Reference Guide

MATFOR 3.0

in Fortran

Contents

CHAPTER 1

Introduction

MATFOR has two main documentations namely `MATFOR User's Guide,` and `MATFOR Reference Guide`.

`MATFOR User's Guide` provides an overview of the MATFOR concepts such as an introduction to mfArray with focus on its constructions and syntax, a introduction to using linear algebra and a quick overview of using the Graphics procedures.

`MATFOR Reference Guide` provides a detailed description of the procedures available in MATFOR. The descriptions include information such as modules to use, procedure syntax, and input and output data type. Examples are included for most procedures. The `Reference Guide` is frequently updated. Please download the latest copy from MATFOR web support page.

This reference guide was written for users who have some background knowledge in programming with Fortran. For more information about using Fortran, please refer to your compiler's documentation.

Typographical Conventions

Before you start using this guide, it is important to understand the terms and typographical conventions used in the documentation.

The following kinds of formatting in the text identify special information.

Formatting Convention	Type of Information
Special Bold	Used to emphasize the importance of a point or a title.
`Emphasis` and `codes`	Represent variable expressions such as parameters, procedures and example codes.

Procedure Descriptions

Convention

The descriptions of MATFOR procedures follow a fixed format. The general format is listed and described below.

Procedure Name

<Summary of procedure >

Module

<This section describes the modules to be included in order to use the procedure>

e.g. `use mod_ess`

Syntax

<This section describes most of the commonly used format of the procedure. Generally, MATFOR procedures accept mfArray as input and output argument. If an argument is not specified, it is an mfArray. Wherever it is convenient, other data types are supported as input argument and are presented as such in the syntax section. For example,

`call msAxis('on')` supports a string data type containing `'on'` as input. You can also use an mfArray containing the string `'on'` as input.>

Descriptions

<This section provides more detailed descriptions of the argument type, and application of the procedures.>

Example

<This section usually presents a program code that uses the current procedure and the result of the program.>

See Also

<This section lists a suggestion of related procedures.>

MATFOR Procedure Naming Convention

MATFOR procedures are provided in function formats and subroutine formats. Three types of prefix are used to classify the MATFOR procedures. One is procedure without prefix, another is procedure with `"mf"` as prefix and the other is procedure with `"ms"` as prefix. Most MATFOR procedures use `"mf"` as prefix for function format and `"ms"` as prefix for subroutine format.

By default, MATFOR procedures use the mfArray as input and output arguments. In special cases, such procedures without prefix accepting Fortran data types as input and output arguments are more convenient. Refer to the MATFOR in Fortran Reference Guide for documents of individual procedures in more details about the type of input and output arguments.

Procedures without prefix

Procedures without prefix such as `SHAPE, SIZE, ALL,` and `ANY` are mainly mfArray inquiry functions that have the same name as intrinsic Fortran array inquiry functions. These functions return a Fortran data type as output. For example, procedure `SHAPE` returns an integer vector as output.

Procedures with "mf" as prefix

Most MATFOR procedures that return a single argument as output use `"mf"` as prefix. These procedures have a function format of the form:

```
out = mfProcedure([mfArray, …])
```

where `out` is the output argument and `[mfArray, …]` is the input argument. For example, `y = mfSin(x), l = mfIsEmpty(a).`

Functions that return only mfArray as the output argument will also have a corresponding subroutine of the format:

```
call msProcedure(mfOut(y, …), x, …)
```

, where x and y are mfArray data types. y is output and x is input.

For example, procedure `y = mfSin(x)` computes sin(x) and returns the result in mfArray `y`. It has a corresponding subroutine counterpart by using the same input and output arguments of the format:

```
call msSin(mfOut(y), x)
```

Procedures with "ms" as prefix

Procedures with "ms" as prefix are subroutines. There are three types of subroutine formats — subroutine that does not return value, subroutine that returns a single output, and subroutine that accepts multi- input and output arguments. Function `mfOut` is used to specify the output arguments.

The subroutines have the following general format:

```
call msSubroutine(mfOut([mfArray]out1,…),
[mfArray]in1,…)
```

where `[mfArray]out1,…` is the list of output arguments and `[mfArray]in1,…` is the list of input arguments. The input and output arguments are optional.

For example,

```
call msViewPause()

call msSurf(x, y, z)

call msSubplot(2, 2, 1)

call msCos(mfOut(y), x)

call msLU(mfOut(l, u), a)

call msMeshgrid(mfOut(a, b), m, n)
```

MATFOR Parameters

The table below lists the MATFOR parameters provided for your convenience. Parameter mf() and MF_COL are mfArrays while the rest of the parameters are double precision data.

Parameter	Data Type	Descriptions
mf()	mfArray	Null mfArray
MF_COL	mfArray	Colon ‘:’ operator
MF_I	Complex(8)	(0.0, 1.0)
MF_PI	Real(8)	π
MF_EPS	Real(8)	The smallest positive number
MF_INF	Real(8)	Positive infinity number.
MF_NAN	Real(8)	Not a number.
MF_E	Real(8)	Natural logarithm number.
MF_REALMAX	Real(8)	Largest representable number.
MF_REALMIN	Real(8)	Smallest representable number.

CHAPTER 2

MOD_ESS

Module MOD_ESS contains the essential set of MATFOR routines to be included in your programs. The procedures included in MOD_ESS are listed below for your reference.

mfArray Manipulation

`mfIsEmpty`	Return true if mfArray is empty.
`mfIsEqual`	Return true if the mfArrays are equal.
`mfIsNumeric`	Return true if mfArray is numerical.
`mfIsReal`	Return true if mfArray is real.
`mfIsComplex`	Return true if mfArray is complex.
`mfIsLogical`	Return true if mfArray is logical.
`mfOut`	Specify a list of mfArrays as output of a procedure.
`mf`	Convert Fortran variable to type mfArray.
`mfSize`	Return the total number of elements in an array, or the extent of an array along a specified dimension.
`mfNDims`	Return the number of dimensions or rank of an mfArray.
`mfShape`	Return shape of the specified mfArray.
`mfAll`	Determination of all elements
`mfAny`	Determination of any element
`mfLength`	Return the largest extent of an mfArray.

Equivalency

`msAssign`	Assign variable to mfArray.

`msPointer`	Assign mfArray target to Fortran pointer.
`mfEquiv`	Make mfArray equivalent to Fortran variable.

`Memory Management`

`msReturnArray`	Set status flag of mfArray to temporary.
`msInitArgs`	Reserve mfArray for computation within procedures.
`msFreeArgs`	Free mfArray for internal housekeeping.

`Display`

`msDisplay`	Display mfArray data on a console window.

mfArray manipulation

mfIsEmpty, mfIsLogical, mfIsReal, mfIsComplex, mfIsNumeric

Return logical true or false for mfArray inquiry.

Module

```
use fml
```

Syntax

```
h = mfIsEmpty(a)
h = mfIsLogical(a)
h = mfIsReal(a)
h = mfIsComplex(a)
h = mfIsNumeric(a)
```

Descriptions

Procedure `mfIsEmpty`, `mfIsLogical`, `mfIsReal`, `mfIsComplex`, and `mfIsNumeric` are a set of mfArray inquiry functions. These functions compare mfArrays or inquire the status of an mfArray and return a logical `.true.` or `.false.`. They can be directly applied in a control if there is a statement. For example, `if (mfIsEmpty(a))`....

`l = mfIsEmpty(a)` returns a logical `.true.` if mfArray `a` is empty, i.e. null, and logical `.false.` otherwise.

`l = mfIsLogical(a)` returns a logical `.true.` if mfArray `a` is logical and logical `.false.` otherwise.

`l = mfIsReal(a)` returns a logical `.true.` if mfArray `a` is real and logical `.false.` otherwise.

`l = mfIsComplex(a)` returns a logical `.true.` if mfArray `a` is complex and logical `.false.` otherwise.

`l = mfIsNumeric(a)` returns a logical `.true.` if mfArray `a` is numeric and logical `.false.` otherwise.

Example

Code

```
program example

use fml
implicit none
type (mfArray):: a, b, c, d, e
```

```
! Construct an empty mfArray.
a = mf()

!Writes a output message if a is empty.
if (mfIsEmpty(a)) write (*,*) 'a is empty'

! Construct three numerical mfArrays.
b = 5.2
c = 5.2
d = (2, 2)

!Writes output message if b is real, b and c are equal, or d is
!numeric or complex
if (mfIsReal(b))    write (*,*) 'b is real'
if (mfIsNumeric(d))  write (*,*) 'd is numeric'
if (mfIsComplex(d))  write (*,*) 'd is complex'

! Construct a logical mfArray.
e = mfMagic(3) > 5

!Writes output message if e is logical.
if (mfIsLogical(e)) write (*,*) 'e is logical'

!Deallocate mfArrays.
call msFreeArgs(a, b, c, d, e)

end program example
```

Result

```
a is empty
 b is real
 d is numeric
 d is complex
 e is logical
```

mf

Convert Fortran data to mfArray.

Module

```
use mod_ess
```

Syntax

```
a = mf([b])
```

Descriptions

Procedure `mf` converts Fortran data type into mfArrays. This is useful in procedure calls where MATFOR restricts the input argument to mfArray type.

`a = mf()`
Return an empty mfArray.

`a = mf(b)`
Convert type real(8), complex(8), integer or character strings argument `b` into an mfArray and return the resulting mfArray as argument `a`.

Example

Procedure `mfTril` extracts lower triangular of an mfArray. You can use `mfTril` to extract a Fortran array by using the mf procedure.

Code

```
program example

use fml
implicit none

type (mfArray) :: lt
real(8) :: A(3,3)

! Initialize A by using the random_number
! procedure provided by Fortran.
call random_number(A)

! Display the content of A by using MATFOR
! display() procedure.
call msDisplay(mf(A), 'A')

! Extract the lower triangular of A by
! using the mfTril() procedure provided by
! MATFOR. Enclose A using mf().
lt = mfTril(mf(A))

call msDisplay(lt, 'Lower Triangular of A')
call msFreeArgs(lt)

end program example
```

Result

```
A  =

   0.0000   0.6669   0.3354
   0.0255   0.9631   0.9153
   0.3525   0.8383   0.7959

 Lower Triangular of A  =

   0.0000   0.0000   0.0000
   0.0255   0.9631   0.0000
   0.3525   0.8383   0.7959
```

See Also

msDisplay

mfOut

Specify a list of mfArrays as output of a function.

Module

```
use fml
```

Syntax

```
mfOut(a, b, c, ...)
```

Descriptions

Procedure `mfOut` specifies a list of mfArrays as output of a subroutine. It differentiates the input arguments from the output arguments in a procedure call. Note that procedure `mfOut` encloses only mfArrays.

For example, you can use function `msFind` to retrieve the row, column indices, and non-zero values of an mfArray as shown below:

```
mfFind(mfOut(i, j, v), x)
```

The procedure returns three mfArrays in this usage. To get the three mfArrays `i`, `j`, and `v`, you must enclose them with `mfOut` in your procedure call. This automatically triggers procedure `msFind` to return the specified mfArrays.

See Also

mfSize, msSize

Show the total number of elements in an array.

Module

```
use fml
```

Syntax

```
type(mfArray) :: a, s, m1, m2, ..., mn

s = mfSize(a)
b = mfSize(a, IDIM)

call msSize(mfOut(m1, m2, ..., mn), a)
```

Descriptions

Procedure `mfSize` returns the total number of elements in an array.

`s = mfSize(a)` returns the number of elements in `a`.

`b = mfSize(a, IDIM)` returns the length of the dimension specified by the scalar `IDIM`.

`call msSize(mfOut(m1, m2, ..., mn), a)` returns the lengths of the first `n` dimensions of `a`.

Note: Procedure msSize assumes the name of mfSize when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a
integer :: s

a = mfRand(3)

s = mfSize(a)
call msDisplay(a, 'mfRand(3)', mf(s), 'mfSize(a)')


s = mfSize(a, 1)
call msDisplay(mf(s), 'mfSize(a, 1)')

s = mfSize(a, 2)
call msDisplay(mf(s), 'mfSize(a, 2)')

call msFreeArgs(a)
```

```
end program example
```

Result

```
mfRand(3)  =

   0.2183   0.3500   0.2183
   0.6074   0.4611   0.6645
   0.1533   0.5731   0.9129

 mfSize(a)  =

  9

 mfSize(a, 1)  =

  3

 mfSize(a, 2)  =

  3
```

See Also

mfShape, mfLength

mfNDims

Return the number of dimensions or rank of an mfArray.

Module

```
use mod_ess
```

Syntax

```
r = mfNDims(a)
```

Descriptions

Procedure `mfNDims` returns the number of dimensions or rank of an mfArray. Each mfArray has a minimum rank of 2. (i.e.scalar, vector and matrix mfArrays have mfNDims = 2)

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a

! Scalar
a = 3
write(*,*) 'Scalar mfArray has mfNDims = ', mfNDims(a)

! Vector
a = (/1, 2, 3/)
write(*,*) 'Vector mfArray has mfNDims = ', mfNDims(a)

! Matrix
a = mfMagic(3)
write(*,*) 'Matrix mfArray has mfNDims = ', mfNDims(a)

! Three-dimensional mfArray
a = mfReshape(mfColon(1,8), (/2,2,2/))
write(*,*) 'Three-dimensional mfArray has mfNDims = ', mfNDims(a)

! Deallocates mfArray
call msFreeArgs(a)

end program example
```

Result

```
Scalar mfArray has mfNDims =          2
 Vector mfArray has mfNDims =          2
 Matrix mfArray has mfNDims =          2
 Three-dimensional mfArray has mfNDims =          3
```

See Also

mfSize, mfLength

mfShape

Return the current shape of an mfArray.

Module

```
use mod_ess
```

Syntax

```
s = mfShape(x)
```

Descriptions

`s = Shape(x)`

- Procedure `Shape` returns a rank-one integer array whose size is equal to the rank of the mfArray.
- `s` is an integer vector, while argument `x` is an mfArray.
- Note that scalar, vector and matrix mfArrays have a rank of 2.

Example

Code

```
program example

use fml
implicit none

type(mfArray) :: x, s

! Scalar mfArray
x = 2
s = mfShape(x)
call msDisplay( x, 'x', s, 'mfShape(x)')

! Vector mfArray
x = (/2, 3/)
s = mfShape(x)
call msDisplay( x, 'x', s, 'mfShape(x)')

! Matrix mfArray
x = mfMagic(2)
s = mfShape(x)
call msDisplay( x, 'x', s, 'mfShpae(x)')

! Three-dimensional mfArray
x = mfRand(2, 2, 2)
s = mfShape(x)
call msDisplay( x, 'x', s, 'mfShape(x)')

! Deallocate mfArray
call msFreeArgs(x, s)

end program example
```

Result

```
x  =

  2

 Shape(x)  =

  1  1

 x  =

  2  3

 Shape(x)  =

  1  2

 x  =

  1  3
  4  2

 Shape(x)  =

  2  2

 x (:,:,1) =

   0.0341   0.7507
   0.8502   0.1210

 x (:,:,2) =

   0.2173   0.2558
   0.4833   0.7332

 Shape(x)  =

  2  2  2
```

See Also

mfAny, mfAll

mfAll, msAll

Return logical true if all elements of mfArray are non-zeros.

Module

```
use fml
```

Syntax

```
type (mfArray) :: l,a
integer :: IDIMS

l = mfAll(a[, IDIMS])
```

Descriptions

Function `mfAll` determines if all elements of an mfArray are non-zeros.

`l = mfAll(x)`

- `l` is a logical mfArray, containing logical element(s).
- If `x` is a vector, `mfAll` returns a scalar containing logical 1 if all elements of `x` are non-zeros, and logical 0 otherwise.
- If `x` is a matrix, `mfAll` is applied to each column of `x`, returning a row vector containing 0's or 1's.

`l = mfAll(x, IDIM)` operates on the dimension of `x` specified by `IDIM`.

Note: Procedure msAll assumes the name of mfAll when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type(mfArray) :: x, y, z
logical :: l

x = mfMagic(3)  > 2

! Check if all elements of x are greater than 0.
l = mfAll(x)

y = mfAll(x, 1)

z = mfAll(x, 2)

call msDisplay(x, 'x', mf(l), 'mfAll(x)', y, 'mfAll(x, 1)',  z, 'mfAll(x,
```

```
2)')

! Deallocates mfArrays
call msFreeArgs(x, y, z)

end program example
```

Result

```
x =

  1  0  1
  1  1  1
  1  1  0

 mfAll(x) =

  0

 mfAll(x, 1) =

  1  0  0

 mfAll(x, 2) =

  0
  1
  0
```

See Also

mfAny

mfAny, msAny

Return logical true if any element of mfArray is nonzero.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, l

integer :: IDIM
l = mfAny(x[, IDIM])
```

Descriptions

Function `mfAny` determines if any element of an mfArray is nonzero.

`l = mfAny(x)`

- `l` is a logical mfArray, containing logical elements.
- If `x` is a vector, `mfAny` returns a scalar containing logical 1 if any element of `x` is nonzero, and logical 0 otherwise.
- If `x` is a matrix, `mfAny` is applied to each column of `x`, returning a row vector containing 0's and 1's.

`l = mfAny(x, IDIM)` operates on the dimension of `x` specified by `IDIM`.

Note: Procedure msAny assumes the name of mfAny when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type(mfArray) :: x, y
logical :: l

x = mfMagic(3)

! Check if all elements of x are more than 5.
l = mfAny(x>5)
call msDisplay(x, 'x', mf(l), 'mfAny(x>5)')

y = mfAny(x>5, 1)
call msDisplay(y, 'mfAny(x>5, 1)')
y = mfAny(x>5, 2)
call msDisplay(y, 'mfAny(x>5, 2)')
```

```
! Deallocates mfArrays
call msFreeArgs(x, y)

end program example
```

Result

```
x =

  8  1  6
  3  5  7
  4  9  2

 mfAny(x>5) =

  1

 mfAny(x>5, 1) =

  1  1  1

 mfAny(x>5, 2) =

  1
  1
  1
```

See Also

mfAll

mfLength

Retrieve largest extent of an mfArray.

Module

```
use mod_ess
```

Syntax

```
l = mfLength(a)
```

Descriptions

Procedure `mfLength` returns the largest extent of an mfArray.

`l = mfLength(a)` returns an integer scalar containing the largest extent of mfArray `a`. If `a` is matrix, `mfLength(a)` returns `mfMax(Shape(a))`.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a
integer :: l

a = mfOnes(2, 5)
l = mfLength(a)
call msDisplay(a, 'a', mf(l), 'mfLength(a)')

call msFreeArgs(a)

end program example
```

Result

```
a  =

  1  1  1  1  1
  1  1  1  1  1

 mfLength(a)  =

  5
```

See Also

mfSize

mfArray access

mfMatSub, mfS

Retrieve rows and columns of elements from vector or matrix mfArray.

Module

```
use mod_ess
```

Syntax

```
type(mfArray) :: A, row, column

sub_Array = mfMatSub(A, row, column)
```

Descriptions

Procedure `mfMatSub` retrieves elements from an mfArray. The arguments (except the first one) specify the subscripts of the elements

Notice that MATFOR also provides the abbreviated name `mfS` for this Procedure.

For example, if`A` is a vector mfArray, `mfMatSub(A, "1:100:2")` would return a vector containing the 1st, 3rd, 5th, ..., and 99th elements of A.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y, z
integer :: i

x = reshape((/(i, i= 1, 24)/), (/2, 3, 4/))
call msDisplay(x, 'x')

y = mfS(x, x <= 12)
call msDisplay(y, 'mfS(x, x <= 12)')
y = mfS(x, 1.to.24)
call msDisplay(y, 'mfS(x, 1.to.24)')
y = mfS(x, 1.to.2, 1.to.12)
call msDisplay(y, 'mfS(x, 1.to.2, 1.to.12)')
y = mfS(x, 1.to.2, 1.to.12.step.3)
call msDisplay(y, 'mfS(x, 1.to.2, 1.to.12.step.3)')
y = mfS(x, MF_COL)
call msDisplay(y, 'mfS(x, MF_COL)')
y = mfS(x, 1, 2, 3)
call msDisplay(y, 'mfS(x, 1, 2, 3)')
y = mfS(x, mf((/2, 1/)), mf((/3, 1/)), mf((/4, 1, 3/)))
call msDisplay(y, 'mfS(x, mf((/2, 1/)), mf((/3, 1/)), mf((/4, 1, 3/)))')

y = x
call msAssign(mfS(y, y <= 12 .and. y >= 6 ), 0)
call msDisplay(y, 'msAssign(mfS(y, y <= 12 .and. y >= 6), 0)')
```

```
y = x
call msAssign(mfS(y, 1.to.24), 0)
call msDisplay(y, 'msAssign(mfS(y, 1.to.24), 0)')

y = x
call msAssign(mfS(y, 1.to.2, 1.to.12), 0)
call msDisplay(y, 'msAssign(mfS(y, 1.to.2, 1.to.12), 0)')

y = x
call msAssign(mfS(y, 1.to.2, 1.to.12.step.3), 0)
call msDisplay(y, 'msAssign(mfS(y, 1.to.2, 1.to.12.step.3), 0)')

y = x
z = reshape((/(i, i = 31, 38)/), (/2, 2, 2/))
call msAssign(mfS(y, 1.to.2, 1.to.3.step.2, 1.to.4.step.3), z)
call msDisplay(y, 'msAssign(mfS(y, 1.to.2, 1.to.3.step.2,
1.to.4.step.3), z)')

y = x
call msAssign(mfS(y, MF_COL), .t.(/(i, i= 25, 48)/))
call msDisplay(y, 'msAssign(mfS(y, MF_COL), .t.(/(i, i= 25, 48)/))')

y = x
call msAssign(mfS(y, 1, 2, 3), 60)
call msDisplay(y, 'msAssign(mfS(y, 1, 2, 3), 60)')

y = x
z = reshape((/(i, i = 101, 112)/), (/2, 2, 3/))
call msAssign(mfS(y, mf((/2, 1/)), mf((/3, 1/)), mf((/4, 1, 3/))), z)
call msDisplay(y, 'msAssign(mfS(y, mf((/2, 1/)), mf((/3, 1/)), mf((/4,
1, 3/))), z)')

call msFreeArgs(x, y)

end program example
```

Result

```
x(:,:,1) =

  1  3  5
  2  4  6

 x(:,:,2) =

   7   9  11
   8  10  12

 x(:,:,3) =

  13  15  17
  14  16  18

 x(:,:,4) =

  19  21  23
  20  22  24

 mfS(x, x <= 12) =

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

```
  12

 mfS(x, 1.to.24) =

column  1  to  19 ...
   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
19

column  20  to  24
  20  21  22  23  24

 mfS(x, 1.to.2, 1.to.12) =

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

 mfS(x, 1.to.2, 1.to.12.step.3) =

   1   7  13  19
   2   8  14  20

 mfS(x, MF_COL) =

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

 mfS(x, 1, 2, 3) =

  15

 mfS(x, mf((/2, 1/)), mf((/3, 1/)), mf((/4, 1, 3/)))(:,:,1) =

  24  20
  23  19

 mfS(x, mf((/2, 1/)), mf((/3, 1/)), mf((/4, 1, 3/)))(:,:,2) =

  6  2
  5  1

 mfS(x, mf((/2, 1/)), mf((/3, 1/)), mf((/4, 1, 3/)))(:,:,3) =

  18  14
  17  13

 msAssign(mfS(y, y <= 12 .and. y >= 6), 0)(:,:,1) =

  1  3  5
  2  4  0

 msAssign(mfS(y, y <= 12 .and. y >= 6), 0)(:,:,2) =
```

```
 0  0  0
 0  0  0

msAssign(mfS(y, y <= 12 .and. y >= 6), 0)(:,:,3) =

 13  15  17
 14  16  18

msAssign(mfS(y, y <= 12 .and. y >= 6), 0)(:,:,4) =

 19  21  23
 20  22  24

msAssign(mfS(y, 1.to.24), 0)(:,:,1) =

 0  3  5
 2  4  6

msAssign(mfS(y, 1.to.24), 0)(:,:,2) =

  7   9  11
  8  10  12

msAssign(mfS(y, 1.to.24), 0)(:,:,3) =

 13  15  17
 14  16  18

msAssign(mfS(y, 1.to.24), 0)(:,:,4) =

 19  21  23
 20  22  24

msAssign(mfS(y, 1.to.2, 1.to.12), 0)(:,:,1) =

 0  0  0
 0  0  0

msAssign(mfS(y, 1.to.2, 1.to.12), 0)(:,:,2) =

 0  0  0
 0  0  0

msAssign(mfS(y, 1.to.2, 1.to.12), 0)(:,:,3) =

 0  0  0
 0  0  0

msAssign(mfS(y, 1.to.2, 1.to.12), 0)(:,:,4) =

 0  0  0
 0  0  0

msAssign(mfS(y, 1.to.2, 1.to.12.step.3), 0)(:,:,1) =

 0  3  5
 0  4  6

msAssign(mfS(y, 1.to.2, 1.to.12.step.3), 0)(:,:,2) =

  0   9  11
  0  10  12

msAssign(mfS(y, 1.to.2, 1.to.12.step.3), 0)(:,:,3) =

  0  15  17
  0  16  18

msAssign(mfS(y, 1.to.2, 1.to.12.step.3), 0)(:,:,4) =

  0  21  23
  0  22  24

msAssign(mfS(y, 1.to.2, 1.to.3.step.2, 1.to.4.step.3), z)(:,:,1) =
```

```
 31   3  33
 32   4  34

 msAssign(mfS(y, 1.to.2, 1.to.3.step.2, 1.to.4.step.3), z)(:,:,2) =

  7   9  11
  8  10  12

 msAssign(mfS(y, 1.to.2, 1.to.3.step.2, 1.to.4.step.3), z)(:,:,3) =

 13  15  17
 14  16  18

 msAssign(mfS(y, 1.to.2, 1.to.3.step.2, 1.to.4.step.3), z)(:,:,4) =

 35  21  37
 36  22  38

 msAssign(mfS(y, MF_COL), .t.(/(i, i= 25, 48)/))(:,:,1) =

 25  27  29
 26  28  30

 msAssign(mfS(y, MF_COL), .t.(/(i, i= 25, 48)/))(:,:,2) =

 31  33  35
 32  34  36

 msAssign(mfS(y, MF_COL), .t.(/(i, i= 25, 48)/))(:,:,3) =

 37  39  41
 38  40  42

 msAssign(mfS(y, MF_COL), .t.(/(i, i= 25, 48)/))(:,:,4) =

 43  45  47
 44  46  48

 msAssign(mfS(y, 1, 2, 3), 60)(:,:,1) =

 1  3  5
 2  4  6

 msAssign(mfS(y, 1, 2, 3), 60)(:,:,2) =

  7   9  11
  8  10  12

 msAssign(mfS(y, 1, 2, 3), 60)(:,:,3) =

 13  60  17
 14  16  18

 msAssign(mfS(y, 1, 2, 3), 60)(:,:,4) =

 19  21  23
 20  22  24

 msAssign(mfS(y, mf((/2, 1/)), mf((/3, 1/)), mf((/4, 1, 3/))), z)(:,:,1)
=

 108   3  106
 107   4  105

 msAssign(mfS(y, mf((/2, 1/)), mf((/3, 1/)), mf((/4, 1, 3/))), z)(:,:,2)
=

  7   9  11
  8  10  12

 msAssign(mfS(y, mf((/2, 1/)), mf((/3, 1/)), mf((/4, 1, 3/))), z)(:,:,3)
=

 112   15  110
```

```
  111   16  109

 msAssign(mfS(y, mf((/2, 1/)), mf((/3, 1/)), mf((/4, 1, 3/))), z)(:,:,4)
=

  104   21  102
  103   22  101
```

See Also

Equivalency

msAssign

Assign the value of an mfArray to another.

Module

```
use mod_ess
```

Syntax

```
call msAssign(a, b)
```

Descriptions

Procedure `msAssign(a, b)` assigns value of mfArray `a` to mfArray `b`. This is equivalent to the statement `b = a`.

Note that when you assign one mfArray to another mfArray, there is no memory copy involved. However, when you assign an mfArray to a Fortran data type, the Fortran compiler accomplishes the task by creating a temporary storage space in the stack memory. When the array size involved is large, stack overflow might result. To avoid large data, it is advisable to use `msAssign(a, F)` or `a = mf(F)` instead of `a = F`.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a
real(8) :: F(2,5)

call random_number(F)
call msAssign(a, F)  !Avoids stack memory copy.

call msDisplay(a, 'a')

!Deallocate mfArray
call msFreeArgs(a)

end program example
```

Result

```
a  =

   0.0000   0.3525   0.9631   0.3354   0.7959
   0.0255   0.6669   0.8383   0.9153   0.8327
```

See Also

Shape, Size, mfSize

msPointer

Assign pointer to mfArray.

Module

```
use mod_ess
```

Syntax

```
call msPointer(a, P)
```

```
a : mfArray
P : real(8) or complex(8) pointer
```

Descriptions

Procedure `msPointer` assigns a real(8) or complex(8) pointer to a target mfArray.

```
call msPointer(a, P)
call msPointer(a, P, s)
```

- Procedure `msPointer` associates pointer `P` and mfArray `a`, i.e the memory space occupied by mfArray `a` becomes the target of pointer `P`. Pointer `P` automatically assumes the shape of mfArray `a`.

Warning

Deallocating pointer `P` would lead to exception error.

Example

Code

```
program example

use fml
implicit none

type (mfArray) ::  a, z
real(8), pointer :: pd(:,:)
complex(8), pointer :: pz(:,:)

a = mfOnes(3, 3)
z = mfComplex(a, a)

call msDisplay(a, 'a')

! pd targets a
call msPointer(a, pd)

! a(3, 3) has been set to 5.0.
pd(3, 3) = 5.0
call msDisplay(a, 'a')

call msDisplay(z, 'z')

! pz targets z
```

```
call msPointer(z, pz)

! z(3, 3) has been set to  5.0+5.0i
pz(3, 3) = cmplx(5.0, 5.0)
call msDisplay(z, 'z')

!Deallocate mfArray
call msFreeArgs(a, z)

end program example
```

Result

```
a  =

  1  1  1
  1  1  1
  1  1  1

 a  =

  1  1  1
  1  1  1
  1  1  5

 z  =


 1.0000 +1.0000i   1.0000 +1.0000i   1.0000 +1.0000i
 1.0000 +1.0000i   1.0000 +1.0000i   1.0000 +1.0000i
 1.0000 +1.0000i   1.0000 +1.0000i   1.0000 +1.0000i


 z  =


 1.0000 +1.0000i   1.0000 +1.0000i   1.0000 +1.0000i
 1.0000 +1.0000i   1.0000 +1.0000i   1.0000 +1.0000i
 1.0000 +1.0000i   1.0000 +1.0000i   5.0000 +5.0000i
```

See Also

mfEquiv

mfEquiv

Make mfArray equivalent to Fortran variable.

Module

```
use mod_ess
```

Syntax

```
a = mfEquiv(T)
a = mfEquiv(Z)
```

```
a : mfArray
T : real(8)array or scalar
Z : complex(8)array or scalar
```

Descriptions

Procedure `mfEquiv` is provided for sharing memory storage between a Fortran variable and an mfArray.

`a = mfEquiv(T)`

- Procedure `mfEquiv` returns a restricted mfArray `a` which shares the same memory location as Fortran variable `T`.
- Variable `T` can be 1) real(8) array or scalar or 2) complex(8) array or scalar.
- Use `mfEquiv` when you have an existing Fortran array. The Fortran variable must not be a temporary variable. It must be explicitly declared. The following operation is invalid.

```
a = mfEquiv(dble(A))
! dble(A) returns a temporary variable.
```

Restrictions

MATFOR enforces a few restrictions on the operations of an mfArray when it is constructed by using procedure `mfEquiv`.

- Procedure `msRelease` releases and disassociates the restricted mfArray from the Fortran variable but does not deallocate the memory space occupied by the Fortran variable.
- Operations that change the shape, data type, rank of the restricted mfArray are invalid.
- The restricted mfArray cannot be used as a returned argument in procedures (functions and subroutines).e.g.

```
function(A, B) result (out)
real (8) :: A(3,4)
out = mfEquiv(A)
```

```
! This is illegal, out will be null.
```

- Deallocating the associated Fortran array invalids the mfArray.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: t
real(8) :: A(3,3)
integer :: i

A = dble(ReShape( (/(i,i=1,9)/), (/3,3/)) )
t = mfEquiv(A)
call msDisplay(t, 't')
call msDisplay(mf(A), 'A')

A = A - 1
call msDisplay(t, 't')
call msDisplay(mf(A), 'A')

! If you use the msFreeArgs routine on t,
! t is Deallocated, but A is still valid and usable.
call msFreeArgs(t)
call msDisplay(mf(A), 'A')

end program example
```

Result

```
t  =

  1  4  7
  2  5  8
  3  6  9

 A  =

  1  4  7
  2  5  8
  3  6  9

 t  =

  0  3  6
  1  4  7
  2  5  8

 A  =

  0  3  6
  1  4  7
  2  5  8

 A  =

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

See Also

msPointer

Memory Management

msReturnArray

Set status flag of mfArray to temporary.

Module

```
use mod_ess
```

Syntax

```
call msReturnArray(a)
```

Descriptions

Procedure `msReturnArray(a)` sets the status flag of an mfArray to temporary. This procedure should be added to the end of functions that use mfArray to return values. It ensures that temporary mfArrays created by function or subroutine are deallocated, preventing memory leakage.

For example, when you develop a function `foo` that returns mfArray `t` as below,

```
function foo(A) return (t)
implicit none
integer :: A
type (mfArray) :: t
! program codes
-------------------
-------------------
call msReturnArray(t)
return
end function foo
```

It is recommended to add the statement `call msReturnArray(t)` at the end of the function. This statement sets the status property of the mfArray to temporary. MATFOR releases such temporary mfArrays automatically.

More Details

The mfArray is a dynamically allocated data object. It is a good practice in Fortran to release dynamically allocated memory space once you no longer need it. In lieu with the above principle, you should clear all temporary mfArrays to prevent memory leakage.
When you develop a function for end-users in Fortran, there are many possible ways a user might use. For example, `a = foo(x)` or the user might use it as an input to another function, such as `b = gee(foo(x))`.

In the second case, the function `foo` returns a temporary data object. When the returned data object is an mfArray, it should be released from memory to prevent memory leakage. MATFOR procedures release such temporary mfArrays automatically. By specifying the statement `call msRelease(a, b, c, ...)` at the end of your function, you set the status flag of the mfArray to temporary. MATFOR automatically clears mfArrays once it encounters temporary

flags.

See Also

msInitArgs, msFreeArgs

msInitArgs, msFreeArgs

Set status of mfArrays in procedure development.

Module

```
use mod_ess
```

Syntax

```
call msInitArgs(a, b, ...)
call msFreeArgs(a, b, ...)
```

Descriptions

Procedures `msInitArgs` and `msFreeArgs` are used in procedure development to set the status flag of mfArrays.

Procedure `msInitArgs` stops temporary mfArrays from being released by MATFOR. Procedure `msFreeArgs` reverses the 'lock' action performed by procedure `msInitArgs`, so that temporary mfArrays are automatically released by MATFOR.
For example, when developing a subroutine `foo` that accepts mfArrays `a`, `b`, and `c`, as below, it is recommended to enclose the program codes with `call msInitArgs(a, b, c)`, and `call msFreeArgs(a, b, c)` as below:

```
Subroutine foo(a, b, c)
implicit none
type (mfArray) :: a, b, c
call msInitArgs(a, b, c)
 ! program codes
  ------------------
  ------------------
call msFreeArgs(a, b, c)
return
end Subroutine foo
```

More Details

The mfArray is a dynamically allocated data object. It is a good practice in Fortran to release dynamically allocated memory space once you no longer need it. In lieu of the above principle, MATFOR releases all temporary mfArrays automatically once they are passed into MATFOR procedures.
A temporary mfArray is created when you use a function without assigning its output to a variable. For example,

```
call bar(foo(x))
```

returns a temporary mfArray used as input to the subroutine bar(). Such temporary mfArrays have their status flag sets to temporary if the function developer added the statement below at the end of the function:

```
call msReturnArray(a)
```

MATFOR automatically clears mfArrays with temporary flag 'on' in MATFOR procedures. To prevent temporary mfArray from being released in a subroutine code, the procedure `msInitArgs()` is used to stop MATFOR from releasing the specified mfArrays. The `msFreeArgs()` is added at the end of the subroutine to enable the memory clearing action of MATFOR.

In conclusion, use the procedure pair `msInitArgs()` and `msFreeArgs()` to enclose program codes in your subroutine.

See Also

msRelease, msReturnArray

Display

msDisplay

Display mfArray data.

Module

```
use mod_ess
```

Syntax

```
call msDisplay(x[, name])
call msDisplay(x, name[, x1 , name1, ...])
```

Descriptions

Procedure `msDisplay` shows the content of mfArrays on a console window.

```
call msDisplay(x)
call msDisplay(x, name)
```

- Displays the content of mfArray `x`.
- Argument `name` is a character string specifying the accompanying output message. E.g. `call msDisplay(x, 'x')` prints 'x =' on the console window followed by the content of mfArray `x`. When argument `name` is not specified, `msDisplay` prints the caption 'ans =' followed by the content of mfArray.
- You can specify the number of display digits by using procedure `msFormat`.

```
call msDisplay(x, name)
call msDisplay(x, name, x1, name1,...)
```

- Display the content of the specified mfArrays with captions specified by the corresponding name strings. When multiple mfArrays are displayed, the arguments must be specified in pairs, i.e., one mfArray accompanied by one name string. For example, `call msDisplay(x, 'x', y, 'y', z, 'z')`.
- The number of mfArrays displayed in a single procedure call is limited to thirty-two.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x
integer :: i

x = (/(i, i=1, 5)/)

call msDisplay(x, 'x')
```

```
call msFreeArgs(x)
end program example
```

Result

```
x  =

  1  2  3  4  5
```

See Also

msFormat, msGDisplay

FileIO

mfLoad

Load binary file into mfArray.

Module

```
use fml
```

Syntax

Descriptions

Procedure `mfLoad` reads data from a binary file, created by procedure `msSave`, into an mfArray.

```
x = mfLoad(filename)
```

- Argument `filename` is a Fortran string containing the name of the binary file. For example, `x = mfLoad("y.mfb")`.
- The MATFOR binary file uses ".mfb" as a file extension.

Example

The following example uses function `mfLoad`.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, y

x = Reshape((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  x    is 1 3 5
!         2 4 6

call msSave('x.mfb', x)
y = mfLoad('x.mfb')
call msDisplay(x, 'x', y, 'y')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

  1  3  5
  2  4  6

 y  =

  1  3  5
  2  4  6
```

See Also

msSave, msSaveAscii, mfLoadAscii

mfLoad.m

Load MATFOR binary file into MATLAB workspace.

Module

`mfLoad.dll`

Syntax

```
x = mfLoad(filename)
```

Descriptions

Function `mfLoad` retrieves data in a MATFOR *.mfb binary file into a MATLAB matrix `x`.

`x = mfLoad(filename)`

- Argument `filename` is a string containing the name of the target binary file. An extension of .mfb is assumed if not specified.
- You can use this function to load data saved using `msSave` in MATFOR and msSave.m in MATLAB.

Note that to use this function, ensure that the files mfLoad.m and mfLoad.dll are in your MATLAB working directory. The two files are installed in your MATFOR directory <MATFOR>/common/tools/matlab when MATFOR is installed.

See Also

msSave, mfLoad, msSave.m

mfLoadAscii

Load ASCII file into mfArray.

Module

```
use fml
```

Syntax

Descriptions

Procedure `mfLoadAscii` reads in data from an ASCII file into an mfArray.

`x = mfLoadAscii(filename)`

- The ascii file must be in 2-D matrix format with m rows and n columns corresponding to an m x n matrix mfArray.
- Argument `filename` is a string containing the filename of the ASCII file. For example, `x = mfLoadAscii("y.txt")`.

You can use procedure `msSaveAscii` to produce an ASCII file for loading by using procedure `mfLoadAscii`.

Example

The following example uses the `mfSaveAscii` function.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, y

x = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  x    is  1 3 5
!           2 4 6

call msSaveAscii('x.txt', x)
y = mfLoadAscii('x.txt')
call msDisplay(x, 'x', y, 'y')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

  1  3  5
  2  4  6

 y  =

  1  3  5
```

```
2  4  6
```

See Also

msSave, mfLoad, msSaveAscii

msSave

Save mfArray as binary file.

Module

```
use fml
```

Syntax

```
call msSave(filename, x)
```

Descriptions

Procedure `msSave` writes data contained in an mfArray to a binary file with the extension 'mfb.'

`call msSave(filename, x)`

- Argument `filename` is a Fortran string containing the name of the target binary file. For example, `call msSave("x.mfb",x)`.
- Argument `x` is the name of the mfArray targeted for output.

Example

The following example uses the function `msSave`.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, y

x = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  x    is  1 3 5
!           2 4 6

call msSave('x.mfb', x)
y = mfLoad('x.mfb')
call msDisplay(x, 'x', y, 'y')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

  1  3  5
  2  4  6

 y  =

  1  3  5
  2  4  6
```

See Also

mfLoad, msSaveAscii, mfLoadAscii

mfSave.m

Save MATLAB matrix data as MATFOR binary file.

Module

`mfSave.dll`

Syntax

`mfSave(filename, x)`

Descriptions

Procedure `mfSave(filename, x)` saves data contained in MATLAB matrix `x` in MATFOR *.mfb binary file format.

- Argument `filename` is a string containing the name of the target binary file. An extension of .mfb is assumed if not specified.
- The data saved by using this function can be reloaded into MATLAB workspace using function `mfLoad` or retrieved in Fortran using procedure `mfLoad`. This facilitates exchange of data between Fortran and MATLAB environment.

Note that to use this function, ensure that the files mfSave.m and mfSave.dll are in your MATLAB working directory. The two files are installed in your MATFOR directory <MATFOR>/common/tools/matlab when MATFOR is installed.

See Also

msSave, mfLoad, mfLoad.m

msSaveAscii

Save mfArray as ASCII file.

Module

```
use fml
```

Syntax

```
call msSaveAscii(filename, x)
```

Descriptions

Procedure `msSaveAscii` writes data contained in an mfArray to an ASCII file with the extension ".dat".

`call msSaveAscii(filename, x)`

- The data is written in a 2-D matrix format, with m rows and n columns, corresponding to an m x n matrix mfArray.
- Argument `filename` is a string containing the name of the target binary file. For example, `msSaveAscii("x.dat", x)`.
- Argument `x` is the name of the mfArray targeted for output.

Example

The following example uses the `mfSaveAscii` function.

See Also

msSave, mfLoad, mfLoadAscii

CHAPTER 2

MOD_DATAFUN

mfMax, msMax

Find the maximum value.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, y, a, i
integer(4) :: IDIM

a = mfMax(x[, mf(), IDIM])
a = mfMax(x, y)

call msMax(mfOut(a, i), x[, mf(), IDIM])
```

Descriptions

`a = mfMax(x)`
For vectors, `mfMax` returns the largest element in `x`. For matrices, `mfMax` returns a row vector containing the maximum elements from each column.

`y = mfMax(x, mf(), IDIM)` operates along the dimension `IDIM`.

`a = mfMax(x, y)` returns an array the same size as `x` and `y` with the larger elements taken from `x` or `y`. Either one can be a scalar.

For complex number, its absolute value is used for comparison.

Note: Procedure msMax assumes the name of mfMax when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfMax` function.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, c, y
integer(4) :: i

x = Reshape((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  x    is  1 3 5
!           2 4 6

y = Reshape((/3,5,4,6,2,1/), (/2, 3/))
!  y    is  3 4 2
!           5 6 1
```

```
i = mfMax(mf((/ 1, 2, 3 /)))
call msDisplay(mf(i), 'mfMax(mf((/ 1, 2, 3 /)))')

c = mfMax(x, MF_NULL, 1)
call msDisplay(x, 'x', y, 'y', c,'mfMax(x, MF_NULL, 1)')
c = mfMax(x, MF_NULL, 2)
call msDisplay(x, 'x', y, 'y', c,'mfMax(x, MF_NULL, 2)')
c = mfMax(x, y)
call msDisplay(c, 'mfMax(x, y)')

call msFreeArgs(x, c, y)

end program example
```

Result

```
mfMax(mf((/ 1, 2, 3 /)))  =

  3

 x  =

  1  3  5
  2  4  6

 y  =

  3  4  2
  5  6  1

 mfMax(x, MF_NULL, 1)  =

  2  4  6

 x  =

  1  3  5
  2  4  6

 y  =

  3  4  2
  5  6  1

 mfMax(x, MF_NULL, 2)  =

  5
  6

 mfMax(x, y)  =

  3  4  5
  5  6  6
```

See Also

mfMin

mfMin, msMin

Find the minimum value.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, y, a, i
integer(4) :: IDIM

a = mfMin(x[, mf(), IDIM])
a = mfMin(x, y)

call msMin(mfOut(a, i), x[, mf(), IDIM])
```

Descriptions

`a = mfMin(x)`
For vectors, `mfMin` returns the largest element in `x`. For matrices, `mfMin` returns a row vector containing the minimum elements from each column.

`y = mfMin(x, mf(), IDIM)` operates along the dimension `IDIM`.

`a = mfMin(x, y)` returns an array the same size as `x` and `y` with the smaller elements taken from `x` or `y`. Either one can be a scalar.

For complex number, its absolute value is used for comparison.

Note: Procedure msMin assumes the name of mfMin when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfMin` function.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, c, y
integer(4) :: i

x = Reshape((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  x    is  1 3 5
!           2 4 6
y = Reshape((/3,5,4,6,2,1/), (/2, 3/))
!  y    is  3 4 2
!           5 6 1
```

```
i = mfMin(mf((/ 1, 2, 3 /)))
call msDisplay(mf(i), 'mfMin(mf((/ 1, 2, 3 /)))')

c = mfMin(x, MF_NULL, 1)
call msDisplay(x, 'x', y, 'y', c, 'mfMin(x, MF_NULL, 1)')
c = mfMin(x, MF_NULL, 2)
call msDisplay(c, 'mfMin(x, MF_NULL, 2)')
c = mfMin(x, y)
call msDisplay(c, 'mfMin(x,y) ')

call msFreeArgs(x, c, y)

end program example
```

Result

```
mfMin(mf((/ 1, 2, 3 /)))  =

  1

 x  =

  1  3  5
  2  4  6

 y  =

  3  4  2
  5  6  1

 mfMin(x, MF_NULL, 1)  =

  1  3  5

 mfMin(x, MF_NULL, 2)  =

  1
  2

 mfMin(x,y)   =

  1  3  2
  2  4  1
```

See Also

mfMax

mfProd, msProd

Product of elements.

Module

```
use fml
```

Syntax

```
a = mfProd(x[, IDIM])
```

Descriptions

`a = mfProd(x)`

- For matrices, `mfProd(x)` returns a row vector containing the product over each column of mfArray x.
- For N-dimension arrays, `mfProd(x)` operates on the first non-singleton dimension.

`a = mfProd(x, IDIM)`

- Argument `IDIM` is an integer(4) scalar specifying the dimension where procedure `mfProd(x)` works along.

Note: Procedure msProd assumes the name of mfProd when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfProd` function.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, y, c
integer(4) :: i

x = (/1, 2, 3/)
y = Reshape((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  x   is  1 3 5
!          2 4 6

i = mfProd(x)     ! returns 6
call msDisplay(x, 'x', mf(i), 'mfProd(x)')
c = mfProd(y, 1)   ! returns 2 12 30
call msDisplay(y, 'y', c, 'mfProd(y, 1)')

call msFreeArgs(x, y, c)

end program example
```

Result

```
x  =

  1  2  3

 mfProd(x)  =

  6

 y  =

  1  3  5
  2  4  6

 mfProd(y, 1)  =

   2  12  30
```

See Also

mfSum

mfSort, msSort

Sort in ascending order.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, y, i
integer:: IDIM

y=mfSort(x,[, IDIM])
call msSort(mfOut(y, i), x)
```

Descriptions

`y = mfSort(x), y=mfSort(x, IDIM)`

- The procedure returns mfArray `y` containing the elements of mfArray `x` sorted in ascending order along different dimensions.
- For vector `x`, the elements are sorted in ascending order.
- For matrix `x`, the elements in each column of `x` are sorted in ascending order accordingly. In effect, the procedure returns an mfArray `y` assuming the shape of `x`, with the elements in each column sorted in ascending order.
- For n-dimensional `x`, the elements are sorted along the first non-singleton dimension of `x`.
- Argument `IDIM` is an integer specifying the dimension to be sorted along. By default, `IDIM = 1`, corresponding to sorting along the column.
- When `x` is complex, the function returns mfArray y containing sorted elements of `mfAbs(x)`. Complex matches are further sorted according to `mfAtan2(mfImag(x), mfReal(x))`.

`call msSort(mfOut(y, i), x)`

- This format returns an additional index mfArray `i`, containing the indices of elements in `x`, corresponding to the sorted elements in `y`.

For example,

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

`call msSort(mfOut(y, i), x)` returns mfArray `y` and `i` containing,

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

i = 2   1   3
  3   2   1
  1   3   2
```

- For two values of the same magnitude, their original orders are preserved.

Note: Procedure msSort assumes the name of mfSort when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfSort` function.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, a

x = RESHAPE((/3,5,4,6,2,1/), (/2, 3/))
!  x    is  3 4 2
!           5 6 1

a = mfSort(x)
call msDisplay(x, 'x', a, 'mfSort(x)')
a = mfSort(x,2)
call msDisplay(x, 'x', a, 'mfSort(x,2)')
call msFreeArgs(x, a)

end program example
```

Result

```
x  =

  3  4  2
  5  6  1

 mfSort(x)  =

  3  4  1
  5  6  2

 x  =

  3  4  2
  5  6  1

 mfSort(x,2)  =

  2  3  4
  1  5  6
```

See Also

mfSortRows

mfSortRows, msSortRows

Sort rows in ascending order.

Module

```
use fml
```

Syntax

```
y = mfSortRows(x[, col])

call msSortRows(mfOut(y, i), x)
```

Descriptions

`y = mfSortRows(x), y = mfSortRows(x, col)`

- The procedure returns mfArray `y` containing the rows of matrix mfArray `x` sorted in ascending order as a group.
- If `x` contains strings, the function performs a dictionary sort.
- When complex `x`, the rows are sorted according to `mfAbs(x)`. Complex matches are further sorted according to `mfAtan2(mfImag(x), mfReal(x))`.
- If argument `col` is specified, the rows are sorted according to the columns specified in vector mfArray `col`. For example,

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

col = 3

y = mfSortRows(x, col) returns,

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

`call msSortRows(mfOut(y,i),x)`

- This format returns an additional index mfArray `i`, which contains the row indices after the corresponding rows are being swapped.

Note: Procedure msSortRows assumes the name of mfSortRows when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type(mfArray) :: x, y, i

x = mfMagic(5)

!Sort according to rows and retrieve index
call msSortRows(mfOut(y, i), x)
call msDisplay(x, 'x', y, 'Sort rows according to first column')
call msDisplay(i, 'Corresponding index')

call msFreeArgs(x, y, i)

end program example
```

Result

```
x  =

  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

 Sort rows according to first column  =

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

 Corresponding index  =

  1
  2
  3
  4
  5
```

See Also

mfSort

mfSum, msSum

Find the sum of elements.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, a
integer(4) :: IDIM

a=mfSum(x[, IDIM])
```

Descriptions

`a = mfSum(x), a = mfSum(x, IDIM)`
`a = mfSum(x)`
For vectors, the procedure returns the sum of the elements of `x`. For matrices, `a` is a row vector with the sum over each column. For N-Dimension arrays, `a` operates along the first non-singleton dimension.

`a = mfSum(X, IDIM)` sums along the dimension `IDIM`.

Note: Procedure msSum assumes the name of mfSum when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfSum` function.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, y, c
integer(4) :: I

x = (/1, 2, 3/)
y = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  x    is  1 3 5
!           2 4 6

i = mfSum(x)            ! returns 6
call msDisplay(x, 'x', mf(i), 'mfSum(x)')
c = mfSum(y, 1)         ! returns 3 7 11
call msDisplay(y, 'y', c, 'mfSum(y, 1)')
call msFreeArgs(x, y, c)

end program example
```

Result

```
x  =

  1  2  3

 mfSum(x)  =

  6

 y  =

  1  3  5
  2  4  6

 mfSum(y, 1)  =

   3   7  11
```

See Also

mfProd

CHAPTER 4

MOD_OPS

Module MOD_OPS contains a set of operators and functions for matrix and array manipulation.

This chapter describes the operators and functions available with topics as listed below:

Operator Precedence

Arithmetic Operators

Relational Operators

Special Operator Functions

Operator Precedence

The table below lists the MATFOR operator precedence. Operators sharing the same symbol as Fortran operators, share also the same precedence level. All MATFOR-defined operators occupy the lowest precedence level.

Note that operators having the same precedence level are evaluated from left to right.

Operators	Descriptions	Precedence
`.h.`	mfArray complex transpose	Highest
`.t.`	mfArray transpose	
`**`	mfArray power	
`*`	mfArray multiplication	
`/`	mfArray right division	
`+`	mfArray addition or unary plus	
`-`	mfArray subtraction or unary minus	
`>=`	mfArray greater than or equal to comparison	
`>`	mfArray greater than comparison	
`<=`	mfArray less than or equal to	
`<`	mfArray less than comparison	
`/=`	mfArray inequality comparison	
`==`	mfArray equality comparison	
`.hc.`	Horizontal concatenation	

`.vc.`	Vertical concatenation	Lowest

Arithmetic Operators

Arithmetic operators mean symbols of addition, subtraction, multiplication and division.

Module

```
use fml
or
use mod_ops
```

Syntax

```
type (mfArray) :: x,y,z
integer :: n

z = x + y
z = + x
z = x – y
z = - x
z = x*y
z = x/y
z = x**y
z = x**n
y = .t.x
z = mfMul(x, y)
z = mfLDiv(x, y)
z = mfRDiv(x, y)
z = .h.y
```

Descriptions

Arithmetic operators provided by MATFOR include both array and matrix operators. Array operators operate element-by-element on the mfArrays while matrix operators operate on whole mfArrays.

The following lists the operators available with a brief

description of their operation.

+ Array addition or unary plus

`x + y` adds mfArrays `x` and `y`. The two mfArrays must have the same size unless one is a scalar. A scalar can be added to mfArrays of any size.

- Array subtraction or unary minus

`x – y` subtracts mfArray `y` from `x`. The two mfArrays must have the same size unless one is a scalar. A scalar can be subtracted from mfArrays of any size.

* Array multiplication

`x * y` returns element-by-element the multiplication of mfArrays `x` and `y`. The two mfArrays must be in the same size unless one is a scalar. A scalar can be multiplied to an mfArray of any size.

mfMul Matrix multiplication

`mfMul(x, y)` returns matrix product of mfArrays x and y, where x is an m-by-p matrix and y is a p-by-n matrix. The product of the matrix multiplication is an m-by-n mfArray.

mfRDiv Matrix right division

`mfRDiv(x, y)` returns an mfArray approximately equal to `x*mfInv(y)`. The result is the same as `.t.( mfLDiv((.t.y), (.t.x)))`.

mfLDiv Matrix left division

`mfLDiv(y, x)` returns an mfArray approximately equal to `inv(y)*x`. Depending on the structure of `y`, MATFOR overloads several methods for solving the simultaneous linear equation.

**

Array power

`x**y` returns an mfArray containing elements of `x(i,j)` raised to the power of corresponding elements of `y(i,j)`. The two mfArrays must be in the same size, unless one is a scalar.

`x**n`, where n is a Fortran scalar, returns an mfArray containing element-by-elements of x raised to the power of n.

.t., mfTranspose

Array transpose

`.t.x` returns an mfArray containing the transpose of mfArray `x`. In a transpose operation, `aij` and `aji` are interchanged.

Complex elements are not conjugated.

You can use the `msTranspose` procedure to perform the same operation.

```
c = .t. a
c = mfTranspose(a)
call msTranspose(mfOut(c),a)
```

.h., mfCTranspose

Complex conjugate transpose

`.h.x` returns an mfArray containing the complex conjugate transpose of mfArray `x`. If `x` is real, this operation returns the same result as `.t.x`.

When performing linear algebra operations on complex matrices, it is almost always the complex conjugate

transpose (also called the Hermitian transpose) that is needed.

You can use the `mfCTranspose` procedure to perform the same operation.

```
c = .h. a
c =  mfCTaraspose(a)
call msCTranspose(mfOut(c),a)
```

See Also

`Relational Operators`

mfRDiv, mfLDiv

Matrix left divide and right divide operators.

Module

```
use fml
```

Syntax

```
type (mfArray) :: a, b, x

x = mfLDiv(a, b)
x = mfRDiv(b, a)
```

Descriptions

Function `mfLDiv` and `mfRDiv` are normally used in solving systems of linear equations represented by ax=b.

`x = mfLDiv(a, b)` is an approximation to `mfMul(mfInv(a), b))`.
`x = mfRDiv(b, a)` is an approximation to `mfMul(b, mfInv(a)`.

The two functions are related by

```
mfLdiv(a,                              b)                              =
mfTranspose(mfRDiv((mfTranspose(b)),(mfTranspose(a))))
```

Depending on the structure of the coefficient matrix `a`, MATFOR uses different algorithms to solve the simultaneous linear equations `mfLDiv(a, b)` and `mfRDiv(b, a)`. Figure 2.2 provides an overview of the different methods used for solving the linear equation, depending on the structure of matrix `a`.

Figure 2.2 Algorithms applicable for each type of matrix `a`.

If `a` is an n x n square matrix, and `b` is a n-by-p matrix, then `mfLDiv(a, b)` is solved by using Gaussian elimination. MATFOR performs a structural test on matrix `a` to select the optimum factorization method. Non-symmetry and non-positive definite systems are detected almost immediately, hence this does not take much of the computation time.

If `a` is an m-by-n rectangle matrix, and `b` is an m-by-p matrix, for m /= n, MATFOR uses the least square method for solving the under-determined or over-determined system. There are two approaches to solving a least square problem - QR and normal equations method. MATFOR uses the normal equations method as it requires half the arithmetic when m<n and much lesser storage space compared to the QR method.

Note that MATFOR uses LAPACK for solving Linear Algebra equations.

See Also

Arithmetic Operators

Relational Operators

Module

```
use mod_ops
```

Syntax

```
type (mfArray) :: x, y, z

z = x == y
z = x >= y
z = x > y
z = x <= y
z = x < y
z = x /= y
```

Descriptions

The relational operators ==, >=, >, <=, and /=, perform element-by-element comparisons between two mfArrays.

The returned logical mfArray records the status of the comparison. Each element records a logical `true(1)` if the relationship is true, and logical `false(0)` otherwise.

Note that the two mfArrays `x,y` must be in the same size, unless one is a scalar. The scalar will be expanded to an mfArray the same size as the other mfArray.

== mfArray equality comparison

`x == y` checks if each element of `x` equals the corresponding element of `y`. The operator tests in both real and imaginary parts of the elements.

When you use the equality (= =) operator in an `if` statement, use the `all` function to enclose it. The `all` function ensures that the equality condition is satisfied by all elements of the returned mfArray.

>= mfArray greater than or equal to comparison

`x >= y` checks if each element of `x` is more than or equal to the corresponding element of `y`. The operator compares only the real part of the elements.

> mfArray greater than comparison

`x > y` checks if each element of `x` is greater than the corresponding element of `y`. The operator compares only the real part of the elements.

<= mfArray less than or equal to comparison

`x <= y` checks if each element of `x` is less than or equal to the corresponding element of `y`. The function returns 1 if the relation is true and 0 otherwise. The operator compares only the real part of the elements.

< mfArray less than comparison

`x < y` checks if each element of `x` is less than the corresponding element of `y`. The function returns 1 if the relation is true and 0 otherwise. The operator compares only the real part of the elements.

/= mfArray inequality comparison

`x /= y` checks if each element of `x` for inequality to the corresponding element of `y`. The operator tests in both real and imaginary parts of the elements.

See Also

`Arithmetic Operators`, `Any`, `All`

Special Operator Functions

mfColon, msColon

Pick out selected rows, columns and elements of vector, matrices and mfArrays.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x
integer :: start, step, end

x = mfColon(start[, step][, end])
```

Descriptions

Procedure `mfColon` constructs a regularly spaced vector mfArray. The input arguments, `start`, `step` and `end` can be integers or real. For complex inputs, imaginary parts are ignored.

`x = mfColon(a, b, c)`

- Vector mfArray `x` is constructed with elements [a, a+b, ..., a+b*m, ...,c] where `m = mfFix((c-a)/b`.
- The procedure returns empty when `b>0, a>c`, or when `b<0, a<c`.

`x = mfColon(a, c)`

- Vector mfArray `x` is constructed with elements [a, a+1, ...,c].
- It returns empty if `a>c`.

Note: Procedure msColon assumes the name of mfColon when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example constructs an mfArray x by using mfColon.

Code

```
program example

use fml
implicit none

type (mfArray) :: x

x = mfColon(1d0, 0.5d0, 2d0)
call msDisplay (x, 'x')

call msFreeArgs(x)

end program example
```

Result

```
x  =

   1.0000   1.5000   2.0000
```

CHAPTER 5

MOD_ELFUN

Module `mod_elfun` contains a set of elementary math functions including trigonometric, exponential, complex, rounding and remainder.

The functions are listed below:

Trigonometric	
`mfACos`	Inverse cosine function
`mfACosh`	Inverse hyperbolic cosine function
`mfACot`	Inverse cotangent function
`mfACoth`	Inverse hyperbolic cotangent function
`mfACsc`	Inverse cosecant function
`mfACsch`	Inverse hyperbolic cosecant function
`mfASec`	Inverse secant function.
`mfASech`	Inverse hyperbolic secant function.
`mfASin`	Inverse sine function.
`mfASinh`	Inverse hyperbolic sine function.
`mfATan`	Inverse tangent function.
`mfATan2`	Four quadrant arctangent function
`mfATanh`	Inverse hyperbolic tangent function
`mfCos`	Cosine function
`mfCosh`	Hyperbolic cosine function
`mfCot`	Cotangent function
`mfCoth`	Hyperbolic cotangent function
`mfCsc`	Cosecant function
`mfCsch`	Hyperbolic cosecant function

mfSec	Secant function
mfSech	Hyperbolic secant function
mfSin	Sine function
mfSinh	Hyperbolic sine function
mfTan	Tangent function
mfTanh	Hyperbolic tangent function

Exponential

mfExp	Exponential function
mfLog	Natural logarithm
mfLog10	Common logarithm (base 10)
mfLog2	Base 2 Logarithmic and floating point dissection
mfPow2	Base 2 power and floating point number scaling
mfSqrt	Square Root function

Complex

mfAbs	Absolute of real and complex value
mfAngle	Phase angle of complex
mfComplex	Convert input number to complex
mfConj	Conjugate of complex
mfImag	Imaginary part of complex
mfReal	Real part of complex

Rounding and Remainder

mfCeil	Round towards positive infinity
mfFix	Round towards zero
mfFloor	Round towards minus infinity
mfMod	Modulus (signed remainder after division)
mfRem	Remainder after division
mfRound	Round towards nearest integer
mfSign	Signum function

Trigonometry

mfACos, msACos

Request inverse cosine function.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, y

y = mfACos(x)
```

Descriptions

Procedure `mfACos(x)` returns the arccosine of the mfArray `x`, in radians, where

$\cos^{-1}(x) = -i*\log[x + i*(1-x^2)^{1/2}]$
The function domain and ranges include real and complex data.
For |x| <= 1, the function result is real and lies in the range 0 to π.
For |x| > 1, the function returns a complex value.

Note: Procedure msACos assumes the name of mfACos when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 2.02d0
y = mfACos(x)

call msDisplay(x, 'x', y, 'mfACos(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   2.0200

 mfACos(x)  =

 0.0000 +1.3284i
```

See Also

mfACosh, mfCos

mfACosh, msACosh

Request inverse hyperbolic cosine function.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, y

y = mfACosh(x)
```

Descriptions

Procedure `mfACosh(x)` returns element-by-element the inverse hyperbolic cosine of the mfArray `x`, in radians, where

$\cosh^{-1}(x) = \log[x + (x^2 - 1)^{1/2}]$

The function domain and ranges include real and complex data.

Note: Procedure msACosh assumes the name of mfACosh when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = (2, -2)
y = mfACosh(x)

call msDisplay(x, 'x', y, 'mfACosh(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

 2.0000 -2.0000i

 mfACosh(x)  =

 1.7343 -0.8165i
```

See Also

mfCos, mfCosh

mfACot, msACot

Request inverse cotangent function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfACot(x)
```

Descriptions

Procedure `mfACot(x)` returns element-by-element the arccotangent of the mfArray `x`, in radians, where

$\cot^{-1}(x) = \tan^{-1}(1/x)$

The function domain and range includes real and complex data.

Note: Procedure msACot assumes the name of mfACot when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = (-5, -5)
y = mfACot(x)

call msDisplay(x, 'x', y, 'mfACot(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

-5.0000 -5.0000i

 mfACot(x)  =

-0.1007 +0.0993i
```

See Also

mfACoth, mfCot, mfCoth

mfACoth, msACoth

Request inverse hyperbolic cotangent function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfACoth(x)
```

Descriptions

Procedure `mfACoth(x)` returns element-by-element the inverse hyperbolic cotangent of the mfArray `x` in radians, where

$\coth^{-1}(x) = \tanh^{-1}(1/x)$

The function domain and range includes real and complex data. For |x| <= 1, the function result is complex or infinity.

Note: Procedure msACoth assumes the name of mfACoth when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.5d0
y = mfACoth(x)

call msDisplay(x, 'x', y, 'mfACoth(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.5000

 mfACoth(x)  =

 0.5493 -1.5708i
```

See Also

mfACot, mfCot, mfCoth

mfACsc, msACsc

Request inverse cosecant function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfACsc(x)
```

Descriptions

Procedure `mfACsc(x)` returns element-by-element the inverse cosecant of the mfArray `x` in radians, where

$\csc^{-1}(x) = \sin^{-1}(1/x)$

The function domain and range includes real and complex data.

For $|x| < 1$, the function result is complex or NaN.

Note: Procedure msACsc assumes the name of mfACsc when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 3.0d0
y = mfACsc(x)

call msDisplay(x, 'x', y, 'mfACsc(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

  3

 mfACsc(x)  =

   0.3398
```

See Also

mfACsch, mfCsc, mfCsch

mfACsch, msACsch

Request inverse hyperbolic cosecant function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfACsch(x)
```

Descriptions

Procedure `mfACsch(x)` returns element-by-element the inverse hyperbolic cosecant of the mfArray `x` in radians, where

$csch^{-1}(x) = sinh^{-1}(1/x)$
The function domain and range includes real and complex data.
For x = 0, the function result is infinity.

Note: Procedure msACsch assumes the name of mfACsch when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.5d0
y = mfACsch(x)

call msDisplay(x, 'x', y, 'mfACsch(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.5000

 mfACsch(x)  =

   1.4436
```

See Also

mfACsc, mfCsc, mfCsch

mfASec, msASec

Request inverse secant function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfASec(x)
```

Descriptions

Procedure `mfASec(x)` returns element-by-element the inverse secant of the mfArray `x` in radians, where

$\sec^{-1}(x) = \cos^{-1}(1/x)$
The function domain and range includes real and complex data.
For $|x| < 1$, the function result is complex or NaN.

Note: Procedure msASec assumes the name of mfASec when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.1d0
y = mfASec(x)

call msDisplay(x, 'x', y, 'mfASec(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.1000

 mfASec(x)  =

 0.0000 +2.9932i
```

See Also

mfASech, mfSec, mfSech

mfASech, msASech

Request inverse hyperbolic secant function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfASech(x)
```

Descriptions

Procedure `mfASech(x)` returns element-by-element the inverse hyperbolic secant of the mfArray `x` in radians, where

$sech^{-1}(x) = cosh^{-1}(1/x)$

The function domain and range includes real and complex data.

Note: Procedure msASech assumes the name of mfASech when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.1d0
y = mfASech(x)

call msDisplay(x, 'x', y, 'mfASech(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.1000

 mfASech(x)  =

   2.9932
```

See Also

mfASec, mfSec, mfSech

mfASin, msASin

Request inverse sine function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfASin(x)
```

Descriptions

Procedure `mfASin(x)` returns element-by-element the arcsine of the mfArray `x` in radians, where

$\sin^{-1}(x) = -i*\log[i*x+(1-x^2)^{1/2}]$
The function domain and range includes real and complex data.
For real $|x| < 1$, the function result is real and lies in the range $[-\pi/2, \pi/2]$.
For real $|x| > 1$, the function returns a complex value.

Note: Procedure msASin assumes the name of mfASin when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.5d0
y = mfASin(x)

call msDisplay(x, 'x', y, 'mfASin(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.5000

 mfASin(x)  =
```

```
0.5236
```

See Also

mfASinh, mfSin, mfSinh

mfASinh, msASinh

Request inverse hyperbolic sine function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfASinh(x)
```

Descriptions

Procedure `mfASinh(x)` returns element-by-element the inverse hyperbolic sine of the mfArray `x` in radians, where

$\sinh^{-1}(x) = \log[x + (x^2+1)^{1/2}]$

The function domain and range includes real and complex data.

Note: Procedure msASinh assumes the name of mfASinh when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = (2.d0, -2.d0)
y = mfASinh(x)

call msDisplay(x, 'x', y, 'mfASinh(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

 2.0000 -2.0000i

 mfASinh(x)  =

 1.7343 -0.7542i
```

See Also

mfASin, mfSin, mfSinh

mfATan, msATan

Request inverse tangent function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfATan(x)
```

Descriptions

Procedure `mfATan(x)` returns element-by-element the inverse tangent of the mfArray `x` in radians, where

$\tan^{-1}(x) = (i/2)* \log((i+x)/(i-x))$
The function domain and range include real and complex data.
For real x, the result lies in the range $[-\pi/2, \pi/2]$.

Note: Procedure msATan assumes the name of mfATan when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.154d0
y = mfATan(x)

call msDisplay(x, 'x', y, 'mfATan(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.1540

 mfATan(x)  =

   0.1528
```

See Also

mfATan2, mfATanh, mfTan, mfTanh

mfATan2, msATan2

Request four quadrant arctangent function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y, z

z = mfATan2(y, x)
```

Descriptions

Procedure `mfATan2(y, x)` returns an mfArray `z`, containing element-by-element the principal value of the complex number defined by mfArrays `y` and `x`, where `y` is the imaginary part and `x` is the real part.
For x approaching to 0, the function result is approximately mfATan(y/x). However, in contrast to mfATan(y/x), which is limited to the range -π/2 <= mfATan(y/x) <= π/2, the function result lies in the range of - mfATan2(y,x) in four quadrant.

Note: Procedure msATan2 assumes the name of mfATan2 when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.5d0
y = mfATan2(x, x)

call msDisplay(x, 'x', y, 'mfATan2(x, x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =


   0.5000

 mfATan2(x, x)  =

   0.7854
```

See Also

mfATan, mfATanh, mfTan, mfTanh

mfATanh, msATanh

Request inverse hyperbolic tangent function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfATanh(x)
```

Descriptions

Procedure `mfATanh(x)` returns element-by-element the inverse hyperbolic tangent of the mfArray `x` in radians, where

$\tan^{-1}(x) = (1/2)* \log((1+x)/(1-x))$
The function domain and range include real and complex data.

Note: Procedure msATanh assumes the name of mfATanh when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 1.5782d0
y = mfATanh(x)

call msDisplay(x, 'x', y, 'mfATanh(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   1.5782

 mfATanh(x)  =

 0.7475 -1.5708i
```

See Also

mfATan, mfATan2, mfTan, mfTanh

mfCos, msCos

Request cosine function.

Module

```
use mod_elfun
```

Syntax

```
type (mfArray) :: x, y

y = mfCos(x)
```

Descriptions

Procedure `mfCos(x)` returns element-by-element the circular cosine of mfArray `x`, where `x` is in radians.

The function domain and range include real and complex data.

Note: Procedure msCos assumes the name of mfCos when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = (3.14d0, -3.14d0)
y = mfCos(x)

call msDisplay(x, 'x', y, 'mfCos(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

 3.1400 -3.1400i

 mfCos(x)  =

-11.5736  +0.0184i
```

See Also

mfACos, mfACosh, mfCosh

mfCosh, msCosh

Request hyperbolic cosine function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfCosh(x)
```

Descriptions

Procedure `mfCosh(x)` returns element-by-element the hyperbolic cosine of mfArray `x` where `x` is in radians.

The function domain and range include real and complex data.

Note: Procedure msCosh assumes the name of mfCosh when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.1d0
y = mfCosh(x)

call msDisplay(x, 'x', y, 'mfCosh(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.1000

 mfCosh(x)  =

   1.0050
```

See Also

mfACos, mfACosh, mfCos

mfCot, msCot

Request cotangent function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfCot(x)
```

Descriptions

Procedure `mfCot(x)` returns element-by-element the cotangent of mfArray `x` in radians where

mfCot(x) = 1/mfTan(x).
The function domain and range include real and complex data.

Note: Procedure msCot assumes the name of mfCot when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.7d0
y = mfCot(x)

call msDisplay(x, 'x', y, 'mfCot(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.7000

 mfCot(x)  =

   1.1872
```

See Also

mfCoth, mfACot, mfACoth

mfCoth, msCoth

Request hyperbolic cotangent function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfCoth(x)
```

Descriptions

Procedure `mfCoth(x)` returns element-by-element the hyperbolic cotangent of the mfArray `x` in radians, where

mfCoth(x) = 1/mfTanh(x).
The function domain and range include real and complex data.

Note: Procedure msCoth assumes the name of mfCoth when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.5d0
y = mfCoth(x)

call msDisplay(x, 'x', y, 'mfCoth(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.5000

 mfCoth(x)  =

   2.1640
```

See Also

mfCot, mfACot, mfACoth

mfCsc, msCsc

Request cosecant function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfCsc(x)
```

Descriptions

Procedure `mfCsc(x)` returns the element-by-element cosecant of mfArray `x`, where

mfCsc(x) = 1/mfSin(x)
The function domain and range include real and complex data. All angles are in radians.

Note: Procedure msCsc assumes the name of mfCsc when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 2.0d0
y = mfCsc(x)

call msDisplay(x, 'x', y, 'mfCsc(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

  2

 mfCsc(x)  =

   1.0998
```

See Also

mfCsch, mfACsc, mfACsch

mfCsch, msCsch

Request hyperbolic cosecant function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfCsch(x)
```

Descriptions

Procedure `mfCsch(x)` returns element-by-element the hyperbolic cosecant of mfArray `x`, where

mfCsch(x) = 1/mfSinh(x)
The function domain and range include real and complex data. All angles are in radians.

Note: Procedure msCsch assumes the name of mfCsch when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 0.5d0
y = mfCsch(x)

call msDisplay(x, 'x', y, 'mfCsch(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.5000

 mfCsch(x)  =

   1.9190
```

See Also

mfCsc, mfACsc, mfACsch

mfSec, msSec

Request secant function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfSec(x)
```

Descriptions

Procedure `mfSec(x)` returns element-by-element the secant of mfArray `x`, where

mfSec(x) = 1/mfCos(x).
The function domain and range include real and complex data. All angles are in radians.

Note: Procedure msSec assumes the name of mfSec when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = MF_PI
y = mfSec(x)

call msDisplay(x, 'x', y, 'mfSec(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   3.1416

 mfSec(x)  =

 -1
```

See Also

mfSech, mfASec, mfASech

mfSech, msSech

Request hyperbolic secant function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfSech(x)
```

Descriptions

Procedure `mfSech(x)` returns element-by-element the hyperbolic secant of mfArray `x`, where

mfSech(x) = 1/mfCosh(x).
The function domain and range include real and complex data. All angles are in radians.

Note: Procedure msSech assumes the name of mfSech when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = (-1, 4)
y = mfSech(x)

call msDisplay(x, 'x', y, 'mfSech(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

-1.0000 +4.0000i

 mfSech(x)  =

-0.5578 -0.4918i
```

See Also

mfSec, mfASec, mfASech

mfSin, msSin

Request sine function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfSin(x)
```

Descriptions

Procedure `mfSin(x)` returns element-by-element the circular sine of mfArray `x`.
The function domain and range include real and complex data. All angles are in radians.

Note: Procedure msSin assumes the name of mfSin when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use mod_ess
use mod_elfun
implicit none

type (mfArray) :: x, y

x = 0.5236d0
y = mfSin(x)

call msDisplay(x, 'x', y, 'mfSin(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   0.5236

 mfSin(x)  =

   0.5000
```

See Also

mfSinh, mfASin, mfASinh

mfSinh, msSinh

Request hyperbolic sine function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfSinh(x)
```

Descriptions

Procedure `mfSinh(x)` returns element-by-element the hyperbolic sine of mfArray `x` where `x` is in radians.
The function domain and range include real and complex data.

Note: Procedure msSinh assumes the name of mfSinh when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 3
y = mfSinh(x)

call msDisplay(x, 'x', y, 'mfSinh(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

  3

mfSinh(x)  =

   10.0179
```

See Also

mfSin, mfASin, mfASinh

mfTan, msTan

Request tangent function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfTan(x)
```

Descriptions

Procedure `mfTan(x)` returns element-by-element the tangent of mfArray `x`.
The function domain and range include real and complex data. All angles are in radians.
mfTan(x) = mfSin(x)/mfCos(x)

Note: Procedure msTan assumes the name of mfTan when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 1.0d0
y = mfTan(x)

call msDisplay(x, 'x', y, 'mfTan(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

  1

mfTan(x)  =

   1.5574
```

See Also

mfTanh, mfATan, mfATanh

mfTanh, msTanh

Request hyperbolic tangent function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfTanh(x)
```

Descriptions

Procedure `mfTanh(x)` returns element-by-element the hyperbolic tangent of mfArray `x`, where

mfTanh(x) = mfSinh(x)/mfCosh(x)
The function domain and range include real and complex data. All angles are in radians.

Note: Procedure msTanh assumes the name of mfTanh when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 1
y = mfTanh(x)

call msDisplay(x, 'x', y, 'mfTanh(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

  1

 mfTanh(x)  =

   0.7616
```

See Also

mfTan, mfATan, mfATanh

Exponential

mfExp, msExp

Request exponential function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfExp(x)
```

Descriptions

Procedure `mfExp(x)` returns element-by-element the exponential of mfArray `x`. The function domain and range include real and complex data.

Note: Procedure msExp assumes the name of mfExp when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = (2.d0, 3.d0)
y = mfExp(x)

call msDisplay(x, 'x', y, 'mfExp(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

 2.0000 +3.0000i

 mfExp(x)  =

-7.3151 +1.0427i
```

See Also

mfLog, mfLog10

mfLog, msLog

Request natural logarithm.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfLog(x)
```

Descriptions

Procedure `mfLog(x)` returns element-by-element the natural logarithm of mfArray `x`. The function domain and range include real and complex data.

Note: Procedure msLog assumes the name of mfLog when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = MF_E
y = mfLog(x)

call msDisplay(x, 'x', y, 'mfLog(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

   2.7183

 mfLog(x)  =

  1
```

See Also

mfExp, mfLog10, mfLog2

mfLog10, msLog10

Common logarithm (base 10).

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfLog10(x)
```

Descriptions

Procedure `mfLog10(x)` returns element-by-element the natural logarithm of mfArray `x`. The function domain and range include real and complex data.

Note: Procedure msLog10 assumes the name of mfLog10 when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = 5.d0
y = mfLog10(x)

call msDisplay(x, 'x', y, 'mfLog10(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

  5

 mfLog10(x)  =

   0.6990
```

See Also

mfExp, mfLog, mfLog2

mfLog2, msLog2

Base 2 logarithm and floating-point dissection.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y, f, e

y = mfLog2(x)

call msLog2(mfOut(f, e), x)
```

Descriptions

Procedure `mfLog2` computes base 2 logarithm or extract mantissa and exponent of a floating-point number.

`y = mfLog2(x)` returns the elemental base 2 logarithm of mfArray `x`.

`call msLog2(mfOut(f,e),x)` dissects each element of mfArray `x` into the binary floating-point format consisting of a mantissa, mfArray `f`, and an exponent, mfArray `e`, where x = f*(2**e) for real `x`. MfArray `f` contains real values lying in the range of 0.5 <= mfAbs(f) < 1. For elements of x=0, the corresponding elements of `f` and `e` are equal to zero.

Note: Procedure msLog2 assumes the name of mfLog2 when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below evaluates the `mfLog2` of mfArray `x`.

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y, f, e

! MF_PI is MATFOR intrinsic parameter for pi.

x = (/MF_PI, 2* MF_PI, 3* MF_PI/)
y = mfLog2(x)

call msLog2(mfout(f, e), x)
call msDisplay(y, 'mfLog2(x)' )
call msDisplay(f, 'the mantissa', e, 'the exponent')
call msFreeArgs(x, y, f, e)

end program example
```

Result

```
mfLog2(x)  =

   1.6515   2.6515   3.2365

 the mantissa  =

   0.7854   0.7854   0.5890

 the exponent  =

  2  3  4
```

See Also

mfLog, mfPow2, mfLog10

mfPow2, msPow2

Base 2 power and floating point number scaling.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y, f, e

y = mfPow2(x)
y = mfPow2(f,e)
```

Descriptions

Procedure `mfPow2` computes base 2 power or floating point number scaling.

`y = mfPow2(x)` returns an mfArray `y` with elements computed from two raised to the power of each element of mfArray `x`, i.e., y = 2**x.

`y = mfPow2(f, e)` returns the mfArray `y` containing elements computed from y = f*(2**e). The result is equivalent to scaling each element of `f` by exponent `e` or adding each element of `e` to the corresponding floating-point exponent of `e`.

Note: Procedure msPow2 assumes the name of mfPow2 when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below evaluates the `mfPow2` of mfArray `x`, a 1-by-10 vector.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, y, z, f, e

! Initialize the mfArrays
x = 3
f = (/1.0, 0.5, -0.75/)
e = (/2, 3, 5/)

! Compute base 2 power y = 2**x
y = mfPow2(x)

! Perform floating point scaling z = f*(2**e)
z = mfPow2(f, e)

! Display the values
call msDisplay(x, 'x', y, 'mfPow2(x)')
call msDisplay(f, 'f', e, 'e', z, 'mfPow2(f, e)')
```

```
! Deallocate mfArrays
call msFreeArgs(x, y, z, f, e)

end program example
```

Result

```
x  =

  3

 mfPow2(x)  =

  8

 f  =

   1.0000   0.5000  -0.7500

 e  =

  2  3  5

 mfPow2(f, e)  =

   4   4 -24
```

See Also

mfLog2, mfExp

mfSqrt, msSqrt

Square root function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfSqrt(x)
```

Descriptions

Procedure `mfSqrt(x)` returns element-by-element the square root of mfArray `x`. The procedure domain includes real and complex data.
For negative and complex elements of `x`, complex results are returned.

Note: Procedure msSqrt assumes the name of mfSqrt when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below evaluates `mfSqrt(-4)`.

Code

```
program example

use fml
implicit none

type (mfArray):: x, y

x = -4
y = mfSqrt(x)

call msDisplay(x, 'x', y, 'mfSqrt(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

 -4

 mfSqrt(x)  =

 0.0000 +2.0000i
```

See Also

mfExp, mfLog, mfLog2, mfLog10

Complex

mfAbs, msAbs

Absolute of real and complex value.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfAbs(x)
```

Descriptions

Procedure `mfAbs(x)` returns element-by-element the absolute of mfArray `x`.

For real `x`, `mfAbs(x)` returns |x|.

For complex z = x + iy, `mfAbs(z)` returns the magnitude of the complex computed as sqrt(x**2+y**2).

Note: Procedure msAbs assumes the name of mfAbs when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below evaluates the absolute of complex (2, -2)

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = (2, -2)
y = mfAbs(x)

call msDisplay(x, 'x', y,'mfAbs(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

 2.0000 -2.0000i

 mfAbs(x)  =
```

```
2.8284
```

See Also

mfAngle, mfSign

mfAngle, msAngle

Phase angle of complex.

Module

```
use fml
```

Syntax

```
type (mfArray) :: z, p

p = mfAngle(z)
```

Descriptions

Procedure `mfAngle(z)` returns an mfArray `p` containing the element-by-element phase angle of complex mfArray `z`.

Note: Procedure msAngle assumes the name of mfAngle when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below evaluates the phase angle of complex mfArray `z` over a range of value.

Code

```
program example

use fml
implicit none

type(mfArray) :: theta, z, x, y

x = mfColon(0, 5, 10)
y = mfColon(-2, 5, 8)
z = mfComplex(x, y)
theta = mfAngle(z)

call msDisplay(z, 'z', theta, 'mfAngle(z)')
call msFreeArgs(theta, z, x, y)

end program example
```

Result

```
z  =

column   1  to   3

  0.0000  -2.0000i    5.0000  +3.0000i   10.0000  +8.0000i

 mfAngle(z)  =

  -1.5708   0.5404   0.6747
```

See Also

mfAbs

mfComplex, msComplex

Convert input numbers to complex.

Module

```
use fml
```

Syntax

```
type (mfArray) :: z, x, y

z = mfComplex(x, y)
z = mfComplex(x)
```

Descriptions

Procedure `mfComplex(x, y)` returns a complex mfArray whose real part is specified by the elements of mfArray `x` and imaginary part is specified by elements of mfArray `y` ie z = x + yi. `z = mfComplex(x)` returns a complex mfArray whose real part is specified by the elements of mfArray `x` and imaginary part is zero ie z = x + 0i.

Note: Procedure msComplex assumes the name of mfComplex when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below constructs a complex mfArray `z` from two real mfArrays `x` and `y`.

Code

```
program example

use fml
implicit none

type (mfArray) :: z, x, y

x = (/-5, 7, 0/)
y = (/ 3, 1, 2/)
z = mfComplex(x, y)

call msDisplay(x, 'x', y, 'y', z, 'mfComplex(x, y)')
call msFreeArgs(x, y, z)

end program example
```

Result

```
x  =

 -5  7  0

 y  =

  3  1  2

 mfComplex(x, y)  =
```

```
-5.0000 +3.0000i   7.0000 +1.0000i   0.0000 +2.0000i
```

See Also

mfImag, mfReal

mfConj, msConj

Conjugate of complex.

Module

```
use fml
```

Syntax

```
type (mfArray) :: z, c

c = mfConj(z)
```

Descriptions

Procedure `mfConj(z)` returns an mfArray containing the element-by-element conjugate of complex mfArray `z`.

Note: Procedure msConj assumes the name of mfConj when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below evaluates the conjugate of complex mfArray `z`.

Code

```
program example

use fml
implicit none

type (mfArray) :: c, z

z = (-2, 2)
c = mfConj(z)

call msDisplay(z, 'z', c, 'mfConj(z)')
call msFreeArgs(c, z)

end program example
```

Result

```
z  =

-2.0000 +2.0000i

 mfConj(z)  =

-2.0000 -2.0000i
```

See Also

mfImag, mfReal

mfImag, msImag

Imaginary part of complex.

Module

```
use fml
```

Syntax

```
type (mfArray) :: z, y

y = mfImag(z)
```

Descriptions

Procedure `mfImag(z)` returns element-by-element the imaginary part of complex mfArray `z`.

Note: Procedure msImag assumes the name of mfImag when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below gets the imaginary part of complex mfArray `z`.

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = (2, -2)
y = mfImag(x)

call msDisplay(x, 'x', y, 'mfImag(x)')
call msFreeArgs(x, y)

end program example
```

Result

```
x  =

 2.0000 -2.0000i

 mfImag(x)  =

 -2
```

See Also

mfReal, mfConj

mfReal, msReal

Real part of complex.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, z

x = mfReal(z)
```

Descriptions

Procedure `mfReal(z)` returns the elemental real part of complex mfArray `z`.

Note: Procedure msReal assumes the name of mfReal when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below gets the real part of complex mfArray `z`.

Code

```
program example

use fml
implicit none

type (mfArray) :: z, x

z = (-2, 2)
x = mfReal(z)

call msDisplay(z, 'z', x, 'mfReal(z)' )
call msFreeArgs(z, x)

end program example
```

Result

```
z  =

-2.0000 +2.0000i

 mfReal(z)  =

 -2
```

See Also

mfImag, mfConj

Rounding and Remainder

mfCeil, msCeil

Round towards positive infinity.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfCeil(x)
```

Descriptions

Procedure `mfCeil(x)` returns an mfArray containing elements of mfArray `x` rounded to the nearest integer `x` towards infinity. The real and imaginary parts of a complex number are rounded independently.

Note: Procedure msCeil assumes the name of mfCeil when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below performs the `mfCeil` operation on mfArray `x`.

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y, u, v

x = (/-1.6, 2.3/)
u = (3.2, 2.2)
y = mfCeil(x)
v = mfCeil(u)

call msDisplay(x, 'x', y, 'mfCeil(x)')
call msDisplay(u, 'u', v, 'mfCeil(u)')
call msFreeArgs(x, y, u, v)

end program example
```

Result

```
x  =

  -1.6000   2.3000

 mfCeil(x)  =

 -1  3

 u  =
```

```
 3.2000 +2.2000i

 mfCeil(u)  =

 4.0000 +3.0000i
```

See Also

mfFix, mfFloor, mfRound

mfFix, msFix

Round the elements towards zero.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfFix(x)
```

Descriptions

Procedure `mfFix(x)` rounds the elements of mfArray `x` towards zero, returning an mfArray of integers. The real and imaginary parts of a complex number are rounded towards the nearest integer independently.

Note: Procedure msFix assumes the name of mfFix when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below performs the `mfFix` operation on mfArray `x`.

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y, u, v

x = (/-1.6, 2.3/)
y = mfFix(x)
u = dcmplx(3.2, 2.2)
v = mfFix(u)

call msDisplay(x, 'x', y, 'mfFix(x)')
call msDisplay(u, 'u', v, 'mfFix(u)')
call msFreeArgs(x, y, u, v)

end program example
```

Result

```
x  =

  -1.6000   2.3000

 mfFix(x)  =

 -1  2

 u  =
```

```
 3.2000 +2.2000i

 mfFix(u)  =

 3.0000 +2.0000i
```

See Also

mfCeil, mfFloor, mfRound

mfFloor, msFloor

Round towards minus infinity.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfFloor(x)
```

Descriptions

Procedure `mfFloor(x)` rounds the elements of mfArray `x` towards nearest integer less than or equal to `x` and towards minus infinity. The real part and imaginary part of a complex number are rounded towards the nearest integer independently.

Note: Procedure msFloor assumes the name of mfFloor when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below performs the `mfFloor` operation on mfArray `x`.

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y, u, v

x = (/-1.6, 2.3/)
u = (3.2, 2.2)
y = mfFloor(x)
v = mfFloor(u)

call msDisplay(x, 'x', y, 'mfFloor(x)')
call msDisplay(u, 'u', v, 'mfFloor(u)')
call msFreeArgs(x, y, u, v)

end program example
```

Result

```
x  =

  -1.6000   2.3000

 mfFloor(x)  =

 -2  2

 u  =
```

```
 3.2000 +2.2000i

 mfFloor(u)  =

 3.0000 +2.0000i
```

See Also

mfCeil, mfFix, mfRound

mfMod, msMod

Return modulus (signed remainder after division).

Module

```
use fml
```

Syntax

```
type (mfArray) :: m, x, y

m = mfMod(x, y)
```

Descriptions

Procedure `mfMod(x, y)` returns an mfArray containing element-by-element the signed remainder of `x`, `y` division.
The procedure uses the following algorithm:

$y \neq 0$, mfMod(x, y) = x - y*mfFloor(x/y)
$y = 0$, mfMod(x, y) = x

Note that :

- The shape of the input arguments `x` and `y` must conform.
- `mfMod(x, y)` always differ from `x` by a multiple of `y`.
- `mfMod(x, y)` has the same sign as `y` while `mfRem(x,y)` has the same sign as `x`.
- `mfMod(x, y)` and `mfRem(x, y)` are equal if `x` and `y` are of the same sign. They differ if the sign of `x` and `y` are different.

Limitations

Arguments `x` and `y` should be integers. Due to the inexact representation of floating-point numbers on a computer, real (or complex) inputs may lead to unexpected results.

Note: Procedure msMod assumes the name of mfMod when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below finds the modulus of `x` and `y`.

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y, m

x = (/-5, 7, -15/)
```

```
y = (/2, -3, -4/)
m = mfMod(x, y)

call msDisplay(x, 'x', y, 'y', m, 'mfMod(x,y)')
call msFreeArgs(x, y, m)

end program example
```

Result

```
x  =

  -5   7 -15

 y  =

  2 -3 -4

 mfMod(x,y)  =

  1 -2 -3
```

See Also

mfRem

mfRem, msRem

Remainder after division.

Module

```
use fml
```

Syntax

```
type (mfArray) :: r, x, y

r = mfRem(x, y)
```

Descriptions

Procedure `mfRem(x, y)` returns an mfArray containing the element-by-element remainder of `x/y` division. The result lies between 0 and `mfSign(x)*mfAbs(y)`. The input `x` and `y` are either scalars or conformable arrays.

Note that :

- `mfRem(x,y) = x - y*mfFix(x/y)` for `y ≠ 0`
- `mfFix(x,y)` is the integer of the quotient `x/y`.
- If `y` is zero, `mfRem` returns `MF_NAN`.

Limitations

Arguments `x` and `y` should be integers. Due to the inexact representation of floating-point numbers on a computer, real (or complex) inputs may lead to unexpected results.

Note: Procedure msRem assumes the name of mfRem when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below finds the remainder of x/y.

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y, r

x = (/-5, 7, -15/)
y = (/2, -3, -4/)

r = mfRem(x,y)
call msDisplay(x, 'x', y, 'y', r, 'mfRem(x,y)')
call msFreeArgs(x, y, r)

end program example
```

Result

```
x  =

  -5   7 -15

 y  =

  2 -3 -4

 mfRem(x,y)  =

 -1  1 -3
```

See Also

mfMod

mfRound, msRound

Round towards nearest integer.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfRound(x)

call msRound(mfOut(y), x)
```

Descriptions

Procedure `mfRound(x)` rounds the elements of mfArray `x` to the nearest integer. The real part and imaginary part of a complex number is rounded towards the nearest integer independently.

Note: Procedure msRound assumes the name of mfRound when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below performs the round operation on mfArray `x`.

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y, u, v

x = (/-1.6, 2.3/)
u = (3.2, 2.2)

y = mfRound(x)
v = mfRound(u)

call msDisplay(x, 'x',  y, 'mfRound(x)')
call msDisplay(u, 'u',  v, 'mfRound(u)')
call msFreeArgs(x, y, u, v)

end program example
```

Result

```
x  =

  -1.6000   2.3000

 mfRound(x)  =
```

```
-2  2
u  =

3.2000 +2.2000i

mfRound(u)  =

3.0000 +2.0000i
```

See Also

mfCeil, mfFix, mfFloor

mfSign, msSign

Signum function.

Module

```
use fml
```

Syntax

```
type (mfArray) :: x, y

y = mfSign(x)
```

Descriptions

Procedure `mfSign(x)` returns an mfArray `y` containing the element-by-element information on the sign of each element of mfArray `x`.

Note that :
For elements `x > 0`, corresponding element of `y = 1`.
For elements `x = 0`, corresponding element of `y = 0`.
For elements `x < 0`, corresponding element of `y = -1`.

For nonzero elements of complex `x`, `mfSign(x) = x/mfAbs(x)`

Note: Procedure msSign assumes the name of mfSign when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below finds the sign of elements of mfArray `x`.

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y

x = (/-5, 7, 0/)
y = mfSign(x)

call msDisplay(x, 'x', y, 'mfSign(x)')

call msFreeArgs(x, y)

end program example
```

Result

```
x  =

 -5  7  0
```

```
mfSign(x)  =
-1  1  0
```

See Also

mfAbs, mfConj, mfImag, mfReal

CHAPTER 6

MOD_ELMAT

Module MOD_ELMAT contains a set of elementary matrix manipulation functions. This chapter describes the functions available, as listed below:

Matrices	
`mfEye`	Identity matrix
`mfLinspace`	Constructs linearly spaced vectors.
`mfMagic`	Constructs magic matrix.
`msMeshgrid`	Constructs grids for solving functions of 2 variables.
`mfRepmat`	Replicate and tile an array
`mfOnes`	Arrays containing ones.
`mfRand`	Arrays containing random numbers.
`mfZeros`	Arrays containing zeros.

Matrix Manipulation	
`mfDiag`	Diagonal matrices and diagonals of a matrix.
`mfFind`	Find indices and values of nonzero elements.
`mfReshape`	Change shape of an array.
`mfTril`	Extracts lower triangular of an mfArray.
`mfTriu`	Extracts upper triangular of an mfArray.

`mfLogical`	Converts numerical values to logical.

Matrices

mfEye, msEye

Construct identity matrix.

Module

```
use mod_elmat
```

Syntax

```
type(mfArray) :: a, m, n

a = mfEye(m[, n])
```

Descriptions

Procedure `mfEye` generates an identity matrix.

`a = mfEye(m)` returns an m-by-m identity matrix for scalar `m`. If mfArray `m` contains information about the shape of an array, `mfEye` returns mfArray whose shape is specified by `m`. For example, `a = mfEye(mf(Shape(b)))`.

`a = mfEye(m, n)` returns an m-by-n identity matrix.

Note: Procedure msEye assumes the name of mfEye when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a

a = mfEye(3,3)

call msDisplay(a, 'mfEye(3,3)')
call msFreeArgs(a)

end program example
```

Result

```
mfEye(3,3)  =

  1  0  0
  0  1  0
  0  0  1
```

See Also

fml

mfLinSpace, msLinSpace

Construct linearly spaced vector.

Module

```
use fml
```

Syntax

```
type(mfArray) :: a, l, u, n

a = mfLinSpace(l, u)
a = mfLinSpace(l, u, n)
```

Descriptions

Procedure `mfLinSpace` generates linearly spaced row vectors.

`a = mfLinSpace(l, u)` returns a row vector mfArray with 100 linearly and equally spaced points between `l` and `u`, where `l` is the initial value and `u` is the last value.

`a = mfLinSpace(l, u, n)` generates `n` linearly and equally spaced points between `l` and `u`.

Note: Procedure msLinSpace assumes the name of mfLinSpace when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a

a = mfLinspace(3, 4, 5)

call msDisplay(a, 'mfLinspace(3, 4, 5)')
call msFreeArgs(a)

end program example
```

Result

```
mfLinspace(3, 4, 5)  =

   3.0000   3.2500   3.5000   3.7500   4.0000
```

See Also

mfColon, mfMeshgrid

mfMagic, msMagic

Construct magic square.

Module

```
use fml
```

Syntax

```
type(mfArray) :: a, m

a = mfMagic(m)
```

Descriptions

Procedure `mfMagic` creates a magic square mfArray. A magic square is a special matrix with equal row, column and diagonal sum.

`a = mfMagic(m)` generates an m-by-m magic matrix constructed from the integers1 through `m**2`. This procedure produces valid magic squares for all `m > 0`, except for `m = 2`.

Note: Procedure msMagic assumes the name of mfMagic when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a, rsum, csum

a = mfMagic(3)
rsum = mfSum(a, 2)
csum = mfSum(a, 1)

call msDisplay(a, 'mfMagic(3)', rsum, 'row sum', csum, 'column sum')
call msFreeArgs(a, rsum, csum)

end program example
```

Result

```
mfMagic(3)  =

  8  1  6
  3  5  7
  4  9  2

 row sum  =

  15
  15
  15
```

```
column sum  =
 15  15  15
```

See Also

mfZeros, mfOnes

mfMeshgrid, msMeshgrid

Generate x and y matrices for three-dimensional plots.

Module

```
use fml
```

Syntax

```
type(mfArray) :: a, b, x, y, m, n, k

x = mfMeshgrid(m)
x = mfMeshgrid(m, n)
x = mfMeshgrid(m, n, k)

call msMeshgrid(mfOut(a, b),m)
call msMeshgrid(mfOut(a, b),m, n)
call msMeshgrid(mfOut(a, b),m, n, k)
```

Descriptions

Procedure `msMeshgrid` generates grids from vectors for solving functions of two variables and plotting three-dimensional graphs.

`x = mfMeshgrid(m)` returns `x` is equivalent to `call msMeshgrid(mfOut(a, b), m)` returns output `a`.

`x = mfMeshgrid(m, n)` returns `x` is equivalent to `call msMeshgrid(mfOut(a, b), m, n)` returns output `a`.

`x = mfMeshgrid(m, n, k)` returns `x` is equivalent to call msMeshgrid(mfOut(a, b, c), m, n, k) returns output `a`.

`call msMeshgrid(mfOut(a, b), m, n)` transforms the domain specified by vectors `m` and `n` into matrix mfArrays `a` and `b`. The rows of the output matrix `a` are copies of the vector `m` and the columns of the output matrix `b` are copies of the vector `n`.

`call msMeshgrid(mfOut(a, b), m)` is an abbreviation for `call msMeshgrid(mfOut(a, b), m, m)`.

`call msMeshgrid(mfOut(a, b, c), m, n, k)` returns a three-dimensional arrays that can be used to evaluate functions of three variables and three-dimensional volumetric plots.

Note: Procedure msMeshgrid assumes the name of mfMeshgrid when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: x, y, a, b

! Generate vectors a and b using the colon function.
a = mfColon(-1d0, 0.5d0, 1d0)
b = mfColon(-1.5d0, 0.3d0, 1.5d0)

! Use the meshgrid procedure to transform the domain
! specified by vectors a and b into a two-dimensional function domain.
call msMeshgrid(mfOut(x, y), a, b)

! Display the generated matrices.
call msDisplay(x, 'x', y, 'y')

! Release the memory occupied by the mfArrays at the
! end of the program.
call msFreeArgs(x, y, a, b)

end program example
```

Result

```
x  =

  -1.0000  -0.5000   0.0000   0.5000   1.0000
  -1.0000  -0.5000   0.0000   0.5000   1.0000
  -1.0000  -0.5000   0.0000   0.5000   1.0000
  -1.0000  -0.5000   0.0000   0.5000   1.0000
  -1.0000  -0.5000   0.0000   0.5000   1.0000
  -1.0000  -0.5000   0.0000   0.5000   1.0000
  -1.0000  -0.5000   0.0000   0.5000   1.0000
  -1.0000  -0.5000   0.0000   0.5000   1.0000
  -1.0000  -0.5000   0.0000   0.5000   1.0000
  -1.0000  -0.5000   0.0000   0.5000   1.0000

 y  =

  -1.5000  -1.5000  -1.5000  -1.5000  -1.5000
  -1.2000  -1.2000  -1.2000  -1.2000  -1.2000
  -0.9000  -0.9000  -0.9000  -0.9000  -0.9000
  -0.6000  -0.6000  -0.6000  -0.6000  -0.6000
  -0.3000  -0.3000  -0.3000  -0.3000  -0.3000
   0.0000   0.0000   0.0000   0.0000   0.0000
   0.3000   0.3000   0.3000   0.3000   0.3000
   0.6000   0.6000   0.6000   0.6000   0.6000
   0.9000   0.9000   0.9000   0.9000   0.9000
   1.2000   1.2000   1.2000   1.2000   1.2000
   1.5000   1.5000   1.5000   1.5000   1.5000
```

See Also

mfLinSpace

mfOnes, msOnes

Construct a matrix of ones.

Module

```
use fml
```

Syntax

```
type(mfArray) :: a, m, n, d3, ..., d7

a = mfOnes(m)
a = mfOnes(m, n)
a = mfOnes(m, n[, d3, d4, d5, d6, d7])
```

Descriptions

Procedure `mfOnes` generates a matrix containing ones.

`a = mfOnes(m)` returns a m-by-m matrix of ones.

`a = mfOnes(m, n)` returns a m-by-n matrix of ones.

`a = mfOnes(m, n, d3, d4, d5, d6, d7)` returns an m-by-n-by-d3-by-d4-by-d5-by-d6-by-d7 array of ones.

Note: Procedure msOnes assumes the name of mfOnes when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a

a = mfOnes(3, 2)

call msDisplay(a, 'mfOnes(3, 2)')
call msFreeArgs(a)

end program example
```

Result

```
mfOnes(3, 2)  =

  1  1
  1  1
  1  1
```

See Also

mfZeros, mfEye

mfRand, msRand

Uniformly distribute random numbers.

Module

```
use fml
```

Syntax

```
type(mfArray) :: a, b, m, n, d3, ..., d7

a = mfRand(m)
a = mfRand(m, n)
a = mfRand(m, n[, d3, d4, d5, d6, d7])
```

Descriptions

Procedure `mfRand` randomly generates an mfArray.

`a = mfRand(m)` returns a m-by-m matrix with random entries chosen from a uniform distribution in the interval (0,1).

`a = mfRand(m, n)` generates a m-by-n matrix with random entries chosen from a uniform distribution in the interval (0,1).

`a = mfRand(m, n, d3, d4, d5, d6, d7)` generates a m-by-n-by-d3-byd4-by-d5-by-d6-by-d7 matrix with random entries chosen from a uniform distribution in the interval (0,1).

Note: Procedure msRand assumes the name of mfRand when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a

a = mfRand(3, 2)

call msDisplay(a, 'mfRand(3, 2)')
call msFreeArgs(a)

end program example
```

Result

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

```
0.5497   0.0012
0.5496   0.6887
0.4378   0.6035
```

See Also

mfZeros, mfOnes, mfMagic

mfRepmat, msRepmat

Replicate and tile an array.

Module

```
use fml
```

Syntax

```
type(mfArray) :: a, m, n, p

x = mfRepmat(a, m, n)
x = mfRepmat(a, p)
```

Descriptions

Procedure `mfRepmat` generates matrix mfArrays by replicating copies of an array into a larger block array.

`x = mfRepmat(a, m, n)`, `call msRepmat(mfOut(x), a, m, n)` generate an mfArray `x` consisting of m-by-n tiling of copies of vector mfArray a.

`x = mfRepmat(a, p)`, `call msRepmat(mfOut(x), a, p)` generate an mfArray `x` containing `p` block copies of mfArray `a`. Vector `p` contains information about the number of blocks mfArray `x` in each dimension. The information is in the form of [m, n], or [m, n, d3, ..., d7].

Note: Procedure msRepmat assumes the name of mfRepmat when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) ::  a

! Create a 3-by-2 mfArray consisting of ones.
a = mfRepmat(mf(1), mf((/3, 2/)))
! This is similar to mfOnes(3,2) but is much faster.
call msDisplay (a, ' mfRepmat(mf(1), mf((/3,2/)))')

! Create  3-by-2 block copies of mfMagic(3)
a = mfRepmat(mfMagic(2), mf((/3,2/)))
call msDisplay (a, 'mfRepmat(mfMagic(2), (/3,2/))')

! Deallocate mfArrays.
call msFreeArgs(a)

end program example
```

Result

```
mfRepmat(mf(1), mf((/3,2/)))  =

  1  1
  1  1
  1  1

 mfRepmat(mfMagic(2), (/3,2/))  =

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

See Also

mfMeshgrid

mfZeros, msZeros

Show matrix with zeros.

Module

```
use fml
```

Syntax

```
type(mfArray) :: a, m, n, d3, ..., d7

a = mfZeros(m)
a = mfZeros(m, n)
a = mfZeros(m, n[, d3, ..., d7])
```

Descriptions

Procedure `mfZeros` generates the mfArrays of zeros. It used to allocate memory for mfArrays.

`a = mfZeros(m)` returns a m-by-m mfArray of zeros.

`a = mfZeros(m, n)` returns a m-by-n mfArray of zeros.

`a = mfZeros(m, n, d3, ..., d7)` returns a m-by-n-by-d3-by-...-by-d7 array of zeros.

Note: Procedure msZeros assumes the name of mfZeros when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a

a = mfZeros(2, 2)

call msDisplay(a, 'mfZeros(2, 2)')
call msFreeArgs(a)

end program example
```

Result

```
mfZeros(2, 2)  =

  0  0
  0  0
```

See Also

mfOnes, mfEye

Matrix Manipulation

mfDiag, msDiag

Request diagonals of a matrix.

Module

```
use fml
```

Syntax

```
type(mfArray) :: d, a
integer :: k

d = mfDiag(a[, k])
a = mfDiag(d[, k])
```

Descriptions

Procedure `mfDiag(a)` returns a vector mfArray `d`, containing the elements extracted from the main diagonal of matrix mfArray `a`.

`a = mfDiag(d)` returns a diagonal matrix mfArray `a`, with its main diagonal composed of member elements of vector `d`.

`d = mfDiag(a, k)` returns a vector mfArray `d`, containing the elements extracted from the kth diagonal of matrix mfArray `a`.

`a = mfDiag(d, k)` returns a diagonal matrix `a` of order mfLength(d) + mfAbs(k) whose kth diagonal is composed of elements from vector mfArray `d`. k = 0 represents the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the main diagonal.

Figure 4.8: The kth diagonal of a matrix.

Note: Procedure msDiag assumes the name of mfDiag when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a, d, b

! Construct an mfArray using the magic function.
a = mfMagic(3)

! Extract the 1st diagonal of mfArray a.
d = mfDiag(a, 1)
```

```
! Construct an mfArray b, whose main diagonal is
! composed of d.
b = mfDiag(d)

! Display the resulting mfArrays.
call msDisplay(a, 'mfMagic(3)', d, 'k = 1 diagonal')
call msDisplay(b, 'mfDiag(d)')

! Release the memory occupied by mfArrays a ,b, and d.
call msFreeArgs(a, b, d)

end program example
```

Result

```
mfMagic(3)  =

  8  1  6
  3  5  7
  4  9  2

 k = 1 diagonal  =

  1
  7

 mfDiag(d)  =

  1  0
  0  7
```

See Also

mfTriu, mfTril

mfFind, msFind

Find indices of nonzero elements.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, y, i, j, v

y = mfFind(x)

call msFind(mfOut(i, j), x)
call msFind(mfOut(i, j, v), x)
```

Descriptions

Procedure `mfFind` generates indices of nonzero elements.

`y = mfFind(x)` returns an mfArray `y`, containing long column indices of nonzero entries in the mfArray `x`. If none is found, `mfFind` returns an empty matrix.

`call msFind(mfOut(i, j), x)` returns two mfArrays `i` and `j` containing the row and column indices of nonzero entries in matrix mfArray `x`.

`call msFind(mfOut(i, j, v), x)` returns three mfArrays `i`, `j`, and `v`, containing the row indices, column indices, and nonzero entries of matrix mfArray `x` respectively.

Note: Procedure msFind assumes the name of mfFind when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The example below retrieves the indices of non-zero elements of a matrix mfArray.

Code

```
program example

use fml
implicit none

type (mfArray) :: a, i, j

a = mfMagic(3) > 7
call msFind(mfOut(i,j), a)

call msDisplay(a, 'a', i, 'i', j, 'j')
call msFreeArgs(a, i, j)

end program example
```

Result

```
a  =

  1  0  0
  0  0  0
  0  1  0

 i  =

  1
  3

 j  =

  1
  2
```

See Also

mfColon, relational_operators

mfLogical, msLogical

Convert numeric values to logical.

Module

```
use fml
```

Syntax

```
type(mfArray) :: l, x

l = mfLogical(x)
```

Descriptions

Procedure `mfLogical(x)` returns an logical mfArray. An element of mfArray `l` is assigned logical "true" if the corresponding element in mfArray `x` is nonzero, otherwise it is assigned "false".

Notice that most arithmetic operations remove the logical characteristic from an array. For example, adding zero to a logical array.

Note: Procedure msLogical assumes the name of mfLogical when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: l, x, y, z

x = mfEye(3)
l = mfLogical(x)
y = (/1,2,3/) .vc. (/4,5,6/) .vc. (/7,8,9/)
z = mfS(y, l)

call msDisplay(y, 'y',z, 'z')

call msFreeArgs(l,x,y,z)

end program example
```

Result

```
y =

  1  2  3
  4  5  6
  7  8  9

 z =
```

```
1
5
9
```

See Also

mfZeros, mfOnes, mfMagic

mfReshape, msReshape

Change size of a matrix.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, y, d3, ..., d7
integer :: m, n

y = mfReshape(x, m, n)
a = mfReshape(b, m[, n, d3, …, d7])
```

Descriptions

Procedure `mfReshape` reshapes elements.

`y = mfReshape(x, m, n)` returns the m-by-n matrix mfArray `y` whose elements are taken column wise from mfArray `x`. An error occurs if `x` is not an m-by-n matrix.

`a = mfReshape(b, m, n, d3, ..., d7)` returns the m-by-n-by-d3-by-d4-by-d5-by-d6-by-d7 matrix mfArray `a` whose elements are taken column wise from mfArray `b`. An error occurs if `b` is not an m-by-n-by-d3-by-d4-by-d5-by-d6-by-d7 matrix.

Note: Procedure msReshape assumes the name of mfReshape when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type(mfArray):: x, y

x = (/1, 2, 3, 4, 5, 6/)
y = mfReshape(x, (/3, 2/))
call msDisplay(x, 'x', y, 'mfReshape(x, (/3, 2/))')

call msFreeArgs(x, y)

end program example
```

Result

```
x  =

  1  2  3  4  5  6
```

```
mfReshape(x, (/3, 2/))  =

 1  4
 2  5
 3  6
```

See Also

mfSize

mfTril, msTril

Lower triangular part of a matrix.

Module

```
use fml
```

Syntax

```
type(mfArray) :: l, a
integer :: k

l = mfTril(a[, k])
```

Descriptions

Procedure `mfTril` generates the lower triangular part of a matrix

`l = mfTril(a)` returns the mfArray `l` containing the elements on and below the main diagonal of mfArray `a`.

`l = mfTril(a,k)` returns the mfArray `l` containing the elements on and below the kth diagonal of mfArray `a`. k = 0 is the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the main diagonal.

Figure 4.12: The kth diagonal of a matrix.

Note: Procedure msTril assumes the name of mfTril when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a, l, l1

! Construct an mfArray using the magic function.
a = mfMagic(3)

! Get the lower triangular of mfArray a from the main
! diagonal downwards.
l = mfTril(a)

! Get the lower triangular a from the 1sdiagonal downwards.
l1 = mfTril(a, 1)

! Display the resulting mfArrays.
call msDisplay(a, 'mfmagic(3)', l, 'lower triangular')
```

```
call msDisplay(l1, 'lower triangular from k = 1')

! Release the memory occupied by mfArrays a, l, and l1.
call msFreeargs(a, l, l1)

end program example
```

Result

```
mfmagic(3)  =

  8  1  6
  3  5  7
  4  9  2

 lower triangular  =

  8  0  0
  3  5  0
  4  9  2

 lower triangular from k = 1  =

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

See Also

mfTriu, mfDiag

mfTriu, msTriu

Upper triangular part of a matrix.

Module

```
use fml
```

Syntax

```
type(mfArray) :: u, a
integer :: k

u = mfTriu(a[, k])
```

Descriptions

Procedure `mfTriu` generates the upper triangular part of a matrix.

`u = mfTriu(a)` returns an mfArray `u` containing the elements on and above the main diagonal of mfArray `a`.

`u = mfTriu(a, k)` returns an mfArray `u` containing the elements on and above the kth diagonal of mfArray `a`. k = 0 is the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the main diagonal.

Figure 4.13 The kth diagonal of a matrix.

Note: Procedure msTriu assumes the name of mfTriu when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

Code

```
program example

use fml
implicit none

type (mfArray) :: a, u, u1

! Construct an mfArray using the magic function.
a = mfMagic(3)

! Extract the upper triangular of mfArray a from the main
! diagonal upwards.
u = mfTriu(a)

! Extract the upper triangular a from the 1sdiagonal upwards.
u1 = mfTriu(a, 1)

! Display the resulting mfArrays.
call msDisplay(a, 'mfMagic(3)', u, 'upper triangular')
call msDisplay(u1, 'upper triangular from k = 1')
```

```
! Release the memory occupied by mfArrays a, u, and u1.
call msFreeArgs(a, u, u1)

end program example
```

Result

```
mfMagic(3)  =

  8  1  6
  3  5  7
  4  9  2

 upper triangular  =

  8  1  6
  0  5  7
  0  0  2

 upper triangular from k = 1  =

  0  1  6
  0  0  7
  0  0  0
```

See Also

mfTril, mfDiag

CHAPTER 7

MATFUN

Module MATFUN contains a set of matrix functions for solving numerical algebra problems.

This chapter describes the functions available, as listed below:

Matrix Analysis

mfDet

mfNorm

mfRank

mfTrace

Linear Equations

mfChol

mfCond

mfInv

mfRcond

mfLu

mfQr

Eigenvalues and singular values

```
mfEig

mfHess

mfQz

mfSchur

mfSvd
```

Factorization Utilities

```
mfBalance
```

Matrix Analysis

mfDet

Matrix determinant.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, d

d = mfDet(x)
```

Descriptions

Procedure `mfDet(x)` returns a scalar mfArray containing the determinant of square matrix mfArray `x`. For matrices of modest order with small integer entries, it can be used as a test for matrix singularity.

Note: Procedure msDet assumes the name of mfDet when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfDet` procedure to compute the determinant of a non-singular square matrix mfArray `x`.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, d

! Construct a 3-by-3 mfArray x using vertical concatenation.
x = (/1.0d0, 2.0d0, 3.0d0/) .vc. &
    (/7.0d0, 8.0d0, 9.0d0/) .vc. &
    (/1.0d0, 2.0d0, 4.0d0/)

! Compute determinant of mfArray x
d = mfDet(x)

! Display value of x and determinant of x
call msDisplay(x, 'x', d, 'mfDet(x)')

! Deallocate mfArrays x and d
call msFreeArgs(x, d)

end program example
```

Result

```
x  =

  1  2  3
  7  8  9
```

```
 1  2  4
d  =
-6
```

See Also

mfCond, mfInv, mfLu

mfNorm

Matrix or vector norm.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, p, n

n = mfNorm(x[, p])
```

Descriptions

`n = mfNorm(x)`, `n = mfNorm(x, p)`
Procedure `mfNorm` generates norm value differently for matrices and vectors.

For matrices:

- `mfNorm(x)` returns a scalar mfArray containing the largest singular value of mfArray `x`.
- `mfNorm(x, p)` returns a different kind of norm, depending on the value of `p`.
- `mfNorm(x, 1)` returns the largest column sum of `x`.
- `mfNorm(x, 2)` is equiv. to `mfNorm(x)`. It returns the largest singular value of mfArray `x`.
- `mfNorm(x, MF_INF)` returns the largest row sum of `x`, which is also the infinity norm of `x`.
- `mfNorm(x, 'fro')` returns the Frobenius norm.

For vectors:

- `mfNorm(v, 1)` returns a scalar mfArray equal to the sum of elements of mfArray `v`.
- `mfNorm(v)` is the same as `mfNorm(v, 2)`, and returns the value of `mfSum(mfAbs(v).^2)^(1/2)`.

Note: Procedure msNorm assumes the name of mfNorm when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfNorm` procedure to compute the norm of a non-singular square matrix mfArray `x`.

Code

```
program example

use fml
implicit none
```

```
type(mfArray) :: x, n

! Construct a 3-by-3 mfArray x using vertical concatenation.
x = (/1.0d0, 2.0d0, 3.0d0/) .vc. &
    (/7.0d0, 8.0d0, 9.0d0/) .vc. &
    (/1.0d0, 2.0d0, 4.0d0/)

! Compute norm of mfArray x
n = mfNorm(x)

! Display value of x and norm n of x
call msDisplay(x, 'x', n, 'norm')

! Deallocate mfArrays x and n
call msFreeArgs(x, n)

end program example
```

Result

```
x  =

  1  2  3
  7  8  9
  1  2  4

 norm  =

   15.0130
```

See Also

mfCond

mfRank

Estimate for the number of linearly independent rows or columns.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, r, tol

r = mfRank(x[, tol])
```

Descriptions

For matrices, procedure `mfRank(x)` is used as an estimation for the number of linearly independent rows and columns of a matrix `x`.

`mfRank(x), mfRank(x[, tol])`

- Procedure `mfRank(x)` returns a scalar mfArray containing the number of independent rows or columns of a matrix `x`.
- Procedure `mfRank(x)` uses the default `tol = mfMax(mfSize(x)) * mfNorm(x) * MF_EPS`.
- Procedure `mfRank(x, tol)` returns the singular values of matrix `x` that are larger than `tol`.

Note: Procedure msRank assumes the name of mfRank when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfRank` procedure to compute the rank of a square matrix mfArray `x`.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, r

! Construct a 3-by-3 mfArray x using vertical
! concatenation.
x = (/1.0d0, 2.0d0, 3.0d0/) .vc. &
    (/7.0d0, 8.0d0, 9.0d0/) .vc. &
    (/2.0d0, 4.0d0, 6.0d0/)

! Compute the rank of mfArray x
r = mfRank(x)
```

```
! Display value of x and the rank r of x
call msDisplay(x, 'x', r, 'mfRank(x)')

! Deallocate mfArrays x and r
call msFreeArgs(x, r)

end program example
```

Result

```
x  =

  1  2  3
  7  8  9
  2  4  6

 mfRank(x)  =

  2
```

See Also

mfSize

mfTrace, msTrace

Return the sum of diagonal elements.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, s

s = mfTrace(x)
```

Descriptions

For matrices, procedure `mfTrace(x)` returns the sum of diagonal elements of mfArray `x`, which is equivalent to the sum of eigenvalues of mfArray `x`.

Note: Procedure msTrace assumes the name of mfTrace when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfTrace` procedure to compute the sum of the diagonal elements of the square matrix mfArray `x`.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, s

! Construct a 3-by-3 mfArray x using vertical
! concatenation.
x = (/10.0d0, 0.0d0, 6.0d0/) .vc. &
   (/0.0d0, -3.0d0, 9.0d0/) .vc. &
   (/0.0d0, 2.0d0, 4.0d0/)

! Compute the trace of mfArray x
s = mfTrace(x)

! Display value of x and trace s of x
call msDisplay(x, 'x', s, 'mfTrace(x)')

! Deallocate mfArrays x and s
call msFreeArgs(x, s)

end program example
```

Result

```
x  =

  10   0   6
   0  -3   9
```

```
  0   2   4
mfTrace(x)  =
 11
```

Linear Equations

mfChol, msChol

Cholesky factorization.

Module

```
use fml
```

Syntax

```
type(mfArray) :: r, x, p

r = mfChol(x)

call msChol(mfOut(r, p), x)
```

Descriptions

Procedure `mfChol(x)` uses only the diagonal and upper triangle of `x`.

`r = mfChol(x)`

For matrices, if mfArray `x` is positively definite, then Procedure `mfChol(x)` returns an upper triangular mfArray `r` so that `.h.r * r = x`. If `x` is not a positive definite mfArray, an error is occurred.

`call msChol(mfOut(r, p), x)`

Error is prevented from occurring when `call msChol(mfOut(r, p), x)` is used. If `x` is positively definite, `p` is 0 and `r` is the same as above. Otherwise, `p` is a positively scalar mfArray and `r` is an upper triangular mfArray such that: `.h.r*r = mfGet(x, mfColon(1, p-1),mfColon(1,p-1))`

Note: Procedure msChol assumes the name of mfChol when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfChol` procedure to derive the Cholesky factorization of a non-positive square matrix mfArray `x`.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, r ,p

! Construct a 3-by-3 mfArray x using vertical concatenation.
x = (/10.0d0, 2.0d0, 6.0d0/) .vc. &
    (/6.0d0, 3.0d0, 9.0d0/)  .vc. &
    (/3.0d0, 2.0d0, 4.0d0/)

! Compute mfChol of mfArray x
```

```
call msChol(mfout(r, p), x)

! Display value of x, r and p
call msDisplay(x, 'x', r, 'r', p, 'p')

! Deallocate mfArrays x and r and p
call msFreeArgs(x, r, p)

end program example
```

Result

```
x  =

  10   2   6
   6   3   9
   3   2   4

 r  =


   3.1623   0.6325
   0.0000   1.6125


 p  =

  3
```

See Also

mfLu

mfCond

Return condition number of a matrix.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, c, p

c = mfCond(x[, p])
```

Descriptions

For matrices, Procedure `mfCond` returns the p-norm condition number of `x`.

`c = mfCond(x)`, `c = mfCond(x, p)`

- Procedure `mfCond(x)` returns the 2-norm condition number. It is also the ratio of the largest singular value of `x` to the smallest. A large condition number indicates a nearly singular mfArray `x`.
- Specifying argument `p`, it returns a p-norm condition number of `x`, which is equal to `mfNorm(x, p) * mfNorm(mfInv(x), p)`, where `p` is `1, 2, MF_INF or 'fro.'`

[p=1	The mfCond returns the 1-norm condition number
p=2	The mfCond returns the 2-norm condition number
p='fro'	Return the Frobenius norm condition number
p=MF_INF	Return the Infinity norm condition number]

Note: Procedure msCond assumes the name of mfCond when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfCond` procedure to compute the condition number with respect to inversion in the 2-norm of a square matrix mfArray `x`.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, c

! Construct a 3-by-3 mfArray x using vertical concatenation.
x = (/10.0d0, 8.0d0, 6.0d0/) .vc. &
   (/5.0d0, 15.0d0, 5.0d0/) .vc. &
```

```
    (/6.0d0, 7.0d0, 8.0d0/)

! Compute the condition number with respect to inversion
! of mfArray x
c = mfCond(x)

! Display value of x and c of x
call msDisplay(x, 'x', c, 'mfCond(x)')

! Deallocate mfArrays x and c
call msFreeArgs(x, c)

end program example
```

Result

```
x  =

  10   8   6
   5  15   5
   6   7   8

 mfCond(x)  =


   8.2853
```

See Also

mfNorm

mfInv

Matrix inverse.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, y

y = mfInv(x)
```

Descriptions

For matrices, Procedure `mfInv(x)` returns the inverse of mfArray `x`.

`y = mfInv(x)`
Procedure `mfInv` is applicable only for square matrix `x`. A warning message will be printed when `x` is badly scaled or nearly singular.

Note: Procedure msInv assumes the name of mfInv when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfInv` procedure to compute inverse of a matrix mfArray `x`.

See Also

mfCond

mfRcond

LINPACK reciprocal condition estimator.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, c

c = mfRcond(x)
```

Descriptions

For matrices, Procedure `mfRcond(x)` uses the LAPACK condition estimator to get an estimate for the reciprocal of the condition of `x` in the 1-norm.

`c = mfRcond(x)` returns a value near 1.0 when matrix `x` is well conditioned. And it returns a value near 0.0 when `x` is badly conditioned.

Note: Procedure msRcond assumes the name of mfRcond when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfRcond` procedure to compute the reciprocal of the condition of x in the 1-norm of a square matrix mfArray `x`.

See Also

mfCond, mfNorm

mfLu, msLue

LU factorization.

Module

```
use fml
```

Syntax

```
type(mfArray) :: x, y, l, u, p

y = mfLu(x)

call msLu(mfOut(l, u), x)
call msLu(mfOut(l, u, p), x)
```

Descriptions

Procedure `mfLu(x)` returns LU decomposition of a square mfArray `x`.

`call msLu(mfout(l, u), x)`
When it is used, the procedure returns mfArray `u` containing the upper triangular matrix, and mfArray `l` containing product of the lower triangular matrix and permutation array, so that `x = l.Mul.u`

`call msLu(mfout(l, u, p), x)`
When it is used, the procedure returns an upper triangular mfArray `u`, lower triangular mfArray `l`, and permutation mfArray `p`, such that `p.Mul.x = l.Mul.u.`
When `y = mfLu(x)` is used, the procedure returns the one output from LINPACK'S ZGEFA routine.

Note: Procedure msLu assumes the name of mfLu when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfLu` procedure to compute the LU decomposition of matrix mfArray `x`.

Code

```
program example

use fml
implicit none

type(mfArray) :: x, l, u, p

! Construct a 3-by-3 mfArray x using vertical concatenation.
x = (/1.0d0, 2.0d0, 3.0d0/) .vc. &
   (/7.0d0, 8.0d0, 9.0d0/) .vc. &
   (/1.0d0, 2.0d0, 4.0d0/)
```

```
! Compute lu decomposition of mfArray x
call msLu(mfout(l, u, p), x)

! Display value of x, l, u and p
call msDisplay(x, 'x', l, 'l' ,u, 'u', p, 'p')

! Deallocate mfArrays x, l , u, p
call msFreeArgs(x, l, u, p)

end program example
```

Result

```
x  =

  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  =

  0  1  0
  1  0  0
  0  0  1
```

See Also

mfQr

mfQr, msQr

Orthogonal-triangular decomposition.

Module

```
use fml
```

Syntax

```
type(mfArray) :: q, r, a, e

call msQr(mfOut(q, r), a[, 0])
call msQr(mfOut(q, r, e), a[, 0])
```

Descriptions

Procedure `mfQr` returns the orthogonal-triangular decomposition of a matrix.

`call msQr(mfOut(q, r), a)`
This procedure returns an upper triangular mfArray `r` of the same shape as `a` and a unitary mfArray matrix `q`, such that `a = q*r`.

`call msQr(mfOut(q, r), a, 0)`
This procedure performs an "economy size" decomposition. If `a` is a m-by-n mfArray with `m > n`, only the first `n` columns of `q` will be computed.

`call msQr(mfOut(q, r, e), a)`
This procedure returns an upper triangular mfArray `r`, a unitary mfArray `q`, and a permutation matrix mfArray `e`, so that `a*e = q*r`.

`call msQr(mfOut(q, r, e), a, 0)`
This procedure performs an "economy size" decomposition, returning a permutation vector `e`, so that `q*r = mfGet(a, MF_COL, e)`. The column permutation `e` is chosen so that `mfAbs(mfDiag(r))` is decreasing.

Note: Procedure msQr assumes the name of mfQr when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfQr` procedure to compute the orthogonal -triangular decomposition of matrix mfArray `x`.

Code

```
program example

use fml
implicit none
```

```
type(mfArray) :: q, r, a, e

! Construct a 3-by-3 mfArray a using vertical
! concatenation.
a = (/1.0d0, 2.0d0, 3.0d0/) .vc. &
    (/7.0d0, 8.0d0, 9.0d0/) .vc. &
    (/1.0d0, 2.0d0, 4.0d0/)

! Compute qr decomposition of mfArray a
call msQr(mfOut(q, r, e), a)

! Display value of a, q, r and e
call msDisplay(a, 'a', q, 'q' ,r, 'r', e, 'e')

! Deallocate mfArrays a, q, r, e
call msFreeArgs(q, r, a, e)

end program example
```

Result

```
a  =

  1  2  3
  7  8  9
  1  2  4

 q  =


  -0.2914   0.4491  -0.8447
  -0.8742  -0.4836   0.0445
  -0.3885   0.7513   0.5335


 r  =


  -10.2956   -6.7990   -8.3531
    0.0000   -2.1849   -1.4681
    0.0000    0.0000   -0.2667


 e  =

  0  1  0
  0  0  1
  1  0  0
```

See Also

mfLu

Eigenvalues and singular values

mfEig, msEig

Eigenvalues and eigenvectors.

Module

```
use fml
```

Syntax

```
type(mfArray) :: a, b, e, v, d

e = mfEig(a[, flag])
e = mfEig(a, b[, flag])

call msEig(mfOut(v, d), a[, flag])
call msEig(mfOut(v, d), a, b[, flag])
```

Descriptions

Procedure `mfEig` computes the eigenvalues and eigenvectors of a matrix mfArray `a`.

```
e = mfEig(a)
e = mfEig(a, flag)
```

- This procedure returns a vector `e` containing the eigenvalues of square mfArray `a`.
- Argument `flag` can be a string containing 'nobalance', so that the procedure can perform the computation with balancing switched off. This usually produces more accurate results for specific problems.

```
call msEig(mfOut(v, d), a)
call msEig(mfOut(v, d), a, flag)
```

- This procedure returns a diagonal matrix `d` of eigenvalues, and a full matrix mfArray `v` whose columns are the corresponding eigenvectors such that `a*v = v*d`.

```
e = mfEig(a, b)
e = mfEig(a, b, flag)
```

- This procedure returns a vector containing the generalized eigenvalues of square matrix mfArrays `a` and `b`.
- The procedure specifies the algorithm used to compute eigenvalues and eigenvectors through the argument `flag`, which is specified in More Detail below.

```
call msEig(mfOut(v, d), a, b)
call msEig(mfOut(v, d), a, b ,flag)
```

- This procedure returns a diagonal matrix mfArray `d` of generalized eigenvalues and a full matrix mfArray `v` whose columns are the corresponding eigenvectors so that `a*v = b*v*d`.

- The procedure specifies the algorithm used to compute eigenvalues and eigenvectors through the argument `flag`, which is specified in More Detail below.

More Detail
Argument `flag` can be:
'chol'
This is the default for symmetric(Hermitian) `a` and symmetric(Hermitian) positive definite `b`. The generalized eigenvalues of `a` and `b` is computed by using Cholesky factorization of `b`.
'qz'
This uses the mfQz algorithm to compute eigenvlaues as for nonsymmetrical (non-Hermitian) `a` and `b`.

Note: Procedure msEig assumes the name of mfEig when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfEig` procedure to compute the eigenvalues and eigenvectors of a square matrix mfArray `a`.

Code

```
program example

use fml
implicit none

type(mfArray) :: v, d, a

! Construct a 3-by-3 mfArray a using vertical concatenation.
a = (/5.0d0, 4.0d0, 3.0d0/) .vc. &
    (/2.0d0, 4.0d0, 6.0d0/) .vc. &
    (/10.0d0, 15.0d0, 25.0d0/)

! Compute eigenvalues and eigenvectors of mfArray a
call msEig(mfout(v, d), a)

! Display value of a, v and d
call msDisplay(a, 'a', v, 'v' ,d, 'd')

! Deallocate mfArrays a, v, d
call msFreeArgs(a, v, d)

end program example
```

Result

```
a  =

   5   4   3
   2   4   6
  10  15  25

 v  =


  -0.1512  -0.9354   0.5446
  -0.2317   0.2120  -0.7954
  -0.9610   0.2830   0.2660
```

```
d  =

  30.1904    0.0000    0.0000
   0.0000    3.1857    0.0000
   0.0000    0.0000    0.6238
```

See Also

mfBalance, mfHess, mfQz, mfSchur

mfHess, msHess

Hessenberg form.

Module

```
use fml
```

Syntax

```
type(mfArray) :: p, h, a

h = mfHess(a)

call msHess(mfout(p, h), a)
```

Descriptions

Procedure `mfHess` returns the Hessenberg form of an mfArray.

`h = mfHess(a)`

This procedure returns an mfArray `h` with zeros below the first sub diagonal and has the same eigenvalues as `a`. If the original mfArray `a` is symmetric or Hermitian, `h` will be tridiagonal.

`call msHess(mfout(p, h), a)`

This procedure produces an unitary matrix `p` and a Hessenberg matrix `h` so that `a = p*h*.h.p` and `.h.p*p` is an identity matrix.

Note: Procedure msHess assumes the name of mfHess when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfHess` procedure to compute the Hessenberg form of a square matrix mfArray `a`.

Code

```
program example

use fml
implicit none

type(mfArray) :: a, p, h

! Construct a 3-by-3 mfArray a using vertical concatenation.
a = (/5.0d0, 4.0d0, 3.0d0/) .vc. &
   (/9.0d0, 8.0d0, 7.0d0/) .vc. &
   (/5.0d0, 1.0d0, 2.0d0/)

! Compute the Hessenberg form of mfArray a
call msHess(mfout(p, h), a)

! Display value of a, p and h
call msDisplay(a, 'a', p, 'p' , h, 'h')
```

```
! Deallocate mfArrays a, p, h
call msFreeArgs(a, p, h)

end program example
```

Result

```
a  =

  5  4  3
  9  8  7
  5  1  2

 p  =


   1.0000   0.0000   0.0000
   0.0000  -0.8742  -0.4856
   0.0000  -0.4856   0.8742


 h  =


    5.0000   -4.9536    0.6799
  -10.2956    9.9811   -2.5660
    0.0000    3.4340    0.0189
```

See Also

mfQz, msQz

QZ factorization for generalized eigenvalues.

Module

```
use fml
```

Syntax

```
type(mfArray) :: aa, bb, q, z, v, a, b

aa  = mfQz(a, b[, flag])

call msQz(mfOut(aa[, bb, q, z, v]), a, b[, flag])
```

Descriptions

Procedure `mfQz` performs qz factorization for generalized eigenvalues of square mfArrays.

`call msQz(mfOut(aa, bb, q, z, v), a, b)`
This procedure returns upper triangular mfArrays `aa` and `bb`, the left and right transformed mfArrays `q` and `z`, and the generalized eigenvector mfArray `v`, such that `q*a*z = aa`, and `q*b*z = bb`.

`call msQz(mfOut(aa, bb, q, z, v), a, b[, flag])`
This procedure depends on the value of flag:
'complex' produces a possibly complex decomposition with a triangular `aa`. This is the default option.
'real' produces a real decomposition with a quasitriangular `aa`, containing 1-by-1 and 2-by-2 blocks on its diagonal.

Note: Procedure msQz assumes the name of mfQz when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfQz` procedure to compute the qz factorization of square matrices in mfArray `a` and `b`.

Code

```
program example

use fml
implicit none

type(mfArray) :: aa, bb, q, z, v, a, b

! Construct a 3-by-3 mfArray a and b using vertical
! concatenation.
a = (/4.0d0, 2.0d0, 3.0d0/) .vc. &
   (/7.0d0, 8.0d0, 9.0d0/) .vc. &
   (/5.0d0, 2.0d0, 4.0d0/)
```

```
b = (/3.0d0, 5.0d0, 7.0d0/) .vc. &
    (/2.0d0, 1.0d0, 1.0d0/) .vc. &
    (/3.0d0, 2.0d0, 5.0d0/)

! Compute qz factorization of mfArray a and b
call msQz(mfOut(aa, bb, q, z, v ), a, b)

! Display values of aa, bb, q, z and v
call msDisplay(aa, 'aa', bb, 'bb' ,q, 'q', z, 'z', v, 'v')

! Deallocate mfArrays a, b, aa, bb, q, z, v
call msFreeArgs(a, b, aa, bb, q, z, v)

end program example
```

Result

```
aa  =

column   1  to   3

  6.3661  +0.0000i   -9.8642  +0.0000i   10.4110  +0.0000i
  0.0000  +0.0000i   -1.4632  +0.0000i    4.2403  +0.0000i
  0.0000  +0.0000i    0.0000  +0.0000i    1.2882  +0.0000i


 bb  =


 2.3112 +0.0000i  -6.1276 +0.0000i   6.0482 +0.0000i
 0.0000 +0.0000i   4.3735 +0.0000i  -4.9870 +0.0000i
 0.0000 +0.0000i   0.0000 +0.0000i   1.8797 +0.0000i


 q  =


-0.4555 +0.0000i  -0.7072 +0.0000i  -0.5407 +0.0000i
 0.6410 +0.0000i  -0.6820 +0.0000i   0.3521 +0.0000i
-0.6178 +0.0000i  -0.1862 +0.0000i   0.7640 +0.0000i


 z  =


-0.9116 +0.0000i   0.4095 +0.0000i   0.0353 +0.0000i
-0.1805 +0.0000i  -0.3219 +0.0000i  -0.9294 +0.0000i
 0.3693 +0.0000i   0.8536 +0.0000i  -0.3674 +0.0000i


 v  =


-1.0000 +0.0000i  -0.7564 +0.0000i   0.0489 +0.0000i
-0.1980 +0.0000i  -0.4240 +0.0000i  -1.0000 +0.0000i
 0.4051 +0.0000i   1.0000 +0.0000i   0.8466 +0.0000i
```

See Also

mfEig

mfSchur, msSchur

Perform Schur decomposition.

Module

```
use fml
```

Syntax

```
type(mfArray) :: u, qt, a, flag

qt = mfSchur(a[, flag])

call msSchur(mfOut(u, qt), a)
```

Descriptions

Procedure `mfSchur` performs the Schur decomposition of a square matrix mfArray.

```
qt = mfSchur(a)
qt = mfSchur(a, flag)
```

- This procedure returns a quasi-triangular Schur matrix mfArray `qt`. If `a` is complex, `mfSchur` returns the complex Schur form in matrix `qt`. The complex Schur form is an upper triangular matrix containing the eigenvalues of `a` on the diagonal.
- This procedure returns a Schur matrix `qt` in one of two forms for real matrix `a`, depending on the value of flag:
- If `flag` is 'complex', `qt` is triangular. If `a` has complex eigenvalues, `qt` is complex. If `flag` is 'real', `qt` has real eigenvalues on the diagonal. Complex eigenvalues are located in 2-by-2 blocks on the diagonal. By default, `flag` is 'real'.

```
call msSchur(mfOut(u, qt), a)
```

- This procedure returns a unitary matrix `u` and a quasi-triangular Schur matrix mfArray `qt` such that `a = u*qt*.h.u` and `.h.u*u` is an identity matrix given by `mfEye(Shape(u))`.

Note: Procedure msSchur assumes the name of mfSchur when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfSchur` procedure to compute the Schur decomposition of a square matrix mfArray `a`.

Code

```
program example
```

```
use fml
implicit none

type(mfArray) :: u,  qt,  a

! Construct a 3-by-3 mfArray a using vertical
! concatenation.
a = (/3.0d0, 4.0d0, 3.0d0/) .vc. &
    (/2.0d0, 1.0d0, 4.0d0/) .vc. &
    (/6.0d0, 2.0d0, 2.0d0/)

! Compute the Schur decomposition of mfArray a
call msSchur(mfOut(u, qt), a)

! Displays values of a, u and qt
call msDisplay(a, 'a', u, 'u' , qt, 'qt')

! Deallocate mfArrays a, u, qt
call msFreeArgs(a, u, qt)

end program example
```

Result

```
a  =

  3  4  3
  2  1  4
  6  2  2

 u  =


   0.6118   0.7878  -0.0707
   0.4634  -0.4295  -0.7751
   0.6410  -0.4415   0.6279


 qt  =


   9.1729   1.2110  -0.5957
   0.0000  -1.5865  -1.4934
   0.0000   2.4025  -1.5865
```

See Also

mfSvd, msSvd

Perform singular value decomposition.

Module

```
use fml
```

Syntax

```
type(mfArray) :: u, s, v, a

s = mfSvd(a)

call msSvd(mfOut(u, s, v), a[, 0])
```

Descriptions

Procedure `mfSvd` performs the singular value decomposition on matrix mfArray `a`.

`s = mfSvd(a)` returns a vector in mfArray, containing singular values.

`call msSvd(mfout(u, s, v), a)` returns two unitary matrices `u` and `v`, and a diagonal matrix `s` such that `a = u*s*.h.v`. Diagonal matrix `s` has the same shape as `a` and contains nonnegative elements in decreasing order.

`call msSvd(mfout(u, s, v), a, 0)` returns the "economy size" decomposition. If `a` is m-by-n and m > n, then only the first `n` columns of `u` are computed. `s` is n-by-n.

Note: Procedure msSvd assumes the name of mfSvd when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfSvd` procedure to compute the singular value decomposition of a square matrix mfArray `a`.

Code

```
program example

use fml
implicit none

type(mfArray) :: a, u, s, v

! Construct a 3-by-3 mfArray a using vertical
! concatenation.
a = (/10.0d0, 20.0d0, 30.0d0/) .vc. &
   (/7.0d0, 8.0d0, 9.0d0/)    .vc. &
   (/5.0d0, 15.0d0, 25.0d0/)

! Compute singular value decomposition of mfArray a
call msSvd(mfOut(u, s, v), a)
```

```
! Display value of a, u, s and v
call msDisplay(a, 'a', u, 'u' ,s, 's', v, 'v')

! Deallocate mfArrays a, u, s, v
call msFreeArgs(a, u, s, v)

end program example
```

Result

```
a  =

  10  20  30
   7   8   9
   5  15  25

 u  =


  -0.7568  -0.1344  -0.6397
  -0.2688  -0.8280   0.4921
  -0.5958   0.5444   0.5905


 s  =


   49.4333    0.0000    0.0000
    0.0000    5.0350    0.0000
    0.0000    0.0000    0.0000


 v  =


  -0.2514  -0.8776   0.4082
  -0.5305  -0.2279  -0.8165
  -0.8095   0.4219   0.4082
```

See Also

Factorization Utilities

mfBalance, msBalance

Perform diagonal scaling to improve eigenvalue accuracy.

Module

```
use fml
```

Syntax

```
type(mfArray) :: t, b, a

b = mfBalance(a)

call msBalance(mfOut(t, b), a)
```

Descriptions

For matrices, procedure `mfBalance` performs diagonal scaling to improve the eigenvalue accuracy.

`b = mfBalance(a)` returns the balanced matrix mfArray `b`.

`call msBalance(mfOut(t, b), a)` returns a similarity transformation `t`, such that `b = mfLDiv(t, a*t)` has as closely as possible, approximately equal row and column norms.

Note: Procedure msBalance assumes the name of mfBalance when used in function call format. Refer to **Section 1.4 Function Naming Convention** of the **User's Guide** for more detail on MATFOR function naming convention.

Example

The following example uses the `mfBalance` procedure to find the balanced matrix mfArray `b` of the original mfArray `a`.

Code

```
program example

use fml
implicit none

type(mfArray) :: a, b

! Construct a 3-by-3 mfArray a using vertical concatenation.
a = (/100.0d0, 200.0d0, 300.0d0/) .vc. &
   (/4.0d0, 5.0d0, 6.0d0  /)    .vc. &
   (/7.0d0, 8.0d0, 9.0d0  /)

! Compute the balanced matrix b of mfArray a
b = mfBalance(a)

! Display values of a and b
call msDisplay(a, 'a', b, 'mfBalance(a)')

! Deallocate mfArrays a and b
call msFreeArgs(a, b)
```

```
end program example
```

Result

```
a  =

  100  200  300
    4    5    6
    7    8    9

 mfBalance(a)  =

   100.0000   25.0000   37.5000
    32.0000    5.0000    6.0000
    56.0000    8.0000    9.0000
```

See Also

mfEig

CHAPTER 8

MATFOR Visualization Routines

Windows Frame and Figure

Figure

`msFigure`	Create figure in Graphics Viewer.
`msCloseFigure`	Close figure in Graphics Viewer.
`mfFigureCount`	Number of figures in graphics viewer.

Window Frame

`mfWindowCaption`	Graphics Viewer title.
`mfWindowSize`	Graphics Viewer frame size.
`mfWindowPos`	Graphics Viewer frame position.

Display

`msGDisplay`	Display mfArray data on a MATFOR Data Viewer.
`msDrawNow`	Draw all pending graphs in current figure.
`msViewPause`	Pause program execution.

Recording

`mfRecordStart`	Record animation as avi file or MATFOR mfa file.
`mfRecordEnd`	Stop recording animation.

`mfExportImage`	Save figure graph as picture file.

Subplot

Plot Creation and Control

`mfSubplot`	Create subplot in active figure.
`msClearSubplot`	Remove all draws in specified subplot.
`msHold`	Hold previous graph on plot space.
`mfIsHold`	Return status of plot space.

Plot Annotation and Appearance

`mfTitle`	Graph title.
`mfXLabel`	X-axis label.
`mfYLabel`	Y-axis label.
`mfZLabel`	Z-axis label.
`mfText`	2-D text annotation
`mfAnnotation`	3-D text annotation
`msShading`	Shading methodology of surface object.
`msColorbar`	Display color scale.
`msColormap`	Colormap type.
`mfColormapRange`	Range of colormap.
`mfBackgroundColor`	Background color of plot space.

Axis Control

`mfAxis`	Manipulate axis object.
`msAxisWall`	Manipulate the axis wall object.
`msAxisGrid`	Display grid lines.

Object & Camera

Object Manipulation

`msObjRotateX`	Rotate draw object in degrees about the x-axis using the right hand rule.

`msObjRotateY`	Rotate draw object in degrees about the y-axis using the right hand rule.
`msObjRotateZ`	Rotate draw object in degrees about the z-axis using the right hand rule.
`msObjRotateWXYZ`	Rotate draw object in degrees about an arbitrary axis.
`mfObjScale`	Scale of draw object.
`mfObjPosition`	Position of draw object in world coordinates.
`mfObjOrigin`	Origin of draw object.
`mfObjOrientation`	Return WXYZ orientation of draw object.

Camera Manipulation

`mfView`	Viewpoint specification.
`mfCamZoom`	Zoom in/out.
`mfCamPan`	Move the camera horizontally and vertically.
`mfCamProj`	Set the camera projection mode.

Graphics

Linear Graphs

`mfPlot`	2-D linear plot.
`mfPlot3`	3-D linear graphs.
`mfRibbon`	3-D ribbons.
`mfTube`	3-D tubes.

Surface Graphs

`mfSurf`	Surface plot.
`mfMesh`	Mesh plot.
`mfSurfc`	Combined plot of suface and contour3.
`mfMeshc`	Combined plot of mesh and contour3.
`mfPColor`	Pseudocolor plot of a matrix.
`mfContour`	2-D contour.
`mfContour3`	3-D contour.
`mfSolidContour`	2-D solid contour.
`mfSolidContour3`	3-D solid contour.

`mfOutline`	Wireframe outline corners.
`mfIsoSurface`	3-D plot isovalue surface from volume data.

Slice Graphs

`mfSliceXYZ`	Display orthogonal slice-planes through volumetric data.
`mfSliceIJK`	Display orthogonal slice-planes through volumetric data.
`mfSlicePlane`	Display orthogonal slice-planes along i, j or k indices.
`mfGetSliceXYZ`	Display orthogonal slice-planes along arbitrary direction.
`mfGetSliceIJK`	Retreive orthogonal slice-plane(s) through volumetric data.
`mfGetSlicePlane`	Retreive orthogonal slice-plane(s) along i, j or k indices.

Streamline Graphs

`mfStreamLine`	Streamlines from 2-D or 3-D vector data.
`mfStreamDashedLine`	Stream of dashed lines from 2-D or 3-D vector data.
`mfStreamRibbon`	Stream of ribbons in ribbon from 2-D or 3-D vector data.
`mfStreamTube`	Stream of tubes from 2-D or 3-D vector data.

Triangular Surface Graphs

`mfTriSurf`	Polygonal surface plot.
`mfTriMesh`	Polygonal mesh plot.
`mfTriContour`	Contour on polygonal plot.
`mfPatch`	Add patch on 2-D or 3-D coordinates.

Unstructured Grids

`mfTetSurf`	Polyhedral surface plot.
`mfTetMesh`	Polyhedral mesh plot.

`mfTetContour`	Contour on polyhedral plot.
`mfTetIsoSurface`	Poyhedral isosurface plot.

Unstructured Point Set

`mfPoint`	Display input points in 3-D space.
`mfDelaunay`	2-D Delaunay triangulation of input points.
`mfDelaunay3`	3-D Delaunay triangulation of input points.
`mfGetDelaunay`	2-D Delaunay triangulation of input points.
`mfGetDelaunay3`	3-D Delaunay triangulation of input points.

Velocity Vectors

`mfQuiver`	3-D velocity vectors.
`mfQuiver3`	3-D velocity vectors.

Image

`mfImage`	Display image file.
`mfImRead`	Read in image file.
`mfImWrite`	Write to image file.

3-D Objects

`mfMolecule`	Draw stick and ball model of molecules.
`mfSphere`	Draw stick and ball model of molecules.
`mfCube`	Draw a sphere.
`mfCylinder`	Draw Cube.
`mfCone`	Draw Cylinder.
`mfAxisMark`	3-directional mark on arbitrary point.

Property Setting

`msGSet`	Set property of specified graph.

`msDrawMaterial`	Set draw object's transparency reflectance, ambient reflectance, diffuse reflectance and specular reflectance.
`msDrawTexture`	Texture mapping.
`mfIsValidDraw`	Check validity of draw object.
`mfGetCurrentDraw`	Return handle of current draw object.
`msRemoveDraw`	Remove draw object from plot space.
`msSetDrawName`	Name of draw object.

Simple GUI

`msShowMessage`	Pop up message dialog box.
`mfInputString`	Pop up string insertion dialog box.
`mfInputValue`	Pop up value insertion dialog box.
`mfInputVector`	Pop up vector insertion dialog box.
`mfInputMatrix`	Pop up matrix insertion dialog box.
`mfFileDialog`	Pop up file open dialog box.
`mfInputYesNo`	Pop up yes-no query dialog box.

Figure

mfFigure, msFigure

Create figure in Graphics Viewer.

Module

```
use fgl
```

Syntax

```
call msFigure(figure_id)
call msFigure(figure_name)
call msFigure(figure_id, figure_name)
id = mfFigure()
```

Descriptions

Procedure `mfFigure` creates a new figure with ID specified in argument `figure_id` and with name specified in argument `figure_name.` The ID and the name of the figure is displayed on the figure tab.

`call msFigure()`

- If argument `figure_id` is not provided, it creates a new figure with an automatically selected figure ID.

`id = mfFigure(...)`

- If the procedure is used in function format, it will return the figure ID once the figure is created.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: x, y1, y2, y3
integer :: num

x = mfLinspace(-MF_PI, MF_PI, 100)
y1 = mfSin(x)
y2 = mfCos(x)
y3 = mfLinspace(-1, 1, 100)

! Create figure 1 and plot x, y1
call msFigure(1)
call msPlot(x, y1)
call msAxis(mf((/-MF_PI, MF_PI, -1.0d0, 1.0d0/)))

! Returns number of figures
num = mfFigureCount()
call msDisplay(mf(num), 'mfFigureCount()')
call msViewPause()
```

```
! Create figure 2 and plot x, y2
call msFigure(2)
call msPlot(x, y2, 'r')
call msAxis(mf((/-MF_PI, MF_PI, -1.0d0, 1.0d0/)))

! Returns number of figures
num = mfFigureCount()
call msDisplay(mf(num), 'mfFigureCount()')
call msViewPause()

! Create figure 3 and plot x, y3
call msFigure(3)
call msPlot(x, y3, 'g')
call msAxis(mf((/-MF_PI, MF_PI, -1.0d0, 1.0d0/)))

! Returns number of figures
num = mfFigureCount()
call msDisplay(mf(num), 'mfFigureCount()')
call msViewPause()

! Close figure 2
call msCloseFigure(2)

! Returns number of figures
num = mfFigureCount()
call msDisplay(mf(num), 'mfFigureCount()')
call msViewPause()

call msFreeArgs(x, y1, y2, y3)

end program example
```

See Also

msCloseFigure, mfFigureCount

msCloseFigure

Close figure in Graphics Viewer.

Module

```
use fgl
```

Syntax

```
call msCloseFigure(figure_id)
```

Descriptions

Procedure `msCloseFigure` closes the target figure specified by argument `figure_id.`

See Also

mfFigure, mfFigureCount

mfFigureCount

Number of figures in graphics viewer.

Module

```
use fgl
```

Syntax

```
num = mfFigureCount()
```

Descriptions

Procedure `mfFigureCount` returns the number of figures that are in Graphics Viewer.

See Also

mfFigure, msCloseFigure

Window Frame

mfWindowCaption, msWindowCaption

Graphics Viewer title.

Module

```
use fgl
```

Syntax

```
title = mfWindowCaption()
call msWindowCaption(title)
```

Descriptions

Procedure `mfWindowCaption` sets the caption on the top window panel of the Graphics Viewer. The default caption is "MATFOR 3.0".

`title = mfWindowCaption()`

It can also be used as an inquiry procedure if given no argument.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: x, y

x = mfLinspace(0, 2*MF_PI, 101)
y = mfSin(x)

call msPlot(x, y)
call msWindowCaption('Example Change WindowCaption to 2D Plot')

call msViewPause()

call msFreeArgs(x, y)

end program example
```

See Also

mfWindowSize, mfWindowPos

mfWindowSize, msWindowSize

Graphics Viewer frame size.

Module

```
use fgl
```

Syntax

```
size = mfWindowSize()
call msWindowSize(width, height)
```

Descriptions

Procedure `mfWindowSize` sets the frame size of the Graphics viewer.

`call msWindowSize(width, height)`

Arguments `width` and `height` can be integer scalars or mfArrays containing integer scalars.

`size = mfWindowSize()`

It can be used as inquiry function for the frame size if no argument is specified. The return argument `size` is a double vector in the format of [width, height].

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: x, y

x = mfLinspace(0, 2*MF_PI, 101)
y = mfSin(x)

call msPlot(x, y)
call msWindowSize(600, 600)
call msViewPause()

call msFreeArgs(x, y)

end program example
```

See Also

mfWindowPos, msWindowPos

Graphics Viewer frame position.

Module

```
use fgl
```

Syntax

```
pos = mfWindowPos();
call msWindowPos(x, y);
```

Descriptions

Procedure `mfWindowSize` sets the frame position of the Graphics viewer.

`call msWindowPos(x, y);`
Arguments `x` and `y` can be integer scalars or mfArrays containing integer scalars.

`pos = mfWindowPos()`
It can be used as inquiry function for the frame position if no argument is specified. The return argument `pos` is a integer vector in the format of [x, y].

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: x, y

x = mfLinspace(0, 2*MF_PI, 101)
y = mfSin(x)

call msPlot(x, y)
call msWindowPos(220, 0)
call msViewPause()

call msFreeArgs(x, y)

end program example
```

See Also

mfWindowSize

Display

msGDisplay

Display mfArray data on a MATFOR Data Viewer.

Module

```
use fgl
```

Syntax

```
call msGDisplay(x[, 'name'])
call msGDisplay(x, 'name'[, x1, 'name1', ...])
```

Descriptions

Procedure `msGDisplay` displays your mfArray data on a Data Viewer.

You can output single or multiple mfArray data to the Data Viewer.

```
call msGDisplay(x)
call msGDisplay(x , 'name')
```

- Display data of mfArray `x` on Data Viewer.
- Character string 'name' specifies the label on the spreadsheet tab displaying data of `x`.

```
call msGDisplay(x, 'name', x1, 'name1', ...)
```

- Display multiple datasets `x, x1, ...`, on the Data Viewer, labeled with 'name', 'name1', ... respectively. The arguments must be specified in pairs.

Example

Code

```
program example

use fgl
use fml
implicit none

type (mfArray):: x, y, z
integer :: i

x = (/(i, i=1, 10)/)
y = (/(i, i=-10, -1)/)
z = mfComplex(x,y)

! Next, display the data of X, Y and Z on a MATFOR Data
! Viewer.
call msGDisplay(x, 'x', y, 'y', z, 'z')

! Pause program to display Data Viewer
call msViewPause

! Deallocate mfArrays
call msFreeArgs(x, y, z)
```

```
end program example
```

See Also

msDisplay

msDrawNow

Draw all pending graphs.

Module

```
use fgl
```

Syntax

```
call msDrawNow()
```

Descriptions

Procedure `msDrawNow` draws all pending graphics on the current Figure.

The procedure is used mainly for animation. It does not pause program execution. As a result, the Figure is displayed, updated, and flushed almost immediately. Animation results when `msDrawNow` is used within a do loop in which the graphics object is continuously being updated.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: x, y, h
integer :: i

x = mfLinspace(-MF_PI, MF_PI, 50)
y = mfSin(x)

! Create an initial copy of the graph to be animated.
! This is recommended as you can obtain the current
! graphics handle and use erase mode for the animation.
h = mfPlot(x, y)
call msAxis(mf((/-MF_PI, MF_PI, -1d0, 1d0/)))

! Use a Do Loop to animate the sin(x) curve. Note,
! msGSet continuously updates the specified data and
! sleep slows down the program execution.
do i = 1, 100
   y = mfSin(x+0.1d0*i)
   call msGSet(h, 'ydata', y)
   call msDrawNow()
end do

! Pause the program to continue displaying the
! Graphics Viewer after the Do Loop ends.
call msViewPause()

! Deallocate mfArrays
call msFreeArgs(x, y, h)

end program example
```

See Also

msViewPause

msViewPause

Pause program execution.

Module

```
use fgl
```

Syntax

```
call msViewPause()
```

Descriptions

Procedure `msViewPause` pauses program execution for graphical display.
Use this procedure wherever you wish to pause a program to visualize your data. To continue your program execution, you can either close the Graphics Viewer, or click on the "Continue" button located at the top right corner of the Graphics Viewer.

You must add at least one line of `call msViewPause()`after each set of graphical creation routines. If the procedure were left out, you would only see a flash as the Graphics Viewer is opened and closed almost immediately by the program.

See Also

Recording

msRecordStart, msRecordEnd

Record animation as avi file, MATFOR mfa file or bitmap files.

Module

```
use fgl
```

Syntax

```
call msRecordStart(filename[, property1, value1, ...])
call msRecordEnd()
```

Descriptions

Procedures `msRecordStart` and `msRecordEnd` are MATFOR build-in procedures for recording visualized data as avi animation files or mfa files.

`call msRecordStart(filename, property1, value1, property2, value2, property3, value3)`

- Records the animations as avi files or MATFOR mfa files.
- Argument `filename` can take the following values.

Value	Meaning
"*.avi"	Example: "filename.avi". Record the current animation in an avi file. Avi or video recording uses frame capturing method to capture the animation playing on the current Graphics Viewer. The recorder automatically detects the type of video compression utilities available in your system and presents a drop-list for you to choose.
"*.mfa"	Example: "filename.mfa". Record the current animation as an mfa file. The mfa file format is a MATFOR-specific record of all data used for generating the current animation on the Graphics Viewer. You can playback the animation by using MATFOR mfPlayer at a later time. Using mfPlayer, you can perform graphical manipulations on the animation, such as zoom in/out, rotation, colormap adjustment, etc.
"*.bmp"	Example: "filename.bmp". Save each frame of the animation into a bitmap file. The name of each bitmap file is set to be the input file name followed by four digits starting from 0000 counting up. In the example, the names of the first two bitmap files would be "filename0000.bmp" and "filename0001.bmp".
"*.tif"	Example: "filename.tif". Save each frame of the animation into a TIFF file. The naming method of the saved files is similar to saving as bitmap files.

- The optional arguments property1, property2 and property3 and the corresponding arguments value1, value2 and value3 can take the following values.

Property	Meaning
"framerate"	Specify the number of frames captured per second. The argument applies only when the animation file is saved in avi format. By default, MATFOR records avi file at 15 frames per second. The recommended range is 5 to 30 frames per second. The higher the frame rate, the faster the frames are looped through. Likewise, smaller frame rate slows down the animation.
"width"	Specify the width of the captured figure frame. The argument applies only when the animation is saved in avi format or bmp format. Without specifying the argument, the captured figure frame will have the exact same width as the displaying figure frame.
"height"	Specify the height of the captured figure frame. The argument applies only when the animation is saved in avi format or bmp format. Without specifying the argument, the captured figure frame will have the exact same height as the displaying figure frame.

```
call msRecordEnd()
```

- Stop the recording.

The general syntax of the recording procedures is as follows:

```
call msRecordStart('animation.avi')
```

or

```
call msRecordStart('animation.bmp')
```

------- <animation codes>

```
call msRecordEnd()
```

Example

Code

```
program example
```

```
use fml
use fgl
implicit none

type(mfArray) :: a, b, c, x, y, z, indxi, indxj, h
integer :: i

a = mfLinspace(-3, 7, 51)
b = mfLinspace(-2, 8, 51)
call msMeshgrid(mfout(x, y), a, b)

! Next, initialize indxi and indxj using Meshgrid.
! Compute z using indxi and indxj.
! Note, a 'd0' is added to the integers, to ensure
! double precision. MATFOR uses only double precision
! data.
c = mfColon(1, 51)
call msMeshgrid(mfout(indxi, indxj), c)
z = 3d0*mfSin((indxi+1)/10d0)*mfCos((indxj+1)/10d0) &
   + 2d0*mfSin((indxi+indxj)/10d0)

! Plot a mesh grid using mfArray x, y and z for the grid
! intersections.
h = mfMesh(x, y, z)

! Start record of an animation using avi file format
! Records scarf.avi in the Debug directory
call msRecordStart('scarf.avi')

! Animate the mesh using a do loop.
do i = 1, 10
    z = 3d0*mfSin((indxi+i+1)/10d0)*mfCos((indxj+1-i)/10d0) &
       + 2d0*mfSin((indxi+indxj+i)/10d0)

! Update z
  call msGSet(h, 'zdata', z)
! Update Graphics Viewer
  call msDrawNow()
end do

! end video record
call msRecordEnd()

! Pause to display the graph.
call msViewPause()

! Deallocate mfArray
call msFreeArgs(a, b, c, x, y, z, indxi, indxj)

end program example
```

See Also

msGSet, mfFigure

msExportImage

Save figure graph as picture file.

Module

```
use fgl
```

Syntax

```
call msExportImage(filename[, width, height])
```

Descriptions

Procedure `msExportImage` saves the graph(s) in current figure as a picture file. You can choose the picture format by specifying the extension in argument `filename.` For example, `"filename.bmp"` would save it as a bitmap file. The supported formats are: BMP, JPEG, TIFF, PS and PNG.

See Also

Plot Creation and Control

mfSubplot, msSubplot

Create subplot in active figure.

Module

```
use fgl
```

Syntax

```
subplot_id = mfSubplot(row, col, index)
```

Descriptions

Procedure `mfSubplot` divides the plot space of a Graphics Viewer into m-by-n rectangular sub-plot spaces, as specified by the `rol, col` arguments.

Each sub-plot space is numbered row-wise so that a subplot space at position (1,2) is numbered 2 and (2,2) is numbered 4.

The current subplot is set to the subplot with the subplot number `index.` Any subsequent operations will be performed on the current subplot.

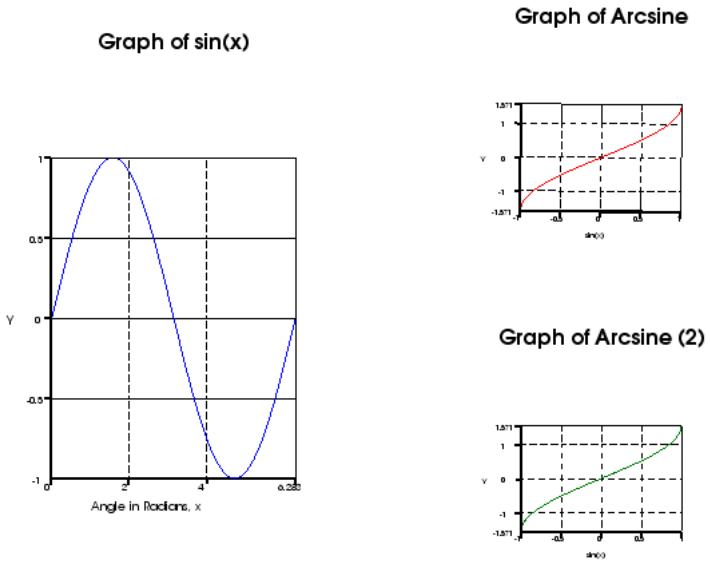
Example

Code

```
program Example_msSubplot

use fml
use fgl
implicit none

type(mfArray) :: x, y1, y2

x = mfLinspace(0, 2*MF_PI, 101)
y1 = mfSin(x)
y2 = mfASin(y1)

! Divide the plotting space into 1 on left 2 on right sub-plot spaces,
! and specify the subplot space p =1 or left sub-plot as current.
call msSubplot("5, 5[5, 5]", 1)
! Plot and label the graph.
call msPlot(x, y1)
call msAxis(mf((/0d0, 2*MF_PI, -1d0, 1d0/)))
call msTitle('Graph of sin(x)')
call msXlabel('Angle in Radians, x')
call msCamZoom(0.8d0)

! Next, specify subplot space p=2, or the right-top subplot
! space as current.
call msSubplot("", 2)
! Again, plot and label the graph.
call msPlot(y1, y2, 'r')
call msAxis(mf((/-1d0, 1d0, -MF_PI/2, MF_PI/2/)))
call msTitle('Graph of Arcsine')
call msXlabel('sin(x)')
call msCamZoom(0.8d0)
```

```
! Finally, specify subplot space p=3, or the right-bottom subplot
! space as current.
call msSubplot("", 3)
! Again, plot and label the graph.
call msPlot(y1, y2, 'g')
call msAxis(mf((/-1d0, 1d0, -MF_PI/2, MF_PI/2/)))
call msTitle('Graph of Arcsine (2)')
call msXlabel('sin(x)')
call msCamZoom(0.8d0)

! Pause the program to display the graphs.
call msViewPause()

! Deallocate the mfArrays
call msFreeArgs(x, y1, y2)

end program Example_msSubplot
```

Result



See Also

msClearSubplot

msClearSubplot

Remove all draws in specified subplot.

Module

```
use fgl
```

Syntax

```
call msClearSubplot()
call msClearSubplot(subplot_id)
```

Descriptions

Procedure `msClearSubplot` removes all draws in current or specified subplot if argument `subplot_id` is specified.

See Also

mfSubplot

msHold

Hold previous graph on plot space.

Module

```
use fgl
```

Syntax

```
call msHold(mode)
```

Descriptions

Procedure `msHold` holds the current graph in plot space so that it would not be overwritten by the next graph creation. Subsequent creations of graphs would be appended one after another in the plot space.

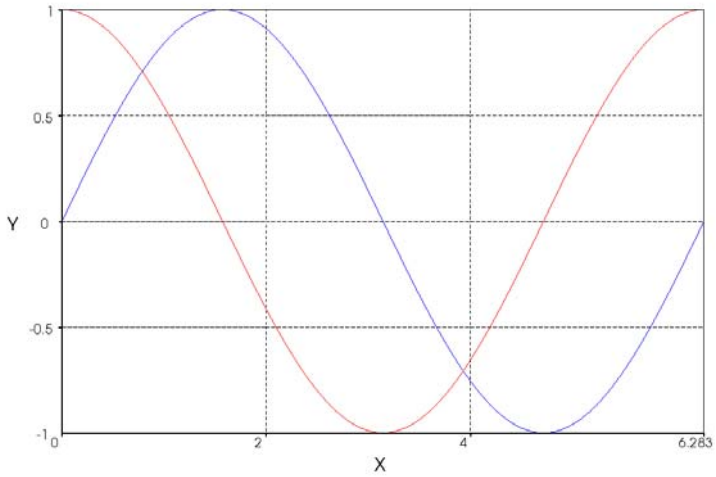
Argument `mode` is a string containing "on" or "off".

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: x, y1, y2, a

x = mfLinspace(0, 2*MF_PI, 100)
y1 = mfSin(x)
y2 = mfCos(x)

! Plot y1 = mfSin(x)
call msPlot(x, y1)
call msAxis(mf((/0d0, 2*MF_PI, -1d0, 1d0/)))

! Hold the figure for plotting
call msHold('on')

! Plot y2 = mfCos(x) to the same figure
call msPlot(x, y2, 'r')

call msHold('off')
! Pause the figure for viewing
call msViewPause()

! Deallocates the mfArrays
call msFreeArgs(x, y1, y2)

end program example
```

Result



See Also

mfIsHold

mfIsHold

Return status of plot space.

Module

```
use fgl
```

Syntax

```
status = mfIsHold()
```

Descriptions

Procedure `mfIsHold` returns the status of current plot space. The output is an mfArray containing logical data. It returns true if the plot space is held on, false otherwise.

See Also

msHold

Plot Annotation and Appearance

mfTitle, mfXLabel, mfYLabel, mfZLabel

Label the axis objects.

Module

```
use fgl
```

Syntax

```
title = mfTitle()
xlabel = mfXLabel()
ylabel = mfYLabel()
zlabel = mfZLabel()

call msTitle(title[, color][, font_size])
call msXLabel(xlabel)
call msYLabel(ylabel)
call msZLabel(zlabel)
```

Descriptions

Procedures `mfTitle, mfXLabel, mfYLabel` and `mfZLabel` annotate a graph with title, x-axis label, y-axis label and z-axis label respectively. By default, x-axis is labeled as "x", y-axis is labeled as "y" and z-axis label is labeled as "z".

You can also annotate the graph through the Axis menu ->Title, XLabel, YLabel and ZLabel functions of the Graphics Viewer.

```
title = mfTitle()
xlabel = mfXLabel()
ylabel = mfYLabel()
zlabel = mfZLabel()
```

They can also be used as inquiry procedures to retrieve the title and labels that are set by users.

```
call msTitle(title, color, font_size)
```

- Set the rgb color code and the font size of the title by specifying arguments `color` and `font_size`. The rgb color code is specified as [r, g, b] where 0 < r, g, b < 1.

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: x, y
```

```
x = mfLinspace(0, 2*MF_PI, 51)
y = mfSin(x)

! Plot the x, y curve using msPlot.
call msPlot(x, y, 'rx')

! Annotate the graph with title, x-axis label and y-axis label.
call msTitle('Graph of sin(x)')
call msXLabel('x in radians')
call msYLabel('sin(x)')

! Pause program execution.
call msViewpause

! Deallocate mfArray
call msFreeArgs(x, y)

end program example
```

See Also

mfCaption

mfText, msText

2-D text annotation.

Module

```
use fgl
```

Syntax

```
handle = mfText(text[, loc][, color][, font_size])
```

Descriptions

Procedure `mfText` places a two-dimensional text on the current subplot.

`call msText(text, loc, color, font_size)`

- Argument `text` can be a string or an mfArray containing a string.
- Argument `loc` is a 1-by-2 vector in the format [m, n]. Each element contains a value ranging from 0 to 1. Specifying m as 0 would place the text annotation to the left-most position of the subplot window and specifying n as 0 would place the text annotation on the bottom of the subplot window. For example, the vector `[0, 0]`would place the text annotation on the bottom-left corner of the subplot and vector [0.5, 0.5] would place the text annotation in the center of the plot space.

- You can set the color and the font size through arguments `color` and `font_size.` Argument `color` contains the rgb color code which is specified as [r, g, b] where 0 < r, g, b < 1.

`h = mfText(...)`

- Handle `h` retrieves a handle to the text annotation created by `mfText(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the text annotation through handle `h` with procedure `msGSet.`

The properties available are:

1. text 2. location 3. color 4. font_size 5. bold 6. italic 7. shadow 8. just (justification): "left", "right" or "center"

See Also

mfAnnotation, msAnnotation

3-D text annotation.

Module

```
use fgl
```

Syntax

```
handle = mfAnnotation(text[, loc][, color][, font_size])
```

Descriptions

Procedure `mfAnnotation` places a three-dimensional text annotation on the current subplot.

```
call msAnnotation(text, loc, color, font_size)
```

- Argument `text` can be a string or an mfArray containing a string.
- Argument `loc` is a 1-by-3 vector in the format [m, n, p]. Each element contains a value ranging from 0 to 1. Specifying m as 0 would place the text annotation to the left-most position of the subplot window and specifying n as 0 would place the text annotation on the bottom of the subplot window. For example, the vector `[0, 0, 0]`would place the text annotation on the bottom-left corner of the subplot window and vector [0.5, 0.5] would place the text annotation in the center of the plot space.

```
h = mfAnnotation(...)
```

- Handle `h` retrieves a handle to the text annotation created by `mfAnnotation(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the text annotation through handle `h` with procedure `msGSet.`

The properties available are:

1. text 2. location 3. color 4. font_size 5. bold 6. italic 7. shadow // 8. just (justification): "left", "right" or "center", not open 9. offset, (x,y) 2x1 array

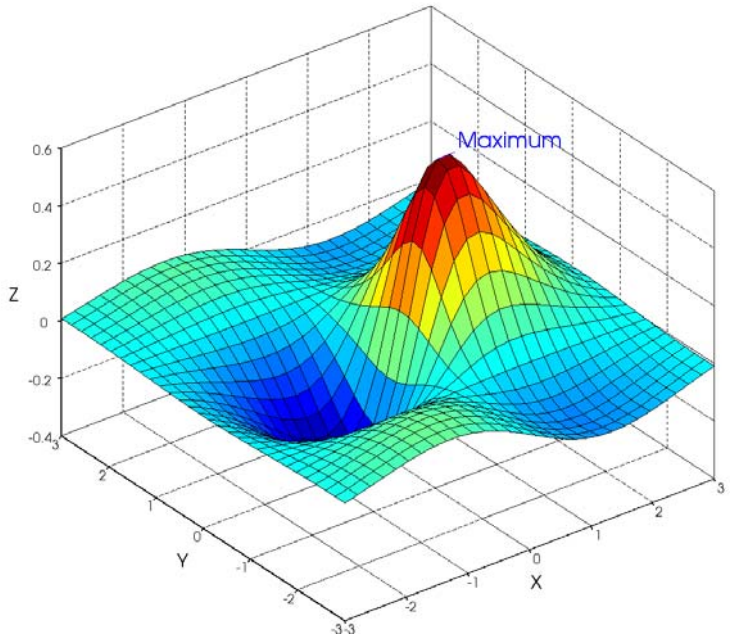
Example

Code

```
program example
```

```
use fml
use fgl
implicit none

type(mfArray) :: m, x, y, z, cm

m = mfLinspace(-3, 3, 30)
call msMeshGrid(mfOut(x, y), m)
z = mfSin(x) * mfCos(y) / ( x*(x-0.5d0) + (y+0.5d0)*y + 1)

call msSurf(x, y, z)
call msAxis(-3.0d0, 3.0d0, -3.0d0, 3.0d0, -0.4d0, 0.6d0)
call msAnnotation('Maximum', mf((/0.7241d0, -0.1034d0, 0.5877d0/)),
mf((/0, 0, 1/)))
call msViewPause()

call msFreeArgs(m, x, y, z)

end program example
```

Result



See Also

mfText

msShading

Shading methodology of surface object.

Module

```
use fgl
```

Syntax

```
call msShading(mode)
call msShading(draw_id, mode)
```

Descriptions

Procedure `msShading` sets the shading mode for all draws in plot space.

`call msShading(mode)`

- Specify shading type for procedures `mfSurf` and `mfMesh.` The options available are listed in the table below.

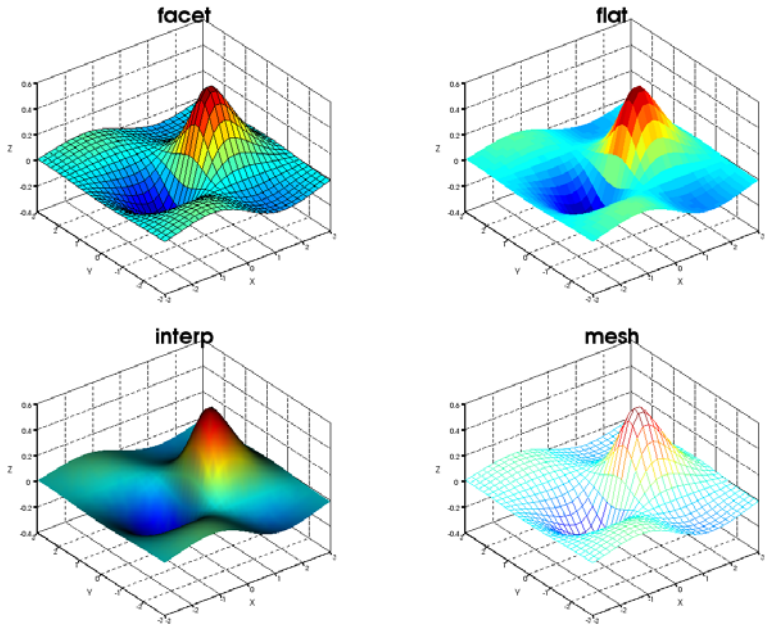
Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: a, x, y, z, h

a = mfLinspace(-3, 3, 30)
call msMeshGrid( mfOut(x, y), a)
z = mfSin(x) * mfCos(y) / ( x*(x-0.5d0) + (y+0.5d0)*y + 1)

call msSubplot(2, 2, 1)
call msSurf(x, y, z)
call msShading('facet')
call msTitle('facet')
call msCamZoom(1.2d0)

call msSubplot(2, 2, 2)
call msSurf(x, y, z)
call msShading('flat')
call msTitle('flat')
call msCamZoom(1.2d0)

call msSubplot(2, 2, 3)
call msSurf(x, y, z)
call msShading('interp')
call msTitle('interp')
call msCamZoom(1.2d0)

call msSubplot(2, 2, 4)
call msSurf(x, y, z)
call msShading('mesh')
call msTitle('mesh')
```

```
call msCamZoom(1.2d0)
call msViewPause()
call msFreeArgs(a, x, y, z)
end program example
```

Result



See Also

msColorbar

Display color scale.

Module

```
use fgl
```

Syntax

```
call msColorbar(mode)
call msColorbar(property, value)
```

Descriptions

Procedure `msColorbar` controls the colorbar through specification of argument `mode.`

A colorbar displays the current color map and acts as a color scale showing the relationship between graphics data and color. In the case of a surface object, it shows the relationship between color and height of the surface object. You can also select the colorbar setting from the menu and toolbar functions of the Graphics Viewer.

```
call msColorbar(mode)
call msColorbar(property, value)
```

Argument `mode` can be:

mode	Meaning
`"on"`	Display a colorbar on the Graphics Viewer.
`"off"`	Hide the colorbar.
`"vert"`	Display a vertical colorbar.
`"horz"`	Display a horizontal colorbar.

Argument `property` can be:

Property	Meaning
`label_count`	Number of labels displayed on the colorbar.
`label_color`	Color of the colorbar labels. Corresponding argument `value` is an mfArray containing a string that specifies the color, e.g.

	“y”, or a 1-by-3 mfArray contains the rgb codes.

See Also

msColormap

Colormap type.

Module

```
use fgl
```

Syntax

```
call msColormap(type)
call msColormap(colormap)
```

Descriptions

Procedure `msColormap` specifies the colormap type used for drawing surface objects.

`call msColormap(type)`

- Argument `type` specifies the type of colormap used. The argument can be an mfArray containing a string specifying the type of colormap or a character string. MATFOR provides the following types of colormap:

Value	Meaning
"jet"	Range from blue to red, and pass through the colors cyan, yellow, and orange.(default)
"gray"	Return a linear grayscale colormap.
"hot"	Vary smoothly from black, through shades of red, orange, and yellow, to white.
"cool"	Vary smoothly from cyan to magenta.
"copper'	Vary smoothly from black to bright copper.
"hsv"	Vary the hue component of the hue-saturation-value color model.
"spring"	Consist of colors that are shades of magenta and yellow.
"summer"	Consist of colors that are shades of green and yellow.
"autumn"	Vary smoothly from red, through orange, to yellow.

"winter"	Consist of colors that are shades of blue and green.

`call msColormap(colormap)`

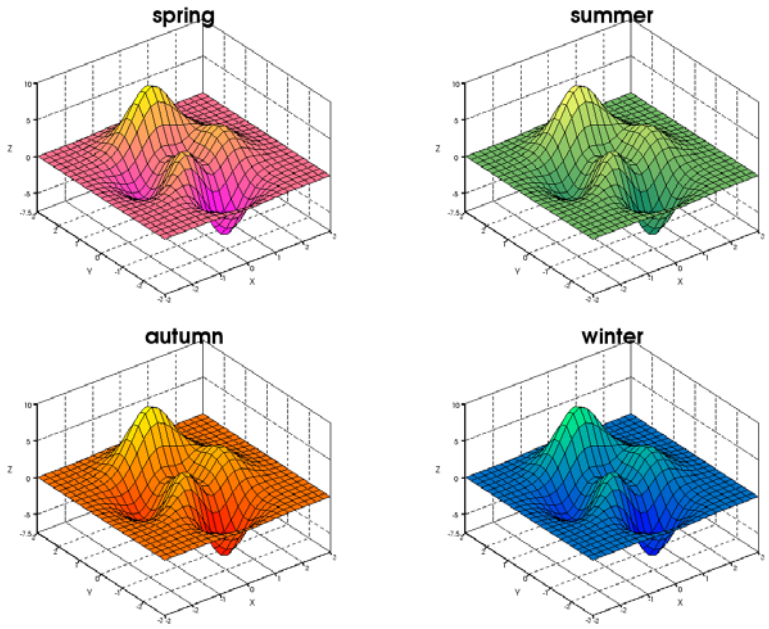
- Allows users to customize the colormap through the argument `colormap` which is a m-by-3 matrix consists of m sets of rgb color code. The rgb color code is specified as [r, g, b] where 0 < r, g, b < 1.

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: x, u, v, z, h

x = mfLinspace(-3, 3, 25)
call msMeshgrid(mfout(u, v), x)
z = 3*((1-u)**2)*mfExp(-(u**2)-((v+1)**2))           &
    - (10*(u/5-(u**3)-(v**5))*mfExp(-(u**2)-(v**2)))  &
    - mfExp(-(u+1)**2-v**2)/3

call msSubPlot(2, 2, 1)
call mstitle('spring')
call msSurf(u, v, z)
h = mfColorMap('spring')
call msCamZoom(1.2d0)

call msSubPlot(2, 2, 2)
call mstitle('summer')
call msSurf(u, v, z)
h = mfColorMap('summer')
call msCamZoom(1.2d0)

call msSubPlot(2, 2, 3)
call mstitle('autumn')
call msSurf(u, v, z)
h = mfColorMap('autumn')
call msCamZoom(1.2d0)

call msSubPlot(2, 2, 4)
call mstitle('winter')
call msSurf(u, v, z)
h = mfColorMap('winter')
call msCamZoom(1.2d0)

call msViewPause()

call msFreeArgs(x, u, v, z)

end program example
```

Result



See Also

msColorbar

mfColormapRange, msColormapRange

Range of color map.

Module

```
use fgl
```

Syntax

```
range = mfColormapRange()
call msColormapRange(min, max)
call msColormapRange([min, max])
call msColormapRange("auto")
```

Descriptions

Procedure `mfColormapRange` sets the range of color map.

```
call msColormapRange(min, max)
call msColormapRange([min, max])
```

- Arguments `min` and `max` specify the minimum and maximum of the range.
- The range can also be input as a vector containing the min and max of the range.

```
range = mfColormapRange()
```

- It can also be used as an inquiry procedure to retrieve the color map range in the format of [min, max] if no argument is specified.

See Also

mfBackgroundColor, msBackgroundColor

Background color of plot space.

Module

```
use fgl
```

Syntax

```
colorCode = mfBackgroundColor()
call msBackgroundColor(r, g, b)
call msBackgroundColor(colorCode)
```

Descriptions

Procedure `mfBackgroundColor` set the background color of plot space.

```
call msBackgroundColor(r, g, b)
call msBackgroundColor(colorCode)
```

- Arguments `r, g, b` contain a real number within the range 0 to 1 specifying the rgb code of the background color.
- Argument `colorCode` is a vector containing the rgb color code in the format of [r, g, b] where 0 < r, g, b < 1.

```
colorCode = mfBackgroundColor()
```

- Retrieves the color code of current plot space in the format of [r, g, b] where 0 < r, g, b < 1.

As an example, `call msBackgroundColor(1, 1, 1)`sets the background color to white.

See Also

Axis Control

mfAxis, msAxis

Manipulate the axis object.

Module

```
use fgl
```

Syntax

```
xyzrange = mfAxis()

call msAxis(xyzrange)
call msAxis(x_min, x_max, y_min, y_max[, z_min, z_max])
call msAxis(mode)
call msAxis(property, value)
```

Descriptions

Procedure `mfAxis` sets the properties of the x-axis, y-axis and z-axis, such as the range, mode and color. It can also be used an inquiry function for the ranges of the axes.

```
xyzrange = mfAxis()
```

- Retrieves the ranges of the axis objects. The output argument `xyzrange` is a vector in the format [x_min, x_max, y_min, y_max, z_min, z_max].

```
call msAxis(xyzrange)
call msAxis(x_min, x_max, y_min, y_max)
call msAxis(x_min, x_max, y_min, y_max, z_min, z_max)
```

- Sets the ranges of the axis objects. The input data can be provided in two ways. One is through a vector `xyzrange` in which the ranges of the axes objects are specified; whereas the other one is specified in the element-by-element way.
- Argument `xyzrange` is a vector in the format [x_min, x_max, y_min, y_max, z_min, z_max].
- Arguments `x_min`, `x_max`, `y_min`, `y_max`, `z_min`, `z_max` specify the displaying ranges of x-axis, y-axis and z-axis.

```
call msAxis(mode)
call msAxis(property, value)
```

- Sets the mode and property of the axis objects.

Argument `mode` can be:

mode	Meaning

“on”	Display the tick marks, labeling and background of the current axis object.
“off”	Remove the tick marks, labeling and background of the current axis object.
“Normal”	Restore the current axis object to its full size and remove any restrictions on scaling.
“equal”	Use the same aspect ratio for each axis of the axis object. In other words, tick marks of equal increments have the same size on all axes.
“Auto”	Set axes scaling to automatic mode. MATFOR sets the limits, minimum and maximum of each axis based on the extents of the graphs plotted. It is the default setting.

Argument `property` can be:

property	Meaning
“axis_color”	Color of all three axes. Corresponding argument `value` is an mfArray containing a string that specifies the color, e.g. “y”, or a 1-by-3 mfArray contains the rgb codes.
"xaxis_color"	Color of the x-axis. Corresponding argument `value` is an mfArray containing a string that specifies the color, e.g. “y”, or a 1-by-3 mfArray contains the rgb codes.
"yaxis_color"	Color of the y-axis. Corresponding argument `value` is an mfArray containing a string that specifies the color, e.g. “y”, or a 1-by-3 mfArray contains the rgb codes.
“zaxis_color”	Color of the z-axis. Corresponding argument `value` is an mfArray containing a string that specifies the color, e.g. “y”, or a 1-by-3 mfArray contains the rgb codes.
“xaxis_ticks”	Ticks on the x-axis. Corresponding argument `Value` is a vector of integers. Each of the elements corresponds to a tick displayed on x-axis.

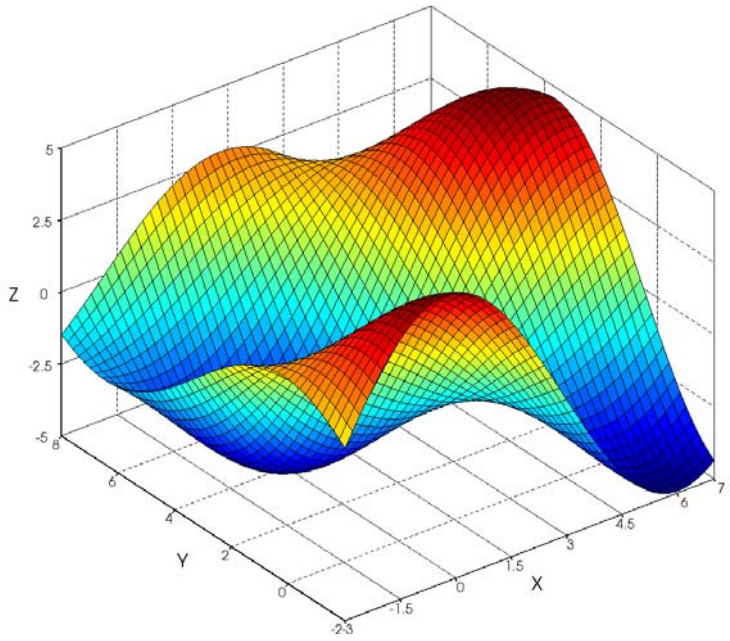
“yaxis_ticks”	Ticks on the y-axis. Corresponding argument `Value` is a vector of integers. Each of the elements corresponds to a tick displayed on y-axis.
“zaxis_ticks”	Ticks on the z-axis. Corresponding argument `Value` is a vector of integers. Each of the elements corresponds to a tick displayed on z-axis.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: a, b, c, x, y, z, indxi, indxj

a = mfLinspace(-3, 7, 51)
b = mfLinspace(-2, 8, 51)
c = mfColon(1, 51)
call msMeshgrid(mfout(x, y), a, b)
call msMeshgrid(mfout(indxi, indxj), c)
z = 3*mfSin((indxi+1)/10)*mfCos((indxj+1)/10) &
    + 2*mfSin((indxi+indxj)/10)

call msSurf(x, y, z)
call msAxis(-3, 7, -2, 8, -5, 5)
call msAxis(mf('xaxis_ticks'), mf((/-3.0d0, -1.5d0, 0.0d0, 1.5d0, 3.0d0,
4.5d0, 6.0d0/)))
call msAxis(mf('yaxis_ticks'), mf((/-2.0d0, 0.0d0, 2.0d0, 4.0d0, 6.0d0,
8.0d0/)))
call msAxis(mf('zaxis_ticks'), mf((/-5.0d0, -2.5d0, 0.0d0, 2.5d0,
5.0d0/)))
call msViewPause()

call msFreeArgs(a, b, c, x, y, z, indxi, indxj)

end program example
```

Result



See Also

msAxisWall, msAxisGrid

msAxisWall

Manipulate the axis wall object.

Module

```
use fgl
```

Syntax

```
call msAxisWall(mode)
call msAxisWall(property, value)
```

Descriptions

Procedure `msAxisWall` sets the color of the three axis-wall objects and switches them on or off. The axis-wall object represents the three axis planes....

`call msAxisWall(mode)`

- Switches the three axis-planes on or off. Argument `mode` is either "on" or "off".

`call msAxisWall(property, value)`

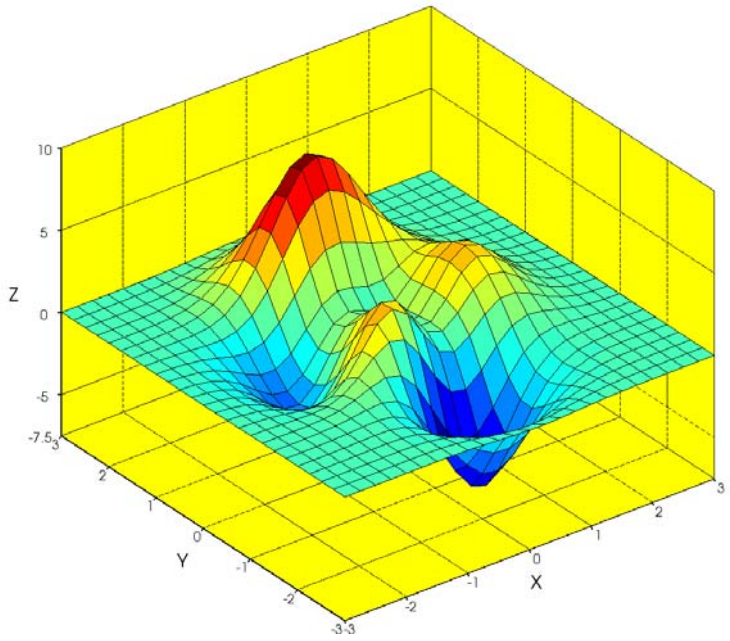
- Sets the color of the axis-wall object. Argument `property` can be a string specified as "color" and argument `value` contains the rgb color code which is specified as [r, g, b] where 0 < r, g, b < 1.

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: u, v, x, z

x = mfLinspace(-3, 3, 25)
call msMeshgrid(mfout(u, v), x)
z = 3*((1-u)**2)*mfExp(-(u**2)-((v+1)**2))           &
    - (10*(u/5-(u**3)-(v**5))*mfExp(-(u**2)-(v**2)))  &
    - mfExp(-(u+1)**2-v**2)/3

call msSurf(u, v, z)
call msAxisWall('color', mf((/1, 1, 0/)))
call msViewPause()

call msFreeArgs(u, v, x, z)

end program example
```

Result



See Also

mfAxis

msAxisGrid

Display grid lines.

Module

```
use fgl
```

Syntax

```
call msAxisGrid(axis, mode)
call msAxisGrid(property, value)
```

Descriptions

Procedure `msAxisGrid` sets the properties of the axis grid objects, such as the width, color and pattern.

`call msAxisGrid(axis, mode)`

- Switch an axis on or off. Argument `axis` can be "xaxis", "yaxis" or "zaxis" which corresponds to the three axes respectively. Argument `mode` is either "on or "off".

`call msAxisGrid(property, value)`

- Sets a property of the axis grid objects.

Argument `property` can be:

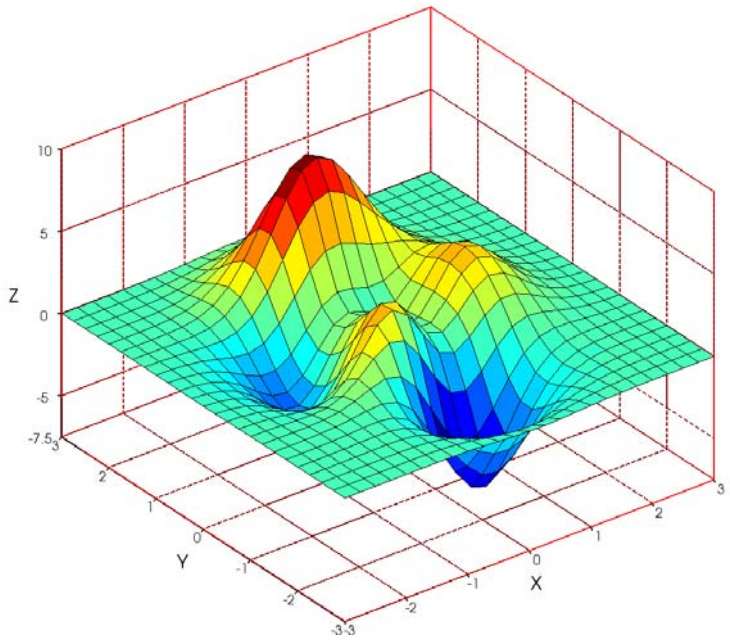
property	Meaning
`“width”`	Line width. Corresponding argument `value` is a scalar integer value.
`“color”`	Line color. Corresponding argument `value` can be an mfArray containing a string specifies the color, e.g. “y”, or a 1-by-3 mfArray contains the rgb codes.
`“pattern”`	Line pattern. Corresponding argument `value` can be an mfArray containing the string that is specified as “`solid", "dashed", "dotted" or "dashdot"`.

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: u, v, x, z

x = mfLinspace(-3, 3, 25)
call msMeshgrid(mfout(u, v), x)
z = 3*((1-u)**2)*mfExp(-(u**2)-((v+1)**2))           &
    - (10*(u/5-(u**3)-(v**5))*mfExp(-(u**2)-(v**2)))  &
    - mfExp(-(u+1)**2-v**2)/3

call msSurf(u, v, z)
call msAxisGrid('width', mf(2))
call msAxisGrid('color', mf((/1, 0, 0/)))
call msAxisGrid('pattern', 'dashed')
call msViewPause()

call msFreeArgs(u, v, x, z)

end program example
```

Result



See Also

mfAxis, msAxisWall

Object Manipulation

msObjRotateX, msObjRotateY, msObjRotateZ

Rotate draw object in degrees about the x-axis, y-axis and z-axis using the right hand rule.

Module

```
use fgl
```

Syntax

```
call msObjRotateX(handle, angle)
call msObjRotateY(handle, angle)
call msObjRotateZ(handle, angle)
```

Descriptions

Procedures `msObjRotateX`, `msObjRotateY` and `msObjRotateZ` rotate the draw object that is associated with argument `handle` in degrees about the x-, y-, z- axes respectively using the right hand rule. The axes are the draw object's axes.

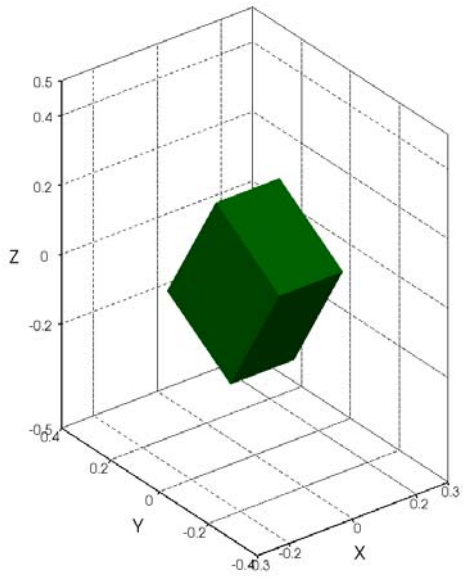
If you want to rotate about the world x-, y- and z- axes, use `msObjRotateWXYZ(handle, angle, 1, 0, 0)`, `msObjRotateWXYZ(handle, angle, 0, 1, 0)` and `msObjRotateWXYZ(handle, angle, 0, 0, 1)`.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: center, cubesize, h

center = (/0, 0, 0/)
cubesize = (/0.2, 0.3, 0.4/)

h = mfCube(center, cubesize, 'g')
call msAxis('equal')
call msAxis(-0.3d0, 0.3d0, -0.4d0, 0.4d0, -0.5d0, 0.5d0)
call msViewPause()
call msObjRotateX(h, 30)
call msViewPause()
call msObjRotateY(h, 30)
call msViewPause()
call msObjRotateZ(h, 10)
call msViewPause()

call msFreeArgs(center, cubesize, h)

end program example
```

Result



See Also

msObjRotateWXYZ

Rotate draw object in degrees about an arbitrary axis.

Module

```
use fgl
```

Syntax

```
call msObjRotateWXYZ(handle, angle, x, y, z)
```

Descriptions

Procedure `msObjRotateWXYZ` rotates the draw object that is associated with argument `handle` in degrees about an arbitrary axis specified by the last three arguments `x, y` and `z.`

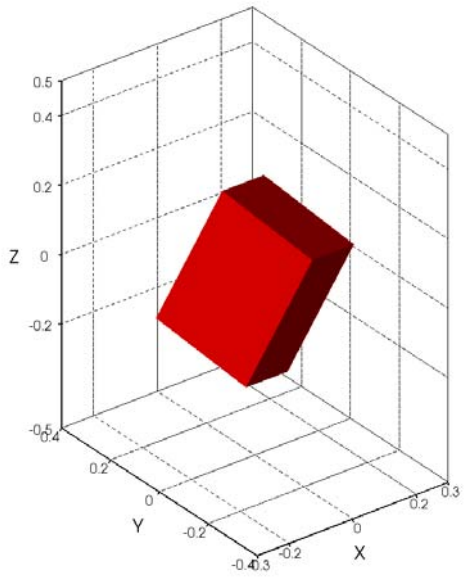
In other words, (x,y,z) is the axis that the rotation will be performed around. If you want to rotate about the object's axes, use `msObjRotateX`, `msObjRotateY` and `msObjRotateZ`.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: center, cubesize, h

center = (/0, 0, 0/)
cubesize = (/0.2, 0.3, 0.4/)

h = mfCube(center, cubesize, 'r')
call msAxis('equal')
call msAxis(-0.3d0, 0.3d0, -0.4d0, 0.4d0, -0.5d0, 0.5d0)
call msViewPause()
call msObjRotateWXYZ(h, 30, 1, 1, 1)
call msViewPause()

call msFreeArgs(center, cubesize, h)

end program example
```

Result



See Also

msObjRotateX

mfObjScale, msObjScale

Scale of draw object.

Module

```
use fgl
```

Syntax

```
scale = mfObjScale(handle)
call msObjScale(handle, scale)
call msObjScale(handle, x, y, z)
```

Descriptions

Procedure `msObjScale` resets the scale independently on the x-, y- and z- axes. A scale of zero is illegal and will be replaced with one.

`scale = mfObjScale(handle)`

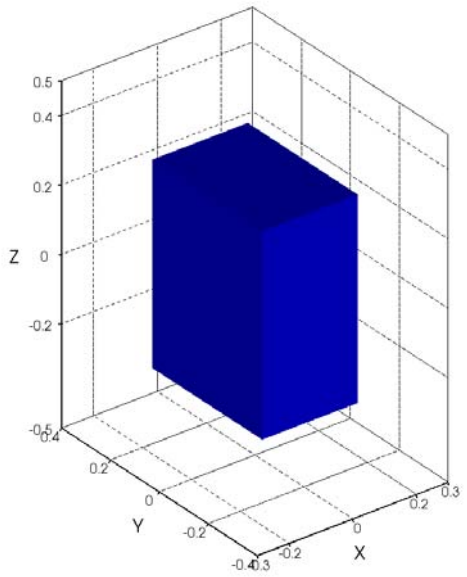
It can also be used as an inquiry procedure to retrieve the scale of the draw object if given only the handle that associates with the draw object.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: center, cubesize, h

center = (/0, 0, 0/)
cubesize = (/0.2, 0.3, 0.4/)

h = mfCube(center, cubesize, 'b')
call msAxis('equal')
call msAxis(-0.3d0, 0.3d0, -0.4d0, 0.4d0, -0.5d0, 0.5d0)
call msViewPause()
call msObjScale(h, 1.5d0, 1.5d0, 1.5d0)
call msViewPause()

call msFreeArgs(center, cubesize, h)

end program example
```

Result



See Also

mfObjPosition, msObjPosition

Position of draw object in world coordinates.

Module

```
use fgl
```

Syntax

```
position = mfObjPosition(handle)
call msObjPosition(handle, [x, y, z])
```

Descriptions

Procedure `mfObjPosition` sets the position of the draw object that is associated with argument `handle` in world coordinates specified in argument `[x, y, z]`.

`position = mfObjPosition(handle)`

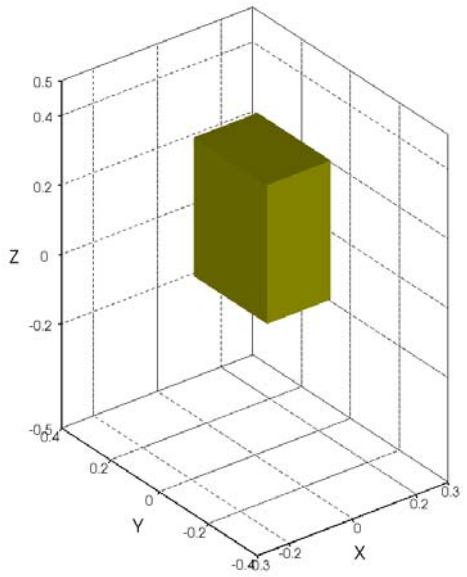
It can also be used as an inquiry procedure to retrieve the position of the draw object if given only the handle that associated with the draw object.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: center, cubesize, h

center = (/0, 0, 0/)
cubesize = (/0.2, 0.3, 0.4/)

h = mfCube(center, cubesize, 'y')
call msAxis('equal')
call msAxis(-0.3d0, 0.3d0, -0.4d0, 0.4d0, -0.5d0, 0.5d0)
call msViewPause()
call msObjPosition(h, 0.1d0, 0.1d0, 0.1d0)
call msViewPause()

call msFreeArgs(center, cubesize, h)

end program example
```

Result



See Also

mfObjOrigin, msObjOrigin

Origin of the draw object.

Module

```
use fgl
```

Syntax

```
origin = mfObjOrigin(handle)
call msObjOrigin(handle, [x, y, z])
```

Descriptions

Procedure `mfObjOrigin` sets the origin of the draw object. All rotations perform on the draw object take place against the origin. Notice that the origin is relative to the position of the object. Whenever the object moves, the origin moves with the position of the object so that they maintain a relative relationship.

`origin = mfObjOrigin(handle)`

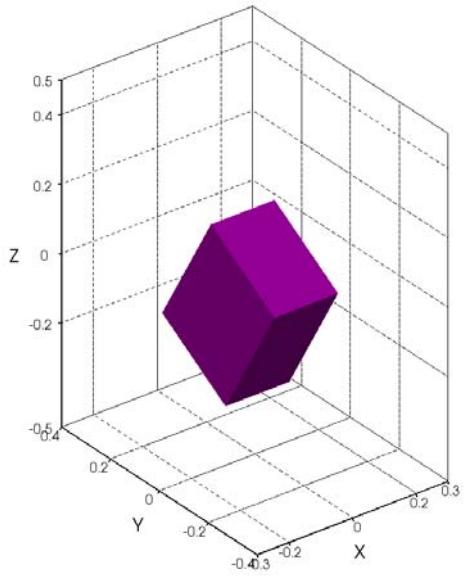
It can also be used as an inquire procedure to retrieve the origin of the draw object if given only the handle that associates with the draw object.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: center, cubesize, h

center = (/0, 0, 0/)
cubesize = (/0.2, 0.3, 0.4/)

h = mfCube(center, cubesize, 'm')
call msAxis('equal')
call msAxis(-0.3d0, 0.3d0, -0.4d0, 0.4d0, -0.5d0, 0.5d0)

! Set origin (0.0d0, 0.15d0, 0.0d0)
call msObjOrigin(h, 0.0d0, 0.15d0, 0.0d0)
call msViewPause()

call msObjRotateX(h, 30)
call msViewPause()
call msObjRotateY(h, 30)
call msViewPause()
call msObjRotateZ(h, 10)
call msViewPause()

call msFreeArgs(center, cubesize, h)

end program example
```

Result



See Also

mfObjOrientation, msObjOrientation

WXYZ orientation of the draw object.

Module

```
use fgl
```

Syntax

```
call msObjOrientation(handle, [x, y, z])
orientation = mfObjOrientation(handle)
```

Descriptions

Procedure `mfObjOrientation` sets the WXYZ orientation of the draw object as a vector of x, y and z rotation. The ordering in which these rotations are performed is Rotate z, Rotate x and then Rotate y.

`orientation = mfObjOrientation(handle)`
It can also be used as an inquiry procedure to retrieve the orientation of the draw object if given only the handle that associates with the draw object.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: center, cubesize, h

center = (/0, 0, 0/)
cubesize = (/0.2, 0.3, 0.4/)

h = mfCube(center, cubesize, 'c')
call msAxis('equal')
call msAxis(-0.3d0, 0.3d0, -0.4d0, 0.4d0, -0.5d0, 0.5d0)
call msViewPause()
call msObjOrientation(h, 15, 15, 15)
call msViewPause()

call msFreeArgs(center, cubesize, h)

end program example
```

See Also

Camera Manipulation

msView

Viewpoint specification.

Module

```
use fgl
```

Syntax

```
call msView(az, el)
call msView([az, el])
call msView(mode)
```

Descriptions

Procedure msView specifies the orientation of an axis object. The orientation of the axis object is determined by the azimuth `az` and elevation `el` of the viewing angle from a viewpoint or can be determined by the setting the view mode.

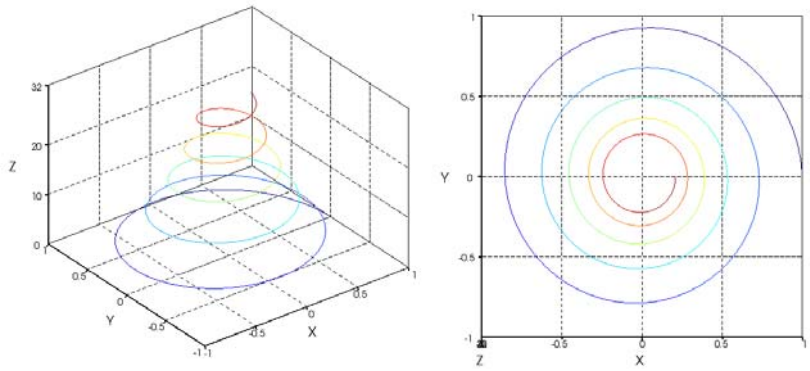
Argument `mode` can be `"2"`, `"3"`, `"home"`, `"top"`, `"bottom"`, `"front"`, `"back"`, `"left"` or `"right"`.

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: x, y, z

z = mfLinspace(0, 10*MF_PI, 315)
x = mfExp(-z/20)*mfCos(z)
y = mfExp(-z/20)*mfSin(z)

!Plot a 3-D line graph using mfPlot3() routine and title it.
call msSubplot(1, 2, 1)
call msPlot3(x, y, z)
call msAxis(mf((/-1, 1, -1, 1, 0, 32/)))

call msSubplot(1, 2, 2)
call msPlot3(x, y, z)
call msView(0, 90)
call msAxis(mf((/-1, 1, -1, 1, 0, 32/)))

!Pauses the program to display graph.
call msViewPause()

!Deallocate mfArray
call msFreeArgs(x, y, z)

end program example
```

Result



See Also

msCamZoom

Zoom in/out.

Module

```
use fgl
```

Syntax

```
call msCamZoom(zf)
```

Descriptions

Procedure `msCamZoom` zooms the displaying object in or out. In perspective mode, it decreases the view angle by the specified zoom factor `zf.` In parallel mode, it decreases the parallel scale by the specified zoom factor `zf.` A value greater than 1 is a zoom-in whereas a value less than 1 is a zoom-out.

See Also

msCamPan

Move the camera horizontally and vertically.

Module

```
use fgl
```

Syntax

```
call msCamPan(dx, dy)
```

Descriptions

Procedure `msCamPan` moves the camera along the horizon and vertical. Arguments `dx` and `dy` specify the horizontal and vertical distances of the displacement respectively.

See Also

msview, msCamZoom

mfCamProj, msCamProj

Set the camera projection mode.

Module

```
use fgl
```

Syntax

```
mode = mfCamProj()
call msCamProj(mode)
```

Descriptions

Procedure `mfCamProj` sets the camera projection mode to be either perspective or parallel projection. Argument `mode` can be `"orthographic"` or `"perspective".`

`mode = mfCamProj()`

- It can also be used an inquiry function to retrieve the camera projection mode. The output argument is a logical mfArray whose value is true if the projection mode is orthographic, false otherwise.

See Also

msView, msCamZoom

Linear Graphs

mfPlot, msPlot

Two-dimensional linear graphs.

Module

```
use fgl
```

Syntax

```
h = mfPlot(y[, linespec])
h = mfPlot(x, y[, linespec])
h = mfPlot(x1, y1, linespec1, x2, y2, linespec2, ...)
call msPlot(mfOut(h1, h2, h3, ...), x1, y1, linespec1, x2, y2,
linespec2, ...)
```

Descriptions

Procedure `mfPlot` generates two-dimensional line graphs.

```
call msPlot(y)
call msPlot(y, linespec)
```

- Plot elements of vector `y` against their indices. If `y` is a matrix, multiple lines are plotted from each column of `y.`
- Argument `linespec` contains special characters that specify line color and marker type of the graph. For example, `"yo",` specifies a graph drawn from yellow-colored, circular markers. `linespec` can be an mfArray containing the special characters or a character string. Refer to linespec for a list of special characters applicable for line spec.

Linespec

The table below lists the `linespec` characters.

Character	Line color	Character	Marker Type	Character	Line Type
y	yellow	.	point	-	solid
m	magenta	o	circle	:	dotted
c	cyan	x	x-mark	-.	dashdot
r	red	+	plus	--	dashed
g	green	*	star		

b	blue	s	square	
w	white	d	diamond	
k	black	v	triangle down	
		^	triangle up	
		<	triangle left	
		>	triangle right	
		p	pentagram	
		h	hexagram	

```
call msPlot(mfOut(h1, h2, h3, ...), x1, y1, linespec1, x2, y2, linespec2, ...)
```

- Handles `h1, h2, h3,` ... retrieve the handles to the plot objects created by `msPlot(...)` respectively.
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the plot object through handle `h` with procedure `msGSet.`

The property available is:

1. linespec

See Also

mfPlot3

mfPlot3, msPlot3

Three-dimensional linear graphs.

Module

```
use fgl
```

Syntax

```
handle = mfPlot3(x, y, z[, c])
handle = mfPlot3(xyz[, c])
```

Descriptions

Procedure `mfPlot3` draws three-dimensional linear graphs.

```
call msPlot3(x, y, z)
call msPlot3(x, y, z, c)
```

- If arguments `x, y` and `z` are vectors, `mfPlot3` draws a line whose x-, y-, and z-coordinates are elements of arguments `x, y` and `z` respectively.
- If arguments `x, y` and `z` are matrices, `mfPlot3` draws multiple lines from the columns of `x, y` and `z` matrices.
- Shape of each argument must be conformed.
- Complex data are not supported.
- Argument `c` contains the corresponding scalar values of the corresponding coordinates `x, y` and `z.` By default, `c` = `z.`

```
call msPlot3(xyz)
call msPlot3(xyz, c)
```

- Vertex vectors are defined in the n-by-3 matrix `xyz.`

```
h = mfPlot3(...)
```

- Handle `h` retrieves a handle to the three-dimensional linear graph objects created by `mfPlot3(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the graphics objects through handle `h` with procedure `msGSet.`

The property available is:

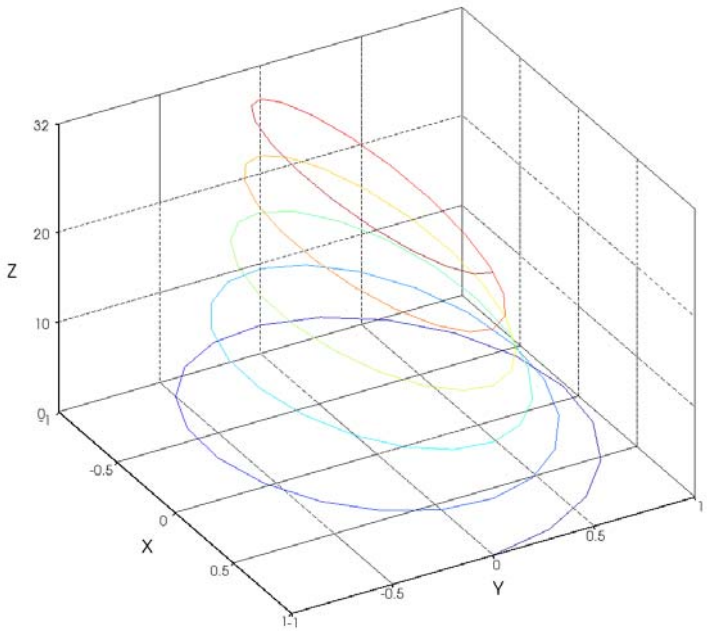
1. xyz: Vertex vectors.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: x, y, z

z = mfLinspace(0, 10*MF_PI, 101)
x = mfCos(z)
y = mfExp(-z/20)*mfSin(z)

call msAxis(mf((/-1, 1, -1, 1, 0, 32/)))

! Plot the three-dimensional graph
call msPlot3(x, y, z)

! Specify the viewpoint
call msView(60,30)

! Pause program for graphics display
call msViewPause()


! Deallocate memory
call msFreeArgs(x, y, z)

end program example
```

Result



See Also

mfViewPause, mfAxis, mfSubplot, mfView, mfSurf, mfMesh

mfRibbon, msRibbon

Three-dimensional ribbons.

Module

```
use fgl
```

Syntax

```
handle = mfRibbon(x, y, z[, c])
handle = mfRibbon(xyz[, c])
```

Descriptions

Procedure `mfRibbon` draws three-dimensional ribbons.

```
call msRibbon(x, y, z)
call msRibbon(x, y, z, c)
```

- If arguments `x, y` and `z` are vectors, `mfRibbon` draws a ribbon whose x-, y-, and z-coordinates are elements of arguments `x, y` and `z` respectively.
- If arguments `x, y` and `z` are matrices, `mfRibbon` draws multiple ribbons from the columns of `x, y` and `z` matrices.
- Shape of each argument must be conformed.
- Complex data are not supported.
- Argument `c` contains the corresponding scalar values of the corresponding coordinates `x, y` and `z.` By default, `c` = `z.`

```
call msRibbon(xyz)
call msRibbon(xyz, c)
```

- Vertex vectors are defined in the n-by-3 matrix `xyz.`

```
h = mfRibbon(...)
```

- Handle `h` retrieves a handle to the three-dimensional ribbon objects created by `mfRibbon(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the graphics objects through handle `h` with procedure `msGSet.`

The properties available are:

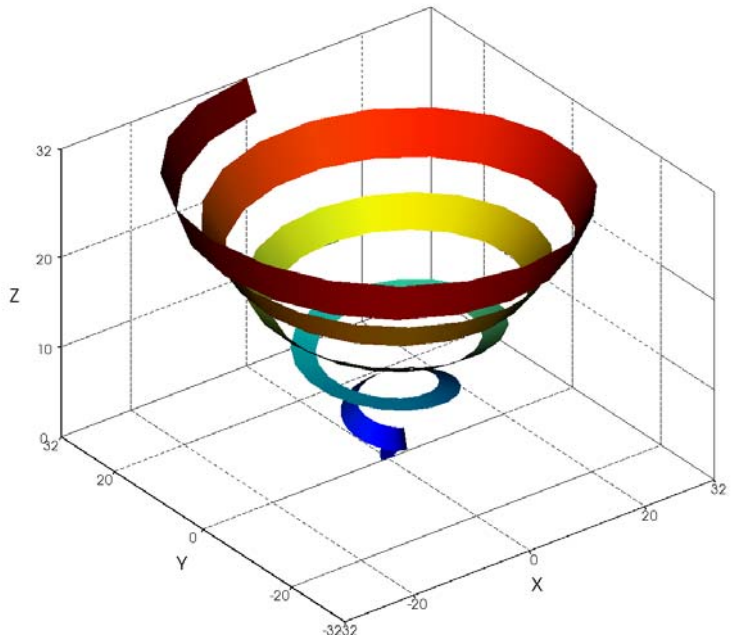
1. sizefactor: The width of the ribbon objects. By default, sizefactor is 1.
2. xyz: Vertex vectors.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: x, y, z

z = mfLinspace(0, 10*MF_PI, 100)
x = z*mfSin(z)
y = z*mfCos(z)

call msRibbon(x, y, z)
call msAxis(-32, 32, -32, 32, 0, 32)
call msViewPause()

call msFreeArgs(x, y, z)

end program example
```

Result



See Also

mfTube, msTube

Three-dimensional tubes.

Module

```
use fgl
```

Syntax

```
handle = mfTube(x, y, z[, c])
handle = mfTube(xyz[, c])
```

Descriptions

Procedure `mfTube` draws three-dimensional tubes.

```
call msTube(x, y, z)
call msTube(x, y, z, c)
```

- If arguments `x, y` and `z` are vectors, `mfTube` draws a tube whose x-, y-, and z-coordinates are elements of arguments `x, y` and `z` respectively.
- If arguments `x, y` and `z` are matrices, `mfTube` draws multiple tubes from the columns of `x, y` and `z` matrices.
- Shape of each argument must be conformed.
- Complex data are not supported.
- Argument `c` contains the corresponding scalar values of the corresponding coordinates `x, y` and `z.` By default, `c` = `z.`

```
call msTube(xyz)
call msTube(xyz, c)
```

- Vertex vectors are defined in the n-by-3 matrix `xyz.`

```
h = mfTube(...)
```

- Handle `h` retrieves a handle to the three-dimensional tube objects created by `mfTube(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the graphics objects through handle `h` with procedure `msGSet.`

The properties available are:

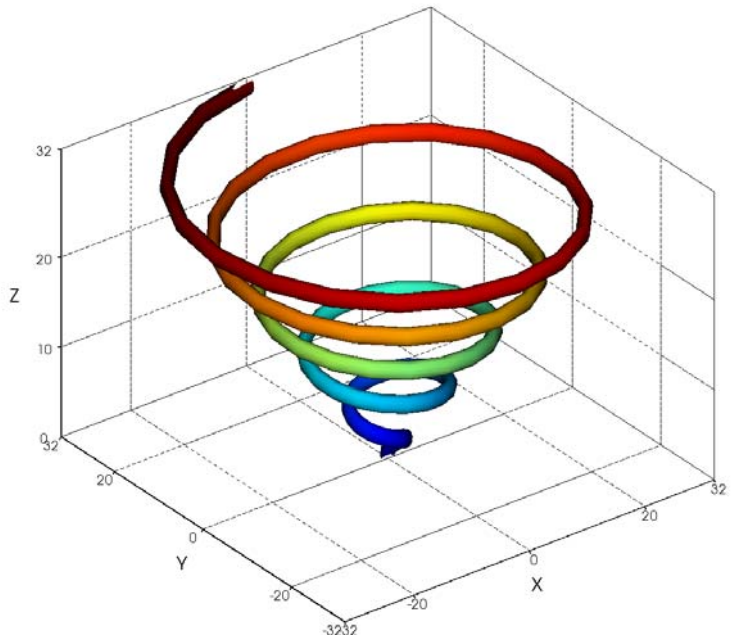
1. sizefactor: The diameter of the tube objects. By default, sizefactor is 1.
2. xyz: Vertex vectors.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: x, y, z, h

z = mfLinspace(0, 10*MF_PI, 100)
x = z*mfSin(z)
y = z*mfCos(z)

h = mfTube(x, y, z)
call msAxis(-32, 32, -32, 32, 0, 32)
call msViewPause()

call msFreeArgs(x, y, z, h)

end program example
```

Result



See Also

Surface Graphs

mfSurf, msSurf

Surface plot.

Module

```
use fgl
```

Syntax

```
handle = mfSurf(x, y, z[, c])
handle = mfSurf(z[, c])
```

Descriptions

Procedure `mfSurf` creates three-dimensional graphs composed of colored quadrilateral surfaces. You can choose several shading options including mesh, flat, faceted, and interpolated. The options can be set using procedure `mfShading` or through the menu and toolbar functions of the Graphics Viewer. Note that mesh surface can also be plotted procedure `mfMesh.`

```
call msSurf(x, y, z)
call msSurf(x, y, z, c)
```

- Plot surface objects from arguments `x, y` and `z.` The arguments `x, y` and `z` contain the x-, y-, and z- coordinates respectively of the surface object's grid intersections.
- If `x, y` and `z` are matrices, their shapes should conform. The grid intersections are given by (x(i,j), y(i,j), z(i,j)).
- If `x` and `y` are vectors, and Shape(z)=[m, n], then if mfLength(x) = n, and mfLength(y)= m, the grid intersections are given by (x(i), y(j), z(i,j)).
- By default, the edge color of the wire-frame grid is proportional to the `z` coordinates of the surface object. Specifying argument `c` would override the default color scale.
- By default, `mfSurf` draws surface with faceted shading.

```
call msSurf(z)
call msSurf(z, c)
```

- Create a three-dimensional surface from the m-by-n matrix `z.` Arguments `x` and `y` are set to the default `x=mfColon(1, n)`, `y=mfColon(1, m)`. Argument `z` is a single-valued function defined over a rectangular grid formed by `x` and `y.`

```
h = mfSurf(...)
```

- Handle `h` retrieves a handle to the surface object created by `mfSurf(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

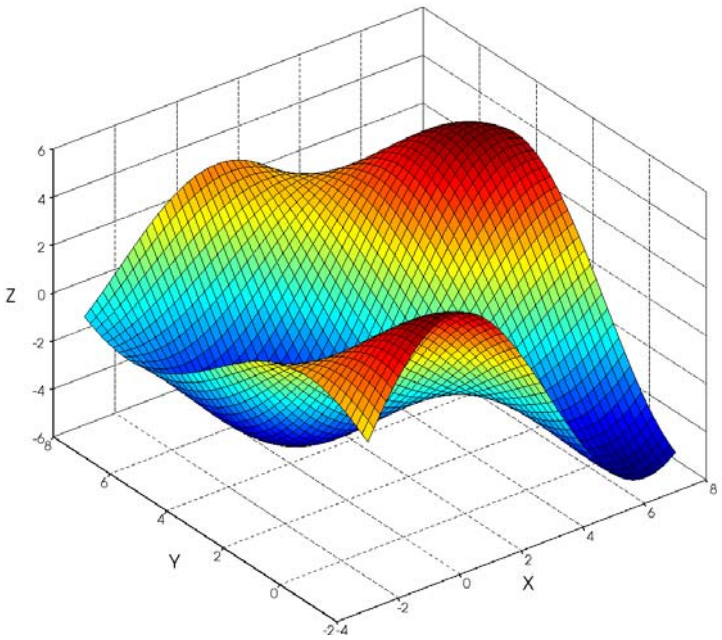
You can specify properties of the surface objects through handle `h` with procedure `msGSet`.

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: a, b, c, x, y, z, indxi, indxj

a = mfLinspace(-3, 7, 51)
b = mfLinspace(-2, 8, 51)
c = mfColon(1, 51)
call msMeshgrid(mfout(x, y), a, b)
call msMeshgrid(mfout(indxi, indxj), c)
z = 3*mfSin((indxi+1)/10)*mfCos((indxj+1)/10) &
    + 2*mfSin((indxi+indxj)/10)

! Plot a surf using mfArray x, y and z
call msSurf(x, y, z)

! Pause to display the graph
call msViewPause()

! Deallocate mfArray
call msFreeArgs(a, b, c, x, y, z, indxi, indxj)

end program example
```

Result



See Also

mfViewPause, mfAxis, mfSubplot, mfView, mfMesh, mfPlot3

mfMesh, msMesh

Mesh plot.

Module

```
use fgl
```

Syntax

```
handle = mfMesh(x, y, z[, c])
handle = mfMesh(z[, c])
```

Descriptions

Procedure `mfMesh` plots a three-dimensional mesh surface consisting of criss-crossed lines that looks like a net draped over the surface defined by your data.

```
call msMesh(x, y, z)
call msMesh(x, y, z, c)
```

- Plot surface objects from arguments `x, y` and `z.` The arguments `x, y` and `z` contain the x-, y-, and z- coordinates respectively of the surface object's grid intersections.
- If `x, y` and `z` are matrices, their shapes should conform. The grid intersections are given by (x(i,j), y(i,j), z(i,j)).
- If `x` and `y` are vectors, and Shape(z)=[m, n], then if mfLength(x) = n, and mfLength(y)= m, the grid intersections are given by (x(i), y(j), z(i,j)).
- By default, the edge color of the wire-frame grid is proportional to the `z` coordinates of the surface object. Specifying argument `c` would override the default color scale.

```
call msMesh(z)
call msMesh(z, c)
```

- Create a three-dimensional surface from the m-by-n matrix `z.` Arguments `x` and `y` are set to the default `x=mfColon(1, n)`, `y=mfColon(1, m)`. Argument `z` is a single-valued function defined over a rectangular grid formed by `x` and `y.`

```
h = mfMesh(...)
```

- Handle `h` retrieves a handle to the mesh surface object created by `mfMesh(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

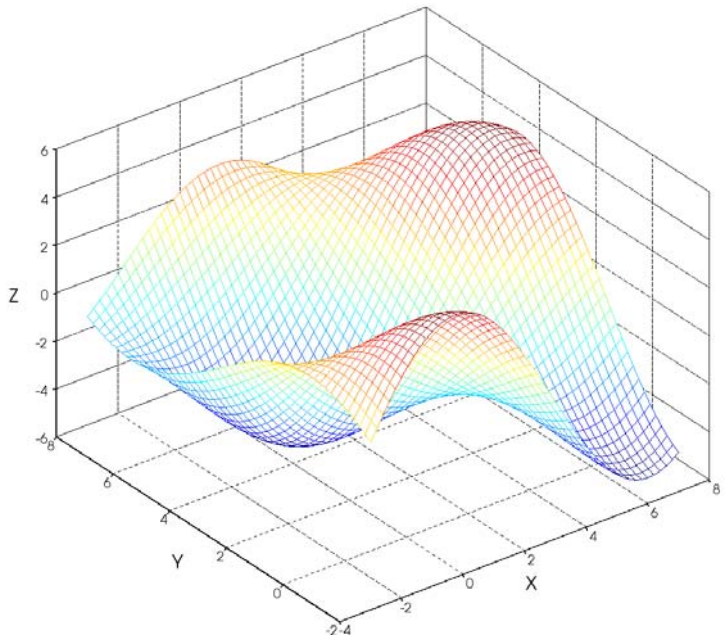
You can specify properties of the surface objects through handle `h` with procedure `msGSet.`

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: a, b, c, x, y, z, indxi, indxj

a = mfLinspace(-3, 7, 51)
b = mfLinspace(-2, 8, 51)
c = mfColon(1, 51)
call msMeshgrid(mfout(x, y), a, b)
call msMeshgrid(mfout(indxi, indxj), c)
z = 3*mfSin((indxi+1)/10)*mfCos((indxj+1)/10) &
    + 2*mfSin((indxi+indxj)/10)

! Plot a mesh grid using mfArray x, y and z for the grid
! intersections.
call msMesh(x, y, z)

! Pause to display the graph
call msViewPause()

! Deallocate mfArray
call msFreeArgs(a, b, c, x, y, z, indxi, indxj)

end program example
```

Result



See Also

mfSurfc, msSurfc

Combined plot of surface and contour3.

Module

```
use fgl
```

Syntax

```
handle = mfSurfc(x, y, z[, c])
handle = mfSurfc(z[, c])
call msSurfc(mfOut(h1, h2), x, y, z[, c])
call msSurfc(mfOut(h1, h2), z[, c])
```

Descriptions

Procedure `mfSurfc` draws a contour below the surface object.

`call msSurfc(z)` draws a contour below the surface object created by `mfSurf(z)`. `call msSurfc(x, y, z)` draws a contour below the surface object created by `mfSurf(x, y, z)`.

`call msSurfc(mfOut(h1, h2), ...)`

- Handles `h1` and `h2` retrieve the handles to the surface object and contour object that are created by `mfSurfc(...)` respectively.
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the surface object through handle `h` with procedure `msGSet`.

For the properties, see the description on `mfSurf`.

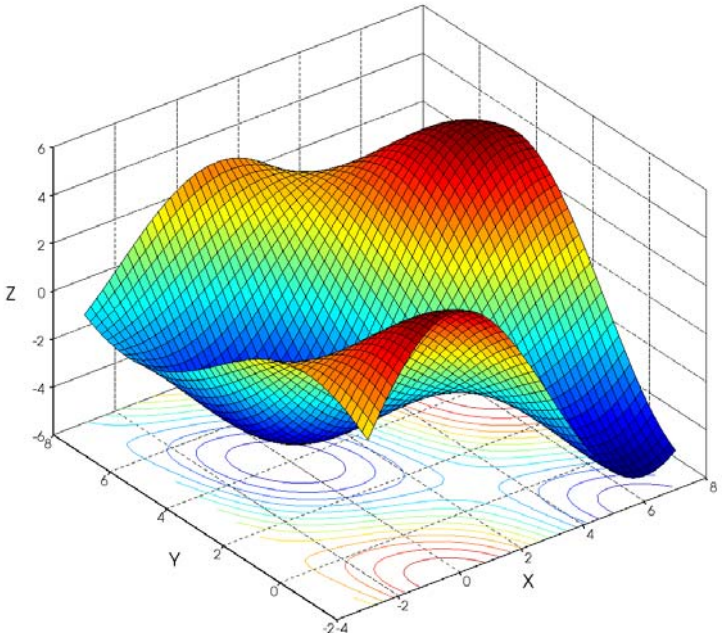
Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: a, b, c, x, y, z, indxi, indxj

a = mfLinspace(-3, 7, 51)
b = mfLinspace(-2, 8, 51)
c = mfColon(1, 51)
call msMeshgrid(mfout(x, y), a, b)
call msMeshgrid(mfout(indxi, indxj), c)
z = 3*mfSin((indxi+1)/10)*mfCos((indxj+1)/10) &
   + 2*mfSin((indxi+indxj)/10)
```

```
! Plot a surf using mfArray x, y and z and Plot Contour below the surf
call msSurfc(x, y, z)

! Pause to display the graph
call msViewPause()

! Deallocate mfArray
call msFreeArgs(a, b, c, x, y, z, indxi, indxj)

end program example
```

Result



See Also

mfSurf

mfMeshc, msMeshc

Combined plot of mesh and contour3.

Module

```
use fgl
```

Syntax

```
handle = msMeshc(x, y, z[, c])
handle = mfMeshc(z[, c])
call msMeshc(mfOut(h1, h2), x, y, z[, c])
call msMeshc(mfOut(h1, h2), z[, c])
```

Descriptions

Procedure `mfMeshc` draws a contour below the meshed surface object.

`call msMeshc(z)` draws a contour below the meshed surface object created by `mfSurf(z)`. `call msMeshc(x, y, z)` draws a contour below the meshed surface object created by `mfSurf(x, y, z)`.

`call msSurfc(mfOut(h1, h2), ...)`

- Handles `h1` and `h2` retrieve the handles to the meshed surface object and contour object that are created by `mfMeshc(...)` respectively.
- If only one handle `h` is given, the procedure returns the two handles in a vector mfArray.
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the surface object through handle `h` with procedure `msGSet`.

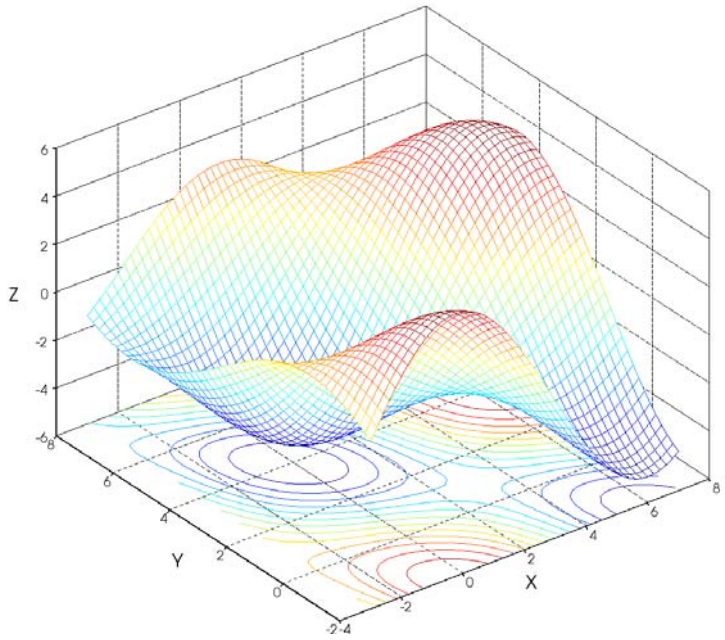
Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: a, b, c, x, y, z, indxi, indxj

a = mfLinspace(-3, 7, 51)
b = mfLinspace(-2, 8, 51)
c = mfColon(1, 51)
call msMeshgrid(mfout(x, y), a, b)
call msMeshgrid(mfout(indxi, indxj), c)
z = 3*mfSin((indxi+1)/10)*mfCos((indxj+1)/10) &
```

```
    + 2*mfSin((indxi+indxj)/10)

! Plot a mesh grid using mfArray x, y and z for the grid
! intersections and Plot Contour below the mesh
call msMeshc(x, y, z)

! Pause to display the graph
call msViewPause()

! Deallocate mfArray
call msFreeArgs(a, b, c, x, y, z, indxi, indxj)

end program example
```

Result



See Also

mfMesh

mfPColor, msPColor

pseudocolor plot.

Module

```
use fgl
```

Syntax

```
handle = mfPColor(c)
handle = mfPColor(x,y,c)
```

Descriptions

Procedure `mfPColor` produces pseudocolor plot of a matrix mfArray `c` by mapping the elements of `c` to the current colormap. This procedure is equivalent to a top-view of `mfSurf.`

`call msPColor(c)`

- The procedure displays matrix mfArray `c` as a checker-board plot with elements of `c` specifying each cell of the plot, mapped to the index of the current colormap.
- The smallest and largest elements of matrix `c` correspond to the minimum and maximum indices of the colormap.
- By default, the shading is `"faceted",` with each cell containing a constant color. Each element of matrix `c` specifies the color of a rectangular patch of the image.

`call msPColor(x, y, c)`

- It plots the checker-board plot on the grid defined by arguments `x` and `y.` The arguments `x` and `y` can be vectors or matrices.

`h = mfPColor(...)`

- Handle `h` retrieves a handle to the pseudocolor plot object created by `mfPColor(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the pseudocolor plot object through handle `h` with procedure `msGSet.`

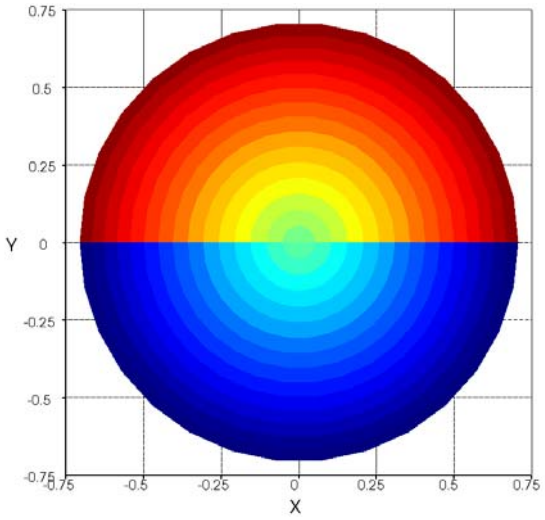
Example

Code

```
program example
```

```
use fml
use fgl
implicit none

type(mfArray) :: theta, phi, x, y, z

theta = mfLinspace(-MF_PI, MF_PI, 31)
phi = .t. theta /4
x = mfMul(mfSin(phi), mfCos(theta))
y = mfMul(mfSin(phi), mfSin(theta))
z = mfMul(mfSin(phi), mfOnes(1,31))

! Plot the surface object using X, Y and Z
call msPColor(x, y, z)
call msAxis('equal')
call msCamZoom(0.8d0)

! Pause the program for display
call msViewPause()

! Deallocate mfArray
call msFreeArgs(x, y, z)

end program example
```

Result



See Also

mfSurf

Module

```
use mod_
```

Syntax

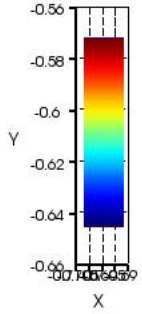
Descriptions

Example

Code

```
program Example_msFastPColor

use fml
use fgl
implicit none

type(mfArray) :: theta, phi, x, y, z, extent

theta = mfLinspace(-MF_PI, MF_PI, 31)
phi = .t. theta /4
x = mfMul(mfSin(phi), mfCos(theta))
y = mfMul(mfSin(phi), mfSin(theta))
z = mfMul(mfSin(phi), mfOnes(1,31))
extent = mfMin(x) .hc. mfMax(x) .hc. mfMin(y) .hc. mfMax(y)

! Plot the surface object using X, Y and Z
call msFastPColor(z, extent)
call msAxis('equal')
call msCamZoom(0.8d0)

! Pause the program for display
call msViewPause()

! Deallocate mfArray
call msFreeArgs(x, y, z)

end program Example_msFastPColor
```

Result



See Also

mfContour, msContour

Two-dimensional contour.

Module

```
use fgl
```

Syntax

```
handle = mfContour(x, y, c)
handle = mfContour(c)
```

Descriptions

Procedure `mfContour` plots constant value lines in two-dimensional space.

`handle = mfContour(x, y, c)`

- Generate two-dimensional contour lines of matrix `c.` The values plotted are selected automatically.
- Argument `c` is assumed to contain data representing the scalar values.
- Arguments `x` and `y` specify the corresponding x, y coordinates of the scalar values.
- Similar to `mfSurf` and `mfMesh,` the edge colors of the contour lines are selected based on the current colormap.

`call msContour(c)`

- The scalar values `c` is assumed to be defined over a geometrically rectangular grid where `x = mfColon(1, n)`and `y = mfColon(1, m)`

.

`h = mfContour(...)`

- Handle `h` retrieves a handle to the contour object created by `mfContour(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the contour object through handle `h` with procedure `msGSet.`

See mfContour3 for available properties.

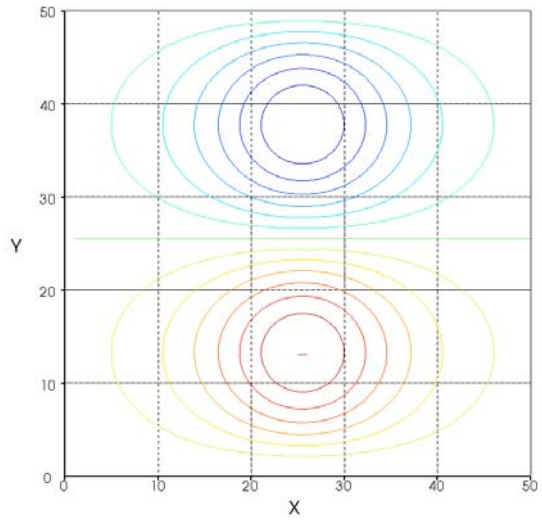
Example

Code

```
program example

use fml
use fgl
implicit none
```

```
type(mfArray) :: a, x, y, z

a = mfLinspace(-MF_PI, MF_PI, 50)
call msMeshgrid(mfout(x, y) ,a)
z = (1/mfCosh(x))*mfCos(y+MF_PI/2)

! Draw a 2-D contour plot
call msContour(z)
call msAxis('equal')
call msCamZoom(0.8d0)

! Pause the program to display the graphics
call msViewPause()

! Deallocate mfArrays
call msFreeArgs(a, x, y, z)

end program example
```

Result



See Also

mfSolidContour

mfContour3, msContour3

Three-dimensional Line Contour.

Module

```
use fgl
```

Syntax

```
handle = mfContour3(x, y, z[, c])
handle = mfContour3(z[, c])
```

Descriptions

Procedure `mfContour3` plots constant value lines of matrix `z` in three-dimensional space. The values plotted are selected automatically.

```
call msContour3(x, y, z, c)
call msContour3(z, c)
```

- This procedure is similar to `mfContour.` The contour lines are presented in three-dimensional perspective reflecting their scalar values. The values plotted are selected automatically.

```
h = mfContour3(...)
```

- Handle `h` retrieves a handle to the three-dimensional contour object created by `mfContour3(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the three-dimensional contour object through handle `h` with procedure `msGSet.`

The properties available are:

1. iso: iso-values, a vector containing iso-value set. Setting this property will replace default set of contour lines.
2. autolevel: given number of levels, it will generate the iso-value set automatically.
3. label: `"on"` or `"off"`

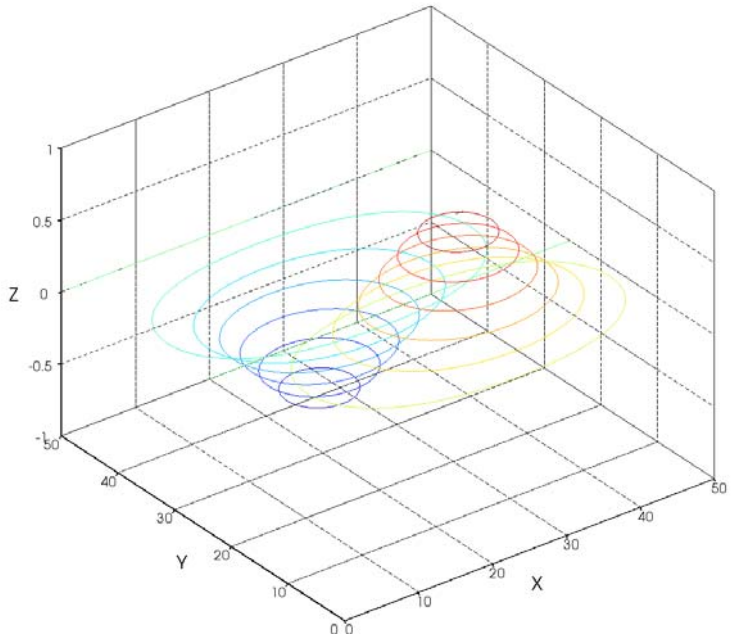
Example

Code

```
program example

use fml
use fgl
```

```
implicit none

type(mfArray) :: a, x, y, z

a = mfLinspace(-MF_PI, MF_PI, 50)
call msMeshgrid(mfout(x, y) ,a)
z = (1/mfCosh(x))*mfCos(y+MF_PI/2)

! Draw a 3-D contour plot
call msContour3(z)

! Pause the program to display the graphics
call msViewPause()

! Deallocate mfArrays
call msFreeArgs(a, x, y, z)

end program example
```

Result



See Also

mfContour, mfSolidContour, mfSolidContour3, mfTriContour

mfSolidContour, msSolidContour

Two-dimensional solid contour.

Module

```
use fgl
```

Syntax

```
handle = mfSolidContour(x, y, c)
handle = mfSolidContour(c)
```

Descriptions

Procedure `mfSolidContour` fills colors in the areas that are in between constant value lines in two-dimensional space.

`handle = mfSolidContour(x, y, c)`

- The values plotted are selected automatically.
- Argument `c` is assumed to contain data representing the scalar values.
- Arguments `x` and `y` specify the corresponding x, y coordinates of the scalar values.
- Similar to `mfSurf` and `mfMesh,` the surface colors are selected based on the current colormap.

`call msSolidContour(c)`

- The scalar values `c` is assumed to be defined over a geometrically rectangular grid where `x = mfColon(1, n)` and `y = mfColon(1, m)`

.

`h = mfSolidContour(...)`

- Handle `h` retrieves a handle to the contour object created by `mfSolidContour(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the contour object through handle `h` with procedure `msGSet.`

See mfSolidContour3 for available properties.

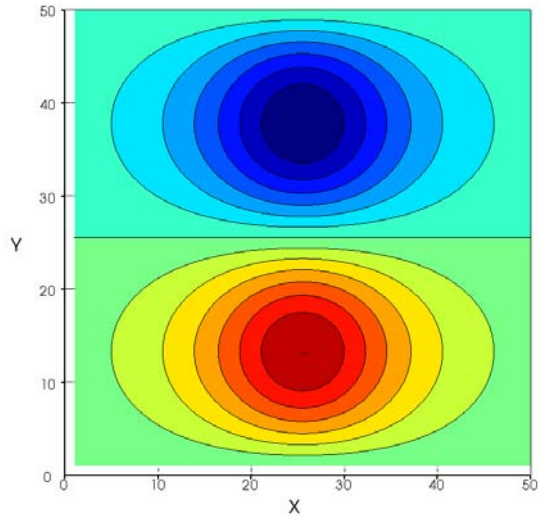
Example

Code

```
program example

use fml
use fgl
```

```
implicit none

type(mfArray) :: a, x, y, z

a = mfLinspace(-MF_PI, MF_PI, 50)
call msMeshgrid(mfout(x, y) ,a)
z = (1/mfCosh(x))*mfCos(y+MF_PI/2)

! Draw a 2-D solidcontour
call msSolidContour(z)
call msAxis('equal')
call msCamZoom(0.8d0)

! Pause the program to display the graphics
call msViewPause()

! Deallocate mfArrays
call msFreeArgs(a, x, y, z)

end program example
```

Result



See Also

mfSoildContour3, msSoildContour3

Three-dimensional solid Contour.

Module

```
use fgl
```

Syntax

```
handle = mfSoildContour3(x, y, z[, c])
handle = mfSoildContour3(z[, c])
```

Descriptions

Procedure `mfSoildContour3` fills colors in the areas that are in between the constant value lines in three-dimensional space.

```
call msSolidContour3(x, y, z, c)
call msSolidContour3(z, c)
```

- This procedure is similar to `mfSolidContour.` The areas are presented in three-dimensional perspective reflecting their scalar values specified in argument `c.`

```
h = mfSolidContour3(...)
```

- Handle `h` retrieves a handle to the three-dimensional contour object created by `mfSolidContour3(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the three-dimensional contour object through handle `h` with procedure `msGSet.`

The properties available are:

1. iso: iso-values, a vector containing iso-value sets.
2. autolevel: given number of levels, it will generate iso-value sets automatically.
3. label: `"on"` or `"off"`

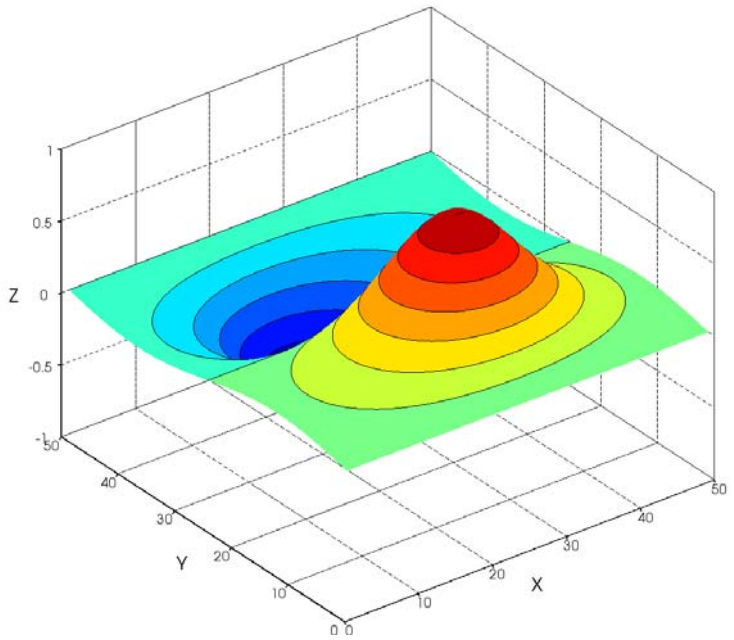
Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: a, x, y, z
```

```
a = mfLinspace(-MF_PI, MF_PI, 50)
call msMeshgrid(mfout(x, y) ,a)
z = (1/mfCosh(x))*mfCos(y+MF_PI/2)

! Draw a 3-D solidcontour
call msSolidContour3(z)

! Pause the program to display the graphics
call msViewPause()

! Deallocate mfArrays
call msFreeArgs(a, x, y, z)

end program example
```

Result



See Also

mfContour, mfContour3, mfSolidContour, mfSolidContour, mfTriContour

mfOutline, msOutline

Wireframe outline boundary.

Module

```
use fgl
```

Syntax

```
handle = mfOutline(x, y, z)
handle = mfOutline(z)
```

Descriptions

Procedure `mfOutline` generates wireframe outline boundary for a given data set. Arguments `x, y` and `z` specify the corresponding coordinates of the points in the data set.

`handle = mfOutline(...)`

- Handle `h` retrieves a handle to the outline object created by `mfOutline(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the outline object through handle `h` with procedure `msGSet.`

See Also

mfIsoSurface, msIsoSurface

Three-dimensional iso-value surface plot from volumetric data.

Module

```
use fgl
```

Syntax

```
handle = mfIsoSurface(x, y, z, c, isovalue)
handle = mfIsoSurface(c, isovalue)
```

Descriptions

Procedure `mfIsoSurface` creates 3-D graphs composed of isosurface data from the volumetric data `c` at the isosurface value specified in argument `isovalue.`

`call msIsoSurface(x, y, z, c, isovalue)`

- The arguments `x, y` and `z` define the coordinates for the volume `c.`

`call msIsoSurface(c, isovalue)`

- The coordinates for the volume `c` is of a geometrically rectangular grid where `x = mfColon(1, n)` and `y = mfColon(1, m) and z = mfColon(1, p)`.

`h = mfIsoSurface(...)`

- Handle `h` retrieves a handle to the isosurface object created by `mfIsoSurface(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the isosurface object through handle `h` with procedure `msGSet.`

The properties available are:

1. iso: iso-value, a vector containing iso-value sets.

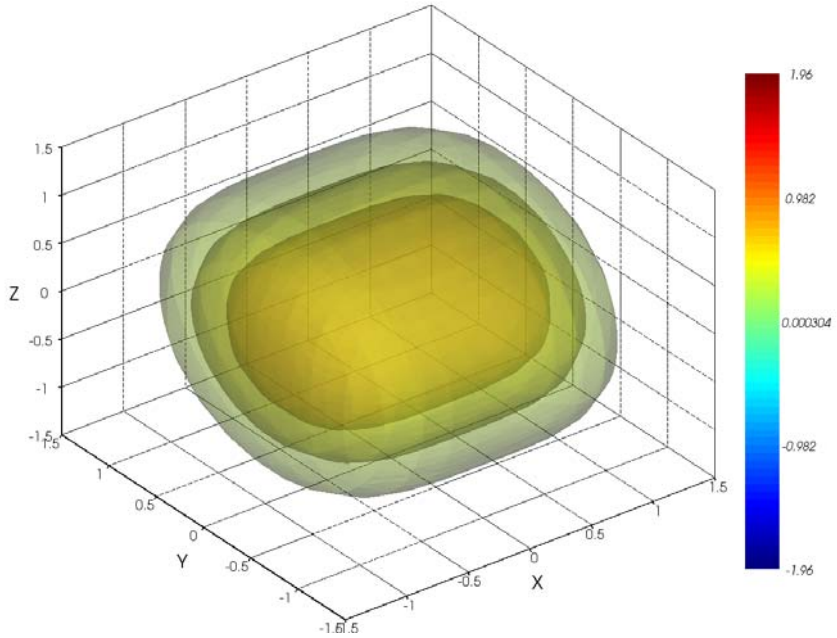
Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: x, y, z, v, a, b, c, h
```

```
a = mfLinspace(-2, 2.2d0, 21)
b = mfLinspace(-2, 2.25d0, 17)
c = mfLinspace(-1.5d0, 1.6d0, 31)
call msMeshgrid(mfout(y, x, z), b, a, c)
v = 2*mfCos(x**2)*mfExp(-(y**2)-(z**2))

! Plot IsoSurface and Set Transparency to 70
h = mfIsoSurface(x, y, z, v, mf((/1.0, 0.6, 0.3/)))
call msDrawMaterial(h, mf('surf'), mf('trans'), mf(70))

! Set Colorbar
call msColorbar('on')

! Pause the program for display
call msViewPause()

! Deallocate mfArrays
call msFreeArgs(x, y, z, v, a, b, c, h)

end program example
```

Result



See Also

msViewPause, mfAxis, mfSubplot, mfView, mfMesh, mfSurf

Slice Graphs

mfSliceXYZ, msSliceXYZ

Display orthogonal slice-planes through volumetric data.

Module

```
use fgl
```

Syntax

```
call msSliceXYZ([x, y, z, ]c, Sx, Sy, Sz)
handle = mfSliceXYZ([x, y, z, ]c, Sx, Sy, Sz)
```

Descriptions

Procedure mfSliceXYZ displays orthogonal slice-planes of a specified set of volumetric data. The information on the slice-planes can be retrieved by using procedure `mfGetSliceXYZ.`

`call msSliceXYZ(x, y, z, c, Sx, Sy, Sz)`

- Displays orthogonal slice-planes along the x, y and z directions specified by points in vector mfArrays `Sx, Sy` and `Sz`
- Use `mf()` to substitute any of the direction vectors `Sx, Sy` or `Sz` if you are not drawing any slice along the respective direction. E.g. call msSliceXYZ(x, y, z, v, Sx, mf(), Sz) draws slices along the x-axis as specified by `Sx` and z-axis as specified by `Sz.`
- The arguments `x, y` and `z` define the corresponding coordinates of scalar values mfArray `c,` where `c` is an m-by-n-by-p three-dimensional array.
- The arguments `x, y` and `z` must be of the same shape as `c` and are monotonic and three-dimensional plaid as if produced by procedure `mfMeshgrid.`
- The color at each point is determined by three-dimensional interpolation into the elements of volume `c,` mapped to the current colormap.

`call msSliceXYZ(c, Sx, Sy, Sz)`

- Assumes `x` is composed of mfColon(1, n) vectors, `y` is composed of mfColon(1, m) vectors and `z` is composed of mfColon(1, p) vectors.

`h = mfSliceXYZ(...)`

- Handle `h` retrieves a handle to the volumetric slice object created by `mfSliceXYZ(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the volumetric slice objects through handle `h` with procedure

`msGSet.`

The properties available are:

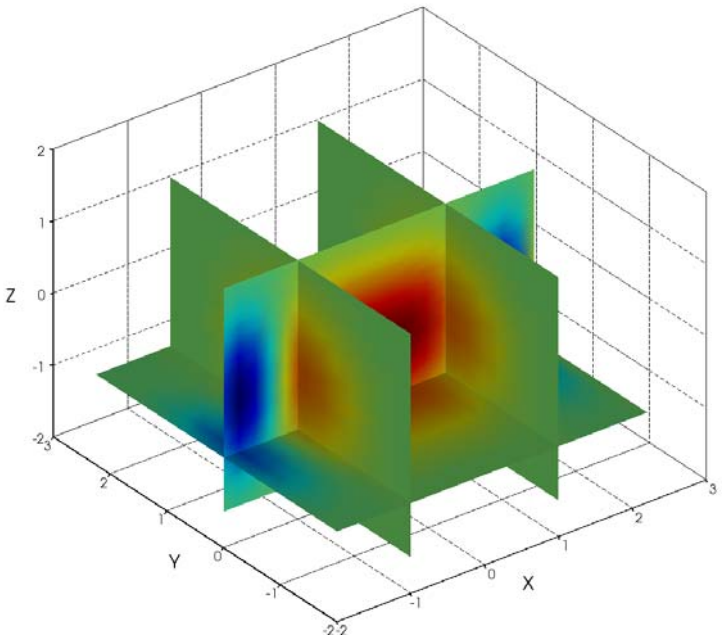
1. slicex: specifies the slice-planes along the x direction.
2. slicey: specifies the slice-planes along the y direction.
3. slicez: specifies the slice-planes along the z direction.

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: nx, ny, nz, x, y, z, c

nx = mfLinspace(-2, 2.2d0, 21)
ny = mfLinspace(-2, 2.25d0, 17)
nz = mfLinspace(-1.5d0, 1.6d0, 31)
call msMeshgrid(mfout(y, x, z), ny, nx, nz)
c = 2*mfCos(x**2)*mfExp(-(y**2)-(z**2))

call msSliceXYZ(x, y, z, c, mf((/-1.0d0, 1.0d0/)), &
                mf(0), mf(-.75d0))
call msViewPause()

call msFreeArgs(nx, ny, nz, x, y, z, c)

end program example
```

Result



See Also

mfSliceIJK, mfSlicePlane, mfGetSliceXYZ

mfSliceIJK, msSliceIJK

Display orthogonal slice-planes along i, j or k indices.

Module

```
use fgl
```

Syntax

```
handle = mfSliceIJK(x, y, z, c, Si, Sj, Sk)
handle = mfSliceIJK(c, Si, Sj, Sk)
```

Descriptions

Procedure `mfSliceIJK` displays slice-planes along i, j and k which are index of `x,` index of `y` and index of `z` respectively.

`call msSliceIJK(x, y, z, c, Si, Sj, Sk)`

- Displays slice-planes along arbitrary indices specified by mfArrays `Si, Sj` and `Sk.`
- Use `mf()` to substitute any of the index vectors `Si, Sj` or `Sk` if you are not drawing any slice along the respective index. For example, `call msSlice(x, y, z, c, Si, mf(), Sk)` draws slices along indices of `x` as specified by `Si` and `z` as specified by `Sz.`
- Arguments `x, y` and `z` define the corresponding coordinates of volumetric data `c,` where `c` is a m-by-n-by-p three-dimensional array.
- Arguments `x, y` and `z` must be of the same shape as `c` and are monotonic and three-dimensional plaid as if produced by procedure mfMeshgrid.

`h = mfSliceIJK(c, Sx, Sy, Sz)`

- Handle `h` retrieves a handle to the slice objects created by `mfSliceXYZ(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the slice objects through handle `h` with procedure `msGSet.`

The properties available are:

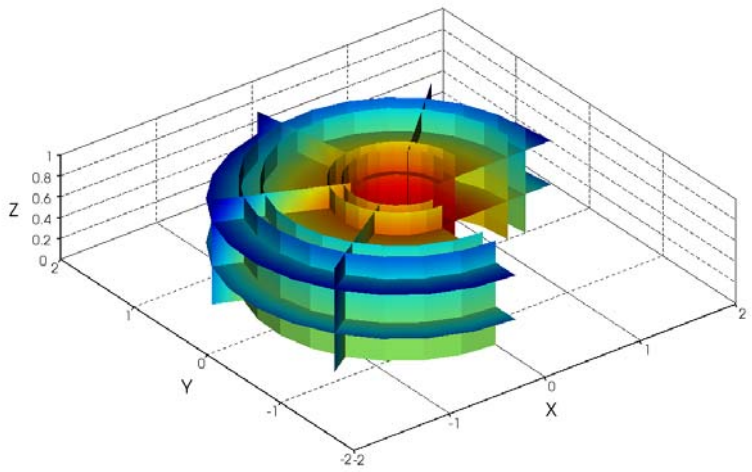
1. slicei: specifies indices of x.
2. slicej: specifies indices of y.
3. slicek: specifies indices of z.

Example

Code

```
program Example

use fml
use fgl
implicit none

type(mfArray) :: nu, nv, nw, u, v, w, x, y, z, c, h

nu = mfLinspace(0, 1.5d0*MF_PI, 20)
nv = mfLinspace(0, 2*MF_PI, 20)
nw = mfLinspace(0, 1, 20)
call msMeshGrid( mfOut(u, v, w), nu, nv, nw )
x = ( 1 + 0.6d0 * mfCos(v) ) * mfCos(u)
y = ( 1 + 0.6d0 * mfCos(v) ) * mfSin(u)
z = w
c = 1 - ( x ** 2 + y ** 2 + z ** 2)

h = mfSliceIJK(x, y, z, c, mf((/4, 8, 12, 16/)), &
              mf((/4, 8, 12, 16/)), mf((/8, 16/)))
!call msAxis('equal')
call msCamZoom(1.8d0)
call msViewPause()

call msFreeArgs(nu, nv, u, v, x, y, z, c, h)

end program example
```

Result



See Also

mfSlicePlane, msSlicePlane

Display orthogonal slice-planes along arbitrary direction.

Module

```
use fgl
```

Syntax

```
handle = mfSlicePlane(x, y, z, c, plane)
handle = mfSlicePlane(c, plane)
```

Descriptions

Procedure `mfSlice` displays orthogonal slice-planes of a specified set of volumetric data along arbitrary direction. The information on the slice-planes can be retrieved by using procedure `mfGetSlicePlane.`

`call msSlicePlane(x, y, z, c, plane)`

- Displays orthogonal slice-planes along the direction specified by `plane.`
- The arguments `x, y` and `z` are structured grid data which define the corresponding coordinates of scalar values specified in argument `c,` where `c` is a m-by-n-by-p three-dimensional array.
- Argument `plane` is a vector of size 4 representing the coefficients of the sliced plane equation, i.e. [a, b, c, d], where ax + by + cz + d = 0.

- The arguments `x, y` and `z` must be of the same shape as `c` and are monotonic and three-dimensional plaid as if produced by procedure `mfMeshgrid.`
- The color at each point is determined by three-dimensional interpolation into the elements of volume `c,` mapped to the current colormap.

`call msSlicePlane(c, plane)`

- Assumes `x` is composed of mfColon(1, n) vectors, `y` is composed of mfColon(1, m) vectors and `z` is composed of mfColon(1, p) vectors.

`h = mfSlicePlane(...)`

- Handle `h` retrieves a handle to the volumetric slice objects created by `mfSlicePlane(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the volumetric slice objects through handle `h` with procedure `msGSet.`

The property available is:

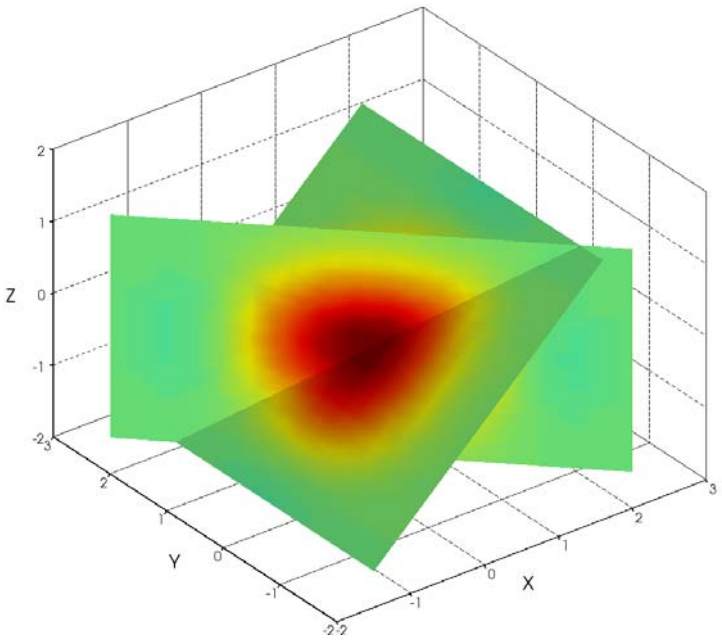
1. plane

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: nx, ny, nz, x, y, z, c

nx = mfLinspace(-2, 2.2d0, 21)
ny = mfLinspace(-2, 2.25d0, 17)
nz = mfLinspace(-1.5d0, 1.6d0, 31)
call msMeshgrid(mfout(y, x, z), ny, nx, nz)
c = 2*mfCos(x**2)*mfExp(-(y**2)-(z**2))

call msSlicePlane(x, y, z, c, mf((/1, 0, -1, 0/)))
call msHold('on')
call msSlicePlane(x, y, z, c, mf((/1, 1, 0, 0/)))
call msViewPause()

call msFreeArgs(nx, ny, nz, x, y, z, c)


end program example
```

Result



See Also

mfSlice, mfGetSlicePlane

mfGetSliceXYZ, msGetSliceXYZ

Retrieve orthogonal slice-plane(s) through volumetric data.

Module

```
use fgl
```

Syntax

```
call msGetSliceXYZ(mfOut(tri, xyz), [x, y, z, ]c, Sx, Sy, Sz)
tri = mfGetSliceXYZ([x, y, z, ]c, Sx, Sy, Sz)
```

Descriptions

Procedure `mfGetSliceXYZ` displays and retrieves orthogonal slice-planes of a specified set of volumetric data. It returns the triangular mesh `tri` and the vertex vectors of the sliced-planes defined in the n-by-3 matrix `xyz.`

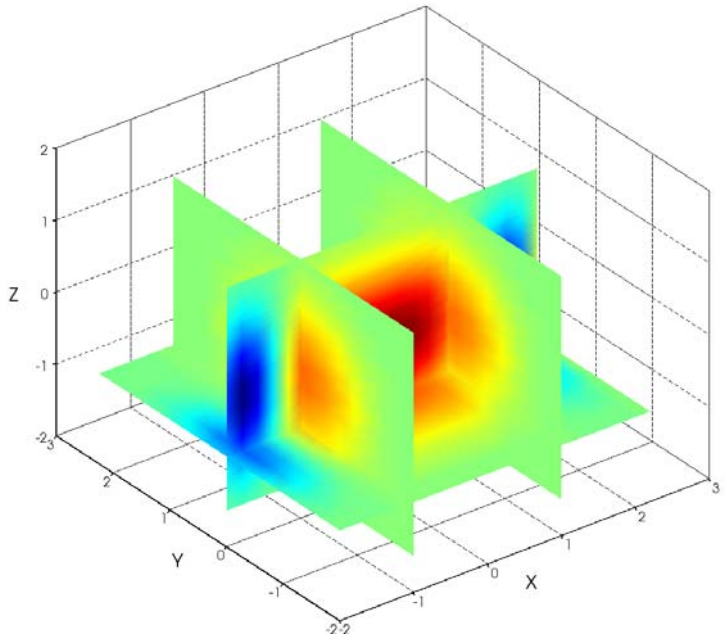
For details on the input arguments, refer to the description of procedure `mfSliceXYZ.`

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: nx, ny, nz, x, y, z, c, h, tri, xyz

nx = mfLinspace(-2, 2.2d0, 21)
ny = mfLinspace(-2, 2.25d0, 17)
nz = mfLinspace(-1.5d0, 1.6d0, 31)
call msMeshgrid(mfout(y, x, z), ny, nx, nz)
c = 2*mfCos(x**2)*mfExp(-(y**2)-(z**2))

call msGetSliceXYZ(mfout(tri, xyz), x, y, z, c, mf((/-1.0d0, 1.0d0/)),
&
                mf(0), mf(-.75d0))
c = 2*mfCos(mfS(xyz, MF_COL, 1)**2)*mfExp(-(mfS(xyz, MF_COL,
2)**2)-(mfS(xyz, MF_COL, 3)**2))
h = mfTriSurf(tri, xyz, c)
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'))
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on'))
call msViewPause()

call msFreeArgs(nx, ny, nz, x, y, z, c, h, tri, xyz)

end program example
```

Result



See Also

mfSliceXYZ

mfGetSliceIJK, msGetSliceIJK

Retrieve orthogonal slice-plane(s) along i, j or k indices.

Module

```
use fgl
```

Syntax

```
call msGetSliceIJK(mfOut(tri, xyz), [x, y, z, ]c, Sx, Sy, Sz)
tri = mfGetSliceIJK([x, y, z, ]c, Sx, Sy, Sz)
```

Descriptions

Procedure `mfGetSliceXYZ` displays and retrieves orthogonal slice-planes along i, j and k which are index of `x,` index of `y` and index of `z` respectively.

It returns the triangular mesh `tri` and the vertex vectors of the sliced-planes defined in the n-by-3 matrix `xyz.`

For details on the input arguments, refer to the description of procedure `mfSliceIJK.`

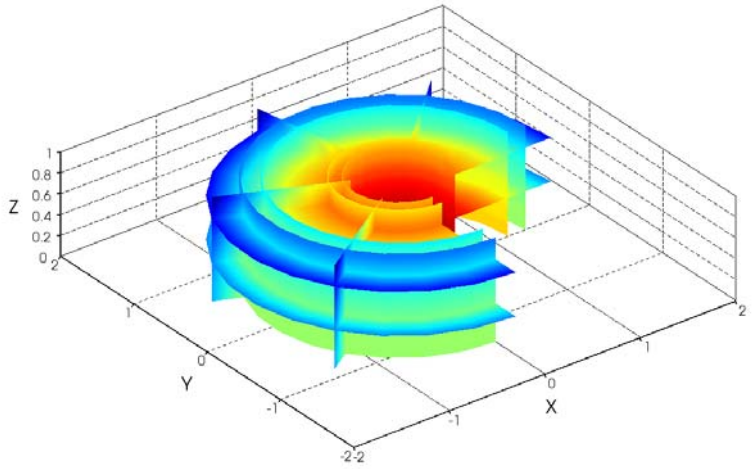
Example

Code

```
program Example

use fml
use fgl
implicit none

type(mfArray) :: nu, nv, nw, u, v, w, x, y, z, c, h, tri, xyz

nu = mfLinspace(0, 1.5d0*MF_PI, 20)
nv = mfLinspace(0, 2*MF_PI, 20)
nw = mfLinspace(0, 1, 20)
call msMeshGrid( mfOut(u, v, w), nu, nv, nw )
x = ( 1 + 0.6d0 * mfCos(v) ) * mfCos(u)
y = ( 1 + 0.6d0 * mfCos(v) ) * mfSin(u)
z = w
c = 1 - ( x ** 2 + y ** 2 + z ** 2)

call msGetSliceIJK(mfout(tri, xyz), x, y, z, c, mf((/4, 8, 12, 16/)),
&
                mf((/4, 8, 12, 16/)), mf((/8, 16/)))
c = 1 - ( mfS(xyz, MF_COL, 1)** 2 + mfS(xyz, MF_COL, 2) ** 2 + mfS(xyz,
MF_COL, 3) ** 2)
h = mfTriSurf(tri, xyz, c)
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'))
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on'))
call msAxis('equal')
call msCamZoom(1.5d0)
call msViewPause()

call msFreeArgs(nu, nv, u, v, x, y, z, c, h, tri, xyz)
```

```
end program example
```

Result



See Also

mfSliceIJK

mfGetSlicePlane, msGetSlicePlane

Retrieve orthogonal slice-plane(s) along arbitrary direction.

Module

```
use fgl
```

Syntax

```
call msGetSlicePlane(mfOut(tri, xyz), [x, y, z, ]c, Sx, Sy, Sz)
tri = mfGetSlicePlane([x, y, z, ]c, Sx, Sy, Sz)
```

Descriptions

Procedure `mfGetSliceXYZ` displays and retrieves orthogonal slice-planes of a specified set of volumetric data along arbitrary direction. It returns the triangular mesh `tri` and the vertex vectors of the sliced-planes defined in the n-by-3 matrix `xyz.`

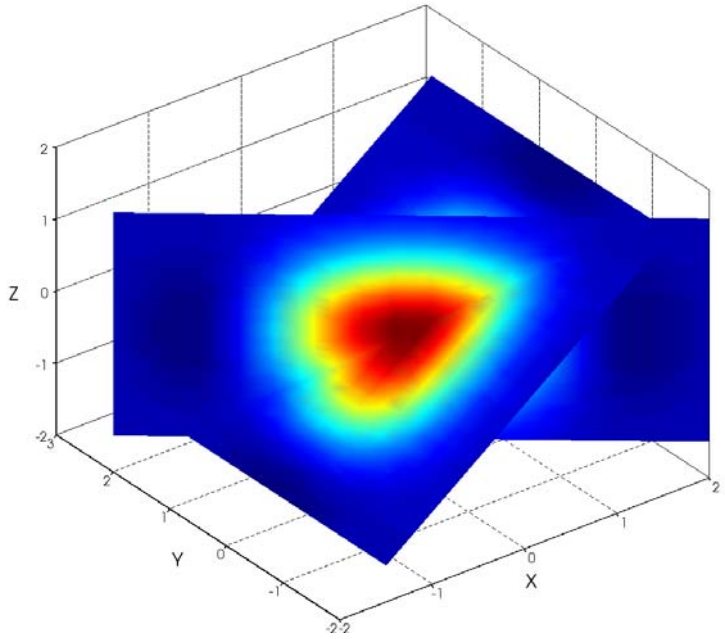
For details on the input arguments, refer to the description of procedure `mfSlicePlane.`

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: nx, ny, nz, x, y, z, c, h
type(mfArray) :: tri, xyz, c1

nx = mfLinspace(-2, 2.2d0, 21)
ny = mfLinspace(-2, 2.25d0, 17)
nz = mfLinspace(-1.5d0, 1.6d0, 31)
call msMeshgrid(mfout(y, x, z), ny, nx, nz)
c = 2*mfCos(x**2)*mfExp(-(y**2)-(z**2))

call msGetSlicePlane(mfout(tri, xyz), x, y, z, c, mf((/1, 0, -1, 0/)))
c1 = 2*mfCos(mfS(xyz, MF_COL, 1)**2)*mfExp(-(mfS(xyz, MF_COL,
2)**2)-(mfS(xyz, MF_COL, 3)**2))
h = mfTriSurf(tri, xyz, c1)
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'))
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on'))
call msHold('on')
call msGetSlicePlane(mfout(tri, xyz), x, y, z, c, mf((/1, 1, 0, 0/)))
c1 = 2*mfCos(mfS(xyz, MF_COL, 1)**2)*mfExp(-(mfS(xyz, MF_COL,
2)**2)-(mfS(xyz, MF_COL, 3)**2))
h = mfTriSurf(tri, xyz, c1)
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'))
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on'))
call msViewPause()

call msFreeArgs(nx, ny, nz, x, y, z, c, h, &
             tri, xyz, c1)

end program example
```

Result



See Also

mfSlicePlane

Streamline Graphs

mfStreamLine, msStreamLine

StreamLine from three-dimensional vector data.

Module

```
use fgl
```

Syntax

```
handle = mfStreamLine(x, y, z, u, v, w, Sx, Sy, Sz)
handle = mfStreamLine(u, v, w, Sx, Sy, Sz)
```

Descriptions

Procedure `mfStreamLine` creates streamlines from three-dimensional vector components `u, v` and `w.`

`call msStreamLine(x, y, z, u, v, w, Sx, Sy, Sz)`

- Arguments `u, v` and `w` are three-dimensional orthogonal vector components corresponding to the x, y and z-direction respectively.
- Arguments `x, y` and `z` define the coordinates for `u, v` and `w` and must be monotonic and three-dimensional plaid (as if produced by `mfMeshgrid).`
- Arguments `Sx, Sy` and `Sz` define the starting positions of the streamlines.

`call msStreamLine(u, v, w, Sx, Sy, Sz)`

- Arguments `x, y` and `z` are derived from `mfMeshgrid(mfOut(x, y, z), mfColon(1, n), mfColon(1, m), mfColon(1, p))`, where `n, m` and `p` are dimensions of `u.`

`h = mfStreamLine(...)`

- Handle `h` retrieves a handle to the streamline objects created by `mfStreamLine(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the streamline objects through handle `h` with procedure `msGSet.`

The properties available are:

1. start: a n-by-3 matrix containing starting point coordinates.
2. sizefactor: a double value containing the width factor.

See Also

mfQuiver, mfQuiver3

mfStreamDashedLine, msStreamDashedLine

StreamLine from three-dimensional vector data.

Module

```
use fgl
```

Syntax

```
handle = mfStreamDashedLine(x, y, z, u, v, w, Sx, Sy, Sz)
handle = mfStreamDashedLine(u, v, w, Sx, Sy, Sz)
```

Descriptions

Procedure `mfStreamDashedLine` creates a stream of dashed lines from three-dimensional vector components `u, v` and `w.`

`call msStreamDashedLine(x, y, z, u, v, w, Sx, Sy, Sz)`

- Arguments `u, v` and `w` are three-dimensional orthogonal vector components corresponding to the x, y and z-direction respectively.
- Arguments `x, y` and `z` define the coordinates for `u, v` and `w` and must be monotonic and three-dimensional plaid (as if produced by `mfMeshgrid).`
- Arguments `Sx, Sy` and `Sz` define the starting positions of the stream of dashed lines.

`call msStreamDashedLine(u, v, w, Sx, Sy, Sz)`

- Arguments `x, y` and `z` are derived from `mfMeshgrid(mfOut(x, y, z), mfColon(1, n), mfColon(1, m), mfColon(1, p))`, where `n, m` and `p` are dimensions of `u.`

`h = mfStreamDashedLine(...)`

- Handle `h` retrieves a handle to the stream of dashed line objects created by `mfStreamDashedLine(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the stream of dashed line objects through handle `h` with procedure `msGSet.`

The properties available are:

1. start: a n-by-3 matrix containing starting point coordinates.
2. sizefactor: a double value containing the width factor.

See Also

mfStreamRibbon, msStreamRibbon

Stream of ribbons from three-dimensional vector data.

Module

```
use fgl
```

Syntax

```
handle = mfStreamRibbon(x, y, z, u, v, w, Sx, Sy, Sz)
handle = mfStreamRibbon(u, v, w, Sx, Sy, Sz)
```

Descriptions

Procedure `mfStreamRibbon` creates a stream of ribbon objects from three-dimensional vector components `u, v` and `w.`

`call msStreamRibbon(x, y, z, u, v, w, Sx, Sy, Sz)`

- Arguments `u, v` and `w` are three-dimensional orthogonal vector components corresponding to the x, y and z-direction respectively.
- Arguments `x, y` and `z` define the coordinates for `u, v` and `w` and must be monotonic and three-dimensional plaid (as if produced by `mfMeshgrid).`
- Arguments `Sx, Sy` and `Sz` define the starting positions of the stream of ribbon objects.

`call msStreamRibbon(u, v, w, Sx, Sy, Sz)`

- Arguments `x, y` and `z` are derived from `mfMeshgrid(mfOut(x, y, z), mfColon(1, n), mfColon(1, m), mfColon(1, p))`, where `n, m` and `p` are dimensions of `u.`

`h = mfStreamRibbon(...)`

- Handle `h` retrieves a handle to the stream of ribbon objects created by `mfStreamRibbon(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the stream ribbon objects through handle `h` with procedure `msGSet.`

The properties available are:

1. start: a n-by-3 matrix containing starting point coordinates.
2. sizefactor: a double value containing the width factor.

pr

See Also

mfStreamTube, msStreamTube

Stream of tubes from three-dimensional vector data.

Module

```
use fgl
```

Syntax

```
handle = mfStreamTube(x, y, z, u, v, w, Sx, Sy, Sz)
handle = mfStreamTube(u, v, w, Sx, Sy, Sz)
```

Descriptions

Procedure `mfStreamTube` creates a stream of tube objects from three-dimensional vector components `u, v` and `w.`

`call msStreamTube(x, y, z, u, v, w, Sx, Sy, Sz)`

- Arguments `u, v` and `w` are three-dimensional orthogonal vector components corresponding to the x, y and z-direction respectively.
- Arguments `x, y` and `z` define the coordinates for `u, v` and `w` and must be monotonic and three-dimensional plaid (as if produced by `mfMeshgrid).`
- Arguments `Sx, Sy` and `Sz` define the starting positions of the stream of tube objects.

`call msStreamTube(u, v, w, Sx, Sy, Sz)`

- Arguments `x, y` and `z` are derived from `mfMeshgrid(mfOut(x, y, z), mfColon(1, n), mfColon(1, m), mfColon(1, p))`, where `n, m` and `p` are dimensions of `u.`

`h = mfStreamTube(...)`

- Handle `h` retrieves a handle to the stream of tube objects created by `mfStreamTube(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the stream tube objects through handle `h` with procedure `msGSet.`

The properties available are:

1. start: a n-by-3 matrix containing starting point coordinates.
2. sizefactor: a double value containing the width factor.

See Also

Triangular Surface Graphs

mfTriSurf, msTriSurf

Polygonal surface plot.

Module

```
use fgl
```

Syntax

```
handle = mfTriSurf(tri, x, y, z[, c])
handle = mfTriSurf(tri, xyz[, c])
```

Descriptions

Procedure `mfTriSurf` displays polygons defined by a face matrix. The polygons must be convex polygons.

```
call msTriSurf(tri, x, y, z)
call msTriSurf(tri, x, y, z, c)
```

- Display the polygons defined by a m-by-n face matrix `tri` as a surface object, where m is the number of polygons to be drawn and n is the number of edges of each polygon. For example, a 4-by-3 face matrix `tri` draws a surface object of 4 triangles and a 3-by-4 face matrix `tri` draws a surface object of 3 quadrilaterals.
- Each row of face matrix `tri` contains indices into `x, y` and `z` vertex vectors to define a single polygonal face.
- As with procedure `mfSurf,` the color scale is assumed to be proportional to the surface height specified by vertex `z.`
- Argument `c` overrides the default color specification and defines the new edge color.
- The shapes of the four mfArrays `x, y, z` and `c` should be conformed.
- Note that you can use `mfSurf` to plot the polygons and switch the shading mode using the toolbar function **shading mode**.

```
call msTriSurf(tri, xyz)
call msTriSurf(tri, xyz, c)
```

- Vertex vectors are defined in the n-by-3 matrix `xyz.`

```
h = mfTriSurf(...)
```

- Handle `h` retrieves a handle to the polygonal surface object created by `mfTriSurf(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the polygonal surface object through handle `h` with procedure

`msGSet.`

The properties available are:

1. tri
2. xyz

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: tri, x, y, z, c

x = mfRand(400,1)
y = mfRand(400,1)
z = 1 - ((x-0.5d0)**2 + (y-0.5d0)**2)
c = mfSin(z) * mfCos(z)
tri = mfGetDelaunay(x, y)

call msTitle('msTriSurf')
call msTriSurf(tri, x, y, z)
call msViewPause()

call msFreeArgs(tri, x, y, z, c)

end program example
```

See Also

mfTriMesh, msTriMesh

Polygonal mesh plot.

Module

```
use fgl
```

Syntax

```
handle = mfTriMesh(tri, x, y, z[, c])
handle = mfTriMesh(tri, xyz[, c])
```

Descriptions

Procedure `mfTriMesh` displays polygons in mesh defined by a face matrix. The polygons must be convex polygons.

```
call msTriMesh(tri, x, y, z)
call msTriMesh(tri, x, y, z, c)
```

- Display the polygons defined by a m-by-n face matrix `tri` as a meshed surface object, where m is the number of polygons to be drawn and n is the number of edges of each polygon. For example, a 4-by-3 face matrix `tri` draws a surface object of 4 triangles and a 3-by-4 face matrix `tri` draws a surface object of 3 quadrilaterals.
- Each row of face matrix `tri` contains indices into `x, y` and `z` vertex vectors to define a single polygonal face.
- As with the procedure `mfSurf,` the color scale is assumed to be proportional to the surface height specified by vertex `z.`
- Argument `c` overrides the default color specification and defines the new edge color.
- The shapes of the four mfArrays `x, y, z` and `c` should be conformed.
- Note that you can use `mfMesh` to plot the polygons and switch the shading mode using the toolbar function **shading mode**.

```
call msTriMesh(tri, xyz)
call msTriMesh(tri, xyz, c)
```

- Vertex vectors are defined in the n-by-3 matrix `xyz.`

```
h = mfTriMesh(...)
```

- Handle `h` retrieves a handle to the polygonal meshed surface object created by `mfTriMesh.`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the polygonal meshed surface object through handle `h` with

procedure `msGSet.`

The properties available are:

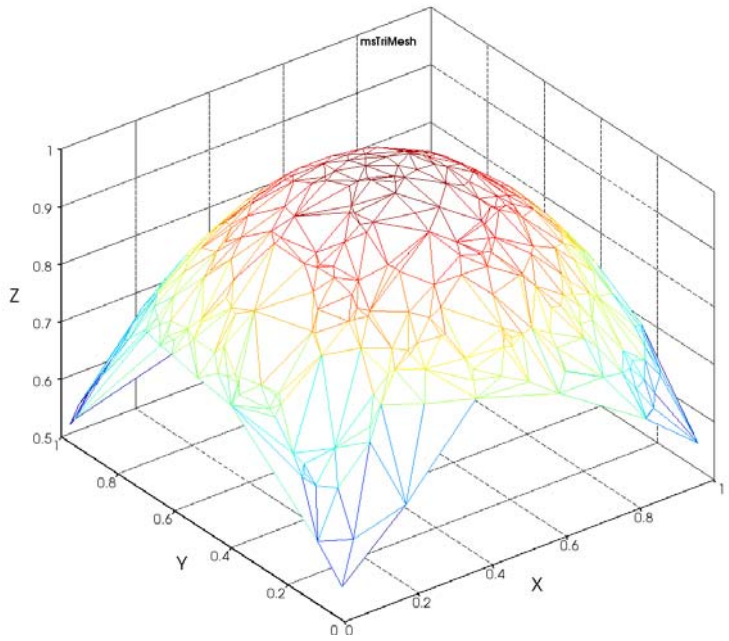
1. tri
2. xyz

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: tri, x, y, z, c

x = mfRand(400,1)
y = mfRand(400,1)
z = 1 - ((x-0.5d0)**2 + (y-0.5d0)**2)
c = mfSin(z) * mfCos(z)
tri = mfGetDelaunay(x, y)

call msTitle('msTriMesh')
call msTriMesh(tri, x, y, z)
call msViewPause()

call msFreeArgs(tri, x, y, z, c)

end program example
```

Result



See Also

mfTriContour

Contour on polygonal plot.

Module

```
use fgl
```

Syntax

```
handle = mfTriContour(tri, x, y, z[, c])
handle = mfTriContour(tri, xyz[, c])
```

Descriptions

Procedure `mfTriSurface` plots contour lines of matrix `z` on the polygons defined by a face matrix. The polygons must be convex polygons.

```
call msTriContour(tri, x, y, z)
call msTriContour(tri, x, y, z, c)
```

- Generate contour lines on the polygons for selected scalar values. The values plotted are selected automatically.
- As with procedures `mfTriSurf` and `mfTriMesh`, the polygons are defined by a m-by-n face matrix `tri`. Each row of face matrix `tri` contains indices into `x, y` and `z` vertex vectors to define a single polygonal face.
- As with the Surface Graphs procedures, the color scale is assumed to be proportional to the surface height specified by vertex `z`.
- Argument `c` overrides the default edge color specification, and defines the new edge color.
- The shapes of the four mfArrays `x, y, z` and `c` should be conformed.

```
call msTriContour(tri, xyz)
call msTriContour(tri, xyz, c)
```

- Vertex vectors are defined in the n-by-3 matrix `xyz`.

```
h = mfTriContour(...)
```

- Handle `h` retrieves a handle to the contour line objects created by `mfTriContour(...)`.
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the contour line objects through handle `h` with procedure `msGSet`.

The properties available are:

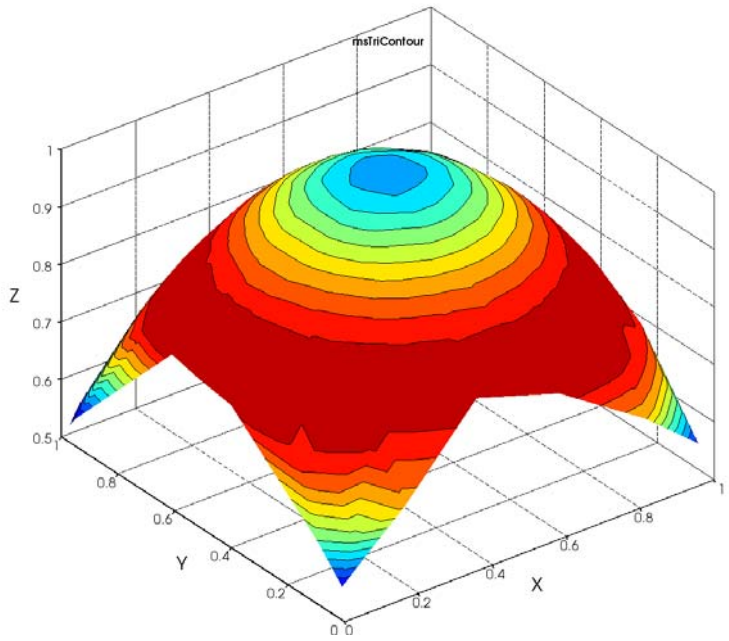
1. tri
2. xyz 3. iso: iso-values, a vector containing iso-value set. Setting this property will replace default set of contour lines. 4. autolevel: given number of levels, it will generate the iso-value set automatically.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: tri, x, y, z, c

x = mfRand(400,1)
y = mfRand(400,1)
z = 1 - ((x-0.5d0)**2 + (y-0.5d0)**2)
c = mfSin(z) * mfCos(z)
tri = mfGetDelaunay(x, y)

call msTitle('msTriContour')
call msTriContour(tri, x, y, z, c)
call msViewPause()

call msFreeArgs(tri, x, y, z, c)

end program example
```

Result



See Also

mfTriMesh

mfPatch, msPatch

Add patch on 2-D or 3-D coordinates.

Module

```
use fgl
```

Syntax

```
dle = mfPatch(x,y,c)
handle = mfPatch(x,y,z,c)
```

Descriptions

Procedure `mfPatch` adds the patch (filled 2-D polygon) on the coordinates specified by arguments `x` and `y`. Notice that only convex polygons can be accepted.

```
call msPatch(x, y)
call msPatch(x, y, c)
```

- Add the patch on the vertices defined by arguments `x` and `y`.
- If arguments `x` and `y` are matrices, then each column defines a single patch.
- Argument `c` defines the color scale for the vertices that determine the interior color of the patch.
- The shapes of the three mfArrays `x, y` and `c` should be conformed.

```
call msPatch(x, y, z)
call msPatch(x, y, z, c)
```

- Add the patch on the 3-D coordinates defined by arguments `x, y` and `z`.
- If `x, y` and `z` are matrices of the same size, then each column defines a single patch.
- The color scale is assumed to be proportional to the surface height specified by vertex `z` and is used to determine the interior color of the added patch.
- If argument `c` is defined, it overrides the default color specification and defines the new color scale for the vertices.
- The shapes of the four mfArrays `x, y, z` and `c` should be conformed.

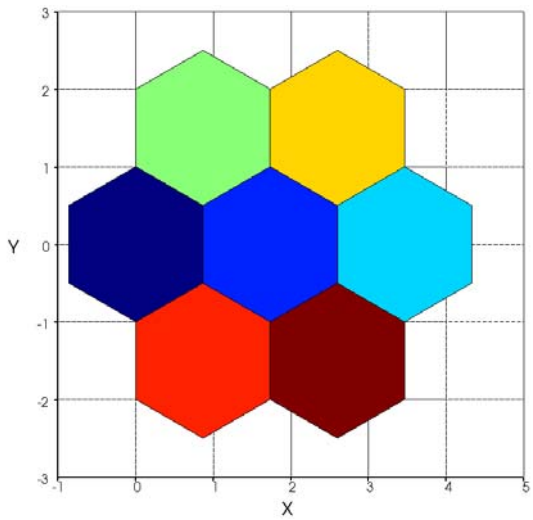
Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: x, y, c, h
real(8) :: p, q
```

```
p = 0.5d0*sqrt(3.0d0)
q = 1.5d0

x = (/0.0d0, p, p, 0.0d0, -p, -p  /) .vc. &
    (/2*p, 3*p, 3*p, 2*p, p, p    /) .vc. &
    (/4*p, 5*p, 5*p, 4*p, 3*p, 3*p/) .vc. &
     (/p, 2*p, 2*p, p, 0.0d0, 0.0d0/) .vc. &
    (/3*p, 4*p, 4*p, 3*p, 2*p, 2*p/) .vc. &
    (/p, 2*p, 2*p, p, 0.0d0, 0.0d0/) .vc. &
    (/3*p, 4*p, 4*p, 3*p, 2*p, 2*p/)

y = (/-1.0d0, -0.5d0, 0.5d0, 1.0d0, 0.5d0, -0.5d0/) .vc. &
    (/-1.0d0, -0.5d0, 0.5d0, 1.0d0, 0.5d0, -0.5d0/) .vc. &
     (/-1.0d0, -0.5d0, 0.5d0, 1.0d0, 0.5d0, -0.5d0/) .vc. &
    (/-1.0d0+q, -0.5d0+q, 0.5d0+q, 1.0d0+q, 0.5d0+q, -0.5d0+q/) .vc. &
    (/-1.0d0+q, -0.5d0+q, 0.5d0+q, 1.0d0+q, 0.5d0+q, -0.5d0+q/) .vc. &
    (/-1.0d0-q, -0.5d0-q, 0.5d0-q, 1.0d0-q, 0.5d0-q, -0.5d0-q/) .vc. &
    (/-1.0d0-q, -0.5d0-q, 0.5d0-q, 1.0d0-q, 0.5d0-q, -0.5d0-q/)

c = (/1, 1, 1, 1, 1, 1/) .vc. &
    (/2, 2, 2, 2, 2, 2/) .vc. &
    (/3, 3, 3, 3, 3, 3/) .vc. &
     (/4, 4, 4, 4, 4, 4/) .vc. &
     (/5, 5, 5, 5, 5, 5/) .vc. &
     (/6, 6, 6, 6, 6, 6/) .vc. &
     (/7, 7, 7, 7, 7, 7/)

x = .t. x
y = .t. y
c = .t. c

h = mfPatch(x, y, c)
call msView('2')
call msAxis('equal')
call msViewPause()

call msFreeArgs(x, y, c, h)

end program example
```

Result



See Also

Unstructured Grids

mfTetSurf, msTetSurf

Polyhedral surface plot.

Module

```
use fgl
```

Syntax

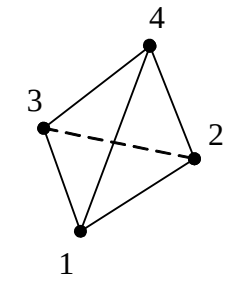
```
handle = mfTetSurf(tet, x, y, z[, c])
handle = mfTetSurf(tet, xyz[, c])
```

Descriptions

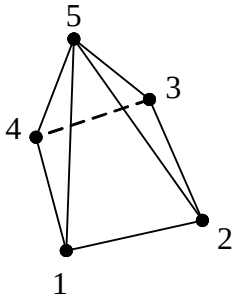
Procedure `mfTetSurf` displays polyhedrons defined by a cell matrix.

```
call msTetSurf(tet, x, y, z)
call msTetSurf(tet, x, y, z, c)
```

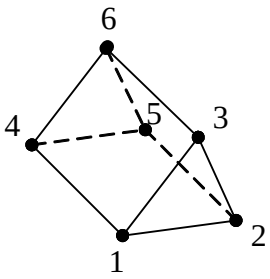
- Display the polyhedrons defined by a m-by-k cell matrix `tet` as a polyhedral object, where m is the number of polyhedrons to be drawn. There are four different types of polyhedrons depending on the value of k as illustrated below.



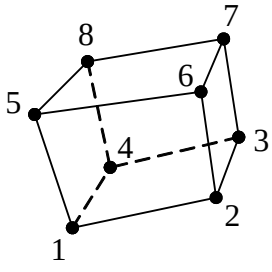
Tetrahedron (k=4)



Pyramid (k=5)



Wedge (k=6)



Hexahedron (k=8)

- Each row of cell matrix `tet` contains indices into `x, y` and `z` vertex vectors to define a single polyhedron.

- As with procedure `mfSurf,` the edge color is assumed to be proportional to the surface height specified by vertex `z.`
- Argument `c` overrides the default color specification and defines the new edge color.
- The shapes of the four mfArrays `x, y, z` and `c` should be conformed.
- Note that you can use either `mfTetSurf` or `mfTetMesh` to plot the polygons and switch the shading mode using the toolbar function **shading mode**.

```
call msTetSurf(tet, xyz)
call msTetSurf(tet, xyz, c)
```

- Vertex vectors are defined in the n-by-3 matrix `xyz` where n is the number of vertices.

```
h = mfTetSurf(...)
```

- Handle `h` retrieves a handle to the polyhedral object created by `mfTetSurf(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the polyhedral object through handle `h` with procedure `msGSet.`

The properties available are:

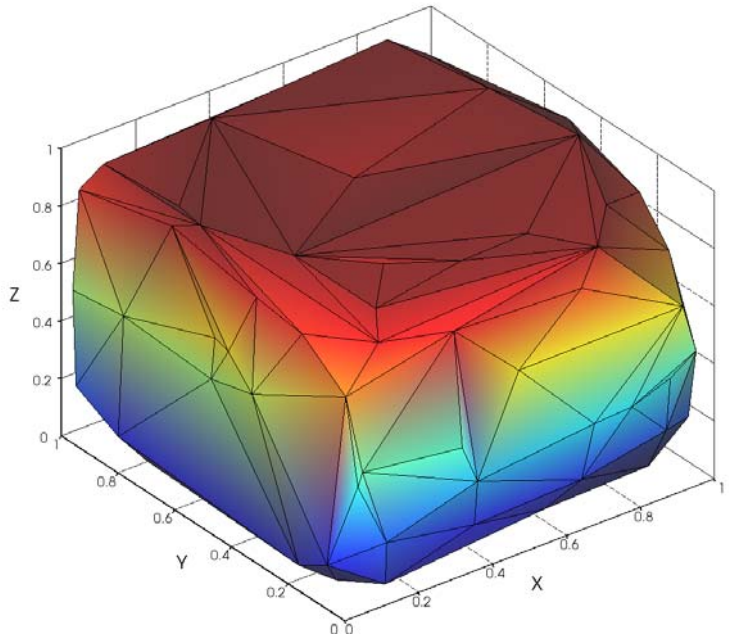
1. tet
2. xyz

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: tet, x, y, z, c

x = mfRand(500,1)
y = mfRand(500,1)
z = mfRand(500,1)
c = 1 - ((x - 0.5d0) ** 2 + (y - 0.5d0) ** 2 + (z - 0.5d0) ** 2)

tet = mfGetDelaunay3(x, y, z)

call msTetSurf(tet, x, y, z)
call msViewPause()

call msFreeArgs(tet, x, y, z, c)

end program example
```

Result



See Also

mfTetMesh, msTetMesh

Polyhedral mesh plot.

Module

```
use fgl
```

Syntax

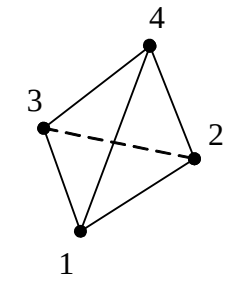
```
handle = mfTetMesh(tet, x, y, z[, c])
handle = mfTetMesh(tet, xyz[, c])
```

Descriptions

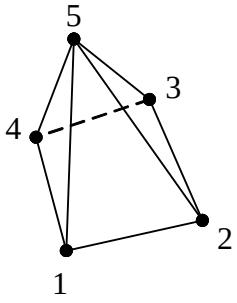
Procedure `mfTetSurf` displays polyhedrons defined by a cell matrix.

```
call msTetMesh(tet, x, y, z)
call msTetMesh(tet, x, y, z, c)
```

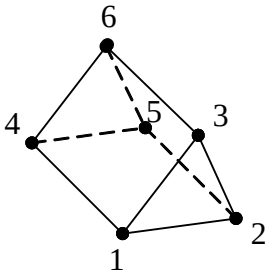
- Display the polyhedrons defined by a m-by-n cell matrix `tet` as a meshed polyhedral object, where m is the number of polyhedrons to be drawn. There are four different types of polyhedrons depending on the value of n as illustrated below.



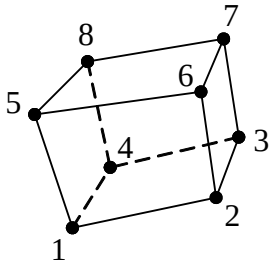
Tetrahedron (k=4)



Pyramid (k=5)



Wedge (k=6)



Hexahedron (k=8)

- Each row of cell matrix `tet` contains indices into `x, y` and `z` vertex vectors to define a single polyhedron.

- As with procedure `mfMesh,` the edge color is assumed to be proportional to the surface height specified by vertex `z.`
- Argument `c` overrides the default color specification and defines the new edge color.
- The shapes of the four mfArrays `x, y, z` and `c` should be conformed.
- Note that you can use either `mfTetSurf` or `mfTetMesh` to plot the polygons and switch the shading mode using the toolbar function **shading mode**.

```
call msTetMesh(tet, xyz)
call msTetMesh(tet, xyz, c)
```

- Vertex vectors are defined in the n-by-3 matrix `xyz.`

```
h = mfTetSurf(...)
```

- Handle `h` retrieves a handle to the polyhedral object created by `mfTetMesh(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the polyhedral object through handle `h` with procedure `msGSet.`

The properties available are:

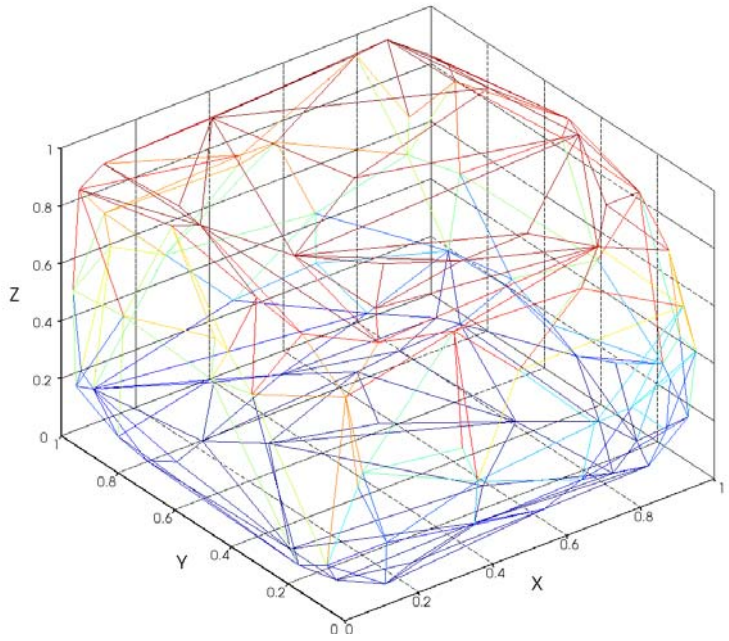
1. tet
2. xyz

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: tet, x, y, z, c

x = mfRand(500,1)
y = mfRand(500,1)
z = mfRand(500,1)
c = 1 - ((x - 0.5d0) ** 2 + (y - 0.5d0) ** 2 + (z - 0.5d0) ** 2)

tet = mfGetDelaunay3(x, y, z)

call msTetMesh(tet, x, y, z)
call msViewPause()

call msFreeArgs(tet, x, y, z, c)

end program example
```

Result



See Also

mfTetContour, msTetContour

Contour on polyhedral plot.

Module

```
use fgl
```

Syntax

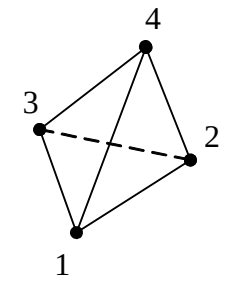
```
handle = mfTetContour(tet, x, y, z[, c])
handle = mfTetContour(tet, xyz[, c])
```

Descriptions

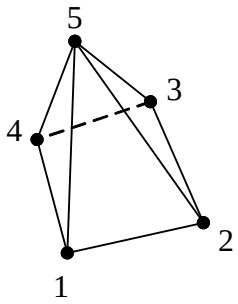
Procedure `mfTetSurface` plots contour lines on the surface of the polyhedral object defined by a face matrix.

```
call msTetContour(tet, x, y, z)
call msTetContour(tet, x, y, z, c)
```

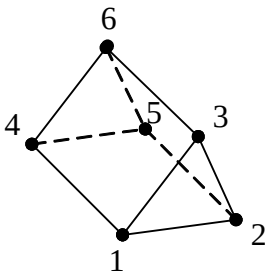
- Display the polyhedrons defined by a m-by-n cell matrix `tet` as a meshed polyhedral object, where m is the number of polyhedrons to be drawn. There are four different types of polyhedrons depending on the value of n as illustrated below.



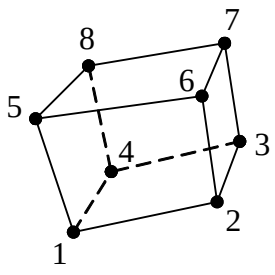
Tetrahedron (k=4)



Pyramid (k=5)



Wedge (k=6)



Hexahedron (k=8)

- Generate contour lines of matrix `z` on the surface of the polyhedral object for selected scalar values. The values plotted are selected automatically.

- Similar to procedures `mfTetSurf` and `mfTetMesh,` the polyhedral object is the combination of the polyhedrons defined by a m-by-n face matrix `tet.` Each row of face matrix `tet` contains indices into `x, y` and `z` vertex vectors to define a single polyhedron.
- The color scale is assumed to be proportional to the surface height specified by vertex `z.`
- Argument `c` overrides the default color specification and defines the new edge color.

- The shapes of the four mfArrays `x, y, z` and `c` should be conformed.

```
call msTetContour(tet, xyz)
call msTetContour(tet, xyz, c)
```

- Vertex vectors are defined in the n-by-3 matrix `xyz.`

```
h = mfTetContour(...)
```

- Handle `h` retrieves a handle to the contour line objects created by `mfTetContour(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the contour line objects through handle `h` with procedure `msGSet.`

The properties available are:

1. tet
2. xyz 3. iso: iso-values, a vector containing iso-value set. Setting this property will replace default set of contour lines. 4. autolevel: given number of levels, it will generate the iso-value set automatically.

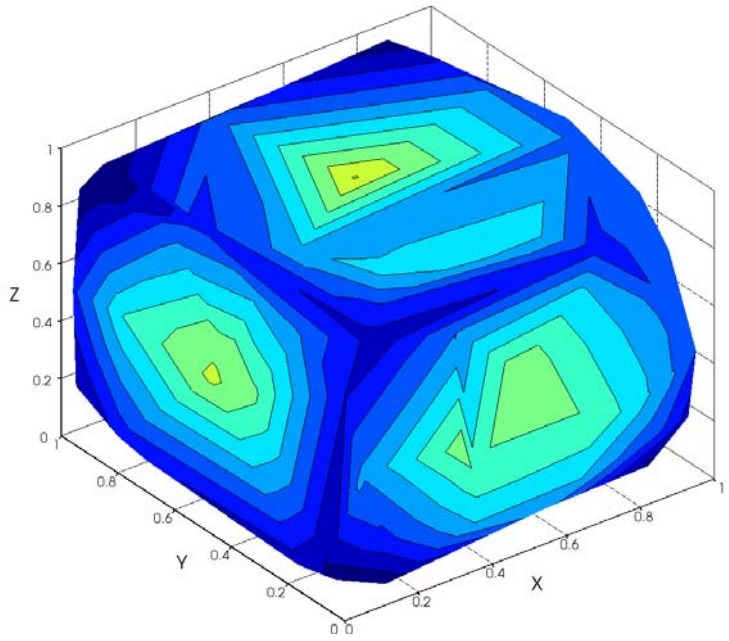
Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: tet, x, y, z, c

x = mfRand(500,1)
y = mfRand(500,1)
z = mfRand(500,1)
c = 1 - ((x - 0.5d0) ** 2 + (y - 0.5d0) ** 2 + (z - 0.5d0) ** 2)

tet = mfGetDelaunay3(x, y, z)

call msTetContour(tet, x, y, z, c)
call msViewPause()

call msFreeArgs(tet, x, y, z, c)
```

```
end program example
```

Result



See Also

mfTetMesh

mfTetIsoSurface, msTetIsoSurface

Polyhedral isosurface plot.

Module

```
use fgl
```

Syntax

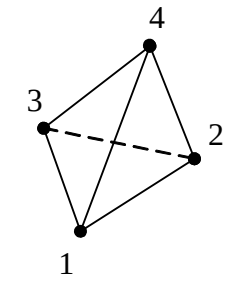
```
handle = mfTetIsoSurface(x, y, z, c, isovalue)
handle = mfTetIsoSurface(c, isovalue)
```

Descriptions

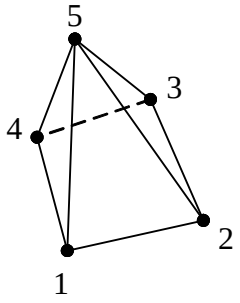
Procedure `mfTetIsoSurface` creates 3-D graphs composed of isosurface data from the volume data `c` at the isosurface value specified in argument `isovalue.`

```
call msTetIsoSurface(x, y, z, c, isovalue)
```

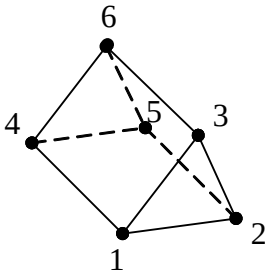
- Display the polyhedrons defined by a m-by-n cell matrix `tet` as a meshed polyhedral object, where m is the number of polyhedrons to be drawn. There are four different types of polyhedrons depending on the value of n as illustrated below.



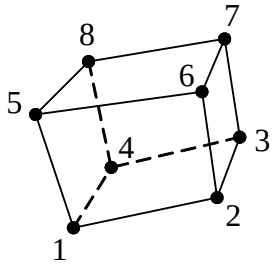
Tetrahedron (k=4)



Pyramid (k=5)



Wedge (k=6)



Hexahedron (k=8)

- The arguments `x, y` and `z` define the coordinates for the volume `c.`

```
call msTetIsoSurface(c, isovalue)
```

- The coordinates for the volume `c` is of a geometrically rectangular grid where `x = mfColon(1, n)` and `y = mfColon(1, m) and z = mfColon(1, p)`.

`h = mfTetIsoSurface(...)`

- Handle `h` retrieves a handle to the isosurface object created by `mfTetIsoSurface(...)`.
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the isosurface object through handle `h` with procedure `msGSet`.

The properties available are:

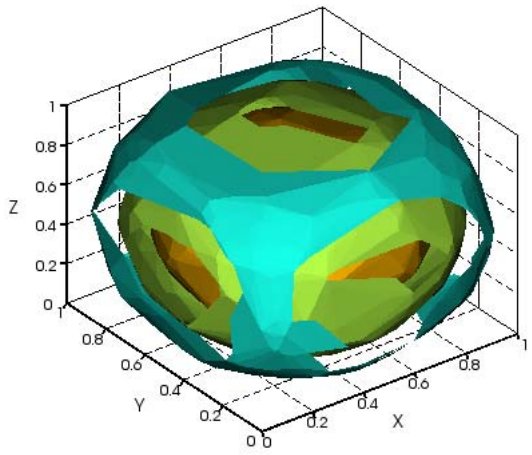
1. iso: iso-value, a vector containing iso-value sets.

Example

Code

```
program Example_msTetIsoSurface

use fml
use fgl
implicit none

type (mfArray) :: tet, x, y, z, c

x = mfRand(500,1)
y = mfRand(500,1)
z = mfRand(500,1)
c = 1 - ((x - 0.5d0) ** 2 + (y - 0.5d0) ** 2 + (z - 0.5d0) ** 2)

tet = mfGetDelaunay3(x, y, z)

call msTetIsoSurface(tet, x, y, z, c, mf((/0.8d0, 0.7d0, 0.6d0/)))
call msViewPause()

call msFreeArgs(tet, x, y, z, c)

end program Example_msTetIsoSurface
```

Result



See Also

mfTetSliceXYZ, msTetSliceXYZ

Display orthogonal slice-planes through volumetric data.

Module

```
use fgl
```

Syntax

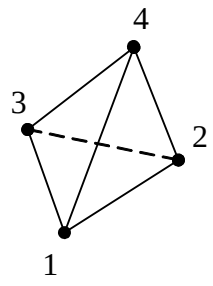
```
call msTetSliceXYZ(tet, [x, y, z, ]c, Sx, Sy, Sz)
handle = mfTetSliceXYZ(tet, [x, y, z, ]c, Sx, Sy, Sz)
```

Descriptions

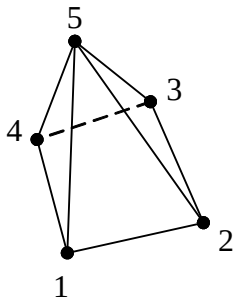
Procedure mfSliceXYZ displays orthogonal slice-planes of a specified set of volumetric data. The information on the slice-planes can be retrieved by using procedure `mfGetSliceXYZ`.

`call msTetSliceXYZ(tet, x, y, z, c, Sx, Sy, Sz)`

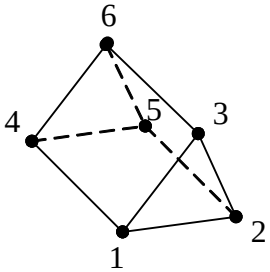
- Display the polyhedrons defined by a m-by-n cell matrix `tet` as a meshed polyhedral object, where m is the number of polyhedrons to be drawn. There are four different types of polyhedrons depending on the value of n as illustrated below.



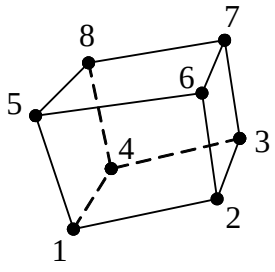
Tetrahedron (k=4)



Pyramid (k=5)



Wedge (k=6)



Hexahedron (k=8)

- Displays orthogonal slice-planes along the x, y and z directions specified by points in vector mfArrays `Sx, Sy` and `Sz`

- Use `mf()` to substitute any of the direction vectors `Sx, Sy` or `Sz` if you are not drawing any slice along the respective direction. E.g. call msSliceXYZ(x, y, z, v, Sx, mf(), Sz) draws slices along the x-axis as specified by `Sx` and z-axis as specified by `Sz.`
- The arguments `x, y` and `z` define the corresponding coordinates of scalar values mfArray `c,` where `c` is an m-by-n-by-p three-dimensional array.
- The arguments `x, y` and `z` must be of the same shape as `c` and are monotonic and three-dimensional plaid as if produced by procedure `mfMeshgrid.`
- The color at each point is determined by three-dimensional interpolation into the elements of volume `c,` mapped to the current colormap.

```
call msTetSliceXYZ(tet, c, Sx, Sy, Sz)
```

- Assumes `x` is composed of mfColon(1, n) vectors, `y` is composed of mfColon(1, m) vectors and `z` is composed of mfColon(1, p) vectors.

```
h = mfTetSliceXYZ(...)
```

- Handle `h` retrieves a handle to the volumetric slice object created by `mfSliceXYZ(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()`to retrieve the handle of the current graphics object.

You can specify properties of the volumetric slice objects through handle `h` with procedure `msGSet.`

The properties available are:

1. tet 2. slicex: specifies the slice-planes along the x direction.
3. slicey: specifies the slice-planes along the y direction.
4. slicez: specifies the slice-planes along the z direction.

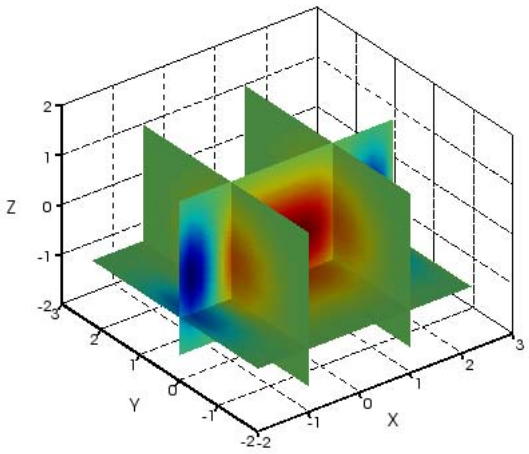
Example

Code

```
program Example_msTetSliceXYZ

use fml
use fgl
implicit none

type(mfArray) :: nx, ny, nz, x, y, z, c, tet

nx = mfLinspace(-2, 2.2d0, 21)
ny = mfLinspace(-2, 2.25d0, 17)
nz = mfLinspace(-1.5d0, 1.6d0, 31)
call msMeshgrid(mfout(y, x, z), ny, nx, nz)
c = 2*mfCos(x**2)*mfExp(-(y**2)-(z**2))

tet = mfGetDelaunay3(x, y, z);
```

```
call msTetSliceXYZ(tet, x, y, z, c, mf((/-1.0d0, 1.0d0/)), &
             mf(0), mf(-.75d0))
call msViewPause()
call msFreeArgs(nx, ny, nz, x, y, z, c)

end program Example_msTetSliceXYZ
```

Result



See Also

mfTetSlicePlane, mfTetIsoSurface

mfTetSlicePlane, msTetSlicePlane

Display orthogonal slice-planes along arbitrary direction.

Module

```
use fgl
```

Syntax

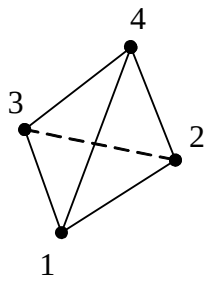
```
handle = mfTetSlicePlane(x, y, z, c, plane)
handle = mfTetSlicePlane(c, plane)
```

Descriptions

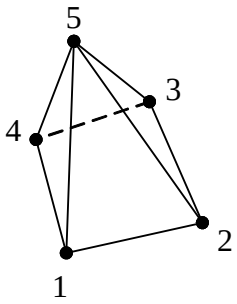
Procedure `mfSlice` displays orthogonal slice-planes of a specified set of volumetric data along arbitrary direction. The information on the slice-planes can be retrieved by using procedure `mfGetSlicePlane.`

```
call msTetSlicePlane(tet, x, y, z, c, plane)
```

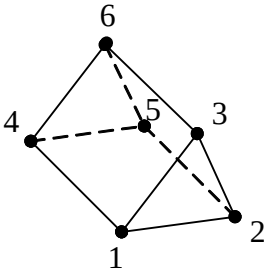
- Display the polyhedrons defined by a m-by-n cell matrix `tet` as a meshed polyhedral object, where m is the number of polyhedrons to be drawn. There are four different types of polyhedrons depending on the value of n as illustrated below.



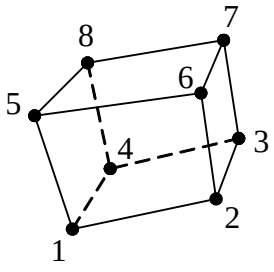
Tetrahedron (k=4)



Pyramid (k=5)



Wedge (k=6)



Hexahedron (k=8)

- Displays orthogonal slice-planes along the direction specified by `plane.`
- The arguments `x, y` and `z` are structured grid data which define the corresponding

coordinates of scalar values specified in argument `c,` where `c` is a m-by-n-by-p three-dimensional array.

- Argument `plane` is a vector of size 4 representing the coefficients of the sliced plane equation, i.e. [a, b, c, d], where ax + by + cz + d = 0.

- The arguments `x, y` and `z` must be of the same shape as `c` and are monotonic and three-dimensional plaid as if produced by procedure `mfMeshgrid.`
- The color at each point is determined by three-dimensional interpolation into the elements of volume `c,` mapped to the current colormap.

```
call msTetSlicePlane(tet, c, plane)
```

- Assumes `x` is composed of mfColon(1, n) vectors, `y` is composed of mfColon(1, m) vectors and `z` is composed of mfColon(1, p) vectors.

```
h = mfTetSlicePlane(...)
```

- Handle `h` retrieves a handle to the volumetric slice objects created by `mfSlicePlane(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the volumetric slice objects through handle `h` with procedure `msGSet.`

The property available is:

1. tet 2. plane

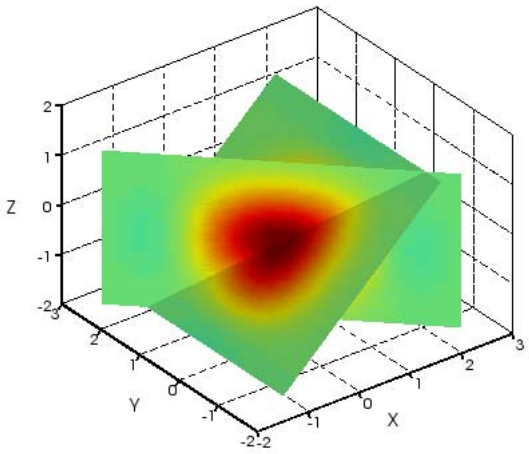
Example

Code

```
program Example_msTetSlicePlane

use fml
use fgl
implicit none

type(mfArray) :: nx, ny, nz, x, y, z, c, tet

nx = mfLinspace(-2, 2.2d0, 21)
ny = mfLinspace(-2, 2.25d0, 17)
nz = mfLinspace(-1.5d0, 1.6d0, 31)
call msMeshgrid(mfout(y, x, z), ny, nx, nz)
c = 2*mfCos(x**2)*mfExp(-(y**2)-(z**2))

tet = mfGetDelaunay3(x, y, z)

call msTetSlicePlane(tet, x, y, z, c, mf((/1, 0, -1, 0/)))
call msHold('on')
call msTetSlicePlane(tet, x, y, z, c, mf((/1, 1, 0, 0/)))
call msViewPause()
```

```
call msFreeArgs(nx, ny, nz, x, y, z, c, tet)

end program Example_msTetSlicePlane
```

Result



See Also

mfSlicePlane, mfGetSlicePlane

Unstructured Point Set

mfPoint, msPoint

Display input points in three-dimensional space.

Module

```
use fgl
```

Syntax

```
handle = mfPoint(x, y, z[, c])
handle = mfPoint(xyz[, c])
```

Descriptions

Procedure `mfPoint` plots a set of input points in three-dimensional space.

```
call msPoint(x, y, z)
call msPoint(x, y, z, c)
```

- Arguments `x, y` and `z` contain the x-, y-, and z- coordinates of the points respectively. They can be either matrices or vectors of the same shape.
- By default, the scalar value of the each point is the height which is specified in argument `z.` Specifying the scalar vector `c` would override the default scalar values.

```
call msPoint(xyz)
call msPoint(xyz, c)
```

- Vertex vectors are defined in the n-by-3 matrix `xyz.`

```
h = mfPoint(...)
```

- Handle `h` retrieves a handle to the points created by `mfPoint(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the points through handle `h` with procedure `msGSet.`

The property available is:

1. xyz

See Also

mfDelaunay, msDelaunay, mfGetDelaunay, msGetDelaunay

2-D Delaunay triangulation of input points.

Module

```
use fgl
```

Syntax

```
h = mfDelaunay(x, y[, bx1, bx2, ...])
tri = mfGetDelaunay(x, y[, p1, p2, ...])
call msGetDelaunay(mfOut(tri, xyz), x, y[, p1, p2, ...])
```

Descriptions

Procedure `mfDelaunay` is a filter that constructs a two-dimensional Delaunay triangulation from a set of input points. You may use procedure `mfGetDelaunay` to retrieve the output of the filter, the triangular mesh.

```
h = mfDelaunay(x, y, bx1, by1, bx2, by2, bx3, by3)
```

- Arguments `x` and `y` specify the x- and y- coordinates of the input points.
- You may define the boundaries which the Delaunay triangulation is constructed upon. Each boundary is defined by two vertex vectors specified in the arguments. For instance, arguments `bx1` and `by1` define the first boundary, arguments `bx2` and `by2` define the second boundary and so on.
- Notice that the order of the boundary points determines how the Delaunay triangulation is constructed. If the boundary points are specified counterclockwise, then the Delaunay triangulation is constructed within the boundary; if the boundary points are specified clockwise, then the Delaunay triangulation is constructed beyond the boundary. On the other hand, an unexpected result may arise if a false order of boundary points is given.

```
 tri = mfGetDelaunay(x, y, bx1, by1, bx2, by2, bx3, by3)
 call msGetDelaunay(mfOut(tri, xyz), x, y, bx1, by1, bx2, by2,
bx3, by3)
```

- Retrieve the triangular mesh `tri.` With `tri` and the coordinates `xyz,` you may use `mfTriSurf` to plot the triangular surface.

```
h = mfDelaunay(...)
```

- Handle `h` retrieves a handle to the polygonal surface object created by `mfDelaunay(...).`

- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the polygonal surface object through handle `h` with procedure `msGSet`.

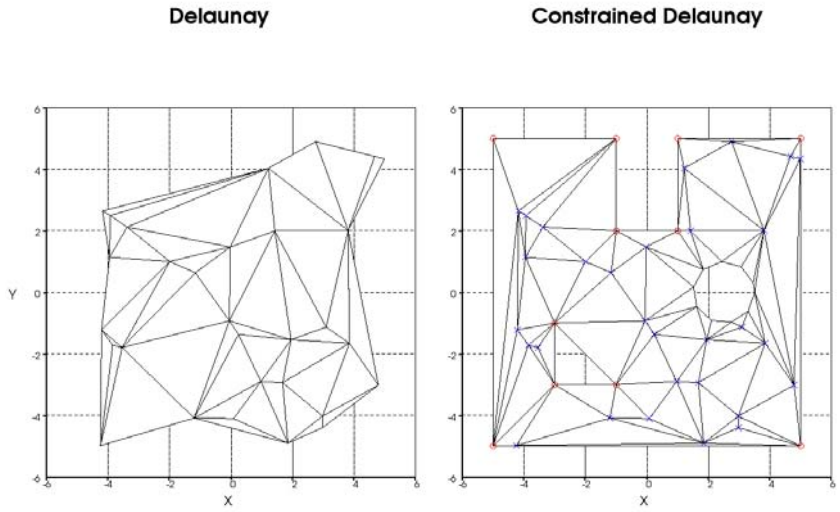
Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: x, y, n, h
type (mfArray) :: bx1, by1, bx2, by2, bx3, by3
integer :: i
real(8) :: rx, ry

bx1 = .t. (/ -5, 5,  5, 1, 1, -1, -1, -5/)
by1 = .t. (/ -5, -5,  5, 5, 2, 2, 5, 5/)
bx2 = .t. (/ -3,  -3, -1 /)
by2 = .t. (/ -3,  -1, -3 /)
n = .t. mfLinspace(0, -1.9*MF_PI, 10)
bx3 = 2.5d0 + mfCos(n)
by3 = mfSin(n)


x = mfZeros(30,1)
y = mfZeros(30,1)
call random_seed()
do i=1,30
    do while (.true.)
        call random_number(rx)
        call random_number(ry)
        rx= rx*10 - 5
        ry= ry*10 - 5
        if ( (rx<5 .or. rx>-5 ) .and. (ry<5 .or. ry>-5) .and. (rx<-1 .or.
rx>1 .or. ry<2) &
            .and. (rx<-3 .or. ry<-3 .or. rx + ry > -2) .and.
( (rx-2.5d0)**2 + ry**2 > 1) ) then
            exit
        end if
    end do

    call msAssign(mfS(x,i,1), rx)
    call msAssign(mfS(y,i,1), ry)
end do

call msFigure('Delaunay');
call msSubplot(1, 2, 1)
call msTitle('Delaunay')
h = mfDelaunay(x, y)
call msAxis('equal')

call msSubplot(1, 2, 2)
call msTitle('Constrained Delaunay')
h = mfDelaunay(x, y, bx1, by1, bx2, by2, bx3, by3)
call msHold('on')
h = mfPlot(bx1, by1, "or", bx2, by2, "or", x, y, "xb")
call msAxis('equal')

call msViewPause()
call msFreeArgs(x, y, n, h, bx1, by1, bx2, by2, bx3, by3)
```

```
end program example
```

Result



See Also

mfDelaunay3, msDelaunay3, mfGetDelaunay3, msGetDelaunay3

3-D Delaunay triangulation of input points.

Module

```
use fgl
```

Syntax

```
h = mfDelaunay3(x, y, z)
h = mfDelaunay3(xyz)
tet = mfGetDelaunay3(x, y, z)
tet = mfGetDelaunay3(xyz)
call msGetDelaunay3(mfOut(tet, xyz), ...)
```

Descriptions

Procedure `mfDelaunay3` is a filter that constructs a three-dimensional Delaunay triangulation from a set of input points. You may use procedure to retrieve the output of the filter, the tetrahedral mesh.

```
h = mfDelaunay3(x, y, z)
tet = mfGetDelaunay3(x, y, z)
```

- Arguments `x, y` and `z` specify the x-, y- and z- coordinates of the input points.

```
h = mfDelaunay3(xyz)
[tet, xyz] = mfGetDelaunay3(xyz)
```

- Vertex vectors tet are defined in the n-by-3 matrix `xyz.`

```
tet = mfGetDelaunay3(xyz)
[tet, xyz] mfGetDelaunay3(xyz)
```

- Retrieve the tetrahedral mesh `tet.` With `tet` and the coordinates `xyz,` you may use `mfTetSurf` to plot the tetrahedral surface.

```
h = mfDelaunay3(...)
```

- Handle `h` retrieves a handle to the polyhedral object created by `mfDelaunay3(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the polyhedral object through handle `h` with procedure `msGSet.`

The property available is:

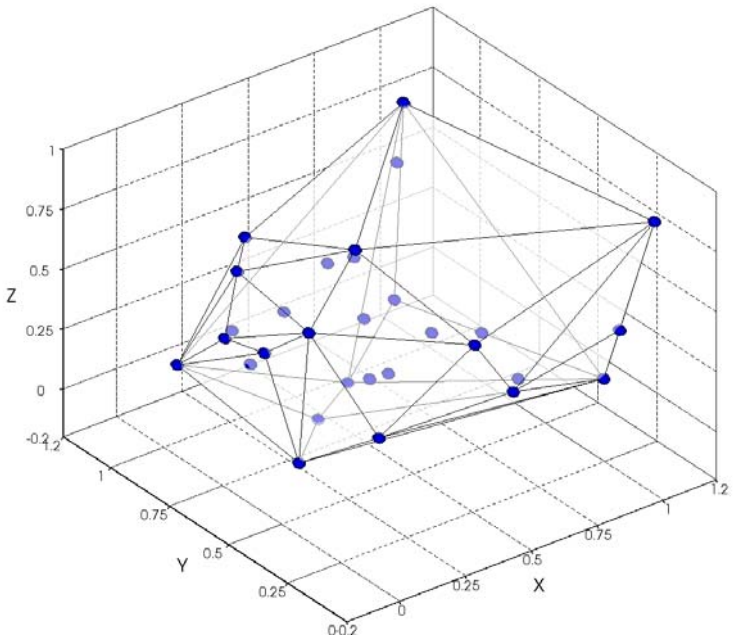
1. xyz

Example

Code

```
Program example

use fgl
use fml
implicit none

type(mfArray) :: xyz, h

xyz = mfRand(30, 3)

call msFigure('Delaunay 3D')
call msTitle('Delaunay 3D')
h = mfDelaunay3( xyz )
call msDrawMaterial(h, 'surf', 'trans', 50)
call msHold('on')
h = mfSphere( xyz, mf(0.02), mf((/0, 0, 1/)) )
call msViewPause()

call msFreeArgs(xyz, h)

end Program example
```

Result



See Also

Velocity Vectors

mfQuiver, msQuiver

Two-dimensional velocity vectors.

Module

```
use fgl
```

Syntax

```
handle = mfQuiver(x, y, u, v[, scale])
handle = mfQuiver(u, v[, scale])
```

Descriptions

Procedure `mfQuiver` plots velocity vectors as arrows with components `(u, v)` at the points `(x, y)`.

```
handle = mfQuiver(x, y, u, v)
handle = mfQuiver(x, y, u, v, scale)
```

- mfArrays `x` and `y` contain positions of the velocity vectors, while the mfArrays `u` and `v` contain the corresponding velocity components.
- You can control the vector scaling by specifying argument `scale.` Specifying `scale` as 0.5 would reduce the relative length of the vector by half.
- The shapes of the four mfArrays `x, y, u` and `v` must conform i.e. All are m-by-n mfArrays.

```
handle = mfQuiver(u, v)
handle = mfQuiver(u, v, scale)
```

- velocity vectors are plotted over a geometrically rectangular grid where `x = mfColon(1, n)` and `y = mfColon(1, m)`.

```
h = mfQuiver(...)
```

- Handle `h` retrieves a handle to the quiver object created by `mfQuiver(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the quiver object through handle `h` with procedure `msGSet.`

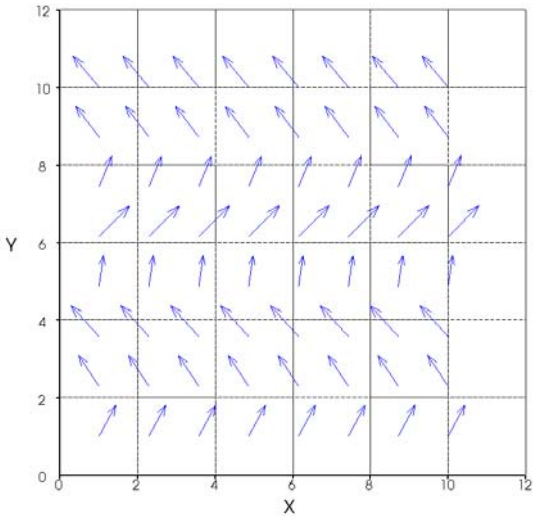
Example

Code

```
program example

use fml
use fgl
```

```
implicit none

type(mfArray) :: u, v, x, y, a

a = mfLinspace(1, 10, 8)
call msMeshgrid(mfout(x, y), a)
u = 4*mfCos(y)
v = 4*mfOnes(8, 8)

call msQuiver(x, y, u, v, mf(0.2))
call msAxis('equal')
call msCamZoom(0.8d0)
call msViewPause()

call msFreeArgs(u, v, x, y, a)

end program example
```

Result



See Also

mfQuiver3

mfQuiver3, msQuiver3

Three-dimensional velocity vectors.

Module

```
use fgl
```

Syntax

```
handle = mfQuiver3(x, y, z, u, v, w[, scale])
handle = mfQuiver3(z, u, v, w[, scale])
```

Descriptions

Procedure `mfQuiver3` plots velocity vectors as arrows with components `(u, v, w)` at the points `(x, y, z)` in three-dimensional space.

```
handle = mfQuiver3(x, y, z, u, v, w)
handle = mfQuiver3(x, y, z, u, v, w, scale)
```

- mfArrays `x, y` and `z` contain corresponding positions of velocity vectors, which are specified by mfArrays `u, v` and `w` corresponding to the three-dimensional orthogonal velocity components.
- You can control the vector scaling by specifying argument `scale`. Specifying `scale` as 0.5 would reduce the relative length of the vector by half. By default, `scale` = 0.
- The shapes of the four mfArrays `x, y, u` and `v` must be conformed, i.e. All are m-by-n mfArrays.

```
call msQuiver3(z, u, v, w)
call msQuiver3(z, u, v, w, scale)
```

- The height data is assumed to be defined over a geometrically rectangular grid where `x = mfColon(1, n)` and `y = mfColon(1, m)`

```
h = mfQuiver3(...)
```

- Handle `h` retrieves a handle to the velocity vector objects created by `mfQuiver3(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the velocity vector objects through handle `h` with procedure `msGSet.`

The property available is:

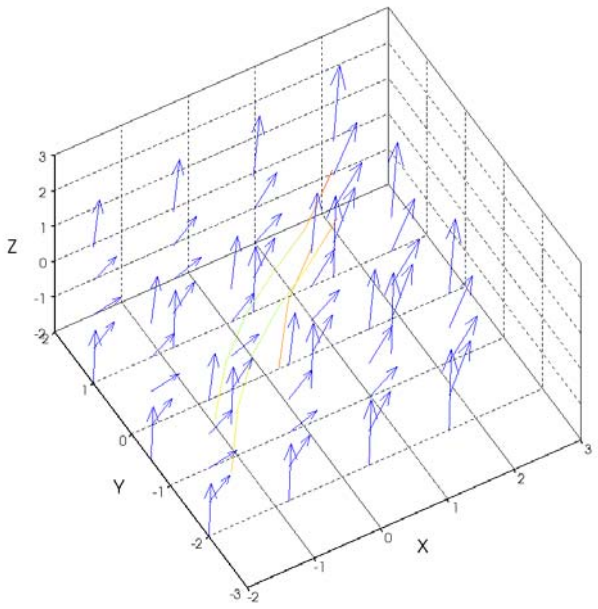
1. symbol: `"arrow", "cone"` or `"flat_arrow"`

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: a, b, c, x, y, z, v, u, w

a = mfLinspace(-2, 1.6d0, 4)
b = mfLinspace(-2, 1, 3)
c = mfLinspace(-2, 1.84d0, 5)
call msMeshgrid(mfout(x, y, z), a, b, c)
u = mfOnes(3, 4, 5)
v = 0.4d0*(z**2)
w = mfExp(0.5d0*x)

call msQuiver3(x, y, z, u, v, w)
call msHold('on')
call msStreamline(x,y,z,u,v,w, mf((/-1.2,0.0,0.2/)), &
                  mf((/-1.2,0.5,0.0/)), mf((/-2.0,-1.0,-2.0/)))
call msView(-30,50)
call msViewPause()

call msFreeArgs(x, y, z, v, u, w)

end program example
```

Result



See Also

mfQuiver

Image

mfImage

Display image file.

Module

```
use fgl
```

Syntax

```
handle = mfImage(ax)
```

Descriptions

Procedure `mfImage` displays an image file in the plot space. Argument `ax` is an m-by-n-by-3 matrix containing the rgb color codes.

See Also

mfImRead, mfImWrite

mfImRead

Read in image file.

Module

```
use fgl
```

Syntax

```
ax = mfImRead(filename)
```

Descriptions

Procedure `mfImRead` reads in an image file with the name specified by argument `filename` and stores it in argument `ax` which is an m-by-n-b-3 matrix storing the rgb color codes.

The supported file formats are: bmp, jpeg and png.

See Also

mfImage, mfImWrite

msImWrite

Write to image file.

Module

```
use fgl
```

Syntax

```
call msImWrite(filename, ax)
```

Descriptions

Procedure `msImWrite` writes the rgb color codes stored in the m-by-n-by-3 matrix `ax` into a file with the name specified by argument `filename.`

The supported file format are: bmp, jpeg, tiff, ps and png.

See Also

Elementary 3D Objects

mfMolecule, msMolecule

Draw stick and ball model of molecules.

Module

```
use fgl
```

Syntax

```
handle = mfMolecule(loc, conn[, rad][, color][, stick_rad][, stick_col][, resolution])
```

Descriptions

Procedure `Molecule` enables you to create three-dimensional stick-and-model of molecules.

```
call msMolecule(loc, conn, rad, color, stick_rad, stick_col, resolution)
```

- Draw n balls specified by argument `loc` and m sticks specified by argument `conn.` Argument `rad` specifies the radius of each ball, argument `color` specifies the color of each ball, argument `stick_rad` specifies the cylindrical radius of each stick, argument `stick_col` specifies the color of the sticks and argument `resolution` specifies the smoothness of the ball objects.
- Argument `loc` is an n-by-3 array containing the three-dimensional Cartesian coordinates (x, y, z) of each ball center, hence specifying the spatial relationship of each ball. Each ball is numbered according to their respective row index. Thus row number 1 specifies ball number 1. The first column contains the x-coordinates, the second column contains the y-coordinates and the third the z-coordinates respectively. As an example, the array below specifies three balls whose center are located at (0,0,0), (1,1,1) and (0,1,0) respectively.

Argument `loc:`

	x	y	z
ball 1	0	0	0
ball 2	1	1	1
ball 3	0	1	0

- Argument `conn` is an m-by-2 array specifying m number of sticks and the balls connected by each stick. Each stick connects two balls and is labeled according to its row number. Columns of the argument `conn` contain the indices of the balls that each

stick connects. For example, the array below specifies 3 sticks connecting ball 1 and ball 2, ball 3 and ball 1, ball 2 and ball 3.

Argument `conn:`

	ball index	ball index
stick 1	1	2
stick 2	3	1
stick 3	2	3

- Argument `rad` is a scalar specifying the radius of all balls or an n-by-1 array specifying the radius of each individual ball respectively. By default, all balls are drawn with a radius of 0.5. As an example, the array below specifies three balls of different sizes, with radius 1, 2 and 3 respectively.

Argument `rad:`

	radius
ball 1	1
ball 2	2
ball 3	3

- Argument `color` contains a string specifying the color of all the balls or an n-by-3 array containing the rgb color code of each ball. The rgb color code is specified as [r, g, b] where 0 < r, g, b < 1.
- The array below specifies three balls of different colors, red, green and blue respectively.

Argument `color:`

	r	g	b
ball 1	0.8	0.1	0.1
ball 2	0.1	0.8	0.1
ball 3	0.1	0.1	0.8

`h = mfMolecule(...)`

- Handle `h` retrieves a handle to the molecule objects created by `mfMolecule(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the molecule objects through handle `h` with procedure `msGSet.`

The properties available are:

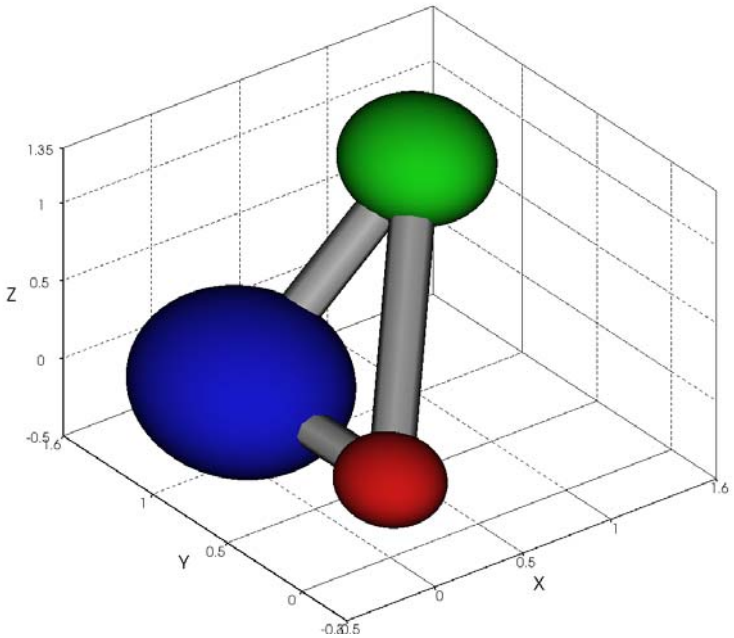
1. location
2. connective
3. radius
4. color
5. stick_radius
6. stick_color
7. resolution

Example

Code

```
program example

use fgl
implicit none

type(mfArray) :: loc, conn, rad, color, stick_rad, stick_col, h

! Specify locations of three balls using vcat
loc = reshape((/0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0/), &
           (/3,3/))

! Specify three sticks and their connections
conn = reshape((/1.0, 1.0, 2.0, 2.0, 3.0, 3.0/), &
            (/3,2/))

! Specify the radius of each ball at radius=0.5,0.7,1.0
rad = (/0.5, 0.7, 1.0/)

! Set the color of each ball to red, green and blue
color = reshape((/0.8, 0.1, 0.1, 0.1, 0.8, 0.1, 0.1, 0.1, 0.8/), &
             (/3,3/))

! Set the cylindrical radius of each stick to 0.1,0.2,0.3
stick_rad = (/0.1, 0.2, 0.3/)

! Set the color of the stick to grey
stick_col = (/0.7, 0.7, 0.7/)

call msAxis(mf((/-0.5d0, 1.6d0, -0.3d0, 1.6d0, -0.5d0, 1.35d0/)))

! Draw the molecules
h = mfMolecule(loc, conn, rad, color, stick_rad, stick_col)

! Pause program to view
call msViewPause()

! Deallocate mfArrays
call msFreeArgs(loc, conn, rad, color, stick_rad, stick_col)

end program example
```

Result



See Also

mfSphere

mfFastMolecule, msFastMolecule

Draw stick and ball model of molecules.

Module

```
use fgl
```

Syntax

```
handle = mfFastMolecule(loc, conn[, rad][, color][, stick_rad][,
stick_col])
```

Descriptions

Procedure `Molecule` enables you to create three-dimensional stick-and-model of molecules.

```
call msMolecule(loc, conn, rad, color, stick_rad, stick_col,
resolution)
```

- Draw n balls specified by argument `loc` and m sticks specified by argument `conn.` Argument `rad` specifies the radius of each ball, argument `color` specifies the color of each ball, argument `stick_rad` specifies the cylindrical radius of each stick, argument `stick_col` specifies the color of the sticks and argument `resolution` specifies the smoothness of the ball objects.
- Argument `loc` is an n-by-3 array containing the three-dimensional Cartesian coordinates (x, y, z) of each ball center, hence specifying the spatial relationship of each ball. Each ball is numbered according to their respective row index. Thus row number 1 specifies ball number 1. The first column contains the x-coordinates, the second column contains the y-coordinates and the third the z-coordinates respectively. As an example, the array below specifies three balls whose center are located at (0,0,0), (1,1,1) and (0,1,0) respectively.

Argument `loc:`

	x	y	z
ball 1	0	0	0
ball 2	1	1	1
ball 3	0	1	0

- Argument `conn` is an m-by-2 array specifying m number of sticks and the balls connected by each stick. Each stick connects two balls and is labeled according to its row number. Columns of the argument `conn` contain the indices of the balls that each

stick connects. For example, the array below specifies 3 sticks connecting ball 1 and ball 2, ball 3 and ball 1, ball 2 and ball 3.

Argument `conn:`

	ball index	ball index
stick 1	1	2
stick 2	3	1
stick 3	2	3

- Argument `rad` is a scalar specifying the radius of all balls or an n-by-1 array specifying the radius of each individual ball respectively. By default, all balls are drawn with a radius of 0.5. As an example, the array below specifies three balls of different sizes, with radius 1, 2 and 3 respectively.

Argument `rad:`

	radius
ball 1	1
ball 2	2
ball 3	3

- Argument `color` contains a string specifying the color of all the balls or an n-by-3 array containing the rgb color code of each ball. The rgb color code is specified as [r, g, b] where 0 < r, g, b < 1.
- The array below specifies three balls of different colors, red, green and blue respectively.

Argument `color:`

	r	g	b
ball 1	0.8	0.1	0.1
ball 2	0.1	0.8	0.1
ball 3	0.1	0.1	0.8

`h = mfMolecule(...)`

- Handle `h` retrieves a handle to the molecule objects created by `mfMolecule(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the molecule objects through handle `h` with procedure `msGSet.`

The properties available are:

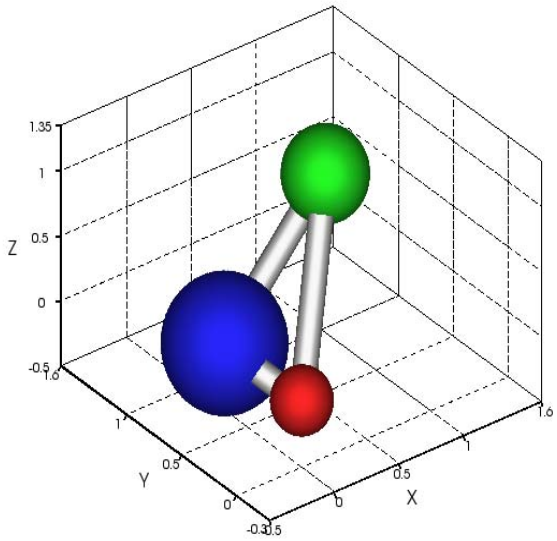
1. location
2. connective
3. radius
4. color
5. stick_radius
6. stick_color

Example

Code

```
program Example_msFastMolecule

    use fgl
    use fml

implicit none

    ! Variable Declaration
    real(8) :: x1, x2, x3, y, z
    type(mfArray) :: loc, conn, rad, color, stickRad, stickColor

    ! Computation
    x1 = 0
    x2 = 1.5
    x3 = 3
    rad = (/1, 1, 2/)
    loc = (/x1, 0d0, 0d0/).vc.(/x3, 0d0, 0d0/).vc.(/x2, 2d0, 2d0/)
    conn = (/2, 3/).vc.(/3, 1/)
    color = (/1, 0, 0/).vc.(/1, 0, 0/).vc.(/0, 0, 1/)
    stickRad = 0.2
    stickColor = (/1, 1, 1/)

    ! Display Graphics
    call msAxis('equal')
    call msFastMolecule(loc, conn, rad, color, stickRad, stickColor)
    call msViewPause()

end program Example_msFastMolecule
```

Result



See Also

mfSphere

mfSphere, msSphere

Draw a sphere.

Module

```
use fgl
```

Syntax

```
handle = mfSphere([loc][, rad][, color][, resolution])
```

Descriptions

Procedure `mfSphere` draws a sphere with center at `loc`, radius specified by `rad`, color specified by `color` and transparency level specified by `resolution`. All arguments are optional.

```
call msSphere(loc, rad, color, resolution)
```

Argument	Meaning
`loc`	A 1-by-3 mfArray containing the x-, y- and z-coordinates of the sphere center. By default, argument `loc` is set to `[0,0,0]`.
`radius`	An mfArray containing a real number specifies the radius of the sphere. By default, argument `radius` is set to `0.5`.
`color`	An mfArray containing a string specifies the color, e.g. “y”, or a 1-by-3 mfArray contains the rgb codes. By default, argument `color` is set to grey.
`resolution`	An mfArray containing the number of polygons used for modeling the circular object. The higher the polygon number, the smoother a sphere appears.

	The lower the polygon number, the faster a sphere is rendered. By default, the sphere is set to a resolution = 64.

`h = mfSphere(...)`

- Handle `h` retrieves a handle to the sphere object created by `mfSphere(...).`
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the sphere object through handle `h` with procedure `msGSet.`

The properties available are:

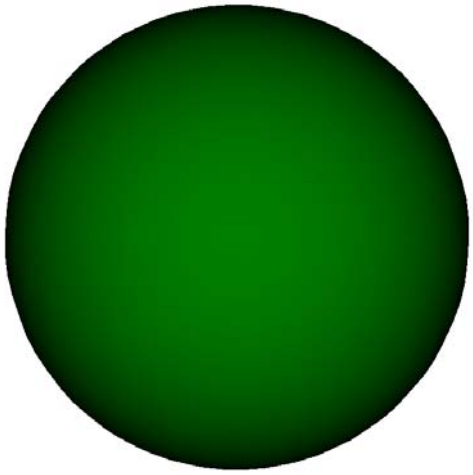
1. location
2. radius
3. color
4. resolution

Example

Code

```
program example

use fgl
implicit none

type(mfArray) :: zeros

zeros =(/0, 0, 0/)

! Center at (0,0,0), Radius = 0.5, Color = 'green'
call msSphere(zeros, mf(1), mf('g'))

! Remove current Axis
call msAxis('off')
call msAxis('equal')

! Pause the program for display
call msViewPause()

! Deallocate mfArray
call msFreeArgs(zeros)

end program example
```

Result



See Also

mfCylinder, mfMolecule, mfCube

mfCube, msCube

Draw a Cube.

Module

```
use fgl
```

Syntax

```
handle = mfCube([loc][, size][, color][, center])
```

Descriptions

Procedure `mfCube` draws a cube with center at `loc`, size specified by `size`, color specified by `color` and transparency level specified by `resolution`. All arguments are optional.

```
call msCube(loc, size, color, center)
```

Argument	Meaning
`pos`	A 1-by-3 mfArray contains the x-, y- and z- coordinates of the cone center. By default, argument `pos` is set to `[0,0,0]`.
`size`	A 1-by-3 mfArray specifies the length , width and height of a cube. By default, argument `Size` is set to `[1,1,1]`.
`color`	An mfArray containing a string specifies the color, e.g. `"y"`, or a 1-by-3 mfArray containing the rgb codes. By default, argument `color` is set to `grey`.

`h = mfCube(...)`

- Handle `h` retrieves a handle to the cube object created by `mfCube(...)`.
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the cube object through handle `h` with procedure `msGSet`.

The properties available are:

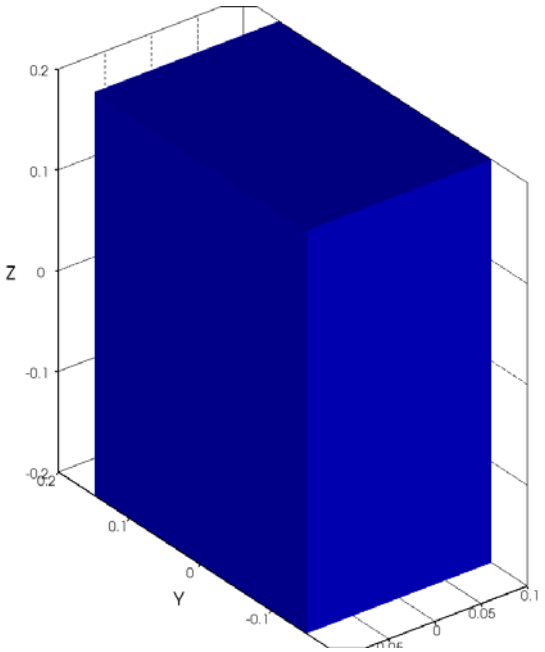
1. location
2. radius
3. color
4. center

Example

Code

```
program example

use fgl
implicit none

type(mfArray) :: zeros, cubesize

zeros = (/0, 0, 0/)
cubesize = (/0.2, 0.3, 0.4/)

! Cube with center at (0,0,0), size of 0.2x0.3x0.4
! and blue in color
call msCube(zeros, cubesize, mf('b'))
call msAxis('equal')

! Pause the program for display
call msViewPause()

! Deallocate mfArrays
call msFreeArgs(zeros, cubesize)

end program example
```

Result



See Also

mfCylinder, mfSphere, mfCone

mfCylinder, msCylinder

Draw a cylinder.

Module

```
use fgl
```

Syntax

```
handle = mfCylinder([loc][, rad][, height][, color][, resolution])
```

Descriptions

Procedure `mfCylinder` draws a cylinder with center at `loc`, radius specified by `rad`, color specified by `color` and transparency level specified by `resolution`. All arguments are optional.

```
call msCylinder(loc, rad, height, color, resolution)
```

Argument	Meaning
`pos`	A 1-by-3 mfArray contains the x-, y- and z- coordinates of the cylinder center. By default, argument `pos` is set to `[0, 0, 0]`.
`radius`	An mfArray containing a real number specifies the radius of the cylinder. By default, argument `radius` is set to `0.5`.
`height`	An mfArray containing a real number specifies the height of the cylinder. By default, argument `height` is set to `1.0`.
`color`	An mfArray containing a string specifies the color, e.g. `"y"`, or a 1-by-3 mfArray contains the rgb codes. By default, argument `color` is set to `grey`.
`resolution`	An mfArray contains the number of polygons used for modeling the circular object. The higher the polygon number, the smoother a sphere appears. The lower the polygon number, the faster a sphere is rendered. By default, the sphere is set to a `resolution = 64`.

```
h = mfCylinder(...)
```

- Handle `h` retrieves a handle to the cylinder object created by `mfCylinder(...)`.

- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the cylinder object through handle `h` with procedure `msGSet.`

The properties available are:

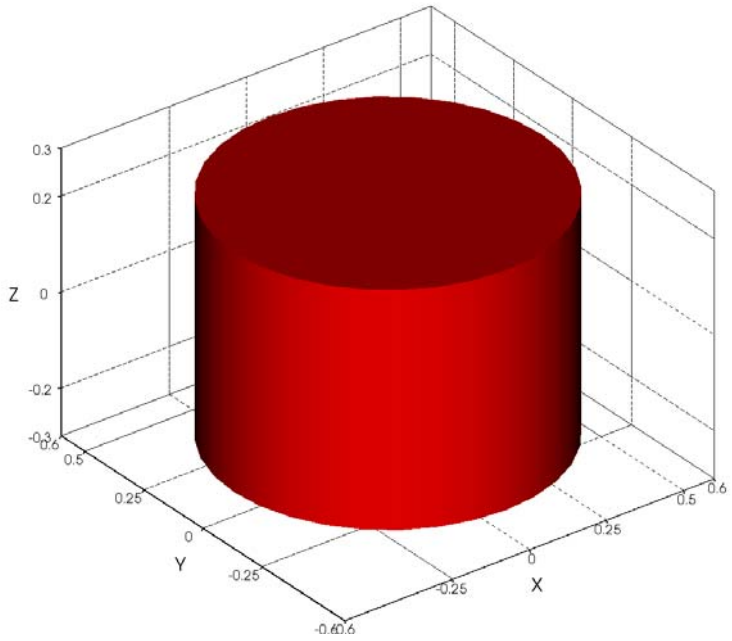
1. location
2. radius
3. height
4. color
5. resolution
6. center

Example

Code

```
program example

use fgl
implicit none

type(mfArray) :: zeros

zeros = (/0, 0, 0/)

! Cube with center at (0,0,0), size of 0.2x0.3x0.4
! and blue in color
call msCylinder(zeros, mf(0.5), mf(0.5), mf('r'))

call msCamZoom(1.2d0)

! Pause the program for display
call msViewPause()

! Deallocate mfArrays zeros
call msFreeArgs(zeros)

end program
```

Result



See Also

mfSphere, mfCone, mfCube

mfCone, msCone

Draw a Cone.

Module

```
use fgl
```

Syntax

```
handle = mfCone([loc][, rad][, height][, color][, resolution])
```

Descriptions

Procedure `mfCone` draws a Cone with center at `loc`, radius specified by `rad`, color specified by `color` and transparency level specified by `resolution`. All arguments are optional.

```
call msCone(loc, rad, height, color, resolution)
```

`h = mfCone(...)`

- Handle `h` retrieves a handle to the cone object created by `mfCone(...)`.
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the cone object through handle `h` with procedure `msGSet`.

The properties available are:

1.location 2.radius 3.height 4.color 5.resolution 6.center

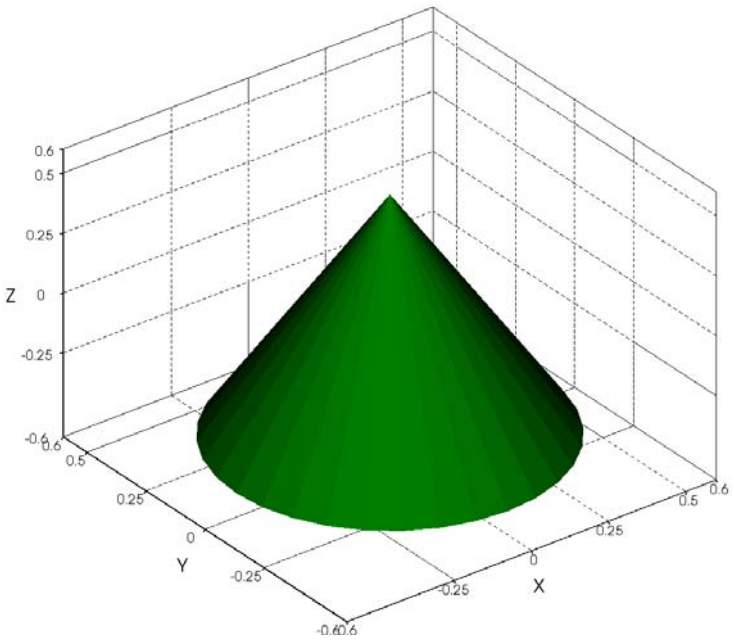
Example

Code

```
program example

use fgl
implicit none

type(mfArray) :: zeros

zeros = (/0, 0, 0/)

! Cone with location (0,0,0), radius = 0.5, height = 1.0,
! color = green.
call msCone(zeros, mf(0.5d0), mf(1.0d0), mf('g'))

call msCamZoom(1.2d0)

! Pause the program for display
call msViewPause()
```

```
! Deallocate mfArrays x and r and p
call msFreeArgs(zeros)

end program
```

Result



See Also

mfCylinder, mfSphere, mfCube

mfAxisMark, msAxisMark

3-directional axis mark on arbitrary point.

Module

```
use fgl
```

Syntax

```
call msAxisMark([loc][, length][, radius])
handle = mfAxisMark([loc][, length][, radius])
```

Descriptions

Procedure `mfAxisMark` draws a 3-directional axis mark on any arbitrary point in the plot space. Its purpose is to .

`call msAxisMark(loc, length, radius)`

- Draws a 3-directional axis mark on the point specified by argument `loc` with length specified by argument `length` and thickness specified by argument `radius`.
- By default, location is [0, 0, 0], length is 1 and radius is 0.1.

`h = mfAxisMark(...)`

- Handle `h` retrieves a handle to the axis mark object created by `mfAxisMark(...)`.
- Alternatively, you use procedure `h = mfGetCurrentDraw()` to retrieve the handle of the current graphics object.

You can specify properties of the molecules object through handle `h` with procedure `msGSet`.

The properties available are:

1. symmetric = `"true"` or `"false"`
If symmetric is on, the axes extend to negative values.
2. location
3. length
4. radius

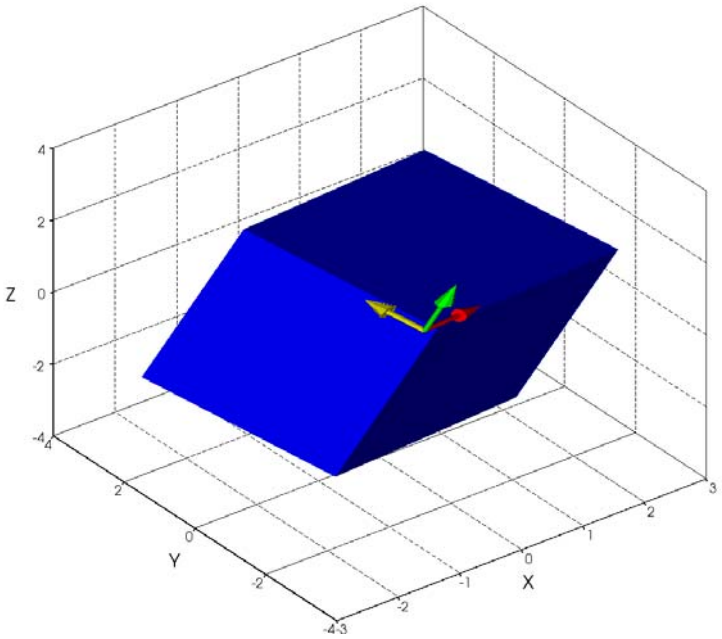
Example

Code

```
program example

use fml
use fgl
implicit none
```

```
type (mfArray) :: cubesize, center, h, loc, length, radius

cubesize = (/4, 4, 4/)
center = (/0, 0, 0/)
loc = mf((/-1.0d0, -3.0d0, 2.1d0/))
length = 1
radius = 0.05

h = mfCube(center, cubesize, 'b')
call msObjOrientation(h, 15, 15, 15)
call msHold('on')
h = mfAxisMark(loc, length, radius)
call msObjOrientation(h, 15, 15, 15)
call msViewPause()

call msFreeArgs(cubesize, center, h)

end program example
```

Result



See Also

Property Setting

msGSet

Set property of specified graph.

Module

```
use fgl
```

Syntax

```
call msGSet(handle, property, value[, property2, value2, ...])
```

Descriptions

Procedure `msGSet` sets the property of a graphics object whose handle is given by `handle`. You can set various properties of a graph object through the procedure .

```
call msGSet(handle, property, value)
```

- Argument `property` is a string specifying the target property to be updated. Argument `value` is an mfArray containing the data to be updated. For example, you can input "xdata" for x-coordinate, "ydata" for y-coordinate, "zdata" for z-coordinate if you want to update the coordinates of a graphics object.

```
call msGSet(handle, property, value, property2, value2, ...)
```

- Multiple properties can be updated in one statement as above.

- The table below lists the common properties available for updating through procedure `msGSet`.

Property	Description	Apply to
Xdata	Specify x data as target for updating	Almost all two-dimensional and three-dimensional graphics objects
Ydata	Specify y data as target for updating	Almost all two-dimensional and three-dimensional graphics objects
Zdata	Specify z data as target for updating	Almost all three-dimensional graphics objects

Cdata	Specify color vector, c as target for updating	Almost all two-dimensional and three-dimensional graphics objects
Udata	Specify velocity in x-direction, u as target for updating.	mfQuiver, mfQuiver3, mfStreamLine, etc.
Vdata	Specify velocity in y-direction, v as target for updating.	mfQuiver, mfQuiver3, mfStreamLine, etc.
Wdata	Specify velocity in z-direction, w as target for updating.	mfQuiver3, mfStreamLine

Note: Not all of the available properties are listed here, please refer to the description of each graphical procedure for a supplementary listing of properties.

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray):: x, y, h
integer::i

!Construct and initialize the mfArrays for plotting.
x = mfLinspace(-MF_PI, MF_Pi, 30)
y = mfCos(x)

!Plot the initial figure and get its handle.
call msPlot(x, y, 'ro')
h=mfGetCurrentDraw()

!Set up an iteration loop for the range of data you
!wish to observe through animation.
Do i=1,1000
   y=mfCos(x+0.02d0*i)

   !Within the iteration loop, use procedure msGSet to update
   !the targeted data of the current draw.
   call msGSet(h, 'ydata', y)

   !Update the current Graphics Viewer by using procedure
   !msDrawNow.
   call msDrawNow()

end do

!Pause the program to observe figure.
call msViewPause()

!Deallocate mfArray
call msFreeArgs(x, y, h)
```

```
end program example
```

See Also

msDrawMaterial

Set transparency reflectance, ambient reflectance, diffuse reflectance and specular reflectance of a draw object.

Module

```
use fgl
```

Syntax

```
call msDrawMaterial(handle, target, property, value)
```

Descriptions

Procedure `msDrawMaterial` sets the color component and the transparency reflectance, ambient reflectance, diffuse reflectance and specular reflectance of the draw object's surface and edge. Each reflectance is specified as a level of intensity ranging from 0 to 100. The resultant lighting effect is produced by applying the intensity levels of the reflectances to the color component.

For example, if the intensity of the draw object's ambient reflectance is set to be 50 and the color component is set to be [1, 1, 1], the draw object's ambient color component becomes [0.5, 0.5, 0.5].

`call msDrawMaterial(handle, target, property, value)`

- You can perform the operation on the draw object's surface, edge or both by specifying argument `target` as "surf", "edge" or "both".
- Arguments `property` and `value` can be:

Property	Meaning
"trans"	Transparency reflectance. The corresponding argument value can be an integer or an mfArray containing an integer that ranges from 0 to 100.
"ambient"	Ambient reflectance. The corresponding argument value can be an integer or an mfArray containing an integer that ranges from 0 to 100.
"diffuse"	Diffuse reflectance. The corresponding argument value can be an integer or an mfArray containing an integer that ranges from 0 to

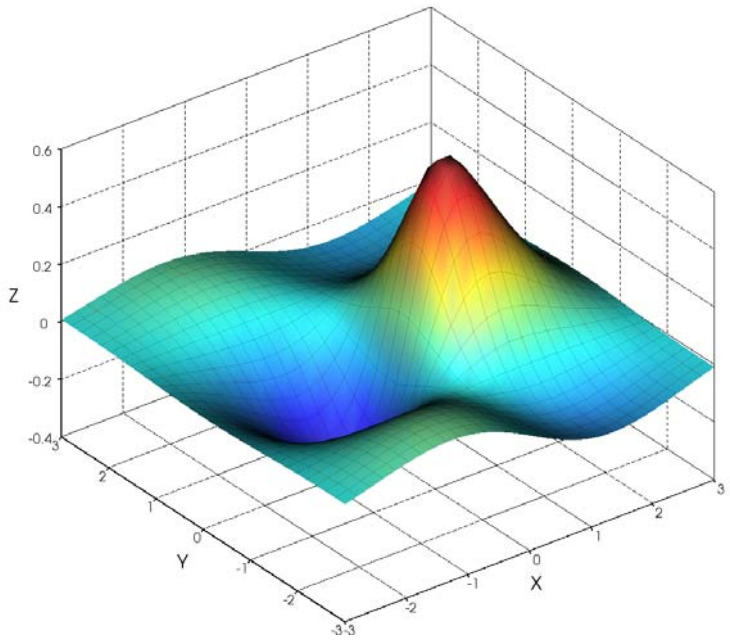
	100.
“specular”	Specular reflectance. The corresponding argument value can be an integer or an mfArray containing an integer that ranges from 0 to 100.
“color”	Color component. The corresponding argument value can be an mfArray containing the rgb vector [r, g, b] or an mfArray containing the string that is specified as “on” or “off”.
“colormap”	Turn the colormap on or off. The corresponding argument value can be an mfArray containing the string that is specified as “on” or “off”.
“visible”	Turn the surface or edge on or off. The corresponding argument value can be an mfArray containing the string that is specified as “on” or “off”.
“smooth”	Interpolate to Gouraud shading. The corresponding argument value can be an mfArray containing the string that is specified as “on” or “off”.
“line_width”	Line width of edge. The corresponding argument value can be an integer or an mfArray containing an integer.
“line_style”	Line style of edge. The corresponding argument value can be an mfArray containing a string that is specified as "solid", "dashed", "dotted" or "dashdot".

Example

Code

```
program example

use fgl
use fml
implicit none

type (mfArray) :: a, x, y, z, h

a = mfLinspace(-3, 3, 30)
call msMeshGrid( mfOut(x, y), a)
z = mfSin(x) * mfCos(y) / ( x*(x-0.5d0) + (y+0.5d0)*y + 1)

h = mfSurf(x, y, z)
call msDrawMaterial(h, mf('surf'), mf('visible'), mf('on'), &
                                   mf('smooth'), mf('on'),       &
                                   mf('colormap'), mf('on'), &
                                   mf('ambient'), mf(0),     &
                                   mf('diffuse'), mf(75),        &
                                   mf('specular'), mf(25))
call msDrawMaterial(h, mf('edge'), mf('color'), mf((/1,0,0/)),  &
                                   mf('smooth'), mf('on'),       &
                                   mf('colormap'), mf('off'),    &
                                   mf('ambient'), mf(0),     &
                                   mf('diffuse'), mf(0),     &
                                   mf('diffuse'), mf(0),     &
                                   mf('specular'), mf(0),        &
                                   mf('trans'), mf(90))
call msViewPause()

call msFreeArgs(a, x, y, z, h)

end program example
```

Result



See Also

msDrawTexture

Texture mapping.

Module

```
use fgl
```

Syntax

```
call msDrawTexture(handle, property1, value1[, property2, value2, ...])
```

Descriptions

Procedure `msDrawTexture` places a texture on a graphics object by mapping the texture coordinates to the object's coordinates. The texture coordinates comprise two coordinates namely s- and t-coordinates which are vectors of values ranging from 0 to 1. They correspond to the object's x- and y-coordinates in order to determine which texel in the texture is mapped to which vertex.

```
call msDrawTexture(handle, property, value)
```

- Arguments `property` and `value` can be:

Property	Meaning
"enable"	Enabling or disabling the texture-mapping. The corresponding argument value can be “on” or “off”.
"map"	Specifying the texture file. The corresponding argument value specifies the name of a bitmap file(e.g. texture.bmp).
"coord_s"	Texture’s s-coordinate. The corresponding argument value is a vector of values ranging from 0 to 1 which specifies the way of mapping.
"coord_t"	Texture’s t-coordinate. The corresponding argument value is a vector of values ranging from 0 to 1 which specifies the way of mapping.

-

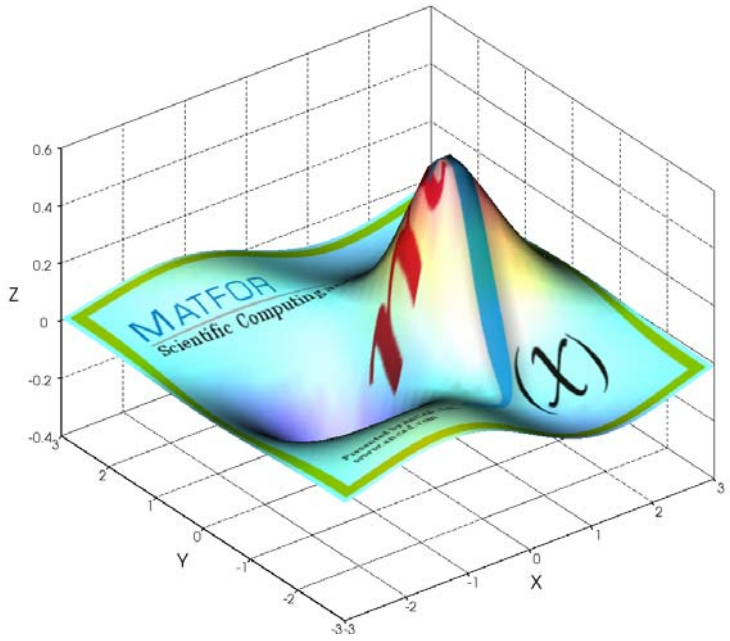
Example

Code

```
program example
```

```
use fgl
use fml
implicit none

type (mfArray) :: a, x, y, z, h

a = mfLinspace(-3, 3, 30)
call msMeshGrid( mfOut(x, y), a)
z = mfSin(x) * mfCos(y) / ( x*(x-0.5d0) + (y+0.5d0)*y + 1)

h = mfSurf(x, y, z)
call msDrawTexture(h, 'map', 'ancad.bmp')
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on'),       &
                                   mf('colormap'), mf('on'), &
                                   mf('ambient'), mf(0),     &
                                   mf('diffuse'), mf(15),        &
                                   mf('specular'), mf(85))
call msDrawMaterial(h, 'edge', 'visible', 'off')
call msViewPause()

call msFreeArgs(a, x, y, z, h)

end program example
```

Result



See Also

mfIsValidDraw

Check validity of draw object.

Module

```
use fgl
```

Syntax

```
validity = mfIsValidDraw(handle)
```

Descriptions

Procedure `mfIsValidDraw` returns the validity of a draw object which is associated with argument `handle.` The output is an mfArray containing logical data. It returns true if the draw object still exists, false otherwise.

See Also

mfGetCurrentDraw

Return handle of current draw object.

Module

```
use fgl
```

Syntax

```
handle = mfGetCurrentDraw()
```

Descriptions

Procedure `mfGetCurrentDraw` returns the handle of current draw object.

Example

Code

```
program example

use fml
use fgl
implicit none

type (mfArray) :: a, b, c, x, y, z, theta, phi, h
integer :: i

a = mfLinspace(-MF_PI/2, MF_PI/2, 31)
b = mfLinspace(-MF_PI, MF_PI, 31)
c = mfOnes(31, 31)
call msMeshgrid(mfout(phi, theta), a, b)
x = mfCos(phi)*mfCos(theta)
y = mfCos(phi)*mfSin(theta)
z = mfSin(phi)

! Plot the graph you wish to animate
call msSurf(x, y, z)
call msAxis(mf((/-MF_PI, MF_PI, -MF_PI, MF_PI, -MF_PI, MF_PI/)))
call msShading('interp')
call msAxis('off')

! Get handle of current draw
h = mfGetCurrentDraw()

! Use a Do Loop to change the value of x, y and z.
do i =1 ,30
   x = mfCos(phi)*mfCos(theta+i*0.1d0)
   y = mfCos(phi)*mfSin(theta+i*0.1d0)+0.05d0*i
   z = mfSin(phi)+mfSin(c*i)
   ! Set the x,y, and z-data of the current graph
   call msGSet(h, mf('xdata'), x, mf('ydata'), y, mf('zdata'), z)
   ! Draw graph on Graphics Viewer
   call msDrawNow()
end do

! Pause program to view graph. If this statement is
! not added, the Graphics Viewer closes once animation
! is completed.
Call msViewPause()
```

```
! Deallocate mfArrays
Call msFreeArgs(a, b, c, x, y, z, theta, phi, h)

end program example
```

See Also

msRemoveDraw

Remove draw object from plot space.

Module

```
use fgl
```

Syntax

```
call msRemoveDraw(handle1[, handle2, ...])
```

Descriptions

Procedure `msRemoveDraw` removes specific draw objects from plot space.

`call msRemoveDraw(handle1, handle2, ...)`
Removes the draw objects that are associated with handles specified in the arguments.

See Also

msSetDrawName

Name draw object.

Module

```
use fgl
```

Syntax

```
call msSetDrawName(handle, name)
```

Descriptions

Procedure `msSetDrawName` sets the name of a draw object that is associated with argument `handle`. By default, the name of a draw object is set to its draw type followed by an incremental integer.

The purpose of giving each draw object a name is for distinguishing between the draw objects. It allows you to perform operations (e.g. custom shading or view draw object data) on a specific draw object when there are two or more draw objects present in the same subplot.

See Also

Simple GUI

msShowMessage

Pop up message dialog box.

Module

```
use fgl
```

Syntax

```
call msShowMessage(msg)
```

Descriptions

Procedure `msShowMessage` pops up a dialog box displaying a message.

Example

Code

```
program example

use fml
use fgl
implicit none

    call msShowMessage("Show Message Test")

end program
```

Result

See Also

mfInputString

Pop up string insertion dialog box.

Module

```
use fgl
```

Syntax

```
string = mfInputString(msg, default_string)
```

Descriptions

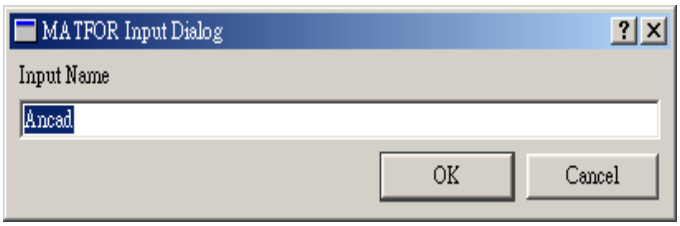
Procedure `mfInputString` pops up a dialog box displaying a message. The input string is passed back to the output argument `string.`

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: str;
    str = mfInputString("Input Name", "Ancad");
    call msShowMessage(str);

end program
```

Result



See Also

mfInputValue

Pop up value insertion dialog box.

Module

```
use fgl
```

Syntax

```
value = mfInputValue(msg, default_value)
```

Descriptions

Procedure `mfInputValue` pops up a dialog box displaying a message. The input value is passed back to the output argument `value.`

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: val;
    val = mfInputValue("Input Number", 10);

    call msDisplay(val, "Number");

end program
```

See Also

mfInputVector

Pop up vector insertion dialog box.

Module

```
use fgl
```

Syntax

```
vector = mfInputVector(msg, default_vector)
```

Descriptions

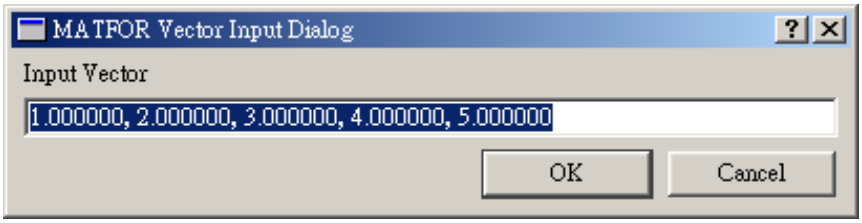
Procedure `mfInputVector` pops up a dialog box for inserting the entries of a vector. The input entries are passed back to the output argument `vector.`

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: vec;
    vec = mfInputVector("Input Vector", mf((/1, 2, 3, 4, 5/)));
    call msDisplay(vec, "vec");

end program
```

Result



See Also

mfInputMatrix

Pop up matrix insertion dialog box.

Module

```
use fgl
```

Syntax

```
matrix = mfInputMatrix(msg, default_matrix)
```

Descriptions

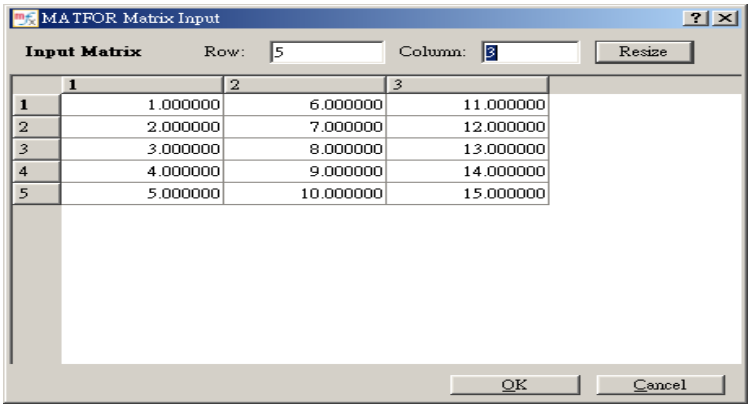
Procedure `mfInputVector` pops up a dialog box for inserting the entries of a matrix. The input entries are passed back to the output argument `matrix.`

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: x, m
real(8) :: a(5, 3)
integer :: i

x=reshape((/(i, i=1,15)/),(/5,3/))
m=mfInputMatrix("Input Matrix", x)
call msDisplay(m, 'm')

end program
```

Result



See Also

mfFileDialog

Pop up file open dialog box.

Module

```
use fgl
```

Syntax

```
string = mfFileDialog(filename, filefilter)
```

Descriptions

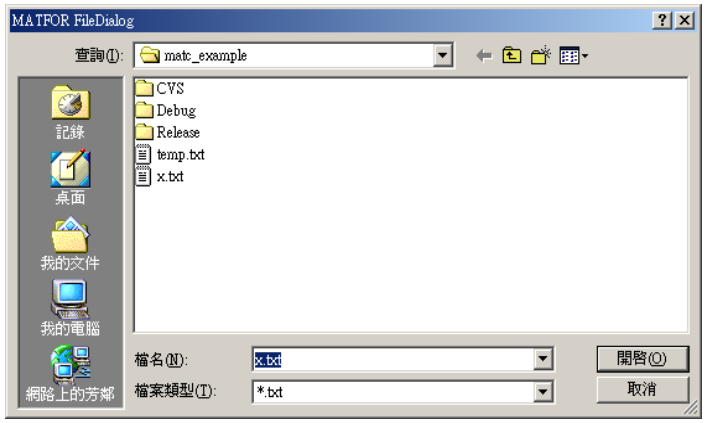
Procedure `mfFileDialog` pops up a file open dialog box for locating a file.

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: a, str;

    str = mfFileDialog("x.txt", "*.txt")
    a = mfLoadAscii(str)
    call msDisplay(a, "a")

end program
```

Result



See Also

mfInputYesNo

Pop up yes-no query dialog box.

Module

```
use fgl
```

Syntax

```
value = mfInputYesNo(msg, default_value)
```

Descriptions

Procedure `mfInputYesNo` pops up a yes-no dialog box. The chosen option is passed back to the output argument `value.`

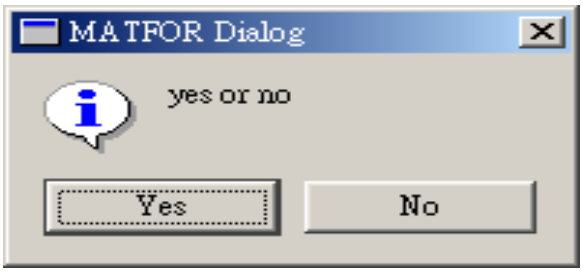
Argument `default_value` can be either 0 or 1.

Example

Code

```
program example

use fml
use fgl
implicit none

type(mfArray) :: val;

    val = mfInputYesNo("yes or no")
    call msDisplay(val, "val")

end program
```

Result



See Also

Index