d0, 0.8d0, 0.1d0, 0.8d0] )
call msAxis( "equal" )

call msColormap( "rainbow" )

x = [ 0., 1., 0. ] ! vertices x-coord.
y = [ 0., 0., 1. ] ! vertices y-coord.
c = [ 0., 1., 0. ] ! vertices color

call msCAxis( [0.0d0,1.0d0] )
call msShading("interp")
call msPatch( x, y, c, opacity=1.0d0 )

call msHold( "on" )

x = [ 1.1, 1.1, 0. ]
y = [ 0., 1.1, 1.1 ]
c = [ 0.5, 0.5, 0.5 ]

call msPatch( x, y, c, opacity=1.0d0 )

x = [ -0.1, 1.1, 1.1 ]
y = [ -0.1, -0.1, 1.1 ]
c = [ 1., 0., 1. ]
call msPatch( x, y, c, opacity=0.65d0 ) ! transparent object

```

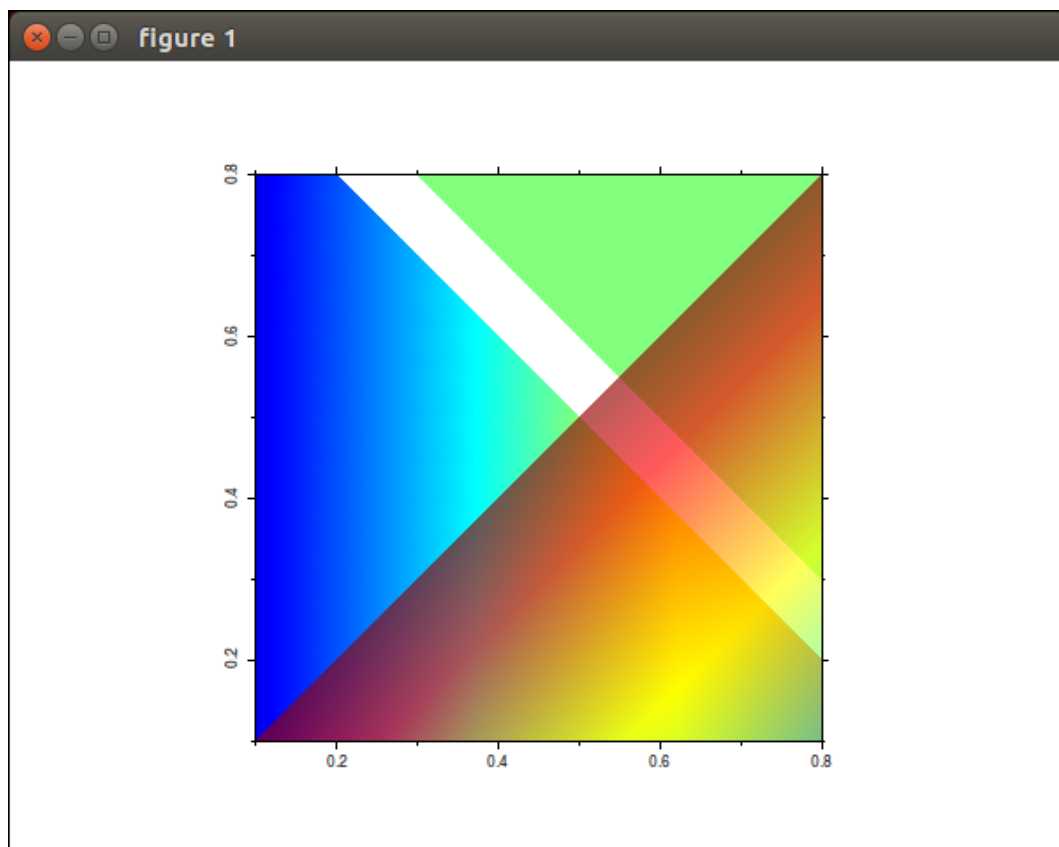


Figure 33: output of ex. #5-21: msPatch use

**msSetGrObj** is used to modify one property of the figure, or one property of a graphic object, which must be accessed by its handle. Therefore, if you plan to dynamically change a property by use of **msSetGrObj**, you must draw the graphic object by the function version of the routine, in order to store its handle. Many properties can be changed, one at a time. See the *MUESLI Reference Manual* for the list of all possibilities.

**msRemoveGrObj** is used to remove a graphic object. Be aware that the object is really deleted; if you want only make it hidden, you must use **msSetGrObj** instead, and set the 'visible' property to 'off'.

**msSetGrObj** and **msRemoveGrObj** make their modifications only in the MUESLI graphic memory. To see the change effects on the screen, the user must call the **msRedrawFigure** which redraw all the objects of the current figure.

## 5.11 Making Animations

Animations can be created using Muesli, but you must be aware that it is not as simple as plotting different curves (or graphic objects) successively.

For example, the following way is too naive and it will lead to flickering, *i.e.* a non smooth display of successive images. More precisely, the longer the drawing time (for example if a large number of graphic objects has to be drawn), the greater the flickering will be, resulting in a quite unpleasant animation.

► Example #5-22: the bad way: *don't use this scheme!*

```
call msFigure(1)
call msHold("off") ! this is the default, but it is showed here just to
                   ! recall that we want to display the curves successively

do k = 1, k_max
  ! ... get or compute some data depending on the iteration number 'k'
  <plotting commands>
  call msPause(duration=0.05d0) ! wait 50 ms to avoid a too fast animation
end do
```

A typical correct sequence of calls should be as follows:

► Example #5-23: the correct way

```
call msFigure(1)
call msAxis( <range> ) ! required to switch to a manual mode
call msAnimation("on")

do k = 1, k_max
  ! ... get or compute some data depending on the iteration number 'k'

  call msCla
  call msDrawGrid      ! optional
  ...
  <plotting commands>
  ...
  call msDrawBox       ! optional

  call msShowNow       ! required to see what has been drawn

  call msPause(duration=0.05d0) ! wait 50 ms to avoid a too fast animation
                                ! (just a timer, not for real-time animation)
end do

call msAnimation("off")
```

The purpose of the `msAnimation` command is multiple: (i) it includes automatically `msHold("on")`, in the case where many plotting commands have to be drawn; (ii) it suppresses the saving of graphic objects' data in the memory, increasing the performance of the plotting; lastly, (iii) it suppresses also the update of the screen after each graphic command (so, the screen update must be done explicitly).

Therefore, a key sequence typically includes:

*Before the loop*

- 1-a `msAxis`: to avoid the automatic change of axes range each time a new plot is done in the loop.
- 1-b `msAnimation`: to enter in the *animation* mode.

*Inside the loop*

- 2-a any computation required by the drawing.
- 2-b `msCla`: erase the area inside the axes (but keep the axes range set by `msAxis`).
- 2-c `msDrawGrid`: (optional) draw the grid before any graphic command (only if the grid has been set before using `msGrid`).
- 2-d the plotting commands: most of FGL routine may be called.
- 2-e `msDrawBox`: (optional) redraw of the axes black line (during the animation, this line may have been colored by some graphic objects clipped at the viewport).
- 2-f `msShowNow`: (required) update the screen.

Please note the important following points:

- just after the animation block, you will not be able to print the last figure (the printed file will be empty), because graphic objects are not saved into memory during the plot, so you cannot *print* the last plot in a PDF file, for example; you must call again the same set of graphic commands after the `msAnimation("off")` call.
- as the graphic objects are volatile, they have no handle; as a consequence, the integer `handle` returned by the `mfPlot` command will be always 0. Therefore, the `msSetGrObj` command cannot be used to modify a previously drawn graphic object.
- the `mfGinput` command may be used in the loop, but only with the `event=.false.` option in order to not stop the animation.

# Index

## MUESLI file extensions

.mbf.gz	46
.mbf	46

## MUESLI operators

**	56
*	21, 56
.and.	18, 23
.but.	17
.by.	17
.hc.	19, 23, 47
.not.	23
.or.	23
.step.	17
.t.	23
.to.	17
.vc.	19, 23, 47
/=	23
/	56
==	23
>	23

## MUESLI parameters

MF_ALL	14
MF_COLON	14
MF_EMPTY	14
MF_END	14, 17
MF_EPS	74–76
MF_E	13
MF_INF	13
MF_I	13
MF_NAN	13
MF_NUMERICAL_CHECK	66
MF_PI	13
MF_REALMAX	74–76
MF_REALMIN	74–76

## MUESLI routines

mfSin	4
All	23, 25
Any	23, 25
mf/msDaeSolve	72
mfAll	74–76
mfAny	74–76
mfBesselI	21
mfBesselJ	21
mfBesselK	21
mfBesselY	21
mfCeil	74–76
mfColon	10
mfComplex	74–76
mfCond	32
mfConj	74–76
mfCount	74, 76
mfCshift	74, 76

mfDaeSolve	45, 67, 73
mfDelaunay	40
mfDiag	23
mfEigs	51
mfEoShift	74, 76
mfErf	21
mfExp	21
mfEye	10
mfFFT	32
mfFSolve	42, 67
mfFZero	42
mfFigure	77
mfFind	32
mfFix	74, 75
mfFlipLR	23, 74, 75, 97
mfFlipUD	23, 74, 75, 97
mfFloor	74–76
mfFourierCos	32
mfFourierLeg	32
mfFourierSin	32
mfFull	50
mfGamma	21
mfGetColorScheme	79
mfGetMsgLevl	62
mfGetTrbLevel	62
mfGet	15, 50
mfGinputRect	98
mfGinput	98, 109
mfGradient	38
mfGridData	36, 37
mfGridFun	97
mfImRead	94
mfImag	74–76
mfInterp2	36
mfInvFFT	32
mfInvFourierCos	32
mfInvFourierLeg	32
mfInvFourierSin	32
mfInv	32
mfIsDense	51
mfIsPosDef	54
mfIsRowSorted	54
mfIsSparse	51
mfIsSymm	54
mfLDiv	26
mfLinSpace	10
mfLoadAscii	46
mfLoadHDF5	46
mfLoadSparse	53
mfLoad	53
mfLsqNonLin	42, 43, 67, 68
mfMax	32, 74–76
mfMean	32
mfMedian	32
mfMerge	74, 76
mfMeshGrid	36

mfMin	32, 74–76	msDisplay	11, 13, 47, 55, 58
mfMul	22, 74–76	msDrawBox	109
mfNnz	75	msDrawGrid	109
mfNodeSearch	40	msEllipKE	21
mfNorm	32	msEnableFPE	63
mfOdeSolve	44, 45, 67	msExitFgl	77
mfOnes	10	msFigure	77
mfOrth	27, 29	msFind	35
mfPLdomain	40	msFlops	59, 60
mfPPVal	35	msFreeArgs	61
mfPack	74, 76	msFreePointer	18
mfPlot	109	msFunFit	39
mfProd	32, 74–76	msGinput	98
mfQuad	42	msGradient	39
mfRDiv	26, 27	msGrid	109
mfRMS	32	msHist	32
mfRand	74–76	msHold	79
mfRank	32	msHorizConcat	47
mfReadLine	14	msImage	78, 94
mfReal	74–76	msInitArgs	61
mfRepMat	20, 23, 74–76	msLU	26
mfReshape	23, 74–76	msLegend	100
mfRot90	23, 74, 75, 97	msLoad	46
mfRound	74–76	msLsqNonLin	44
mfSVDS	51	msMedit	11
mfShape	74–76	msMeshGrid	97
mfSimpson	43	msMoveGrObj	98, 99
mfSin	21	msMoveLegend	98, 101
mfSize	74, 76	msOdeSolve	45
mfSort	32	msPColor	81, 97, 106
mfSpAlloc	46	msPanAndZoom	98
mfSpDiags	46	msPan	98
mfSpEye	46	msPatch	106
mfSpImport	49	msPcolor	78, 91
mfSparse	46	msPlotCubicBezier	79
mfSpline	35, 81	msPlotCubicSpline	80
mfSqrt	21	msPlotHist	95
mfStd	32	msPlotPLdomain	42, 89
mfSum	32, 74–76	msPlot	79, 102
mfSvd	32	msPointer	18
mfTrapz	43	msPolyFit	39
mfTriSearch	40	msPrintHashes	14
mfUnit	56	msPrintPLdomain	40
mfUnpack	74, 76	msPrintProgress	14
mfVar	32	msPrint	106
mfZeros	10	msQR	26
mf_DE.Options	66, 67, 72	msQuad	43
mf_NL.Options	67	msQuiver	86, 91, 97
msAnimation	109	msReadHistoryFile	14
msAssign	59	msRedrawFigure	108
msAxis	77, 79, 109	msRelease	60
msChol	32	msRemoveGrObj	108
msCla	109	msReturnArray	61
msClf	102	msRowSort	54
msColorMap	81	msSVD	26
msContourF	91	msSaveAscii	46
msContour	81, 91, 97	msSaveHDF5	46
msDaeSolve	66	msSaveSparse	53
msDisableFPE	63	msSave	46, 53, 73



msSetAsParameter	14
msSetAutoComplex	64
msSetAutoFilling	19
msSetAutoRowSorted	54
msSetBackgroundColor	77
msSetCharEncoding	102
msSetColorScheme	79
msSetGrObj	108, 109
msSetMsgLevel	14, 59
msSetMsgLevl	62
msSetPdfOC	106
msSetRoundingMode	63
msSetTrbLevel	59, 62
msSet	15, 19, 50
msShowNow	109
msSpExport	49
msSpReAlloc	47
msSpy	78, 96
msStreamline	86, 91, 97
msSvd	32
msText	99, 102
msTitle	99, 102
msTriContourF	91
msTriContour	91
msTriPcolor	91
msTriQuiver	91
msTriStreamline	91
msUsePhysUnits	55
msWriteHistoryFile	14
msX11Device	106
msXLabel	99, 102
msYLabel	99, 102
msZoom	98