Contents

Introduction 7

1.1 Overview 7

1.2 What is MATFOR 8

1.3 The MATFOR Components 9

1.4 MATFOR Procedure Naming Conventions 13

1.5 Array Terminology 15

1.6 MATFOR Installation 16

1.7 MATFOR Documentation and Examples 18

1.8 Technical Support 18

Working with mfArray 21

2.1 What is mfArray 21

2.1.1 Structure of the mfArray 21

2.1.2 mfArray Intrinsic Data Type 22

2.1.3 mfArray Element Ordering 23

2.1.4 Memory Management 24

2.1.5 mfArray Syntax and Expressions 24

2.1.6 Mix mfArray and Fortran Arrays 25

2.2 Create and Initialize mfArray 25

2.2.1 Declaring an mfArray 26

2.2.2 Initializing an mfArray 26

2.2.3 mfArray Creating Procedures 28

2.3 Access Elements and Sections of an mfArray 31

2.3.1 Element Subscripts 31

2.3.2 mfArray Creating Procedures 32
2.3.3 Using Subscript in an mfArray 34
2.4 mfArray I/O 37
2.4.1 Displaying mfArray Data 37
2.4.2 mfArray File I/O 42
2.5 mfArray Inquiry Procedures 47
2.5.1 Logical Inquiry 47
2.5.2 Size, Shape, and Extent 50
2.5.3 Logical Operations 53
2.6 mfArray Operators 56
2.6.1 Arithmetic Operators 57
2.6.2 Relational Operators 58
2.6.3 Matrix Operators and Functions 59
2.6.4 Operators Precedence 63
2.6.5 MATFOR Parameters 64
2.7 Program with mfArray 65
2.7.1 Quick Conversion – Function mf() 66
2.7.2.Using mfArray in If Constructs 66
2.7.3 Using mfArray as Input to Fortran Procedures 68
2.7.4 Using Fortran Arrays as Input to MATFOR Procedures 71
2.7.5 Using mfArray as Input Dummy Arguments 74
2.7.6 Using mfArray as Output Dummy Arguments in Functions ... 75

Linear Algebra 77

3.1 Matrix Inverse 77
3.2 Application of Eigenvalues and Eigenvectors 82
3.3 Least Square Operations 84

Visualization Basics 91

4.1 Plotting Your Data 91

4.2 MATFOR Graphics Viewer 94

4.2.1 Window Frame and Figure Windows 94

4.2.2 Subplots 95

4.2.3 Menu and Toolbar 97

4.3 Creating 3-D Models 100

4.3.1 Generating the Data 100

4.3.2 Loading data (mfb, ascii) 102

4.4 Displaying 3-D objects 104

4.4.1 Adjusting the Viewpoint. 105

4.4.2 Shifting the Objects 106

4.4.3 Rescaling the Objects 107

4.4.4 Changing the Displaying Mode 108

4.4.5 Setting the Axis Object 109

4.5 Colormap, Shading and Texture 111

4.5.1 Adjusting Colormap 111

4.5.2 Displaying Colorbar 115

4.5.3 Shading Objects 116

4.5.4 Mapping Texture 117

4.6 Annotating Your Graph 121

4.6.1 Setting the Title and Axis Labels 122

4.6.2 Text Annotation 123

4.7 Animation and Recording 125

4.7.1 Animation 125

4.7.2 Recording your animation 127

4.7.3 Image Exporting 128

4.8 MATFOR Data Viewer 129
4.8.1 Matrix Table 130
4.8.2 Menu 131
4.8.3 Toolbar 132
4.8.4 Sampling Type 132
4.8.5 Snapshot Panel 133
4.8.6 Analysis Panel 134
4.8.7 Filter Panel 135
4.8.8 Status Bar 135

Visualization Methods 137

5.1 Linear Graph 137
5.1.1 Two-dimensional Linear Graph 137
5.1.2 Three-dimensional Linear Graph 140
5.2 Surface Plot 142
5.2.1 Surface plot 142
5.2.2 Contour plot 144
5.2.3 Pseudocolor plot 146
5.3 Volume Rendering 148
5.3.1 Surface (surf, mesh, outline, contour) 148
5.3.2 Sliced-planes 151
5.3.3 Isosurface 152
5.4 Vector Field 154
5.4.1 Quiver and Streamline 154
5.5 Elementary 3-D Objects 156
5.5.1Primitives 156
5.5.2 Molecule 157
5.6 Unstructured Mesh 159

5.6.1 Surface 160
5.6.2 Contour 163
5.7 Unstructured Grids 164
5.7.1 Surface, Contour, and Iso-surface plots of unstructred grids 165
5.8 Delaunay Triangulation 175
5.8.1 Two-dimensional Delaunay 175
5.8.2 Three-dimensional Delaunay 178
Index 181

1 Introduction

1.1 Overview

This guide is written as an introduction to users who are new to MATFOR in Fortran — a Fortran 90/95 numerical and visualization library. In this guide, the MATFOR foundation array — mfArray is introduced and discussed with some depth in Chapter 2. This is followed by the application of MATFOR in linear algebra in Chapter 3 and then an introduction of visualization basics in Chapter 4. Chapter 5 contains the descriptions on categories of the graphical procedures. We assume that the user has some prior knowledge of programming.

The guide contains the following chapters:

Chapter 1. Introduction, provides an overview of MATFOR including Conventions, Documentation, and Licensing.

Chapter 2. Working with mfArrays, provides an overview of the MATFOR mfArray, including its structure, constructors, operators, and general array syntax.

Chapter 3. Linear Algebra, highlights the linear algebra procedures available in MATFOR and their usage.

Chapter 4. Visualization Basics, contains some basic knowledge regarding MATFOR's visualization toolkits and functional capabilities, including MATFOR Graphics Viewer, MATFOR Data Viewer, and steps to visualization, animation, and presentation.

Chapter 5. Visualization Methods, covers most of the MATFOR Graphical procedures that are categorized into different groups including, Linear Graph, Surface Plot, Volume Rendering, Vector Field, Elementary 3-D Objects, Unstructured Grids, and Delaunay Triangulation.

1.2 What is MATFOR

MATFOR is a set of Fortran 90/95 libraries that enhance your Fortran program with dynamic visualization capabilities, shortens your numerical codes, and speeds up your development process.

By adding a few lines of MATFOR code to your Fortran program, you can easily visualize your computation results, perform run-time animations, or even produce a movie presentation file as you execute your program.

You also have the choice of recording an animation as a MATFOR *.mfa* file for later viewing with the MATFOR mfPlayer.

Debugging is facilitated in the debugging facilities provided by MATFOR Graphics Viewer. You can pause an animation, view the current data using MATFOR Data Viewer, and examine any aberrations.

MATFOR numerical procedures are designed to be intuitive and easy to use. Using the numerical procedures, you can solve many technical computing systems, especially those involving linear algebra systems, in a fraction of the time it would take to write a program in Fortran traditionally.

1.3 The MATFOR Components

The standard edition of MATFOR consists of six main components, namely the MATFOR mfArray, MATFOR Numerical Library *(*named as *fml),* MATFOR Graphics Library *(*named as *fgl),* MATFOR Graphics Viewer, MATFOR Data Viewer, and MATFOR mfPlayer.

MATFOR procedures are divided into two main modules – the *fml* and *fgl* modules. The *fml* module contains the numerical procedures whereas the *fgl* module contains the graphical procedures. MATFOR mfArray is included within both modules. The Graphics Viewer and Data Viewer are included in the *fgl* module.

Numerical procedures included in *fml* are further categorized into several modules such as, *mod_ess, mod_elmat, mod_matfun, mod_ops, mod_datafun, mod_fileio,* and *mod_elfun.* Advanced users who are overhead conscious may choose to include only the modules necessary for the selected procedures as opposed to including the whole *fml* module.

MATFOR mfArray

The heart of MATFOR is a special array - the mfArray. The mfArray is a highly flexible array which does not require explicit data typing nor dimensioning. All you need is a simple declaration:

```
type (mfArray) :: x
```

The data type and dimensions of the mfArray are determined internally by MATFOR when you enter data into it. It is so flexible that you can assign your Fortran array to an mfArray. Then visualize your array data using MATFOR graphical procedures, without having to having to be concerned with the dimensions or data types of your original Fortran array.

MATFOR mfArray enables you to write programs with the ease of an interactive language in the Fortran programming environment. With mfArray, powerful routines, such as those in LAPACK, are packaged into simple and intuitive procedures. For example, the original procedure *DGESVX* for calculating singular value decompositions in LAPACK requires twenty-two individual arguments of different data types and array

dimensions. In MATFOR, the same calculation is achieved by passing three arguments of the same data type to mfArray. As a result, you can focus more of your time on problem-solving, rather than handling inputs and outputs.

MATFOR Numerical Library

The MATFOR Numerical Library is a collection of mfArray inquiries and mathematical procedures, ranging from inquiry procedures such as *mfShape*, *mfSize*, *mfIsLogical*, and elementary mathematical procedures such as *mfSin*, *mfCos*, and complex arithmetic, to sophisticated procedures like eigenvalues, LU decomposition, matrix inverse, and conditioning procedures. Most procedures use mfArray as the input and output argument. The Numerical Library procedures are separated into several smaller modules — *mod_ess*, *mod_ops*, *mod_fileio*, *mod_datafun*, *mod_elfun*, *mod_elmat*, and *mod_matfun*. You may choose to use the individual modules necessary for selected procedures to reduce the size of the library included.

MATFOR Graphics library

MATFOR Graphics library is a set of high-level visualization procedures for two-dimensional and three-dimensional data visualization, animation, and graphical debugging. They are easy-to-use and have a wide-range of applications. All procedures use mfArray as the input and output argument.

MATFOR Graphics Viewer

The Graphics Viewer (shown in Figure 1.3.1) is a window for displaying your graphs on the screen. The Graphics Viewer provides menu and toolbar functions for editing and debugging.

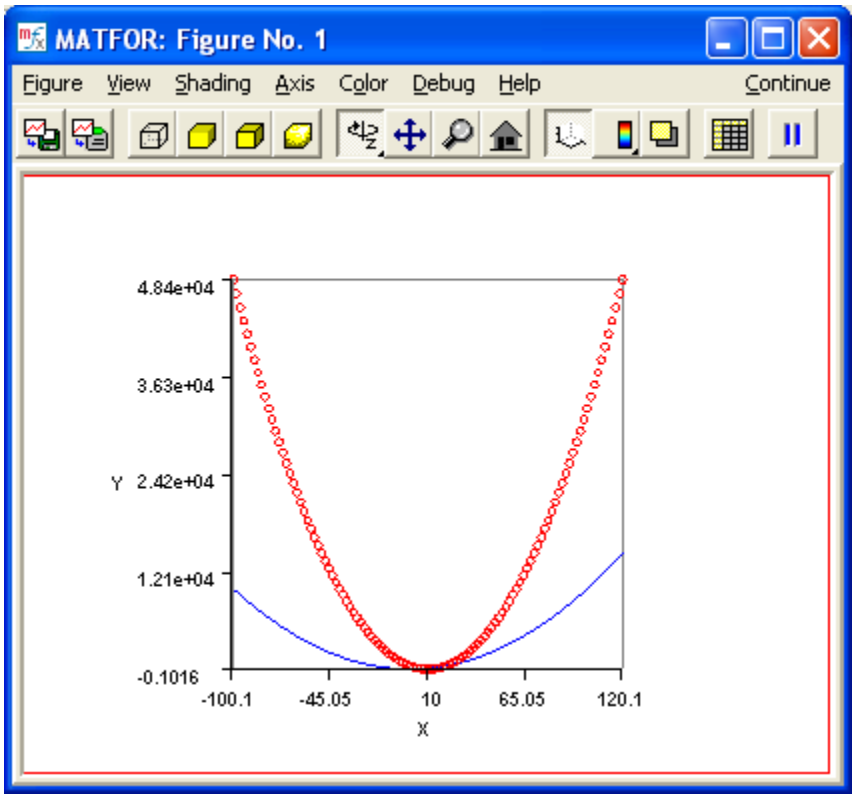


Figure 1.3.1 The Graphics Viewer-MATFOR Data Viewer

MATFOR Data Viewer

MATFOR Data Viewer (shown in Figure 1.3.2) is a spreadsheet-like window used for displaying your mfArray data. It displays both complex and real data. Menu and toolbar functions are available for manipulating the array data. You can perform statistical analysis on your mfArray data and filter the analysis with conditions specified using mathematical expressions. MATFOR Data Viewer also allows you to export your mfArray data to Microsoft Excel using the export function on the toolbar.

MATFOR mfPlayer

MATFOR mfPlayer (shown in Figure 1.3.3) is an offline player similar to Graphics Viewer. You can record your data as *mfa* files using the procedures *msRecordStart* and *msRecordEnd*, then play the animation using mfPlayer at a later time, without slowing down your program computation. The mfPlayer shares the same GUI functions as the Graphics Viewer. mfPlayer is located under the *<MATFOR>\common\tools\player* directory, where *<MATFOR>* is your MATFOR installation directory.

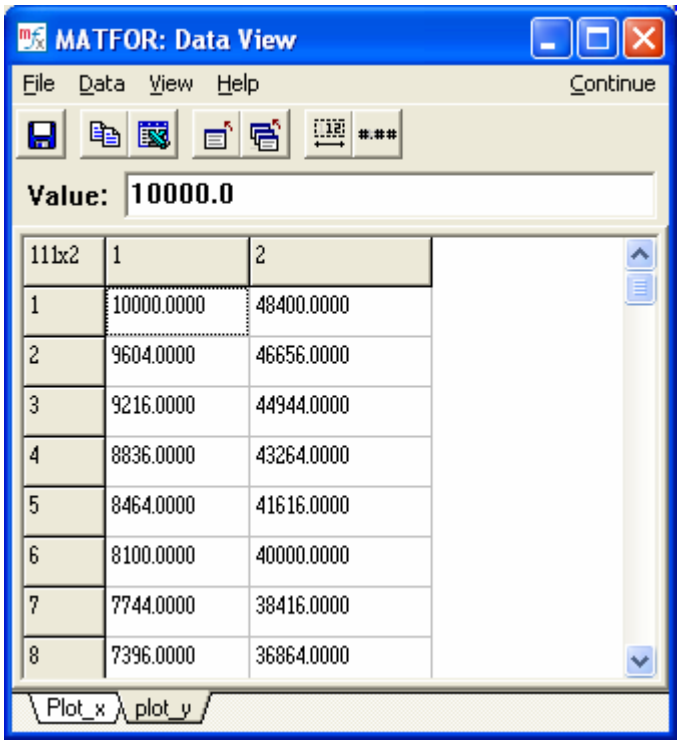


Figure 1.3.2 The Data Viewer

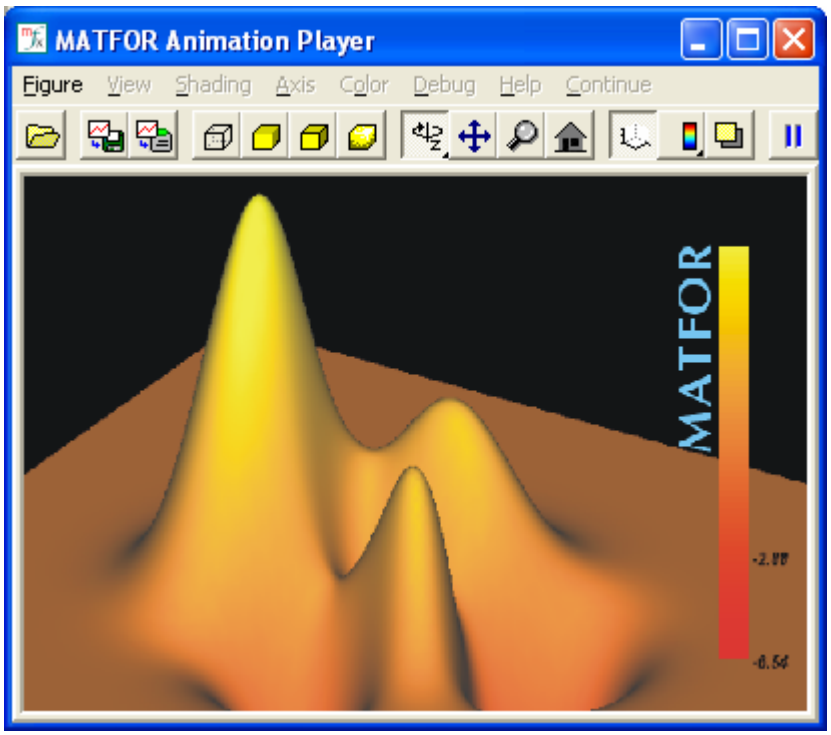


Figure 1.3.3 The mfPlayer

1.4 MATFOR Procedure Naming Conventions

MATFOR procedures are provided in function formats and subroutine formats. Two prefixes are used to classify the MATFOR procedures. The procedures are classified into — procedures with *“mf”* as their prefix and procedures with *“ms”* as their prefix. All MATFOR procedures using *“mf”* as the prefix are function formats whereas procedures using *“ms”* as the prefix are subroutine formats.

By default, MATFOR procedures use the mfArray as input and output arguments. In special cases, it may be more convenient for procedures without a prefix to accept Fortran data types as input and output arguments. Refer to the MATFOR Reference Guide for documentation regarding the individual procedures and for more details on the input and output argument types.

Procedures with “mf” as prefix

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

out = mfFunction ([mfArray]in,…)

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

Procedures with “ms” as prefix

Procedures with *"ms"* as prefix are subroutines. There are three types of subroutine formats — subroutine that do not return a value, subroutine that return a single output (in which case, the prefix changes to *"mf "*), and subroutine that accept multiple input and output arguments. Function *mfOut()* is used to specify the output arguments.

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)

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

call msFunction(mfOut([mfArray]out), [mfArray]in, …).

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

1.5 Array Terminology

Throughout this guide, we will be using the following array terminologies to describe an array.

Properties	Descriptions
rank	number of dimensions
bounds	upper and lower limits of indices
size	total number of elements
shape	rank and extents
conformance	two arrays are of the same shape

Example

```
real :: a(3,3), b(5)
mfArray :: ma, mb
ma = a
mb = b
```

are described by the following:

	ma	mb
rank	2	2
bounds	[1,3] and [1,3]	[1,1] and [1,5]
size	9	5
shape	[3,3]	[1,5]
conformance	a and b do not conform	

1.6 MATFOR Installation

MATFOR comes with an installation package that automatically installs all MATFOR components and tools in your computer. The installation package automatically adds the installed MATFOR directory to the system path and performs some of the Fortran project configuration. In some cases, manual configuration is required. More details on using the installation package are available in the installation instruction that comes with the installation package.

In this section, we shall discuss several issues that might be encountered during your installation of MATFOR. The topics covered include the MATFOR directory structure, project settings, and upgrading a computer with MATFOR previously installed.

MATFOR Directory Structure

MATFOR is installed with the directory structure as illustrated in Figure 1.6.1.

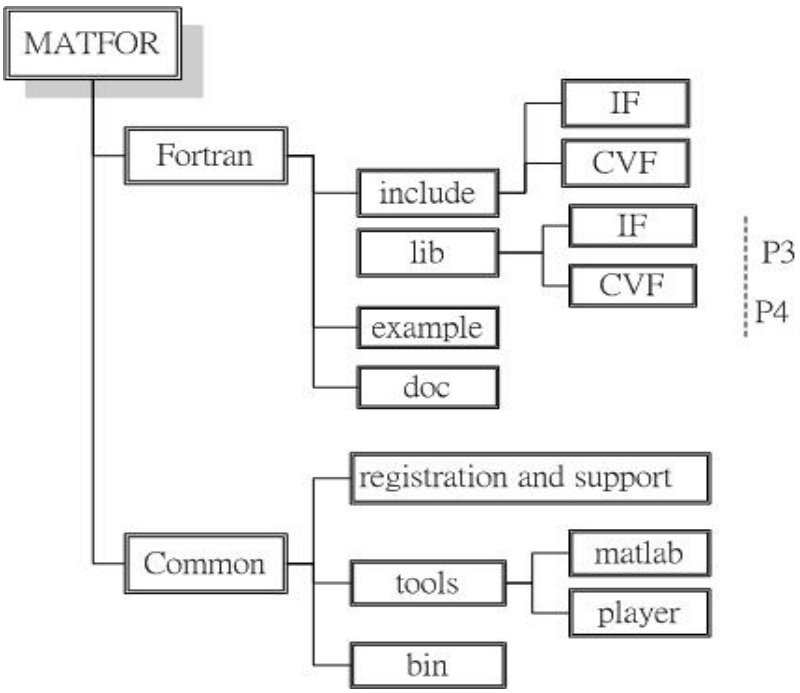


Figure 1.6.1 MATFOR installed directory structure

MATFOR is installed under the *AnCAD* folder, where *AnCAD* is usually located under *C:\Program Files\AnCAD* in the Windows environment. Under the *AnCAD* directory tree, the installed components are further organized into two main categories — the common utilities and the library specifics.

The common utilities include collateral programs, tools, and redistributables such as MATFOR CA, mfPlayer, and dynamically linked libraries, etc. The library (e.g. MATFOR Fortran Library) specifics are mainly components such as Fortran modules *.mod* files, import library *.lib* files, the associated examples, and documentations. Note that for library specifics, MATFOR Fortran Library is installed under *<MATFOR>\Fortran* while MATFOR C++ Library is installed under *<MATFOR>\C++*.

Project Settings

The installation package performs the following settings:

1) **Automatically** adds the path of the folder containing the MATFOR dynamically linked library files to the system path. The added path is *<MATFOR>\bin*.

2) **Automatically** adds the MATFOR Library directories *<MATFOR>\lib* and *<MATFOR>\lib\cvf* to the Visual Studio import library path.

3) **Automatically** adds the MATFOR Include directory *<MATFOR>\include\cvf* .

You are ready to use MATFOR in your program once the above settings are completed successfully.

Bin

Figure 1.6.2 Specify import library file for linker.

Player

Upgrade MATFOR

You should remove all previous versions of MATFOR components before upgrading to a newer version. MATFOR installation package automatically detects previous versions of MATFOR and uninstalls it when you run the installation package. If the installation package fails to uninstall, uninstall through the Windows **Add or Remove Program** utility located in the **Control Panel**.

Once uninstallation is complete, run the MATFOR installation package again to install the new version.

1.7 MATFOR Documentation and Examples

MATFOR has two main documentations, namely *MATFOR in Fortran User's Guide*, and *MATFOR in Fortran Reference Guide*.

If you are new to MATFOR, start with *MATFOR in Fortran User's Guide*. When you need extensive write-up on a procedure, refer to the Reference Guide.

The documents are available in Acrobat Reader pdf format.

Throughout the *MATFOR in Fortran User's Guide*, examples are used to illustrate certain concepts or usages of MATFOR mfArray and procedures. Examples that are labeled with numbers such as Example 2.2.2 are provided as **.f90* Fortran source files and are located in your MATFOR directory. The general path is:

<MATFOR>\Fortran\examples\for_ug

The **.f90* Fortran source files are named followed with their labels in the User's Guide. For example, Example 2.2.2 would be named as Example2_2_2. In most of the examples, the results of the codes are not displayed in this guide, instead, you are encouraged to compile and execute each program as you go through the User's Guide to get a first-hand experience of MATFOR.

1.8 Technical Support

MATFOR for Windows comes with a Support Utility tool, **mfid.exe**, which can be invoked from the *Start menu\Programs\MATFOR\MATFOR support utility* or from the

command window that has the *<MATFOR>\common\tools\registration and support* directory in the path. When submitting a support issue, this tool should be run, and the output is copied and pasted into an email. The information will identify the version of the installed MATFOR, its system environment, Product BIN and corresponding Authorization Key to the AnCAD customer support.

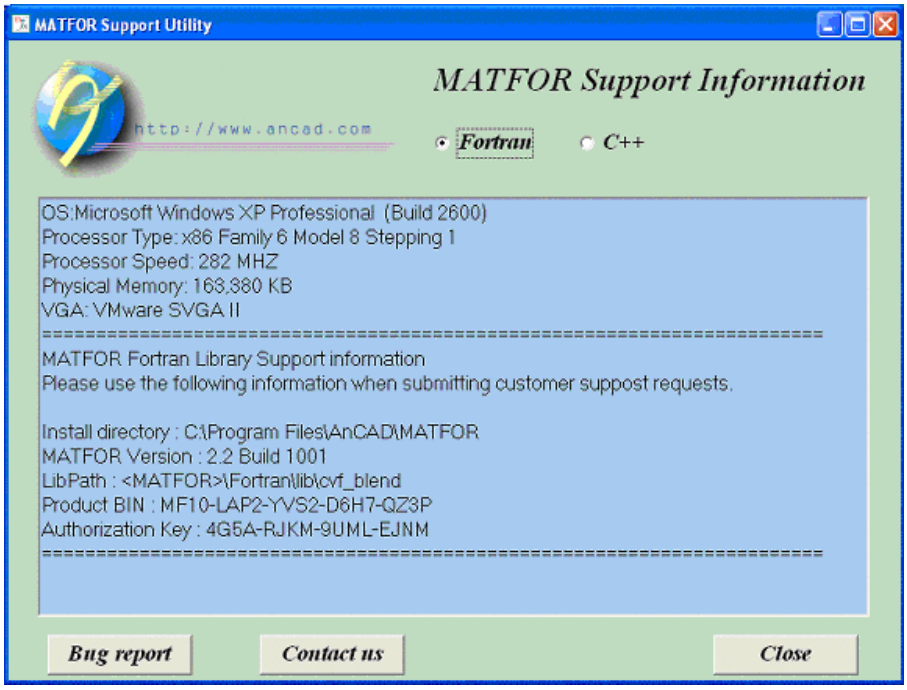


Figure 1.9.1 Information contained in mfid.exe

2

Working with mfArray

2.1 What is mfArray

MATFOR mfArray is an advanced dynamic array defined by MATFOR using modern features of Fortran 90/95 such as modules, function and operator overloading, derived data type and dynamic storage, and pointers.

The mfArray supports automatic data typing and dimensioning. To initialize an mfArray, you only have to write,

```
type (mfArray) :: a
a = 2.0
```

The size, shape, and data type of the mfArray is automatically determined. In the example above, mfArray *a* automatically assumes the shape of the array is 1-by-1, storing *2.0* as a double precision real value.

The mfArray is the foundation of MATFOR. Most MATFOR procedures use mfArray as the input and output arguments. Thus you would need to convert your data to mfArrays in order to use MATFOR procedures. Master it, and you will find your Fortran experience transformed. With mfArray you will not need to handle floating-point data. All you need are mfArrays, integers, and characters in your programs.

2.1.1 Structure of the mfArray

The mfArray contains a set of descriptors and values outlined as below:

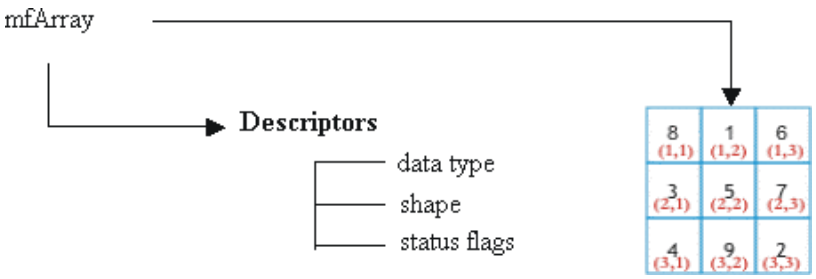


Figure 2.1.1 Structure of the mfArray data object

The descriptors store information such as the current data type of the mfArray, current shape and extent of the mfArray, and status flags such as temporary or restricted. The status flags are used internally by MATFOR for memory management and for restricting methods of the mfArray.

All data are stored as arrays. Scalars are stored in mfArray as 1-by-1 arrays; vectors are stored as 1-by-n arrays where n is the length of the vector; and matrices are stored as m-by-n arrays where *m* and *n* are the lengths of dimensions of the matrices. MATFOR provides supports for up to seven dimensions of data.

2.1.2 mfArray Intrinsic Data Type

The mfArray works with three broad classes of data types — character, Boolean and, numeric. Numeric data are stored in double precision format. Each element of an mfArray assumes the same data type as specified in the data type descriptor.

The data type assumed by an mfArray is automatically determined by MATFOR when you initialize an mfArray. For example,

```
type(mfArray) :: a

! Create a double precision real 1-by-1 mfArray containing 1.0.
a = 1.0d0

! Create a double precision complex 1-by-1 mfArray containing
! 2.0+2.0i.
```

```
a = (2,2)

! Create a double precision real 1-by-1 mfArray containing real
! number 3.0.
a = 3

! Create a 1-by-6 character mfArray containing the characters
! 'string'.
a = 'string'
```

MATFOR displays data in short format when you display an mfArray using the procedure *msDisplay*. Double precision numbers are displayed with four decimal places. Numbers able to be displayed as integers are displayed as such.

2.1.3 mfArray Element Ordering

In actual memory storage, elements of an mfArray are arranged column major-wise as specified by Fortran 90 standard.

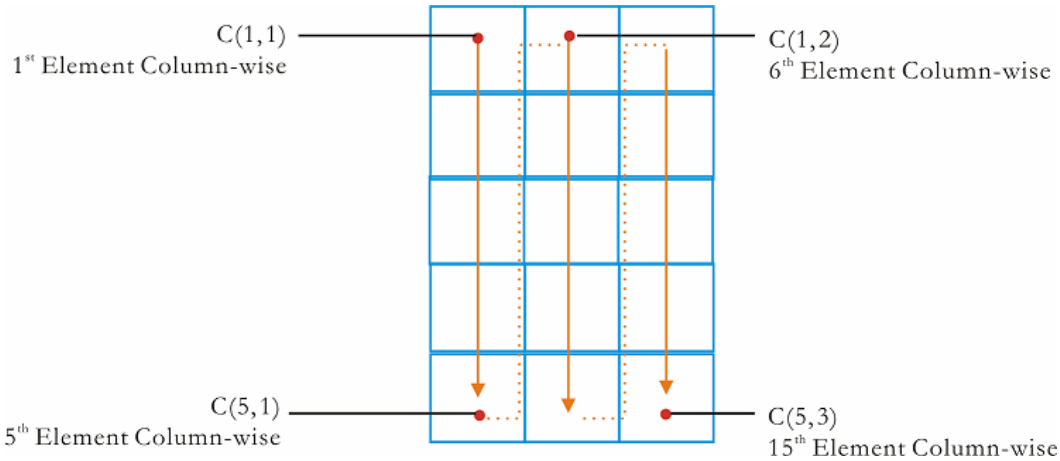


Figure 2.1.3 Element ordering in an mfArray data object

The elements of the mfArray are arranged in such that succeeding columns are placed one after another, into a single long column. The elements are numbered according to their row numbers *(i)*. Thus, as an alternative to describing an element using its element subscript (*See* Access Elements and Sections of an mfArray), you can refer to an element by its position in the actual memory arrangement. From Figure 2.1.3, element *C(5,3)* , can also be referred to as *C(15)*.

2.1.4 Memory Management

MATFOR mfArray is a dynamically allocated data object. As with all Fortran dynamically allocated data objects, it is good practice to free the memory space of variables no longer in use. (Refer to your Fortran compiler's reference manual for more information about dynamic data types and their memory management.)

MATFOR's memory management follows the copy-on-write technique as the memory for mfArrays is always allocated from heap. An mfArray object itself only stores a pointer to a memory space. When copying an mfArray object, it does not actually duplicate the mfArray object, instead it increases the reference count of the mfArray object and lets the two mfArrays objects to share the same memory space. When one of them is modified or deleted, the sharing of the memory space between the two mfArray objects terminates while the reference count is decreased.

MATFOR provides three procedures *msInitArgs*, *msFreeArgs*, and *msRetrunArray*, included in module *mod_ess*, for memory management of mfArrays to prevent them from memory leakage.

Procedure *msInitArgs* stops temporary mfArrays from being released by increasing the reference count. Procedure *msFreeArgs* reduces the reference count so that temporary mfArrays are automatically released by MATFOR.

Procedures *msInitArgs*, *msFreeArgs*, and *msRetrunArray* are used when mfArray is used as a dummy argument for subroutines and functions. Refer to Section 2.7.5 and Section 2.7.6 for more details.

2.1.5 mfArray Syntax and Expressions

MATFOR mfArray has its own sets of operators, inquiry procedures, constructors, logical procedures, and mathematical procedures as described in Section 2.2 to Section 2.6.

Using the MATFOR-defined operators and procedures, you can work with mfArray by referencing the whole array, accessing the individual elements in it, or by treating it as a whole variable.

2.1.6 Mix mfArray and Fortran Arrays

MATFOR caters to the needs of programmers who work interchangeably with mfArray and a Fortran array. Besides the traditional methods of using the assignment operator '=', you can use *msAssign*, *mfEquiv*, and *msPointer* to share data of an mfArray and a Fortran array. More details on the usage of these procedures are covered in Section 2.7.3 and Section 2.7.4. As a quick overview, Table 2.1.6 below summarizes the difference between the different procedures.

Table 2.1.6 Comparison between Equivalency Procedures

Procedure	Memory Duplication	Possibility of Stack Overflow	Main Purpose
mfEquiv	No	No	Used when using a Fortran array to call a MATFOR procedure.
msPointer	No	No	Used when you would like to use Fortran indexing with mfArray or call a Fortran function using mfArray.
msAssign	Yes	No	Memory copy without temporary memory allocation.
=	Yes	Yes	Memory copy. May lead to stack memory overflow for insufficiently allocated stack memory with some Fortran compilers.

2.2 Create and Initialize mfArray

This section provides an opening step to using mfArray in your Fortran program. We shall use examples to introduce the concepts of using MATFOR modules, declaring an mfArray, initializing an mfArray and constructing mfArray using mfArray constructors.

2.2.1 Declaring an mfArray

To use mfArrays in your program, you must use one of the MATFOR modules and declare the mfArray in the variable declaration section of your program.

The following steps outline the general steps of using mfArray in your programs, functions, or subroutines.

Step 1. Add the statement *use <module name>* under your program name, module name, function name, or subroutine name. For example,

```
use fml
use fgl
use mod_ess
use mod_ops
```

Step 2. Declare the mfArray by adding your mfArray declare statement under the *implicit none* declaration. The general format is:

type(mfArray) ::< variable-list> [=< value >]

For Example:

```
Program 2_2_1

! Use module mod_ess

use mod_ess

implicit none
type(mfArray) :: a
```

2.2.2 Initializing an mfArray

The mfArray works with three broad classes of object types, including character, Boolean, and numeric. You can initialize your mfArray by using the following data types:

Table 2.2.2 mfArray Initialization

Data type	Statement
character	a = ‘A’
character(*)	a = ‘string’
Logical	a = .true.
Integer	a = 1
real(4)	a = 1.0
real(8)	a = 3.1418d0
complex(4)	a = (2.0, 3.0)
complex(8)	a = (2.0d0, 3.0d0)

All data are stored as character, double precision real, or double precision complex. The mfArray automatically assumes the shape of the data used for initialization.

Example 2.2.2 below lists some valid initialization of an mfArray. Notice that the mfArray *a* changes its shape and data type automatically with different initialization data.

Example 2.2.2 Initialize mfArray

```
Program Example2_2_2

use fml
implicit none

type(mfArray) :: a
real(8), dimension(10,10) :: b, c

! a is a 1-by-5 vector containing double precision real
! values

a = (/1.0, 2.0, 3.0, 4.0, 5.0/)
call msDisplay(a, 'a')


! a is a 1-by-16 vectors containing 16 characters.
a = "This is a string"
call msDisplay(a, 'a')
```

```
! a is a 1-by-1 double precision complex array.
a = (2.0d0, 1.0d0)
call msDisplay(a, 'a')

! a is a 1-by-1 double precision real array.
a  = 1
call msDisplay(a, 'a')


! a is a 2-by-3 matrix. Operator .vc. is a MATFOR
! operator that concatenates vectors vertically.
! The operator accepts only double precision data.

a = (/1.0d0, 2.0d0, 3.0d0/).vc.(/4.0d0, 5.0d0, 6.0d0/)
call msDisplay(a, 'a')


! a is a 1-by-6 vector. Operator .hc. is a MATFOR
! operator that concatenates vectors horizontally.
! The operator accepts only double precision data.

a = (/1.0d0, 2.0d0, 3.0d0/).hc.(/4.0d0, 5.0d0, 6.0d0/)
call msDisplay(a, 'a')

! a is a 1-by-1 logical array. All values are ones.
b = 10.0
c = 2.0
a = b > c
call msDisplay(a, 'a')

! Deallocate mfArary a
call msFreeArgs(a)

end Program Example2_2_2
```

2.2.3 mfArray Creating Procedures

MATFOR contains a set of mfArray creating procedures for quick creation of special mfArrays. These procedures are located in the *mod_elmat* and *mod_ops* modules. Table 2.2.3 below lists some of the mfArray creating procedures available.

Table 2.2.3 mfArray creating procedures

mod_elmat	
MfEye, msEye	Create identity arrays.

mfLinspace, msLinspace	Create a linearly spaced vector with specified number of points.
mfMagic, msMagic	Create a magic matrix of equal column, row and diagonal sums.
mfMeshgrid, msMeshgrid	Create matrices from vectors for functions of two variables and three-dimensional figure.
mfOnes, msOnes	Create arrays containing all ones.
mfRand, msRand	Create arrays containing random numbers.
mfRepmat, msRepmat	Create an array by tiling smaller arrays.
mfReshape msReshape	Repack vectors into specified array shapes.
mfZeros msZeros	Create arrays containing all zeros.
mod_ops	
.vc.	Vertically concatenate arrays.
.hc.	Horizontally concatenate arrays.
mfColon	Create vectors of specified increment.

Example 2.2.3 below uses the mfArray creating procedures listed above to create mfArrays of a specified shape and data. Notice that in the example, the mfArray *a* changes its shape and data automatically with each creating procedure.

Example 2.2.3 mfArray creating procedures

```
Program Example2_2_3

use fml
implicit none
```

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

! Creates a 3-by-3 identity matrix
a = mfEye(3,3)
call msDisplay(a, 'mfEye(3,3)')


! Creates a 2-by-2 ones matrix
a  = mfOnes(2,2)
call msDisplay(a, 'mfOnes(2,2)')


! Creates a 2-by-2 zeros matrix
a  = mfZeros(2,2)
call msDisplay(a, 'mfZeros(2,2)')


! Creates a 2-by-2-by-2 3-D random array
a  = mfRand(2,2,2)
call msDisplay(a, 'mfRand(2,2,2)')


! Creates a 3-by-3 magic matrix
a  = mfMagic(3)
call msDisplay(a, 'mfMagic(3)')


! Creates a 2-by-2-by-2-by-2 4-D array using
! mfReshape. This is similar to Fortran's
! reshape procedure.
a = mfReshape(mfColon(1,16),(/2,2,2,2/))
call msDisplay(a, &
'mfReshape(mfColon(1,16),(/2,2,2,2/))')

! Creates a vector containing the values from
! 0 to 1, with increment of 0.2. The suffix 'd0'
! forces the number to double precision.
a = mfColon(0d0,0.2d0,1d0)
call msDisplay(a,'mfColon(0d0,0.2d0,1d0)')

! Creates a vector containing 5
! linearly spaced data between the
! numbers -3.1412 to 3.1412. MF_PI
! is a MATFOR parameter for -3.1412.
a = mfLinspace(-MF_PI,MF_PI,5)
call msDisplay(a, &
'mfLinspace(-MF_PI, MF_PI,5)')
```

```
! Creates two matrices, one containing row
! tiling of a vector, and the other column tiling
! of another vector respectively. This
! is useful in creating matrices for drawing
! a surface.
call msMeshgrid(mfOut(b,c),mfColon(1,5),mfColon(6,10))
call msDisplay(b, &
'msMeshgrid(mfOut(b,c),mfColon(1,5), mfColon(6,10)), b', &
c, 'c')


! Creates a matrix containing tiling of a
! vector.
a = mfRepmat(mfColon(1,5),(/2,2/))
call msDisplay(a, 'mfRepmat(mfColon(1,5),(/2,2/))')

! Deallocate mfArrays
call msFreeArgs(a, b, c)

end Program Example2_2_3
```

2.3 Access Elements and Sections of an mfArray

The mfArray is a special MATFOR defined array type. You can use the procedure *mfMatSub* for accessing an element or a range of elements in an mfArray. The subscript convention used by procedures *mfMatSub* is similar to the subscript conventions used in accessing Fortran arrays.

In this section, we shall go through the subscripting conventions used in Fortran 90/95 for accessing individual elements and sections of an array, followed by an introduction to the procedures *mfMatSub*.

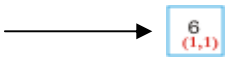
2.3.1 Element Subscripts

In Fortran, array elements are specified through their subscripts containing positions of the element in each dimension. The general syntax is as follows.

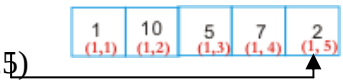
[posdim1, posdim2, posdim3,…, posdim7]

For example,

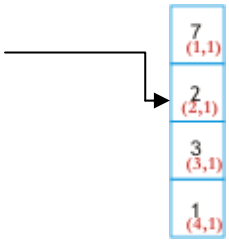
- a scalar has subscript (1, 1)



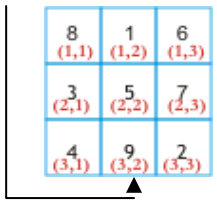
- the fifth element of a row vector is (1,5)



- the second element of a column vector is (2,1)



- the sixth element of a 3-by-3 matrix is (3,2)



2.3.2 mfArray Creating Procedures

An array section is a portion of an array that can be specified by using a combination of vectors. The vector specifies a whole section of individual elements whose positions are described by the element subscript in 2.3.1. Element Subscripts above.

The vector subscript is a rank one array of integer values specifying elements of a section in any order. The vector can contain duplicate values. For example, (/2, 3, 4, 5, 4, 7/) is a valid set of vector subscripts.

As an illustration, the following example specifies three sets of array sections (highlighted in gray) using vector subscript.

- Section subscripts: ((/1, 2, 3/), (/1, 2, 3/))

  The subscript specifies the group of highlighted elements in Figure 2.3.2.1. This is equivalent to specifying the element subscripts (1, 1), (1, 2), (1, 3), (2, 1), (2, 2) (2, 3), (3, 1), (3, 2), and (3, 3).

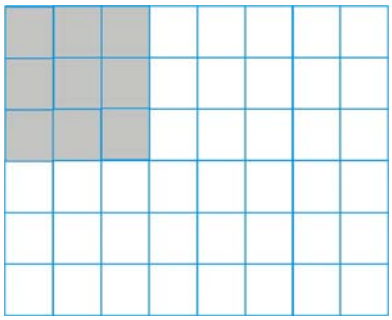


Figure 2.3.2.1 Section subscripts ((/1, 2, 3/), (/1, 2, 3/))

- Section subscripts: ((/2, 4, 6/), (/1, 7, 4/))

  The subscript specifies the group of highlighted elements in Figure 2.3.2.2. This is equivalent to specifying the element subscripts (2, 1), (2, 7), (2, 4), (4, 1), (4, 7) (4, 4), (6, 1), (6, 7), and (6, 4).

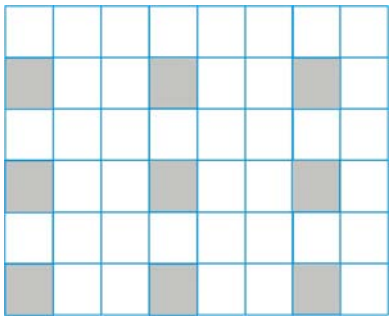


Figure 2.3.2.2 Section subscripts ((/2, 4, 6/), (/1, 7, 4/))

- Section subscripts: ((/2, 3, 4, 5/), 7)

  The subscript specifies the group of highlighted elements in Figure 2.3.2.3.This is equivalent to specifying the element subscripts (2, 7), (3, 7), (4, 7), and (5, 7).

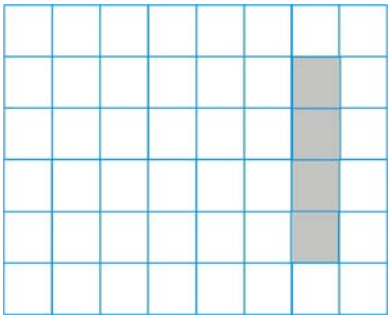


Figure 2.3.2.3 Section subscripts ((/2, 3, 4, 5/), 7)

2.3.3 Using Subscript in an mfArray

The mfArray provides users with the flexibility of automatic dimensioning, emulating the behavior of Matlab arrays. In effect, it means that you do not have to worry about the rank and shape of the array when you work with it. This is especially convenient for Matlab users who would like to translate their works to Fortran.

You can work with procedures with a Matlab-like intuitive interface that does not require users to take care of the data type and dimension of the array input arguments.

To implement the automatic dimensioning feature of mfArray, a special subscript has to be employed — users have to use subscript in mfArray elements through the procedures *mfMatSub* or *mfS* (The abbreviated procedure name).

For Fortran users who prefer to use Fortran array subscripts, you can use procedure *msPointer* to construct a Fortran pointer targeting an mfArray. This way, you can continue to use subscripts in elements of an mfArray through the Fortran pointer. More descriptions of procedure *msPointer* are provided in Section 2.7.3.2.

Procedure *mfMatSub* retrieves data from an mfArray element or subsection. It has the following general syntax:

<mfArray> = mfMatSub(<mfArray>, row, column, …)

The arguments row and column are colon arrays or can be mfArrays.

MATFOR's colon arrays has the format of

i.to.j.step.k

which corresponds to Fortran's colon arrays (i:j:k). .to. and .step. are special operators used only in procedure *mfMatSub*.

Example 2.3.3.1 below illustrates the usage of *mfMatSub* in reading data from mfArrays. The shapes of the mfArray used in the example are chosen to conform to the shape of the arrays used in illustrations in Section 2.3.1 Element Subscripts and Section 2.3.2 Section Subscripts.

Example 2.3.3.1 Use mfMatSub

```
Program Example2_3_3_1

use fml
implicit none

type(mfArray):: a, b

! In the example below, mfArray a is the original array,
! mfArray b is the returned array.

! Read mfArray data using element subscripts

! mfArray is scalar
a = 6
b = mfS(a, 1, 1)
call msDisplay(a, 'a', b, 'mfS(a, 1, 1)')

! mfArray is 1-by-5 row vector
```

```
a = (/1, 10, 5, 7, 6, 2/)
b = mfS(a, 1, 5)
call msDisplay(a, 'a', b, 'mfS(a, 1, 5)')

! mfArray is 4-by-1 column vector
a = reshape((/7, 2, 3, 1/), (/4,1/))
b = mfS(a, 2, 1)
call msDisplay(a, 'a', b, 'mfS(a, 2, 1)')

! mfArray is 4-by-4-by-5 three dimensional array
a = mfRand(4,4,5)
b = mfS(a, 3, 4, 5)
call msDisplay(a, 'a', b, 'mfS(a, 3, 4, 5)')

! mfArray is a 3-by-3 magic square
a = mfMagic(3)
b = mfS(a, 3, 2)
call msDisplay(a, 'a', b, 'b = mfS(a, 3, 2)')

! Read mfArray data using element subscripts

! Construct a 6-by-8 mfArray
a = mfRand(6, 8)
b = mfS(a, 1.to.3, 1.to.3)
call msDisplay(a, 'a', b, 'mfS(a, 1.to.3, 1.to.3)')

b = mfS(a, mf((/2, 2, 6/)), mf((/1, 8, 3/)))
call msDisplay(a, 'a', b, 'mfS(a, mf((/2, 2, 6/), mfS((/1, 8, 3/)))')

b = mfS(a, 2.to.4, 7)
call msDisplay(a, 'a', b, 'mfS(a, 2.to.4, 7)')

! Construct a 6-by-8 matrix
a = mfReshape(mfColon(1,48),(/6, 8/))
call msDisplay(a, 'a')

! Change the value of element at a(5,8) to 0.5
call msAssign(mfS(a, 5, 8), 0.5d0)

! Change the value of element at a(6,5) to 3.14
call msAssign(mfS(a, 6, 5), 3.14d0)
call msDisplay(a, 'a(5,8) = 0.5, a(6,5) =3.14')

! Change the values at section a(mfColon(2,4), 7) to 2
call msAssign(mfS(a, 2.to.4, 7), 2)
call msDisplay(a, 'a(2:4, 7) = 2')

call msFreeArgs(a,b)

end Program Example2_3_3_1
```

2.4 mfArray I/O

This section discusses the procedures for displaying mfArray data: *msDisplay* and *msGDisplay* and the procedures for saving and loading mfArray data to or from files: *mfLoad* and *msSave.*

The section is further divided into the following subsections:

2.4.1 Displaying mfArray Data – This subsection introduces the usage of procedures *msDisplay* and *msGDisplay*. Procedure *msDisplay* displays mfArray data in short format on a Windows console, analogous to the write(*,*) function, but much easier to use. Procedure *msGDisplay* displays mfArray data in a spreadsheet-like format in MATFOR Data Viewer. From here, you can examine the mfArray data, perform statistic analysis on them, and even export it to an Excel spreadsheet.

2.4.2 mfArray Fileio – This subsection introduces the usage of procedures *mfLoad* and *msSave*. These procedures, together with *mfLoadAscii* and *msSaveAscii,* enable you to export mfArray data to an external file in ASCII or binary format for further processing.

We will also cover the two Matlab *.m* file, *mfLoad.m* and *mfSave.m,* provided by MATFOR for exchanging data between Matlab and MATFOR.

2.4.1 Displaying mfArray Data

To view the run-time contents of an mfArray you can use two procedures, *msDisplay* and *msGDisplay,* provided by MATFOR. Procedure *msDisplay* outputs the content of mfArray to a console. Procedure *msGDisplay* outputs the content to the MATFOR Data Viewer.

msDisplay

Procedure *msDisplay* is included in module *mod_ess.* You can use the procedure to specify multiple mfArrays for displaying in the Windows console. The procedure has the following syntax:

call msDisplay(x)

call msDisplay(x, 'name1', y, 'name2',..)

The second multiple-input format requires the mfArray to be specified together with a string type argument-*'name'*. The number of mfArrays that you can display by using a single procedure is limited to 32.

By default, *msDisplay* displays data in the *'short'* format. That is, real numbers are displayed with four decimal places. You can use procedure *msFormat* to change the display format to *'long'*. In which case, real numbers are displayed with 16 digits. The procedure displays data as integers when an mfArray is initialized with integers.

Example 2.4.1.1 below uses procedure *msDisplay* to display mfArrays. Go through the example, try it on your compiler, you will see how integers, *'short'*, and *'long'* format of mfArray data are displayed on the Windows console.

Example 2.4.1.1 Use msDisplay

```
Program Example2_4_1_1

use fml

implicit none
type(mfArray) :: a

! Construct mfArray a using mfMagic procedure
a = mfMagic(3)

! Display a. Notice that the data is displayed as integer.
call msDisplay(a, 'mfMagic(3)')

a = mfRand(2,2)

! Display mfRand(2,2)
call msDisplay(a, 'mfRand(2,2)')

! Change the display format to long
call msFormat('long')

! Display a again
call msDisplay(a, 'Long mfRand(2,2)')
```

```
! Deallocate mfArray from memory
call msFreeArgs(a)

end Program Example2_4_1_1
```

Compile and run the program. The Windows console pops up as shown in Figure 2.4.1.1 below:

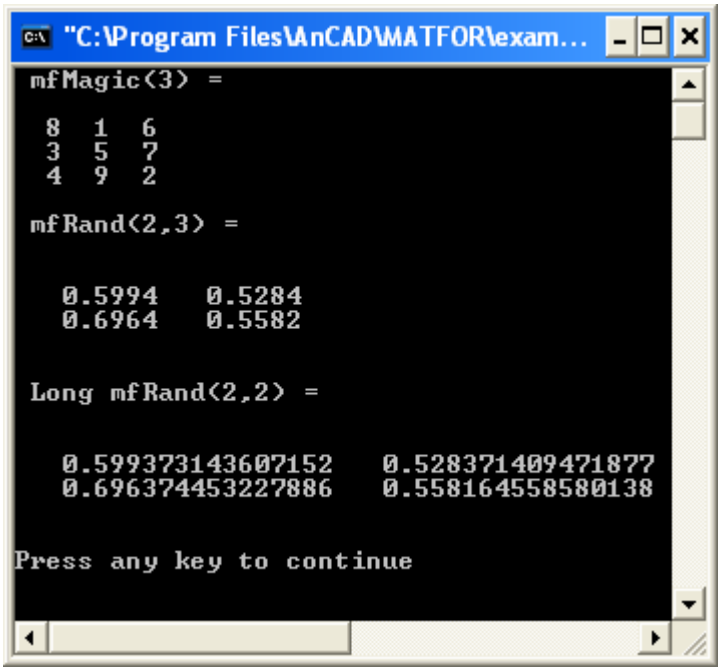


Figure 2.4.1.1 Windows Console displaying mfArray a

Notice that the first line on Figure 2.4.1.1 displays the name of the mfArray as specified by the name argument, such as *'mfMagic(3)'* in the procedure call. If the name is not specified, *'ans'* representing answer is displayed instead. The '=' symbol is automatically added after the name argument.

msGDisplay

Procedure *msGDisplay* is included in the *fgl* module. The procedure displays the contents of mfArrays in the MATFOR Data Viewer.

The syntax of *msGDisplay* is the same as procedure *msDisplay*.

call msGDisplay(x)

call msGDisplay (x,'name1',y,'name2',…)

Example 2.4.1.2 Use msGDisplay

In this example, we shall create a 3-by-3 magic square mfArray *a*, and compute its row sum, column sum, and diagonal sum, then display the data using procedure *msGDisplay* in the MATFOR Data Viewer.

```
Program Example2_4_1_2

use fml

use fgl

implicit none

type (mfArray) :: a, b, c, d

! Magic(3) creates a 3-by-3 matrix with equal

! row and column sums.

a = mfMagic(3)

! Compute the column, row and diagonal sum of a.

b = mfSum(a,1)

c = mfSum(a,2)

d = mfSum(mfDiag(a))
```

```
! Display a,b,c,d using a Data Viewer
call msDisplay( a, 'a', b, 'b', c, 'c', &
d, 'd')

call msGDisplay( a, 'a', b, 'b', c, 'c', &
d, 'd')


! Pause Program for viewing
call msViewPause()

call msFreeArgs(a, b, c, d)

end Program Example2_4_1_2
```

Compile and **run** the program. The MATFOR Data Viewer is displayed, as shown in Figure 2.4.1.2

You can switch between each mfArray by clicking on the Worksheet tabs. The names of the tabs are specified by the name arguments in *call msGDisplay(a, 'a', b, 'b', c, 'c', d, 'd'*), in the second, fourth, sixth, and eighth position respectively.

Navigate between the worksheets to view the values of mfArray.

You will see that *b* is a row vector containing the elements (/15.0000, 15.0000, 15.0000/) corresponding to each column sum. mfArray *c* is a column vector containing the same values while *d* is a scalar with value 15.0000. The row, column, and diagonal sums are the same. The magic square is truly magical!

From Figure 2.4.1.2 you can see that the Data Viewer has many functions available for editing the mfArray data. Play with the buttons to get a feel of each function.

You may refer to Section 4.8 MATFOR Data Viewer for tutorials on manipulations of the Data Viewer functions.

Figure 2.4.1.2 The Data Viewer displaying mfArrays a, b, c, and d

Procedure msViewPause
Procedure *msViewPause* is added to your program to pause program execution for graphical displays. You will need to add this statement to every set of graphical creation routines. If this routine is not called, the Graphical windows will just flash on your computer.

2.4.2 mfArray File I/O

MATFOR supports text and binary format for importing and exporting data to and from mfArray. The procedures provided are: *mfLoad*, *msSave*, *mfLoadAscii*, and *msSaveAscii*. These procedures are located in module *mod_fileio.*

For Matlab users, MATFOR provides two *.m* files — *mfLoad.m and mfSave.m,* for interfacing with the Matlab environment to facilitate the exchange of data in binary format.

In the section below, we shall cover the usage of two procedures, *msSaveAscii* and *mfLoadAscii,* which exports and imports data to and from text files, followed by a discussion on *mfLoad.m* and *mfSave.m*.

2.4.2.1 msSaveAscii

You can save data of mfArray to a text file by using procedure *msSaveAscii*. Procedure *msSaveAscii* has the following syntax:

call msSaveAscii(<mfArray>,'filename')

MATFOR saves the mfArray data in ASCII format, with the data arranged in rows and columns corresponding to the rows and columns of a matrix mfArray. This format is the same as that used by Matlab *Save* function with the *–ascii* option. Example 2.4.2.1 below exports the data of a 3-by-3 magic square to a text file using procedure *msSaveAscii*.

Example 2.4.2.1 Procedure msSaveAscii

```
Program Example2_4_2_1

use fml

implicit none
type(mfArray) :: a

! Construct a 3-by-3 magic square
a  = mfMagic(3)

! Export a to a text file
call msSaveAscii('a.txt',a)

! Deallocate mfArray
call msFreeArgs(a)

end Program Example2_4_2_1
```

Compile and run the program. Open the text file *'a.txt.'* The data are arranged as in Figure 2.4.2.1.

```
8.0000000E+00  1.0000000E+00  6.0000000E+00
3.0000000E+00  5.0000000E+00  7.0000000E+00
4.0000000E+00  9.0000000E+00  2.0000000E+00
```

Figure 2.4.2.1 Content of "a.txt"

2.4.2.2 mfLoadAscii

You can load data from a file into an mfArray by using procedure *mfLoadAscii* or procedure *mfLoad*. Procedure *mfLoadAscii* loads a text file in the format used by *msSaveAscii* or Matlab ASCII data file. Procedure *mfLoad* loads a MATFOR *.mfb binary file created by using procedure *msSave*. Both procedure *mfLoadAscii* and *mfLoad* have the same syntax and are located in module *mod_fileio*. In the following, we will look more closely at the application of procedure *mfLoadAscii*.

Procedure *mfLoadAscii* has the following syntax:

<mfArray> =mfLoadAscii('filename')

where *'filename'* is a string specifying the name of the text file to be loaded. Example 2.4.2.2 below loads the text file *'a.txt'*, created in Example 2.4.2.1 using procedure *mfLoadAscii,* into mfArray *a* and displays the mfArray.

Example 2.4.2.2

```
Program Example2_4_2_2

use fml

implicit none
type(mfArray) :: a

! Import data into mfArray a using mfLoadAscii
a  = mfLoadAscii('a.txt')

! Display data of a
call msDisplay(a, 'a')

! Deallocate mfArray
call msFreeArgs(a)

end Program Example2_4_2_2
```

2.4.2.3 mfLoad.m and mfSave.m

The Matlab *.m f*iles, *mfLoad.m* and *mfSave.m,* are installed in the folder *<MATFOR>\common\tools\matlab* when you install MATFOR. These two files call the MATFOR *mfLoad.dll* and *mfSave.dll,* to export and load MATFOR binary data file, **.mfb,* to and from the Matlab workspace. Copy the *.m* and *.dll* files into your Matlab working directory and you can start exchanging binary data between Matlab and MATFOR.

The functions m*fLoad* and *mfSave* have the following syntax,

x = mfLoad(filename)

mfSave(filename, x)

where *x* is a Matlab matrix, and *filename* is a string containing the name of the target binary file. If the file extension is not specified, the file extension *.mfb* is automatically appended.

In your Fortran environment, you can retrieve the data contained in a **.mfb* binary file, exported using function *mfSave* in Matlab, into MATFOR mfArray through procedure *mfLoad*. Likewise, you can save your MATFOR mfArray data in binary format using procedure *msSave* and load it into a Matlab matrix using function *mfLoad* in Matlab.

Example 2.4.2.3 Exchange binary data between Matlab and MATFOR

In this example, we shall export a binary file from Matlab using function *mfSave* and retrieve the data in Fortran using procedure *mfLoad.*

Step 1. First, ensure that the files *mfLoad.m, mfSave.m, mfLoad.dll,* and *mfSave.dll,* installed in *<MATFOR>\common\tools\Matlab*, are in your Matlab working directory.

Step 2. We shall export the data in *[X, Y, Z]* matrices, computed using Matlab sample function –*peaks*, to a MATFOR binary file, using the following commands in Matlab workspace.

```
[X, Y, Z] = peaks;
mfSave('X.mfb', X);
mfSave('Y.mfb', Y);
mfSave('Z.mfb', Z);
```

The data from matrices *X, Y, Z* are saved as *X.mfb, Y.mfb,* and *Z.mfb,* respectively in your Matlab working directory.

Step 3. Copy the binary files into your MATFOR project file. In this case, we shall copy the files into *<MATFOR>\Fortran\example\for_ug.*

Step 4. Start a Fortran program titled Example2_4_2_3. We shall retrieve the data into mfArrays *x, y,* and *z* and plot the data using procedure *msSurf.*

```
Program Example2_4_2_3

use fml
use fgl
implicit none

type(mfArray) :: x, y, z

x = mfLoad('.\data\x.mfb')
y = mfLoad('.\data\y.mfb')
z = mfLoad('.\data\z.mfb')

call msSurf(x,y,z)

call msViewPause()

call msFreeArgs(x,y,z)

end Program Example2_4_2_3
```

Step 5. Compile and run the program. Figure 2.4.2.3 displays.

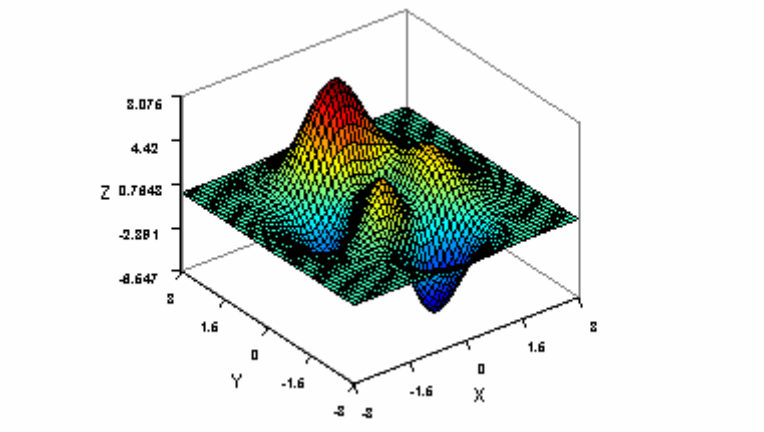


Figure 2.4.2.3 Surface plot produced by importing data from Matlab

2.5 mfArray Inquiry Procedures

The mfArray has a set of inquiry procedures for querying its data type, array type, and attributes. In this section, we shall look into the three types of inquiry procedures as below.

2.5.1 Logical Inquiry Procedures – This topic covers the logical functions that are used to query the status of mfArrays, such as *mfIsNumeric*, *mfIsReal*, etc. The logical functions return a logical .true. or .false.

2.5.2 Size, Shape, and Extent – This topic covers the functions for checking array properties such as array size, shape, rank, and extent.

2.5.3 Logical Operations – This topic covers the functions used for querying ones and zeros in an mfArray.

2.5.1 Logical Inquiry

The logical inquiry procedures listed in Table 2.5.1 return Fortran logical type as output. The logical inquiry procedures are located in module *mod_ess.*

Table 2.5.1 Logical Inquiry Procedures.

mod_ess	
l = mfIsEmpty(a)	Return *.true.* if mfArray is empty.
l = mfIsNumeric(a)	Return *.true.* if mfArray contains numerical data.
l = mfIsReal(a)	Return *.true.* if mfArray is real.
l = mfIsComplex(a)	Return *.true.* if mfArray contains complex values.
l = mfIsLogical(a)	Return *.true.* if mfArray is logical.

Usually, MATFOR procedures return mfArray as output argument. However, in the case of logical inquiry procedures, Fortran logical types are returned. This design is adopted to simplify the programming involved in *if* constructs. You can use these procedures to determine the data type of mfArrays, compare mfArrays, and apply them directly in an *if* construct.

For example,

type(mfArray) :: a

if (mfIsEmpty(a)) then a = 5

Below, we go through details on the application of each procedure.

- *mfIsEmpty* – An mfArray is empty if it has not been initialized and points to a null storage space.

- *mfIsLogical* – An mfArray is logical if it is constructed using logical operations. The statements below construct a logical mfArray.

```
type (mfArray) :: a, b
-----
a = b>5
```

- *mfIsNumeric* – An mfArray is numeric if it contains complex, logical, or real data type. A character type mfArray is non-numeric.

- *mfIsReal* –A real mfArray contains real data type.

- *mfIsComplex* –A complex mfArray contains double precision complex data type.

Example 2.5.1 Logical Inquiry Procedures

The example below uses the Logical Inquiry procedures.

```
Program Example2_5_1

use fml
implicit none

type(mfArray) a, b, c, d
logical :: L

a = mfMagic(3)
d ='string'

! Is mfArray C empty?

If(mfIsEmpty(c)) then
   c = (2,-2)
end if

! Is mfArray Complex?
If (mfIsComplex(c)) then
  call msDisplay(c, 'c is complex')
end if

! Is mfArray numeric?
L = mfIsNumeric(d)
call msDisplay(d, 'd', mf(L), 'mfIsNumeric(d)')

! Is mfArray numeric Real?
L = mfIsReal(c)
call msDisplay(c, 'c', mf(L), 'mfIsReal(c)')

! Is mfArray logical?
d = mfANY(a)
If (mfIslogical(d)) then
    write (*,*) 'd is logical'
```

```
end if

call msFreeArgs(a, b, c, d)

end Program Example2_5_1
```

2.5.2 Size, Shape, and Extent

You can get information on the number of elements, shape, and extent of an mfArray using procedures *Size*, *mfNDims*, *Shape*, *mfLength*, and *mfSize*. These procedures are located in modules *mod_ess* and *mod_elmat* as listed in Table 2.5.2 below.

Table 2.5.2 Size, Shape and Extent

mod_ess	
S = mfSize(a)	Return the total number of elements.
S = mfSize(a, dims)	Return the extent along the specified dimension.
N = mfNDims(a)	Return the rank or number of dimensions.
V = mfShape(a)	Return the shape of mfArray.
S = mfLength(a)	Return the largest extent of mfArray.
mod_elmat	
b = mfSize(a)	Return a vector containing the shape of the mfArray.
b = mfSize(a, dims)	Return the extent along the specified dimension.
call msSize(mfOut(n1,n2,…), a)	Return the lengths of the first n dimensions of mfArray a.

These procedures return Fortran scalars, vectors, or mfArray as output. Procedures *Size*, *mfNDims*, and *mfLength* returns a Fortran scalar integer data type as output. Procedure *Shape* returns a Fortran vector integer data type as output. Procedure *mfSize* returns an mfArray as output.

To get the shape of an mfArray, you can use either procedure *Shape*, which returns a Fortran integer vector, or procedure *mfSize*, which returns an mfArray. The following statements are examples of valid inquiries for getting the shape of *a*, a square mfArray.

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

```
integer :: V(2)
---
! To get shape of a, you can use....
b = mfSize(a)

! or
V = Shape(a)
```

To get the size or total number of elements in an mfArray, use procedure *Size*.

```
type(mfArray) :: a
integer :: S
---
! Get total number of elements of a
S = Size(a)
```

To get the rank or number of dimensions of an mfArray, use procedure *mfNDims*.

```
type(mfArray) :: a
integer :: N
---
! Get number of dimensions or rank of a
N = mfNDims(a)
```

To get the largest extent, or the extent in a certain dimension, of an mfArray, use procedures *mfLength*, *SIZE*, or *mfSize*.

```
type (mfArray) :: a, e
integer :: S
---
! Get largest extent
S = mfLength(a)

! Get size in specified dimension ==> 1
S = Size(a, 1)
! or
e = mfSize(a,1)
```

Example 2.5.2 below provides further examples on the usage of procedures *Size*, *mfNDims*, *Shape*, and *mfLength* to determine the size, shape, extent, and rank of an mfArray.

Example 2.5.2 Size, shape and extent

```
Program Example2_5_2

use fml
implicit none

type(mfArray) :: a, b, m, n, o
integer :: S, S1(3)


! Construct a 2-by-2-by-4 mfArray
a = mfRand(2,3,4)
call msDisplay(a, 'a')

! Procedure Size
S = mfSize(a)
call msDisplay(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)')

S = mfSize(a,3)
call msDisplay(mf(S),'mfSize(a,3)')



! Procedure mfNDims
b = mfNDims(a)
call msDisplay(b, 'mfNDims(a)')


! Procedure Shape
S1 = SHAPE(a)
call msDisplay(mf(S1), 'SHAPE(a)')

! Procedure mfLength
S = mfLength(a)
call msDisplay(mf(S), 'mfLength(a)')


! Deallocate mfArray
call msFreeArgs(a,b,m,n,o)

end Program Example2_5_2
```

2.5.3 Logical Operations

You can use procedures *All*, *Any,* and *mfFind* as listed in Table 2.5.3 to find nonzero elements in mfArray. These procedures are located in modules *mod_ops* and *mod_elmat*.

Table 2.5.3 Logical Operations

mod_ess	
L = All(a)	Return .true. if all are nonzero.
L = Any (a)	Return .true. if any is nonzero.
mod_ops	
b = All(a)	Return a vector containing the status of nonzero in each column. If a column contains all nonzero, then its corresponding status is true.
b = All (a, dims)	Return a vector containing the status of nonzero in specified dimension.
b = Any(a)	Return a vector containing the status of nonzero in each column. If a column contains any nonzero, then its corresponding status is true.
b = Any(a, dims)	Return a vector containing the status of nonzero in specified dimension.
mod_elmat	
I = mfFind(a)	Return the column major indices of nonzero elements.
call msFind(mfOut(i,j),a)	Return the nonzero elements subscripts in vectors i and j.
call msFind(mfOut(i,j,v),a)	Return the nonzero elements subscripts in vectors i and j, and the nonzero values in vector v.

Each procedure provides different information about the nonzero elements in mfArray. In MATFOR, logical *.false.* is represented as zero while *.true.* is represented as a number one.

- Procedures *All* and *Any* query the status of nonzero elements of mfArray. The procedures operate on the whole mfArray and return a Fortran logical scalar as output. For example,

```
type(mfArray) :: a
logical :: L
---
! Return true if all elements of a > 2
L = All(a>2)

! Return true if any elements of a > 2
L = Any(a>2)
```

- Procedures *mfAll* and *mfAny* operate on specified dimensions of mfArray and returns an mfArray as output. By default the procedures operate column-wise. For example,

  ```
  type(mfArray) ::: a, b
  ----
  ! If a is a 2-by-2 mfArray, b is a size 2 vector.
  b = mfAll(a)
  ```

  In the example above, mfArray *b* is a vector containing the status of each column of mfArray *a*. If all elements of (shape 2-by-2) mfArray *a is greater than zero*, then *b* is *(/1,1/)*. If only the first column contains all nonzero, while the second column contains zeros, then *b* is *(/1,0/)*.

You can use procedure mfFind to get indices of non-zero elements of mfArray. Depending on your input argument, you can return: 1) a vector mfArray containing the column-major index, 2) two vector mfArrays containing the corresponding row and column element subscripts, or 3) three vectors containing the row and column element subscripts and values corresponding to nonzero values. For example,

```
type(mfArray) :: a, i, j, v
---
! Vector i contains the column-major indices of non-zeros
i = mfFind(a)

! Vectors i and j contain the row and column index of non-zero
call msFind(mfOut(i,j), a)

! Vectors i and j contain the row and column index of non-zeros,
! while vector v contains the corresponding non-zero elements.
call msFind(mfOut(i,j,v), a)
```

Example 2.5.3 provides further examples on the usage of procedures *ANY*, *ALL*, *mfAll*, *mfAny*, *mfFind* to query the status of nonzero elements.

Example 2.5.3

```
Program Example2_5_3

use fml
implicit none

type(mfArray) :: a, b, c, d
logical :: l

a = mfMagic(3)

! Procedure ALL
l = mfAll(a>2)
call msDisplay(mf(l), 'mfAll(a>2)')

b = mfAll(a>2,1)
call msDisplay(b, 'mfAll(a>2,1)')

b = mfAll(a>2,2)
call msDisplay(b, 'mfAll(a>2,2)')



! Procedure ANY
l = mfAny(a>2)
call msDisplay(mf(l), 'mfAny(a>2)')

b = mfAny(a>2,1)
call msDisplay(b, 'mfAny(a>2,1)')

b = mfAny(a>2,2)
call msDisplay(b, 'mfAny(a>2,2)')



! Procedure mfFind
b = mfFind(a>2)
call msDisplay(a, 'a')
call msDisplay(b, 'mfFind(a>2)')

call msFind(mfOut(b,c),a)
call msDisplay(b, 'call msFind(mfOut(b,c) a), b', c, 'c')

call msFind(mfOut(b,c,d),a)
call msDisplay(b, 'call msFind(mfOut(b,c,d) a), b', c, 'c',d,'d')

! Deallocate mfArray
call msFreeArgs(a, b, c, d)
```

```
end Program Example2_5_3
```

2.6 mfArray Operators

MATFOR mfArray supports the usual set of Fortran 90/95 operators and a set of MATFOR - defined operators and operator functions. Table 2.6 below lists the available MATFOR operators and operator functions. The operators are listed according to their precedence.

MATFOR-defined operators include matrix transpose *(.t.)*, matrix complex transpose *(.h.)*, horizontal concatenation *(.hc.)*, and vertical concatenation *(.vc.)*. The operator functions include *mfMul* for matrix multiplication, *mfLDiv* for matrix left divide, and *mfRDiv* for matrix right divide. These operators and operator functions enable you to perform matrix manipulations conveniently.

Table 2.6 mfArray Operators and Operator functions.

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

.hc.	Horizontal concatenation	Lowest
.vc.	Vertical concatenation	
c = mfColon(s,e)	Colon function	
c = mfMul(a,b)	mfArray matrix multiplication function	
c = mfLDiv(a,b)	mfArray matrix left division function	
c = mfRDiv(a,b)	mfArray matrix right division function	

2.6.1 Arithmetic Operators

The mfArray arithmetic operators *, **, /, \, +, -, operate element-wise on the mfArray. Example 2.6.1 shows some valid operations of the arithmetic operators.

Example 2.6.1 Arithmetic operators

```
Program Example2_6_1

use fml
implicit none

type(mfArray) :: a, b, c, d, e, f

a = mfOnes(3,3)
call msDisplay(a, 'a')

! * element-by-element multiplication
b = 2*a
call msDisplay(b, '2*a')

! ** element-by-element power
c = b**2
call msDisplay(c, 'b**2')

! - element-by-element subtraction
d = c -a
call msDisplay(d, 'c-a')

! / element-by-element division
e = c/b
call msDisplay(e, 'c/b')
```

```
! Deallocate mfArray
call msFreeArgs(a, b, c, d, e, f)

end Program Example2_6_1
```

2.6.2 Relational Operators

The mfArray relational operators include >=, >, <=, <, /=, ==. These operators perform element-by-element comparisons between mfArrays that conform in size and shape, or between mfArray and a scalar. These operators return a logical mfArray. Example 2.6.2 shows some valid operations of the mfArray relational operators.

Example 2.6.2 mfArray Relational Operators

```
Program Example2_6_2

use fml
implicit none

type(mfArray) :: a, b, c

a = mfMagic(3)
b = 2*mfRand(3, 3)

call msDisplay(a, 'a', b, 'b')

! >= element-by-element greater than or equal comparison
c = a >= 3
call msDisplay(c, 'a >= 3')

! > element-by-element greater than comparison
c = a > b
call msDisplay(c, 'a > b')

! < = element-by-element less than or equal comparison
c = a <= 5
call msDisplay(c, 'a <= 5')

! < element-by-element less than comparison
c = a < b
call msDisplay(c, 'a < b')

! /= element-by-element not equal comparison
c = a /= b
call msDisplay(c, 'a /= b')

! == element-by-element equal comparison
```

```
c = a == b
call msDisplay(c, 'a == b')


! Deallocate mfArray
call msFreeArgs(a, b, c)

end Program Example2_6_2
```

2.6.3 Matrix Operators and Functions

You can perform matrix operations using MATFOR defined operators such as *.t., .h., .hc., .vc.,* and operator functions such as *mfMul, mfLDiv, mfRDiv.*

The *.t.* transpose operator performs a matrix transpose.

Example,

a =

8 1 6

3 5 7

4 9 2

b = *.t.a* =

8 3 4

1 5 9

6 7 2

The *.h.* complex transpose operator performs a complex conjugate transpose.

Example,

a =

```
                1 + 2i   2+3i

b = .h.a

                1 - 2i

                2 - 3i
```

Matrix multiplication function, *mfMul(x, y)*, returns the linear algebraic product of two mfArrays, *x* and *y*, where *x* is an m-by-p matrix and *y* is a p-by-n matrix. The product returns an m-by-n matrix.

Matrix left division function, *mfLDiv(a, b)*, and matrix right division function, *mfRDiv(a, b)*, solves linear matrix inverse problems.

The result of *mfLDiv(a, b)* is approximately *mfMul(mfInv(a),b)*. The result of *mfRDiv(a, b)* is approximately *mfMul(b, mfInv(a))*. Depending on the structure of the mfArray, MATFOR uses different algorithms for the computation as shown in Figure 2.6.3 below. More details on the difference between matrix right division and matrix left division are covered under *Matrix Division* below.

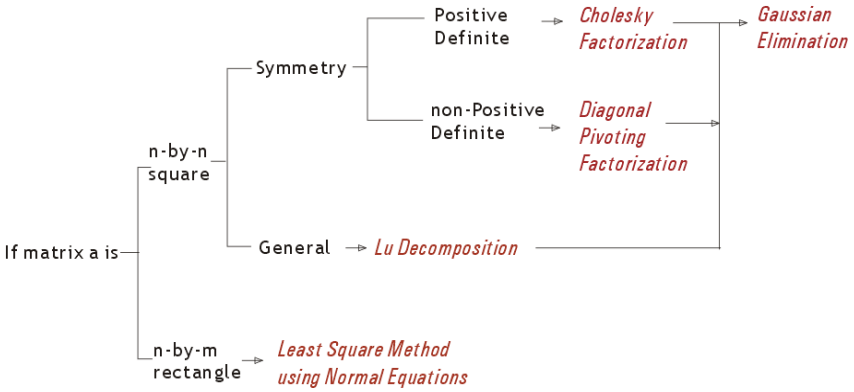


Figure 2.6.3 Algorithms applicable for each matrix type in matrix division operation

Matrix Division

Matrix division is often used to solve the linear matrix inverse problem *ax = b*, where *a* is m-by-m square matrix, while *x* and *b* are m-by-1 column vectors. There are, however, other fields of study that prefer writing the equation in a different format. The appreciation of writing *x* and *b* as row vectors has turned the equation into *xa = b*. To

accommodate both conventions, MATFOR introduces left and right matrix divisions. Use procedures *mfLDiv* or *msLDiv* to solve systems where matrix *a* is put on the left of unsolved variable *x*. On the other hand, use procedures *mfRDiv* or *msRDiv* for row vector major problem. For example:

x = mfLDiv(a, b)

solves for *x* in equation *ax = b* and

x = mfRDiv(a,b)

solves for *x* in equation *xa = b*.

In many cases *x* and *b* are more than vectors. They are matrices representing different conditions for the same system. For a given large matrix *b*, we might want to save storage by putting the solution *x* into the same memory. Instead of using functions, this can be done without extra memory copying by using subroutines *msLDiv* and *msRDiv*. Expressions like

call msLDiv(mfOut(b), a, b)

and

call msRDiv(mfOut(b), a, b)

put the solution *x* back to the memory of mfArray *b*.

Note that if *x* is assigned to be the *mfOut* argument, these expressions are identical to using the function format *x = mfLDiv(a, b)* and *x = mfRDiv(a, b).*

For non-square matrix *a,* equation *ax = b* represents an over-determined or under-determined system. In either case, matrix division routines provide solutions in least square sense.

Example 2.6.3 below lists some valid operations of the matrix operators and operator functions.

Example 2.6.3 Matrix Operators

```
Program Example2_6_3

use fml
implicit none

type(mfArray) :: a, b, c

! .t. matrix transpose
a  = mfMagic(3)
b = .t.a
call msDisplay(a, 'a', b, '.t.a')


! .h. complex conjugate transpose
a =(/(1,2),(2,3)/)
b = .h.a
call msDisplay(a, 'a', b, '.h.a')


! , matrix multiplication
a = mfRand(3,3)
b = mfRand(3,2)
c = mfMul(a,b)
call msDisplay(a, 'a', b, 'b', c, 'mfMul(a,b)')


! .mldiv. matrix left division
b = mfMul(mfInv(a),c)
call msDisplay(b, 'mfMUL(mfInv(a),c)')
```

```
b = mfLdiv(a,c)
call msDisplay(b, 'mfLDiv(a,c)')


! Deallocate mfArrays
call msFreeArgs(a, b, c)

end Program Example2_6_3
```

2.6.4 Operators Precedence

MATFOR operators follow the Fortran intrinsic operator precedence. The precedence of the operators is as shown in Table 2.6 above. In the following, we look at the operation of mfArray *x, a, b, c, d, e,* and *f,* using MATFOR operators, and the resulting operation when the operator precedence is put into perspective.

For example, the following expression,

*x = .t.a**2 + b/5.0d0 - c**d + 2*mfMul(e,f)*

is equivalent to

*x = ((.t.a)**2) + b/5.0d0 - (c**d) + 2*mfMul(e,f)*

as *.t.* has the highest precedence followed by **. This is equivalent to

*x =((.t.a)**2)+(b/5.0d0)-(c**d)+(2*mfMul(e,f))*

as / and * are the next highest, followed by – and +.

The codes for the above statements are presented in Example 2.6.4.

Example 2.6.4 Operator precedence

```
Program Example2_6_4

use fml
implicit none

type(mfArray) ::x, a, b, c, d, e, f

! Array construction
a = mfMagic(3)
b = mfRand(3,3)
c = 2*mfOnes(3,3)
d = mfReshape(mfColon(1,9),(/3,3/))
e = mfRand(3,2)
f = mfRand(2,3)

! Expression 1
x = .t.a**2 + b/5.0d0 - c**d + 2*mfMUL(e,f)
call msDisplay(x, 'x = .t.a**2 + b/5.0d0 - c**d + 2*mfMUL(e,f)')

! Expression 2: Equivalent to expression 1. as .t. has the highest
precedence
!              followed by **.
x = ((.t.a)**2) + b/5.0d0 - (c**d) + 2*mfMUL(e,f)
call msDisplay(x, 'x = ((.t.a)**2) + b/5.0d0 - (c**d) + 2*mfMUL(e,f)')

! Expression 3: Equivalent to expressions 1 and 2 as / and * has a
higher precedence
!             than - and +.

x = ((.t.a)**2) + (b/5.0d0) - (c**d) + (2*mfMUL(e,f))
call msDisplay(x, 'x = ((.t.a)**2) + (b/5.0d0) - (c**d) + (2*mfMUL(e,f))')

call msFreeArgs(x, a, b, c, d, e, f)

end Program Example2_6_4
```

2.6.5 MATFOR Parameters

Table 2.6.5 below lists some MATFOR pre-defined parameters, provided for your convenience.

Table 2.6.5 MATFOR parameters

Parameter	Data Type	Description
MF_EMPTY	mfArray	Empty mfArray
MF_COLON	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

2.7 Program with mfArray

This section focuses on the various issues encountered when you program with mfArray. The topics covered are listed below.

2.7.1 Quick Conversion – Function mf() — this topic introduces the ubiquitous function *mf()*. It is a convenient function for you to convert non–mfArray data into mfArray that can be used as MATFOR procedure input arguments.

2.7.2 Use mfArray in If Constructs — this topic goes through examples of using mfArray in *if* constructs and note the special requirements.

2.7.3 Use mfArray as input to Fortran Procedures — this topic looks at creating an equivalent Fortran array that shares the content of the mfArray, as an input to a Fortran procedure. The assignment operator and procedure *msPointer* are covered.

2.7.4 Use Fortran Arrays as input to MATFOR Procedures — this topic looks at using procedure *mfEquiv* to construct an mfArray that shares the same memory address as the Fortran array.

2.7.5 Use mfArray as Dummy Arguments — this topic looks at the special requirements when using mfArray as dummy arguments in functions and subroutines.

2.7.6 Use mfArray as Returned Dummy Arguments in Function — this topic covers the requirements for using mfArray as the returned argument in a function.

2.7.1 Quick Conversion – Function mf()

Function *mf()* provides a quick conversion from Fortran data objects into an mfArray.

The general syntax is: *mf(<Fortran Data>)*

The Fortran data object can be a scalar, vector, matrix, or arrays of up to seven dimensions. Its data type can be integer, real (4), and real (8), complex (4), complex (8), or strings.

The function is particularly useful in cases where you wish to use a MATFOR procedure that accepts only mfArrays as input argument. For example, procedure msDisplay accepts only mfArrays as input argument. By using function *mf()*, you can display a Fortran data.

call msDisplay(mf((/1, 2, 3, 4, 5/)), 'Vector')

2.7.2.Using mfArray in If Constructs

There are four types of *if* constructs in Fortran 90/95, namely:

if (<logical statement>) <execution statement>

if ... then ...

if ... then ... else ... endif

if ... then ... elseif ... else ...endif

All of these four *if* constructs use the <*logical statement*> or a logical scalar, *.true.* or *.false.*, for conditional control.

mfArray logical inquiry procedures such as *mfIsEmpty*, *mfIsReal*, etc, and logical procedures such as mf*All*, *mfAny*, can be used directly as the <*logical statement*>. For example,

```
If (mfIsNumeric(a)) write (*,*) 'mfArray is numeric!'

If (mfAll(a>2)) then
a = b +c
end if
```

However, MATFOR logical operators such as <, >, <=, >=, ==, return mfArray as output, thus they are not applicable to the <*logical statement*> directly. Instead, use procedure mf*All* or mf*Any* to get a scalar logical result.

For example,

```
If (mfAll(a==2)) then
 a = a + 3
end if

If (mfAny(a>=2)) then
```

```
 a = b > 2
end if
```

2.7.3 Using mfArray as Input to Fortran Procedures

To use a Fortran procedure, you would need to assign your mfArray to a Fortran array of the same data type, shape, and extent. You can assign the mfArray to a Fortran array using two methods - assignment and *msPointer*.

The assignment method involves memory copy. A memory block, of the same size as that occupied by the mfArray, is assigned to the new Fortran array. This means that memory resource is not optimized. The two data objects are independent from each other. In effect, changes made to the Fortran array is not reflected in the mfArray.

The *msPointer* method associates an allocatable Fortran pointer of the same data type and dimension as the mfArray, to the memory space occupied by the mfArray. This conserves memory resource. However, changes made to the memory storage through the Fortran pointer are also reflected in the mfArray. Deallocating the Fortran pointer nullifies the mfArray.

2.7.3.1 Assignment operation

To assign mfArray to a Fortran array, you would have to declare a Fortran array of the same data type, shape, and extent as the mfArray. The assignment is easily done through the assignment operator '=*'*. Example, *a = b.*

Example 2.7.3.1 below shows some valid expressions of the assignment operation.

Example 2.7.3.1 Assign an mfArray to a Fortran array

```
Program Example2_7_3_1

use fml
implicit none

type(mfArray):: a
double complex, allocatable :: COMPLEXA(:,:)
real(8), allocatable :: REALA(:,:)

! Construct mfArray a
```

```
a = mfMagic(3)
call msDisplay(a, 'a')

! Assign mfArray to Fortran array.
If (mfIsReal(a)) then
   allocate (REALA(mfSize(a,1),mfSize(a,2)))
   REALA = a
   write(*,*) 'REALA = '
   write(*,*) REALA

elseif (mfIsComplex(a)) then
   allocate (COMPLEXA(mfSize(a,1),mfSize(a,2)))
   COMPLEXA = a
   write(*,*) 'COMPLEXA = '
   write(*,*) COMPLEXA

else
   write (*,*) 'mfArray is of different data type'

endif

! Deallocate mfArray
call msFreeArgs(a)

end Program Example2_7_3_1
```

2.7.3.2 msPointer operation

Procedure *msPointer*, located in module *mod_ess*, associates the storage space of an mfArray to a Fortran pointer of the same rank and data type as the mfArray. In effect, the Fortran pointer and the mfArray share the same memory space. Thus operations performed on the Fortran variable are reflected in the content of the mfArray and vice versa. This enables you to use Fortran subscript method to access elements of the mfArray.

MATFOR handles the mfArray as the primary variable and the Fortran pointer as the secondary variable. In effect, MATFOR does not restrict you to reshape the mfArray that effectively releases the original data container. It is the programmer's responsibility to reassign the Fortran pointer after changing the shape of an mfArray or releasing its memory space. Note that deallocating the Fortran pointer will nullify the mfArray. Refer to procedure *mfEquiv* for more information about equivalency of mfArray and Fortran data object.

Procedure *msPointer* has the following general syntax.

call msPointer(<mfArray>, <pointer>)

call msPointer(<mfArray>, <pointer>, shape)

Example 2.7.3.2 below shows an example of using procedure *msPointer*.

```
Program Example2_7_3_2

use fml
implicit none

type(mfArray) :: b
real(8), pointer :: PD(:,:)
complex(8), pointer :: PZ(:,:,:)

! Construct 10-by-10 mfArray b
b = mfReshape(mfColon(1,100), (/10,10/))
call msDisplay(b, 'b')

! Assign pointer of mfArray to Fortran pointer
if(mfIsReal(b)) then
   ! PD now targets b
   call msPointer(b, PD)

   PD(10,10) = 5.0
   call msDisplay(b, 'PD(10,10) =5.0, b ')


else if(mfIsComplex(b)) then
! PZ- now targets b
   call msPointer(b, PZ)

! calls Fortran complex sine function
   write(*,*) SIN(PZ)


else
write(*,*) 'Unknown data type'

end if

call msFreeArgs(b)
```

```
end Program Example2_7_3_2
```

2.7.4 Using Fortran Arrays as Input to MATFOR Procedures

To use a Fortran array as input to a MATFOR procedure that supports only mfArray I/O, you can convert the Fortran array to an mfArray using function *mf()*, or construct a new mfArray using the assignment operator '=' or procedure *msAssign*. The above operations involve a duplication of the memory space occupied by the Fortran array. The constructed mfArray acts independently from the Fortran array. To avoid duplication of memory, you can use procedure *mfEquiv*, located in module *mod_ess*, to construct an mfArray that is equivalent to the Fortran array. The target Fortran array must be of type double precision real or complex, and of a rank of not more than 7. Procedure *mfEquiv* constructs a restricted mfArray. The restricted mfArray does not support operations that change the shape nor deallocate memory space. This avoids illegal memory manipulation of the Fortran variable should the shape or memory space of the equivalent mfArray be changed accidentally?

Procedure *mfEquiv* has the following general syntax.:

<mfArray> = mfEquiv (<Fortran Array>)

For example,

```
COMPLEX(16), ALLOCATABLE :: F(:,:)
type (mfArray) :: a

ALLOCATE(F(2,2))
a = mfEquiv(F)
```

In the above codes, procedure *mfEquiv* constructs an mfArray *a*, equivalent to array *F*. The procedure associates the memory space occupied by array *F* as the target of mfArray *a*. Effectively, this means that you can use either mfArray *a* or array *F* to control the content of the same memory space.

Warning! Be aware that caution must be taken when you program with two variables sharing the same memory space. In this case, deleting the Fortran variable *F*, would result in an invalid mfArray.

Difference between operator '=' and msAssign
You can use either operator '=' or procedure *msAssign* to construct a new mfArray from an existing Fortran array. Both operations involve a duplication of the memory space occupied by the Fortran array in the heap memory. However, when operator '=' is used, stack overflow may occur in some compilers. This is avoided by using procedure *msAssign.*

Example 2.7.4.1 shows some valid usages of procedure mfEquiv. Example 2.7.4.2 shows a dangerous manipulation of a Fortran array associated with mfArray using mfEquiv. Example 2.7.4.3 shows the effect of reshaping an mfArray constructed using mfEquiv.

Example 2.7.4.1

```
Program Example2_7_4_1

use fml
use fgl
implicit none

type(mfArray) :: a, l, u
REAL(8), DIMENSION(10,10) :: T

call RANDOM_NUMBER(T)

! mfArray a shares the same storage space as T.
a = mfEquiv(T)
call msDisplay(a, 'a')

! Perform LU decomposition of T
call msLu(mfOut(l, u), a)
call msDisplay(l, 'l', u, 'u')

! Surface plot of T is plotted
call msSurf( a)
call msShading('interp')
call msViewPause()

! Deallocate mfArray
call msFreeArgs(a, l, u)

end Program Example2_7_4_1
```

In the example above, procedure *msFreeArgs* is used at the end of the program to release mfArray *a*. Note that, in this case, procedure *msFreeArgs* nullifies mfArray *a*, but does not deallocate Fortran array *T*?

Example 2.7.4.2 Dangerous example

This example below shows a dangerous operation where the Fortran variable is deallocated. The resulting mfArray returns erroneous data.

```
Program Example2_7_4_2

use fml
use fgl
implicit none

type(mfArray)::t
REAL(8), ALLOCATABLE::A(:,:)

ALLOCATE(A(3,4))
call RANDOM_NUMBER(A)
t = mfEquiv(A)

call msDisplay(mfSvd(t), 'mfSvd(t)')  !==> OK


DEALLOCATE(A)  !== >inconsistent mfArray data
! status, t cannot be used any more.

call msDisplay(mfSvd(t), 'mfSvd(t)')
! ==> result is invalid. t's data point
! to a, but a is deallocated.

call msFreeArgs(t)

end Program Example2_7_4_2
```

Example 2.7.4.3 – Effects of Reshaping

```
Program Example2_7_4_3

use fml
implicit none

type(mfArray)::t
REAL(8),POINTER::A(:,:)

ALLOCATE(A(3,3))
```

```
call RANDOM_NUMBER(A)

t = mfEquiv(A)
call msDisplay(t, 't')

call msAssign(mfMatSub(t, 3, 2), 5)
call msDisplay(t, 't(3,2) = 5, t ')

t = mfReshape(mfColon(1,9),(/3,3/))
!== > operation proceeds as shape is not modified.
call msDisplay(t, 't = mfReshape(mfColon(1,9),(/3,3/))')

! Content of array A is also changed.
! Note write prints array data as a single
! line following column-major arrangement.
write(*,*) 'A is also changed! A ='
write(*,*) A

t=(/1,2,3/)
! ==> error occurs as the operation changes
! the shape of mfArray t.

call msFreeArgs(t)

end Program Example2_7_4_3
```

In the example above, you can see that changing values of a single element of the mfArray is a valid operation. The second operation, reshaping, does not change the shape of the mfArray (mfArray remains as 3-by-3), hence the operation proceeds, updating the contents of array *A* and mfArray *t.* The third operation changes mfArray *t* to a size 3 vector. This action changes the shape of the mfArray *t,* which was restricted to the shape of array *A,* hence an error occurs.

2.7.5 Using mfArray as Input Dummy Arguments

When you use mfArray as input dummy arguments in your subroutines and functions, it is recommended for you to use procedures *msInitArgs* and *msFreeArgs* to enclose the dummy mfArrays. For example,

```
function mffun(x,y,z) result(out)
    type(mfArray) ::x , y , out
    type(mfArray) ::a,b
    INTEGER(4) :: Z(:,:)

    call msInitArgs(x,y)
    a=x-y
```

```
    b=a+3
    …
    call msFreeArgs(x,y)
    call msReturnArray(out)
end function myfun
```

More information about procedures *msInitArgs* and *msFreeArgs* is provided in Section 2.7.5.1.

2.7.5.1 More about msInitArgs and msFreeArgs

MATFOR mfArray is a dynamic data type. To prevent from memory-leakage, MATFOR assigns a temporary flag to all temporarily constructed mfArrays and frees their associated memory spaces automatically. Examples of temporary mfArrays include MATFOR function outputs that are used as input to MATFOR procedures or user-defined subroutines accepting mfArray as input argument.

To prevent accidental deletion of mfArray used as input to your subroutines, MATFOR provides you with the procedures *msInitArgs* and *msFreeArgs.* Procedure *msInitArgs* is used at the beginning of your subroutine while procedure *msFreeArgs* is used at the end of your subroutine.

Procedure *msInitArgs* specifies mfArrays for use in your subroutine. This prevents MATFOR from automatically deleting temporary mfArrays that might be used as entries into your subroutine. ?

Procedure *msFreeArgs* complements the function of *msInitArgs* by restoring the status of temporary mfArrays back to temporary, so that they are removed automatically by MATFOR.

2.7.6 Using mfArray as Output Dummy Arguments in Functions

When you use mfArray as function output argument, it is recommended to use procedure *msReturnArray* to clean any temporary mfArrays. For example,

```
function mffun(ra,rb) result(out)
type(mfArray) ::x，y，out
```

```
REAL(8) :: RA(:),RB(:)

x=RA
...

call msReturnArray(out)
end function myfun
```

3 Linear Algebra

Matrix operation is used in many engineering and scientific problems. For the purpose of numerical computation, these problems are normally represented in the form of linear algebra using matrices. MATFOR provides users with a set of linear algebra procedures located in module *mod_matfun* to solve matrix operation intuitively and efficiently.

Three kinds of matrix operations are often encountered in real problems, namely matrix inverse, eigenvalues and eigenvectors, and least square approximation. Algorithms used for solving these problems depend heavily on the characteristics of the matrices. For efficient performance, different algorithms must be employed for each type of matrix.

MATFOR *mod_matfun* has built in mechanisms for handling the details of algorithm selection in matrix operations. Intuitive interfaces are provided so that users do not have to know the details of the algorithm used. We shall go through three examples in Sections 3.1, 3.2, and 3.3 to familiarize with the *mod_matfun* procedures.

3.1 Matrix Inverse

Matrix inverse is often used in mathematical applications. The following is an example employing MATFOR matrix inverse procedure, *mfInv*.

Example 3.1 Matrix Inverse

The objective in this example is to determine the relationship between the value of export from Hong Kong, to the Gross National Product and Per Capita Import of each of its fourteen overseas markets.

Using the relationship determined, we attempt to compute the value of export from Hong Kong when a target overseas market has a Gross National Product equal to 367.56 (million millions U.S. dollars) and a Per Capita equal to 1230.08 (U.S. dollars).

The data for computation is listed in Table 3.1 below.

Table 3.1 Relationship between export value from Hong Kong and GNP + Per Capita Import of overseas market

Overseas markets (i)	Export value of Hong Kong (Yi：million HK$)	Gross National Product (Xi 1:million millions US$)	Per Capita Import (Xi 2:US$)
1. America	6825	1298	437.26
2. Canada	512	119.8	1283.48
3. Germany	1902	344.28	1128.33
4. French	146	235.56	600.58
5. England	2814	163.79	783.15
6. Brazil	37	76.72	65.26
7. Panama	52	17.81	441.26
8. Venezuela	56	30.66	242.33
9. Indonesia	187	15.92	23.98
10. Japan	1065	345.08	371.98
11. Malaysia	107	6.7	324.4
12. South Africa	173	28	262.11
13. Australia	771	75	1058.16
14. New Zealand	192	12.47	1072.27

In this example, we shall use the regression model below to determine the relationship.

Regression Model:

$$Y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + E_i$$

where,

i =1,2,…,14

Y_i : Value of export from Hong Kong to i-th market.

X_{i1} : Gross National Product of i-th market.

X_{i2} : Per Capital Import of i-th market.

E_i : i-th error

β_i : regression constant i= 0,1,2

The regression model can be expressed in matrix form:

$$Y = X\beta + E$$

where,

$Y = [Y_1 \;\; Y_2 \ldots Y_{14}]$ (Y is a 14x1 row vector)

$$X = \begin{bmatrix} 1 & X_{1,1} & X_{1,2} \\ 1 & X_{2,1} & X_{2,2} \\ \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots \\ 1 & X_{14,1} & X_{14,2} \end{bmatrix}$$ (X is a 14 x 3 matrix)

$\beta = [\beta_1 \ \beta_2 \ \beta_3]^t$ (β is a 1x3 column vector)

$E = [E_1 E_2 \ldots E_{14}]^t$ (E is a 14x 1 row vector)

Using least square approximation, we obtain an estimate for β and Y:

$\tilde{\beta} = (X'X)^{-1}X'Y;$

$\tilde{Y} = X\tilde{\beta} = X(X'X)^{-1}X'Y;$

where $\tilde{\beta}$ is an estimate of β and $\tilde{Y}$ is an estimate of Y.

To solve $\tilde{\beta}$ and $\tilde{Y}$, the inverse of $(X'X)$ must be determined. The following code uses MATFOR procedures to determine the inverse of $(X'X)$.

```
Program Example3_1

use fml
use fgl
implicit none

type(mfArray) :: y, x1, x2, beta, a, ey, x

real(8) :: t1, t2


x1=.t.(/1298d0, 119.8d0, 344.28d0, 235.56d0, &


163.79d0, 76.72d0,17.81d0, 30.66d0,15.92d0, &


345.08d0, 6.70d0, 28d0,75d0,12.47d0/)
```

```
x2=.t.(/437.26d0, 1283.48d0, 1128.33d0, 600.58d0,&

783.15d0, 65.26d0, 441.26d0, 242.33d0, 23.98d0,&

371.98d0, 324.4d0, 262.11d0, 1058.16d0, 1072.27d0/)


! input two values (Gross National product and

! Per Capita Import)

write(*,*) "Input two numbers"
read(*,*) t1,t2

a = .t.mfOnes(1,14)

x = a .hc. x1 .hc. x2



y=.t.(/6825d0, 512d0, 1902d0, 146d0, 2814d0, &

37d0, 52d0, 56d0, 184d0, 1065d0, 107d0, 173d0, &

771d0, 192d0/)


! beta = mfMul(mfInv(mfMul(.t.x, x)), mfMul(.t.x, y))
beta = mfLDiv(x,y)
ey = mfMatSub(beta ,1 ,1) + t1*mfMatSub(beta ,2 ,1) +
t2*mfMatSub(beta ,3 ,1)

call msDisplay(beta,'beta',ey,'ey')
call msGDisplay(x1,'x1',x2,'x2',x,'x',a,'a',y,'y',beta,'beta',ey,'ey')
call msViewPause()

call msFreeArgs(y, x1, x2, beta, a, ey, x)
```

```
end Program Example3_1
```

Compile and run the code.

Use Gross National Product, t1 = 367.59 and Per Capta Import, t2 = 1230.08

$\tilde{\beta} = \begin{bmatrix} -177.9518 \\ 5.094 \\ 0.3975 \end{bmatrix}$; and $\tilde{Y}$ = 2183.6 (million Hong Kong dollars)

3.2 Application of Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors are important in many areas of science and engineering. It is often applied in solving differential equations, and finding physical characteristics of structures. In MATFOR, you can use procedure *mfEig* to determine eigenvalues and eigenvectors of a matrix. The example below applies procedure *mfEig* in solving a differential equation.

Example 3.2 Solving a differential equation

In this example, we shall use procedure *mfEig* to find the solution to a set of differential equations.

Consider the following differential system:

1. $\frac{dx_1}{dt} = x_1 - x_2 - x_3;$
2. $\frac{dx_2}{dt} = -x_1 + x_2 - x_3;$
3. $\frac{dx_3}{dt} = -x_1 - x_2 + x_3;$

where x_1, x_2, and x_3 are functions of t.

The system can be rewritten in matrix-vector format as:

$$\frac{dx}{dt} = AX;$$

where $A = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix}$, $X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$, $\frac{dx}{dt} = \begin{bmatrix} \frac{dx_1}{dt} \\ \frac{dx_2}{dt} \\ \frac{dx_3}{dt} \end{bmatrix}$

Then, the solution to the differential system is:

$$X = K_1 * e^{\lambda_1 t} * v_1 + K_2 * e^{\lambda_2 t} * v_2 + K_3 * e^{\lambda_3 t} * v_3$$

where K_1, K_2, and K_3 are constants.

λ_1, λ_2, and λ_3 are the eigenvalues of A.

v_1, v_2, and v_3 are eigenvectors corresponding to λ_1, λ_2, and λ_3.

The eigenvalues and eigenvectors of the system can be obtained by using MATFOR procedure as illustrated in the code below:

```
Program Example3_2

use fml
implicit none

type (mfArray) :: a, p, d

a = (/1,-1,-1/) .vc. (/-1,1,-1/) .vc. (/-1,-1,1/)

! Compute eigenvalues and eigenvectors
```

```
call msEig(mfOut(p, d),a)

! Display results
call msDisplay(a,'a',d,'d',p,'p')

! Deallocate mfArrays
call msFreeArgs(a, p, d)

end Program Example3_2
```

Compile and run the program. The eigenvector *p* and the eigenvalue *d* are computed to be as follow:

$$p = \begin{bmatrix} 0.5774 & -0.6112 & -0.5414 \\ 0.5774 & 0.7745 & -0.2586 \\ 0.5774 & -0.1633 & 0.8000 \end{bmatrix} \quad d = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

Using the eigenvector and eigenvalue, the solution to the differential system is:

$$X = K_1 * e^{-t} * \begin{bmatrix} 0.5774 \\ 0.5774 \\ 0.5774 \end{bmatrix} + K_2 * e^{2t} * \begin{bmatrix} -0.6112 \\ 0.7745 \\ -0.1633 \end{bmatrix} + K_3 * e^{2t} * \begin{bmatrix} -0.5414 \\ -0.2586 \\ 0.8000 \end{bmatrix}$$

3.3 Least Square Operations

Least square method is often used in the determination of optimal solutions such as obtaining an optimal polynomial equation approximating a collection of data. The example below presents an application of MATFOR procedures in Least Square operations.

Example 3.3 Determining the optimal binomial

There are four data points with coordinates (2, 1), (4, 3), (5, 5), and (8, 12) as shown in Figure 3.3.

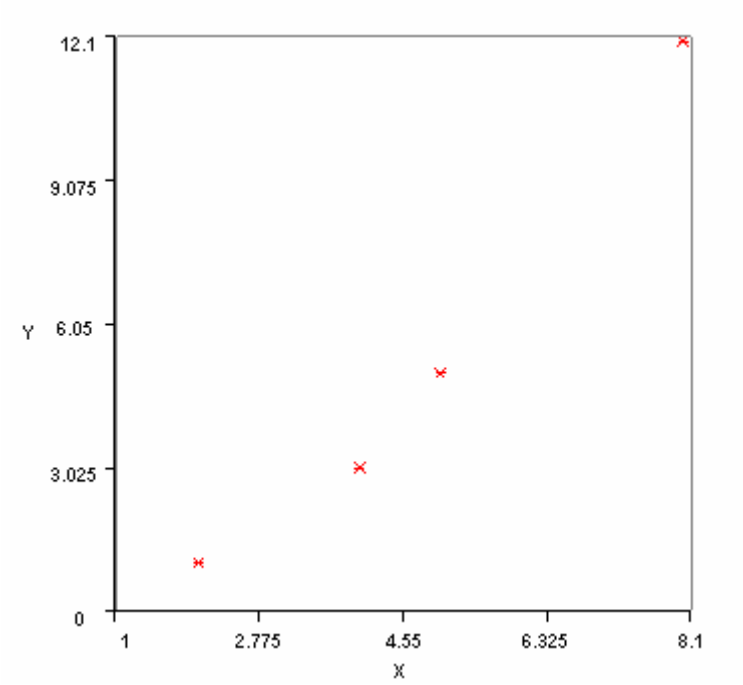


Figure 3.3.1 Plot of four data points (2,1), (4,3), (5,5) and (8,12).

The points can be approximated using a binomial equation $f(x) = r_0 + r_1x + r_2x^2$, where r_0, r_1, and r_2 are the constants of the polynomial. From the four data points, four systems equations are formed as follows.

$$\begin{cases} 1 = r_0 + 2r_1 + 4r_2 \\ 3 = r_0 + 4r_1 + 16r_2 \\ 5 = r_0 + 5r_1 + 25r_2 \\ 12 = r_0 + 8r_1 + 64r_2 \end{cases}$$

The system equation can be represented in matrix-vector form as $Ar = b$ where,

$$A=\begin{bmatrix}1 & 2 & 4\\ 1 & 4 & 16\\ 1 & 5 & 25\\ 1 & 8 & 64\end{bmatrix},\ b=\begin{bmatrix}1\\ 3\\ 5\\ 12\end{bmatrix},\ r=\begin{bmatrix}r_0\\ r_1\\ r_2\end{bmatrix}$$

As $Ar = b$ has no linear solution, an approximate solution to $Ar \approx b$ needs to be determined.

Using least square method, vector *r* can be approximated by,

$$r=(A^T A)^{-1}A^T b,$$

Instead of solving *r* by computing the inverse of $(A^T A)$, which is an expensive operation, you can use MATFOR matrix left division *mfLDiv* to solve *r*. The codes below use procedures in *mod_matfun* to obtain an approximation to the coefficient vector *r*.

```
Program Example3_3

use mod_ops
use mod_elmat
use mod_matfun
use fgl
implicit none
type(mfArray) :: x, y, r,A,x1

! Obtain a solution to vector r in linear equation Ar=y.

! Initialize matrix A.
x = (/2,4,5,8/)
A=mfOnes(4,1).hc.(.t.x).hc.(.t.x**2)

! Initialize vector y to contain the four data points
y=.t.(/1,3,5,12/)

! Plot the four data points using red *
```

```
call msPlot(x,y,'r*')

! Compute the solution to vector r.

! r = mfMul(mfMul(mfInv(mfMul(.t.A, A)),.t.A), y)
r = mfLDiv(A,y)

call msDisplay(r, 'r')


! Using the result from r, compute the resulting binomial equation
! y =r0 + r1*x+ r2*x**2 over the range x =[1:8].
! y is computed using y = mfMul(A, r)

x = mfColon(1,8)
A = mfOnes(8,1).hc.(.t.x).hc.(.t.(x**2))
y = mfMul(A, r)

! Stop the Graphics Viewer from erasing the first graph
call msHold('on')

! Plot the binomial equation using a blue line
call msPlot(x,y,'b-')

! Set the axis range
call msAxis(1d0,8.1d0,0d0,12.1d)

! Pause program to view the resulting graphics
call msViewpause()

! Deallocate mfArrays
call msFreeArgs(x, y, r,A)

end Program Example3_3
```

Using the program, we obtain ***r*** as:

$$r = \begin{bmatrix} 0.2067 \\ 0.0100 \\ 0.1833 \end{bmatrix}$$

Thus, the optimal binomial equation to fit the data by least square method is:

$$f(x) = 0.207 + 0.010x + 0.183x^2$$

Table 3.3 shows the comparison of the raw data of the solution to the binomial approximation using a_i, $i=1,2,3,4$. Figure 3.3.2 shows the resulting binomial curve and the four data points.

Table 3.3 (comparison of the data with the polynomial)

i	a_i	b_i	$f(a_i)$
1	2	1	0.959
2	4	3	3.17
3	5	5	4.83
4	8	12	12.0

a_i and b_i, are the data, and $f(a_i)$ is the solution to the binomial equation using data a_i, i=1,2,3,4.

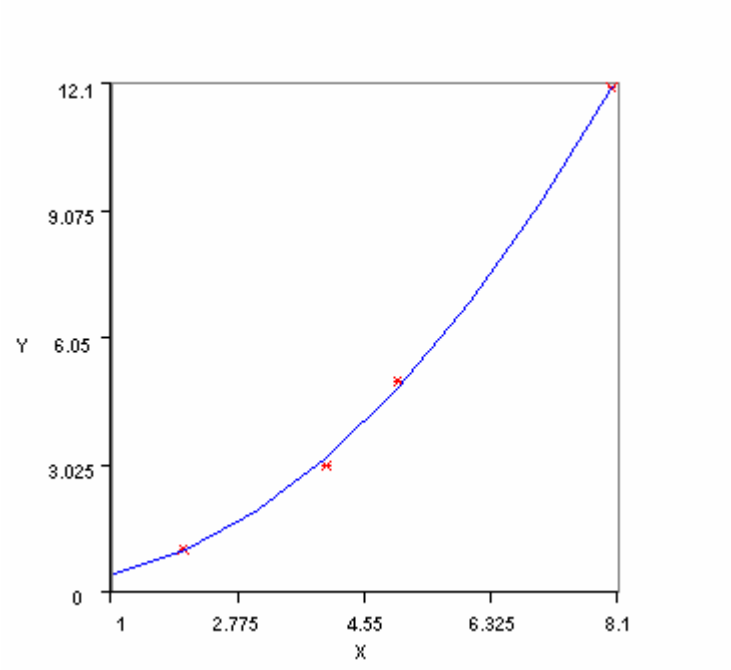


Figure 3.3.1 The polynomial equation and the four data points.

4 Visualization Basics

The basic idea of visualization is to transform your computational data into a format that is more communicative and instructive. MATFOR Graphics Library contains a set of high-level visualization procedures for data visualization, animation, graphical debugging, and presentation. They are designed to be intuitive and require minimal programming.

In this chapter, we will introduce some fundamental capabilities MATFOR provides and explores what is possible. A few examples are provided to guide you to the steps of using MATFOR Visualization Routines.

4.1 Plotting Your Data

This section outlines the general steps for creating a graph using MATFOR graphical procedures. We'll begin by plotting a linear graph.

Step 1. Use MATFOR modules in your program by adding the statement *use <module>* under your program name. You'll need to use the module *fgl* when using any of the visualization routines. For example,

```
Program Example4_1
use fgl
use mod_ops
use mod_elfun
```

Step 2. Construct and initialize the mfArray for plotting. For example,

```
type (mfArray) :: x, y
integer :: i
x = (/(i,i=-10,10)/)
y = x**2
```

Step 3. Initiate a Graphics Viewer for plotting to by using procedure *msFigure.* All Figures are numbered automatically, depending on the sequence of creation. For example,

```
call msFigure(1)
```

Step 4. Create a graph using one of the graph creation procedures such as *msPlot.* For example,

```
call msPlot(x,y)
```

Step 5. You can touch up the graph by using procedures such as *msShading, msAxis,* and *msBackGroundColor,* or annotate the graph by adding axis labels and a title. By default, x-axis is labeled x, y-axis labeled y, and z-axis labeled z.

For example,

```
call msTitle('y = x**2')
```

Step 6. Pause the program execution to view the graph by using:

```
call msViewPause()
```

Step 7. Deallocate the mfArrays and end the program.

```
call msFreeArgs(x,y)

end Program Example4_2
```

Step 8. Compile and run the program to view the graph as shown in Figure 4.2.

Step 9. When you have finished viewing your graph, press the Continue button on the top right corner of the Graphics Viewer to continue program execution.

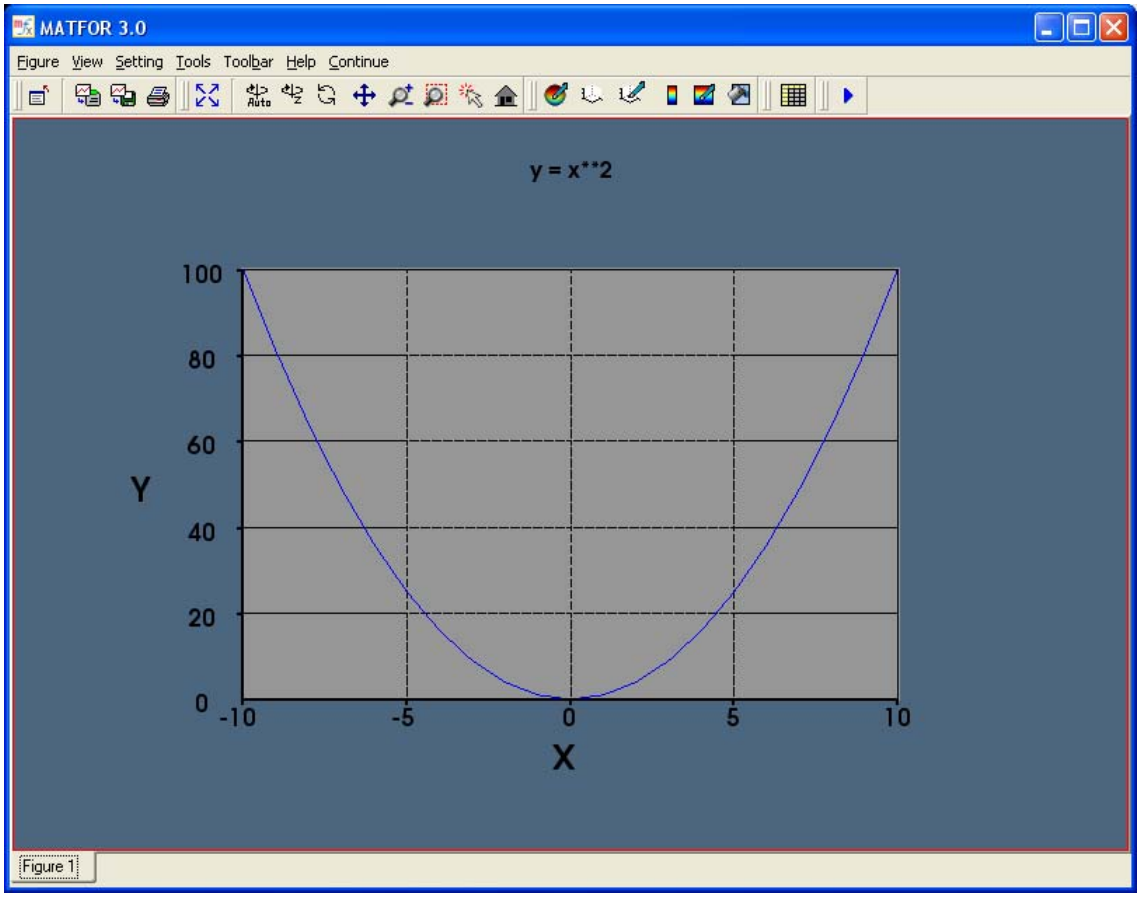


Figure 4.1 Plotting of y = cos(x)

Example 4.1 Below is a summarization of the above codes.

Example 4.2 Steps to visualization

```
Program Example4_1

use fgl
use mod_ops
use mod_elfun
implicit none

type (mfArray) :: x, y

x = mfColon(-10,10)
```

```
y = x**2

! Specify a new Graphics Viewer
call msFigure(1)

! Plot a 2-D x-y plot
call msPlot(x,y)

! Add a Title to the graph
call msTitle('y = x**2')

! Pause Program to view Graphics
call msViewPause()

! Deallocate mfArray
call msFreeArgs(x,y)

end Program Example4_1
```

4.2 MATFOR Graphics Viewer

When you use graph-creating procedures such as *msPlot,* the created graphs will be displayed on MATFOR Graphics Viewer as shown in Figure 4.1.

Graphics Viewer is composed of six major components, namely the window frame, figure windows, subplots, menu, toolbar, and a couple of dialog box editors. These components collaborate to display the graphics objects you created on your monitor screen and provide you with graphics formatting functions.

In this section, we shall briefly describe each component and its usage.

4.2.1 Window Frame and Figure Windows

The properties of the Window Frame can be set through procedures *mfWindowCaption*, *mfWindowSize,* and *mfWindowPos*.

A Graphics Viewer can contain a number of figure windows. Each figure window is attached to a window tab that shows the ID and name of that particular figure window. It enables you to navigate through the figure windows more easily.

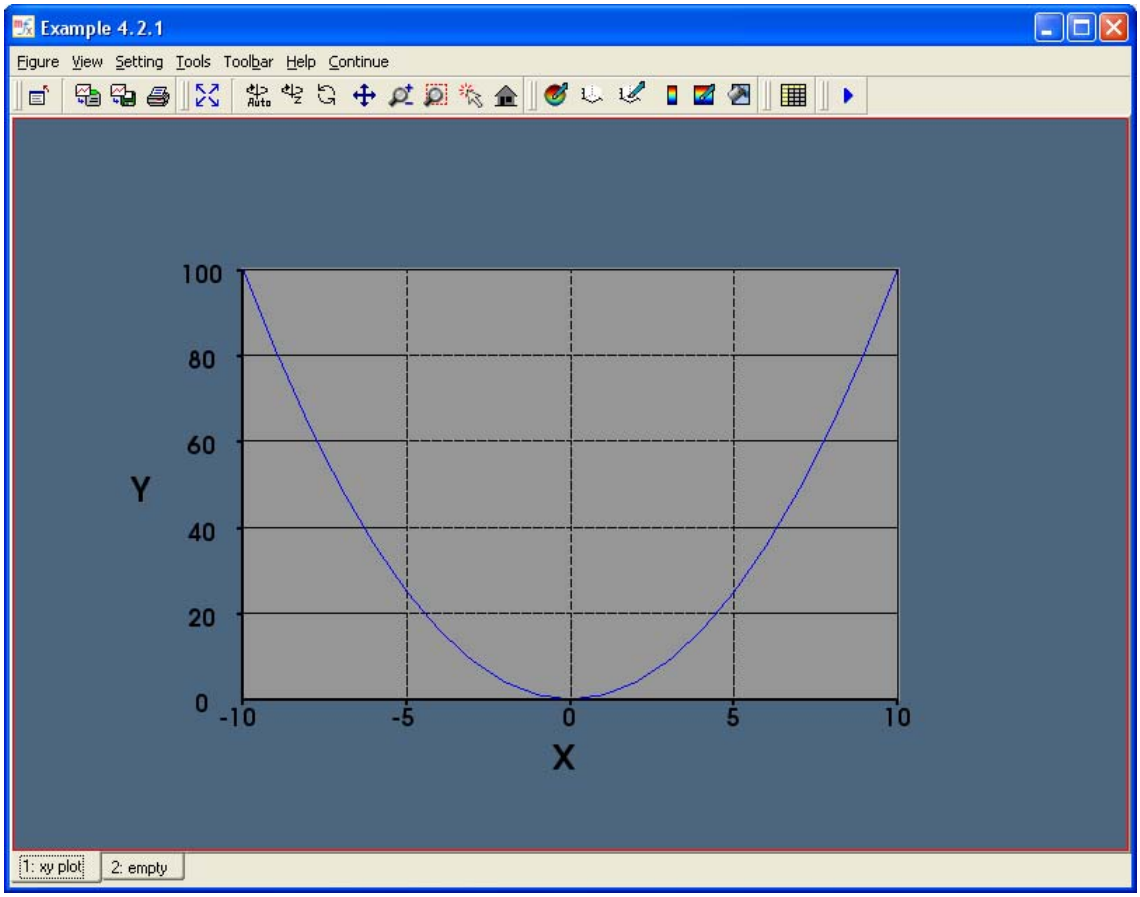


Figure 4.2.1 Windows and figure windows

4.2.2 Subplots

Each figure window can be further divided into multiple subplots by using the procedure *msSubplot*. The procedure has the following syntax:

```
call msSubplot(m, n, p)
```

Where *m* is the number of rows, *n* is the number of columns and *p* specifies the current sub-plot space number. The procedure divides the plot space of a figure window into m-by-n rectangular sub-plot spaces. Each sub-plot space is numbered row-wise, so that a sub-plot space at position (1,2) is numbered p =2 and (2,2) is numbered p=4.

Example 4.2.2 Using msSubplot

In the following, we shall create a new figure window and plot the two graphs using a 1-by-2 convention.

Create a new figure window with the ID 1.

```
call msFigure(1)
```

Divide the plot space into 1-by-2 subplot spaces and plot the first pair of graphs on subplot space 1.

```
call msSubplot(1,2,1)
h = mfPlot(x, y1, 'b')
call msHold('on')
h = mfPlot(x, y2, 'ro')
call msGSet(h, 'symbol_scale', 25)
call msCamZoom(0.8d0)
call msCamPan(20, 0)
```

Plot the third of graph defined by *x, y3* on sub-plot space 2. The axis will be automatically scaled by MATFOR to fit the third graph.

```
call msSubplot(1,2,2)
h = mfPlot(x, y3, 'gx')
call msGSet(h, 'symbol_scale', 25)
call msCamZoom(0.8d0)
call msCamPan(40, 0)
```

Pause the program to view the graph.

```
call msViewPause()
```

Compile and execute the program.

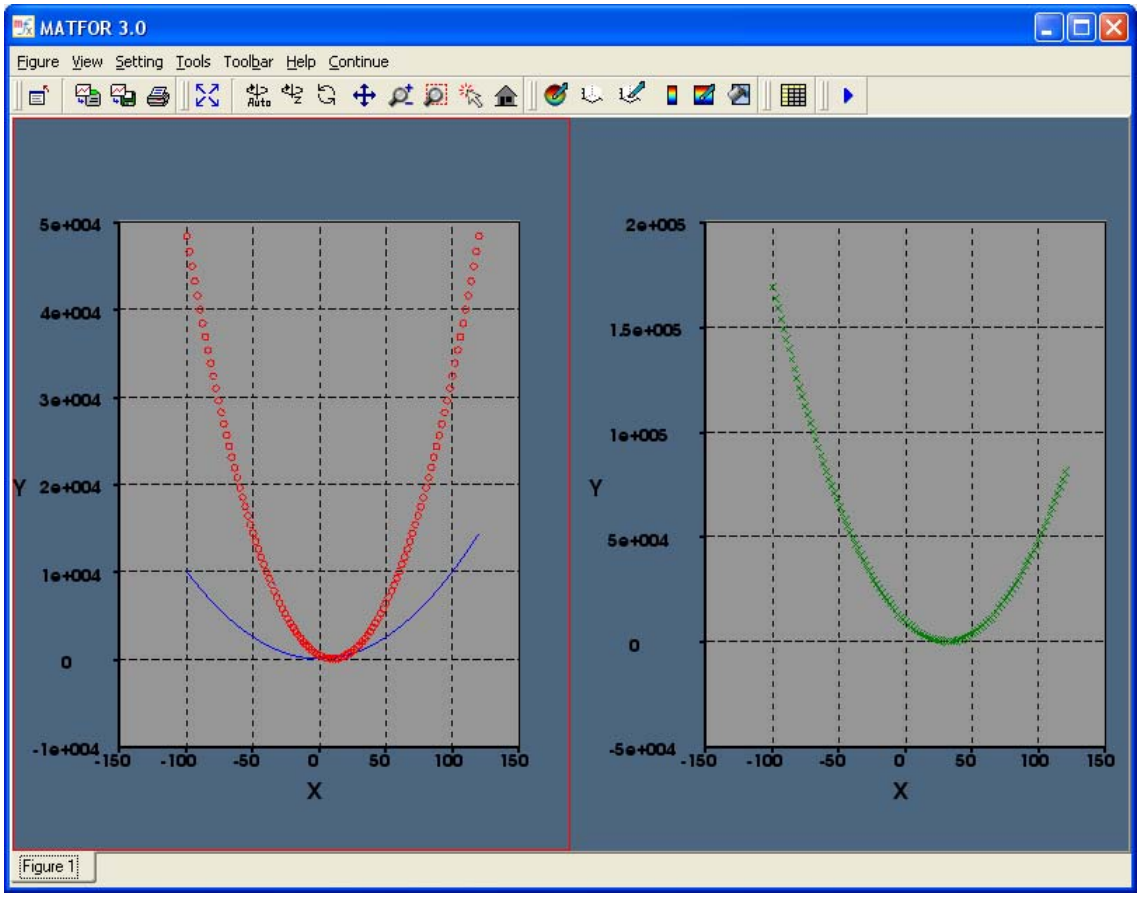


Figure 4.2.2 Two subplots in the same figure window

4.2.3 Menu and Toolbar

The most appearing feature of Graphics Viewer might be the flexibility of manipulations on graphics objects. All of the graphics object editing carried out by the procedures can be manipulated directly using the menu and toolbar functions!

With these functions, you can perform axis adjustment, material shading, colormap setting, and many other things after the program is run and the graphics objects are shown in the plot space.

Just play with the menu and toolbar functions to get a feel of this amazingly easy-to-use feature.

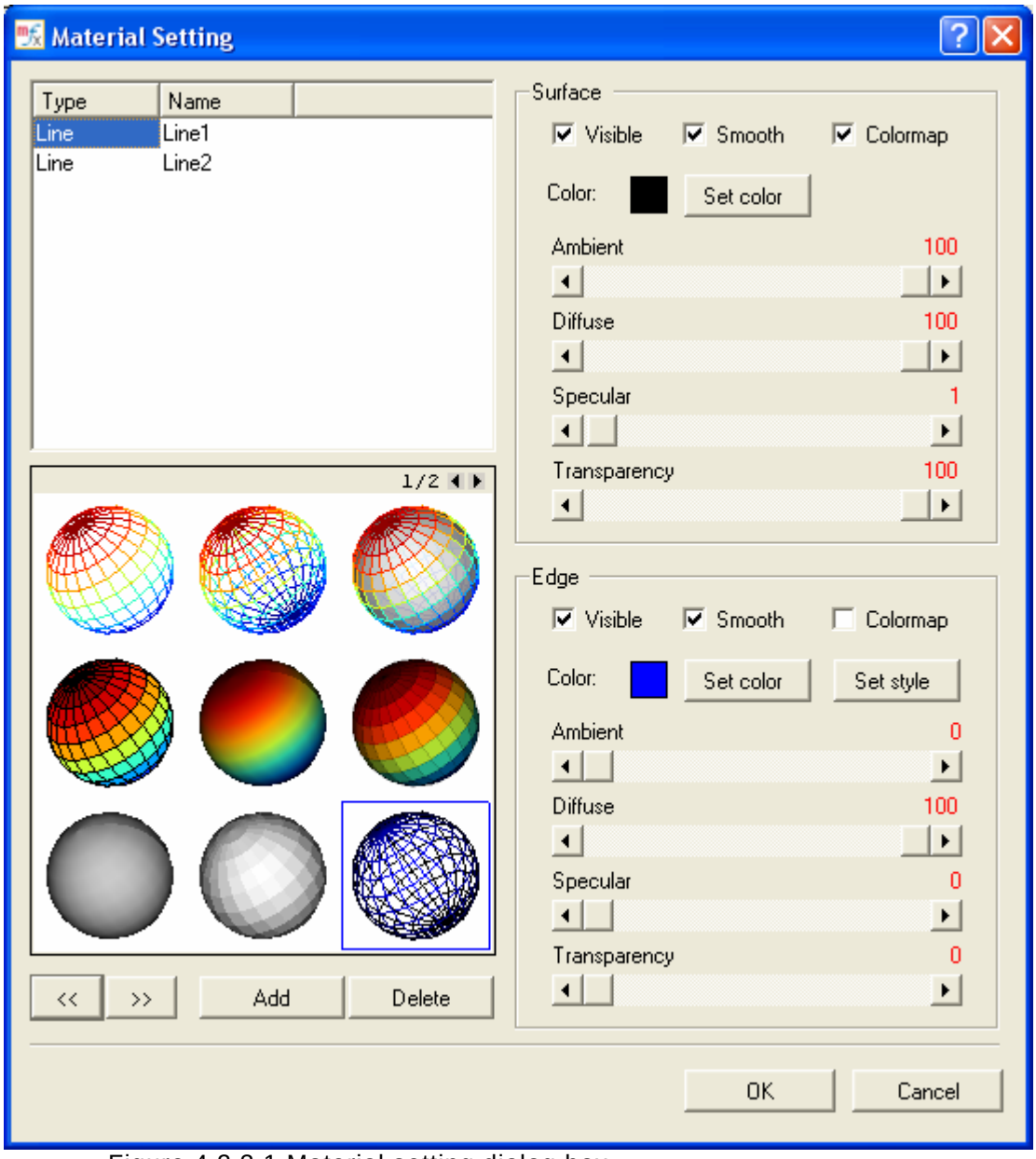


Figure 4.2.3.1 Material setting dialog box

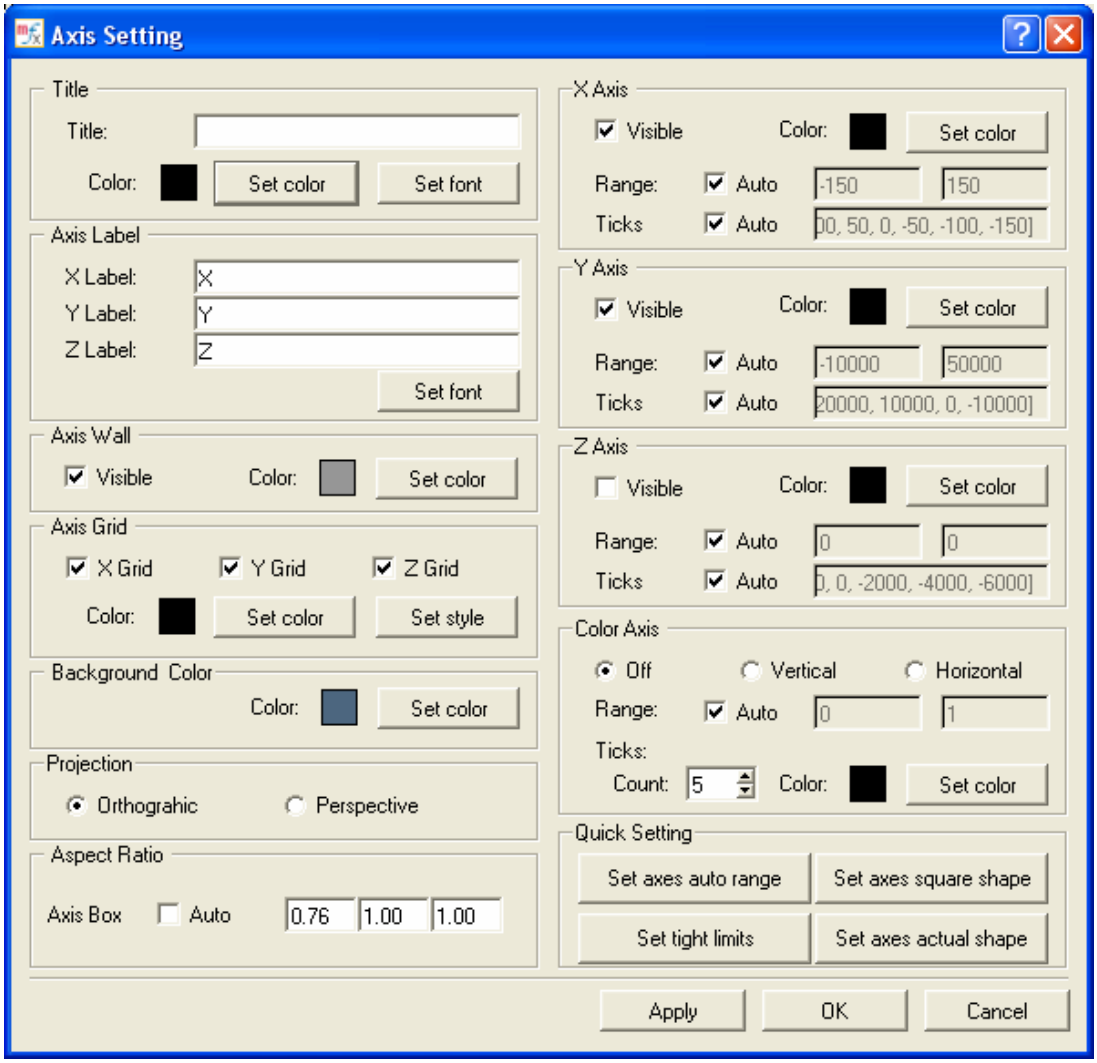


Figure 4.2.3.2 Axis setting dialog box

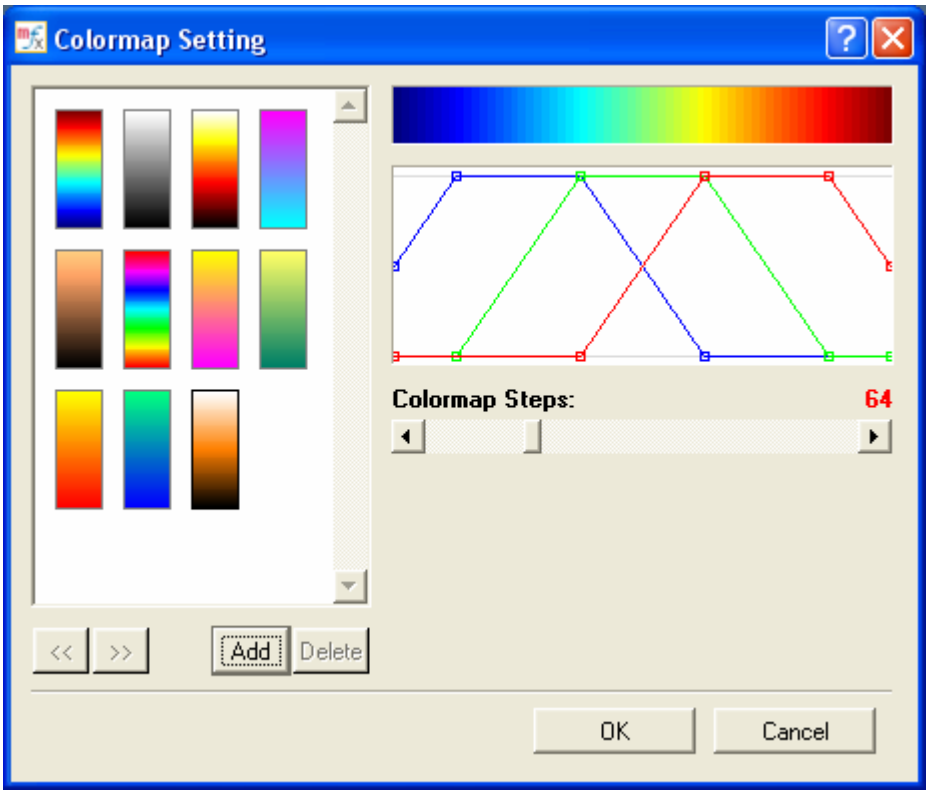


Figure 4.2.3.3 Colormap setting dialog box

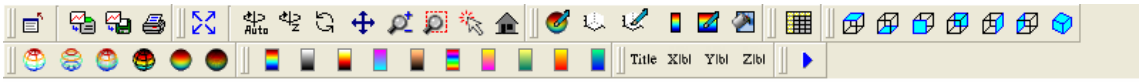


Figure 4.2.3.4 Toolbar

4.3 Creating 3-D Models

In MATFOR, there are two methods to obtain data for plotting. One is to generate the data from algorithm or mathematical expression as shown in example 4.1. The other way is to read a data file into the system.

This section goes through examples that use the two methods to plot three-dimensional graphs. One uses two-dimensional matrices to construct a surface plot; whereas the other one reads data files into the system to draw a dolphin object.

4.3.1 Generating the Data

First, we shall being to generate the data by using *msMeshgrid*, which is a useful procedure to transform the domain specified by two vectors into two-dimensional matrices. The resulting matrices are useful for evaluating functions of two variables and

for plotting surfaces. The matrices are composed of repeating rows and columns of the two vectors. The syntax of the procedure is:

```
call msMeshgrid(mfOut(matrix X,matrix Y), vector x, vector y)
```

As an example, we shall create four matrices *x, y, indxi,* and *indxj* using procedure *msMeshgrid.* Matrices *x* and *y* will be used for plotting the graph, while *indxi,* and *indxj* will be used to evaluate function *z.* The resulting mfArrays will be used in the examples for plotting lines and surfaces in three-dimensional space in the sections that follow.

Example 4.3.1 msMeshgrid- function of two variables

Start a program named Example4_3_1, use modules *fgl* and *fml,* and declare the variables *x, y, z, indxi*, and *indxj* as mfArrays. .

```
Program Example4_3_1

use fgl
use fml

implicit none
type(mfArray) :: m, x, y, z
```

Create two 30-by-30 matrices *x, y* using procedure *msMeshgrid.*

```
m = mfLinspace(-3, 3, 30)
call msMeshGrid(mfOut(x, y), m)
```

Procedure *mfLinspace* is used to construct a linearly spaced vector as input vector. It has the syntax *mfLinspace(lowerbound, upperbound, intervals).*

Function *mfOut* specifies *x* and *y* as output mfArrays.

Calculate *z* from *x* and *y.*

```
z = mfSin(x) * mfCos(y) / ( x*(x-0.5d0) + (y+0.5d0)*y + 1)
```

Using the data created above, we shall draw a surface graph in three-dimensional space using the procedure *msSurf*.

```
call msSurf(x, y, z)
call msViewPause()
```

Compile and run the program.

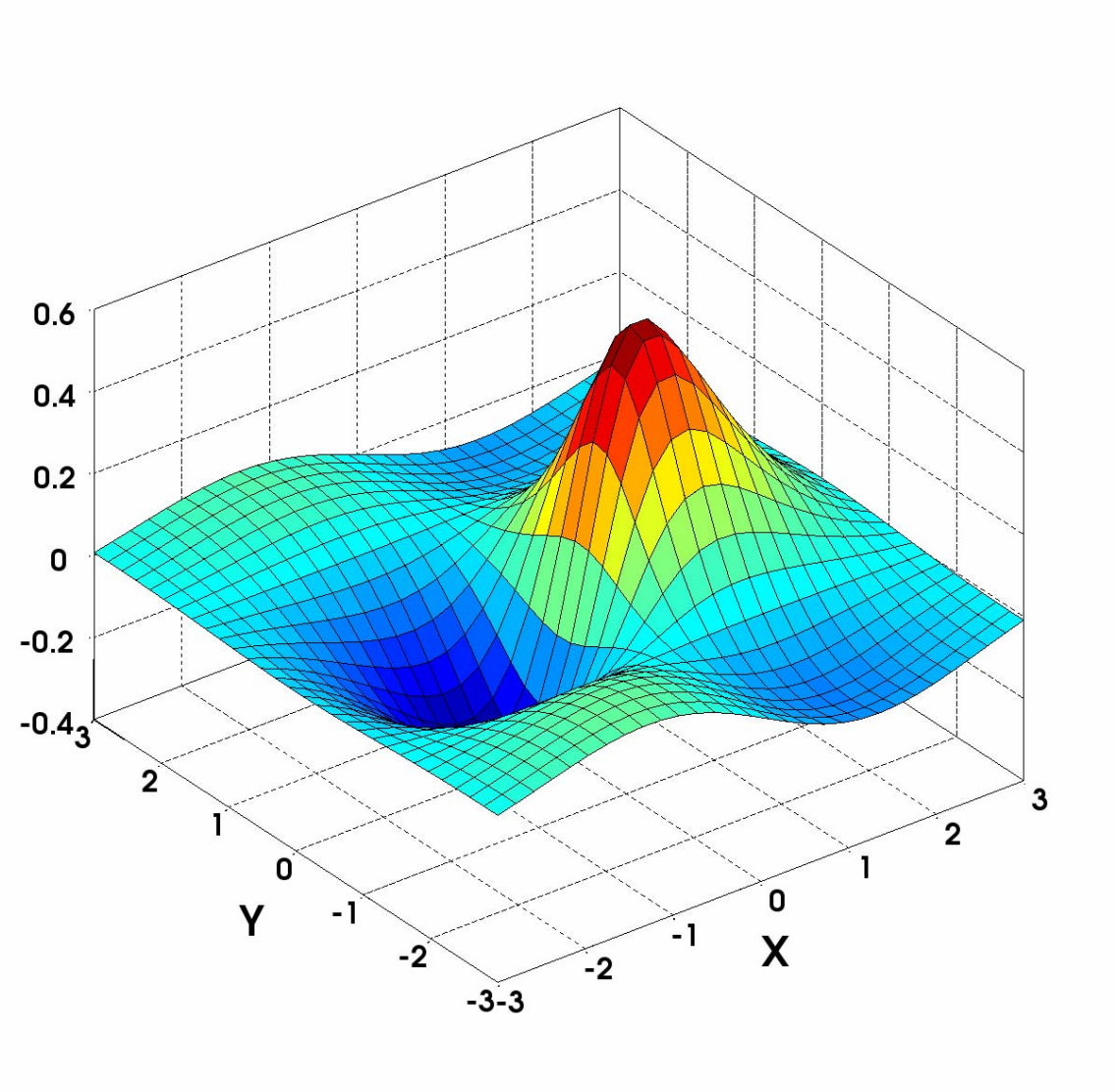


Figure 4.3.1 Surface graph in three-dimensional space

4.3.2 Loading data (mfb, ascii)

Here, we'll demonstrate an example that reads in the data of a dolphin module from ASCII data files and plots it in three-dimensional space.

Example 4.3.1 mfLoadAscii- loading ASCII data files

Start a program named Example4_3_2, use modules *fgl* and *fml*, and declare the variables *xyz* and *tri* as mfArrays.

```
Program Example4_3_2

use fml
use fgl

implicit none
type (mfArray) :: xyz, tri
```

Loads the data from the ASCII files dolphin_tri.txt and dolphin_xyz.txt using procedure *mfLoadAscii*. The example data files can be found in <MATFOR>\examples\for_ug\data\dolphin_tri.txt and <MATFOR>\examples\for_ug\data\dolphin_xyz.txt. The data is loaded into the mfArrays *xyz* and *tri*.

```
xyz = mfLoadAscii('.\data\dolphin_xyz.txt')
tri = mfLoadAscii('.\data\dolphin_tri.txt')
```

Next, construct the polygons defined by the face matrix *tri* and the corresponding vertex matrix *xyz* using procedure *mfTriSurf*. You may refer to Section 5.5 for more details on plotting unstructured mesh graphs.

```
call msTriSurf(tri, xyz))
```

Display the graphics object with proper axis adjustments to make it look neater.

```
call msAxis('equal')
call msAxis('off')
call msViewPause()
```

Compile and run the program.

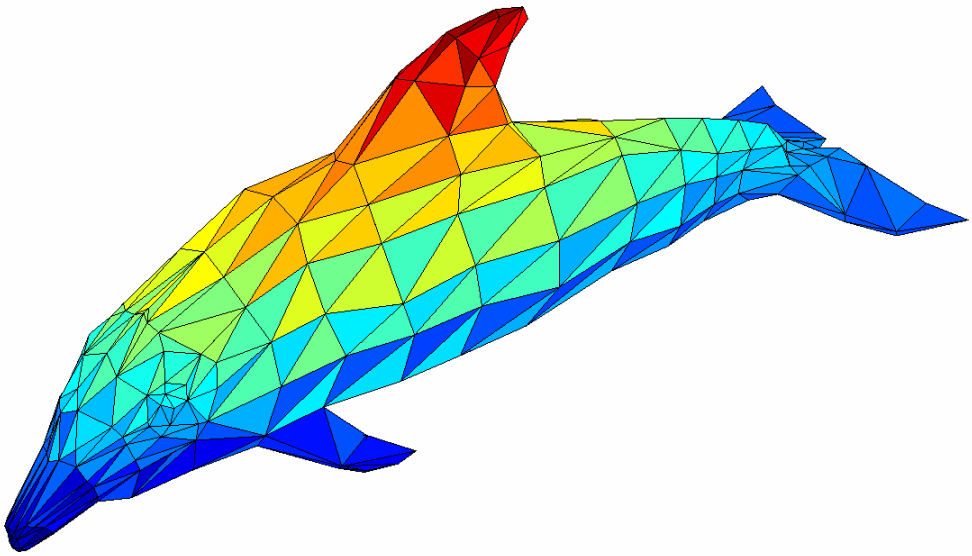


Figure 4.3.2 Dolphin object

4.4 Displaying 3-D objects

Once you have created a graphics object that is either procedurally generated or loaded from a data file. You may want to adjust the viewing of the object in order to make it more meaningful.

Using the data created in example 4.3.1, we shall go through a variety of techniques that are used when displaying the surface object.

4.4.1 Adjusting the Viewpoint.

You can set the way you view a three-dimensional graph by setting the azimuth and elevation angles of a viewpoint using procedure *msView(az, el)*.

The azimuth angle, *az*, is the angle of horizontal rotation about the z-axis, measured from the negative y-axis. The elevation angle, *el*, is the vertical angle.

Example 4.4.1 Setting the viewpoint

Simply add the statement *call msView(45, 45)* right below the surface plot procedure.

```
call msFigure('msView(45, 45)')
call msSurf(x, y, z)
call msView(45, 45)
```

Note that a three-dimensional graph has a default viewpoint at *az = -37.5* and *el = 30*. This default value can be restored by calling *msView("3")*.

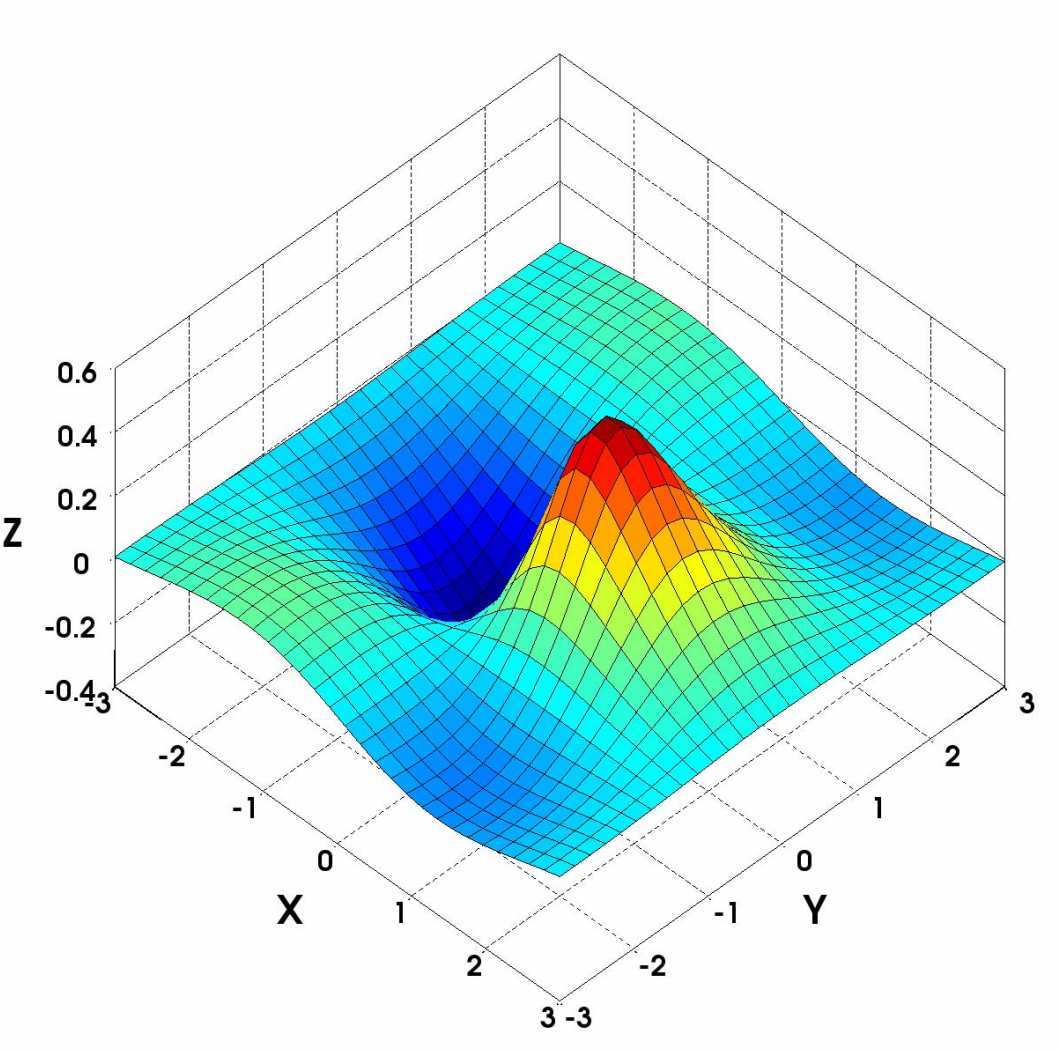


Figure 4.4.1 msView – adjust the viewpoint

4.4.2 Shifting the Objects

The camera manipulations may be handy when you want to shift the graph or rescale its size in the plot space.

Procedure *msCamPan* is used when you want to shift the graph horizontally or vertically. This is accomplished by specifying the horizontal and vertical distances of the camera displacement. Notice that the displacement of the graph is directly opposite to the displacement of the camera. For example, the graphics object will be shifted downward in the plot space when you shift the camera upward.

Example 4.4.2 Shifting in the surface object

Shift the surface object downward by moving the camera upward with the displacement of 80.

```
call msFigure('msCamPan')
call msSubplot(1, 2, 1)
call msSurf(x, y, z)
call msSubplot(1, 2, 2)
call msSurf(x, y, z)
call msCamPan(0, 80)
```

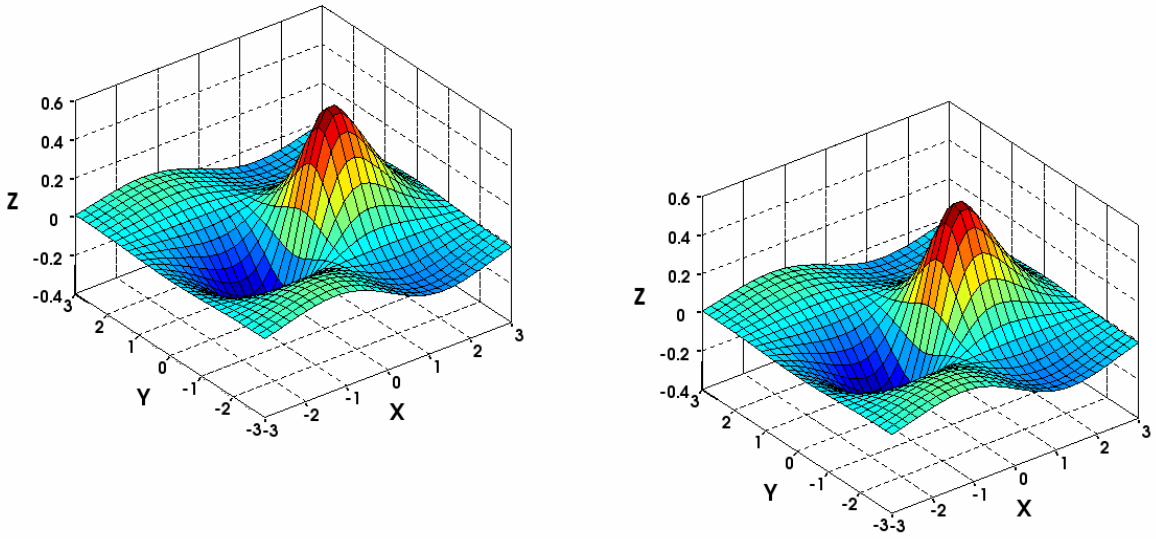


Figure 4.4.2 msCamPan – shift the camera

4.4.3 Rescaling the Objects

Procedure *msCamZoom* rescales the visual size of the graph. It takes a zoom factor as input and performs the zooming effect differently in perspective and parallel modes.

In perspective mode, it decreases the view angle by the zoom factor. In parallel mode, it decreases the parallel scale by the zoom factor.

Example 4.4.3 Zooming in the surface object

We shall zoom in the surface object in parallel mode with a zooming factor of 1.5.

```
call msFigure('msCamZoom')
```

```
call msSubplot(1, 2, 1)
call msSurf(x, y, z)
call msSubplot(1, 2, 2)
call msSurf(x, y, z)
call msCamZoom(1.5d0)
```

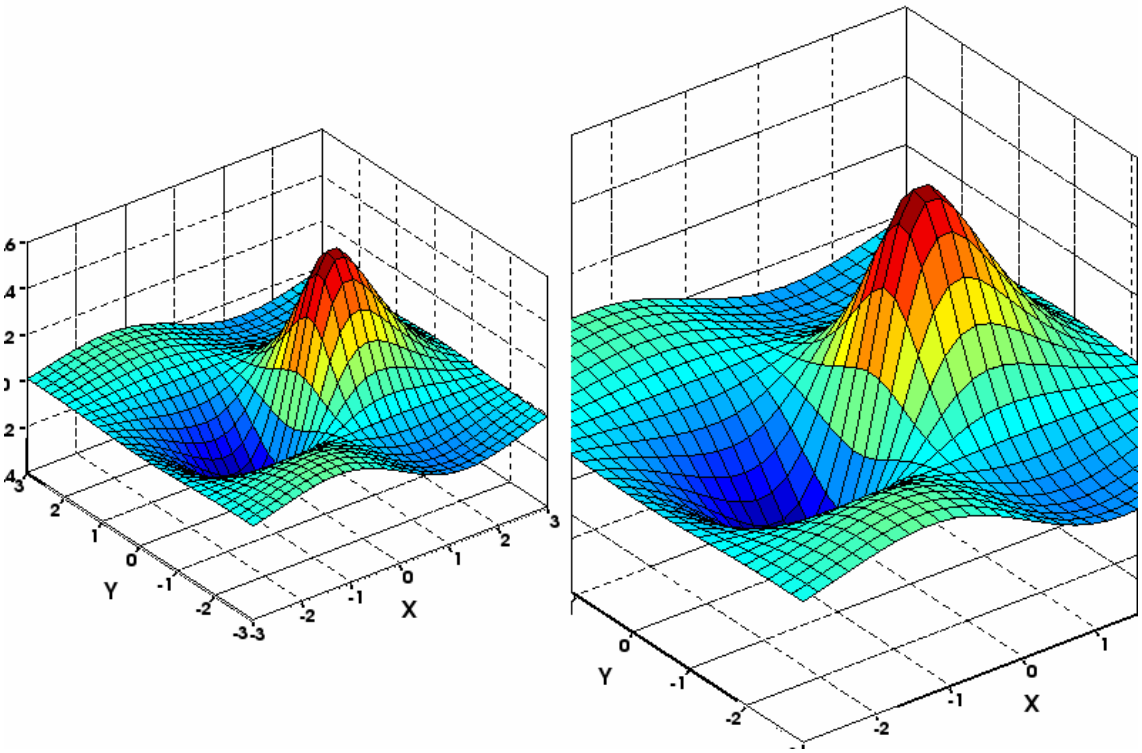


Figure 4.4.3 msCamZoom – rescale the object

4.4.4 Changing the Displaying Mode

The camera projection mode can be either perspective or orthographic (e.g. parallel). It can be done easily with procedure *msCamProj(mode)* that takes the input argument *mode*.

Example 4.4.4 Changing to the perspective displaying mode

The example code is as follows.

```
call msFigure('perspective')
call msSurf(x, y, z)
call msCamProj('perspective')
```

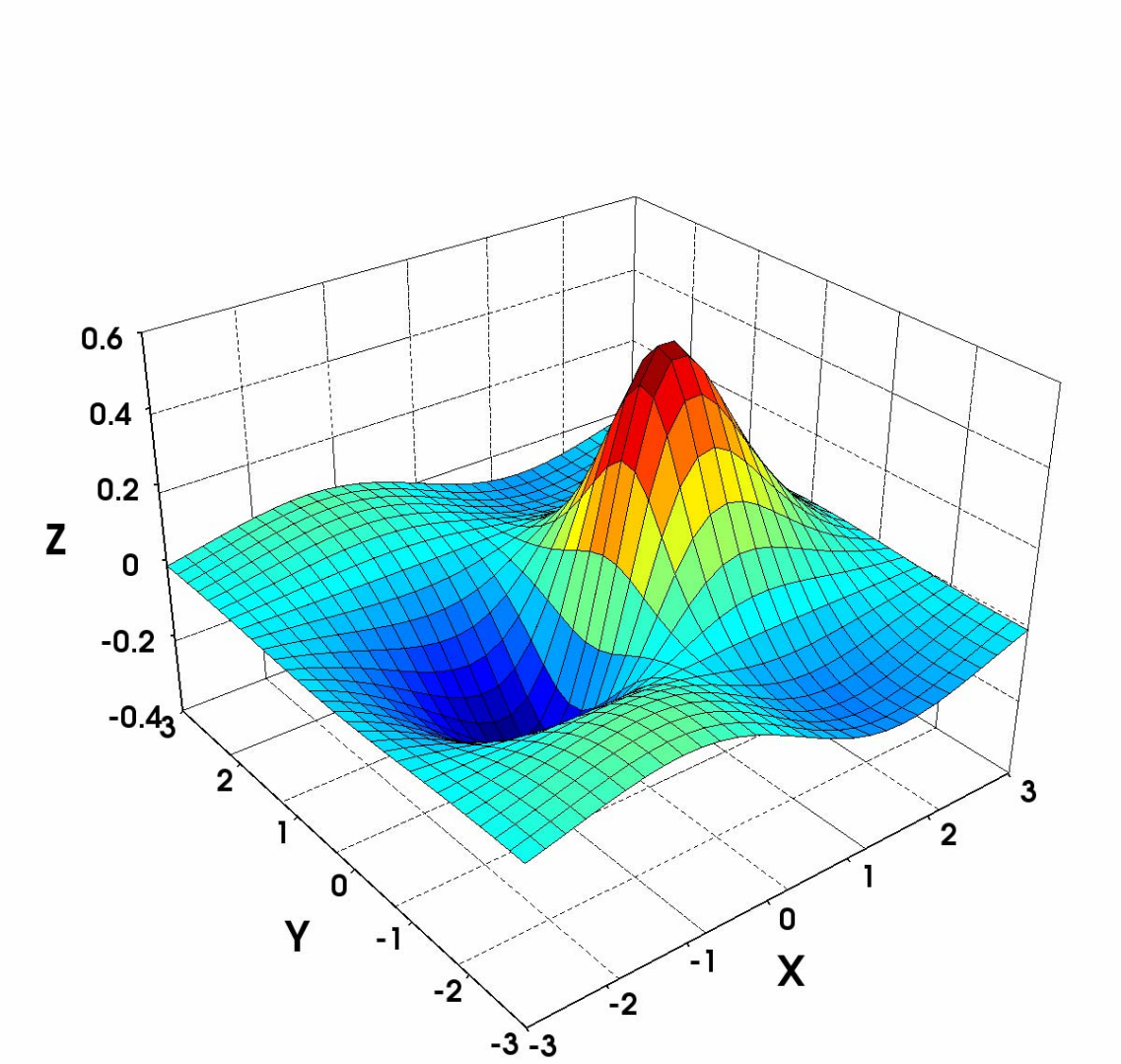


Figure 4.4.4 msCamProj - change the displaying mode

4.4.5 Setting the Axis Object

The axis object is an aggregation of the axis itself, the axis wall and the axis grid. The axis wall is the positive side of the three axis planes, which look like the three joined sides inside a box.

With procedures *msAxis*, *msAxisWall,* and *msAxisGrid*, you set the range of the axes, number of ticks on the axes, displaying grid lines, color of the axis box, etc.

Example 4.4.5 Adjusting the axis object

Using the procedures mentioned above to reset the axis ticks to be displayed, reset the color of the axis wall to white and change the grid line pattern.

```
call msFigure('Axis')
call msSurf(x, y, z)
call msAxis(mf('xaxis_ticks'), mf((/3.0d0, 1.5d0, 0.0d0, -1.5d0, -
3.0d0/)))
call msAxisWall(mf('color'), mf((/1, 1, 1/)))
call msAxisGrid('pattern', 'dashed')
```

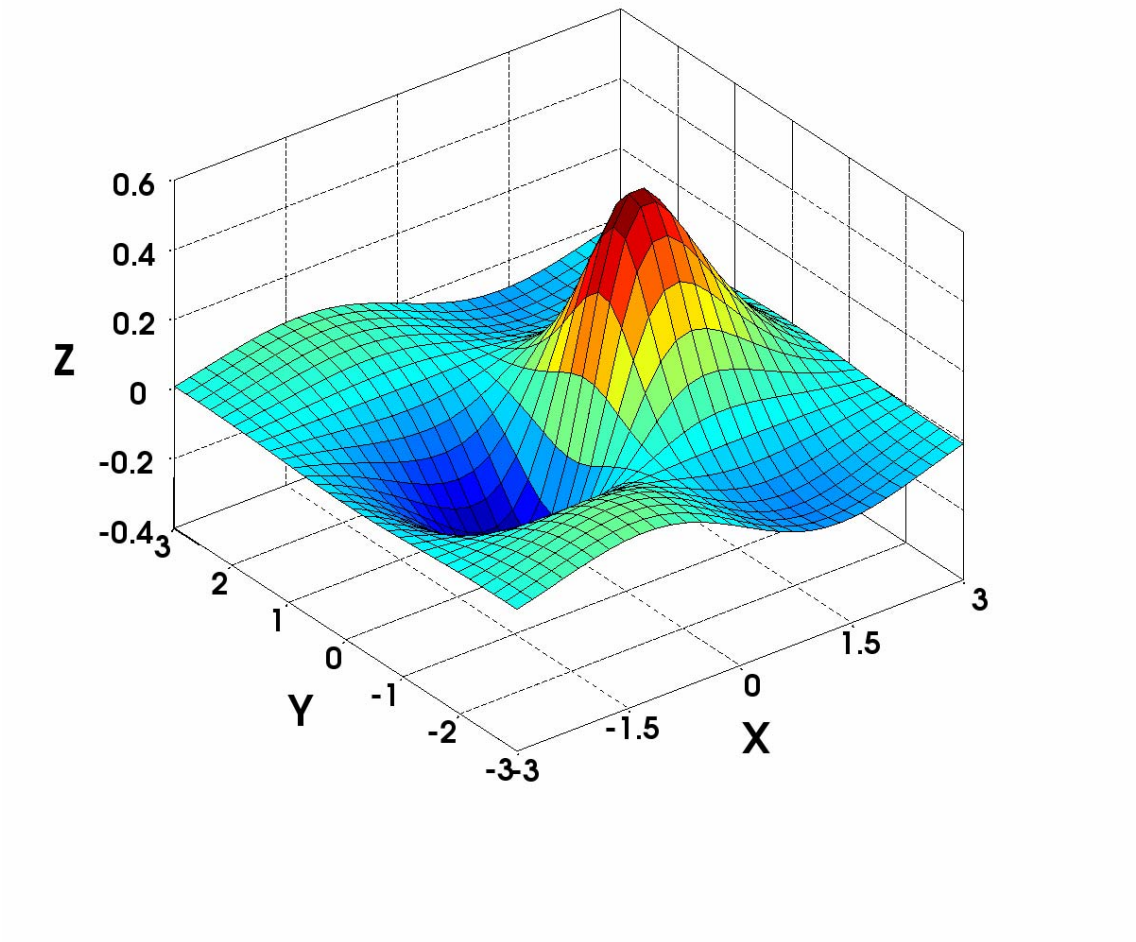


Figure 4.4.5 Set the axis object properties

4.5 Colormap, Shading and Texture

In this section, we shall go through the examples of choosing preset colormaps, displaying colorbar, shading the object surface material, and mapping texture on the surface of the object.

4.5.1 Adjusting Colormap

A surface object can be rendered with different types of colormaps. MATFOR provides a variety of predefined colormaps, such as jet, gray, hot, cool, cooper, hsv, etc.

In case where none of the predefined colormaps meets your need, you can also define custom colormaps through the Colormap Setting dialog box.

Using the data constructed in example 4.3.1, we shall illustrate how to render the surface with different predefined colormaps and define a colormap by using the Colormap Editor.

Example 4.5.1.1 Using predefined colormaps

Specify the colormap types jet, hsv, cool, and gray for drawing the surface object. We shall lay them out in subplot form in one figure window to illustrate the visual effect each of them produces.

```
!  Use colormap 'jet'
call msSubplot(2, 2, 1)
call msTitle('jet')
h = mfSurf(x, y, z)
call msColormap('jet')
call msCamZoom(1.5d0)

!  Use colormap 'hsv'
call msSubplot(2, 2, 2)
call msTitle('hsv')
h = mfSurf(x, y, z)
call msColormap('hsv')
call msCamZoom(1.5d0)

!  Use colormap 'cool'
call msSubplot(2, 2, 3)
call msTitle('cool')
```

```
h = mfSurf(x, y, z)
call msColormap('cool')
call msCamZoom(1.5d0)

!  Use colormap 'gray'
call msSubplot(2, 2, 4)
call msTitle('gray')
h = mfSurf(x, y, z)
call msColormap('gray')
call msCamZoom(1.5d0)
```

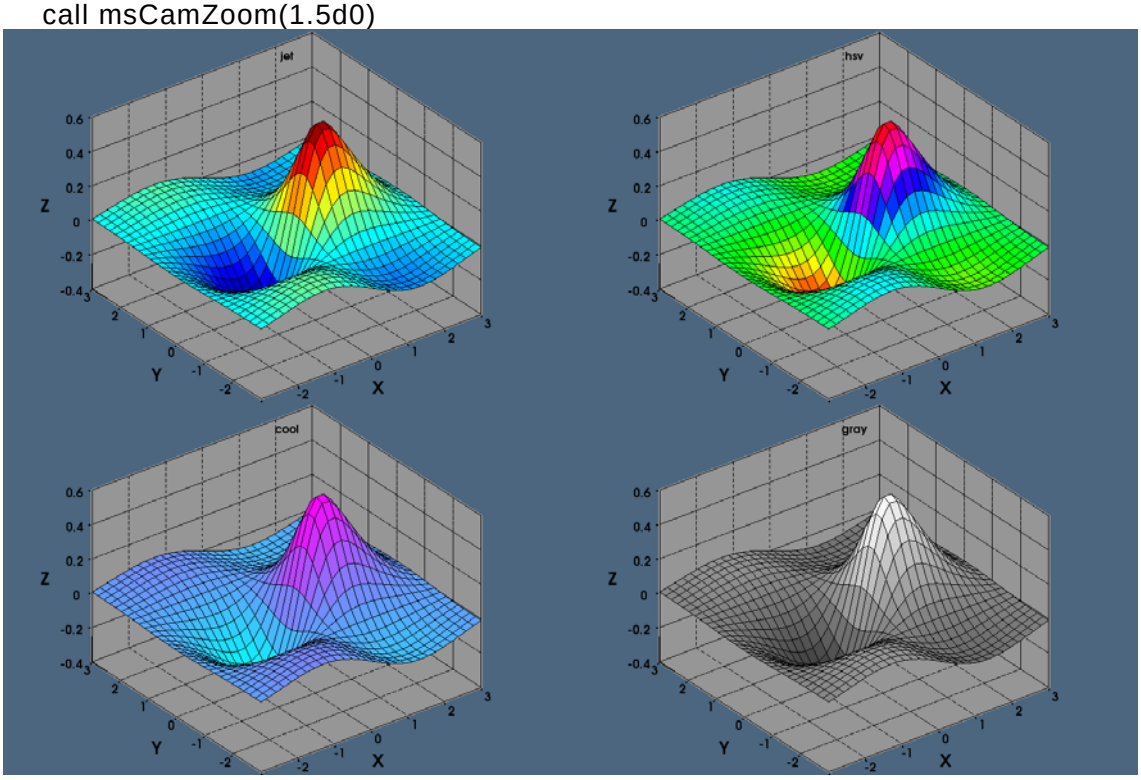


Figure 4.5.1.1 Different types of predefined colormaps

You can also manually define the colormaps by using the Colormap Setting dialog box.

Example 4.5.1.2 Using Colormap Editor

We shall demonstrate an example showing you how to create a colormap that consists of blue color components only. This is achieved by relocating the red, green, and blue lines in the editing box.

The far left portion of the box is mapped to the area which has the lowest value and the far right portion is mapped to the area which has the greatest value.

Step 1: Start by running Example 4.5.1 and extend one of the four subplots. Open the colormap setting editor, which can be found in the toolbar or Setting Menu.

Step 2: Choose the pre-defined colormap gray to work with. You may notice that the three lines are on top of each other.

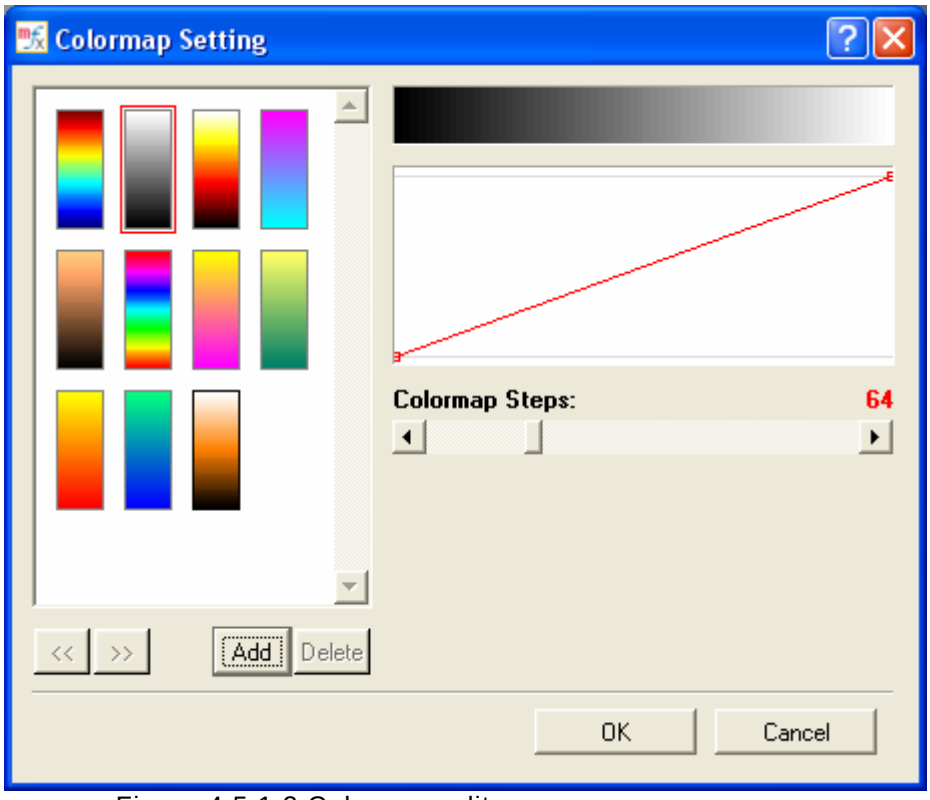


Figure 4.5.1.2 Colormap editor

Step 3: Drag the red and green lines to the bottom of the editing box and leave the blue line unchanged.

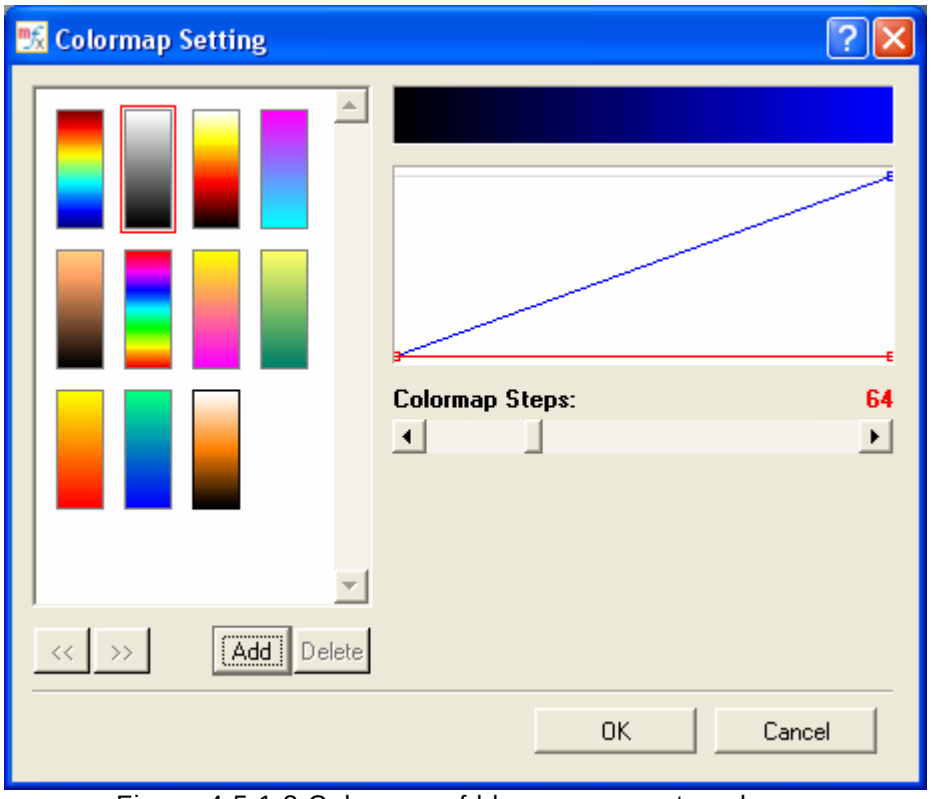


Figure 4.5.1.3 Colormap of blue components only

The resulting graph shows in figure 4.5.1.4

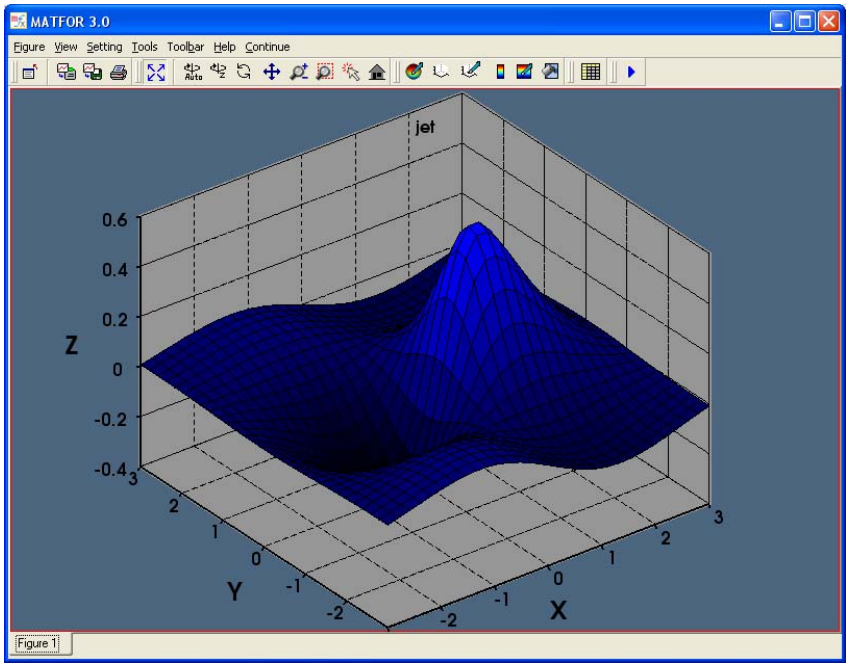


Figure 4.5.1.4 The resulting graph

4.5.2 Displaying Colorbar

The colorbar displays the current colormap and acts as a color scale showing the relationship between graphics data and color. It can be displayed in the figure window vertically or horizontally.

Vertical colorbar is displayed on the right hand side of the graph; whereas horizontal colorbar is displayed on the bottom of the figure window.

The colorbar procedure *msColorbar* has the following syntax:

call msColorbar(mode) or
call msColorbar(property, value)

Where mode specifies the location to display the colorbar. It can be *'vert'*, *'horz'*, *'on'*, or *'off'*.
The number and color of labels can be adjusted using property 'label_count' and 'label_color"

4.5.3 Shading Objects

In MATFOR, graphics objects are often composed of two major components, namely edge and surface. When it comes to shading the graphics objects, these two components are manipulated independently.

The general syntax of *msDrawMaterial* is:

```
call msDrawMaterial(handle, target, property, value)
```

Where *handle* is associated with the graphics object and *target* specifies the component that you are shading.

The shading property varies from the reflectances of the components, color of the components, to the shading mode, and visibility. For the edge component, it has two more properties, which are width and style.

Using the data constructed in Example 4.3.1, we shall demonstrate a simple example of shading the graphics object.

Example4.5.3 msDrawMaterial

First, we shall add a lighting effect on the surface to make it looks more three-dimensional. This is achieved by setting the three reflectances.

```
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))
```

At this point, the edge appears to be too obvious on the surface. We then add a fading effect on the edge by setting the value of its transparency reflectance to be 90.

```
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))
```

Compile and run the program.

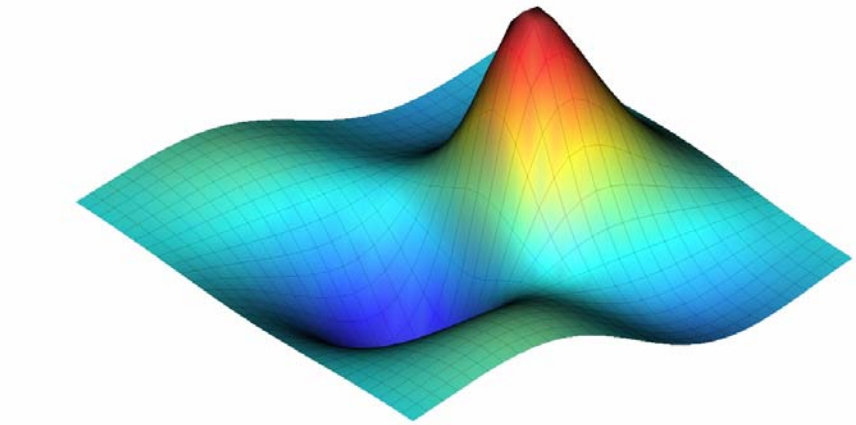


Figure 4.5.3 Shading the surface and edge components

The shading can also be done using the Material Editor that is located on the toolbar or can be located in the Setting Menu.

4.5.4 Mapping Texture

Place a texture on the graphics object by mapping the texture's coordinates to the object's coordinates.

Texture's coordinates comprise two coordinates, s-coordinate and t-coordinate that represent the horizontal and vertical coordinates of the texture respectively. Both of them are vectors of values ranging from 0 to 1. Figure 4.5.4.1 shows the relationship between the coordinate values and the positions on the texture.

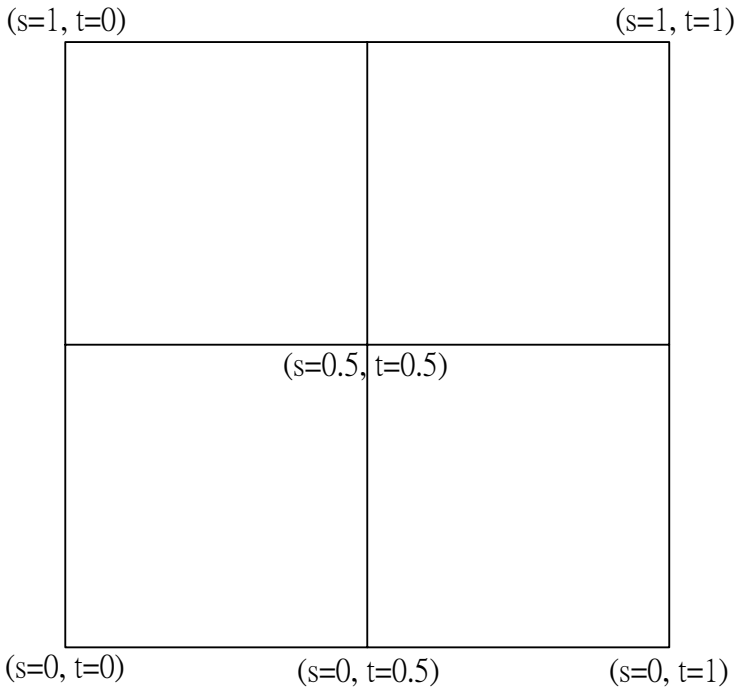


Figure 4.5.4.1 Texture coordinates

call msDrawTexture(handle, property1, value1, property2, value2, ...)

The handle is associated with the graphics object that the texture is to be placed on.

In the following example, we shall construct a tetrahedron (a polyhedron with four faces) using procedure *msTriSurf*.

First, construct the vertices of the tetrahedron using four 1-by-2 double vectors. Then we use mfArrays *x*, *y*, and *z* to store the coordinates accordingly. Each row of mfArray *tri* defines a face of the tetrahedron.

```
type (mfArray) :: tri, x, y, z, h, s, t
real(8), dimension(3) :: p1, p2, p3, p4

p1(1) = -0.25d0 * sqrt(3.0d0)
p1(2) = 0.5d0
p1(3) = 0.0d0
p2(1) = -0.25d0 * sqrt(3.0d0)
p2(2) = -0.5d0
p2(3) = 0.0d0
p3(1) = 0.25d0 * sqrt(3.0d0)
p3(2) = 0.0d0
p3(3) = 0.0d0
```

```
p4(1) = 0.0d0
p4(2) = 0.0d0
p4(3) = sqrt(6.0d0) / 3.0d0

x = (/p1(1), p2(1), p3(1), p1(1), p2(1), p4(1), p1(1), p3(1), p4(1), p2(1),
p3(1), p4(1)/)
y = (/p1(2), p2(2), p3(2), p1(2), p2(2), p4(2), p1(2), p3(2), p4(2), p2(2),
p3(2), p4(2)/)
z = (/p1(3), p2(3), p3(3), p1(3), p2(3), p4(3), p1(3), p3(3), p4(3), p2(3),
p3(3), p4(3)/)
tri = (/1, 2, 3/) .vc. &
      (/4, 5, 6/) .vc. &
      (/7, 8, 9/) .vc. &
       (/10, 11, 12/)

h = mfTriSurf(tri, x, y, z)
```

We now have to define the corresponding texture coordinates *s* and *t* using six 1-by-2 double vectors that represent six vertices on the texture individually.

```
q1(1) = 0.0d0
q1(2) = 0.0d0
q2(1) = 0.5d0
q2(2) = 0.0d0
q3(1) = 1.0d0
q3(2) = 0.0d0
q4(1) = 0.0d0
q4(2) = 0.5d0
q5(1) = 0.0d0
q5(2) = 1.0d0
q6(1) = 1.0d0
q6(2) = 1.0d0

s = (/q2(1), q4(1), q6(1), q2(1), q4(1), q1(1), q2(1), q6(1), q3(1), q4(1),
q6(1), q5(1)/)
t = (/q2(2), q4(2), q6(2), q2(2), q4(2), q1(2), q2(2), q6(2), q3(2), q4(2),
q6(2), q5(2)/)
```

Finally, map the texture file *brick.bmp* onto the tetrahedron that is associated with the handle *h*. Figure 4.5.4.2 illustrates how the texture is divided into four portions to be mapped onto the four faces of the tetrahedron. The bitmap file is located under the directory <MATFOR>\examples\for_ug\data\.

```
call msDrawTexture(h, mf('map'), mf('.\data\brick.bmp'), &
                                  mf('enable'), mf('on'), &
                                  mf('coord_s'), s, &
                                  mf('coord_t'), t )
```

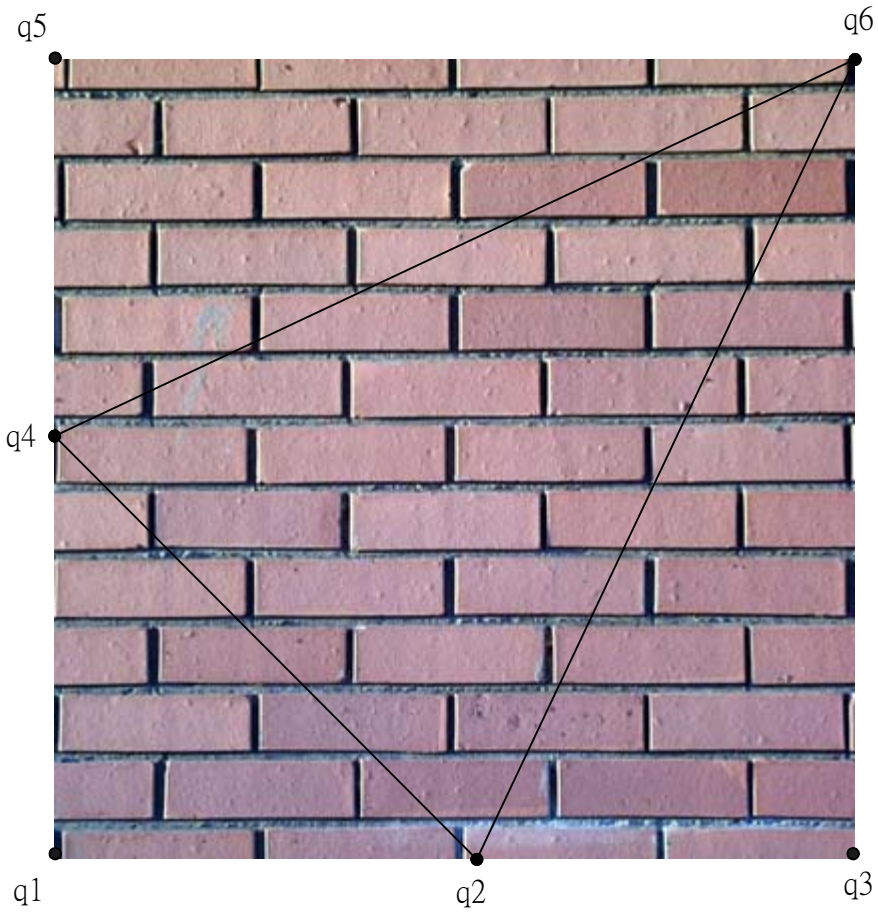


Figure 4.5.4.2 Brick.bmp

The result is in figure 4.5.4.3.

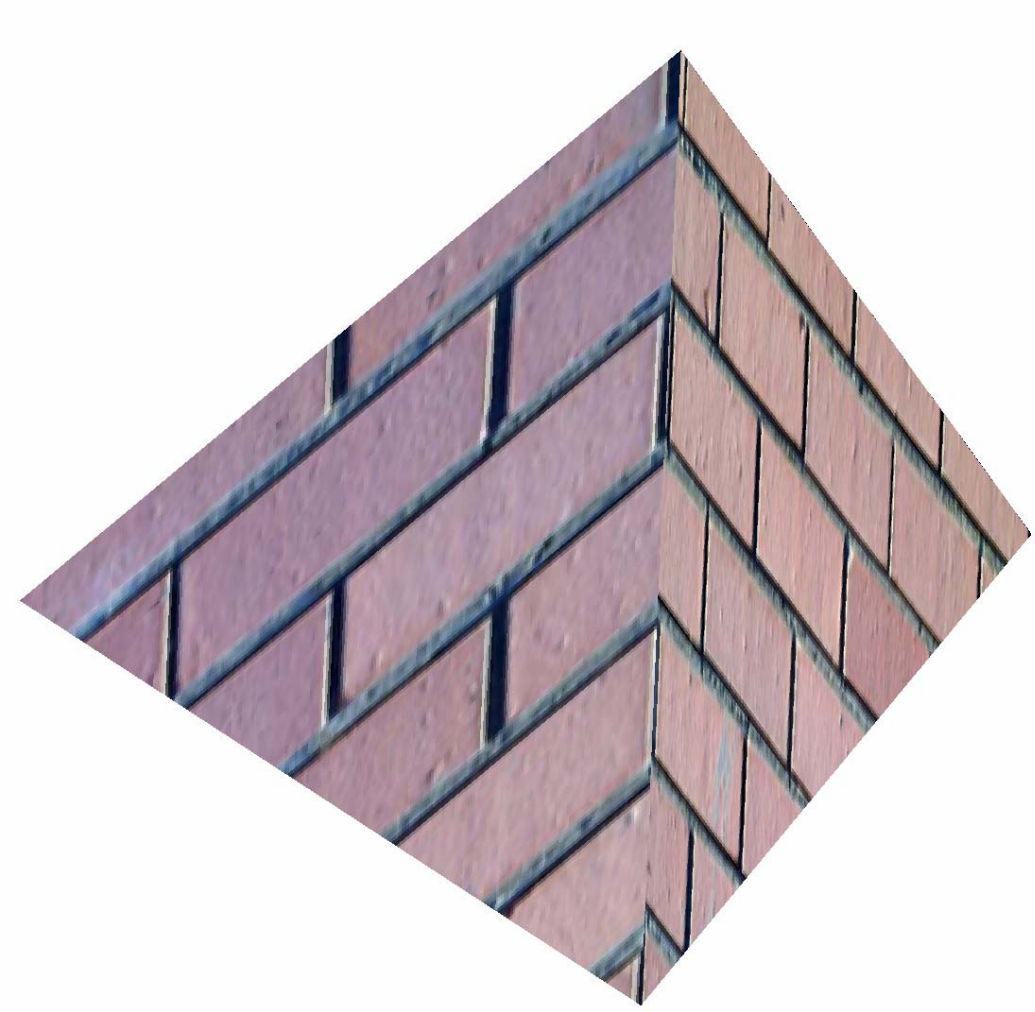


Figure 4.5.4.3 Map texture on the tetrahedron object

4.6 Annotating Your Graph

MATFOR allows you to annotate a graph with a title and axis labels. In addition to that, it also enables you to add floating text annotations on graph.

There are two ways to add the text annotation. Examples of their usages are provided in this section.

4.6.1 Setting the Title and Axis Labels

By default, the x-, y-, and z-axes are labeled as "x", "y", and "z" respectively.

You can change the labels of x-, y-, and z-axes or add a title to your graph by using the annotation procedures such as *msXLabel*, *msYLabel*, *msZLabel,* and *msTitle*, or through the Appearance Setting dialog box that can be find in the View menu.

In the following example, we shall demonstrate how to annotate a graph with a title and axis labels.

Example 4.6.1 Changing the axis labels and title

Change the labels of x-axis and y-axis and add a title to the surface graph.

```
call msXLabel('indxi')
call msYlabel('indxj')
call msTitle('Graph of z = mfSin(x) * mfCos(y) / ( x*(x-0.5d0) +
(y+0.5d0)*y + 1)')
```

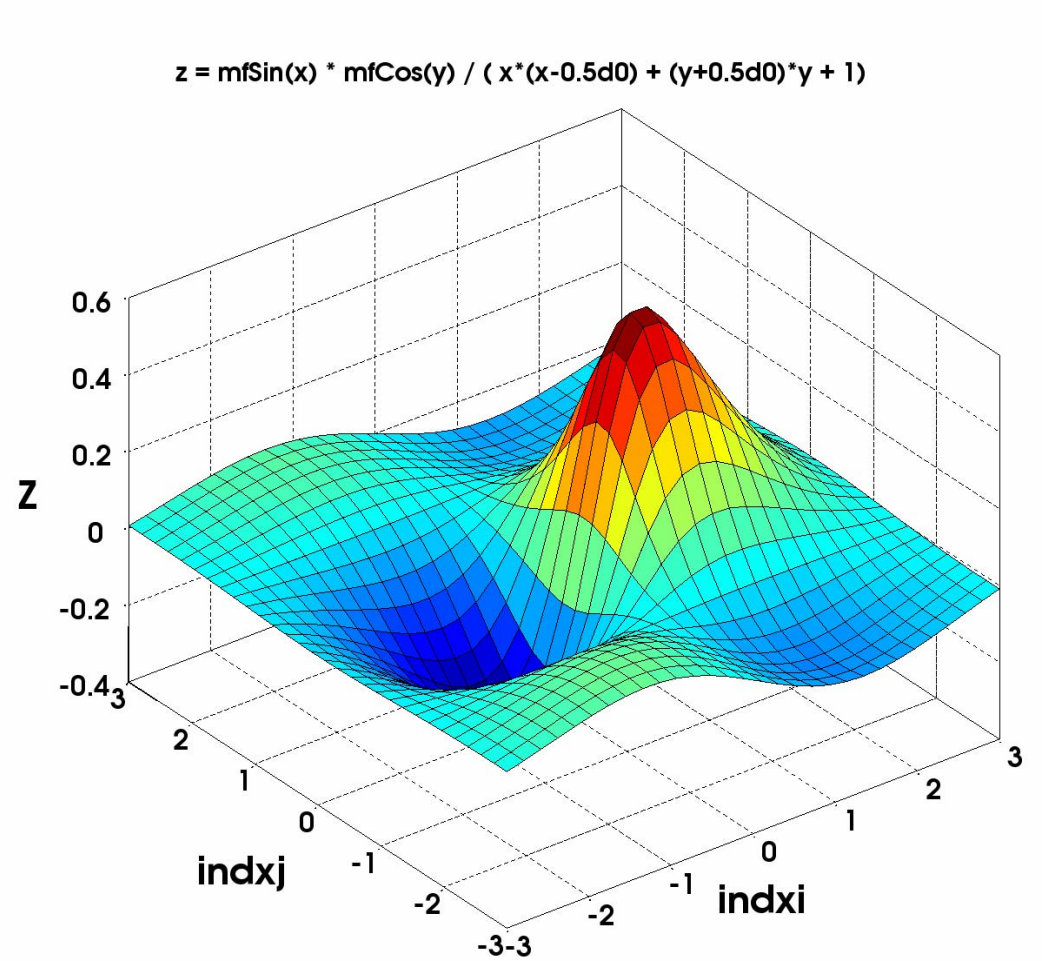


Figure 4.6.1 Axis labels and title

4.6.2 Text Annotation

You can add two-dimensional and three-dimensional text annotations with colors on your graph by using procedure *msText* and *msAnnotation*, respectively.

Example 4.6.2 Adding text annotations

Place a two-dimensional text annotation on the bottom-left corner of the figure window. The color of the text is set to purple.

```
call msText('Annotating Your Graph', mf((/0.1, 0.1/)), mf((/1, 0, 1/)))
```

Next, we shall locate the point with the maximum z value and label it with a three-dimensional text annotation. Try to rotate the graphics object; you'll see the text annotation moves with the maximum point.

```
pos = mfOnes(3, 1)
call msAssign(mfS(pos, 1, 1), mfS(x, 15,19)/6.0d0 + 0.065d0)
call msAssign(mfS(pos, 2, 1), mfS(y, 15,19)/6.0d0 - 0.02d0)
call msAssign(mfS(pos, 3, 1), mfS(z, 15,19) + 0.02d0)

call msAnnotation('Maximum', pos, mf((/0, 1, 1/)))
```

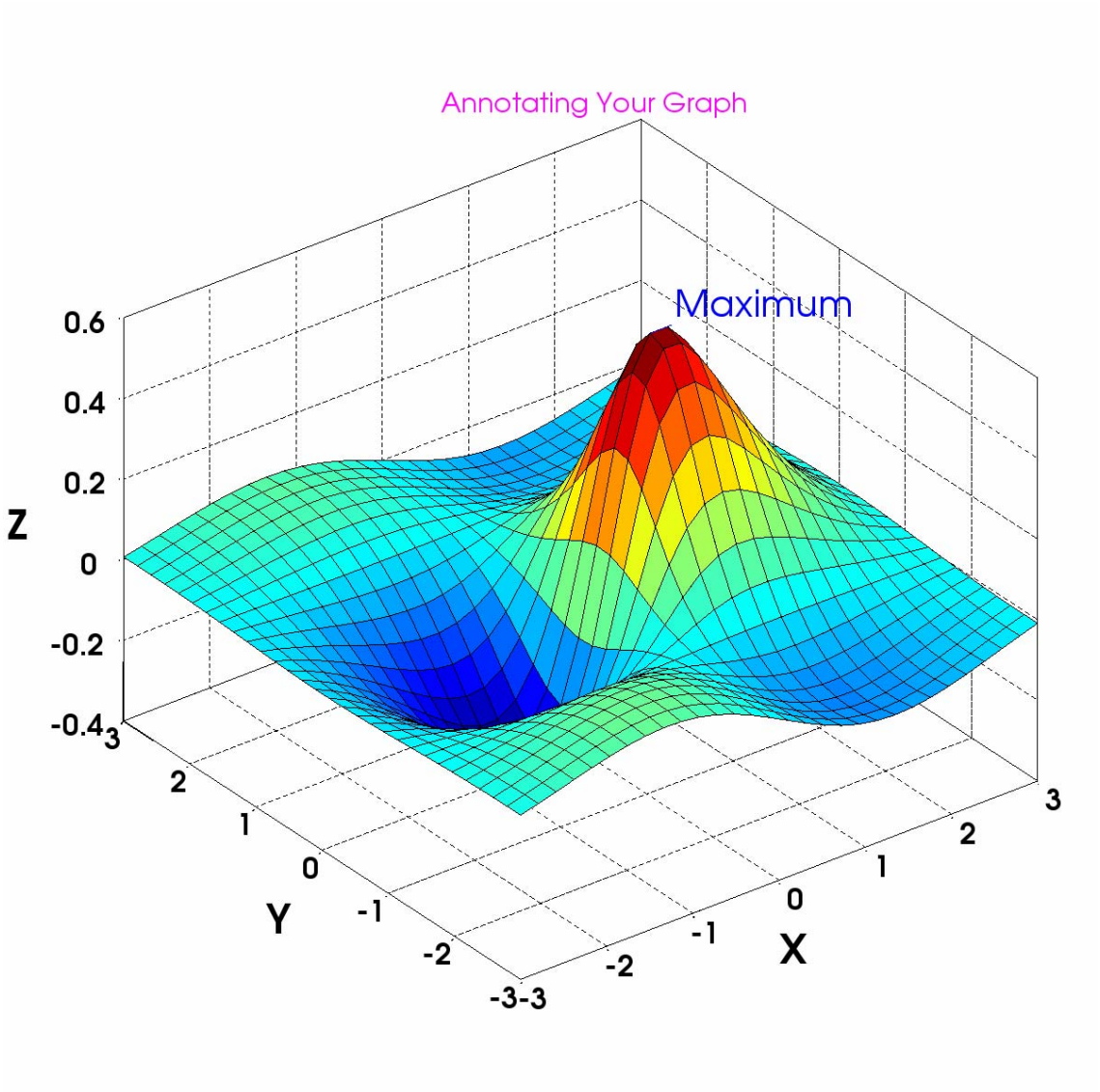


Figure 4.6.2 Two-dimensional and three-dimensional text annotations

4.7 Animation and Recording

Animations effects are produced by the continuous erase and update of data displayed in the Graphics Viewer. Animation can be recorded in two different formats, namely avi and bmp.

In this section, we shall animate the surface object drawn in example 4.3.1 using mfArrays *x*, *y*, and *z*.

4.7.1 Animation

Typically, you create an animation by following the steps below:

Step 1. Construct and initialize the mfArrays for plotting.

Step 2. Create a static plot of the graph you wish to animate and get its handle using *mfGetCurrentDraw*.

Step 3. Set up an iteration loop for the range of data you wish to observe through animation.

Step 4. Within the iteration loop, use procedure *call msGSet(handle, 'axis-data', data)* to update the targeted data of the current draw.

Step 5. Update the current Graphics Viewer by using procedure *msDrawNow;* animation effect is created.

Step 6. Pause the program after completing the animation to observe the static graph using procedure *msViewPause.*

Example 4.7.1 Animation

Create the surface object by using the data with procedure *msSurf* and use mfArray *h* to retrieve the handle of the object created.

```
call msSurf(x, y, z)
h = mfGetCurrentDraw()
```

Then, create an iteration loop to vary *z* data, using integer i = 1 to 300.

```
do i = 1, 300
```

```
   z = mfSin(x+0.08d0*i) * mfCos(y-0.13d0*i) / ( x*(x-0.5d0) +
(y+0.5d0)*y + 1)
   call msGset(h, 'zdata', z)
   call msDrawNow()
end do
```

Pause the program to view the surface object at the point of termination using procedure *msViewPause*.

Compile and run the program.

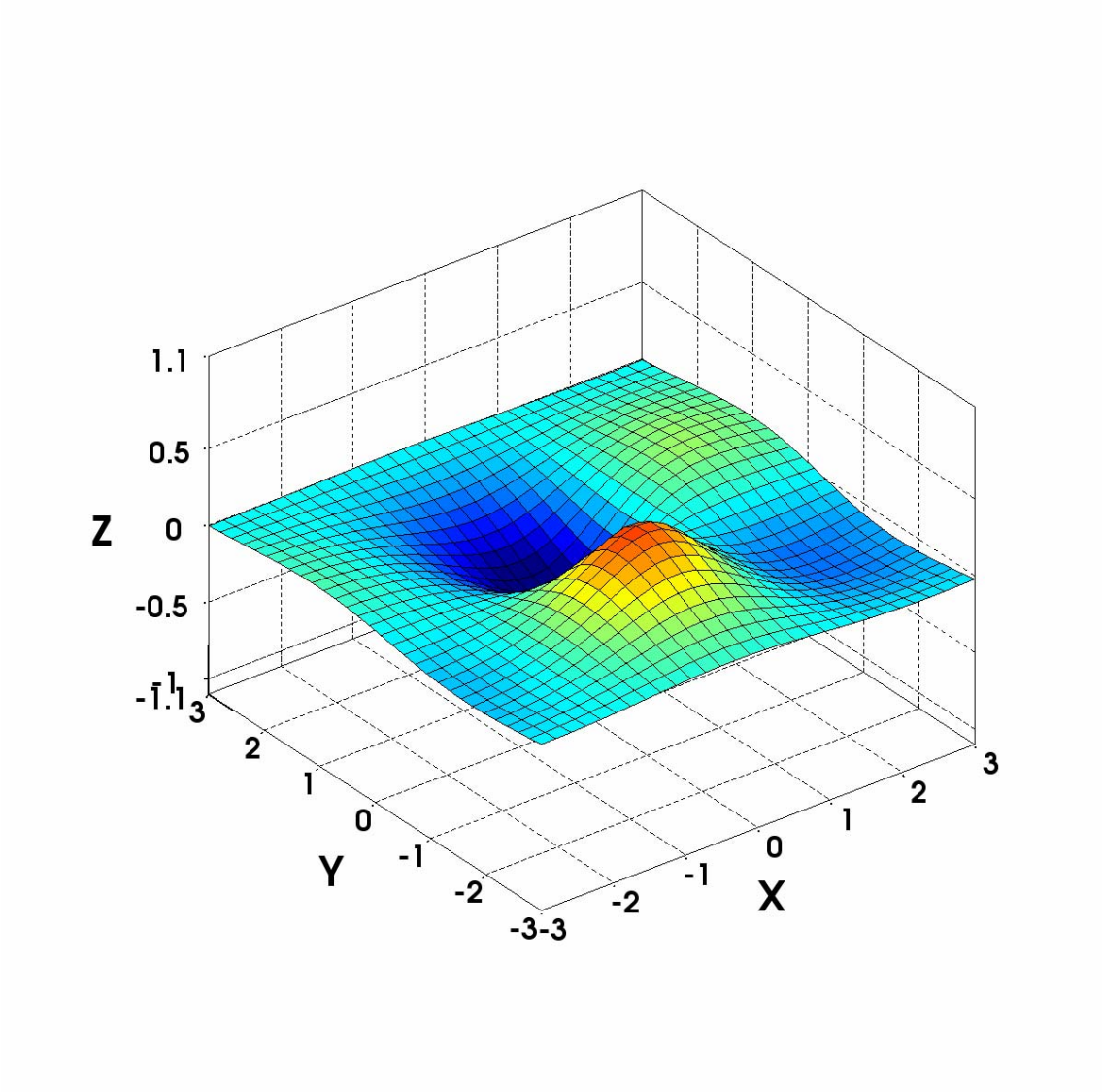


Figure 4.7.1 Animating surface object

4.7.2 Recording your animation

You can record animations of your graphics object as an *avi* movie file, MATFOR *mfa* file, or as picture files to view your simulation process at another time. The general syntax is as follows.

call msRecordStart('animation.avi')

or

call msRecordStart('animation.mfa')

or

call msRecordStart('animation.bmp')

<animation codes>

call msRecordEnd()

You have the options of to temporarily pause the animation or terminate the recording by clicking on the pause or stop button on the toolbar. When the stop bottom is pressed, the played animation up till the point of termination will still be recorded and the remaining animation will keep on rolling.

At the end of the recording, an end of recording dialog box pops out to notify you that the recording has been completed successfully. Click "OK" to continue.

When *call msRecordStart('animation.avi')* is used, MATFOR records an *avi* movie file using the compression method that you select. The Graphics Viewer pops up a Video Compression selection dialog box at the start of the recording, as shown in Figure 4.7.2.3 below.

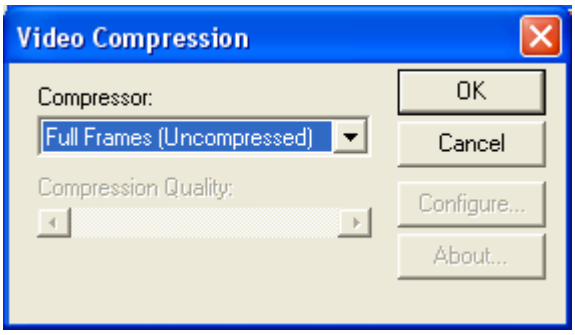


Figure 4.7.2.3 Video Compression selection dialog box

Example 4.7.2 Recording an animation

In this example, we shall record the animation created in example 4.7.1 into an avi file. Simply add the statements call *msRecordStart('.\data\Example4_7.avi')* and *call msRecordEnd()* at the beginning and end of the animation code respectively.

```
call msRecordStart('.\data\Example4_7.avi')

call msSurf(x, y, z)
h = mfGetCurrentDraw()
! Reset the axis ranges to yield a better animation
call msAxis(-3.0d0, 3.0d0, -3.0d0, 3.0d0, -1.1d0, 1.1d0)
do i = 1, 300
  z = mfSin(x+0.08d0*i) * mfCos(y-0.13d0*i) / ( x*(x-0.5d0) +
(y+0.5d0)*y + 1)
  call msGset(h, 'zdata', z)
  call msDrawNow()
end do

call msRecordEnd()
call msViewPause()
```

The recorded *'Example4_7.avi'* file will be located under the data folder in your project directory. By default, you can find it at *<MATFOR>\examples\for_ug\data*.

4.7.3 Image Exporting

A graph displaying on the Graphics Viewer can be captured and saved into a picture file. This is easily accomplished by using the procedure *msExportImage*. Various picture formats are supported, e.g. bmp, jpeg, tiff, ps and png.

It can also be accomplished by using the **Export to File** function located under the File Menu.

You may define the size of the picture format to be saved using either one of the two methods.

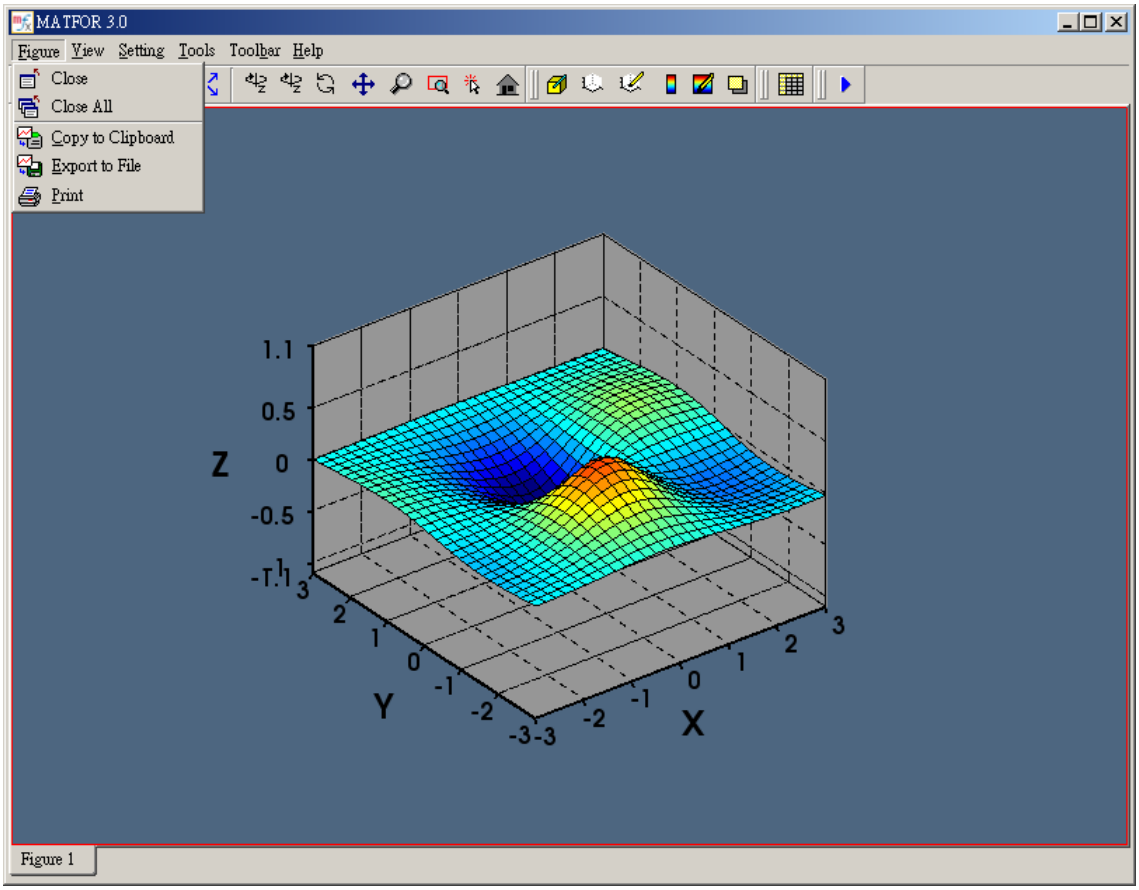


Figure 4.7.3 Capture displaying graph

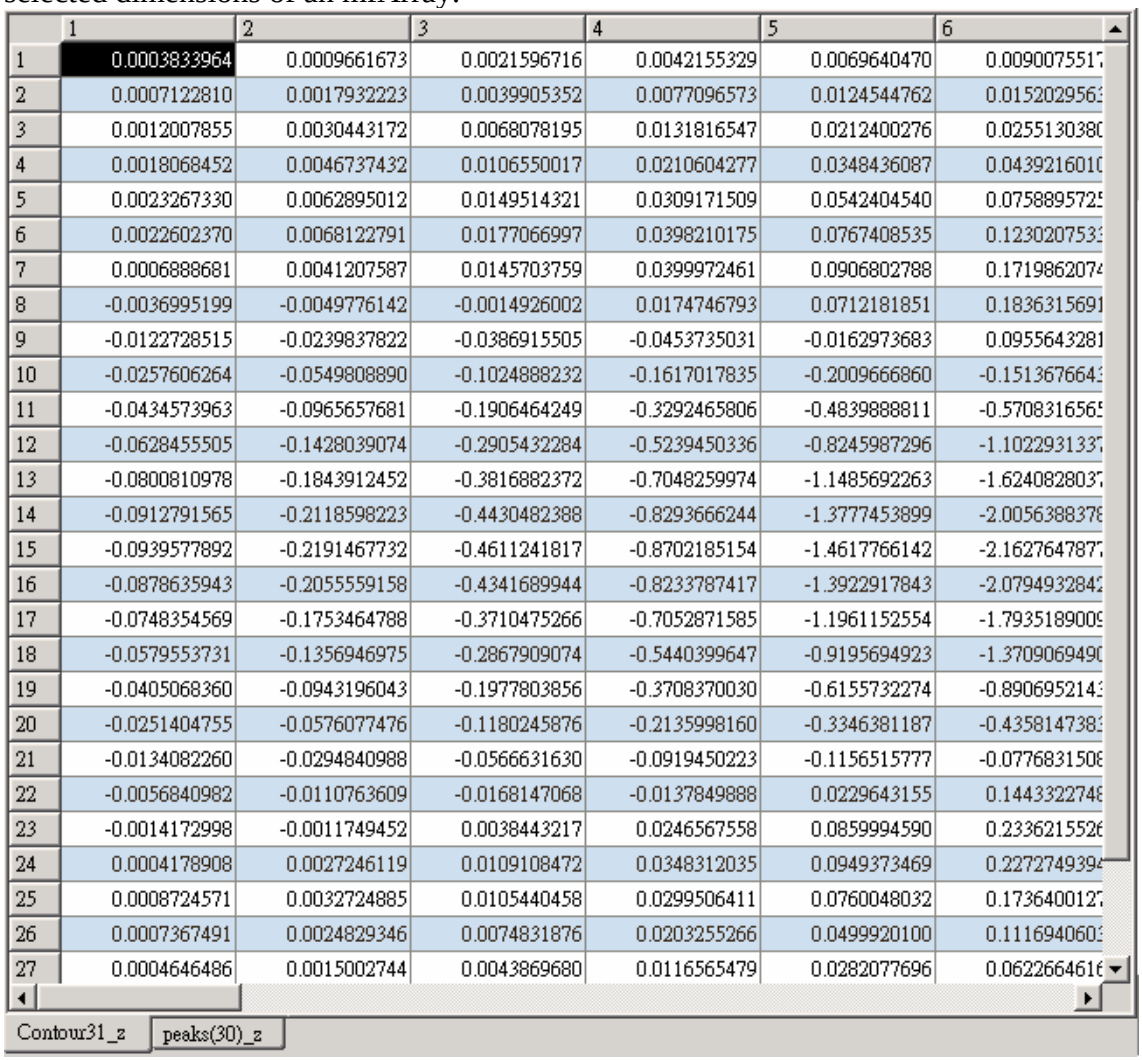
4.8 MATFOR Data Viewer

MATFOR Data Viewer is a powerful tool that displays data in a spreadsheet-like editor and enables you to perform additional manipulations on the data. It is composed of six major components, namely Matrix Table, Menu, Toolbar, Sampling Type, Panels, and Status Bar.

There are three kinds of panels in MATFOR Data Viewer, which are Snapshot Panel, Analysis Panel, and Filter Panel.

4.8.1 Matrix Table

Matrix Table is where actual entries are displayed. The data displayed are entries in two selected dimensions of an mfArray.



	1	2	3	4	5	6
1	0.0003833964	0.0009661673	0.0021596716	0.0042155329	0.0069640470	0.009007551
2	0.0007122810	0.0017932223	0.0039905352	0.0077096573	0.0124544762	0.015202956
3	0.0012007855	0.0030443172	0.0068078195	0.0131816547	0.0212400276	0.025513038
4	0.0018068452	0.0046737432	0.0106550017	0.0210604277	0.0348436087	0.043921601
5	0.0023267330	0.0062895012	0.0149514321	0.0309171509	0.0542404540	0.075889572
6	0.0022602370	0.0068122791	0.0177066997	0.0398210175	0.0767408535	0.123020753
7	0.0006888681	0.0041207587	0.0145703759	0.0399972461	0.0906802788	0.171986207
8	-0.0036995199	-0.0049776142	-0.0014926002	0.0174746793	0.0712181851	0.183631569
9	-0.0122728515	-0.0239837822	-0.0386915505	-0.0453735031	-0.0162973683	0.095564328
10	-0.0257606264	-0.0549808890	-0.1024888232	-0.1617017835	-0.2009666860	-0.151367664
11	-0.0434573963	-0.0965657681	-0.1906464249	-0.3292465806	-0.4839888811	-0.570831656
12	-0.0628455505	-0.1428039074	-0.2905432284	-0.5239450336	-0.8245987296	-1.102293133
13	-0.0800810978	-0.1843912452	-0.3816882372	-0.7048259974	-1.1485692263	-1.624082803
14	-0.0912791565	-0.2118598223	-0.4430482388	-0.8293666244	-1.3777453899	-2.005638837
15	-0.0939577892	-0.2191467732	-0.4611241817	-0.8702185154	-1.4617766142	-2.162764787
16	-0.0878635943	-0.2055559158	-0.4341689944	-0.8233787417	-1.3922917843	-2.079493284
17	-0.0748354569	-0.1753464788	-0.3710475266	-0.7052871585	-1.1961152554	-1.793518900
18	-0.0579553731	-0.1356946975	-0.2867909074	-0.5440399647	-0.9195694923	-1.370906949
19	-0.0405068360	-0.0943196043	-0.1977803856	-0.3708370030	-0.6155732274	-0.890695214
20	-0.0251404755	-0.0576077476	-0.1180245876	-0.2135998160	-0.3346381187	-0.435814738
21	-0.0134082260	-0.0294840988	-0.0566631630	-0.0919450223	-0.1156515777	-0.077683150
22	-0.0056840982	-0.0110763609	-0.0168147068	-0.0137849888	0.0229643155	0.144332274
23	-0.0014172998	-0.0011749452	0.0038443217	0.0246567558	0.0859994590	0.233621552
24	0.0004178908	0.0027246119	0.0109108472	0.0348312035	0.0949373469	0.227274939
25	0.0008724571	0.0032724885	0.0105440458	0.0299506411	0.0760048032	0.173640012
26	0.0007367491	0.0024829346	0.0074831876	0.0203255266	0.0499920100	0.111694060
27	0.0004646486	0.0015002744	0.0043869680	0.0116565479	0.0282077696	0.062266461

Contour31_z | peaks(30)_z

Figure 4.8.1 Matrix Table

4.8.1.1 Cell Color

The cells in the Matrix Table may have different colors and each cell color has a its meaning. The default color (usually white) is to represent entries in odd rows and the sky blue color is to represent entries in even rows.

You can also define new colors for cells for emphasizing purpose through Filter Panel. Descriptions on Filter Panel can be found in Section 4.8.7.

4.8.1.2 Cell Selection

The selected entry will be shown in the Status Bar. You can select a cell entry by clicking the mouse or by using the Goto Cell dialog box.

4.8.1.3 Array Selection

Similar to MATFOR Graphics Viewer, each array-displaying window is attached to a tab. This allows you to switch between arrays windows very easily.

4.8.2 Menu

The menu functions support file saving functions.

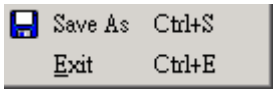


Figure 4.8.2.1 File Menu

You may notice that MATFOR Data Viewer does not support file-opening function. This is because it is not an independent application as all the data are input from procedure calls or through MATFOR Graphics Viewer.

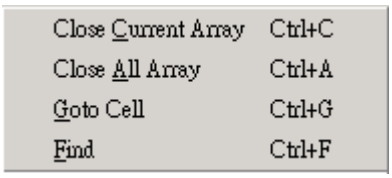


Figure 4.8.2.2 View Menu

The Close Current Array function closes the selected array. The Close All Array function under the View Menu closes all the arrays that are currently displayed in the Data Viewer. It is equivalent to the Exit function.

The Goto Cell function enables you to jump to a specific cell given its row index and column index. A dialog box will pop up prompting you to input, as illustrated in Figure 4.8.2.3.

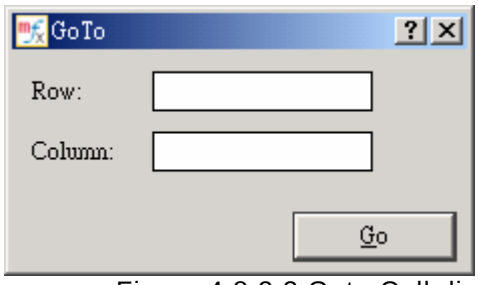


Figure 4.8.2.3 Goto Cell dialog box

Find the entries that satisfy the condition you insert. The format of the condition is similar to the one used for filtering. Refer to Section 4.8.7 Filter Panel.

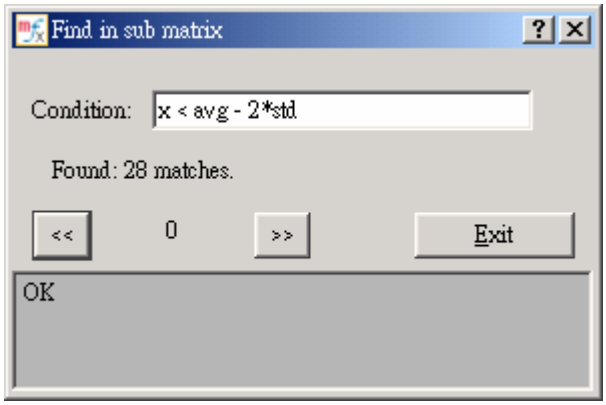


Figure 4.8.2.4 Find dialog box

4.8.3 Toolbar

MATFOR Data Viewer also provides you with some quick buttons to use the menu functions on the Toolbar as shown in the left part of Figure 4.8.3.

You can reset the width of the grids by dragging the slide bar. The width is specified in pixels. The number of precisions can also be reset using the slide bar for number of digits to display.

Figure 4.8.3 Toolbar

4.8.4 Sampling Type

With the Sampling Range options, you can select either to display the full array or sub-matrix. For the sub-matrix option, if the range selection is (:, :, 3), the Data Viewer will display 900 entries of the sub-matrix (1:30, 1:30, 3).

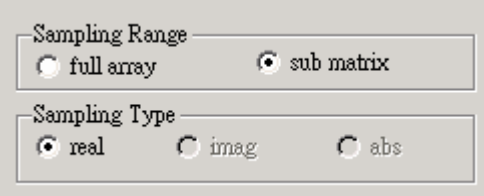


Figure 4.8.4 Range and type sampling

4.8.5 Snapshot Panel

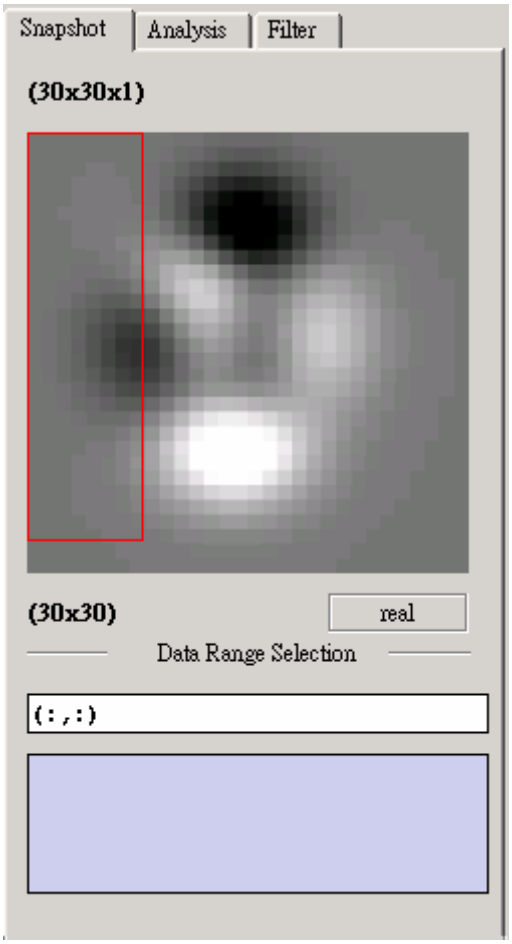


Figure 4.8.5 Snapshot Panel

The window in the middle displays a snapshot of the distribution and size of the two-dimensional data. The darkness of a cell is determined by mapping the value stored in the cell to a predefined range.

The range is defined by setting the upper bound to be the average + 3 x standard deviation and setting the lower bound to be the average – 3 x standard deviation.

In the snapshot window, the values that exceed the upper bound are all treated as the maximum value and the values that are less than the lower bound are treated as the minimum value. The range is further divided into 256 levels of darkness and the cells are drawn accordingly.

The string (30x30x1) above the snapshot window specifies the shape of the array being examined. The string (30x30) right below the snapshot window is the size of the dimensions of displaying array data.

The Data Range Selection input box allows you to specify the range of data to be displayed.

For example, if you want to display all entries in an 30-by-30 matrix, you simply input (:,:) and press Enter. When the input is (10:20, 10:), the Data Viewer displays only the entries in rows 10 to 20 and columns 10 to 30.

4.8.6 Analysis Panel

The Analysis Panel shows the distribution of the data. It also displays the average, standard deviation and min/max values, as displayed in Figure 4.8.6.

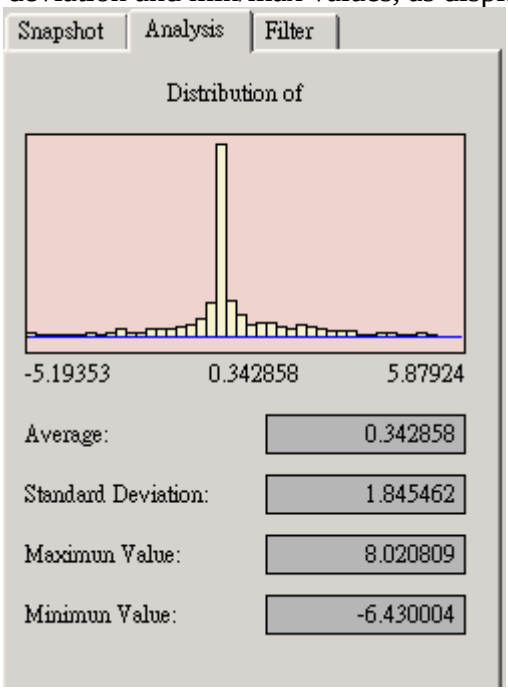


Figure 4.8.6 Analysis Panel

4.8.7 Filter Panel

The Filter Panel allows you to define a range using conditions of inequalities. The conditions of inequalities are specified in the condition boxes provided. The entries that satisfy a condition are highlighted in the color shown at the right side of the condition box.

Use ‘X’ or ‘x’ to represent the data being extracted. You may use *avg*, *std*, *max*, and *min* to represent the average, the standard deviation, maximum value, and minimum value, respectively.

MATFOR Data Viewer supports the inequality operators listed below: <, >, <=, >=. Notice that equal sign is not supported.

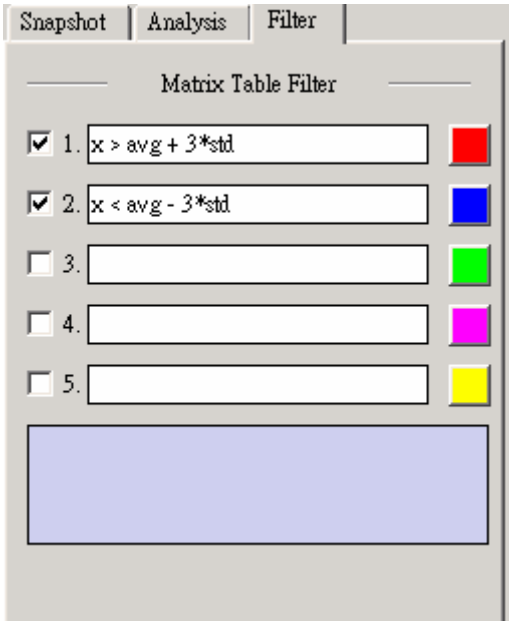


Figure 4.8.7 Filter panel

4.8.8 Status Bar

The Status Bar on the bottom of the Data Viewer contains two parts. The right one shows the system status and the left one shows the progress status.

The system status bar also displays the value when you select a specific cell.

Figure 4.8.8 Status bar

5 Visualization Methods

MATFOR's Graphics Library contains a set of visualization procedures for visualizing data in two-dimensional space and three-dimensional spaces. These visualization procedures can be categorized according to the type of data domain that they use to display.

Not all MATFOR Visualization procedures are described here. You may refer to the MATFOR Reference Guide to see detailed descriptions on every MATFOR procedures.

Examples with figures and diagrams will be presented for each major category.

5.1 Linear Graph

The data used for plotting linear graphs in two-dimensional space and three-dimensional space is the coordinates in vector forms.

This section is divided into two sub-sections. The first one presents the manipulations on the two-dimensional linear graph and the second covers the plotting of three-dimensional linear graphs using different representations.

Examples are provided for each sub-section.

5.1.1 Two-dimensional Linear Graph

Use the two-dimensional linear graph procedure *mfPlot* to visualize your data as trend lines. The procedure accepts different combinations of input arguments and allows you to plot multiple graphs with one procedure call.

The line color and marker type used by these graphs are specified through optional arguments of the procedures.

We shall go through example 5.1.1 to see how to plot a line graph for cosine using different combinations of line colors and market types.

Example 5.1.1 Two-dimensional linear plot

Create a new figure window and plot the cosine graph. The line color is set to red, which is specified with the optional argument *'r'*.

```
x = mfLinspace(-MF_PI, MF_PI,30)
y = mfCos(x)

call msFigure(1)
call msPlot(x, y, 'r')
call msTitle('Cosine graph' )
```

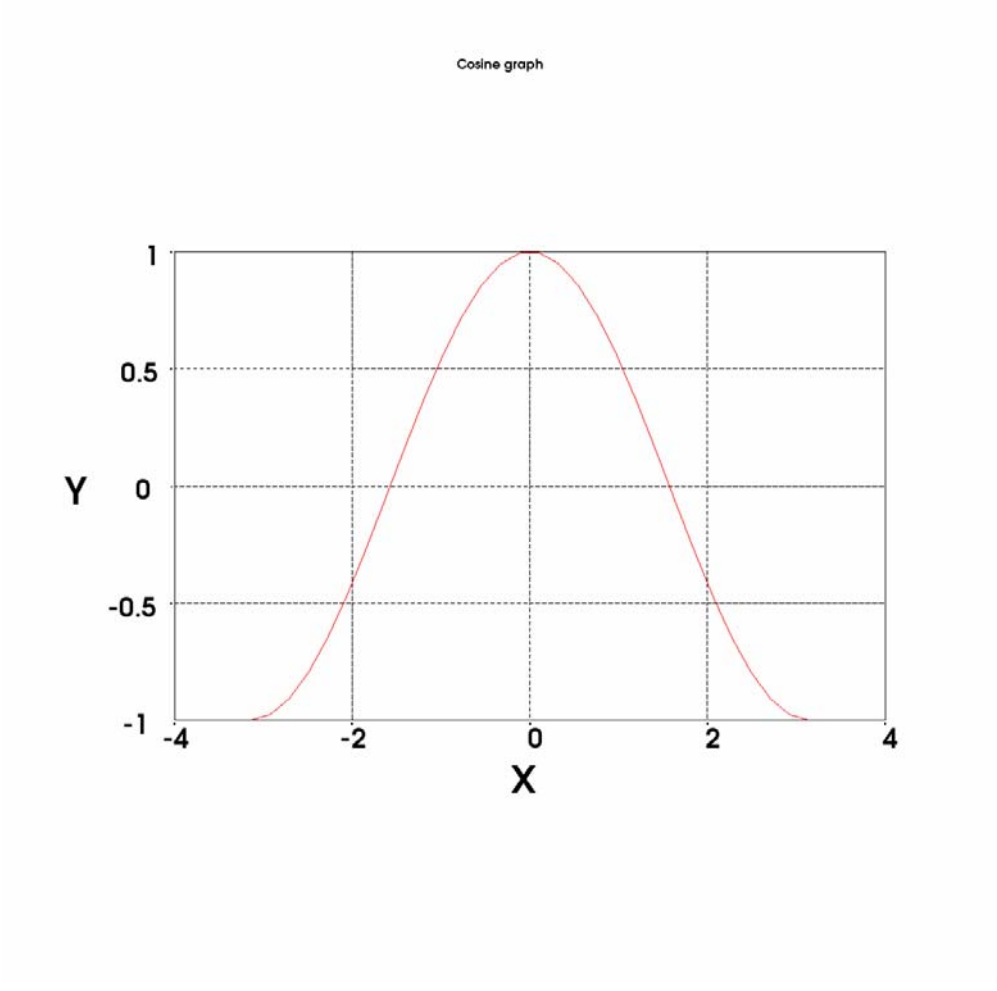


Figure 5.1.1.1 Cosine graph with red line

Plot a second cosine graph with the line specification set to ‘go-‘, which specifies a solid green color line with circle markers.

```
call msFigure(2)
call msPlot(x, y, 'go-')
call msTitle('Cosine graph')
```

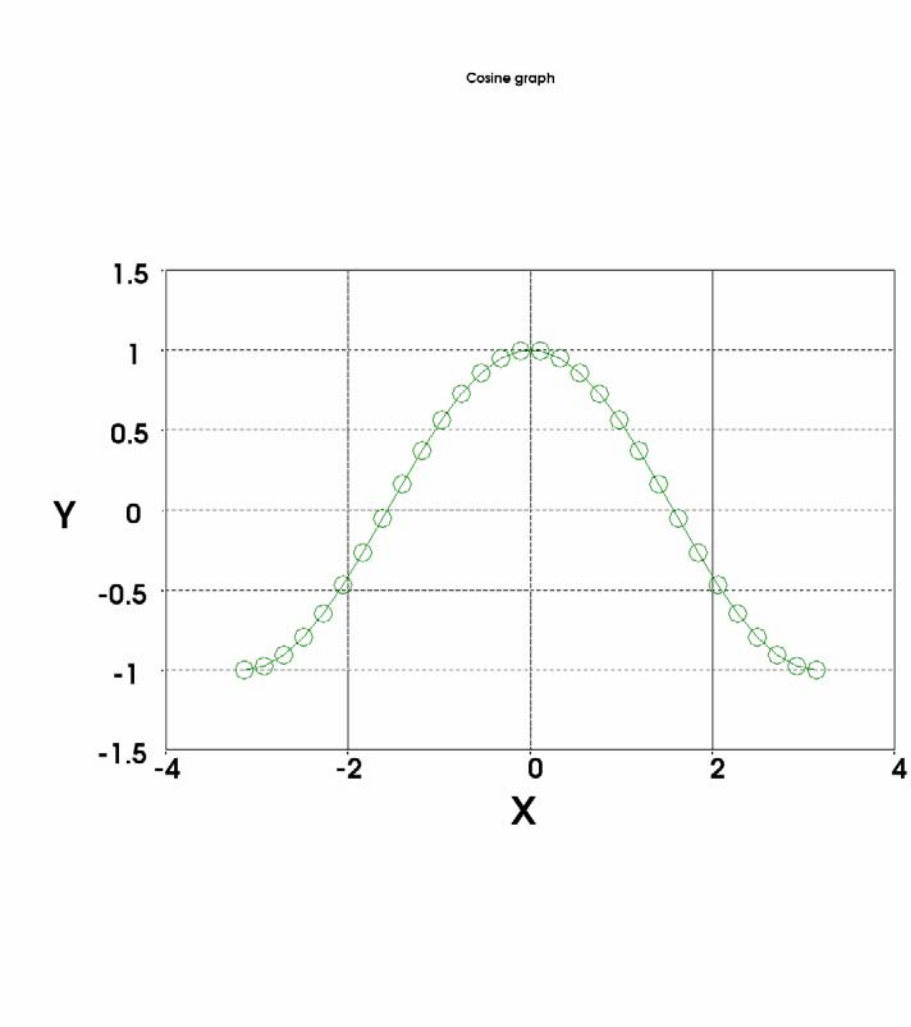


Figure 5.1.1.2 Cosine graph with green line and circle markers

5.1.2 Three-dimensional Linear Graph

MATFOR provides the procedures *mfPlot3*, *mfTube*, and *mfRibbon* for visualizing linear graphs in three-dimensional space as trend lines, tubes, and ribbons.

Example 5.1.2 should give you a clear picture of what the three kinds of three-dimensional linear graphs look like.

Example 5.1.2 Three-dimensional linear plot

Create three figure windows with the names *'plot3'*, *'tube'*, and *'ribbon'* for plotting the data in line graph, tube graph, and ribbon graph respectively.

```
call msFigure('plot3')
h = mfPlot3(x, y, z)
call msAxis(-30,30,-30,30,0,30)
call msCamZoom(1.4d0)

call msFigure('tube')
h = mfTube(x, y, z)
call msAxis(-30,30,-30,30,0,30)
call msCamZoom(1.4d0)

call msFigure('ribbon')
h = mfRibbon(x, y, z)
call msAxis(-30,30,-30,30,0,30)
call msCamZoom(1.4d0)
```

The results are displayed in the following figures.

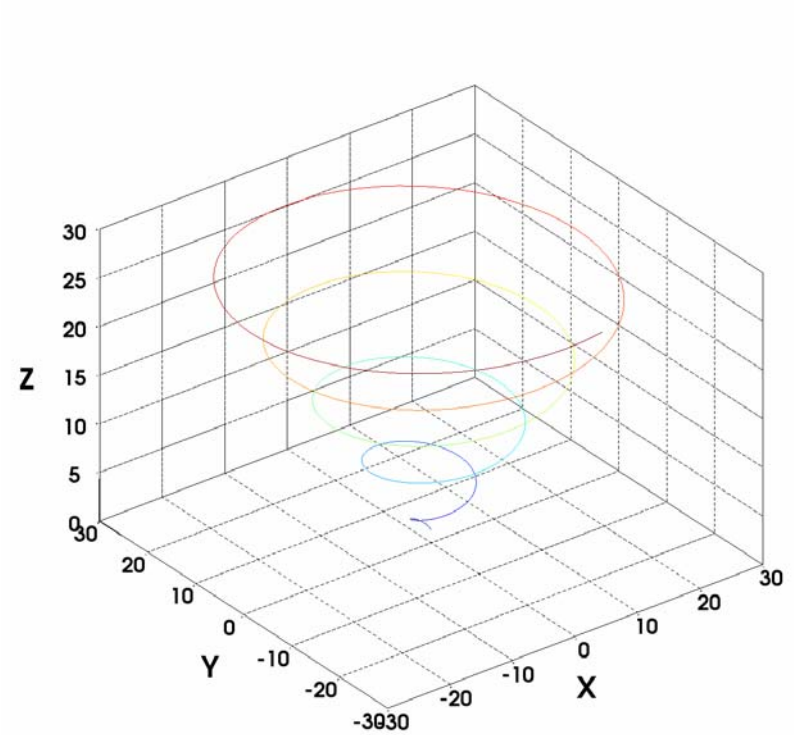


Figure 5.1.2.1 Line graph

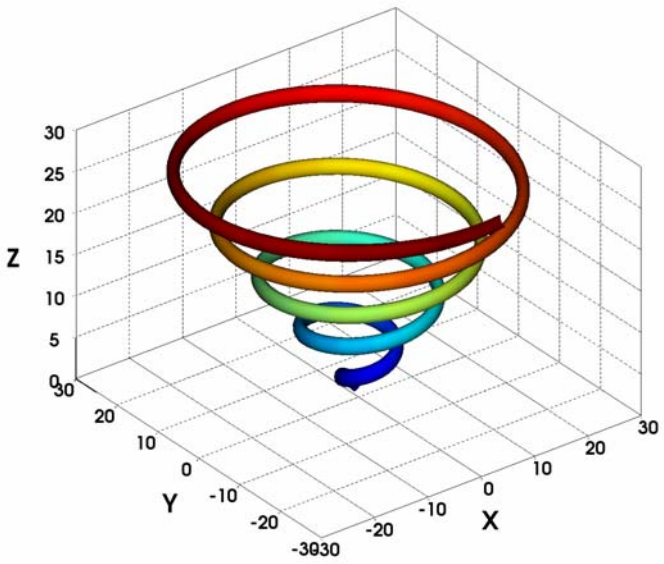


Figure 5.1.2.2 Tube graph

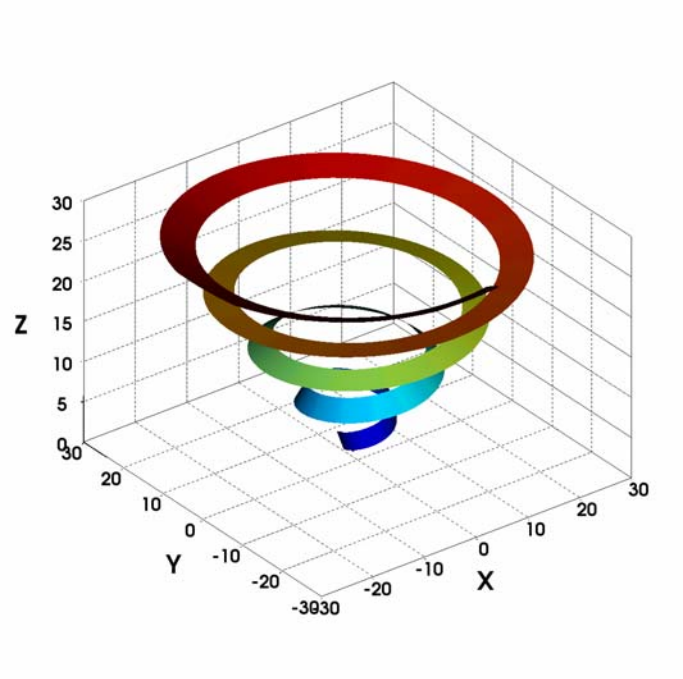


Figure 5.1.2.3 Ribbon graph

5.2 Surface Plot

In MATFOR, quadrilateral grid data can be plotted using various representations, such as surface graph, mesh graph, and contour lines.

This section goes through some examples on plotting a quadrilateral surface using these representations.

5.2.1 Surface plot

In the following example, we shall illustrate how to plot a quadrilateral grid in three-dimensional space with procedures *mfSurf*, *mfMesh*, *mfSurfc*, and *mfMeshc*.

Example 5.2.1 Surface plots using different methods

First, create a new figure window with the name *'surface'*.

```
call msFigure('surface')
```

Divide the figure window into four subplots and draw a surface plot on each of them using different representations.

You may notice that by using procedures *mfSurfc* and *mfMeshc*, a two-dimensional contour plot is added to the surface plot.

```
call msSubplot(2, 2, 1)
call msTitle('surf')
h = mfSurf(x, y, z)
call msCamZoom(1.5d0)

call msSubplot(2, 2, 2)
call msTitle('mesh')
h = mfMesh(x, y, z)
call msCamZoom(1.5d0)

call msSubplot(2, 2, 3)
call msTitle('surfc')
h = mfSurfc(x, y, z)
call msCamZoom(1.5d0)

call msSubplot(2, 2, 4)
call msTitle('meshc')
h = mfMeshc(x, y, z)
call msCamZoom(1.5d0)
```

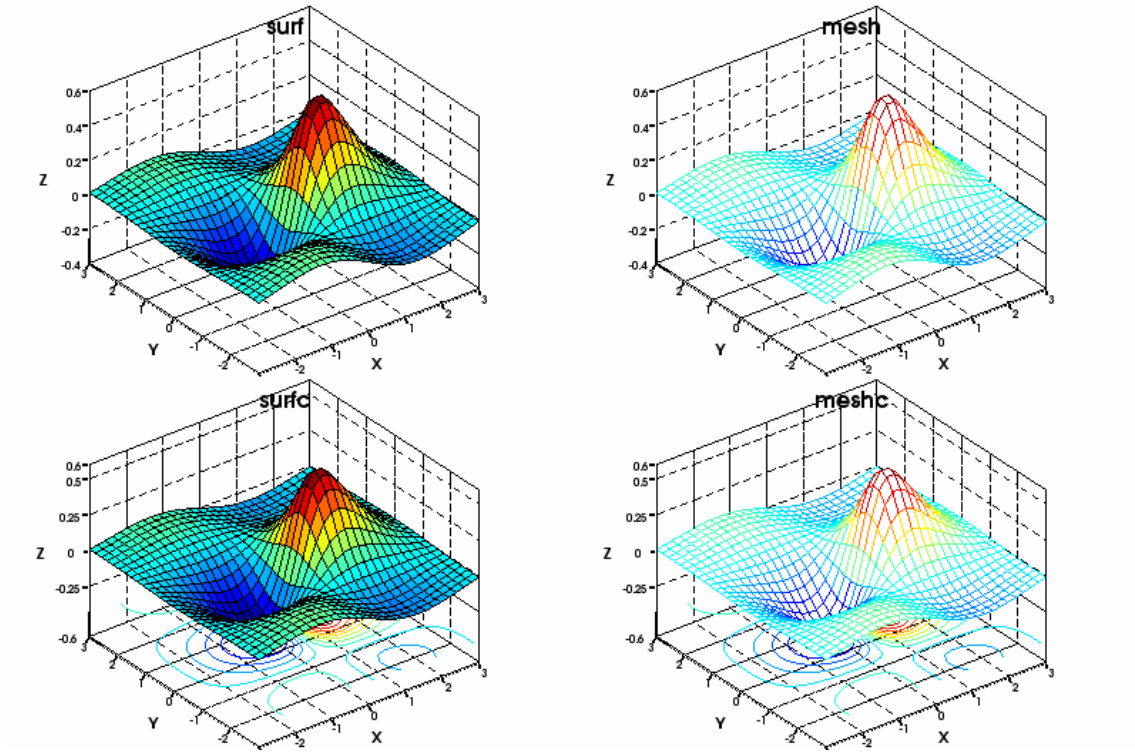


Figure 5.2.1 Surface plots

5.2.2 Contour plot

Contour plots are lines or surfaces of constant scalar values. They can be drawn in two-dimensional space or three-dimensional space using procedures *mfContour*, *mfContour3*, *mfSolidContour*, and *mfSolidContour3*.

The procedure *msOutline* allows you to draw a wireframe outline boundary for a given data set. It is often used when drawing contour plots.

Using the same data set as the one used in example 5.2.1, we shall plot the contour graphs using different representations in the example below.

Example 5.2.2 Using contours

Create a new figure window with the name '*contour*'.

```
call msFigure('contour')
```

Divide the figure window into four subplots and draw a contour plot on each of them using different representations and shading options.

```
call msSubplot(2, 2, 1)
call msTitle('contour3 with outline')
h = mfContour3(x, y, z)
call msHold('on')
h = mfOutline(x, y, z)
call msDrawMaterial(h, 'edge', 'color', mf( (/0, 0, 1/) ))
call msCamZoom(1.5d0)

call msSubplot(2, 2, 2)
call msTitle('contour')
h = mfContour(x, y, z)
call msAxis('equal')

call msSubplot(2, 2, 3)
call msTitle('solidcontour3 with outline')
h = mfSolidContour3(x, y, z)
call msHold('on')
h = mfOutline(x, y, z)
call msDrawMaterial(h, 'edge', 'color', mf( (/0, 0, 1/) ))
call msCamZoom(1.5d0)

call msSubplot(2, 2, 4)
call msTitle('solidcontour')
h = mfSolidContour(x, y, z)
call msAxis('equal')
```

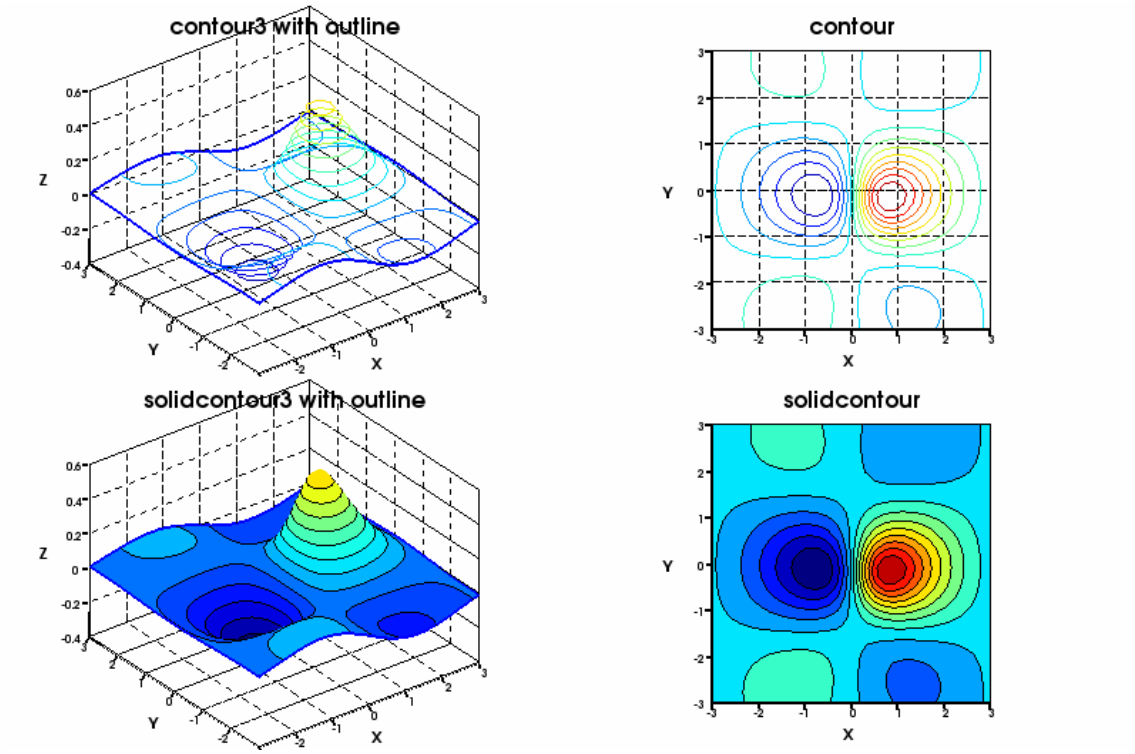


Figure 5.2.2 Contour plots

5.2.3 Pseudocolor plot

Using the pseudocolor plotting procedure, *mfPColor*, the data is mapped to the current colormap to represent the magnitude of the data value. The resulting graph is equivalent to the top-view of the one produced using procedure *mfSurf*.

Example 5.2.3 Pseudocolor plot

Create a new figure with the name *'pcolor'*.

```
call msFigure('pcolor')
```

Divide the figure window into four subplots and draw a pseudocolor plot on each of them using different shading methods.

```
call msSubplot(2, 2, 1)
call msTitle('pcolor')
h = mfPColor(x, y, z)
call msAxis('equal')

call msSubplot(2, 2, 2)
call msTitle('pcolor w/o edge')
```

```
h = mfPColor(x, y, z)
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'))
call msAxis('equal')

call msSubplot(2, 2, 3)
call msTitle('solid with interp and contour')
h = mfPColor(x, y, z)
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on'))
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'))
call msAxis('equal')

call msSubplot(2, 2, 4)
call msTitle('solid with interp and contour')
h = mfPColor(x, y, z)
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on'))
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'))
call msHold('on')
h = mfContour(x, y, z)
call msDrawMaterial(h, mf('edge'), mf('colormap'), mf('off'))
call msAxis('equal')
```

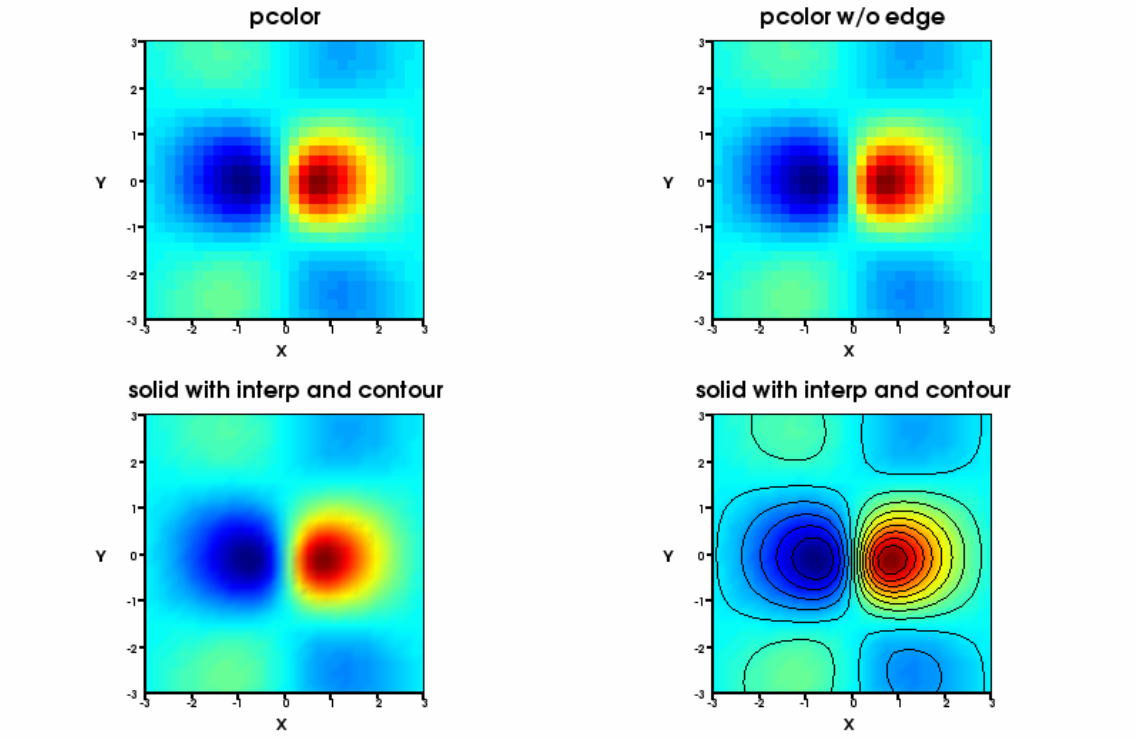


Figure 5.2.3 Pseudocolor plots

5.3 Volume Rendering

This section covers the procedures that visualize volumetric data in the representation of surface, mesh, slice-planes, and iso-surfaces. In MATFOR, the volumetric data is defined in three-dimensional mfArrays that specify the coordinates and scalar values of the data points.

In this section, we use an example that displays a portion of the nose on a missile using various representations to explore different aspects of the application.

We shall load data from hdd.mfb that contains all the information required, including x-, y-, z- coordinates, and a field data set.

In general, the x-, y-, and z- coordinates of the data set are plotted with a field data as the scalar values. If field data is not given, then z-coordinate will be treated as the scale values.

5.3.1 Surface (surf, mesh, outline, contour)

In the following example, displaying the data as surface plot or contour plot would give you different perspectives on the application.

Example 5.3.1

First of all, we simply use procedure *mfSurf* with a transparent shading to visualize the volumetric data as mesh grid.

```
h = mfSurf(x, y, z)
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on'), &
                                                    mf('trans'),
mf(50) )
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('on'), &
                                                    mf('colormap'),
mf('on') )
call msView(30, 45)
call msAxis('off')
call msCamZoom(1.8d0)
call msCamPan(0, -70)
```

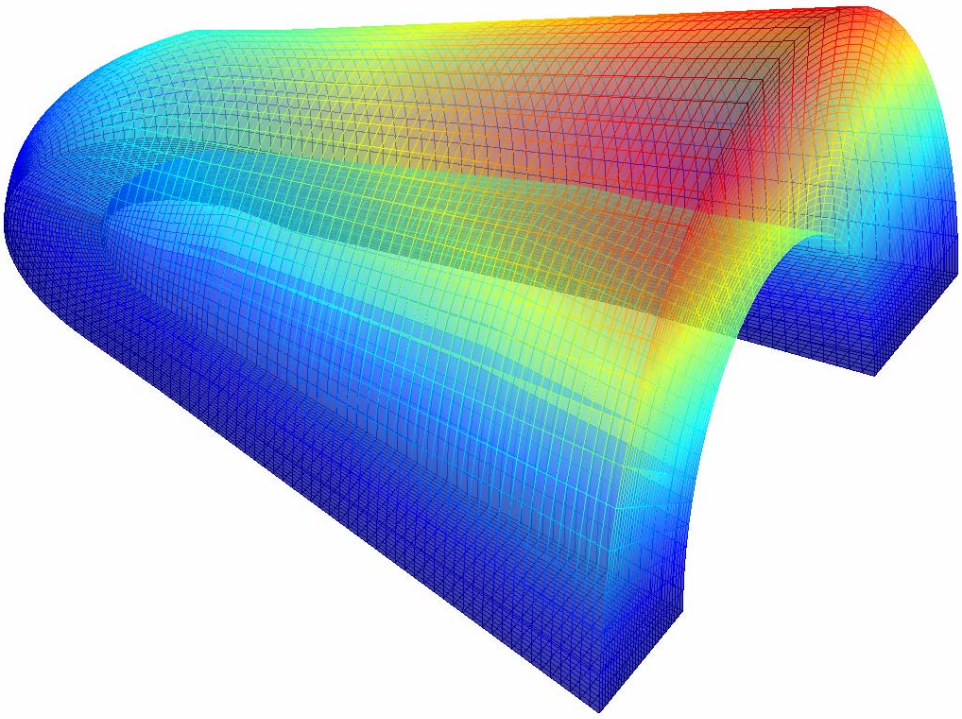


Figure 5.3.1.1 Mesh Plot of the volumetric data

Next, we split the data into two sets of data: (x1, y1, z1) and (x2, y2, z2), then display them in one figure simultaneously by plotting data set (x1, y1, z1) using procedure *mfSolidContour3* with field data mach1, and data set (x2, y2 and z2) using procedure *mfSurf* with field data mach2.

```
h = mfSolidContour3(x1, y1, z1, mach1)
call msHold('on')
```

```
h = mfSurf(x2, y2, z2, mach2)
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on'), &
                       mf('trans'), mf(50) )
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('on'), &
                        mf('colormap'), mf('on') )

call msColorbar('on')
call msView(30, 45)
call msAxis('off')
call msCamZoom(1.8d0)
call msCamPan(0, -70)
```

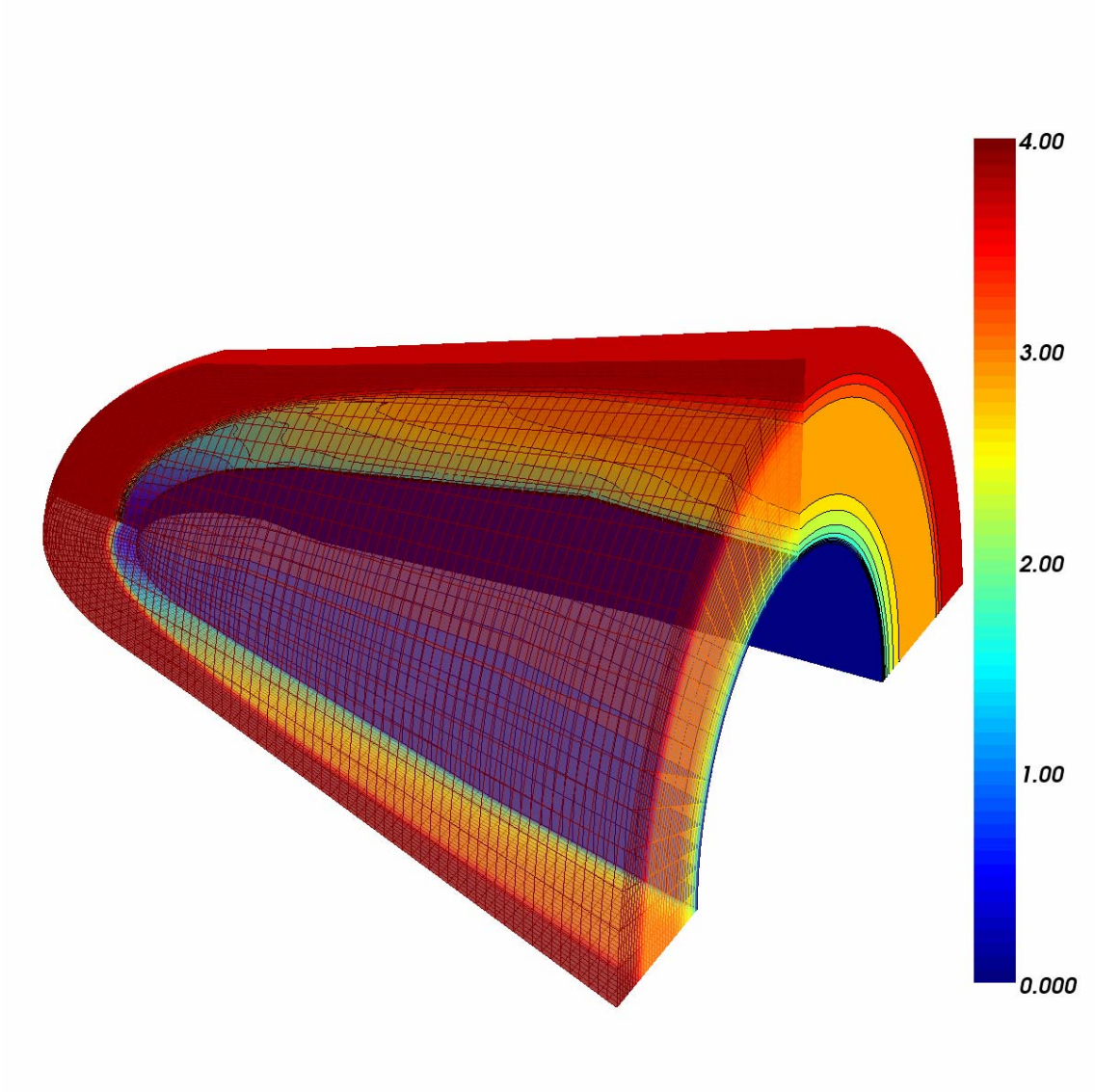


Figure 5.3.1.2 Solid contour and mesh plot of the volumetric data

5.3.2 Sliced-planes

MATFOR supports various slicing techniques to display slice-planes of a set of volumetric data.

Using procedure *mfSliceXYZ*, you can select any orthogonal slice-plane along x, y, and z directions to be displayed. Procedure *mfSlicePlane* allows you to cut a slice-plane along arbitrary directions.

Procedure *mfSliceIJK* displays slice-planes along i, j, and k, which are the indexes of the matrices that specify the coordinates. It has the following syntax:

```
call msSliceIJK(x, y, z, i, j, k)
```

In example 5.3.2, we shall display several slice-planes of the missile node object by using procedure *mfSliceIJK*.

Example 5.3.2

Display one slice-plane along the index of mfArray x, one slice-plane along the index of mfArray y, and five slice-planes along the index of mfArray z.

```
h = mfSliceIJK(x, y, z, mach, mf(m), mf(1), mfLinSpace(1, k, 5) )
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on') )
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'), &
                                                    mf('colormap'),
mf('on') )

call msHold('on')
call msColorbar('on')
call msView(30, 45)
call msAxis('off')
call msCamZoom(1.8d0)
call msCamPan(0, -70)
```

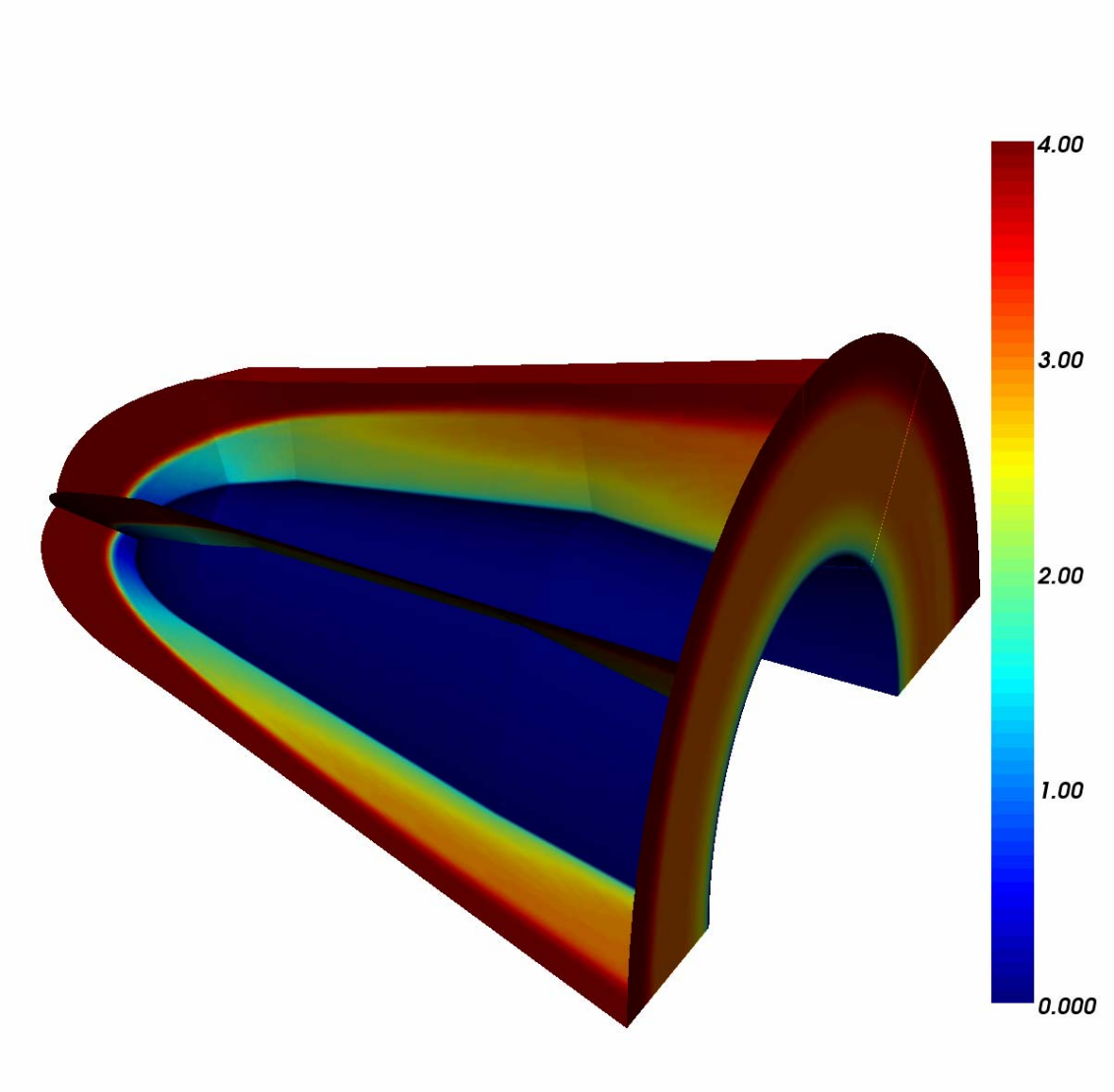


Figure 5.3.2 Show slice-planes of the volumetric data

5.3.3 Isosurface

Procedure *mfIsoSurface* creates 3-D graphs composed of iso-surface data from the volumetric data.

Example 5.3.3 combines a few representations of the missile nose using procedures *mfIsoSurface* and *mfOutline*.

Note that the data on each surface has the same iso-value, thus the color on each surface stays constant.

Example 5.3.3

Display iso-surfaces on the right part and front left part. The wireframe of the missile nose is also drawn.

```
! draw isosurfaces of right part
h = mfIsoSurface(x1, y1, z1, mach1, mfLinspace(1, 4, 6) )
call msHold('on')

! draw isosurfaces of the front left part
h = mfIsoSurface(x3, y3, z3, mach3, mfLinspace(1, 4, 6) )

! draw outline of data
h = mfOutline(x, y, z)
call msColorbar('on')
call msView(30, 45)
call msAxis('off')
call msCamZoom(1.8d0)
call msCamPan(0, -70)
```

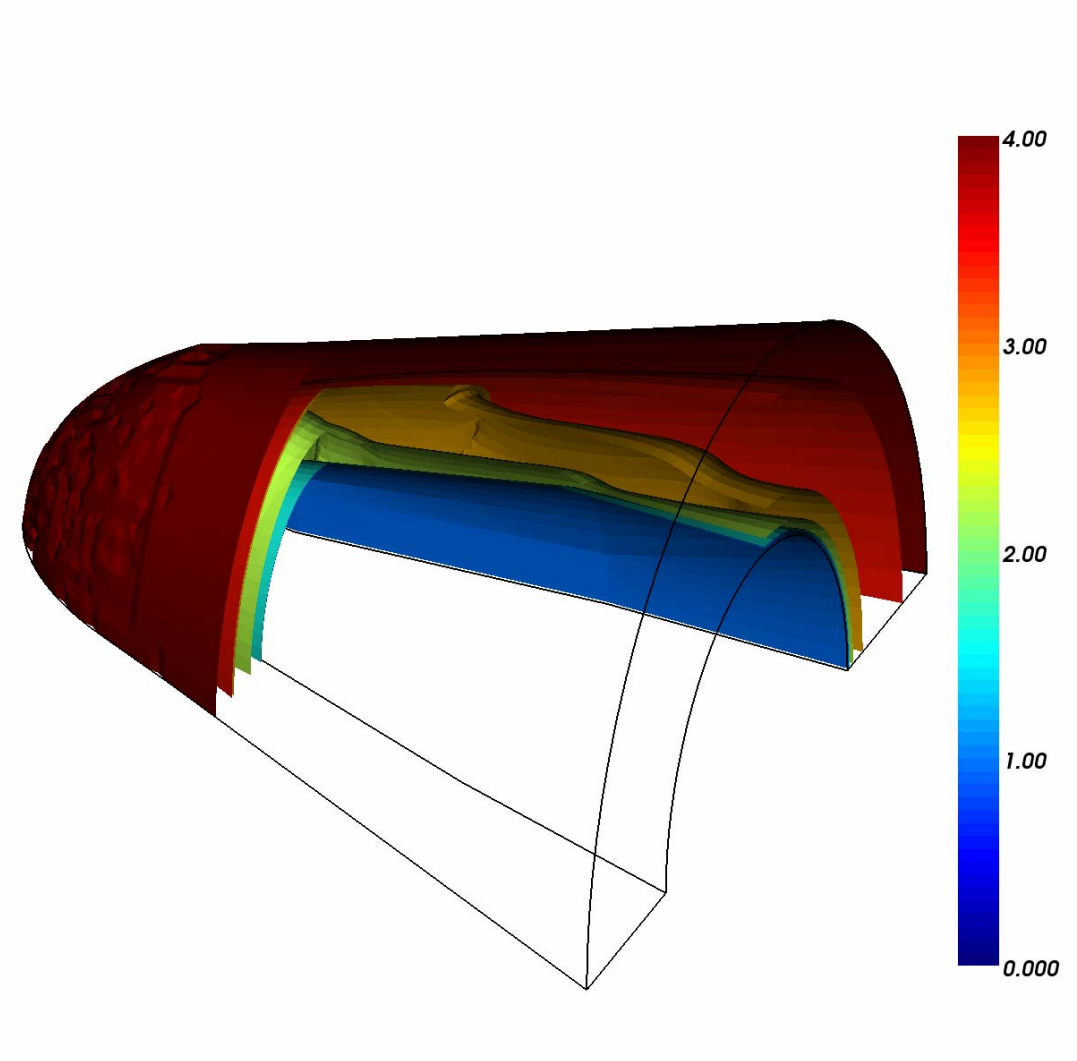


Figure 5.3.3 Isosurface plot of the volumetric data

5.4 Vector Field

Vector set in two-dimensional and three-dimensional space can be represented in quivers or streamlines using procedures *mfQuiver* and *mfStreamlLine.*

The streamlines are plotted by identifying their corresponding starting points.

5.4.1 Quiver and Streamline

In example 5.4.1, we shall generate a set of data and present it with quivers.

Example 5.4.1

Start by generating a set of mesh grid data.

```
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)
```

Plot the quivers using the data set created. The streamlines are then plotted by identifying the starting points.

```
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/)))
```

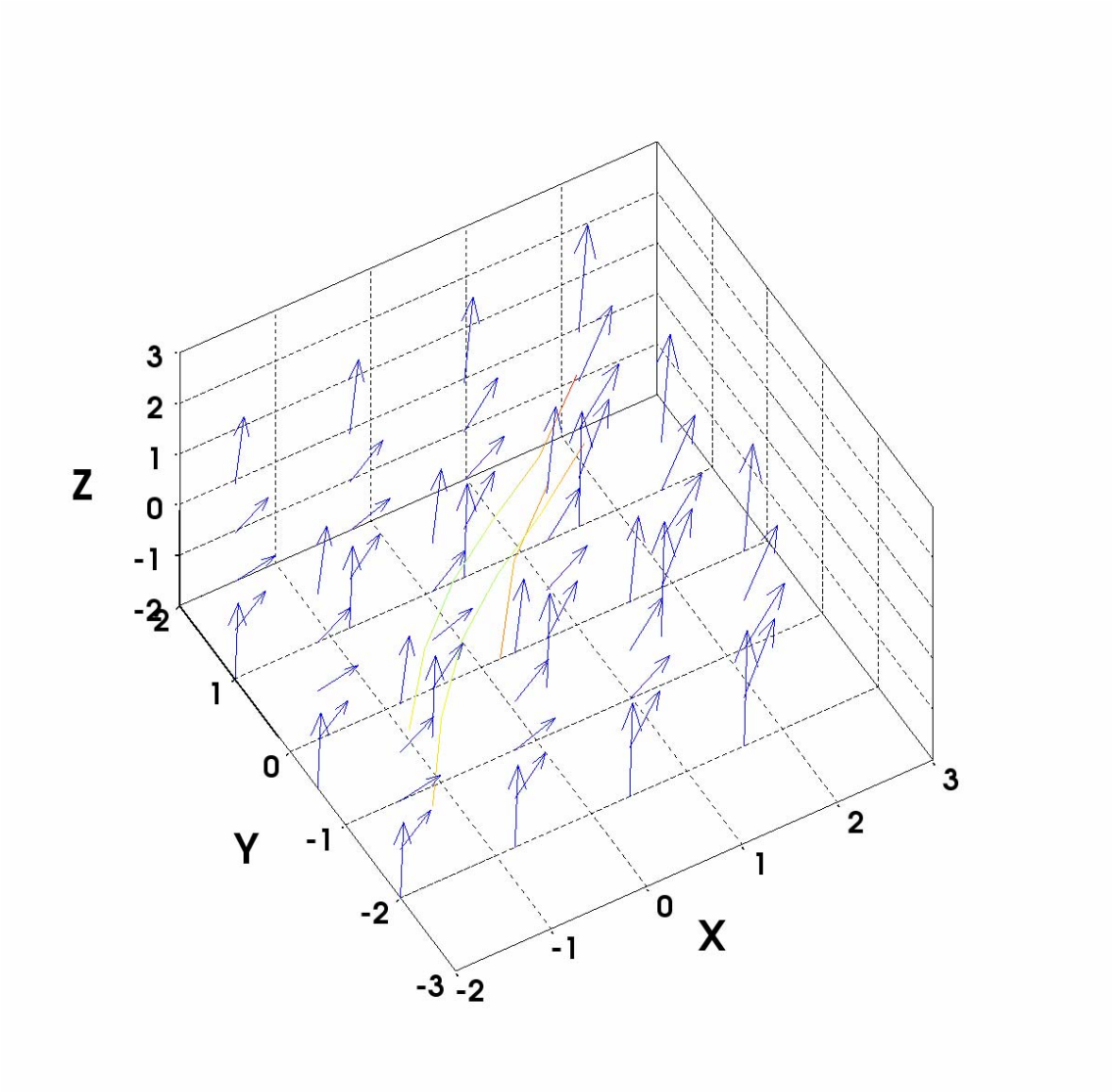


Figure 5.4.1 Plot quivers

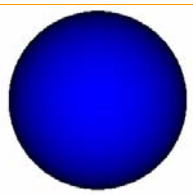
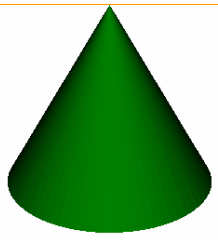
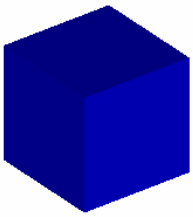
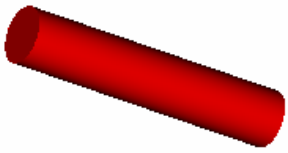
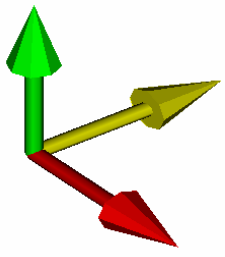
5.5 Elementary 3-D Objects

MATFOR provides you a set of elementary 3-D objects to place on the plot space.

In general, the elements are plotted with the given coordinates, sizes, and colors. When plotting molecules, the connectivity between the ball objects should also be specified.

5.5.1Primitives

The following is a list of 3-D objects MATFOR supports.

msSphere, mfSphere	
msCone, mfCone	
msCube, mfCube	
msCylinder, mfCylinder	
msAxisMark, mfAxisMark	

5.5.2 Molecule

A molecule plot is a bit different from other 3-D objects. It is often composed of balls and sticks.

In the example below, we shall import a protein-structured data that specifies the atom types, positions of the atoms, and the connectivity between them.

Here, We use different colors and sizes to represent the different atoms.
In Example 5.5.2, we shall demonstrate how to plot a molecule graph using the input of a structured protein data.

Example 5.5.2 Plotting structured protein graph

Load data from ASCII files to retrieve the atom types, positions of atoms, and the connectivity.

```
pos_data = mfLoadAscii('.\data\Protein1B9G_NPos.data')
conn = mfLoadAscii('.\data\Protein1B9G_Link.data')
s = mfSize(pos_data, 1)

loc = mfS(pos_data, MF_COL, 2.to.4)

atom = mfS(pos_data, MF_COL, 1)
```

Define the radius for each atom type

```
rad = atom + 0
call msAssign( mfS(rad, atom>14 ), 14 )
rad = 0.1d0*rad


color = mfOnes(s, 3) * 0.7d0
do i = 1, s
      ! C
   if (mfAll(mfS(atom, i)==12)) then
         call msAssign( mfS(color, i, MF_COL), (/ 0.8, 0.8, 0.8 /) )
   ! N
   else if (mfAll(mfS(atom, i)==14)) then
         call msAssign( mfS(color, i, MF_COL), (/ 0.2, 0.2, 1.0 /) )
   ! O
   else if (mfAll(mfS(atom, i)==16)) then
         call msAssign( mfS(color, i, MF_COL), (/ 0.9, 0.0, 0.0 /) )
     ! S
   else if (mfAll(mfS(atom, i)==32)) then
         call msAssign( mfS(color, i, MF_COL), (/ 1.0, 1.0, 0.0 /) )
   end if
end do
```

Last, define the radius and color of the sticks to be connected between the atoms and then plot the structured protein.

```
stick_rad = 0.1d0 * mfOnes(s, 1)
```

```
stick_col = ((/0, 1, 0/))

call msMolecule(loc, conn, rad, color, stick_rad, stick_col, mf(6))
```

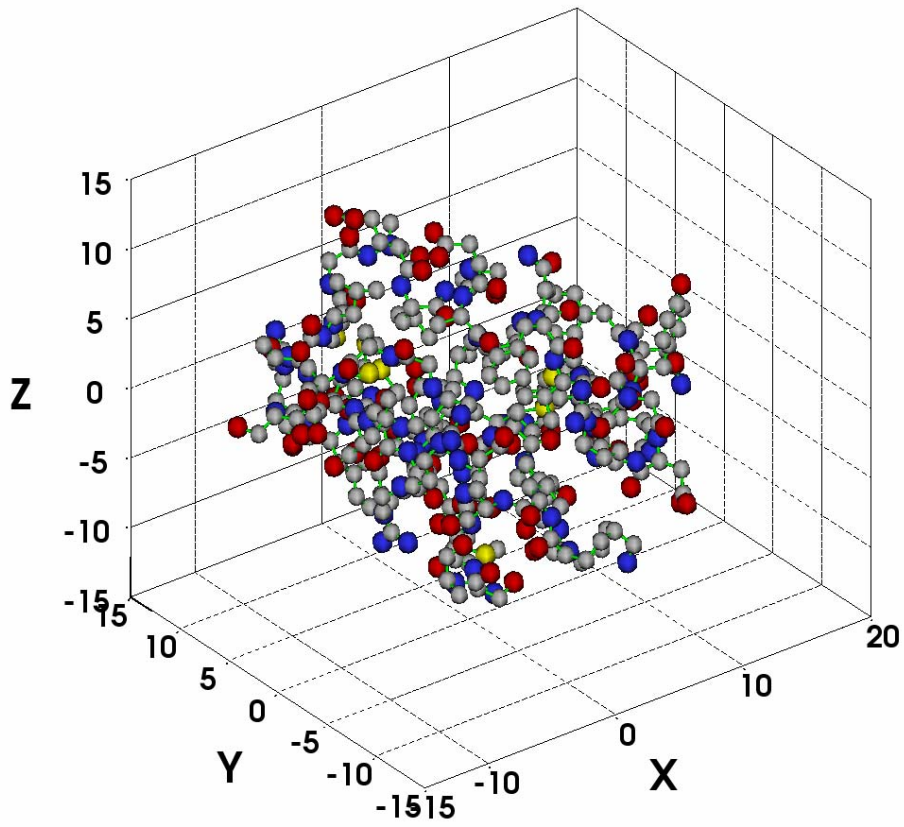


Figure 5.5.2 Structured protein graph

5.6 Unstructured Mesh

An arbitrary mesh or surface can usually be represented using an unstructured mesh that includes a set of vertex and a face set.

The vertices are the coordinates in space and the face set specifies the surface connectivities between the vertices.

In MATFOR, unstructured mesh can be visualized by using procedures *mfTriSurf*, *mfTriMesh*, and *mfTriContour*.

The most often-used face connectivity is triangular representation. MATFOR supports this representation and extends to other polygonal representations, such as quadrilateral, pentagon, etc.

5.6.1 Surface

In Example 5.6.1, we shall prepare an icosahedron (a polyhedron composed of 20 faces that span 12 vertices) and perform sub-divisions on it. Results are displayed using procedure *mfTriSurf*, *mfTriMesh* and *mfTriContour*.

Example 5.6.1 Building an icosahedron

Define the vertices and triangles that make up an icosahedron and draw the icosahedron using *mfTriSurf*.

```
x = 0.525731112119133606
z = 0.850650808352039932
xyz = reshape((/-x, x, -x, x, 0.0d0, 0.0d0, 0.0d0, 0.0d0, z, -z, z, -z, &
            0.0d0, 0.0d0, 0.0d0, 0.0d0, z, z, -z, -z, x, x, -x, -x, &
                        z, z, -z, -z, x, -x, x, -x, 0.0d0, 0.0d0, 0.0d0,
0.0d0/), (/12,3/))
tri1 = reshape((/2, 5, 5, 9, 2, 2, 11, 9, 4, 4, 4, 11, 7, 7, 7, 11, 12, 3, 6,
12, &
             5, 10, 6, 6, 9, 11, 4, 4, 3, 8, 11, 7, 12, 1, 2, 2, 1, 12, 3, 3, &
                    1, 1, 10, 5, 5, 9, 9, 6, 6, 3, 8, 8, 8, 12, 1, 7, 10, 10,
10, 8/), (/20, 3/))
c = mfS(xyz, MF_COL, 1)**2 + mfS(xyz, MF_COL, 2)**2 - mfS(xyz,
MF_COL, 3)**2

call msSubplot(1, 2, 1)
h = mfTriSurf(tri1, xyz, c)
call msAxis('equal')
call msSubplot(1, 2, 2)
h = mfTriMesh(tri1, xyz, c)
call msDrawMaterial(h, mf('surf'), mf('visible'), mf('off'))
call msAxis('equal')
call msViewPause()
```

Perform subdivision three times. Draw the polygonal object in each of the iterations.

```
do j = 1, 3
```

```
  do i = 1, 20 * (4**(j-1))
    p1 = mfS(tri1, i, 1)
    p2 = mfS(tri1, i, 2)
    p3 = mfS(tri1, i, 3)
    p1xyz = mfS(xyz, p1, 1) .hc. mfS(xyz, p1, 2) .hc. mfS(xyz, p1, 3)
    p2xyz = mfS(xyz, p2, 1) .hc. mfS(xyz, p2, 2) .hc. mfS(xyz, p2, 3)
    p3xyz = mfS(xyz, p3, 1) .hc. mfS(xyz, p3, 2) .hc. mfS(xyz, p3, 3)

    p12xyz = (p1xyz + p2xyz) / 2
    p23xyz = (p2xyz + p3xyz) / 2
    p13xyz = (p1xyz + p3xyz) / 2

    p12xyz = normalize(p12xyz)
    p23xyz = normalize(p23xyz)
    p13xyz = normalize(p13xyz)

    p12 = p13 + 1
    p23 = p12 + 1
    p13 = p23 + 1

    if (i == 1) then
      tri2 = (p1 .hc. p12 .hc. p13) .vc. &
            (p2 .hc. p23 .hc. p12) .vc. &
                  (p3 .hc. p23 .hc. p13) .vc. &
                  (p12 .hc. p23 .hc. p13)
    else
      tri2 = tri2 .vc. &
            (p1 .hc. p12 .hc. p13) .vc. &
            (p2 .hc. p23 .hc. p12) .vc. &
                  (p3 .hc. p23 .hc. p13) .vc. &
                  (p12 .hc. p23 .hc. p13)
    end if
    xyz = xyz .vc. p12xyz .vc. p23xyz .vc. p13xyz

  end do
  tri1 = tri2
  tri2 = mf()
  c = mfS(xyz, MF_COL, 1)**2 + mfS(xyz, MF_COL, 2) - mfS(xyz,
MF_COL, 3)**2
  call msSubplot(1, 2, 1)
  h = mfTriSurf(tri1, xyz, c)
  call msAxis('equal')
  call msSubplot(1, 2, 2)
  h = mfTriMesh(tri1, xyz, c)
  call msDrawMaterial(h, mf('surf'), mf('visible'), mf('off'))
  call msAxis('equal')
  call msViewPause()
end do
```

The function *normalize* calculates the normalized cross product of two vectors.

```
    contains

      function normalize(v) result(out)
        implicit none

        type (mfArray) :: v, out, d

        call msInitArgs(v)
        out = mfOnes(1, 3)
        d = mfSqrt(mfS(v, 1, 1)*mfS(v, 1, 1) + &
                   mfS(v, 1, 2)*mfS(v, 1, 2) + &
                   mfS(v, 1, 3)*mfS(v, 1, 3))
        call msAssign(mfS(out, 1, 1), mfS(v, 1, 1) / d)
        call msAssign(mfS(out, 1, 2), mfS(v, 1, 2) / d)
        call msAssign(mfS(out, 1, 3), mfS(v, 1, 3) / d)

        call msFreeArgs(v)
        call msReturnArray(out)

      end function normalize

end program example5_6
```

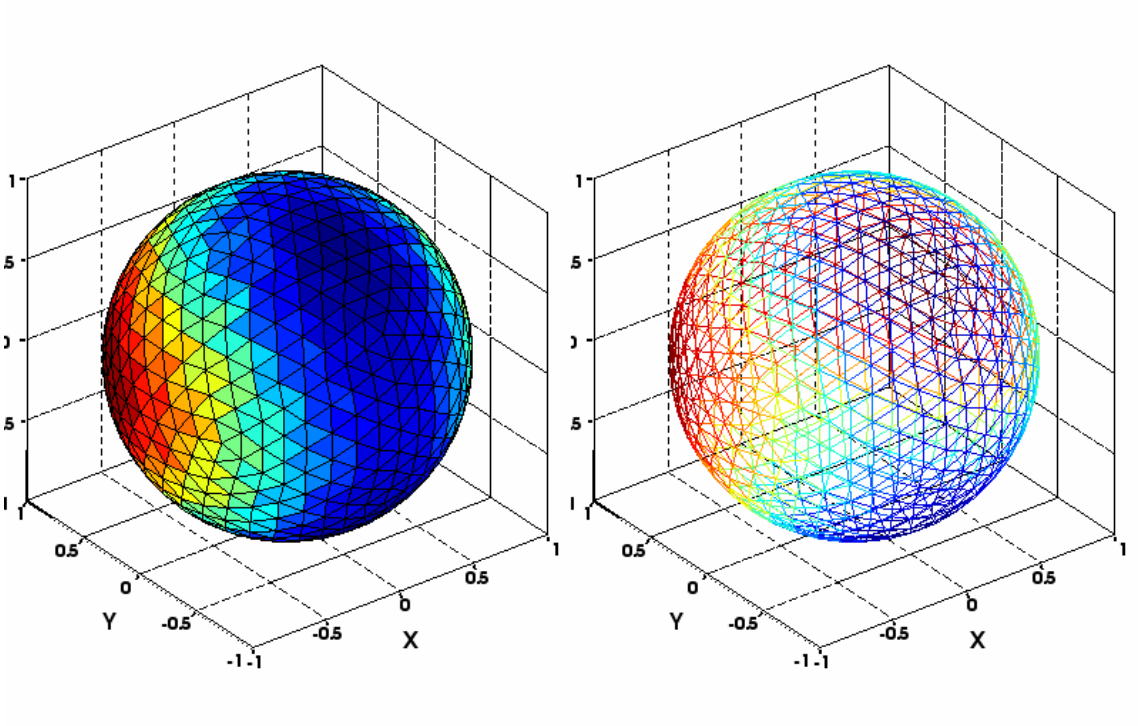


Figure 5.6.1.1 Surface and mesh plot of the polygonal object

5.6.2 Contour

Contour lines are plotted on the constant value line on the surface. The bands between the contour lines are filled with the same color.

Example 5.6.2

Contour representation of the polygonal object.

```
call msSubplot(1, 2, 1)
h = mfTriSurf(tri1, xyz, c)
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'))
call msDrawMaterial(h, mf('surf'), mf('smooth'), mf('on') &
                              , mf('ambient'), mf(50)  &
                              , mf('diffuse'), mf(20) )
call msAxis('equal')
call msSubplot(1, 2, 2)
h = mfTriContour(tri1, xyz, c)
call msAxis('equal')
call msViewPause()
```

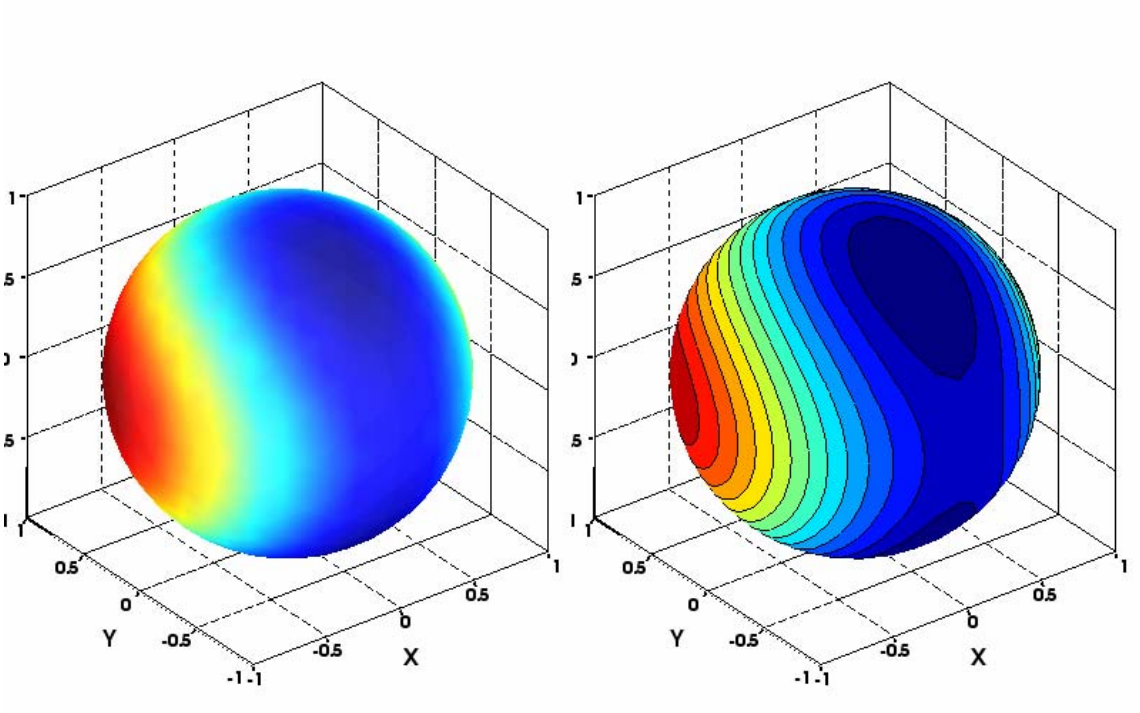


Figure 5.6.2 Contour plot of the polygonal object

5.7 Unstructured Grids

An arbitrary solid model can be represented using an unstructured grid that includes a set of verteces and a cell set.

Verteces are the coordinates in space and cell set specifies the cell connectivity between the verteces.

In MATFOR, unstructured grids can be visualized by using procedure *mfTetSurf*, *mfTetMesh*, *mfTetContour,* and *mfTetIsosurface*.

MATFOR supports four kinds of the cell connectivity representations, as shown in figure 5.7.1:

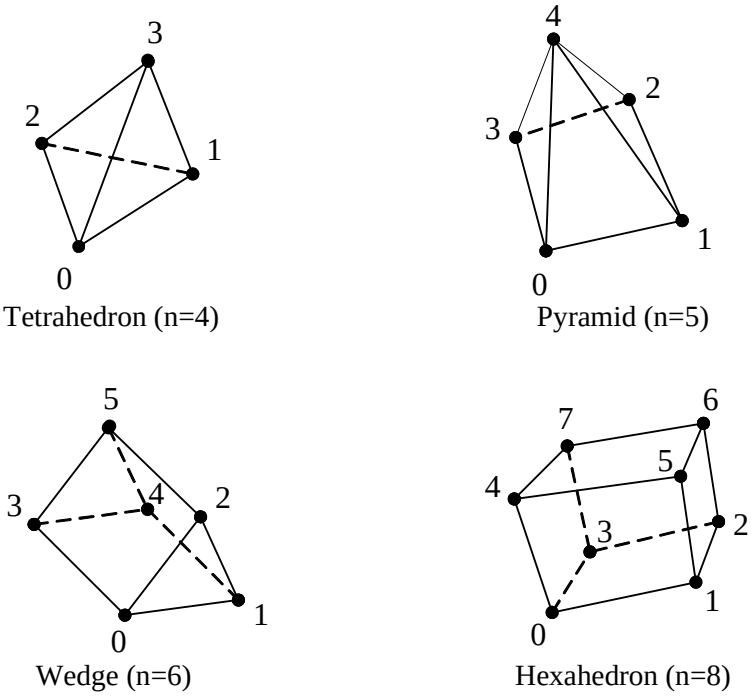


Figure 5.7.1 cell connectivity representations

In this section, we shall demonstrate an example of applying a force on the back of the L-shape steel board, that has a fixed constrain around the hole on the bottom of the steel board.

The force is represented with a cone and the fixed constraint is represented by cylinder.

5.7.1 Surface, Contour, and Iso-surface plots of unstructred grids

Load data from binary files.

```
elem = mfLoad('./data/Lshape_elem.mfb')
node = mfLoad('./data/Lshape_node.mfb')
sxyz = mfLoad('./data/Lshape_sxyz.mfb')

xyz = mfS(node, MF_COL, 1.to.3)
dxyz = mfS(node, MF_COL, 4.to.6) * 10
x = mfS(xyz, MF_COL, 1)
y = mfS(xyz, MF_COL, 2)
z = mfS(xyz, MF_COL, 3)
dx = mfS(dxyz, MF_COL, 1)
dy = mfS(dxyz, MF_COL, 2)
dz = mfS(dxyz, MF_COL, 3)
d_norm = mfSqrt( dx*dx + dy*dy + dz*dz )

sx = mfS(sxyz, MF_COL, 1) * 1.0d-3
sy = mfS(sxyz, MF_COL, 2) * 1.0d-3
sz = mfS(sxyz, MF_COL, 3) * 1.0d-3
sxy = mfS(sxyz, MF_COL, 4) * 1.0d-3
syz = mfS(sxyz, MF_COL, 5) * 1.0d-3
sxz = mfS(sxyz, MF_COL, 6) * 1.0d-3

m1 = mfSize(node, 1)
m2 = mfSize(sxyz, 1)
if (m1>m2) then
    sx = sx .vc. mfZeros(m1-m2, 1)
    sy = sy .vc. mfZeros(m1-m2, 1)
    sz = sz .vc. mfZeros(m1-m2, 1)
    sxy = sxy .vc. mfZeros(m1-m2, 1)
    syz = syz .vc. mfZeros(m1-m2, 1)
    sxz = sxz .vc. mfZeros(m1-m2, 1)
end if

s_norm = mfSqrt( sx*sx + sy*sy + sz*sz )
```

Displays polyhedrons defined by a cell matrix.

```
! ********************************************************************
!  Grid & Mesh
! ********************************************************************
call msFigure('Grid')

! ********************************************************************
!  Grid
```

```
! ********************************************************************
call msSubplot(1, 2, 1)
call msTitle('Grid')

! draw structure element
h = mfTetSurf( elem, x, y, z);
call msDrawMaterial(h, mf('surf'), mf('colormap'), mf('off'))
call msHold('on')

! draw cylinder
h = mfCylinder(mf((/-5.8d0, 0.0d0, 0.25d0/)), mf(0.95d0), mf(1.5d0),
mf((/0, 0, 1/)));

! draw cone
h = mfCone(mf((/1.5d0, 0d0, 5d0/)), mf(0.5d0), mf(2), mf((/0, 0, 1/)));
call msObjOrigin(h, 1.5d0, 0d0, 5d0);
call msObjOrientation(h, 0, -90, 0)

! draw force annotation
h = mfAnnotation(mf("Force=10000NT"), mf((/2.5d0,0d0,5d0/)), mf((/1,
0, 0/)) )
call msGSet(h, mf("offset"), mf((/-50, 10/)))

! draw constraint annotation
h = mfAnnotation(mf("Fix Constraint"), mf((/-5.8d0,0.0d0,1d0/)), mf((/1,
0, 0/)) )
call msGSet(h, mf("offset"), mf((/-50, 10/)))

call msAxis('equal')
call msAxis('off')
call msCamZoom(1.3d0)

! ********************************************************************
!  Mesh
! ********************************************************************
call msSubplot(1, 2, 2)
call msTitle('Mesh')

! draw structure element
h = mfTetSurf( elem, x, y, z);
call msDrawMaterial(h, mf('surf'), mf('colormap'), mf('off'), &
                                                        mf('trans'),
mf(80) )
call msHold('on')

! draw cylinder
h = mfCylinder(mf((/-5.8d0, 0.0d0, 0.25d0/)), mf(0.95d0), mf(1.5d0),
mf((/0, 0, 1/)));

! draw cone
```

```
h = mfCone(mf((/1.5d0, 0d0, 5d0/)), mf(0.5d0), mf(2), mf((/0, 0, 1/)));
call msObjOrigin(h, 1.5d0, 0d0, 5d0);
call msObjOrientation(h, 0, -90, 0)

! draw force annotation
h = mfAnnotation(mf("Force=10000NT"), mf((/2.5d0,0d0,5d0/)), mf((/1,
0, 0/)) )
call msGSet(h, mf("offset"), mf((/-50, 10/)))

! draw constraint annotation
h = mfAnnotation(mf("Fix Constraint"), mf((/-5.8d0,0.0d0,1d0/)), mf((/1,
0, 0/)) )
call msGSet(h, mf("offset"), mf((/-50, 10/)))

call msAxis('equal')
call msAxis('off')
call msCamZoom(1.3d0)
call msViewPause
```

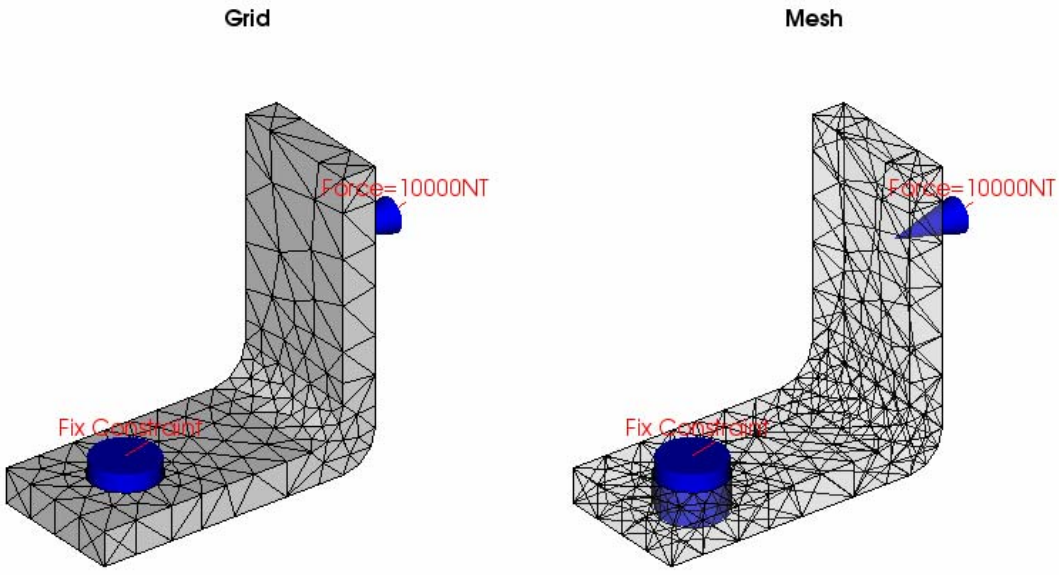


Figure 5.7.1.1 Display the grid and mesh plot of the L-shape steel board

Illustrate the deformation and displacement vectors of the steel board after the force is applied.

```
! ********************************************************************
!  Displacement
! ********************************************************************
call msFigure('Displacement')

! ********************************************************************
!  Deformation
! ********************************************************************
call msSubplot(1, 2, 1)
call msTitle('Deformation')
h = mfTetSurf( elem, x, y, z, d_norm );
call msDrawMaterial(h, mf('surf'), mf('visible'), mf('off'))
call msDrawMaterial(h, mf('edge'), mf('colormap'), mf('on'), &
                                                    mf('trans'),
mf(90) )
call msHold('on')
h1 = mfTetSurf( elem, xyz, d_norm);
call msDrawMaterial(h1, mf('edge'), mf('visible'), mf('off') )

call msColorbar('vert')
call msAxis('equal');
call msAxis('off');
call msAxis(-8,2,-4,4,-2,9)
call msCamZoom(1.05d0)

! ********************************************************************
!  Displacement vector
! ********************************************************************
call msSubplot(1, 2, 2)
call msTitle('Displacement vector')

! draw structure element
h = mfTetSurf( elem, x, y, z, d_norm);
call msDrawMaterial(h, mf('surf'), mf('colormap'), mf('on'), &
                                                    mf('trans'),
mf(90) )
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'), &
                                                    mf('trans'),
mf(90) )
call msHold('on')

h2 = mfQuiver3(x, y, z, dx*0, dy*0, dz*0);
call msDrawMaterial(h2, mf('edge'), mf('colormap'), mf('on') )

call msColorbar('vert')
call msAxis('equal')
call msAxis('off')
call msViewPause
```

```
do i=1,20
    call msGSet(h1, mf('xyz'), xyz + i / 20.0d0 * dxyz)
    call msGSet(h2, mf('udata'), dx*i/20, mf('vdata'), dy*i/20, mf('wdata'),
dz*i/20)
    call msDrawNow
end do

call msViewPause()
```

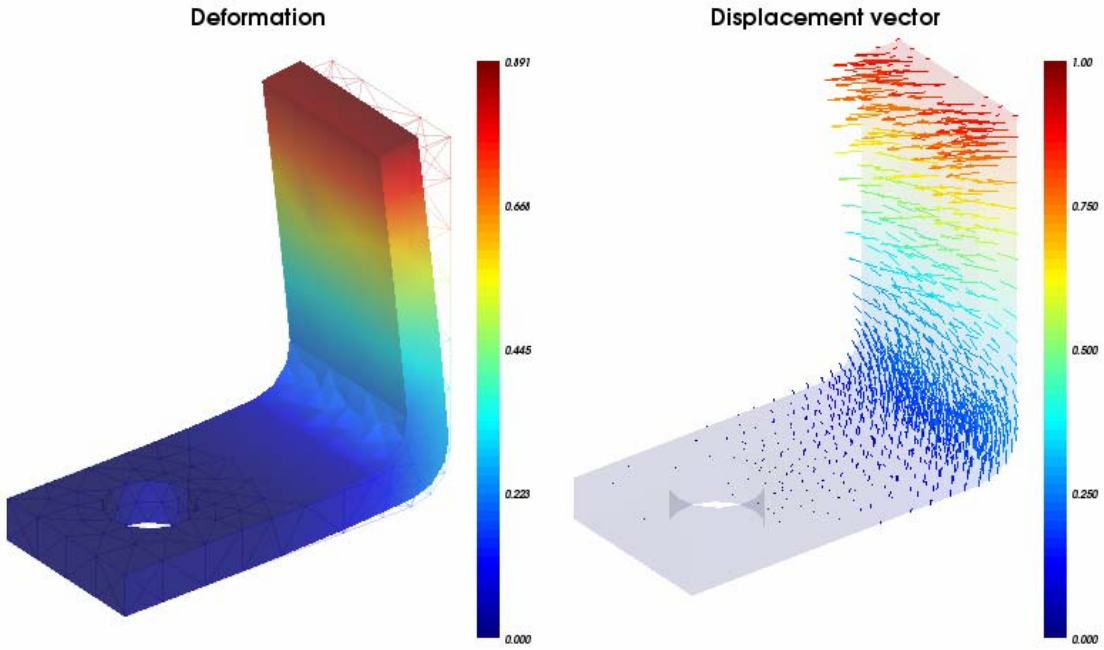


Figure 5.7.1.2 Display the deformation and displacement vector of the L-shape steel board

Display the strength of the shear stress by mapping the graphics object to the current colormap. Notice that the red area represents the deepest impacted area of the shear stress.

We represent the shear stress using vectors as shown in the right-hand figure.

```
! ****************************************************************
```

```
!  Shear stress
! ********************************************************************
call msFigure('Shear stress')

! ********************************************************************
!  Shear value
! ********************************************************************
call msSubplot(1, 2, 1)
call msTitle('Shear stress')
h = mfTetSurf( elem, x+dx, y+dy, z+dz, s_norm);

call msColormapRange(0.0d0, 2d0)
call msAxis('equal');
call msAxis('off');
call msCamZoom(1.05d0)
call msColorbar('vert')

! ********************************************************************
!  Shear vector
! ********************************************************************
call msSubplot(1, 2, 2)
call msTitle('Shear stress vector')

! draw structure element
h = mfTetSurf( elem, x+dx, y+dy, z+dz, d_norm);
call msDrawMaterial(h, mf('surf'), mf('colormap'), mf('off'), &
 mf('trans'), mf(90) )
call msDrawMaterial(h, mf('edge'), mf('visible'), mf('off'), &
 mf('trans'), mf(90) )
call msHold('on')

h1 = mfQuiver3(x+dx, y+dy, z+dz, sx*0, sy*0, sz*0, mf(0.2));
call msDrawMaterial(h1, mf('edge'), mf('colormap'), mf('on') )

call msColormapRange(0.0d0, 2d0)
call msColorbar('vert')
call msAxis('equal')
call msAxis('off')

call msViewPause

do i=1,20
    call msGSet(h1, mf('udata'), sx*i/20, mf('vdata'), sy*i/20, mf('wdata'),
sz*i/20)
    call msDrawNow
end do

call msViewPause
```

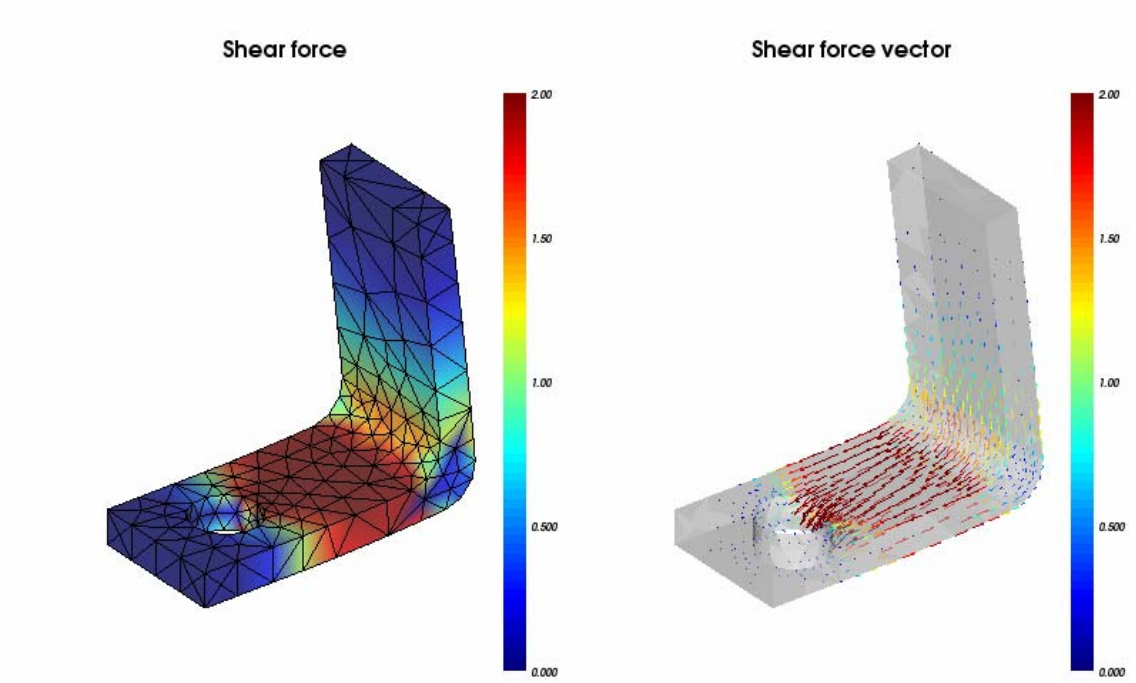


Figure 5.7.1.3 Display the shear force and the shear force vectors of the L-shape steel board

Draw contour on the surface with labels to show the iso-value of each impacted area of the shear stress.

The right-hand figure is similar to the left-hand one, except it only displays the contour lines around each area with its vector values.

```
! ********************************************************************
!  Shear contour
! ********************************************************************
call msFigure('Shear contour')

! ********************************************************************
!  Shear contour
! ********************************************************************
```

```
call msSubplot(1, 2, 1)
call msTitle('Shear force contour')

! draw structure element
h = mfTetContour( elem, x+dx, y+dy, z+dz, s_norm)
call msGSet(h, mf('iso'), mfLinspace(0, 2, 11), mf('label'), mf('on'))
call msDrawMaterial(h, mf('surf'), mf('ambient'), mf(20), &
                                                    mf('diffuse'),
mf(80))

call msColormapRange(0.0d0, 2d0)
call msColorbar('vert')
call msAxis('equal')
call msAxis('off')

! ****************************************************************
!  Shear contour
! ****************************************************************
call msSubplot(1, 2, 2)
call msTitle('Shear force contour + vector')

! draw structure element
h = mfTetContour( elem, x+dx, y+dy, z+dz, s_norm)
call msGSet(h, mf('iso'), mfLinspace(0, 2, 11))
call msDrawMaterial(h, mf('surf'), mf('ambient'), mf(20), &
                            mf('diffuse'), mf(80), &
                            mf('trans'), mf(80) )

call msDrawMaterial(h, mf('edge'), mf('colormap'), mf('on') )

call msHold('on')
h = mfQuiver3(x+dx, y+dy, z+dz, sx, sy, sz, mf(0.2));
call msDrawMaterial(h, mf('edge'), mf('colormap'), mf('on') )


call msColormapRange(0.0d0, 1d0)
call msAxis('equal');
call msAxis('off');
call msCamZoom(1.05d0)
call msColorbar('vert')

call msViewPause()
```

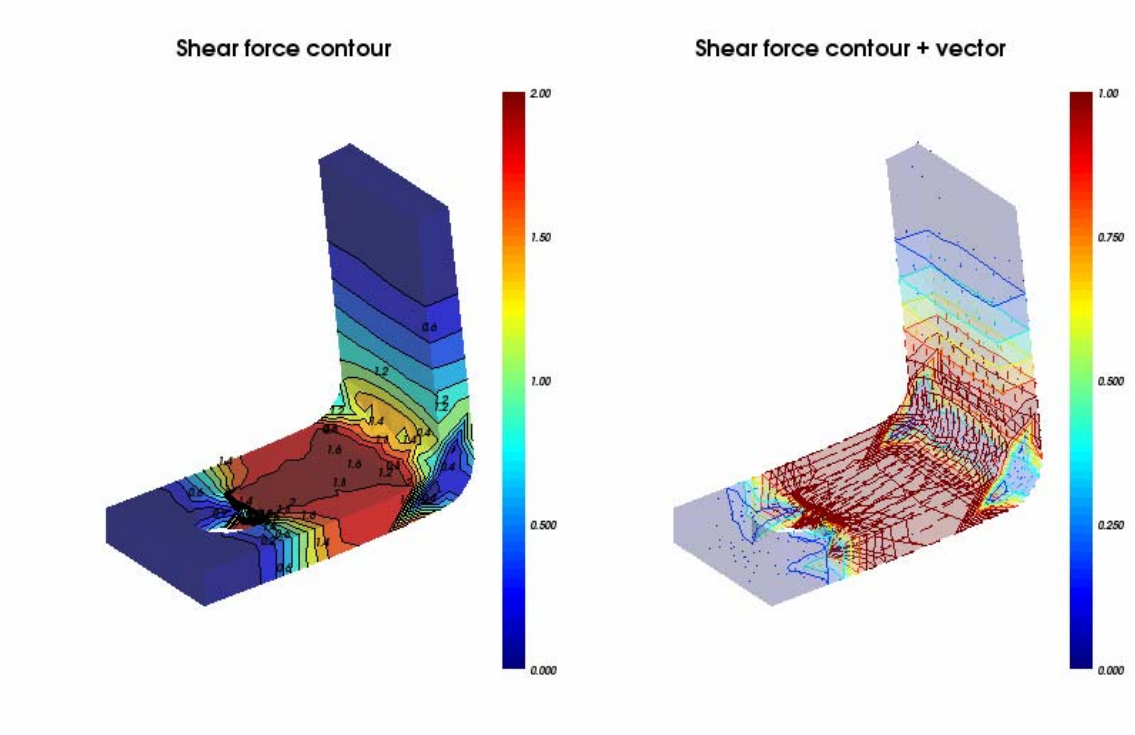


Figure 5.7.1.4 Display the shear force contour and vector of the L-shape steel board

Lastly, show iso-surfaces of the shear stress on slice-planes; six on the left-hand figure and eleven on the right-hand figure.

```
! ********************************************************************
!  Shear iso-surface
! ********************************************************************
call msFigure('Shear iso-surface')

! ********************************************************************
!  Shear iso-surface
! ********************************************************************
call msSubplot(1, 2, 1)
call msTitle('Shear iso-surface')

! draw structure element
h = mfTetContour( elem, x+dx, y+dy, z+dz, s_norm)
call msGSet(h, mf('iso'), mfLinspace(0, 2, 6), mf('label'), mf('on'))
call msDrawMaterial(h, mf('surf'), mf('ambient'), mf(20), &
```

```
                                                        mf('diffuse'),
mf(80), &
                                                        mf('trans'),
mf(80) )

call msDrawMaterial(h, mf('edge'), mf('colormap'), mf('on') )

call msHold('on')
! draw structure element with iso-surfaces of 6 levels
h = mfTetIsoSurface( elem, x+dx, y+dy, z+dz, s_norm, mfLinspace(0, 2,
6))
call msDrawMaterial(h, mf('surf'), mf('ambient'), mf(20), &
                                                        mf('diffuse'),
mf(80))

call msColormapRange(0.0d0, 1d0)
call msColorbar('vert')
call msAxis('equal')
call msAxis('off')

! ********************************************************************
!  Shear iso-surface + contour
! ********************************************************************
call msSubplot(1, 2, 2)
call msTitle('Shear iso-surface')

! draw structure element with contour lines
h = mfTetContour( elem, x+dx, y+dy, z+dz, s_norm)
call msGSet(h, mf('iso'), mfLinspace(0, 2, 11), mf('label'), mf('on'))
call msDrawMaterial(h, mf('surf'), mf('ambient'), mf(20), &
                                                        mf('diffuse'),
mf(80), &
                                                        mf('trans'),
mf(80) )

call msDrawMaterial(h, mf('edge'), mf('colormap'), mf('on') )

call msHold('on')
! draw structure element with iso-surfaces of 11 levels
h = mfTetIsoSurface( elem, x+dx, y+dy, z+dz, s_norm, mfLinspace(0, 2,
11))
call msDrawMaterial(h, mf('surf'), mf('ambient'), mf(20), &
                                                        mf('diffuse'),
mf(80))

call msColormapRange(0d0, 1d0)
call msAxis('equal');
call msAxis('off');
call msCamZoom(1.05d0)
call msColorbar('vert')
```

```
call msViewPause()
```

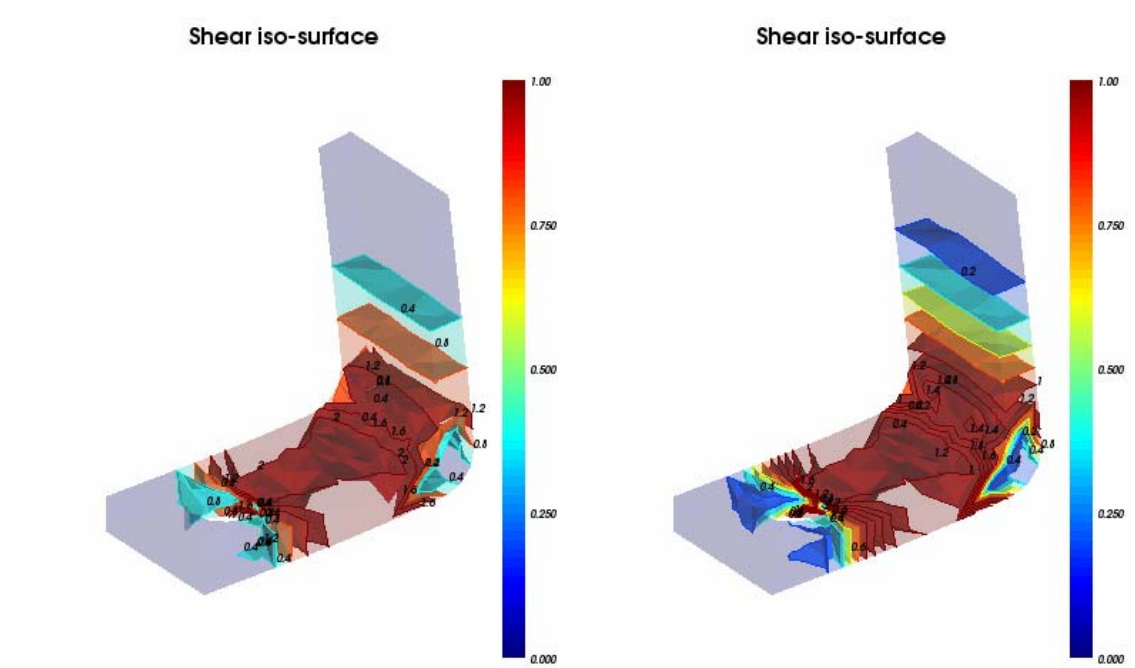


Figure 5.7.1.5 Display the shear iso-surfaces of the L-shape steel board

5.8 Delaunay Triangulation

Delaunay triangulation is performed on a set of input points.

In general, the application of Delaunay triangulation is to create triangles in two-dimensional objects and tetrahedrons in three-dimensional objects from a set of input points.

5.8.1 Two-dimensional Delaunay

Procedure *mfDelaunay* takes a set of input points and plots the triangular mesh.

Procedure *mfGetDelaunay* generates the triangular mesh as output that can be used by *mfTriSurf* or *mfTriMesh* for plotting.

Example 5.8.1

Prompt you to insert the number of points to generate randomly.

```
n = mfInputValue('Input number of random points to generate', 30)
x = mfRand(n, 1) * 10 - 5
y = mfRand(n, 1) * 10 - 5
z = 5 - mfSqrt( x*x + y*y )
```

Next, create a figure window of two subplots. Perform Delaunay triangulation on the points generated above using procedure *mfDelaunay*. To make the points more obvious, we shall plot them on top of the polygons using procedure *mfPlot*.

```
call msFigure('Delaunay 2D')
call msTitle('Delaunay 2D')
call msSubplot(1, 2, 1)
h = mfDelaunay( x, y )
call msHold('on')
h = mfPlot( x, y, 'ob' )
call msAxis('equal')
call msCamZoom(0.7d0)
```

Then, we shall use a different way to perform Delaunay triangulation on the points using procedure *mfGetDelaunay*.

```
call msSubplot(1, 2, 2)
call msTitle('Delaunay 2D surface')
tri = mfGetDelaunay( x, y )
h = mfSphere( x .hc. y .hc. z, mf(0.2), mf((/0, 0, 1/)) )
call msHold('on')
h = mfTriSurf(tri, x, y, z)
call msDrawMaterial(h, mf('surf'), mf('trans'), mf(50), &
                                                    mf('smooth'),
mf('on') )
call msAxis('equal')
call msCamZoom(1.2d0)
```

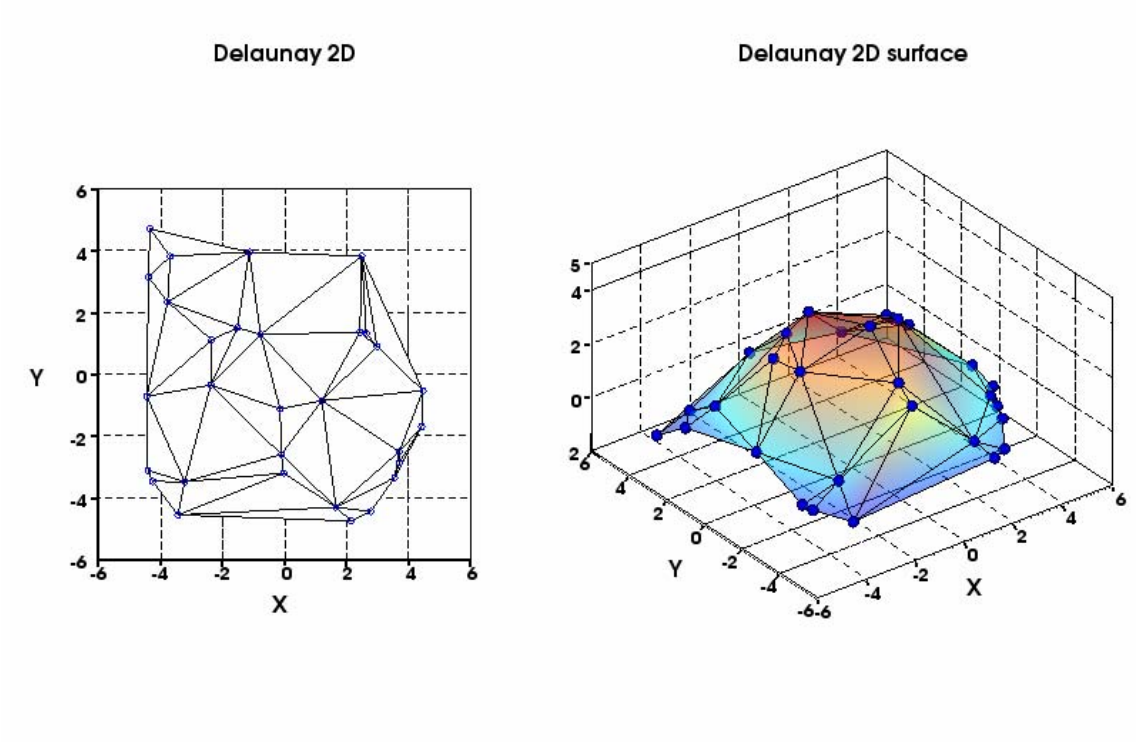


Figure 5.8.1.1 Delaunay 2D and Delaunay 2D surface

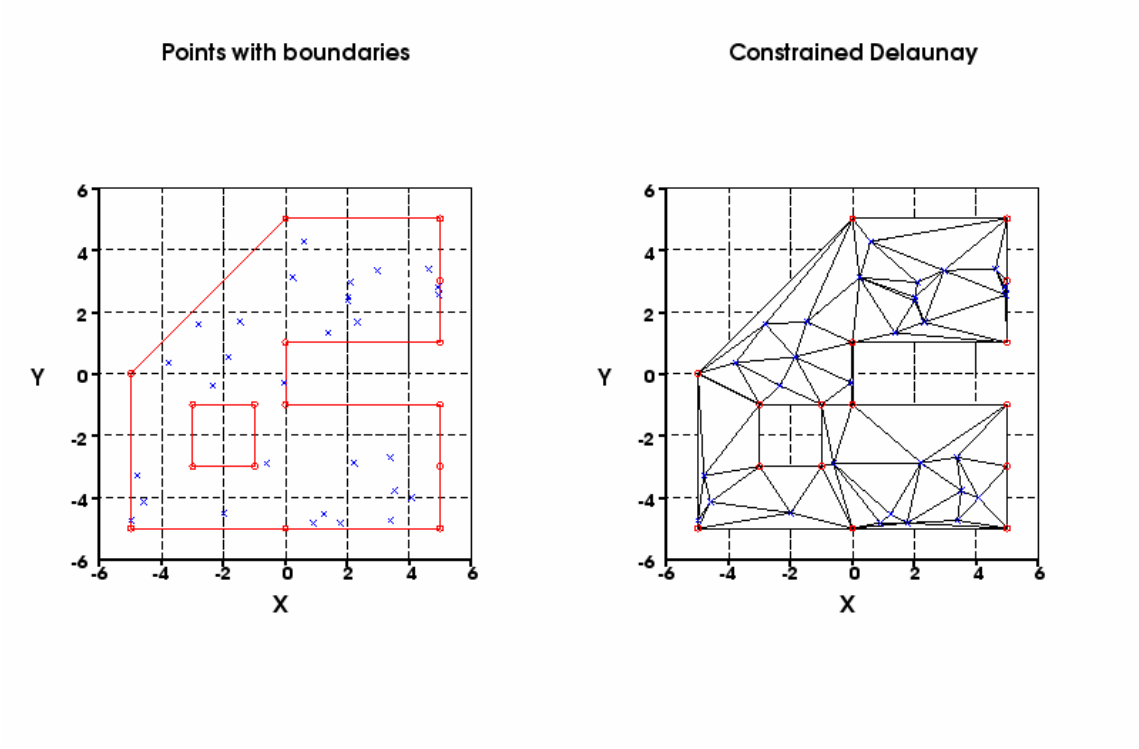


Figure 5.8.1.2 Displays the points within boundary and constrained Delaunay

5.8.2 Three-dimensional Delaunay

Procedure *mfDelaunay3* takes a set of input points and plot the tetrahedron; whereas procedure *mfGetDelaunay3* generates the triangular mesh as output that can be used by *mfTetSurf* or *mfTetMesh* for plotting.

Example 5.8.2

Prompts you to insert the number of points to generate randomly.

```
n = mfInputValue('Input number of random points to generate', 30)
xyz = mfRand(n, 3) * 10 – 5
```

Next, create a figure window of two subplots. Perform three-dimensional Delaunay triangulation on the points generated above using procedure *mfDelaunay3*.

```
call msFigure('Delaunay 3D');
call msSubplot(1, 2, 1)
call msTitle('Distribution')
```

```
h = mfSphere( xyz, mf(0.2), mf((/0, 0, 1/)) );

call msSubplot(1, 2, 2)
call msTitle('Delaunay 3D')
h = mfDelaunay3( xyz );
call msDrawMaterial(h, 'surf', 'trans', 50)
call msHold('on')
h = mfSphere( xyz, mf(0.2), mf((/0, 0, 1/)) );

call msViewPause()
```

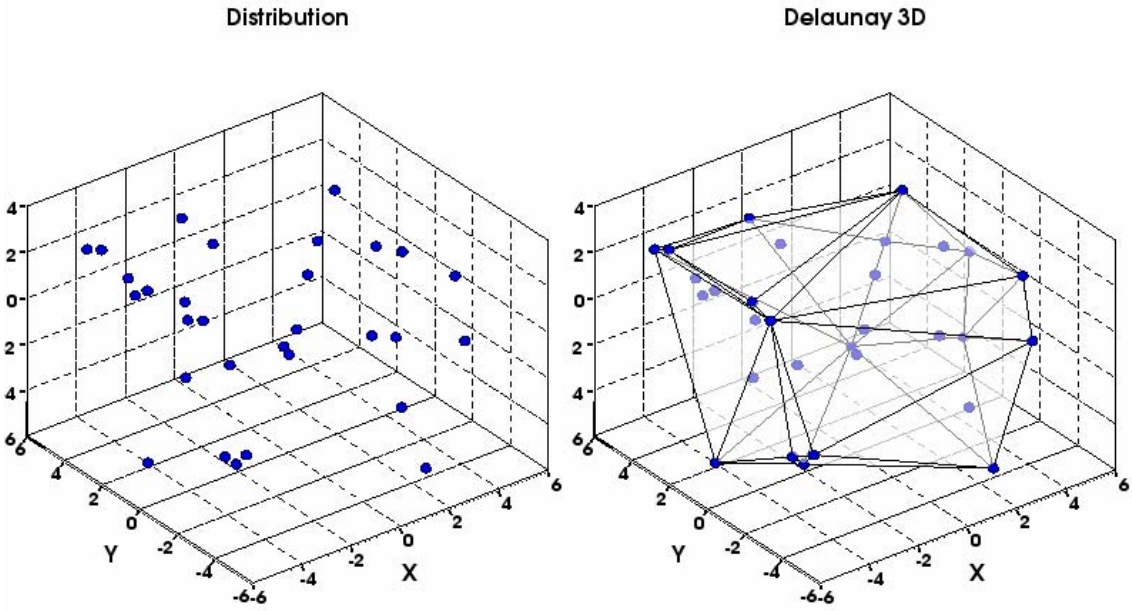


Figure 5.8.2 Display the distributions and delaunay 3D of the points

Index

A

Access Elements and Sections of an mfArray 31

All 53

Any 53

Arithmetic Operators 57

Array Terminology 15

Assignment 68

B

Bounds 15

C

Comparison between Equivalency Procedures 25

Conformance 15

Constructs 66

Create and Initialize mfArray 25

D

Data Viewer 11, 41, 42

Declare an mfArray 26

Determine the optimal binomial 85

Directory Structure 16

Display mfArray Data 37

Documentation and Examples 18

E

Eigenvalues 82

Eigenvectors 82

Element Subscripts 31

Extent 50

F

FGL 9

File I/O 42

FML 9

Function Naming Conventions 13

G

Graphics library 10

Graphics Viewer 10

H

hc 29

I

Initialize an mfArray 26

Inquiry Procedures 47

Installation 16

L

Least Square Operations 84
Logical Inquiry 47
Logical Operations 53

M

MATFOR Parameters 64
Matrix Division 60
Matrix Inverse 77
Matrix Operators and Functions 59
Memory Management 24
mf() 66
MF_COLON 65
MF_E 65
MF_EMPTY 65
MF_EPS 65
MF_I 65
MF_INF 65
MF_NAN 65
MF_PI 65
MF_REALMAX 65
MF_REALMIN 65
mfAll 53
mfAny 53
mfArray 9, 21
mfArray creating procedures 28, 32
mfArray Element Ordering 23
mfArray I/O 37
mfArray Initialization 26
mfArray Intrinsic Data Type 22
mfArray Operators 56
mfArray Syntax and Expressions 24
mfColon 29, 57
mfCone 157
mfCube 157
mfCylinder 157
mfEig 82
mfEquiv 25, 71
mfEye 28
mfFind 53
mfInv 77
mfIsComplex 48
mfIsEmpty 48
mfIsLogical 48
mfIsNumeric 48
mfIsReal 48
mfLDiv 57
mfLength 50
mfLinspace 29
mfLoad.m 45
mfLoadAscii 44
mfMagic 29
mfMeshgrid 29
mfMul 57
mfNDims 50

mfOnes 29
mfPlayer 11
mfRand 29
mfRDiv 57
mfRepmat 29
mfReshape 29
mfSave.m 45
mfSize 50
mfSphere 157
mfZeros 29
Mix mfArray and Fortran arrays 25
mod_datafun 9
mod_elfun 9
mod_elmat 9
mod_ess 9
mod_fileio 9
mod_matfun 9
mod_ops 9
msAssign 25
msCone 157
msCube 157
msCylinder 157
msDisplay 37, 40
msEye 28
msFind 53
msFreeArgs 24, 75
msGDisplay 39
msInitArgs 24, 75
msLinspace 29
msMagic 29
msMeshgrid 29, 100
msOnes 29
msPointer 25, 69
msRand 29
msRecordStart 127
msRepmat 29
msReshape 29
msRetrunArray 24
msReturnArray 75
msSaveAscii 43
msSphere 157
msSubplot 96
msZeros 29

N

Numerical Library 10

O

Operators Precedence 63

P

Procedures with “mf” as prefix 13
Procedures with “ms” as prefix 13
Program with mfArray 65
Project Settings 17

R

Rank 15

Regression Model 78

Relational Operators 58

S

Shape 15, 50

Size 15, 50

Structure of the mfArray 21

Subscript 34

T

Technical Support 18

U

Upgrade MATFOR 17

Use Fortran Arrays as input to MATFOR Procedures 71

Use mfArray as Input Dummy Arguments 74

Use mfArray as input to Fortran Procedures. 68

Use mfArray as Output Dummy Arguments.. 75

V

vc 29

visualization 91, 93