Contents

Introduction.. 5

1.1 Overview 5

1.2 What is MATFOR...6

1.3 The MATFOR Components 6

1.4 MATFOR Function Naming Conventions 9

1.5 Array Terminology 11

1.6 MATFOR Installation 12

1.7 MATFOR Documentation and Examples 13

1.8 Technical Support 13

Working with mfArray....... 15

2.1 What is mfArray 15

2.2 Create and Initialize mfArray 16

2.2.1 Declaring an mfArray 16

2.2.2 Initializing an mfArray 20

2.3 Access Elements and Sections of an mfArray 28

2.3.1 Element Subscripts 28

2.3.2 mfArray Accessing Methods 30

2.4 mfArray I/O 36

2.4.1 Displaying mfArray Data 37

2.4.2 mfArray File I/O 40

2.5 mfArray Inquiry Functions 45

2.5.1 Logical Inquiry...46

2.5.2 Size, Shape, and Extent 48

2.5.3 Logical Operations 50

2.6 mfArray Operators 55

2.6.1 Arithmetic Operators 57

2.6.2 Relational Operators 58

2.6.3 Matrix Operators and Functions 60

2.6.4 MATFOR Parameters 66

Linear Algebra 69

3.1 Matrix Inverse 69

3.2 Application of Eigenvalues and Eigenvectors 73

3.3 Least Square Operations 76

Visualization Basics 81

4.1 Plotting Your Data 81

4.2 MATFOR Graphics Viewer 84

4.2.1 Window Frame and Figure Windows 84

4.2.2 Subplots 85

4.2.3 Menu and Toolbar 87

4.3 Creating 3-D Models 89

4.3.1 Generating the Data 89

4.3.2 Loading Data (mfb, ascii) 92

4.4 Displaying 3-D Objects 94

4.4.1 Adjusting the Viewpoint 94

4.4.2 Shifting the Objects 95

4.4.3 Rescaling the Objects 97

4.4.4 Changing the Displaying Mode 98

4.4.5 Setting the Axis Object 98

4.5 Colormap, Shading and Texture 100

4.5.1 Adjusting Colormap 100

4.5.2 Displaying Colorbar 104

4.5.3 Shading Objects 105

4.5.4 Mapping Texture 109
4.6 Annotating Your Graph 113
4.6.1 Setting the Title and Axis Labels 113
4.6.2 Text Annotation 114
4.7 Animation and Recording 115
4.7.1 Animation 116
4.7.2 Recording your animation 118
4.7.3 Image Exporting 120
4.8 MATFOR Data Viewer 120
4.8.1 Matrix Table 121
4.8.2 Menu 123
4.8.3 Toolbar 124
4.8.4 Sampling Type 125
4.8.5 Snapshot Panel 125
4.8.6 Analysis Panel 127
4.8.7 Filter Panel 127
4.8.8 Status Bar 128

Visualization Methods........ 131

5.1 Linear Graph 131
5.1.1 Two-dimensional Linear Graph 131
5.1.2 Three-dimensional Linear Graph 133
5.2 Surface Plot 136
5.2.1 Surface Plot 136
5.2.2 Contour Plot 138
5.2.3 Pseudocolor Plot 140
5.3 Volume Rendering 142
5.3.1 Surface (surf, mesh, outline, contour) 143

5.3.2 Sliced-planes 145

5.3.3 Isosurface 146

5.4 Vector Field 148

5.4.1 Quiver and Streamline 148

5.5 Elementary 3-D Objects 149

5.5.1 Primitives 149

5.5.2 Molecule 150

5.6 Unstructured Mesh 152

5.6.1 Surface 153

5.6.2 Contour 156

5.7 Unstructured Grids 157

5.7.1 Surface, Contour and Iso-surface plots of unstructured grids 158

5.8 Delaunay Triangulation 169

5.8.1 Two-dimensional Delaunay 169

5.8.2 Three-dimensional Delaunay 172

Index 175

Introduction

1

1.1 Overview

This guide is written as an introduction to users who are new to MATFOR in C++ — a scientific and visualization library for Scientists and Engineers. In this guide, the MATFOR foundation array — mfArray is introduced and discussed with some depths in Chapter 2. This is followed by the application of MATFOR in linear algebra in Chapter 3 and then with the introduction of visualization basics in Chapter 4. Chapter 5 describes the categories of the graphical functions. 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, Conventions, Documentation, and Licensing.

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

Chapter 3. Linear Algebra, highlights the linear algebra functions 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 MATFOR Graphical functions 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 numerical and visualization libraries developed specially for scientists and engineers. The functions in the libraries enhance your C++ program with dynamic visualization capability, shorten your numerical codes, and speed up your development process.

By adding a few lines of MATFOR function calls to your C++ program, you can easily visualize your computing results, perform run-time animations, or even produce a movie presentation file as you execute your program.

You also have the choice to record an animation as a MATFOR *.mfa* file, and view it later using MATFOR mfPlayer.

Debugging is facilitated with 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 functions are designed to be intuitive and simple to use. Using the numerical functions, 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 traditionally in C++.

1.3 The MATFOR Components

The standard edition of MATFOR consists of five main components, namely MATFOR mfArray, MATFOR Numerical Library (named as `cml`), MATFOR Graphics Library (named as `fgl`), MATFOR Graphics Viewer, and MATFOR Data Viewer.

MATFOR functions are packaged into two main modules – the `cml` and `fgl` modules. The `cml` module contains the numerical functions while the `fgl` module contains the graphical functions. MATFOR mfArray is included in both modules. The Graphics Viewer and Data Viewer are included in the `fgl` module.

Numerical functions included in `cml` are further categorized into several groups such as the Essential Set of MATFOR Routines, Elementary Matrix-manipulating Functions, Matrix Functions, Matrix and Array Manipulation Operators and Functions, Data Manipulation Functions, and Elementary Math Functions.

MATFOR mfArray

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

```
mfArray x;
```

The data type and dimensions of the mfArray are determined internally by MATFOR when you initialize it. It is so flexible that you can assign your C++ array to an mfArray, and visualize your array data using MATFOR graphical functions, without having to consider the dimension or data type of your original C++ array.

MATFOR mfArray enables you to write programs with the ease of interactive language in the C++ programming environment. With mfArray, powerful routines from the likes of LAPACK are packaged into simple and intuitive functions. For example, the original function `DGESVX` for calculating Singular Value Decomposition in LAPACK requires twenty-two individual arguments that are of different data type and array dimensions. In MATFOR, the same function is reduced to three arguments of the same data type – the 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 functions, ranging from inquiry functions such as `mfShape`, `mfSize`, `mfIsLogical`, and elementary mathematical functions such as `mfSin`, `mfCos`, and complex arithmetic, to sophisticated functions like eigenvalues, `LU` decomposition, matrix inverse and conditioning functions. Most functions use mfArray as the input and output argument.

MATFOR Graphics Library

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

MATFOR Graphics Viewer

The Graphics Viewer is a window for displaying your graphs on the screen. The Graphics Viewer provides menu and toolbar functions for editing and debugging.

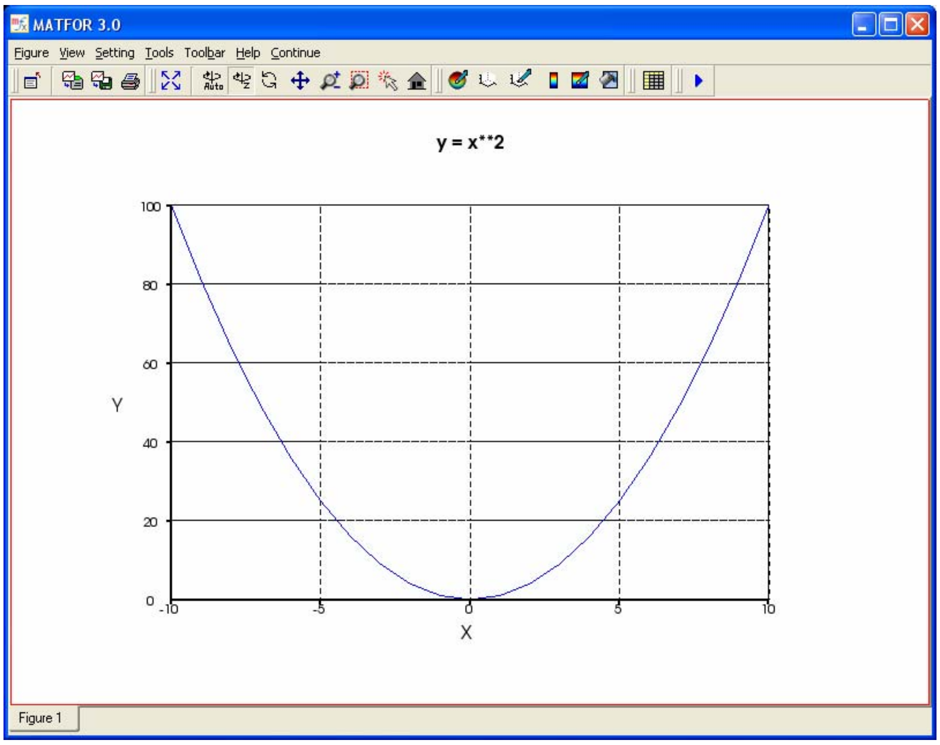
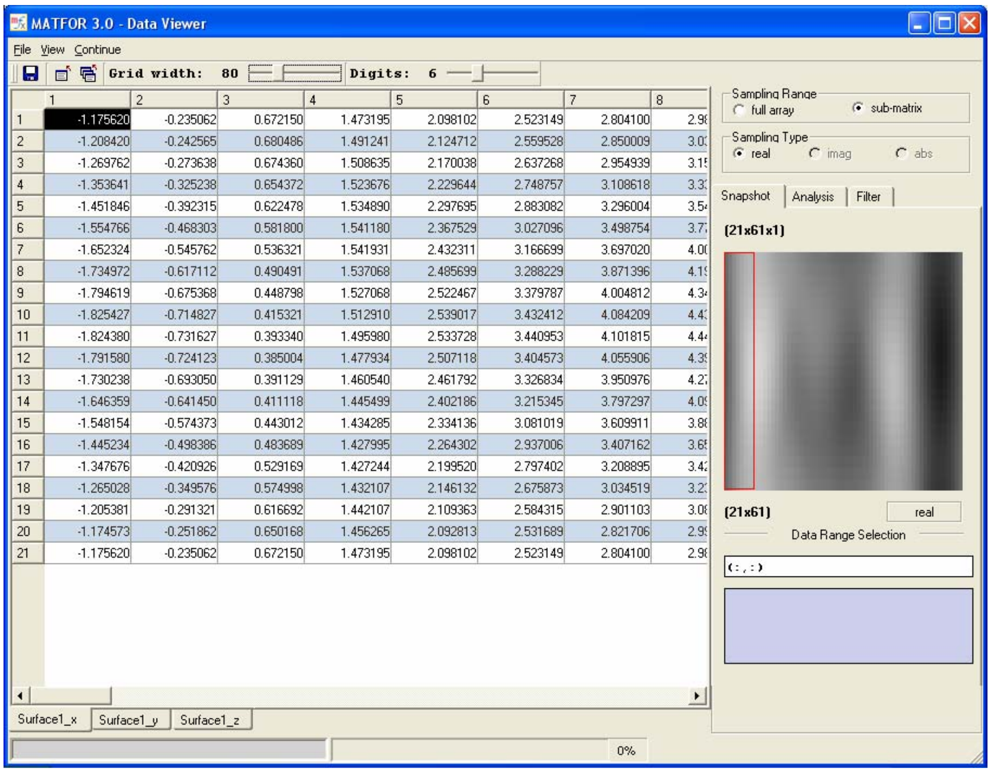


Figure 1.3.1 MATFOR Graphics Viewer

MATFOR Data Viewer

MATFOR Data Viewer (Figure 1.3.2) is a spreadsheet-like window 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 it with conditions specified using mathematical expressions.



Grid width: 80 Digits: 6

Sampling Range: full array [✓ sub-matrix]

Sampling Type: [✓ real] imag abs

	1	2	3	4	5	6	7	8
1	-1.175620	-0.235062	0.672150	1.473195	2.098102	2.523149	2.804100	2.9
2	-1.208420	-0.242565	0.680486	1.491241	2.124712	2.559528	2.850009	3.0
3	-1.269762	-0.273638	0.674360	1.508635	2.170038	2.637268	2.954939	3.1
4	-1.353641	-0.325238	0.654372	1.523676	2.229644	2.748757	3.108618	3.3
5	-1.451846	-0.392315	0.622478	1.534890	2.297695	2.883082	3.296004	3.5
6	-1.554766	-0.468303	0.581800	1.541180	2.367529	3.027096	3.498754	3.7
7	-1.652324	-0.545762	0.536321	1.541931	2.432311	3.166699	3.697020	4.0
8	-1.734972	-0.617112	0.490491	1.537068	2.485699	3.288229	3.871396	4.1
9	-1.794619	-0.675368	0.448798	1.527068	2.522467	3.379787	4.004812	4.3
10	-1.825427	-0.714827	0.415321	1.512910	2.539017	3.432412	4.084209	4.4
11	-1.824380	-0.731627	0.393340	1.495980	2.533728	3.440953	4.101815	4.4
12	-1.791580	-0.724123	0.385004	1.477934	2.507118	3.404573	4.055906	4.3
13	-1.730238	-0.693050	0.391129	1.460540	2.461792	3.326834	3.950976	4.2
14	-1.646359	-0.641450	0.411118	1.445499	2.402186	3.215345	3.797297	4.0
15	-1.548154	-0.574373	0.443012	1.434285	2.334136	3.081019	3.609911	3.8
16	-1.445234	-0.498386	0.483689	1.427995	2.264302	2.937006	3.407162	3.6
17	-1.347676	-0.420926	0.529169	1.427244	2.199520	2.797402	3.208895	3.4
18	-1.265028	-0.349576	0.574998	1.432107	2.146132	2.675873	3.034519	3.2
19	-1.205381	-0.291321	0.616692	1.442107	2.109363	2.584315	2.901103	3.0
20	-1.174573	-0.251862	0.650168	1.456265	2.092813	2.531689	2.821706	2.9
21	-1.175620	-0.235062	0.672150	1.473195	2.098102	2.523149	2.804100	2.9

Figure 1.3.2 MATFOR Data Viewer

1.4 MATFOR Function Naming Conventions

MATFOR functions are all characterized by prefixing with an “`mf`”. By default, MATFOR functions use the mfArray as input and output arguments. In special cases, functions may also accept C++ data types as input and output arguments. Refer to the **MATFOR Reference Guide** for more detailed documentation regarding individual functions and their input and output argument types.

Functions with "mf" as prefix

MATFOR functions generally adopt the following function prototypes.

```
mfArray mfFunction(mfArray x, ...);

void mfFunction(mfOutArray OutArray, mfArray
x, ...);
```

The first function prototype returns an mfArray as a function output. For example,

```
y = mfSin(x).
```

The second function prototype introduced does not return a typical function output, but introduces a new MATFOR class — the mfOutArray class. A variable declared as type *mfOutArray* is recognized by MATFOR internally as a function's return argument. This was designed to handle functions with multiple outputs conveniently. Function `mfOut` is typically used to convert a list of mfArrays to the mfOutArray class. For example, the LU decomposition function mfLu returns two variables `l` and `u`, containing the lower and upper triangular matrix of the input mfArray respectively. This is implemented as,

```
mfLu(mfOut(l, u), a);
```

The statements below list some typical MATFOR function calls.

```
mfViewPause();
mfSurf(x, y, z);
mfSubplot(2, 2, 1);
mfCos(mfOut(y), x);
mfLU(mfOut(l, u), a);
mfMeshgrid(mfOut(a, b), m, n);
```

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	lengths of dimensions
conformance	two arrays are of the same shape

Example

Arrays `a` and `b` with the following values,

```
mfArray a(3,3),b(1,5);
```

are described by the following:

	ma	mb
rank	2	2
bounds	[1,3] and [1,3]	[1,1] and [1,5]
extent	[3,3]	[1,5]
size	9	5
shape	[3,3]	[1,5]
conformance	a and b does 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 Visual Studio project configuration. In some cases, manual configuration is required. More details on using the installation package are available in the **Installation Guide** that comes with the installation package.

MATFOR Directory Structure

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

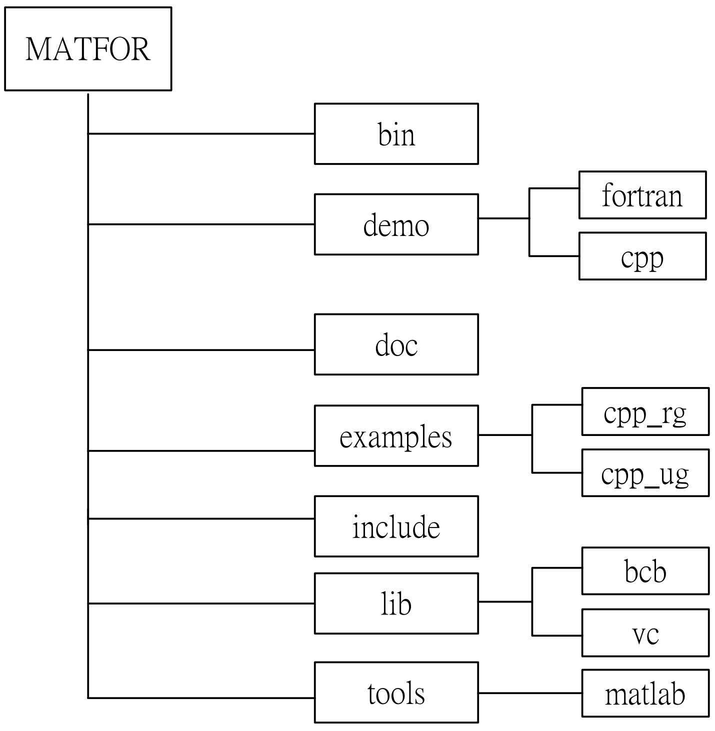


Figure 1.6.1 MATFOR installed directory structure

By default, the installer creates a program group in your C drive under the path `C:\Program Files\AnCAD\MATFOR3\`. We shall use <MATFOR> to represent your installation location henceforth.

The common utilities include collateral program, tools, redistributables, such as MATFOR CA, dynamically linked libraries, etc. The library (e.g. MATFOR in C++ Library) specifics are mainly components such as C++ header file and import library `.lib` files, and the associated examples and documentations.

1.7 MATFOR Documentation and Examples

MATFOR C++ Library has two main documentations, namely **MATFOR in C++ User's Guide**, and **MATFOR in C++ Reference Guide**. The documents are available in Acrobat Reader pdf format.

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

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

```
<MATFOR>\examples\cpp_ug\
```

The `*.cpp` source files are named following their labels in the User's Guide. For example, Example 2.1.6.1 would be saved as `Example2_1_6_1.cpp.` In some 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

For more information about MATFOR 3 in C++, please visit our website http://www.ancad.com. If you have further questions or doubts, you may directly write to support@ancad.com or post them on the online forum at http://ww.ancad.com/servicecounter/onlineforum.html. Our technical support staff will be glad to service you.

Working with mfArray

2.1 What is mfArray

MATFOR mfArray is an advanced dynamic array class that supports automatic data typing and dimensioning. You only have to write,

```
mfArray a
a = 2.0
```

at the variable declaration section and initialize it in order to use an mfArray. The size, shape, and data type of the mfArray is automatically determined when you initialize it. In the example above, mfArray a automatically assumes the shape of 1-by-1, storing double precision real value 2.0.

The mfArray class public interface consists of constructors, a destructor, indexing function and operator, array and matrix operator functions, and member functions.

Using a combination of the mfArray operators and subscript functions, you can perform whole array operations or element-by-element operations.

In this chapter, we will cover the mfArray class, look at its constructors and special array creation functions, its subscript methods, its operators, its I/O interface, and its member functions. We will use examples to help facilitate understanding. At the same time, we will be using functions from other libraries in MATFOR in our examples.

2.2 Create and Initialize mfArray

This section provides a first step on using mfArray in your C++ program. We will be covering the mfArray constructors and special array creation functions.

2.2.1 Declaring an mfArray

The mfArray class public interface contains many useful constructors. You can construct an mfArray using string, double, float, and Booleans. MATFOR stores all data in double format.

Table 2.2.1 mfArray constructors

Constructor	Creates	Example
mfArray();	Uninitialized array	mfArray a;

mfArray(const mfArray & rhs);	Copy of input data	mfArray a = 10;
		mfArray b(a);

mfArray(int row, int col, int depth, …, int idx7);	Construct mfArray whose shape is specified by row, col, depth, idx4, idx5, idx6 and idx7. By default, the array is not initialized.	mfArray a(2,3);
static mfArray mfComplexArray (int row, int col, int depth, …, int idx7);	Construct mfArray of complex values whose shape is specified by row, col, depth, idx4, idx5, idx6 and idx7. By default, the array is not initialized.	mfArray a = mfArray::mfComplexArray(2,3);
mfArray(const char* str);	Construct mfArray from a string.	mfArray a(“string”)
mfArray(int value)	Construct mfArray from an integer value.	mfArray a(2);
mfArray(double value)	Construct mfArray from a double value.	mfArray a(2.5);
mfArray(dcomplex value)	Construct mfArray from a	dcomplex q(2.0,3.0);

Constructor	Creates	Example
	complex value.	mfArray a(q);
mfArray(bool value)	Construct mfArray from a boolean value.	mfArray a(true);
mfArray(const double* p, int idx1, int idx2, …,int idx7);	Construct mfArray from a double pointer and repack the data into shape specified by row, col, depth, idx4, idx5, idx6 and idx7.	double e[3]={1.0,2.0,3.0}; mfArray f(e,3,1);
mfArray(const float* p, int idx1, int idx2, …,int idx7);	Construct mfArray from a float pointer and repack the data into shape specified by row, int, depth, idx4, idx5, idx6, and idx7.	float e[3]={1.0,2.0,3.0}; mfArray f(e,3,1);
mfArray(const dcomplex* p, int idx1, int idx2, …,int idx7);	Construct mfArray from a complex pointer and repack the data into shape specified by row, int, depth, idx4, idx5, idx6, and idx7.	dcomplex c[2] = { dcomplex(1.0,2.0),dcomplex(3.0, 4.0) }; mfArray f(c, 1,2);
mfArray(const bool* p, int idx1, int idx2, …,int idx7);	Construct mfArray from a boolean pointer and repack the data into shape specified by row, int, depth, idx4, idx5, idx6, and idx7.	bool b[3] = {true,true,false}; mfArray f(b, 3, 1);

Example 2.2.1 Construct and initialize mfArray

Next, we shall go through an example below to construct and initialize an mfArray using one of its constructors.

Step 1. Start a new program and save it as `Example2_2_1.cpp.` Add the preprocessor directives to include the header file “`cml.h`” in your function.

```
#include “cml.h”
int main()
```

```
{
```

Step 2. Next we shall construct an uninitialized mfArray `a` and initialize it using the assignment operator to a scalar. You can initialize to a scalar of type double, complex, float, and integer. MATFOR mfArray stores all data in double format.

```
mfArray a;
a = 2.0;
```

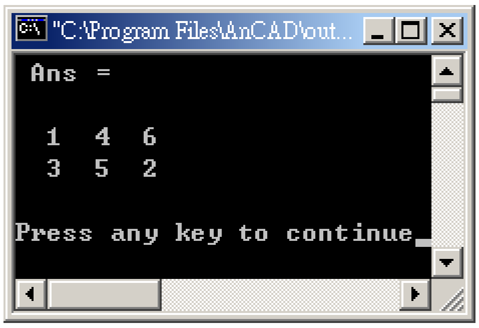
Step 3. That was one of the more common construction and initialization while using MATFOR. We shall now try to construct and initialize an mfArray using an existing double array. You can initialize the mfArray with double, integer, float, and complex arrays of up to seven in dimension.

We shall first create a 6-element double array and use it to construct a 2-by-3 matrix mfArray.

```
double input[6]={1.0,3.0,4.0,5.0,6.0,2.0};
mfArray b(input, 2,3);
```

Step 4. You can display the content of an mfArray on the console using mfArray function – `mfDisplay()`. Display the content of the mfArray to see how the elements have been rearranged.

```
mfDisplay(b);
```



```
Ans =

  1   4   6
  3   5   2

Press any key to continue
```

Figure 2.2.1.1 Repacked mfArray

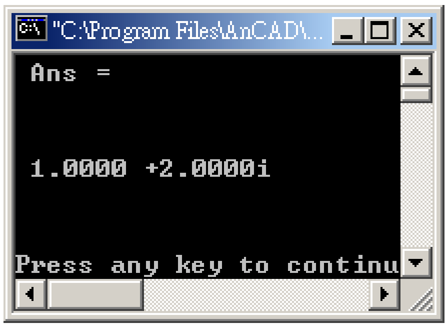
Figure 2.2.1.1 shows the displayed mfArray `b.` Notice that function `mfDisplay()` displays the elements as integers although they stored in double precision format. The function displays data in integer format when only zeroes are trailing behind the decimal place to make the presentation neater.

Also note that the data from array `input` has been repacked in column-major format in the mfArray, first filling the columns then followed by the rows. This is different from the typical row-major arrangement of C++ arrays, but follows the convention used in Fortran. More discussions about the element ordering in mfArray will be covered in Section 2.3.

Step 5. The complex data type is useful in many scientific computations where the result of a computation might lie in the complex domain. MATFOR functions and mfArray class support the complex type.

Next, we shall construct and initialize an mfArray using a complex variable.

```
dcomplex c(1.0, 2.0);
mfArray d(c);
mfDisplay(d);
```



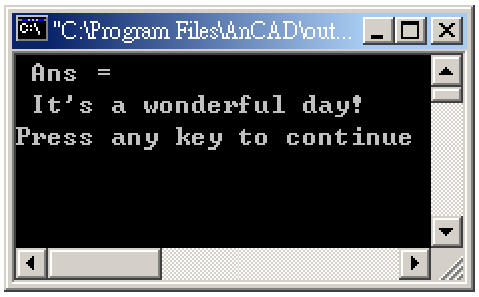
```
Ans =

 1.0000 +2.0000i

Press any key to continu
```

Figure 2.2.1.2 Complex mfArray

Figure 2.2.1.2 shows the displayed complex mfArray. The complex values are displayed in the mathematical format of a + bi.

Step 6. You can use mfArray to store character strings without having to worry about the allocated size of the mfArray. As an example, construct an mfArray using a string, say, "It's a wonderful day!" and display it in the console.

```
mfArray mfg("It's a wonderful day!");
mfDisplay(mfg);
```



```
 Ans  =
 It's a wonderful day!
Press any key to continue
```

Figure 2.2.1.3 mfArray containing string

In the example, we used the `mfDisplay()` function to display content of an mfArray. By default the `mfDisplay()` function displays double data in `"short"` format, i.e. up to four digits. You can display the full double data by specifying `"long"` format through function `mfFormat("long").` Remember to add the function call before using the `mfDisplay()` function.

Here we end our exercise with mfArray constructors. Next, in Section 2.2.2, we shall look at a few of the special array creating functions.

2.2.2 Initializing an mfArray

MATFOR contains a set of special array creating functions for quick creation of mfArray. Table 2.2.2 below lists some of the mfArray creating functions available.

Table 2.2.2 mfArray creating functions

Essential Set of MATFOR routines	
mfHCat	Horizontally concatenate arrays.
mfVCat	Vertically concatenate arrays.
mfColon	Create vectors of specified increment.
mfCreateVector, mfV	Create an mfArray 1-by-n vector from list of data.
mfCreateMatrix,	Create an mfArray of specified shape from list of

Essential Set of MATFOR routines	
mfM	data.
Elementary matrix-manipulating functions	
mfEye	Create identity arrays.
mfLinspace	Create a linearly spaced vector with specified number of points.
mfMagic	Create a magic matrix of equal column, row and diagonal sums.
mfMeshgrid	Create matrices from vectors for functions of two variables and three-dimensional figure.
mfOnes	Create arrays containing all ones.
mfRand	Create arrays containing random numbers.
mfRepmat	Create an array by tiling smaller arrays.
mfReshape	Repack vectors into specified array shapes.
mfZeros	Create arrays containing all zeros.

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 mfArray creating functions

To get you familiar with the functions, we shall use some of the array creating functions to create mfArrays.

Step 1. First create a new program and save it as `Example2_2_2.cpp`. Include the header files in your preprocessor directives.

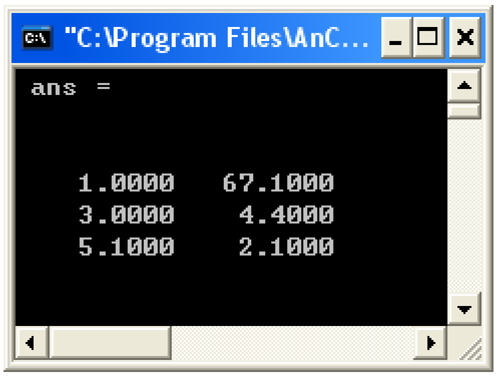
Step 2. Function `mfCreateMatrix` (shorthand as `mfM`), creates two-dimensional mfArray of specified shape from a list of double data. The function has the following prototype.

```
mfArray mfCreateMatrix(int row, int col, double
data, ...);
```

The first two arguments are integers specifying the shape of the mfArray, i.e. the numbers of rows and columns. The trailing arguments are the data to be stored in the elements of the mfArray. The data must be double or dcomplex and should fill the created mfArray. The data are packed column-wise, following the MATFOR convention, filling the elements column-by-column.

Now, to get a feel of the function, we shall create a 3-by-2 mfArray.

```
mfArray a = mfM(3, 2, 1.0, 3.0, 5.1, 67.1, 4.4, 2.1);
mfDisplay(a);
```



```
ans =


   1.0000   67.1000
   3.0000    4.4000
   5.1000    2.1000
```

Figure 2.2.2.1 mfArray created using function mfCreateMatrix (shorthand as mfM)

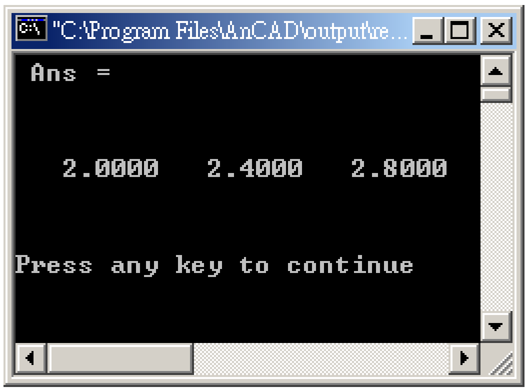
Figure 2.2.2.1 displays the created mfArray. The list of data is packed nicely into a 3-by-2 mfArray column-wise.

Step 3. Function `mfColon` is a useful function for creating a ramp of data. It creates an mfArray consisting of linearly increasing or decreasing values. It has the following prototype.

```
mfArray mfColon(mfArray start, mfArray step,
mfArray end);
```

The first argument specifies the starting number. The second argument specifies the increment per step. The third argument specifies the upperbound or lowerbound of the values, depending on whether the increment is positive or negative, respectively. By default, if the second argument is not specified, the increment is assumed to be 1.0. As an illustration, we shall construct an mfArray using function `mfColon`.

```
mfArray ramp = mfColon(2,0.4,3);
mfDisplay(ramp);
```



```
 Ans  =


   2.0000    2.4000    2.8000


Press any key to continue
```

Figure 2.2.2.2 row vector created using function mfColon

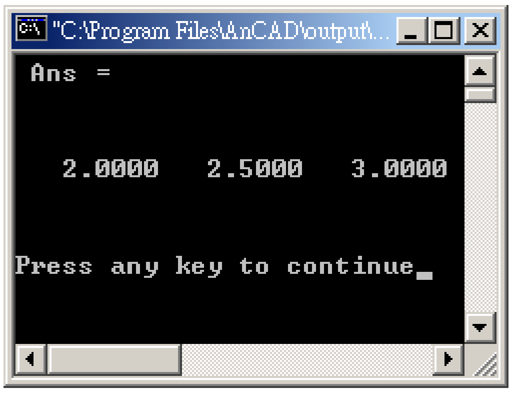
Figure 2.2.2.2 shows the result of mfArray. A row vector is created with an increment of 0.4, and a final value of 2.8 which is the last number that is less than the specified ending number of 3.

Step 4. Function `mfLinspace` is a similar function that generates a linear range of data. However, instead of specifying the increment value, you specify the number of points between the starting and ending numbers, inclusive. It has the following prototype.

```
mfArray mfLinspace(mfArray start, mfArray end,
mfArray npts);
```

The first argument specifies the starting number. The second argument specifies the ending number. The third argument specifies the number of points. By default, the number of points is 100. As an illustration, we shall construct an mfArray using function `mfLinspace.`

```
mfArray ramp1 = mfLinspace(2,3,3);
mfDisplay(ramp1);
```



"C:\Program Files\AnCAD\output\...

```
Ans =

   2.0000    2.5000    3.0000

Press any key to continue
```

Figure 2.2.2.3 mfArray created using mfLinspace

Figure 2.2.2.3 shows the elements of `ramp1.` Compare its values with mfArray `ramp` which was created using `mfColon.` Specifying a starting value of 2 and ending value of 3 created both mfArrays. Both are created with three elements each. However, the increment in value between each element differs.

Step 5. `mfVCat` and `mfHCat` are two member functions of the mfArray class, which can be used to concatenate vectors vertically and horizontally into matrices.

`mfVCat` performs vertical concatenation between vectors that conform in the number of columns. It has the following prototype.

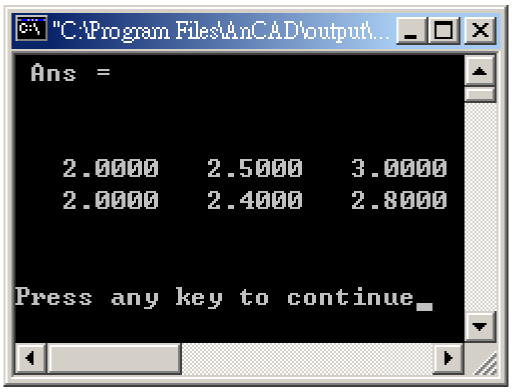
```
mfArray mfVCat(const mfArray& op1, const
mfArray& op2)  { return op1.VC(op2); }
```

`mfHCat` performs horizontal concatenation between mfArrays that conform in the number of rows. It has the following prototype.

```
mfArray mfHCat(const mfArray& op1, const
mfArray& op2)  { return op1.HC(op2); }
```

As an illustration, we shall perform a vertical concatenation between `ramp` and `ramp1`.

```
mfArray ramp2 = mfVCat(ramp1, ramp);
mfDisplay(ramp2);
```



```
 Ans  =


   2.0000    2.5000    3.0000
   2.0000    2.4000    2.8000


Press any key to continue
```

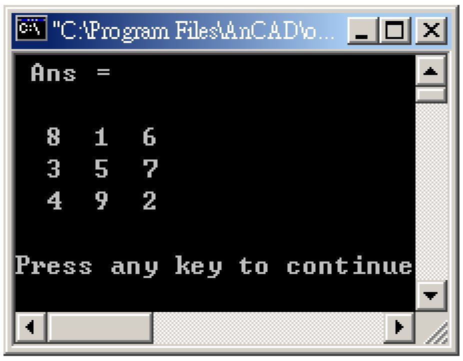
Figure 2.2.2.4 Vertical concatenation

Figure 2.2.2.4 displays the elements of the mfArray created from the vertical concatenation. Observe that the mfArray is composed from elements of `ramp1` placed on the top of elements of `ramp`.

MATFOR provided a few functions that creates special mfArrays such as magic square (`mfMagic`), identity matrix (`mfEye`), arrays containing ones (`mfOnes`), arrays containing zeros (`mfZeros`), and arrays containing random number (`mfRand`).

The magic square is a special matrix containing elements whose row, column, and diagonal sums are equal. You can easily create a magic square by using the function.

```
mfArray magic = mfMagic(3);
mfDisplay(magic);
```



```
Ans  =

  8   1   6
  3   5   7
  4   9   2

Press any key to continue
```

Figure 2.2.2.5 A 3-by-3 magic square

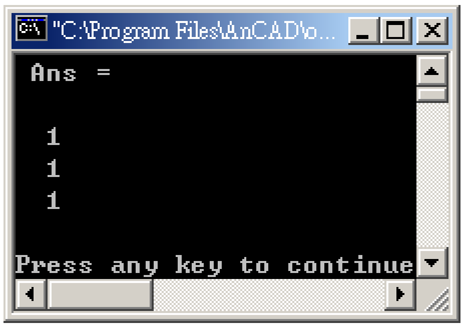
Figure 2.2.2.5 displays the elements of a 3-by-3 magic square created using function `mfMagic.` Observe that the row, column, and diagonal sums are equal. Function `mfMagic` is used in many MATFOR examples to create matrices for illustrating the functionality of MATFOR.

Functions `mfOnes, mfZeros,` and `mfRand` are three array-creating functions that enable users to create arrays of up to seven in dimension. These functions have the following general prototype.

```
mfArray mfFunction(mfArray x);
mfArray mfFunction(mfArray x1, mfArray x2, …,
mfArray x7);
```

The first prototype has a single input argument. It creates an x-by-x square mfArray. The second prototype has seven optional input arguments specifying the shape of the created input array. As an example, we shall create a 3-by-1-column vector containing ones (Figure 2.2.2.6.) and a 4-by-3-by-2 mfArray containing random numbers (Figure 2.2.2.7.).

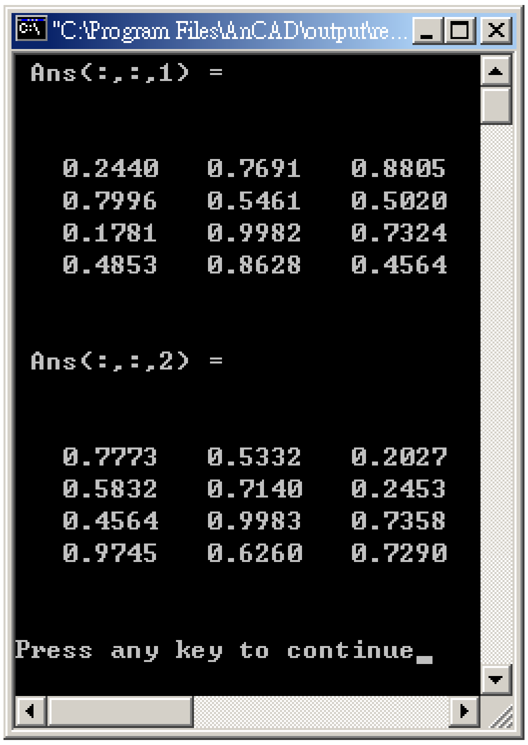
```
mfArray cv_ones = mfOnes(3,1);
mfDisplay(cv_ones);
```



```
Ans =

  1
  1
  1

Press any key to continue
```

Figure 2.2.2.6 A 3-by-1 vector containing ones created by mfOnes

```
mfArray rand = mfRand(4,3,2);
mfDisplay(rand);
```



```
 Ans(:,:,1) =


   0.2440    0.7691    0.8805
   0.7996    0.5461    0.5020
   0.1781    0.9982    0.7324
   0.4853    0.8628    0.4564


 Ans(:,:,2) =


   0.7773    0.5332    0.2027
   0.5832    0.7140    0.2453
   0.4564    0.9983    0.7358
   0.9745    0.6260    0.7290


Press any key to continue_
```

Figure 2.2.2.7 A 4-by-3-by-2 random array

2.3 Access Elements and Sections of an mfArray

The mfArray class has its own unique set of operators and functions for accessing elements and sections of the array. In this section, we first introduce the subscript conventions by the mfArray class, followed by an introduction to using the operator ( ) for accessing elements of an mfArray.

2.3.1 Element Subscripts

An array element is one of the scalar data making up an array. You can access a particular element of an mfArray variable by specifying the element’s storage position or more commonly called the element’s subscript. The mfArray class enables two types of subscripting convention – the scalar subscript and the multi-dimensional subscript. The scalar subscript is equivalent to the position of an element in the storage. Each subscript variable of an mfArray is stored sequentially in column-major convention. The multi-dimensional subscript follows the convention of arranging the array elements into rows, columns, pages, and indices. The mfArray class supports up to seven dimensions of elements.

In this subsection, we will look at the scalar subscript and multi-dimensional subscript.

Scalar subscript

When we specify an array element using a scalar subscript, we are effectively specifying the element’s storage position. In MATFOR, the array elements are stored column-wise following the Fortran convention. That is, elements are stored sequentially by concatenating one column after another. This is visualized in Figure 2.3.1.1, where the storage sequence of the elements of a 3-by-3 magic square is illustrated.

8 **(1)**	3 **(2)**	4 **(3)**	1 **(4)**	5 **(5)**	9 **(6)**	6 **(7)**	7 **(8)**	2 **(9)**

Figure 2.3.1.1 3-by-3 magic square – sequential arrangement of elements in storage

Multi-dimensional subscript

Figure 2.3.1.2 shows the arrangement of elements of a 3-by-3 magic square in table form. Each of the elements are subscripted by specifying their row followed by the column. Thus the fourth element of the 3-by-3 magic square has an equivalent subscript of (1,2) when you use the multi-dimensional subscript.

Although arrays with dimensions larger than three are seldom used, the mfArray class supports up to seven dimensions. You can visualize a three-dimensional array as a book whose pages contain tables of elements. Similarly, a dimensional array can be visualized as books on shelves, while the fifth dimension is like rooms containing shelves of books, and so on. When you specify an element in a multi-dimensional mfArray variable, you specify its position on the mfArray's row, column, page, shelf, room, etc, in the format of `(pos_row, pos_col, pos_page, …, po_sdim7).`

8 (1,1)	1 (1,2)	6 (1,3)
3 (2,1)	5 (2,2)	7 (2,3)
4 (3,1)	9 (3,2)	2 (3,3)

Figure 2.3.1.2 3-by-3 magic square

To further illustrate, we list below a few multi-dimensional subscripts.

λ a scalar has subscript (1,1)

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

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

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

- λ the seventh element on a 2-by-3-by-2 three-dimensional array has a subscript of (1,1,1).

2.3.2 mfArray Accessing Methods

An element of an array is also called a subscript variable, as you need to provide a variable name and subscript in order to specify an element. You can specify an mfArray element by providing the variable name followed by the element's scalar subscript or multi-dimension subscript. You can access an mfArray element through the mfArray operator and the member function as listed in Table 2.3.2.1.

Table 2.3.2.1 mfArray operator and member function for accessing elements

mfArray member function/operator	Action	Descriptions
mfArray operator(mfArray row, mfArray col, mfArray depth, ..., mfArray idx7);	Access elements	Access elements in an mfArray variable. Only the first argument is non-optional. Example, a = b(1,2); a = b("1:2", "1:2");

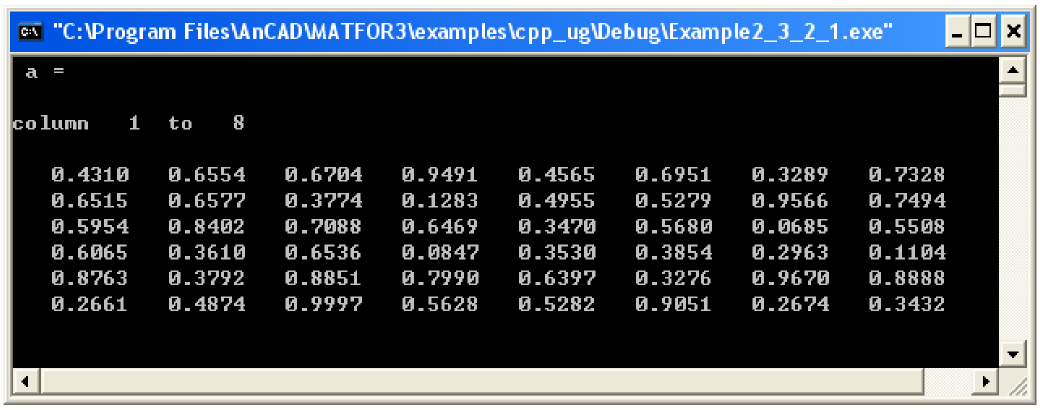
mfArray mfMatSub(mfArray& p, const mfIndexArray& idx1);	Access element	Access elements in an mfArray variable. It is used in the similar fashion as the mfArray operator. For example,
		
		a = mfMatSub(b, "1:100:2");
		
		is equivalent to
		
		a = b("1:100:2");

Example 2.3.2.1 Access an element of an mfArray variable

To get you familiar with using the above functions, we shall work through an example.

Step 1. First, start a new C++ program and save it as `Example2_3_2.cpp`. Include the header file for `cml` in the preprocessor directives. We shall use the function `mfRand` to create a 6-by-8-matrix mfArray (Figure 2.3.2.1) for illustration.

```
#include "cml.h"

void main()
{
    mfArray a;
    a = mfRand(6, 8);
    mfDisplay(a, "a");
```



"C:\Program Files\AnCAD\MATFOR3\examples\cpp_ug\Debug\Example2_3_2_1.exe"

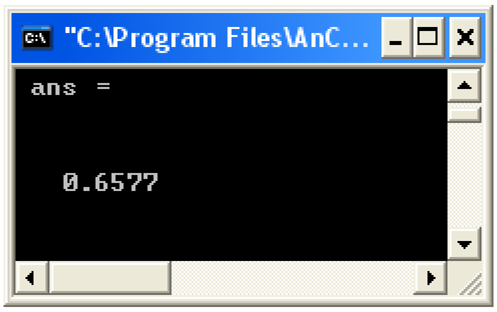
```
 a =

column    1  to    8

   0.4310    0.6554    0.6704    0.9491    0.4565    0.6951    0.3289    0.7328
   0.6515    0.6577    0.3774    0.1283    0.4955    0.5279    0.9566    0.7494
   0.5954    0.8402    0.7088    0.6469    0.3470    0.5680    0.0685    0.5508
   0.6065    0.3610    0.6536    0.0847    0.3530    0.3854    0.2963    0.1104
   0.8763    0.3792    0.8851    0.7990    0.6397    0.3276    0.9670    0.8888
   0.2661    0.4874    0.9997    0.5628    0.5282    0.9051    0.2674    0.3432
```

Figure 2.3.2.1 A 6-by-8 mfArray containing random numbers

Step 2. We shall use the close and open bracket operator () to get an element from the variable `a`. Note that operator () is not the same as the C++ square bracket subscript operator [].

We shall get an element at the 8th position or row 2, column 2 using the scalar subscript (Figure 2.3.2.2).

```
mfDisplay(a(8));
```

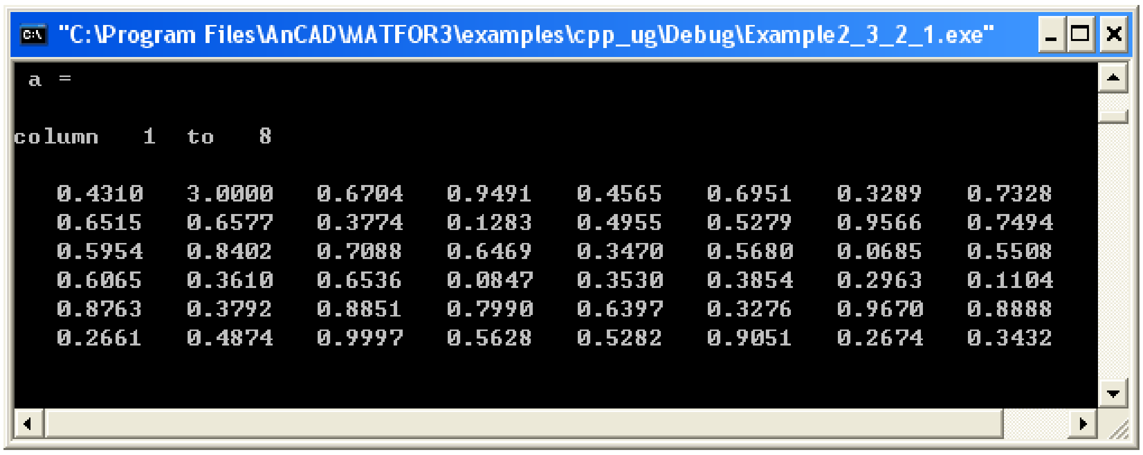


```
 ans =


   0.6577
```

Figure 2.3.2.2 Element at 8th position

Step 3. We have gotten data from an mfArray variable. We shall now attempt to write data to an element of an mfArray variable using the multi-dimensional subscript. As an example, we shall write to the element at row 1, col 2, replacing its existing value with the value 3.0.

```
a(1, 2) = 3.0;
mfDisplay(a, "a")
```



"C:\Program Files\AnCAD\MATFOR3\examples\cpp_ug\Debug\Example2_3_2_1.exe"

```
 a =

column    1  to    8

   0.4310   3.0000   0.6704   0.9491   0.4565   0.6951   0.3289   0.7328
   0.6515   0.6577   0.3774   0.1283   0.4955   0.5279   0.9566   0.7494
   0.5954   0.8402   0.7088   0.6469   0.3470   0.5680   0.0685   0.5508
   0.6065   0.3610   0.6536   0.0847   0.3530   0.3854   0.2963   0.1104
   0.8763   0.3792   0.8851   0.7990   0.6397   0.3276   0.9670   0.8888
   0.2661   0.4874   0.9997   0.5628   0.5282   0.9051   0.2674   0.3432
```

Figure 2.3.2.3 Variable after replacing element at (1,2) with 3.0

Example 2.3.2.1 accesses an mfArray variable one at a time. There are many cases when it is more convenient for programmers to read from or write to a group of elements, or section at once. In Example 2.3.2.2, we shall look at some examples of accessing the whole section of an mfArray variable.

Example 2.3.2.2 Access a section of an mfArray variable

An array section is a subgroup of an array. In many cases, it is more convenient to access a section of an array in one statement. By using the operator `()`, you can specify a section of elements in an mfArray easily. We shall now go through a few instances of accessing a whole section of an mfArray variable.

Step 1. Start a new program and save it as `Example2_3_2_2.cpp`. Include the header files for `cml.h` in your preprocessor directives. For the purpose of this example, we shall create the same 6-by-8 mfArray as completed previously in Example2_3_2_1, as shown in Figure 2.3.2.1.

```
#include "cml.h"
int main()
{
mfArray a;

a = mfRand(6, 8);
mfDisplay(a, "a");
```

Step 2. First, we will try to get data from elements in the section enclosed by row 1 to 3, and column 1 to 3, as in Figure 2.3.2.4. The section contains the elements (1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3). You can use the operator to specify the range 1 to 3 as follows.

```
mfDisplay(a("1:3", "1:3"));
```

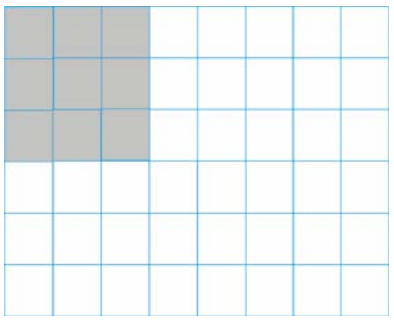
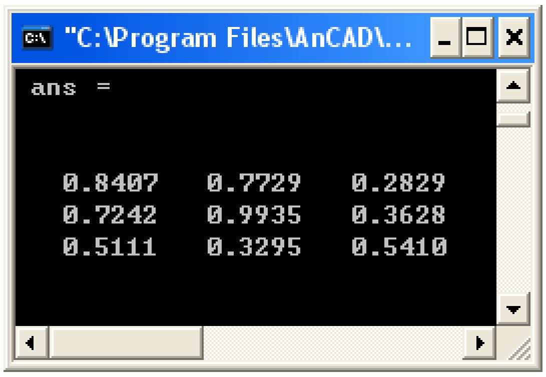


Figure 2.3.2.4 Subscript section shows: row[3] = {1,2,3}, col[3] ={1,2,3}



```
ans =


  0.8407    0.7729    0.2829
  0.7242    0.9935    0.3628
  0.5111    0.3295    0.5410
```

Figure 2.3.2.5 Top left 3-by-3 section of mfArray a

Step 3. You can also retrieve groups of elements that are not adjacent to each other, such as that shown in Figure 2.3.2.6. The elements to be retrieved are (2,1), (2,7), (2,4), (4,1), (4,7) (4,4), (6,1), (6,7) and (6,4).

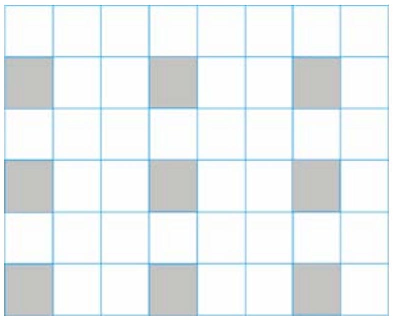
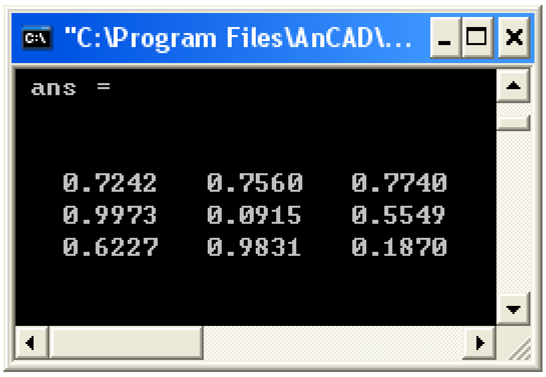


Figure 2.3.2.6 Subscript section shows: Row[3]= {2,4,6}, Column[3] = {1,4,7}

```
mfDisplay(a("2:6:2", "1:7:3"));
```



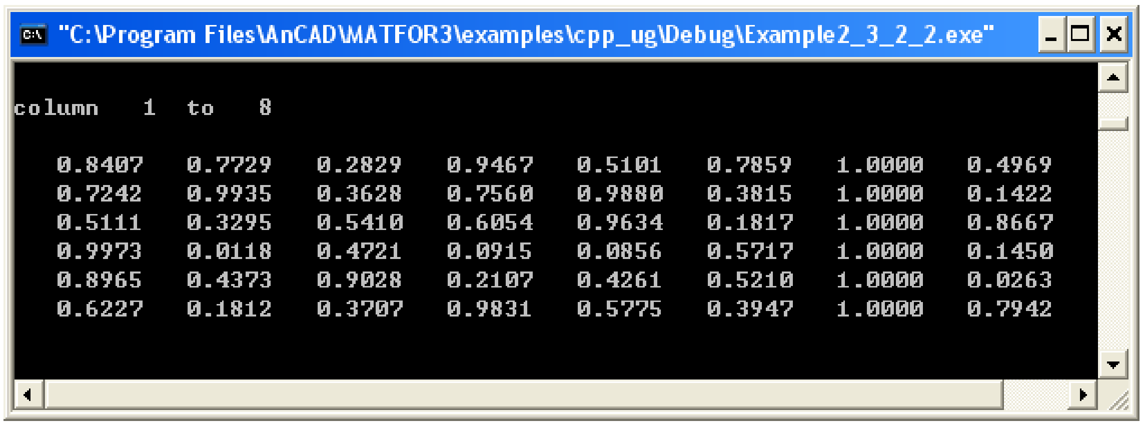
```
ans =

   0.7242    0.7560    0.7740
   0.9973    0.0915    0.5549
   0.6227    0.9831    0.1870
```

Figure 2.3.2.7 Section subscripts: Row[3]= {2,4,6}, Column[3] = {1, 4, 7}

Notice that in Figure 2.3.2.7., the values from column 7 are now on the third column, as we specified it as the second element in vector Column.

Step 4. You can specify a whole row or column by using the constant `MF_COL`. The constants `MF_COL` represent the colon, ":", symbol. In many applications, the ":" is used to represent all elements in subscripts. In mfArray class, you can use either the constant `MF_COL` or the string `":"` to select all elements in a row, column, or a specified dimension. As an example, we shall try to replace all the elements on column 7 of variable `a` with ones, as shown in Figure 2.3.2.8.

```
a(MF_COL, 7) = 1;
mfDisplay(a, ‘a’);
```

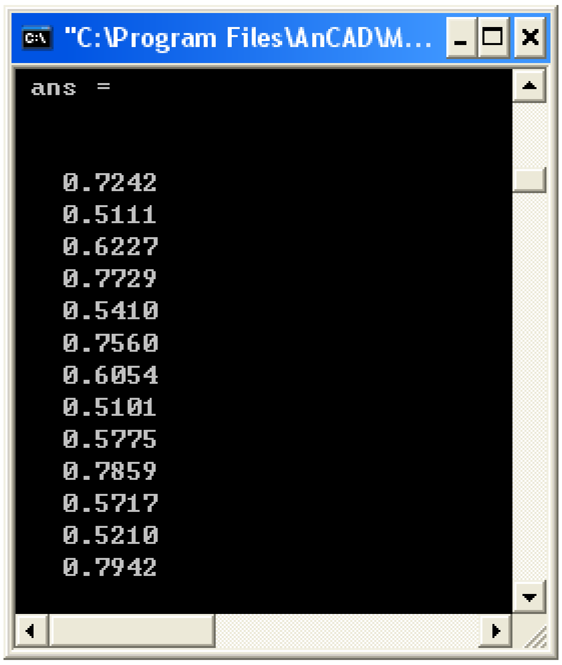


```
"C:\Program Files\AnCAD\MATFOR3\examples\cpp_ug\Debug\Example2_3_2_2.exe"

column    1   to    8

   0.8407   0.7729   0.2829   0.9467   0.5101   0.7859   1.0000   0.4969
   0.7242   0.9935   0.3628   0.7560   0.9880   0.3815   1.0000   0.1422
   0.5111   0.3295   0.5410   0.6054   0.9634   0.1817   1.0000   0.8667
   0.9973   0.0118   0.4721   0.0915   0.0856   0.5717   1.0000   0.1450
   0.8965   0.4373   0.9028   0.2107   0.4261   0.5210   1.0000   0.0263
   0.6227   0.1812   0.3707   0.9831   0.5775   0.3947   1.0000   0.7942
```

Figure 2.3.2.8 Variable a after replacing column 7 with ones

Step 5. You can apply the element-by-element comparison on the mfArray elements using the relational operators, such as <, <=, >, >=, etc. As an example, we shall try to retrieve all the elements whose values are less than or equal to 0.8 and greater than 0.5.

```
mfDisplay(a(a <= 0.8 & a > 0.5));
```



```
"C:\Program Files\AnCAD\M...
 ans  =


   0.7242
   0.5111
   0.6227
   0.7729
   0.5410
   0.7560
   0.6054
   0.5101
   0.5775
   0.7859
   0.5717
   0.5210
   0.7942
```

Figure 2.3.2.9 Elements that have values less than and equal to 0.8 and greater than 0.5

Step 6. Instead of using the operator, you can use the `mfIndexArray()` function to specify the element subscripts of an mfArray. This allows you to use integer variables to specify the element subscripts. The function `mfI()` follows the operator convention and has the following syntax:

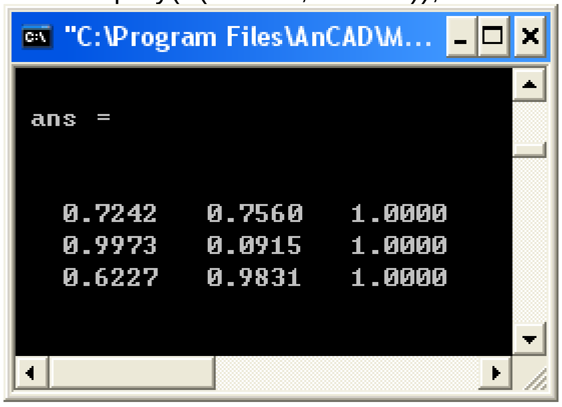
```
mfI(start, end, step);
```

For example,

mfDisplay(a(mfI(2,6,2), mfI(1,7,3)));

is equivalent to

mfDisplay(a("2:6:2", "1:7:3"));



```
ans =

  0.7242    0.7560    1.0000
  0.9973    0.0915    1.0000
  0.6227    0.9831    1.0000
```

Figure 2.3.2.10 Section subscripts : Row[3]= {2,4,6}, Column[3] = {1, 4, 7}

2.4 mfArray I/O

This section discusses the functions `mfDisplay` and `mfGDisplay` for displaying mfArray data, and functions `mfLoad` and `mfSave` for saving mfArray data to files and loading from files.

The section is further divided into the following subsections:

2.4.1 Displaying mfArray Data – This topic covers the usage of functions `mfDisplay` and `mfGDisplay`. Function `mfDisplay` displays mfArray data in short format in a console. Function `mfGDisplay` displays data in an mfArray variable on a spreadsheet-like Data Viewer.

2.4.2 mfArray Fileio – This topic introduces the functions `mfLoad` and `mfSave`. These functions, together with `mfLoadAscii` and `mfSaveAscii`, enable you to export mfArray data to an external file in ASCII and binary format for further processing.

We will also cover the two Matlab `.m` files, `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 content of an mfArray you can use two functions, mf`Display` and `mfGDisplay`, provided by MATFOR. Function `mfDisplay` outputs the content of mfArray to a console. Function `mfGDisplay` outputs the content to a MATFOR Data Viewer.

mfGDisplay

Function `mfGDisplay` is located in the `fgl` library. It displays content of mfArray variables in MATFOR Data Viewer. Data Viewer is a spreadsheet-like window that enables you to cut and paste, save files, and perform some data manipulations. You can work with both complex and real data in Data Viewer.

The general syntax of `mfGDisplay` is listed below.

```
void mfGDisplay(mfArray x);

void mfGDisplay (mfArray x, mfArray
label=“x”, …);
```

The first function call format requires only a single mfArray variable as the input argument. The second, multiple-input format requires the mfArray variable to be specified together with an mfArray variable containing string `“name”.` The number of mfArrays that you can display using a single function call is limited to 32.

By default, `mfGDisplay` displays data in the format of type `“short”`. That is, real numbers are displayed with four decimal places. You can use the slide bar located on the toolbar to change the number of digits displayed.

Example 2.4.1.1 below uses function `mfDisplay` to display mfArrays. Go through the example, try it on your compiler, you will see how integers, `‘short’,` and `‘long’` formats of mfArray data are displayed in the Windows console.

Example 2.4.1.1 Use mfGDisplay

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

Step 1. Start a new program and save it as `Example2_4_1_1.cpp`. Include the header files `fgl` and `cml` in the preprocessor directives and create a 3-by-3 magic square.

```
#include "cml.h"
#include "fgl.h"

void main()
{
   a = mfMagic(3);
```

Step 2. Next, we shall use function `mfSum` and `mfDiag` to compute the column, row, and diagonal sums of the magic square. Function `mfSum` computes the elements in an mfArray variable along a specified dimension. Function `mfDiag` returns a vector containing the diagonal elements of an mfArray variable.

```
mfArray b = mfSum(a, 1);
mfArray c = mfSum(a, 2);
mfArray d = mfSum(mfDiag(a));
```

Step 3. Display the mfArray variables `a, b, c,` and `d,` using function `mfGDisplay` in Data Viewer. Function `mfViewPause` is added after the function call to `mfGDisplay` to pause the program for examining the data.

```
mfGDisplay(a, "a", b, "column sum", c, "row sum", d, "diagonal sum");
mfViewPause();
```

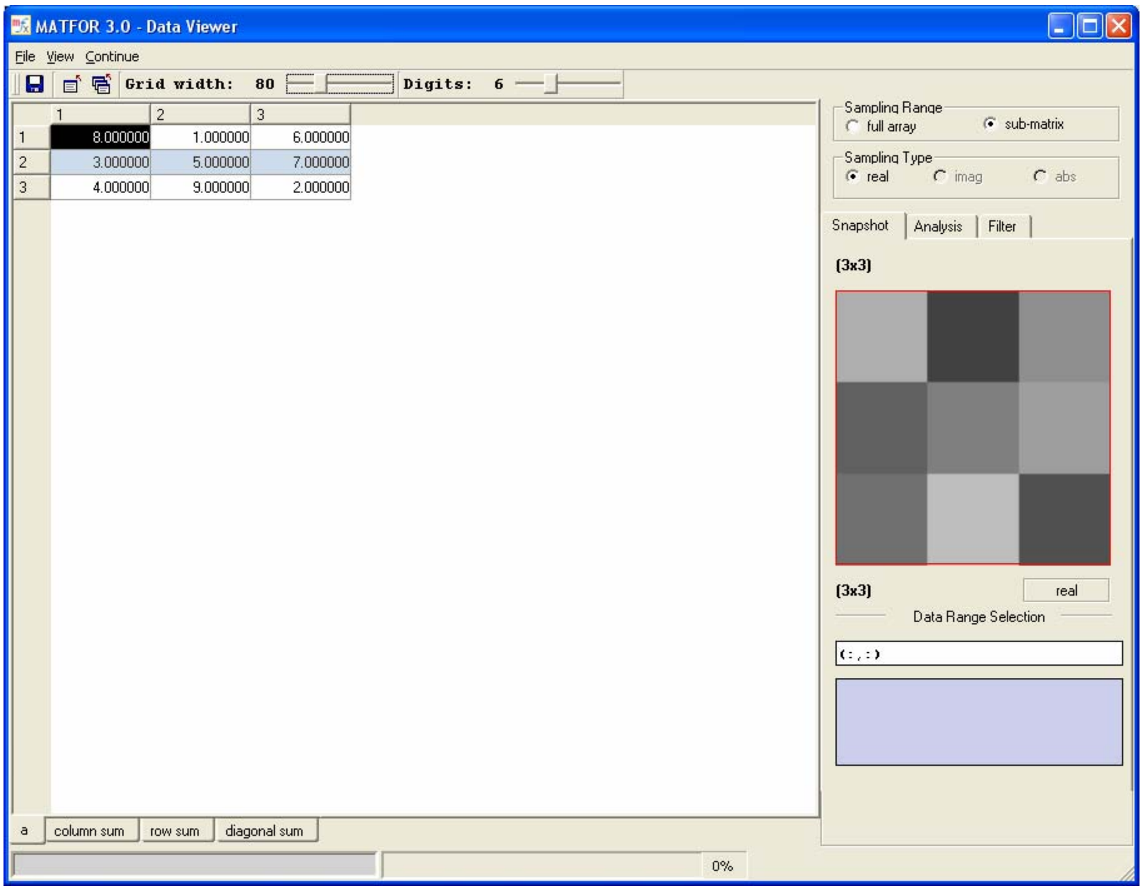
The MATFOR Data Viewer is displayed, as shown in Figure 2.4.1.1.

You can switch between each mfArray variable by clicking on the Worksheet tabs. The name of the tabs are specified by the name arguments in `mfGDisplay(a, "a", b, "column sum", c, "row sum", d, "diagonal sum"),` in the second, fourth, sixth, and eighth position respectively.

Navigate between the worksheets to view the values of the mfArray variables. You will see that `b` is a row vector containing the elements [15.000000, 15.000000, 15.000000] corresponding to each column sum. mfArray `c` is a column vector containing the same values while `d` is a scalar with value 15.000000. The row, column, and diagonal sums agree. The magic square is truly magical!

From Figure 2.4.1.1, 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.

Pause the program execution in order to display the Data Viewer. To continue with program execution, press the **Continue** button located at the top right corner of the Data Viewer. This button is available in both Graphics Viewer and Data Viewer.



Grid width: 80 Digits: 6

	1	2	3
1	8.000000	1.000000	6.000000
2	3.000000	5.000000	7.000000
3	4.000000	9.000000	2.000000

Sampling Range: full array [✓ sub-matrix]

Sampling Type: [✓ real] imag abs

[3x3]

[3x3] real

Data Range Selection

(:,:)

Figure 2.4.1.1 The Data Viewer displaying matrix a

Function mfViewPause

Function `mfViewPause` 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 Graphics 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 functions provided are: `mfLoad`, `mfSave`, `mfLoadAscii`, and `mfSaveAscii`. These functions are located in the `ess` library.

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 subsection below, we shall cover the usage of two functions, `mfSaveAscii` and `mfLoadAscii`, which export and import data to and from text files, followed by a discussion on `mfLoad.m` and `mfSave.m`.

2.4.2.1 mfSaveAscii

You can save data from the mfArray variable to a file by using function `mfSave` or `mfSaveAscii`. Function `mfSave` saves data in from an mfArray variable in MATFOR `*.mfb` binary format. Function `mfSaveAscii` saves data from an mfArray variable into a text file. Function `mfSaveAscii` has the following prototype:

```
void mfSaveAscii(char *fileName, const
mfArray &x);
```

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 function `mfSaveAscii`.

Example 2.4.2.1 Function mfSaveAscii

```
#include "cml.h"

void main()
{
//Create 3-by-3 magic square
mfArray a = mfMagic(3);

//Export a to a text file
mfSaveAscii(“a.txt”, a);

}
```

Compile and run the program. Open the text file `“a.txt”.` The data is arranged as follows:

```
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
```

2.4.2.2 mfLoadAscii

You can load data from a file into an mfArray variable by using function `mfLoadAscii` or function `mfLoad`. Function `mfLoadAscii` loads a text file in the format used by `mfSaveAscii` or Matlab ASCII data file. Function `mfLoad` loads a MATFOR *.mfb binary file created using function `mfSave`. Both functions `mfLoadAscii` and `mfLoad` have the same syntax. Next, we will look more closely at the application of function `mfLoadAscii`.

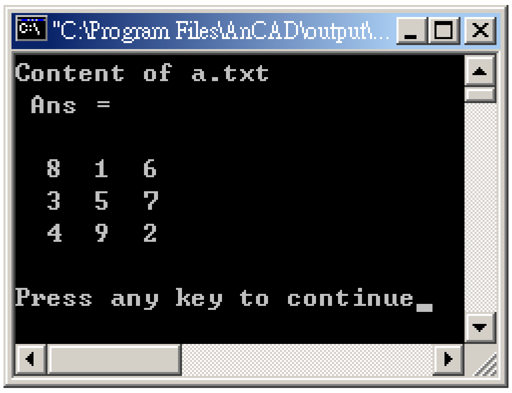
Function `mfLoadAscii` has the following prototype.

```
mfArray mfLoadAscii(char *fileName);
```

where `fileName` is a string specifying the name of the text file to load into the mfArray variable. Example 2.4.2.2 below loads the text file “`a.txt`” created in Example 2.4.2.1 using function `mfLoadAscii` into mfArray `a` and displays the data as in Figure 2.4.2.2.

Example 2.4.2.2

```
#include “cml.h”

void main()
{
// Import data into mfArray a
a = mfLoadAscii("a.txt");

// Display data of a
mfDisplay(a, "a");

}
```



```
Content of a.txt
 Ans =

  8   1   6
  3   5   7
  4   9   2

Press any key to continue
```

Figure 2.4.2.2 Data loaded using mfLoadAscii

2.4.2.3 mfLoad.m and mfSave.m

The Matlab `.m` files `mfLoad.m` and `mfSave.m`, are installed in folder `<MATFOR>\common\tools\matlab\` when you install MATFOR. These two files call 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 `mfLoad` 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 C++ environment, you can retrieve the data contained in `*.mfb` binary file, exported using function `mfSave` in Matlab, into MATFOR mfArray through function `mfLoad`. Likewise, you can save your MATFOR mfArray data in binary format using function `mfSave` 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 C++ using function `mfLoad`.

Step 1. First, ensure that the files `mfLoad.m`, `mfSave.m`, `mfLoad.dll`, and `mfSave.dll`, installed in `<MATFOR>\common\tools\matlab\`, have been copied to your Matlab working directory.

Step 2. We shall export the data in `[X, Y, Z]` matrices, computed using Matlab sample function –`peaks`, into 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>\examples\cpp_ug*.

Step 4. Create a C++ program with the filename Example2_4_2_3. We shall retrieve the data into mfArrays `x, y`, and `z` and plot the data using function `mfSurf`.

```
#include "cml.h"
#include "fgl.h"
void main()
{
//Load data into mfArray
mfArray x = mfLoad("x.mfb");
mfArray y = mfLoad("y.mfb");
mfArray z = mfLoad("z.mfb");

//Draw data using mfSurf
mfSurf(x, y, z);
mfViewPause();
}
```

Step 5. Compile and run the program. The results will be displayed as seen in Figure 2.4.2.3.

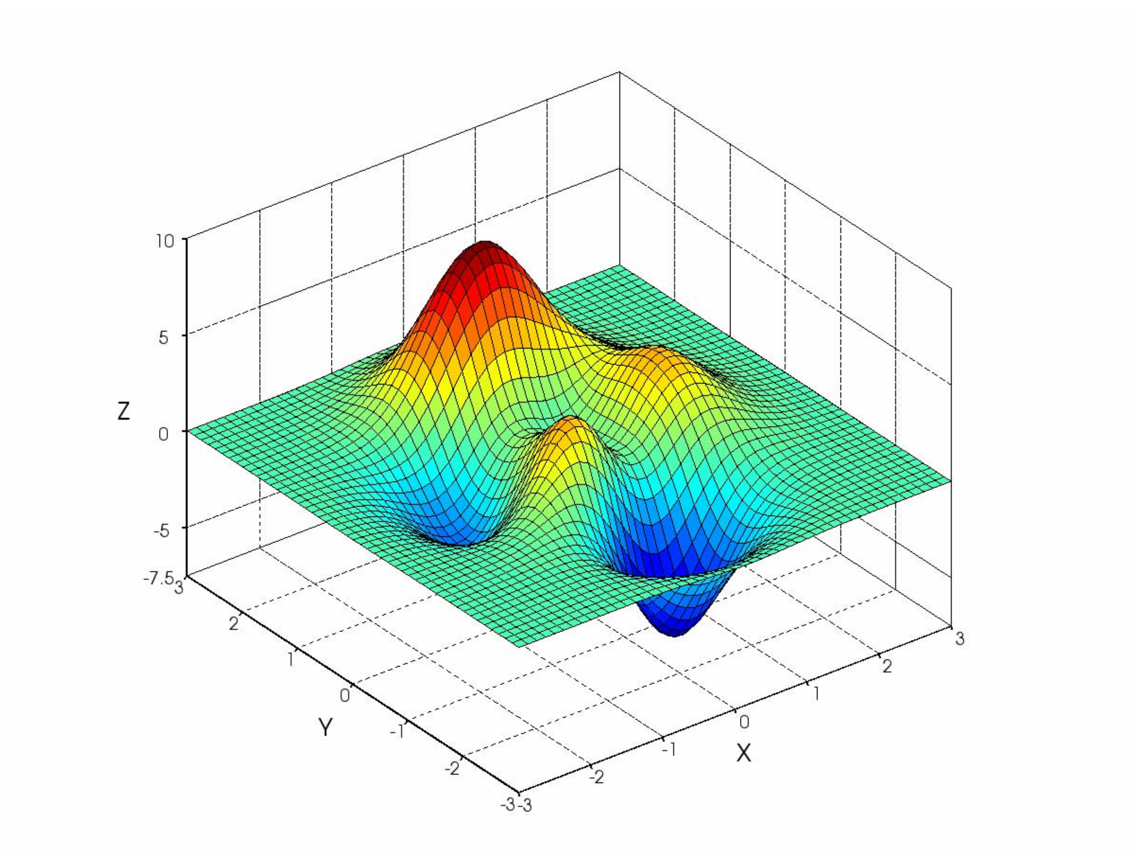


Figure 2.4.2.3 Surface plot produced by importing data from Matlab

2.5 mfArray Inquiry Functions

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

2.5.1 Logical Inquiry Functions – This topic covers the logical functions that are used to query the status of mfArray variables, such as `mfIsReal`, `mfIsScalar`, etc. The logical functions return a Boolean nonzero if true, and returns zero if false.

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

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 member functions listed in Table 2.5.1 return C++ built-in Boolean type as output.

Table 2.5.1 Logical Inquiry Functions.

Essential Set of MATFOR routines	
bool mfIsEmpty() const;	Return nonzero if specified mfArray variable empty.
bool mfIsLogical() const;	Return nonzero if mfArray is Boolean.
bool mfIsNumeric() const;	Return nonzero if mfArray contains numeric.
bool mfIsReal() const;	Return nonzero if mfArray contains double data.
bool mfIsComplex() const;	Return nonzero if mfArray contains complex class data.

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

For example,

```
mfArray :: a
if(mfIsEmpty(c)) then a = 5
```

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

- λ `mfIsEmpty` – An mfArray is empty if it has not been initialized and points to a null storage space.
- λ `mfIsLogical` – An mfArray is logical if it is constructed using logical operations.
- λ `mfIsNumeric` – An mfArray is numeric if it contains complex, logical, or real data type. A character type mfArray is non-numeric.
- λ `mfIsReal` – A real mfArray contains real data type.
- λ `mfIsComplex` –A complex mfArray contains double precision complex data type.

Example 2.5.1 Logical Inquiry functions

The example below uses the Logical Inquiry functions.

```
#include <stdio.h>
#include "cml.h"

void main() {

    mfArray a, b, c, d;
    bool L;

    a = mfMagic(3);
    d ="string";

    // Is mfArray c empty?

    if(mfIsEmpty(c))
            c = dcomplex(2,-2);

    // Is mfArray complex?
    if(mfIsComplex(c))
            mfDisplay(c, "c is complex");

    // Is mfArray numeric?
    L = mfIsNumeric(d);
```

```
        mfDisplay(d, "d", L, "mfIsNumeric(d)");

        // Is mfArray numeric real?
        L = mfIsReal(c);
        mfDisplay(c, "c", L, "mfIsReal(c)");

        // Is mfArray logical?
        d = mfAny(a);
        if(mfIsLogical(d))
                printf("d is logical\n");
    }
```

2.5.2 Size, Shape, and Extent

You can obtain information regarding the number of elements, shape, and extent of an mfArray variable using member functions `GetM`, `GetN`, `GetZ`, `GetDims` and `mfSize`. These functions are as listed in Table 2.5.2 below.

Table 2.5.2 Member Functions

Essential Set of MATFOR routines	
int GetM() const;	int GetM() const;
int GetN() const;	int GetN() const;
int GetZ() const;	int GetZ() const;
void GetDims (int dimf[7]) const;	void GetDims (int dimf[7]) const;
Elementary matrix-manipulating functions	
int mfSize (mfArray x, int i);	int mfSize (mfArray x, int i);
void mfSize(mfOutArray OutArray, mfArray x);	void mfSize(mfOutArray OutArray, mfArray x);
int mfNDims(mfArray x)	int mfNDims(mfArray x)
mfArray mfShape(mfArray x)	mfArray mfShape(mfArray x)
mfArray mfLength(mfArray x)	mfArray mfLength(mfArray x)

mfArray class member functions, `GetM`, `GetN`, and `GetZ`, return C++ built-in scalar integer data type as output, while function `GetDims` returns a C++ vector, with a size of seven, integer as output.

To get the shape of an mfArray, you can use a combination of member functions, `GetM`, `GetN`, `GetZ`, and `GetDims`, to get the information in integer format or you can use function `mfSize`, which returns an mfArray. Effectively, member function `mfSize` serves the same purpose as functions `GetM`, `GetN`, and `GetZ`. You would need to specify the dimension argument to get the size of a certain dimension, while functions `GetM`, `GetN`, and `GetZ` are more intuitive to a reader.

Example 2.5.2 Size and shape

To get you familiar with the functions, we shall go through an example.

Step 1. Start a new function and name it Example2_5_2. Include the header files for `cml.h` and iostream in your preprocessor directives. We shall create a 2-by-3-by-4 mfArray variable containing ones for the purpose of this example.

```
#include "cml.h"

int main()
{
mfArray a, S1;
int S;
//Construct a 2-by-3-by-4 mfArray variable containing ones.
a = mfOnes(2,3,4);
```

Step 2. We shall use member function `mfSize` to check the number of elements in the mfArray.

```
S = mfSize(a);
mfDisplay(S,"mfSize(a)");

S = mfSize(a, 1);
mfDisplay(S, "mfSize(a,1)");

S = mfSize(a, 2);
mfDisplay(S, "mfSize(a,2)");

S = mfSize(a, 3);
mfDisplay(S, "mfSize(a,3)");
```

Step 3. Next, we shall display the number of elements in each dimension of the mfArray.

```
S = a.GetM();
mfDisplay(S, "a.GetM()");
S = a.GetN();
mfDisplay(S, "a.GetN()");
S = a.GetZ();
mfDisplay(S, "a.GetZ()");
```

Step 4. Finally, display the number of dimension and shape of the mfArray using `mfNDims` and `mfShape`.

```
S = mfNDims(a);
mfDisplay(S, "mfNDims(a)");
S1 = mfShape(a);
mfDisplay(S1, "mfShape(a)");
```

2.5.3 Logical Operations

You can use functions `mfAll`, `mfAny`, and `mfFind`, as listed in Table 2.5.3 to find nonzero in mfArray.

Table 2.5.3 Logical Operations

Essential set of MATFOR routines	
mfArray mfAny(const mfArray& op1);	mfArray mfAny(const mfArray& op1);
mfArray mfAny(const mfArray& op1, const mfArray& dim);	mfArray mfAny(const mfArray& op1, const mfArray& dim);
mfArray mfAll(const mfArray& op1);	mfArray mfAll(const mfArray& op1);
mfArray mfAll(const mfArray& op1, const mfArray& dim);	mfArray mfAll(const mfArray& op1, const mfArray& dim);

Elementary matrix-manipulating functions	

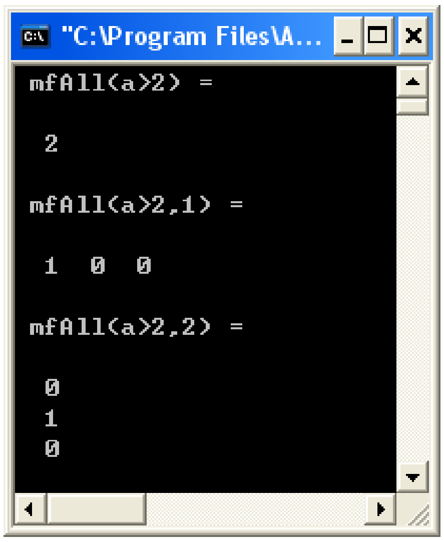
mfArray mfFind(mfArray x);
 	
void mfFind(mfOutArray OutArray, mfArray x);
	mfArray mfFind(mfArray x);
	
	void mfFind(mfOutArray OutArray, mfArray x);

Essential set of MATFOR routines	Essential set of MATFOR routines
mfArray mfAny(const mfArray& op1);	mfArray mfAny(const mfArray& op1);

Each function provides different information about the nonzero elements in mfArray.

λ Functions `mfAll` and `mfAny` query the status of nonzero elements of mfArray. The functions operate along the column or specified dimension of an mfArray, returning a logical mfArray as output. MATFOR uses one to represent logical true and zero to represent logical false. As an example,

```
// Function mfALL
    l = mfAll(a>2);
    mfDisplay(l, "mfAll(a>2)");

    b = mfAll(a>2,1);
    mfDisplay(b, "mfAll(a>2,1)");

    b = mfAll(a>2,2);
    mfDisplay(b, "mfAll(a>2,2)");
```



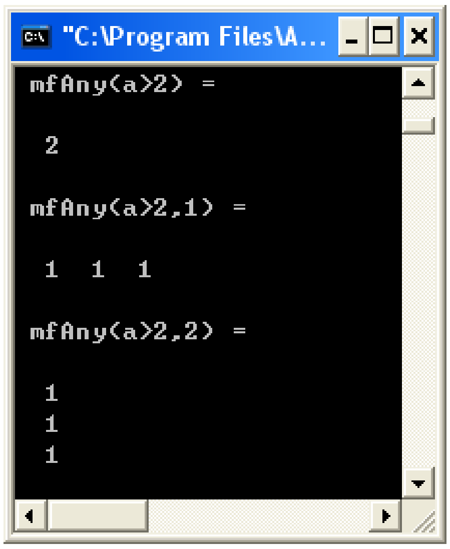
"C:\Program Files\A...

```
mfAll(a>2) =

 2

mfAll(a>2,1) =

 1   0   0

mfAll(a>2,2) =

 0
 1
 0
```

Figure 2.5.3.1 Operation of mfAll

```
// Function ANY
l = mfAny(a>2);
mfDisplay(mf(l), "mfAny(a>2)");

b = mfAny(a>2,1);
mfDisplay(b, "mfAny(a>2,1)");

b = mfAny(a>2,2);
mfDisplay(b, "mfAny(a>2,2)");
```

.



```
mfAny(a>2) =

 2

mfAny(a>2,1) =

 1  1  1

mfAny(a>2,2) =

 1
 1
 1
```

Figure 2.5.3.2 Operation of mfAny

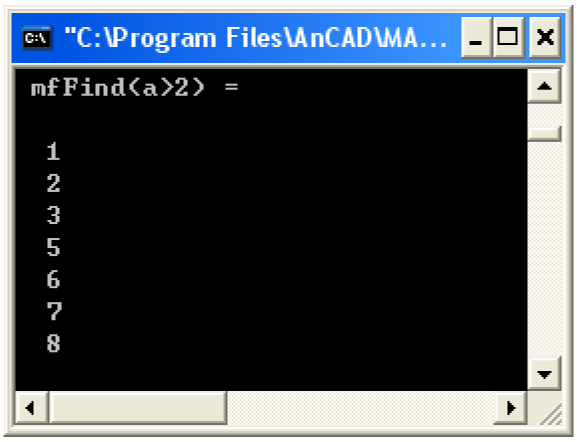
λ You can use function `mfFind` to get indices of non-zero elements of mfArray. Depending on your input argument, you can return:

1) a vector mfArray containing the column-major index

2) two vector mfArrays containing the corresponding row and column element subscripts

3) three vectors containing the row and column element subscripts and values corresponding to nonzero values.

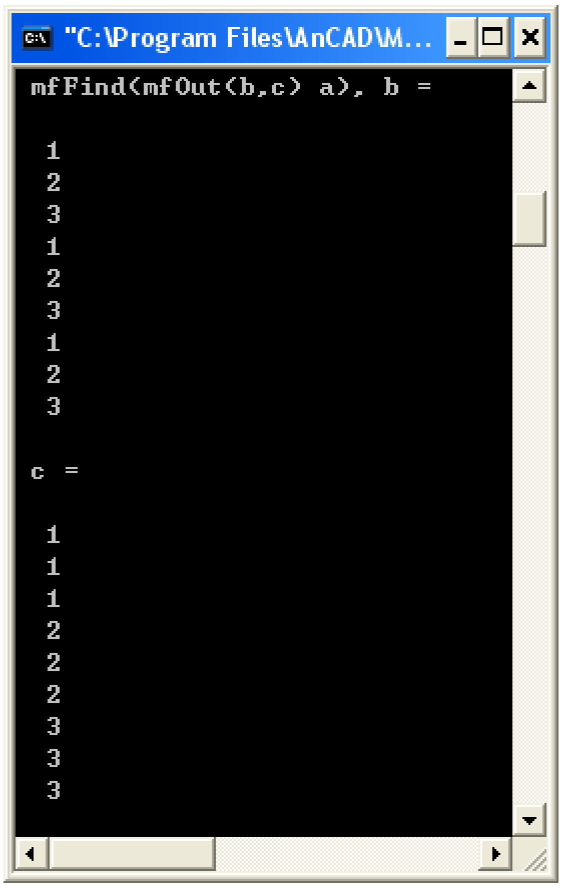
```
b = mfFind(a>2);
mfDisplay(a, "a");
mfDisplay(b, "mfFind(a>2)");
```



```
mfFind(a>2) =

  1
  2
  3
  5
  6
  7
  8
```

Figure 2.5.3.3 Scalar subscripts of values greater than 2

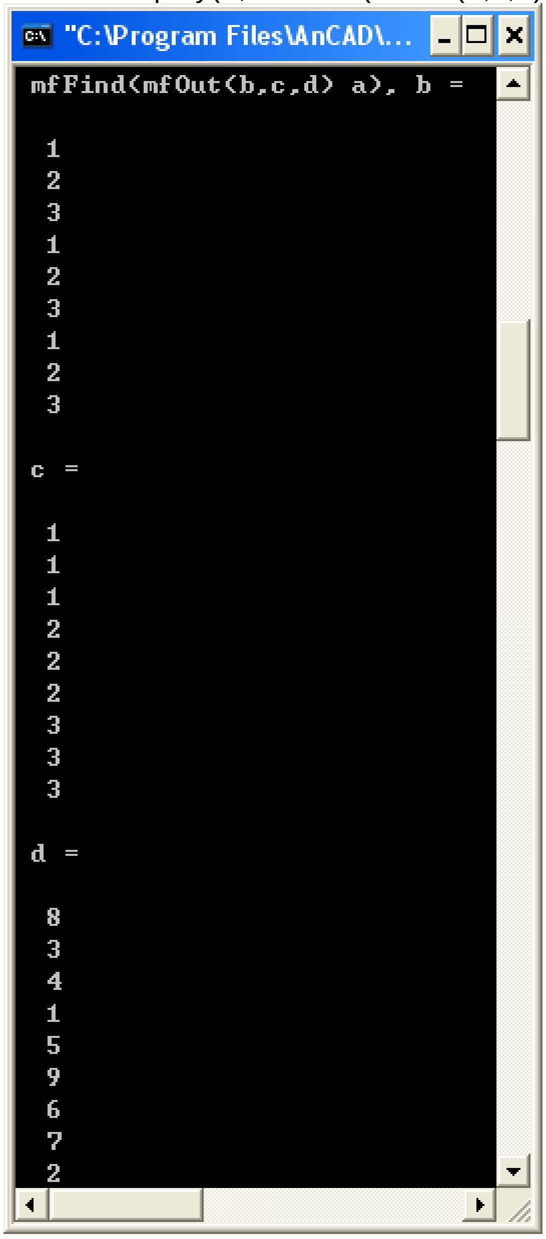
```
mfFind(mfOut(b,c),a);
mfDisplay(b, "mfFind(mfOut(b,c) a), b", c, "c");
```



```
mfFind(mfOut(b,c) a), b =

  1
  2
  3
  1
  2
  3
  1
  2
  3

c =

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

Figure 2.5.3.4 Row and Column element subscripts of nonzeros

```
mfFind(mfOut(b,c,d),a);
mfDisplay(b, "mfFind(mfOut(b,c,d) a), b", c, "c",d,"d");
```



```
mfFind(mfOut(b,c,d) a), b =

 1
 2
 3
 1
 2
 3
 1
 2
 3

c =

 1
 1
 1
 2
 2
 2
 3
 3
 3

d =

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

Figure 2.5.3.5 Row and Column element subscripts and values of nonzeros

2.6 mfArray Operators

MATFOR mfArray class supports a set of operator and operator functions. The operators and functions are listed as below in Table 2.6, which are listed according to their precedence.

The operator functions include `Mul` for matrix multiplication, `LDiv` for matrix left divide, `RDiv` for matrix right divide, `T` for array transpose, `H` for complex conjugate transpose, `HCat` for horizontal concatenation, and `VCat` for vertical concatenation. These operator functions enable you to perform matrix manipulation conveniently.

Table 2.6 mfArray Operators and Operator functions

Operators/ Functions	Descriptions/Prototype

()	Round bracket, subscript operator
	
	mfArray operator()(mfArray r) const;
	mfArray operator()(mfArray r, ..., mfArray indx7) const;

*	mfArray array multiplication
	
	mfArray operator *(const mfArray& rhs) const;
	mfArray operator *(const double& rhs) const;

/	mfArray array right division
	
	mfArray operator /(const mfArray& rhs) const;
	mfArray operator /(const double& rhs) const;

+	mfArray array addition or unary plus
	
	mfArray operator +(const mfArray& rhs) const;
	mfArray operator +(const double& rhs) const;

-	mfArray array subtraction or unary minus
	
	mfArray operator -(const mfArray& rhs) const;
	mfArray operator -(const double& rhs) const;

<	mfArray array less than comparison mfArray operator < (const mfArray& rhs) const;

Operators/ Functions	Descriptions/Prototype
<=	mfArray less than or equal to comparison mfArray operator <=(const mfArray& rhs) const;
>	mfArray greater than comparison mfArray operator > (const mfArray& rhs) const;
>=	mfArray greater than or equal comparison mfArray operator >=(const mfArray& rhs) const;
==	mfArray array equality comparison mfArray operator ==(const mfArray& rhs) const;
!=	mfArray array inequality comparison mfArray operator !=(const mfArray& rhs) const;

=	mfArray assignment operator
	
	mfArray& operator =(const mfArray& rhs);
	mfArray& operator =(const char* str);
	mfArray& operator =(const int value)
	mfArray& operator =(const double value);
	mfArray& operator =(const complex value);

Pow()	mfArray power function.
	
	mfArray Pow(const mfArray& rhs) const;
	mfArray Pow(const double& rhs) const;

T()	mfArray transpose function mfArray T() const;
H()	mfArray complex conjugate transpose function mfArray H() const;
Mul()	mfArray matrix multiplication function mfArray Mul(const mfArray& rhs) const;

Operators/ Functions	Descriptions/Prototype
LDiv()	mfArray matrix left divide function mfArray LDiv(const mfArray& rhs) const;
RDiv()	mfArray matrix right division function mfArray RDiv(const mfArray& rhs) const;
HC()	mfArray horizontal concatenation function. mfArray HCat(const mfArray& rhs) const;
VC()	mfArray horizontal concatenation function. mfArray VCat(const mfArray& rhs) const;

2.6.1 Arithmetic Operators

The mfArray arithmetic operators `*, /, +, -,` operate element-by-element on the mfArray. You can perform operation between two mfArrays or a single mfArray and a double scalar. Example 2.6.1 shows some valid operations of the arithmetic operators.

Example 2.6.1 Arithmetic operators

```
#include "cml.h"

void main() {

    mfArray a, b, c, d, e, f;

    a = mfOnes(3,3);
    mfDisplay(a, "a");

    // * element-by-element multiplication
    b = 2*a;
    mfDisplay(b, "2*a");

    // ** element-by-element power
    c = mfPow(b,2);
    mfDisplay(c, "mfPow(b,2)");

    // - element-by-element subtraction
    d = c -a;
    mfDisplay(d, "c-a");
```

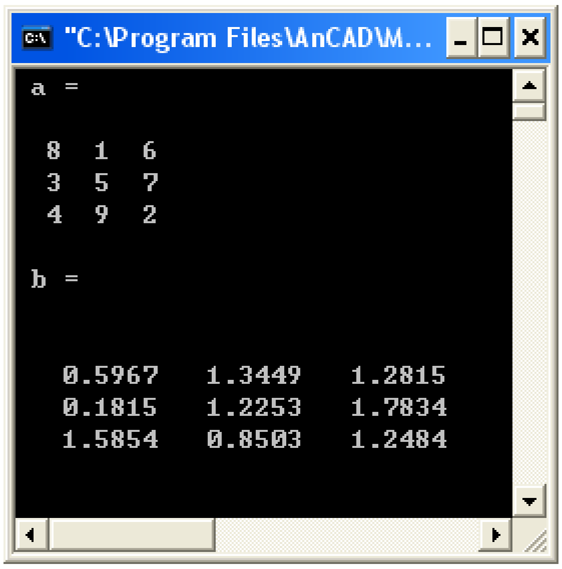
```
        // / element-by-element division
        e = c/b;
        mfDisplay(e, "c/b");

    }
```

2.6.2 Relational Operators

The mfArray class relational operators include `>=, >, <=, <, !=, ==.` These operators perform element-by-element comparison 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

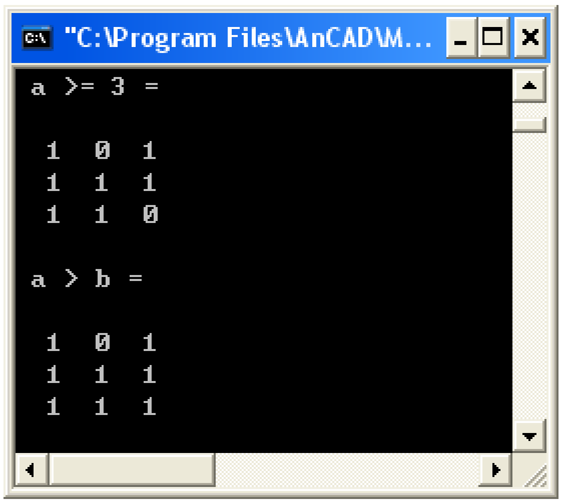
```
    #include "cml.h"


    void main() {

        mfArray a, b, c;

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

        mfDisplay(a, "a", b, "b");
```



```
"C:\Program Files\AnCAD\M...
a =

 8  1  6
 3  5  7
 4  9  2

b =


  0.5967   1.3449   1.2815
  0.1815   1.2253   1.7834
  1.5854   0.8503   1.2484
```

Figure 2.6.2.1 Contents of a and b

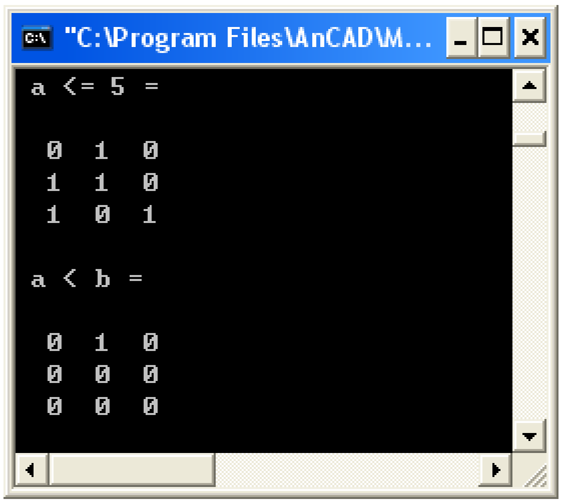
```
c = a >= 3;
mfDisplay(c, "a >= 3");

c = a > b;
mfDisplay(c, "a > b");
```



```
a >= 3 =

 1  0  1
 1  1  1
 1  1  0

a > b =

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

Figure 2.6.2.2 element by element > and >= comparison

```
c = a <= 5;
mfDisplay(c, "a <= 5");

c = a < b;
mfDisplay(c, "a < b");
```

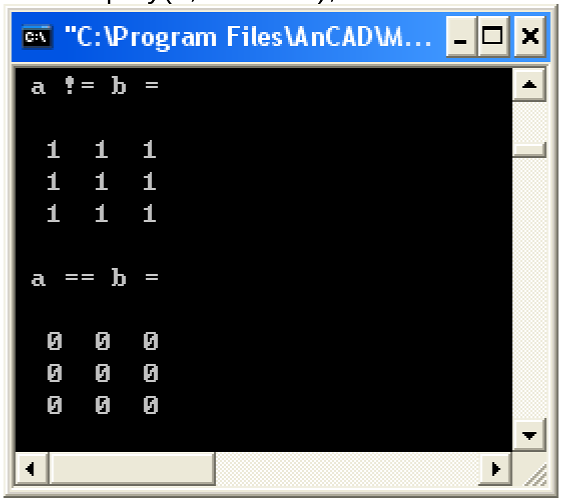


```
a <= 5 =

 0  1  0
 1  1  0
 1  0  1

a < b =

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

Figure 2.6.2.3 element by element <= and < comparison

```
c = a != b;
```

```
mfDisplay(c, "a != b");

c = a == b;
mfDisplay(c, "a == b");
```



```
a != b =

  1   1   1
  1   1   1
  1   1   1

a == b =

  0   0   0
  0   0   0
  0   0   0
```

Figure 2.6.2.4 element by element != and == comparison

2.6.3 Matrix Operators and Functions

You can perform matrix operations using mfArray member functions such as `Transpose, T(), H(), Mul, LDiv, RDiv, HCat, VCat.`

Function `T()` performs a matrix transpose.

Example,

```
a =

        8   1   6

        3   5   7

        4   9   2


b = a.T();
```

```
8  3  4

1  5  9

6  7  2
```

Function `H()` performs a complex conjugate transpose.

Example,

`a =`

```
1 + 2i   2+3i
```

`b = a.H() =`

```
1 - 2i

2 - 3i
```

Matrix multiplication function, x.`Mul(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, `a.LDiv(b)`, and matrix right division function, `a.RDiv(b)`, solve linear matrix inverse problems. The result of `a.LDiv(b)` is approximately `Mul(mfInv(a),b).` The result of `a.RDiv(b)`is approximately `Mul(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 of the difference between matrix right division and matrix left division are covered under `Matrix Division` below.

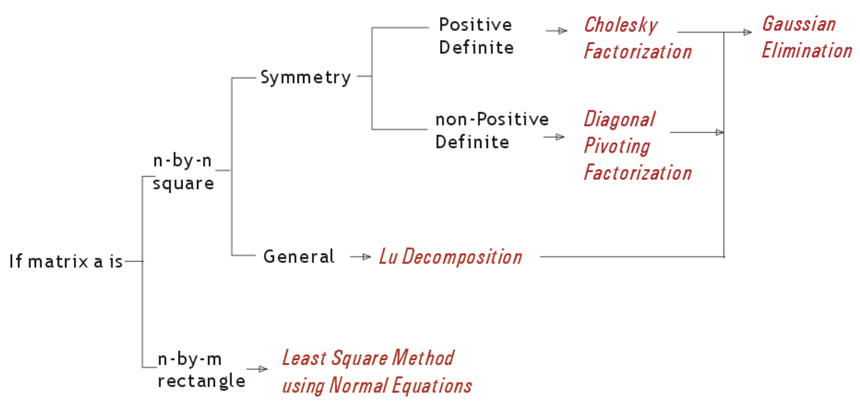


Figure 2.6.3 Algorithms applicable for each matrix type in matrix division operation

Matrix Division

Matrix division is often used to solve the linear matrix inverse problem `ax = b`, where `a` is an 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 way. The appreciation of writing `x` and `b` as row vectors has turned the equation into `xa = b`. To accommodate these two conventions, MATFOR introduces left and right matrix division. Use functions `LDiv` to solve systems where matrix `a` is put on the left of unsolved variable `x`. On the other hand, use mf`RDiv` 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`.

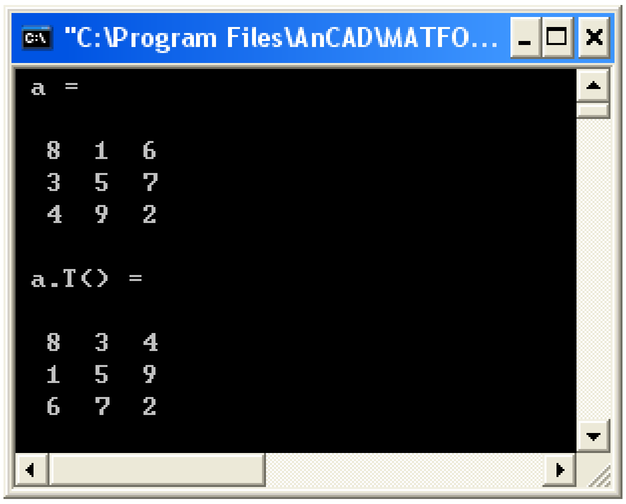
For non-square matrix `a,` equation `ax = b` represents an over-determined or under-determined system. In either case, matrix division routines provide solutions in least square sense.

Example 2.6.3 below lists some valid operations of the mfArray operator functions. The code is divided into four sections, namely matrix transpose, complex conjugate transpose, matrix multiplication, and matrix left divide. The results for each section of code are shown in Figure 2.6.3.1, Figure 2.6.3.2, Figure 2.6.3.3, and Figure 2.6.3.4 respectively.

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

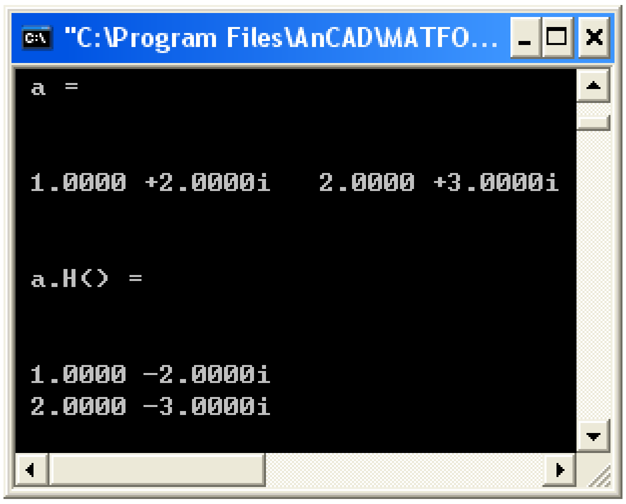
Example 2.6.3 Matrix Operators

```
#include "cml.h"

void main() {

    mfArray a, b, c;

    //  matrix transpose
    a  = mfMagic(3);
    b = a.T();
    mfDisplay(a, "a", b, "a.T()");


    // .h. complex conjugate transpose
    a = mfV(dcomplex(1,2), dcomplex(2,3));
    //a = mfV((1,2), (2,3));
    //b = mfCTranspose(a);
    b = a.H();
    mfDisplay(a, "a", b, "a.H()");
```



```
a =

  8   1   6
  3   5   7
  4   9   2

a.T() =

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

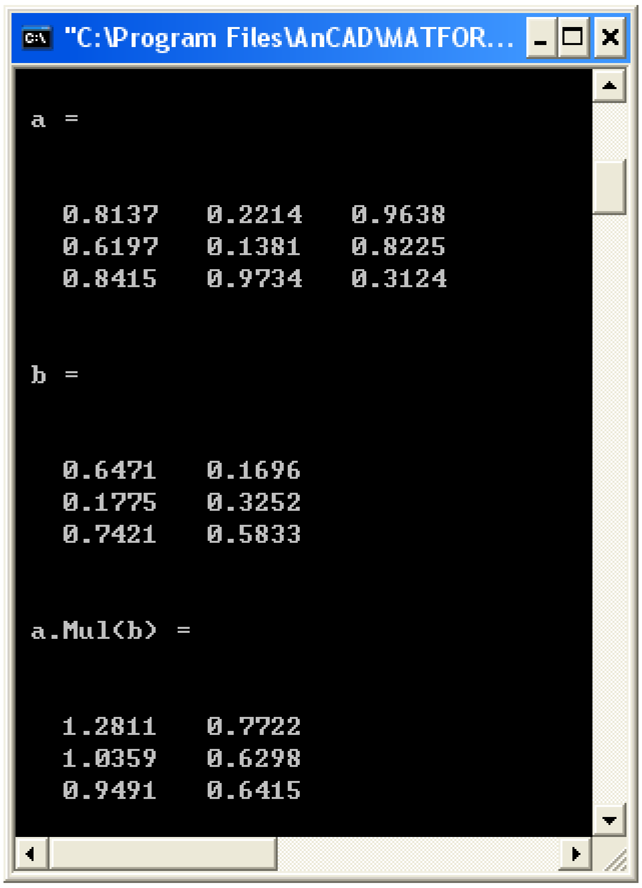
Figure 2.6.3.1 Result of transpose function



```
a =


1.0000 +2.0000i    2.0000 +3.0000i


a.H() =


1.0000 -2.0000i
2.0000 -3.0000i
```

Figure 2.6.3.2 Result of complex conjugate transpose function

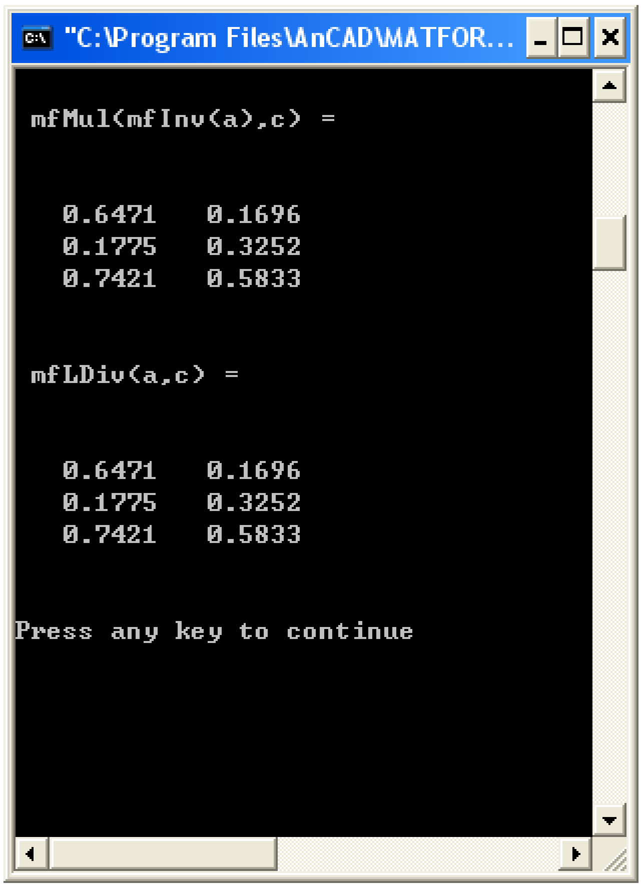
```
// , matrix multiplication
a = mfRand(3);
b = mfRand(3,2);
c = a.Mul(b);
mfDisplay(a, "a", b, "b", c, "a.Mul(b)");
```



```
a =

   0.8137    0.2214    0.9638
   0.6197    0.1381    0.8225
   0.8415    0.9734    0.3124


b =

   0.6471    0.1696
   0.1775    0.3252
   0.7421    0.5833


a.Mul(b) =

   1.2811    0.7722
   1.0359    0.6298
   0.9491    0.6415
```

Figure 2.6.3.3 Result of matrix multiplication

```
        // .mldiv. matrix left division
        b = mfMul(mfInv(a),c);
        mfDisplay(b, "mfMul(mfInv(a),c)");

        b = a.LDiv(c);
        mfDisplay(b, "mfLDiv(a,c)");
    }
```



```
 mfMul(mfInv(a),c) =


   0.6471    0.1696
   0.1775    0.3252
   0.7421    0.5833


 mfLDiv(a,c) =


   0.6471    0.1696
   0.1775    0.3252
   0.7421    0.5833


Press any key to continue
```

Figure 2.6.3.4 Comparing LDiv and mfMul(mfInv(a),c)

2.6.4 MATFOR Parameters

Table 2.6.4 below lists some MATFOR pre-defined parameters.

Table 2.6.4 MATFOR parameters

Parameter	Data Type	Description
mf()	mfArray	Empty mfArray
MF_COL	mfArray	Colon ":" operator
MF_PI	double	π
MF_EPS	double	The smallest positive number

MF_INF	double	Positive infinity number
MF_NINF	double	Negative infinity number
MF_NAN	double	Not a number
MF_E	double	Natural logarithm number
MF_REALMAX	double	Largest representable number
MF_REALMIN	double	Smallest representable number

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 C++ Library provides users with a set of linear algebra functions to solve matrix operation intuitively and efficiently.

Three types 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 has built in mechanisms for handling the details of algorithm selection in matrix operations. Intuitive interfaces are provided so that users do not need to know the details of the algorithm used. We shall go through three examples that are in Sections 3.1, 3.2, and 3.3 to familiarize with the linear algebra functions.

3.1 Matrix Inverse

Matrix inverse is often used in mathematical applications. The following is an example employing MATFOR matrix inverse function, `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:

$Y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{\imath 2} + E_i$

where,

$i = 1,2,\ldots,14$

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:

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

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

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 function to determine the inverse of $(X'X)$.

```
#include "cml.h"
#include "fgl.h"
#include <iostream.h>

void main() {

        mfArray y, x1, x2, beta, a, ey, x;

        double t1, t2;


        x1= mfV(1298, 119.8, 344.28, 235.56, 163.79,
        76.72,17.81,
              30.66, 15.92, 345.08, 6.70, 28,75,12.47).T();


        x2= mfV(
          437.26, 1283.48, 1128.33, 600.58,
          783.15, 65.26, 441.26, 242.33, 23.98,
          371.98, 324.4, 262.11, 1058.16, 1072.27).T();


        // input two values (Gross National product and
        // Per Capita Import)
        cout << "Input two value" << endl;
        cout << "Input 1:";
        cin >> t1;
        cout << "Input 2:";
        cin >> t2;

        a = mfT(mfOnes(1,14));
        x = mfHCat(a, x1, x2);

        y= mfV(6825, 512, 1902, 146, 2814,
          37, 52, 56, 184, 1065, 107, 173,
          771, 192).T();

        // 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);

        mfDisplay(beta,"beta",ey,"ey");
        mfGDisplay(x1,"x1",x2,"x2",x,"x",a,"a",y,"y",beta,"beta",e
        y,"ey");
        mfViewPause();
}
```

Compile and run the code.

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

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

3.2 Application of Eigenvalues and Eigenvectors

Eigenvectors and eigenvalues 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 function `mfEig` to determine eigenvectors and eigenvalues of a matrix. The example below applies function `mfEig` in solving a differential equation.

Example 3.2 Solving a differential equation

In this example, we shall use function `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 $A = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix}$, $\quad X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$, $\quad \frac{dx}{dt} = \begin{bmatrix} \frac{dx_1}{dt} \\ \frac{dx_2}{dt} \\ \frac{dx_3}{dt} \end{bmatrix}$

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 functions as illustrated in the code below:

```
#include "cml.h"

void main() {

    mfArray a, p, d;

    a = mfVCat(mfV(1,-1,-1), mfV(-1,1,-1), mfV(-1,-1,1));

    // Compute eigenvalues and eigenvectors
    mfEig(mfOut(p, d),a);

    // Display results
    mfDisplay(a,"a",d,"d",p,"p");

}
```

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

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

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 functions 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.

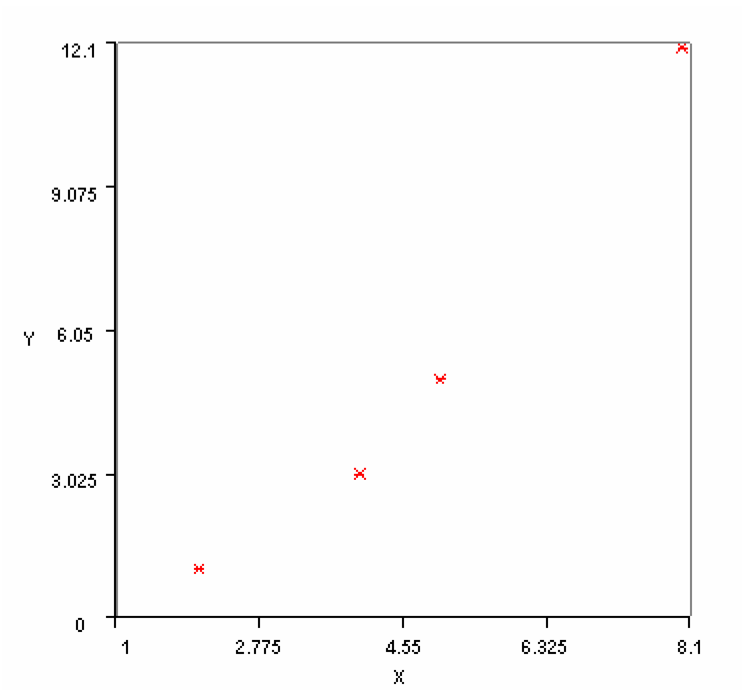


Figure 3.3.1 Plot of four data points (2,1), (4,3), (5,5) and (8,12).

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 shown below.

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

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 $Ar = b$ have 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 C++ Library matrix left division `mfLDiv` to solve for `r.` The following code uses linear algebra functions to obtain an approximation to the coefficient vector `r.`

```
#include "cml.h"
#include "fgl.h"

void main() {

    mfArray x, y, r,A,x1;

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

    // Initialize matrix A.
    x = mfV(2,4,5,8);
    A = mfHCat(mfOnes(4,1), x.T(), mfPow(x,2).T() );

    // Initialize vector y to contain the four data points
    y= mfV(1,3,5,12).T();

    // Plot the four data points using red *
    mfPlot(x,y,"r*");

    // Compute the solution to vector r.

    // r = mfMul(mfMul(mfInv(mfMul(.t.A, A)),.t.A), y)
    r = A.LDiv(y);

    mfDisplay(r, "r");


    // Using the result 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 = mfHCat(mfOnes(8,1), x.T(), mfPow(x,2).T());
    y = mfMul(A, r);

    // Stop the Graphics Viewer from erasing the first graph
    mfHold("on");

    // Plot the binomial equation using a blue line
    mfPlot(x,y,"b-");

    // Set the axis range
    mfAxis(1.0, 8.1, 0, 12.1);

    // Pause program to view the resulting graphics
    mfViewPause();
}
```

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 raw data from 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.

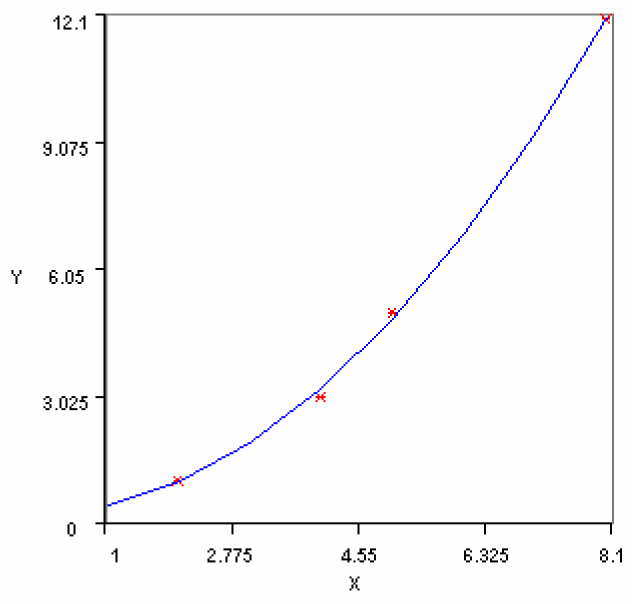


Figure 3.3.1 The polynomial equation and the four data points.

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 functions 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 MATFOR is capable of doing. A few examples will be provided to guide you through 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 functions. We shall begin by plotting a linear graph.

Step 1. Use MATFOR library in your program by adding preprocessor directives to include the MATFOR header files. You'll need to include `fgl.h` when using any of the visualization routines. For example,

```
#include "cml.h"
#include "fgl.h"

void main() {
```

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

```
mfArray x, y;

x = mfColon(-10,10);
y = mfPow(x,2);
```

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

```
mfFigure(1);
```

Step 4. Create a graph using one of the graph creation functions such as `mfPlot.` For example,

```
mfPlot(x,y);
```

Step 5. You can touch up the graph by using functions such as `mfShading,` `mfAxis,` and `mfBackGroundColor,` or annotate the graph by adding the axis labels and the title. By default, x-axis is labeled `'x'`, y-axis is labeled `'y'`, and z-axis is labeled `'z'`. For example,

```
mfTitle("y = x**2");
```

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

```
mfViewPause();
```

Step 7. Compile and run the program to view the graph as shown in Figure 4.1.

Step 8. When you have finished viewing your graph, press the Continue button on the toolbar to continue program execution.

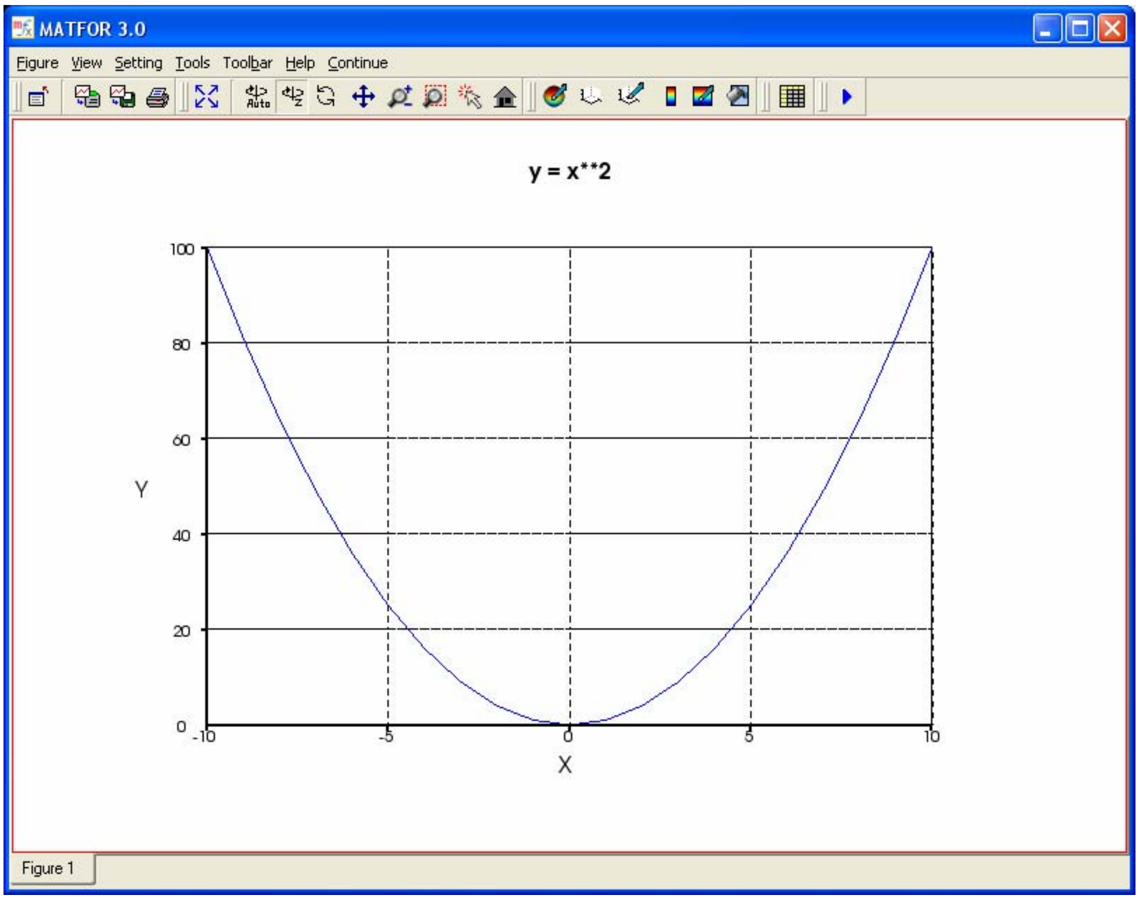


Figure 4.1 Plotting of y = cos(x)

Below is a summarization of the above codes.

Example 4.1 Steps to visualization

```
#include "cml.h"
#include "fgl.h"

void main() {

    mfArray x, y;

    x = mfColon(-10,10);
    y = mfPow(x,2);

    // Specify a new Graphics Viewer
    mfFigure(1);
```

```
        // Plot a 2-D x-y plot
        mfPlot(x,y);

        // Add a Title to the graph
        mfTitle("y = x**2");

        // Pause Program to view Graphics
        mfViewPause();

    }
```

4.2 MATFOR Graphics Viewer

When you use graph-creating functions, such as `mfPlot`, the created graphs will be displayed in MATFOR Graphics Viewer, as shown previously in Figure 4.1.

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

In this section, we shall briefly describe each of the components and their usage.

4.2.1 Window Frame and Figure Windows

The properties of the Window frame can be set using functions `mfWindowCaption`, `mfWindowSize,` and `mfWindowPos`.

MATFOR Graphics Viewer can contain a number of figure windows. Each figure window is attached to a window tab showing the ID and name of that particular figure window, which enables you to easily navigate through the figure windows.

4.2.2 Subplots

Each figure window can be further divided into multiple subplots by using the function `mfSubplot`. The function has the following syntax:

```
mfSubplot(m, n, p)
```

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

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

Example 4.2.2 Using mfSubplot

Create a new figure window with the ID 1.

```
mfFigure(1);
```

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

```
mfSubplot(1,2,1);
h = mfPlot(x, y1, "b");
mfHold("on");
h = mfPlot(x, y2, "ro");
mfGSet(h, "symbol_scale", 25);
mfCamZoom(0.8);
mfCamPan(20, 0);
```

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

```
mfSubplot(1,2,2);
h = mfPlot(x, y3, "gx");
mfGSet(h, "symbol_scale", 25);
mfCamZoom(0.8);
mfCamPan(40, 0);
```

Pause the program to view the graph.

```
mfViewPause();
```

Compile and execute the program.

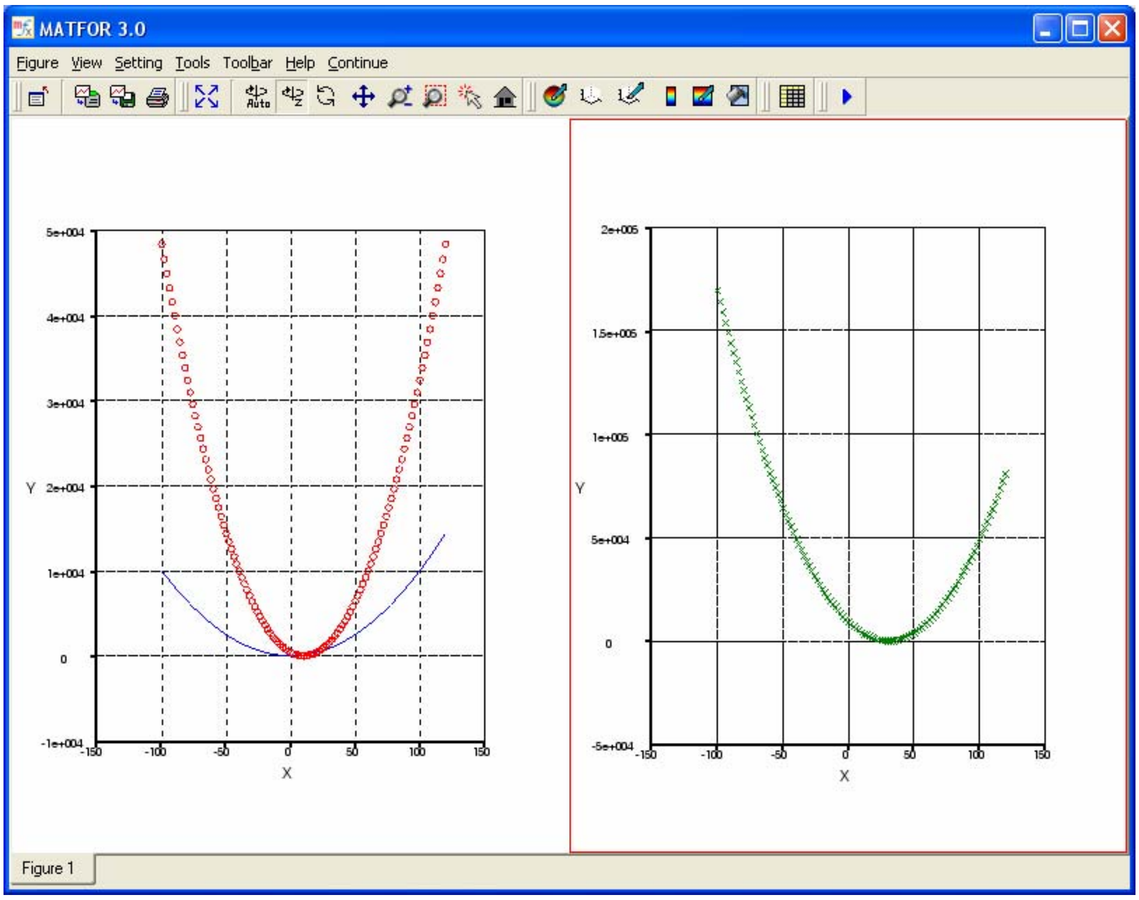


Figure 4.2.2 Two subplots in the same figure window

4.2.3 Menu and Toolbar

Graphics Viewer not only displays graphics objects, it also enables you to perform some graphics manipulations on the displaying objects through the menu and toolbar functions. All of the graphics object manipulations carried out using the function calls 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 manipulations after the program is run. Just play with the menu and toolbar functions to get a feel of this amazingly easy-to-use feature.

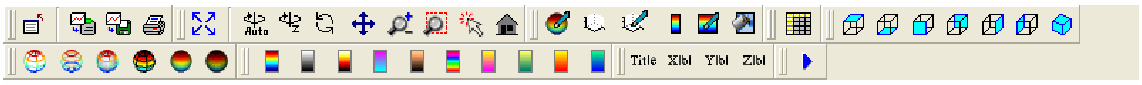
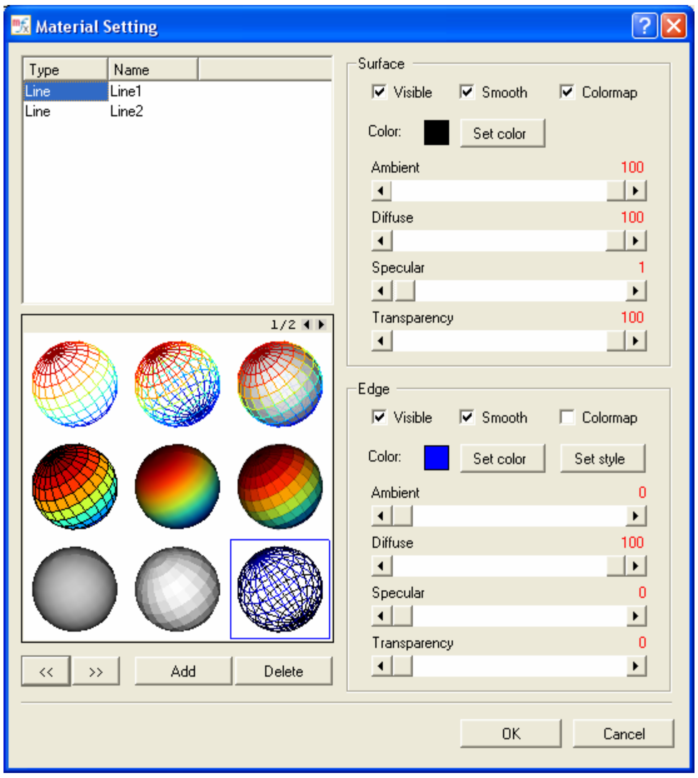


Figure 4.2.3.1 Toolbar

Material setting, axis setting, and colormap setting are manipulated using the special editors, as illustrated below in Figure 4.2.3.2, Figure 4.2.3.3, and Figure 4.2.3.4. They are located under the **Setting Menu**. You may refer to **Section 4.5 Colormap, Shading, and Texture** for examples of using the Material Setting and Colormap Setting editors.


Material Setting

Type	Name
Line	Line1
Line	Line2

1/2

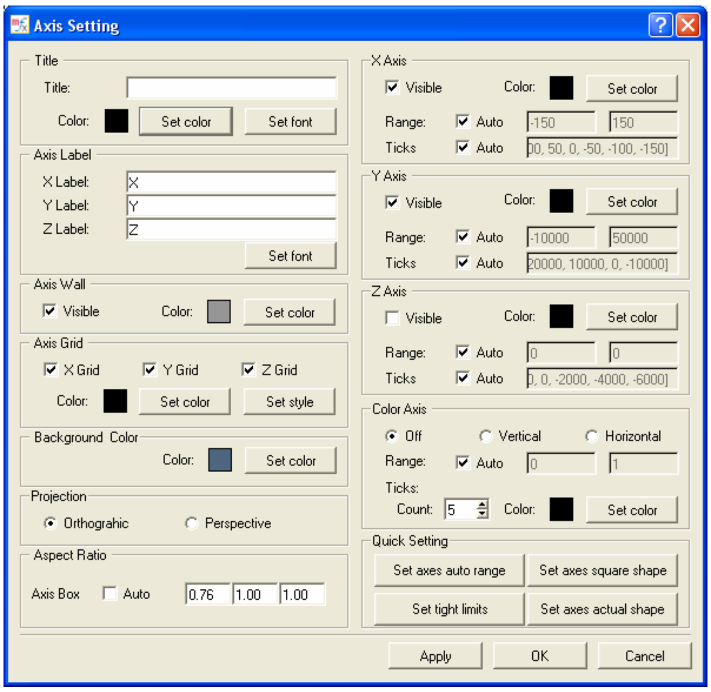
Surface: [✓] Visible [✓] Smooth [✓] Colormap
Color: ■ Set color
Ambient 100
Diffuse 100
Specular 1
Transparency 100

Edge: [✓] Visible [✓] Smooth [ ] Colormap
Color: 🟦 Set color Set style
Ambient 0
Diffuse 100
Specular 0
Transparency 0

<< >> Add Delete

OK Cancel

Figure 4.2.3.2 Material setting dialog box


Axis Setting

Title
Title:
Color: ■ Set color Set font

Axis Label
X Label: X
Y Label: Y
Z Label: Z
Set font

Axis Wall
[✓] Visible Color: ▨ Set color

Axis Grid
[✓] X Grid [✓] Y Grid [✓] Z Grid
Color: ■ Set color Set style

Background Color
Color: ▨ Set color

Projection
[✓ Orthograhic] Perspective

Aspect Ratio
Axis Box [ ] Auto 0.76 1.00 1.00

X Axis
[✓] Visible Color: ■ Set color
Range: [✓] Auto -150 150
Ticks [✓] Auto 00, 50, 0, -50, -100, -150]

Y Axis
[✓] Visible Color: ■ Set color
Range: [✓] Auto -10000 50000
Ticks [✓] Auto 20000, 10000, 0, -10000]

Z Axis
[ ] Visible Color: ■ Set color
Range: [✓] Auto 0 0
Ticks [✓] Auto), 0, -2000, -4000, -6000]

Color Axis
[✓ Off] Vertical Horizontal
Range: [✓] Auto 0 1
Ticks:
Count: 5 Color: ■ Set color

Quick Setting
Set axes auto range Set axes square shape
Set tight limits Set axes actual shape

Apply OK Cancel

Figure 4.2.3.3 Axis setting dialog box

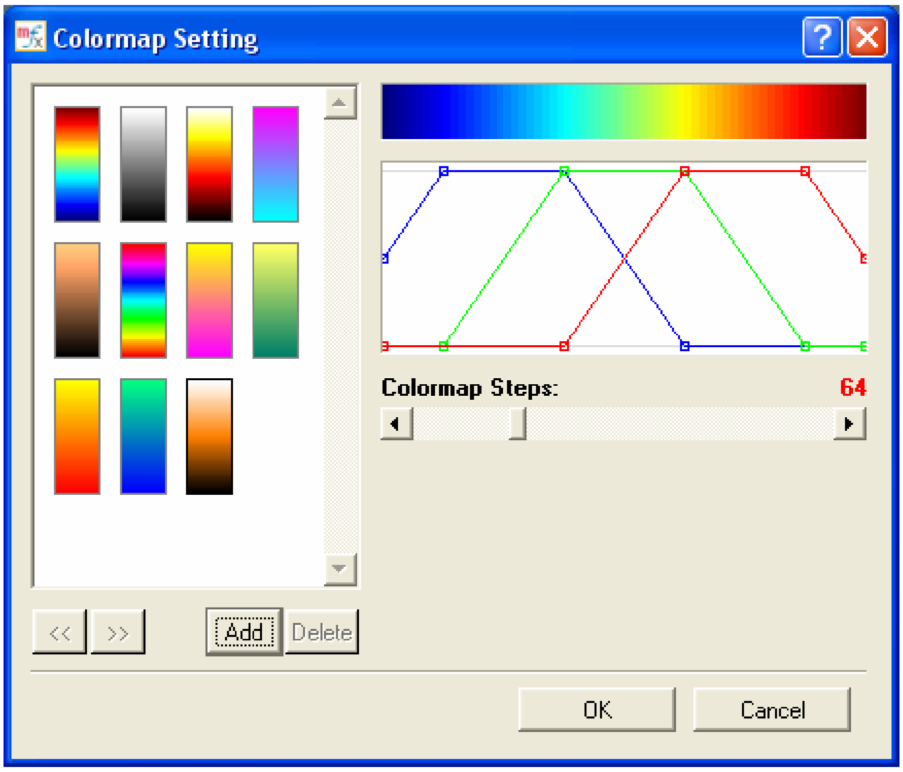


Figure 4.2.3.4 Colormap setting dialog box

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 the one shown in example 4.1. The other, is to read a data file into the system.

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

4.3.1 Generating the Data

First, we shall begin to generate the data by using `mfMeshgrid`, which is a useful function, to transform the domain specified by two vectors into two-dimensional matrices. The matrices are composed of repeating rows and columns of the two vectors.

The resulting matrices are useful for evaluating functions of two variables and for plotting surfaces. The syntax of the function is:

```
mfMeshgrid(mfOut(matrix X,matrix Y), vector x,
vector y)
```

As an example, we shall create four matrices `x`, `y`, `indxi`, and `indxj` using function `mfMeshgrid`. 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 mfMeshgrid- function of two variables

First, include the header files `cml.h` and `fgl.h` in your preprocessor directives. Construct the variables `x`, `y`, and `z` as mfArrays.

```
#include "cml.h"
#include "fgl.h"

void main() {

    mfArray m, x, y, z;
```

Create two 30-by-30 matrices `x`, `y` using function `mfMeshgrid`.

```
    m = mfLinspace(-3, 3, 30);
    mfMeshgrid(mfOut(x, y), m);
```

Function `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-0.5) + (y+0.5)*y + 1);
```

Using the data created above, we shall draw a surface graph in three-dimensional space using the function `mfSurf`.

```
mfSurf(x, y, z);
mfViewPause();
```

Compile and run the program.

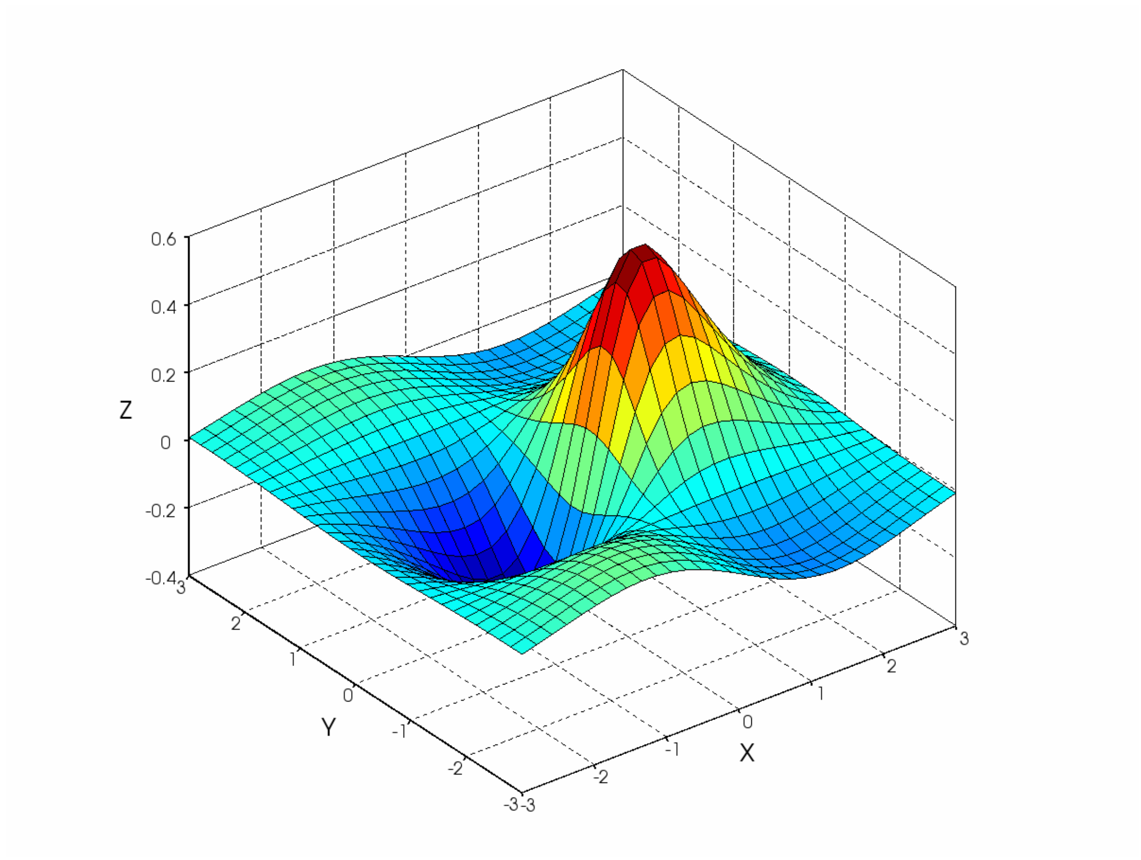


Figure 4.3.1 Surface graph in three-dimensional space

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 plot it in three-dimensional space.

Example 4.3.2 mfLoadAscii- loading ascii data files

Include the header files `cml.h` and `fgl.h` in your preprocessor directives. Construct the variables `xyz` and `tri` as mfArrays.

```
#include "cml.h"
#include "fgl.h"

void main() {
    mfArray xyz, tri;
```

Loads the data from the ASCII files dolphin_tri.txt and dolphin_xyz.txt using function `mfLoadAscii`. The example data files can be found in the directory `<MATFOR>\examples\cpp_ug\data\`. The data are 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 function `mfTriSurf`. You may refer to **Section 5.6 Unstructured Mesh** for more details on plotting unstructured mesh graphs.

```
mfTriSurf(tri, xyz);
```

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

```
mfAxis("equal");
mfAxis("off");
mfViewPause();
```

Compile and run the program.

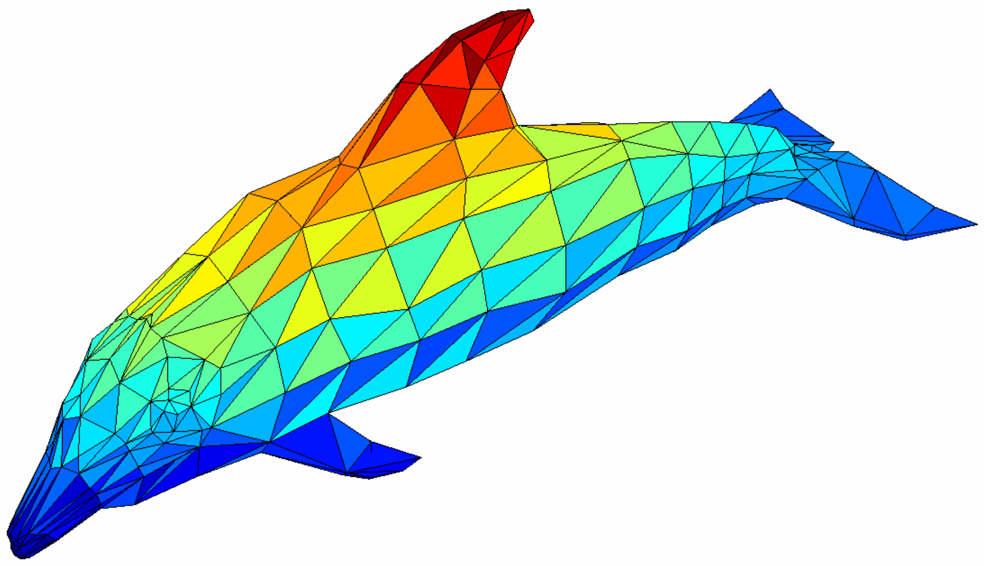


Figure 4.3.2 Dolphin object

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 view angle or scale 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 useful 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 function `mfView(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, `mfView(45, 45),` right below the surface plot function.

```
mfFigure("msView(45, 45)");
mfSurf(x, y, z);
mfView(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 `mfView(“3”)`.

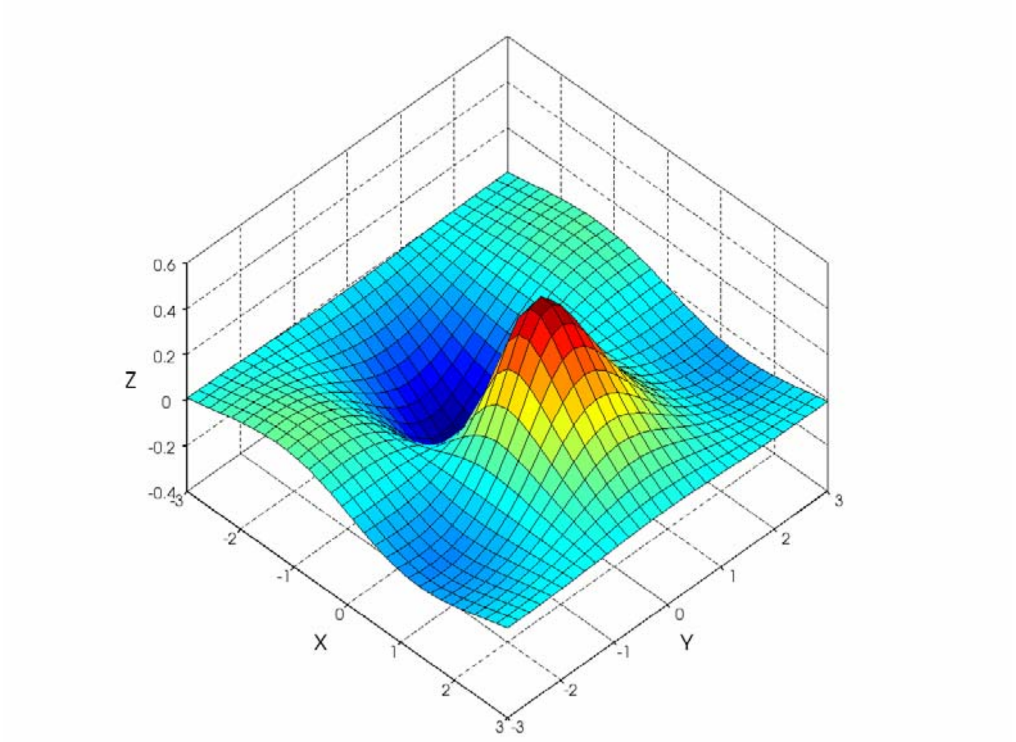


Figure 4.4.1 mfView – adjust the viewpoint

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.

Function `mfCamPan` 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 shifts downward in the plot space as the camera shifts 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.

```
mfFigure("msCamPan");
mfSubplot(1, 2, 1);
mfSurf(x, y, z);
mfSubplot(1, 2, 2);
mfSurf(x, y, z);
mfCamPan(0, 80);
```

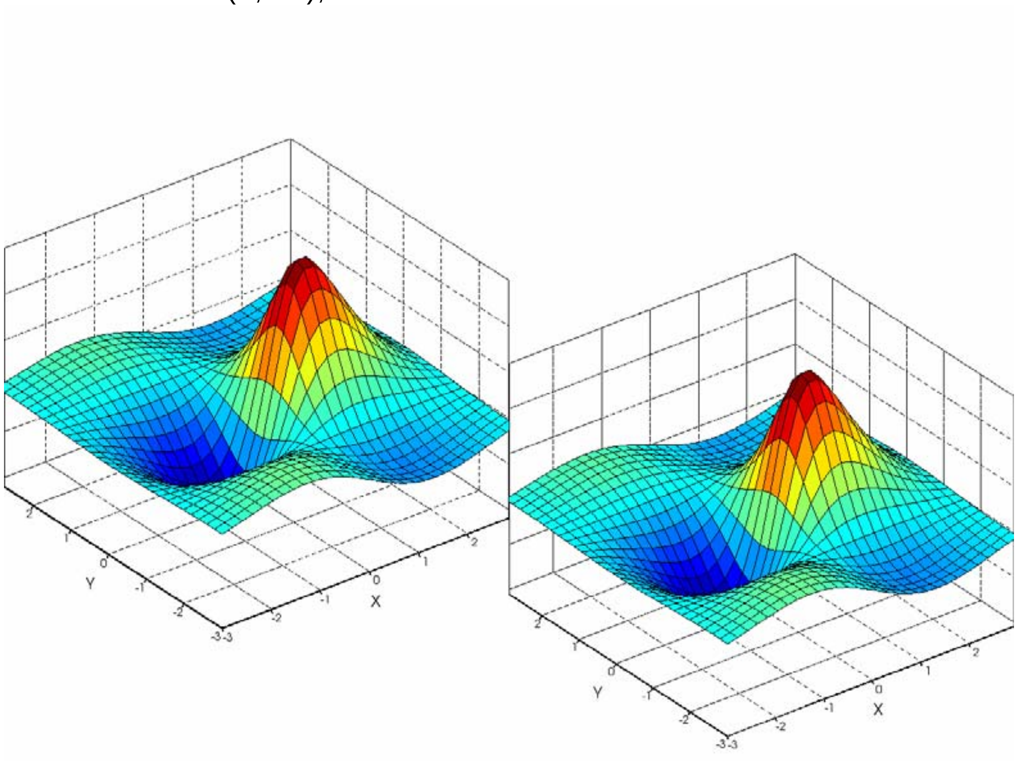


Figure 4.4.2 mfCamPan – shift the camera

4.4.3 Rescaling the Objects

Function `mfCamZoom` 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.

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

Example 4.4.3 Zooming in the surface object

```
mfFigure("msCamZoom");
mfSubplot(1, 2, 1);
mfSurf(x, y, z);
mfSubplot(1, 2, 2);
mfSurf(x, y, z);
mfCamZoom(1.5);
```

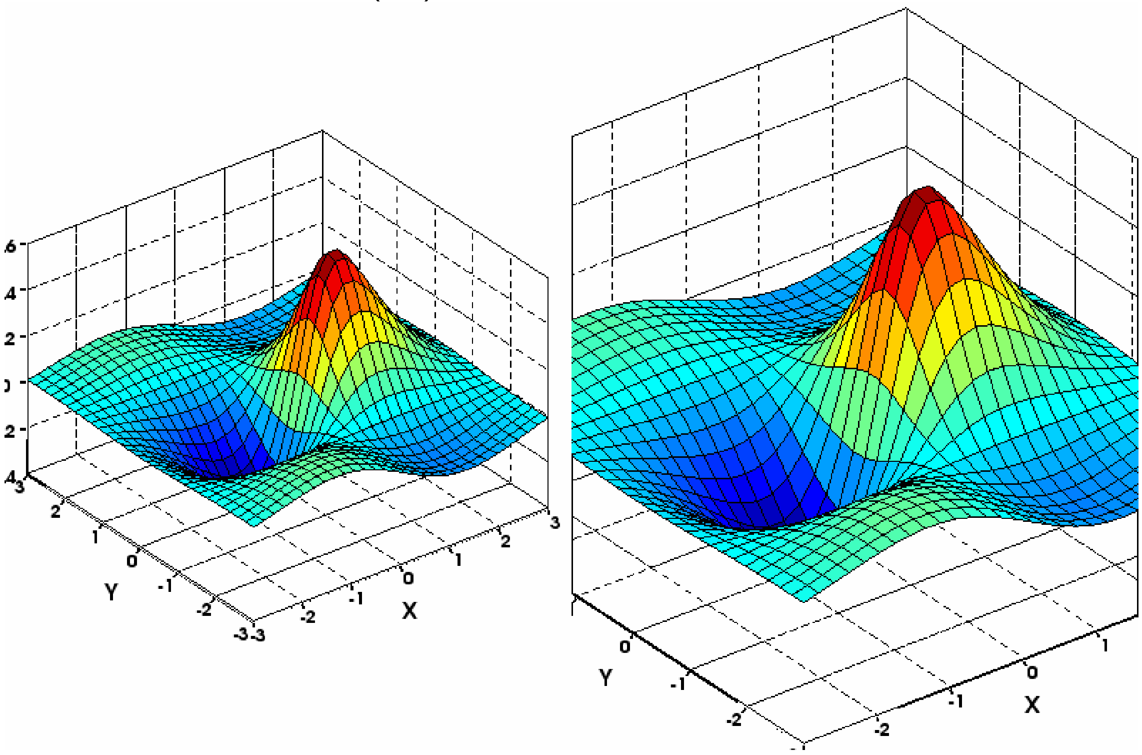


Figure 4.4.3 mfCamZoom – rescale the object

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 function `mfCamProj(mode)` that takes the input argument `mode`.

Example 4.4.4 Changing to the perspective displaying mode

The example code is as follows.

```
mfFigure("perspective");
mfSurf(x, y, z);
mfCamProj("perspective");
```

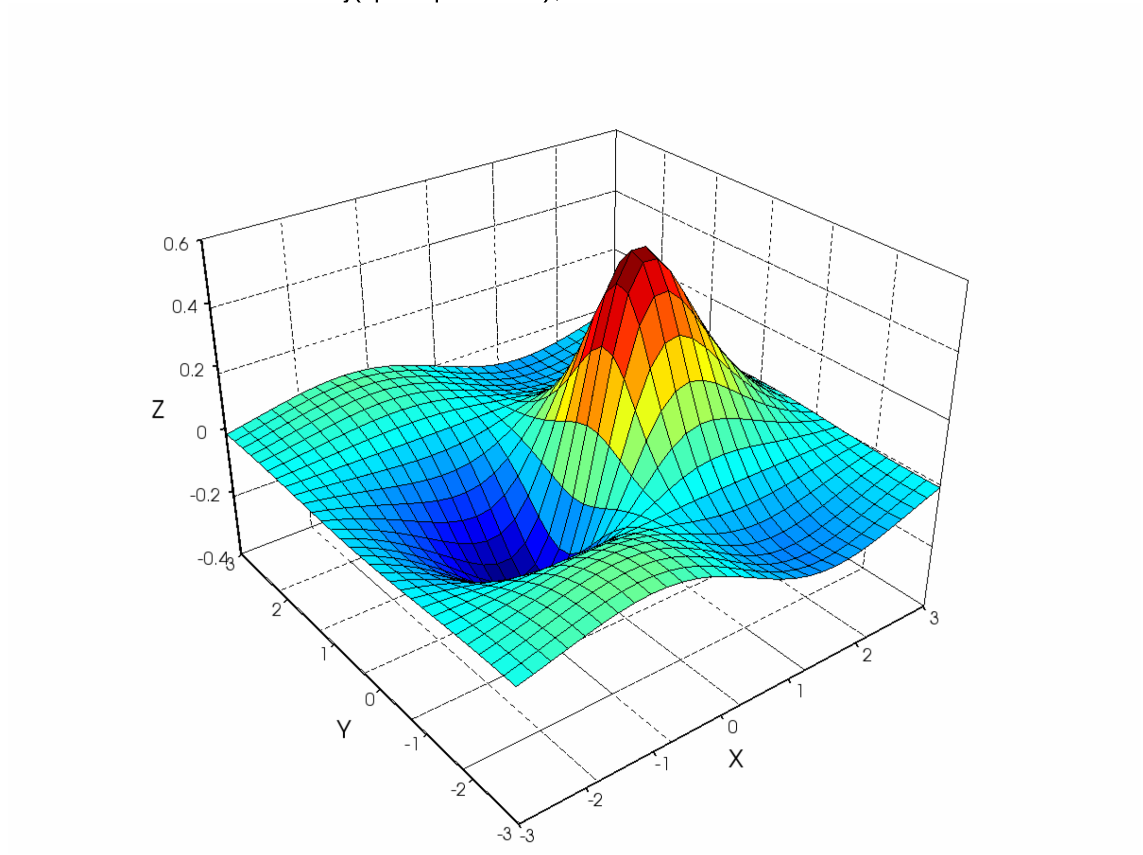


Figure 4.4.4 mfCamProj - change the displaying mode

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. As a whole, it looks like three adjacent sides of a box.

With functions `mfAxis`, `mfAxisWall,` and `mfAxisGrid`, you can set the range of the axes, number of ticks on the axes, color of the axis box, and the pattern of the displaying gridlines, etc.

Example 4.4.5 Adjusting the axis object

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

```
mfFigure("Axis");
mfSurf(x, y, z);
mfAxis(mf("xaxis_ticks"), mfV(3.0, 1.5, 0.0, -1.5, -3.0)) ;
mfAxisWall(mf("color"), mfV(1, 1, 1));
mfAxisGrid("pattern", "dotted");
```

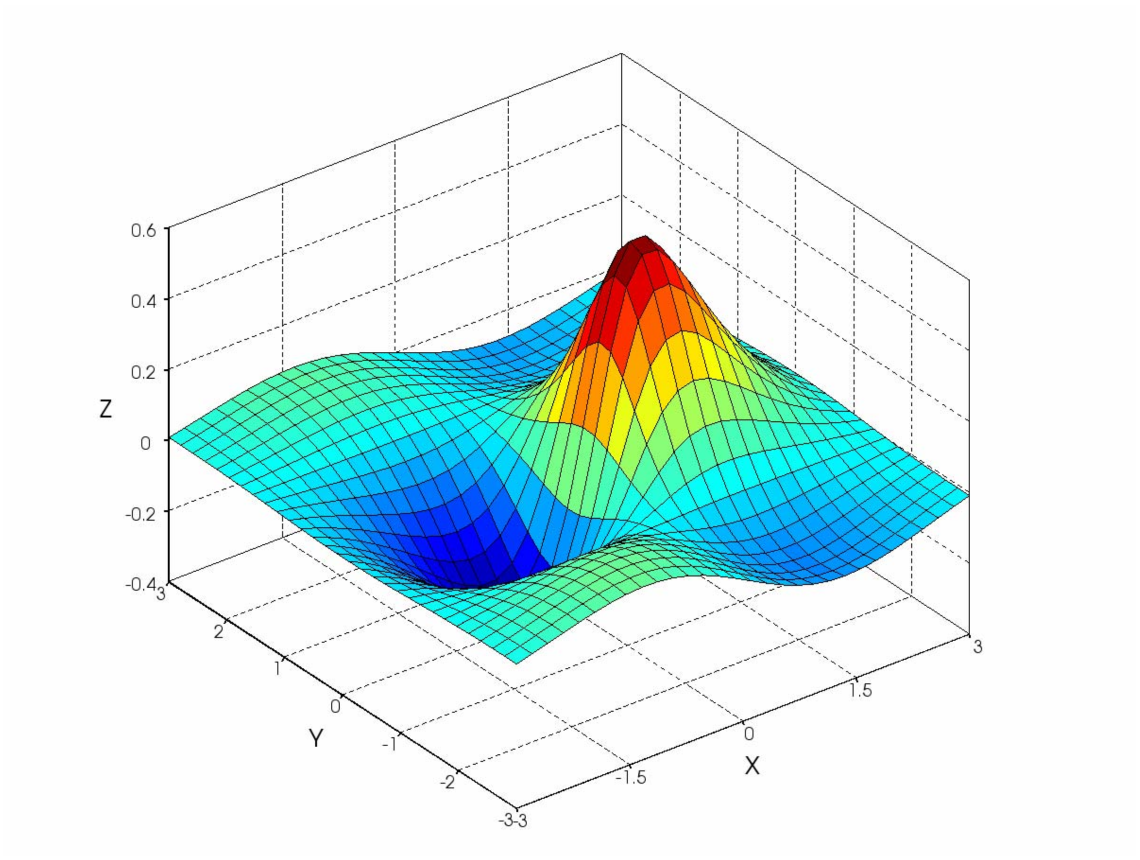


Figure 4.4.5 Set the axis object properties

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 using different types of colormaps. MATFOR provides a variety of predefined colormaps, such as jet, gray, hot, cool, cooper, hsv, etc. This can be accomplished by using the function `mfColormap`, which has the following syntax:

```
mfColormap(mode)
```

If none of the predefined colormaps meets your need, you can also define new sets of colormaps through the Colormap Setting dialog box.

Using the data constructed in example 4.3.1, we shall demonstrate how to render the surface with different predefined colormaps.

Example 4.5.1.1 Using predefined colormaps

Specify the colormap types as 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 produce.

```
//  Use colormap "jet"
mfSubplot(2, 2, 1);
mfTitle("jet");
h = mfSurf(x, y, z);
mfColormap("jet");
mfCamZoom(1.5);

//  Use colormap "hsv"
mfSubplot(2, 2, 2);
mfTitle("hsv");
h = mfSurf(x, y, z);
```

```
mfColormap("hsv");
mfCamZoom(1.5);
//  Use colormap "cool"
mfSubplot(2, 2, 3);
mfTitle("cool");
h = mfSurf(x, y, z);
mfColormap("cool");
mfCamZoom(1.5);

//  Use colormap "gray"
mfSubplot(2, 2, 4);
mfTitle("gray");
h = mfSurf(x, y, z);
mfColormap("gray");
mfCamZoom(1.5);
```

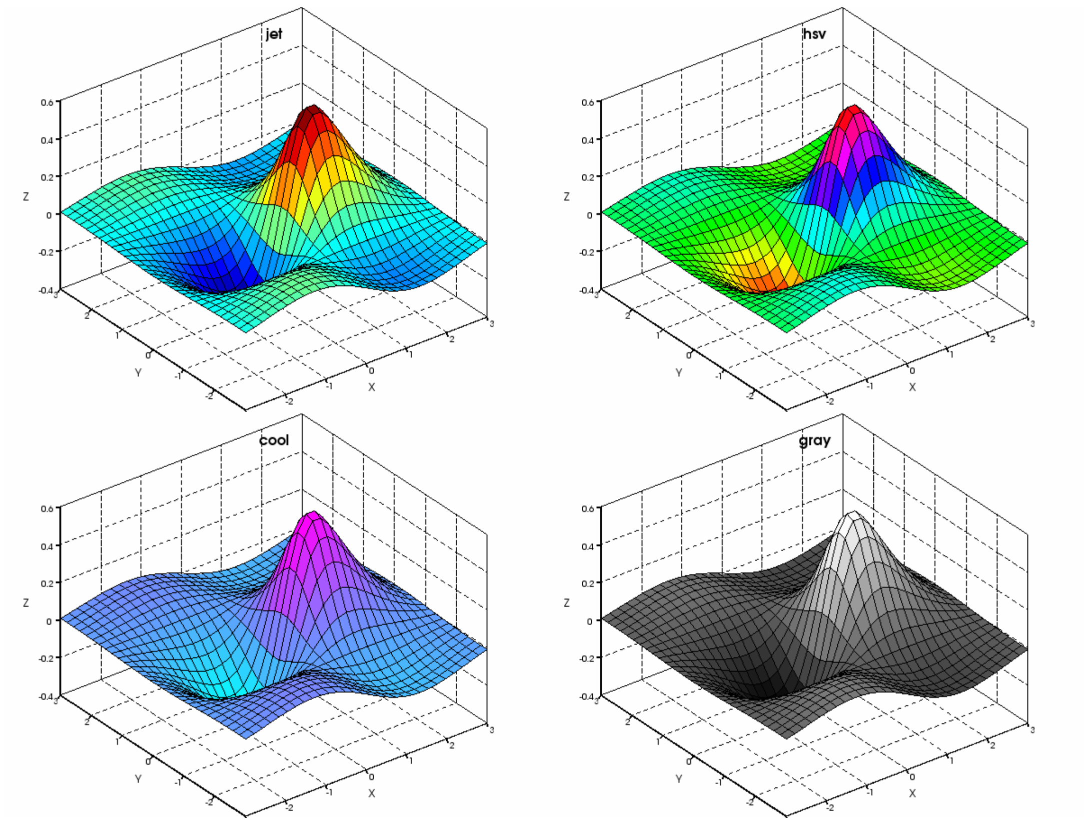


Figure 4.5.1.1 Different types of predefined colormaps

You can also manually define the colormaps by using the Colormap Setting dialog box. The following example is a tutorial showing you how to define a colormap by using the Colormap Editor.

Example 4.5.1.2 Using colormap editor

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

The editing box is located on the right hand side of the Colormap Editor. The relationship between the editing box and the graphics objects is simple. The leftmost part of the box is mapped to the part of the graphics object that has the lowest value and the rightmost part is mapped to the part of the graphics object that has the greatest value.

The following outlines the general steps for using the Colormap Editor.

Step 1. Start by running example 4.5.1. Select any one of the four subplots and extend it by clicking on the extend subplot button on the toolbar.

Step 2. Open the Colormap Setting editor that can be found on the toolbar or under the **Setting Menu**.

Step 3. Choose the predefined colormap gray to begin with. Notice that the three lines are on top of each other.

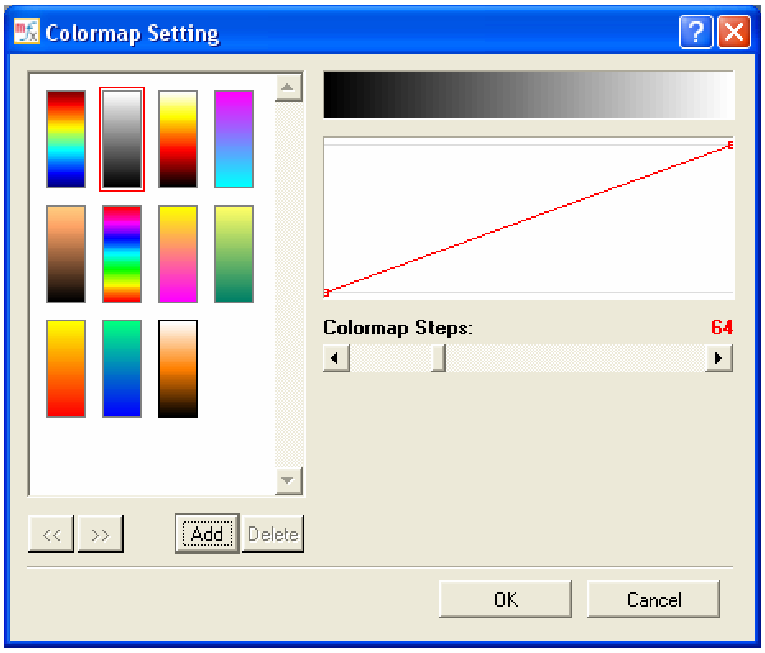


Figure 4.5.1.2 Colormap editor

Step 4. Drag the red line and green line to the bottom of the editing box and leave the blue line unchanged.

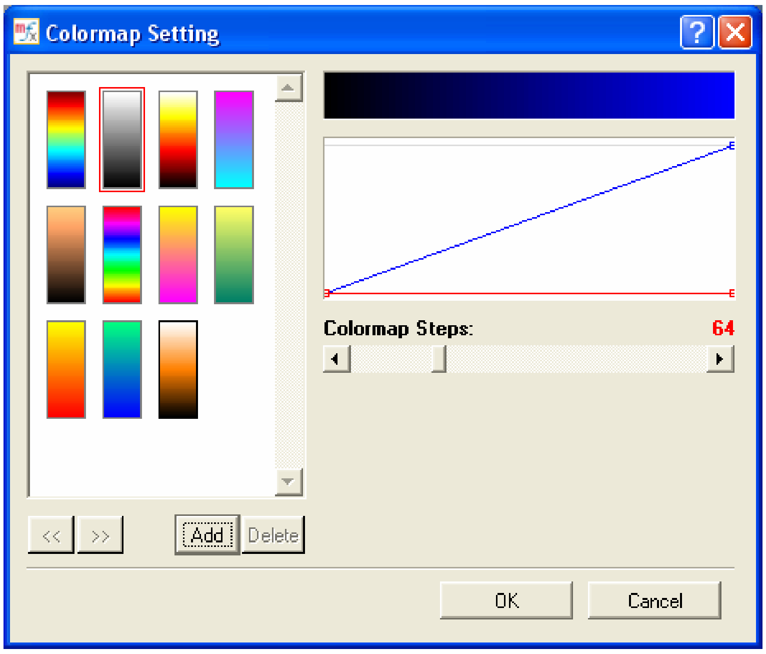


Figure 4.5.1.3 Colormap of blue components only

Step 5. Finally, click on the OK button. The resulting graph shows in Figure 4.5.1.4

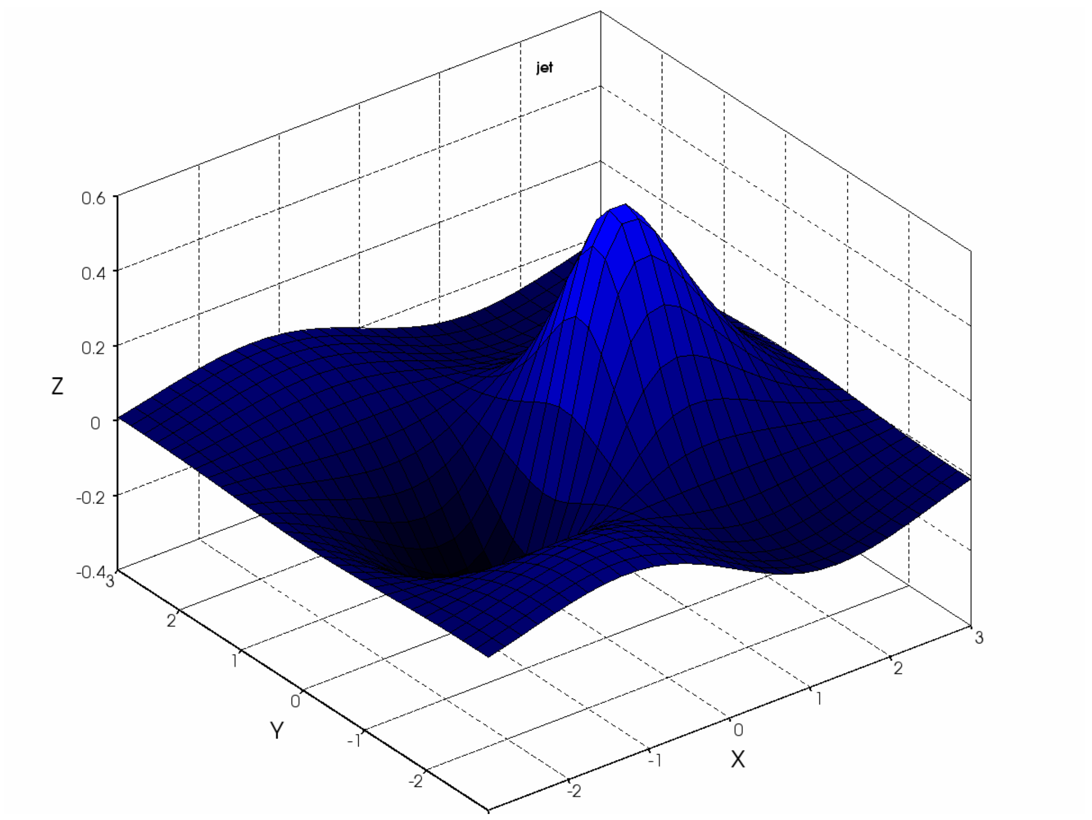


Figure 4.5.1.4 The resulting graph

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 on the figure window vertically or horizontally.

The vertical colorbar is displayed on the right hand side of the graph, whereas the horizontal colorbar is displayed on the bottom of the figure window.

The colorbar function `mfColorbar` has the following syntax:

```
mfColorbar(mode)
```

or

```
mfColorbar(property, value)
```

Where mode specifies the area to display the colorbar. It can be *'vert'*, *'horz'*, *'on'*, or *'off'*.

The number and color of labels can be adjusted using the properties "label_count" and "label_color".

4.5.3 Shading Objects

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

There are two ways to perform the shading. One is through the function `mfDrawMaterial` and the other one is through the Material Setting editor.

The general syntax of *mfDrawMaterial* is:

```
mfDrawMaterial(handle, target, property,
value)
```

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

The shading properties range 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, namely 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 mfDrawMaterial

First, we shall add a lighting effect on the surface to make it looks more three-dimensional, which is done by adjusting the ambient, diffuse, and specular reflectances.

```
mfDrawMaterial(h, "surf", "visible", "on",
        "smooth", "on",
        "colormap", "on",
        "ambient", 0,
        "diffuse", 75,
        "specular", 25);
```

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

```
mfDrawMaterial(h, "edge", "color", mfV(1,0,0),
        "smooth", "on",
        "colormap", "off",
        "ambient", 0,
        "diffuse", 0,
        "diffuse", 0,
        "specular", 0,
        "trans", 90);
```

Compile and run the program.

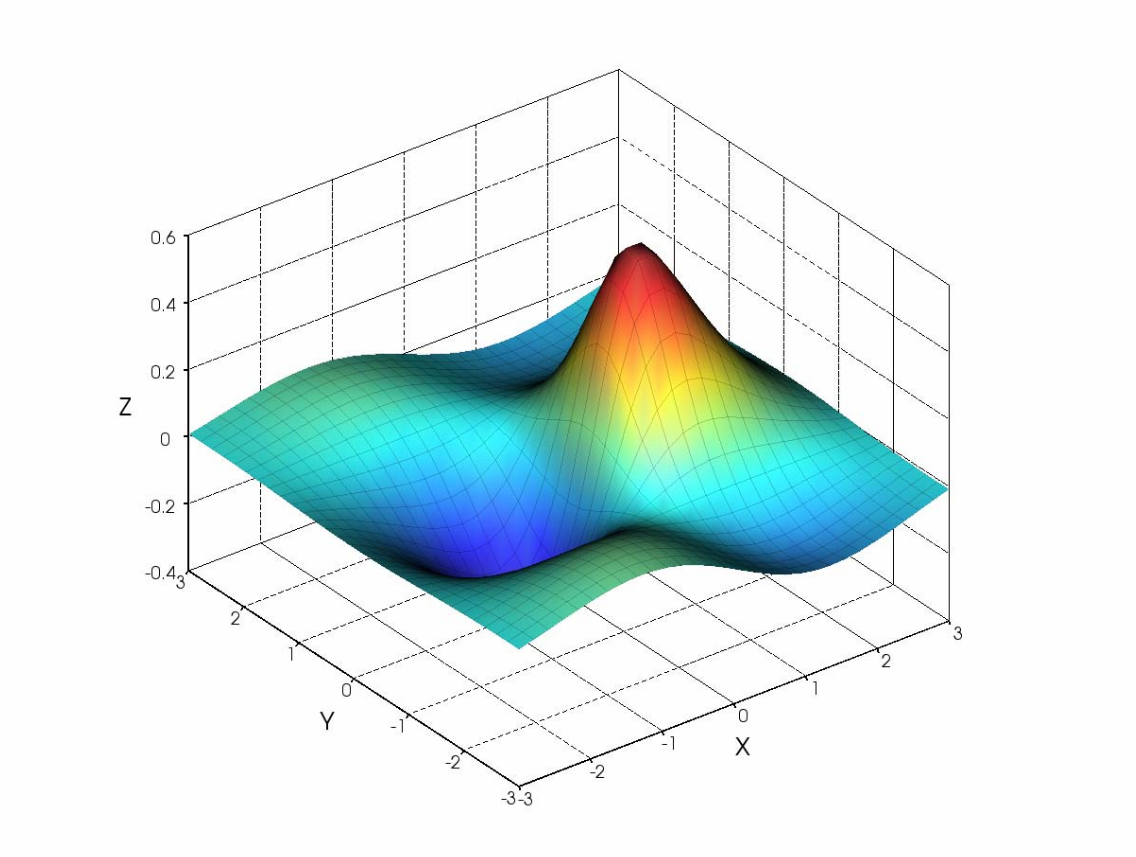
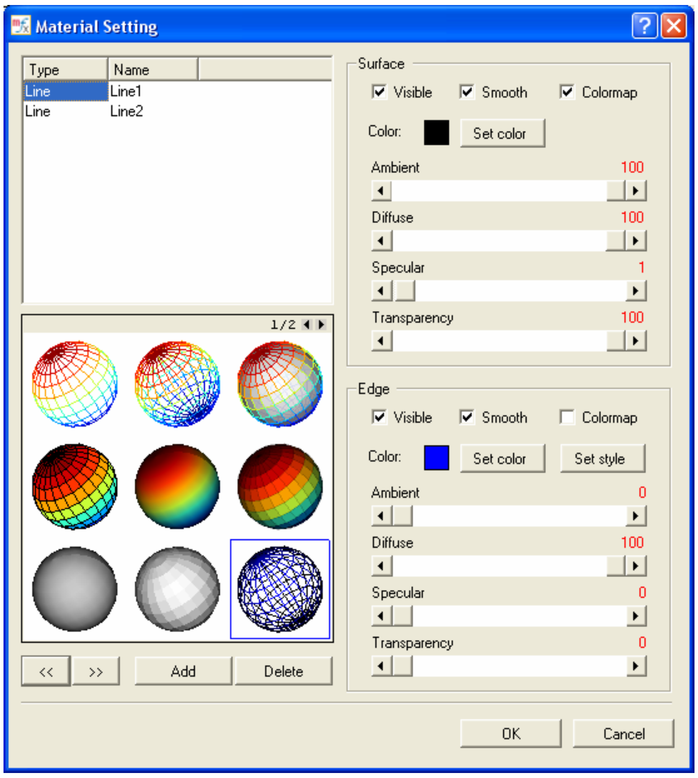


Figure 4.5.3.1 Shading the surface and edge components

The Material Shading Editor, as illustrated in Figure 4.5.3.2, can be invoked from the **Setting Menu** or by clicking the Material Setting icon on the toolbar. Through the editor, you can perform real-time material shading on the displaying graphics objects.


Material Setting

Type	Name
Line	Line1
Line	Line2

1/2

Surface

[✓] Visible [✓] Smooth [✓] Colormap

Color: ■ Set color

Ambient	100
Diffuse	100
Specular	1
Transparency	100

Edge

[✓] Visible [✓] Smooth [ ] Colormap

Color: 🟦 Set color Set style

Ambient	0
Diffuse	100
Specular	0
Transparency	0

Figure 4.5.3.2 Material shading editor

The upper left part of the editor is a table listing all the graphics objects that are currently displaying on the Graphics Viewer. You can simply target a specific graphics object for shading by clicking on the corresponding name.

The bottom left part of the editor is a table that shows all the predefined shading configurations. The one in the blue box is the shading configuration that is currently applied to the graphics object.

Similar to using the function `mfDrawMaterial`, the editor enables you to perform all the shading manipulations on the surface and edge components independently. The manipulations include switching the component on and off, applying the shading configuration, changing the color or shading method (smooth shading), and adjusting the ambient, diffuse, and specular reflectances.

Once you have configured a new shading configuration, you can simply add it to the list by clicking on the **Add** button and it will be appended to the next available slot in the table. The first one added will be located on the second page of the shading configuration table.

4.5.4 Mapping Texture

MATFOR also enables you to place a texture on the graphics object by mapping the texture's coordinates to the object's coordinates.

Texture's coordinates are comprised of two coordinates, s-coordinate and t-coordinate, which 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.

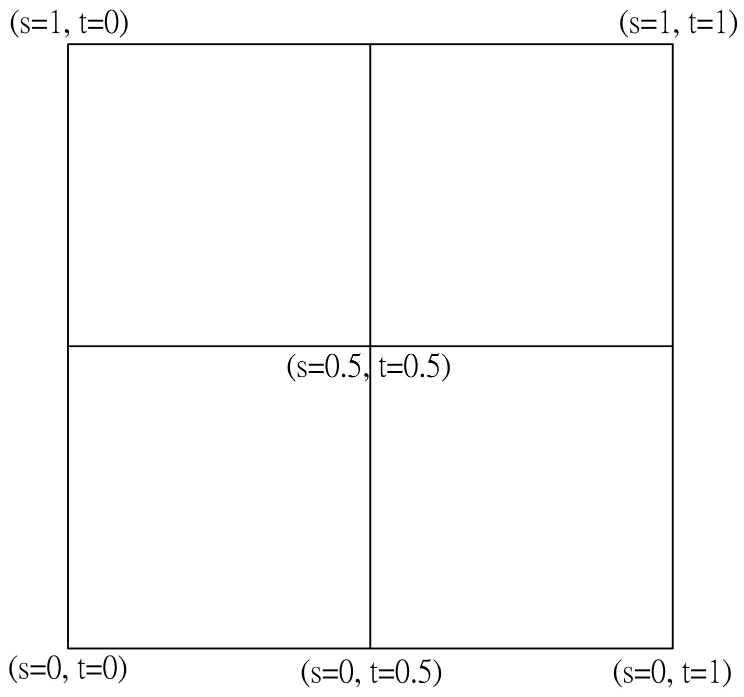


Figure 4.5.4.1 Texture coordinates

You may perform the mapping using the function mfDrawTexture, which has the following syntax:

```
mfDrawTexture(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 function *mfTriSurf* and place a texture on it.

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. Notice that each row of mfArray `tri` defines a face of the tetrahedron.

Example4.5.4 mfDrawTexture

```
#include <math.h>
#include "cml.h"
#include "fgl.h"

void main() {

    mfArray tri, x, y, z, h, s, t;
    double p1[3], p2[3], p3[3], p4[3];
    double q1[2], q2[2], q3[2], q4[2], q5[2], q6[2];

    p1[0] = -0.25 * sqrt(3.0);
    p1[1] = 0.5;
    p1[2] = 0.0;
    p2[0] = -0.25 * sqrt(3.0);
    p2[1] = -0.5;
    p2[2] = 0.0;
    p3[0] = 0.25 * sqrt(3.0);
    p3[1] = 0.0;
    p3[2] = 0.0;
    p4[0] = 0.0;
    p4[1] = 0.0;
    p4[2] = sqrt(6.0) / 3.0;

    x = mfT(mfV(p1[0], p2[0], p3[0], p1[0], p2[0], p4[0], p1[0], p3[0],
p4[0], p2[0], p3[0], p4[0]));
    y = mfT(mfV(p1[1], p2[1], p3[1], p1[1], p2[1], p4[1], p1[1], p3[1],
p4[1], p2[1], p3[1], p4[1]));
```

```
    z = mfT(mfV(p1[2], p2[2], p3[2], p1[2], p2[2], p4[2], p1[2], p3[2],
p4[2], p2[2], p3[2], p4[2]));
   tri = mfVCat(mfV(1, 2, 3), mfV(4, 5, 6), mfV(7, 8, 9), mfV(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[0] = 0.0;
    q1[1] = 0.0;
    q2[0] = 0.5;
    q2[1] = 0.0;
    q3[0] = 1.0;
    q3[1] = 0.0;
    q4[0] = 0.0;
    q4[1] = 0.5;
    q5[0] = 0.0;
    q5[1] = 1.0;
    q6[0] = 1.0;
    q6[1] = 1.0;

    s = mfT(mfV(q2[0], q4[0], q6[0], q2[0], q4[0], q1[0], q2[0], q6[0],
    q3[0], q4[0], q6[0], q5[0]));
   t = mfT(mfV(q2[1], q4[1], q6[1], q2[1], q4[1], q1[1], q2[1], q6[1], q3[1],
   q4[1], q6[1], q5[1]));
```

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 parts to be mapped onto the four faces of the tetrahedron. The bitmap file is located under the directory `<MATFOR>\examples\cpp_ug\data\`.

```
mfDrawTexture(h, "map", "./data/brick.bmp", "enable", "on", "coord_s", s,
"coord_t", t);
```

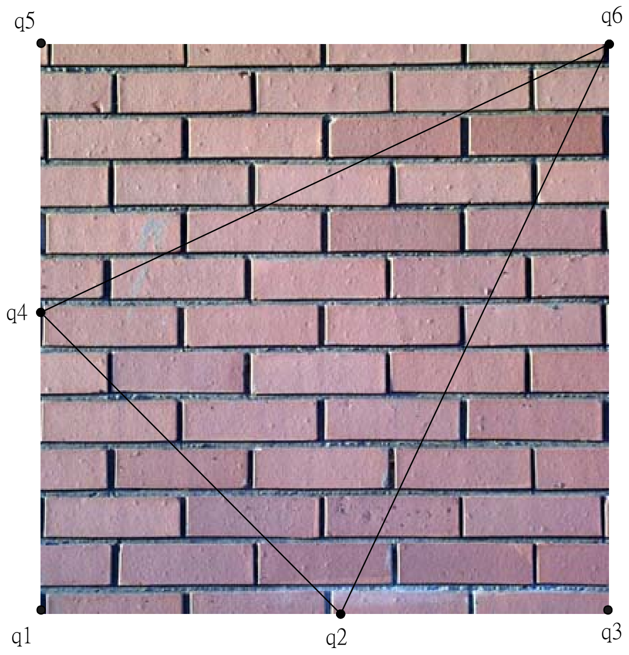


Figure 4.5.4.2 Brick.bmp

Figure 4.5.4.3 shows the result of the mapping.

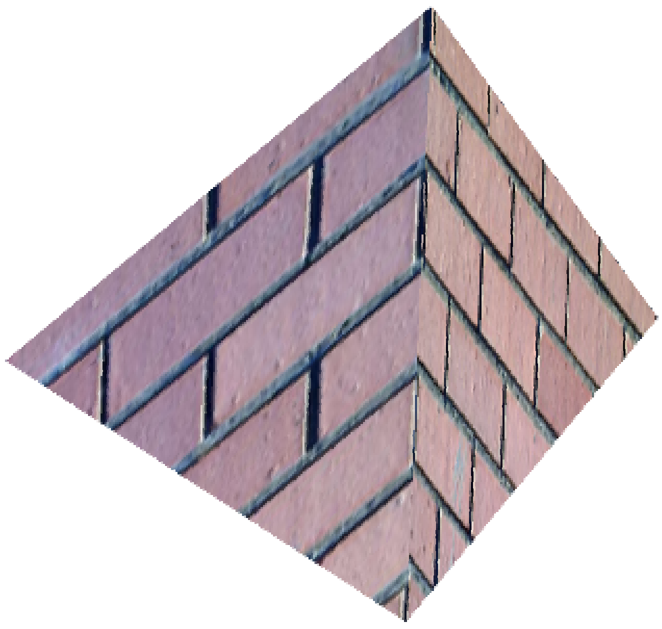


Figure 4.5.4.3 Map texture on the tetrahedron object

4.6 Annotating Your Graph

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

There are two ways to add a text annotation. Examples on 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 functions such as `mfXLabel`, `mfYLabel`, `mfZLabel,` and `mfTitle`, or through the Appearance Setting dialog box, which can be found in the **View Menu**.

In the following example, we shall demonstrate how to annotate a graph with 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.

```
mfTitle(mf("z = mfSin(x) * mfCos(y) / ( x*(x-0.5) + (y+0.5)*y + 1)"),
mfV(0, 0, 0), 16);
mfXLabel("indxi");
mfYLabel("indxj");
```

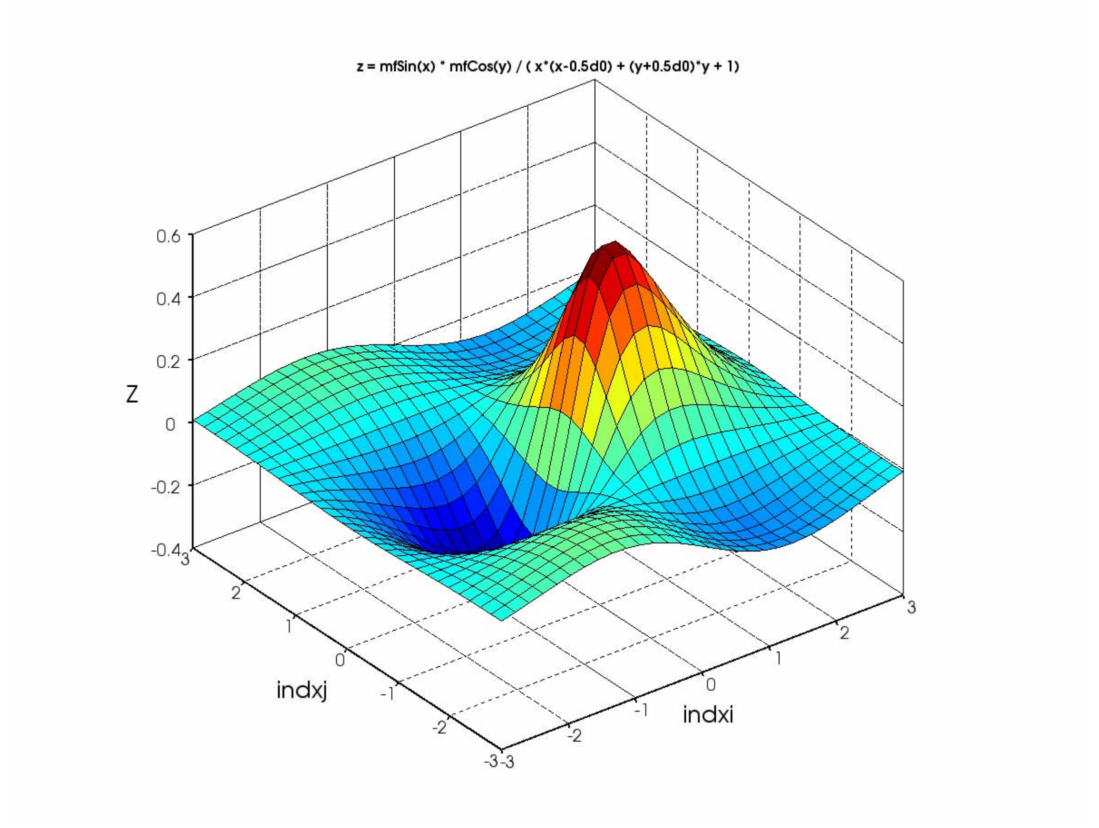


Figure 4.6.1 Axis labels and title

4.6.2 Text Annotation

You can add two-dimensional and three-dimensional text annotations with colors on your graph by using functions `mfText` and `mfAnnotation`, 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.

```
mfText("Annotating Your Graph", mfV(0.4, 0.9), mfV(1, 0, 1), 16);
```

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.

```
mfAnnotation("Maximum", mfV(0.7241, -0.1034, 0.5877), mfV(0, 0,
1));
```

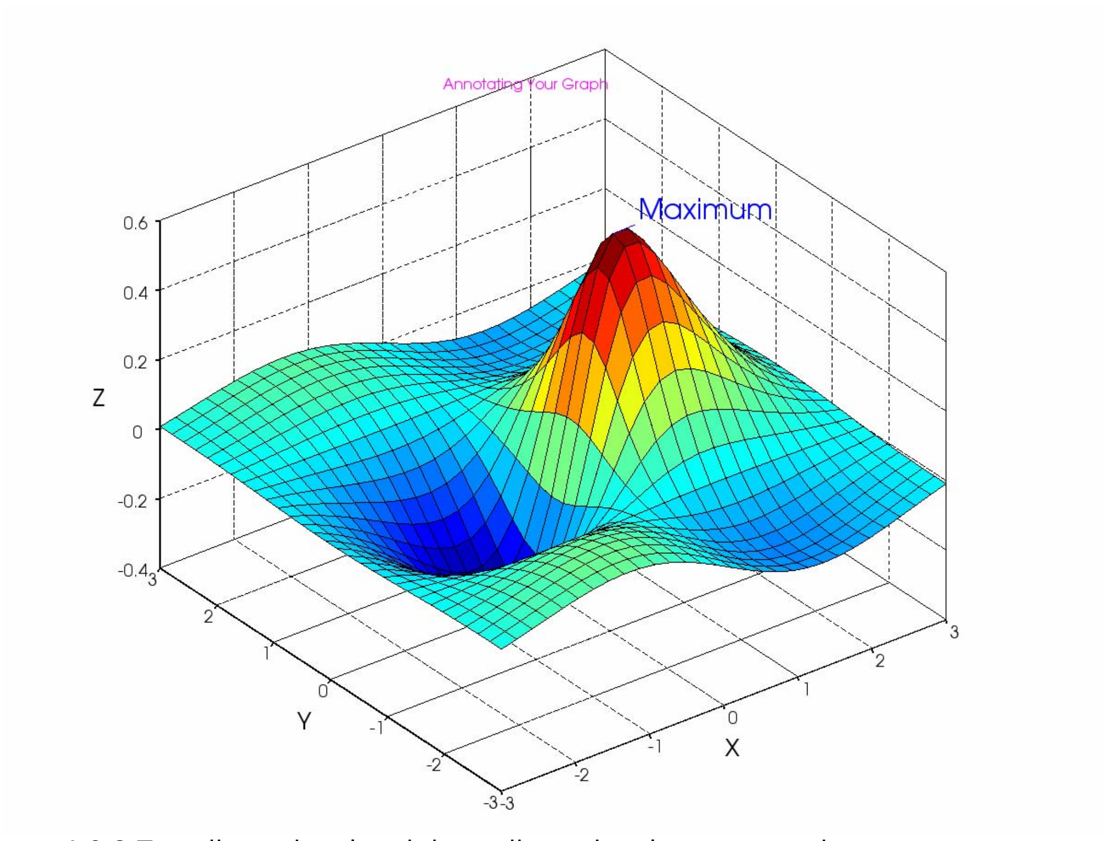


Figure 4.6.2 Two-dimensional and three-dimensional text annotations

4.7 Animation and Recording

The continuous erasing and updating of data displayed in the Graphics Viewer produce the animation effects. 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

```
mfGSet(handle, 'axis-data', data)
```

to update the targeted data of the current draw.

Step 5. Update the current Graphics Viewer by using function `mfDrawNow`. Animation effect is created.

Step 6. Pause the program after completing the animation to observe the static graph using function `mfViewPause`.

Example 4.7.1 Animation

Create the surface object by using the data with function `mfSurf` and use mfArray `h` to retrieve the handle of the object created.

```
mfSurf(x, y, z);
h = mfGetCurrentDraw();
```

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

```
for(i = 1; i <= 300; i++)
{
        z = mfSin(x+0.08*i) * mfCos(y-0.13*i) / ( x*(x-0.5) + (y+0.5)*y
        + 1);
        mfGSet(h, "zdata", z);
        mfDrawNow();
}
```

Pause the program to view the surface object at the point of termination using function `mfViewPause`.

Compile and run the program.

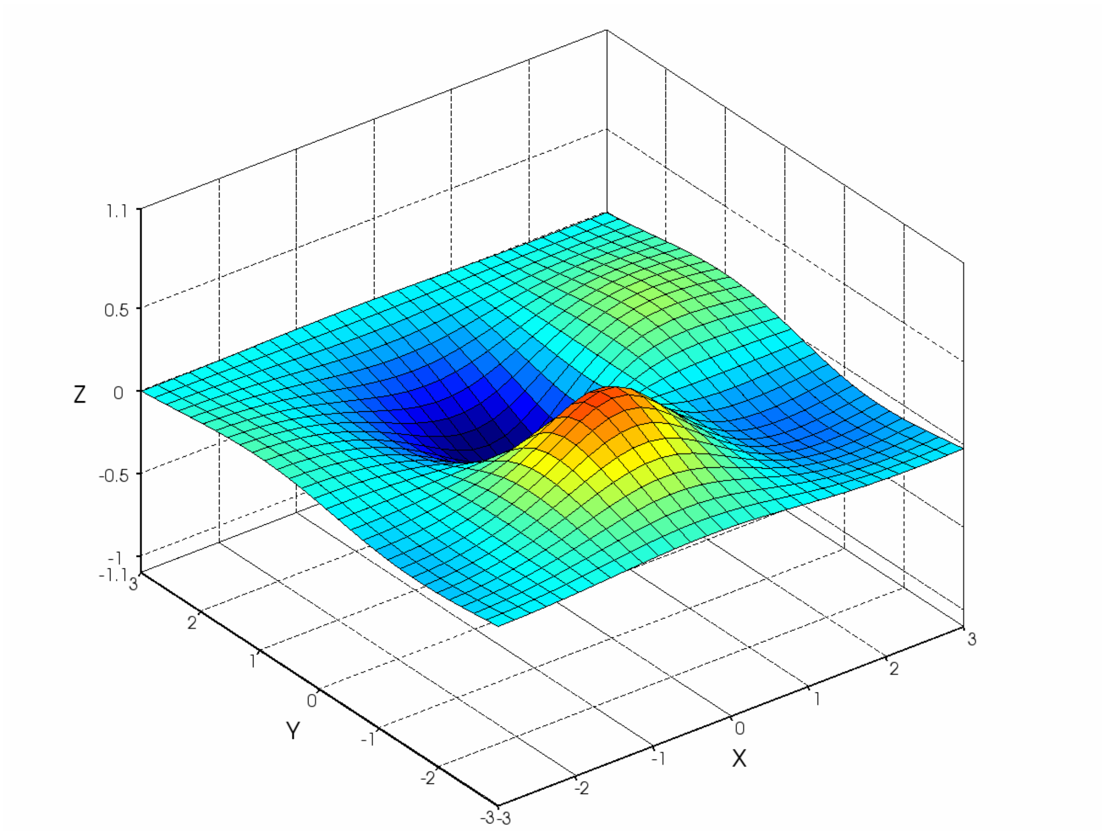


Figure 4.7.1 Animating surface object

4.7.2 Recording your animation

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

```
mfRecordStart('animation.avi')
```

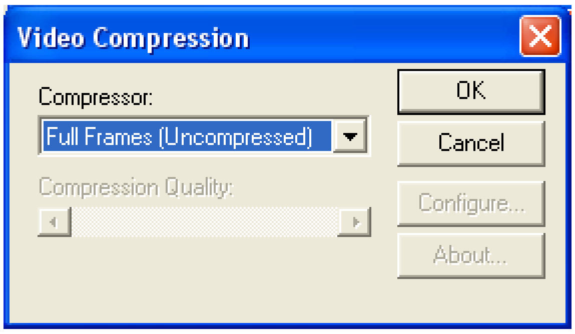
or

```
mfRecordStart('animation.bmp')

-------

<animation codes>


mfRecordEnd()
```

You have the options 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 while 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 `mfRecordStart('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.



Compressor: Full Frames (Uncompressed)

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 `mfRecordStart('.\data\Example4_7.avi')` and `mfRecordEnd()` at the beginning and end of the animation code respectively.

```
mfRecordStart("./Example4_7.avi");

mfSurf(x, y, z);
h = mfGetCurrentDraw();
// Reset the axis ranges to yield a better animation
mfAxis(-3.0, 3.0, -3.0, 3.0, -1.1, 1.1);
for(i = 1; i <= 300; i++)
{
        z = mfSin(x+0.08*i) * mfCos(y-0.13*i) / ( x*(x-0.5) + (y+0.5)*y
        + 1);
        mfGSet(h, "zdata", z);
        mfDrawNow();
}

mfRecordEnd();
mfViewPause();
```

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\cpp_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 function `mfExportImage`. A number of picture formats are supported (e.g. bmp, jpeg, tiff, ps and png). This 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.

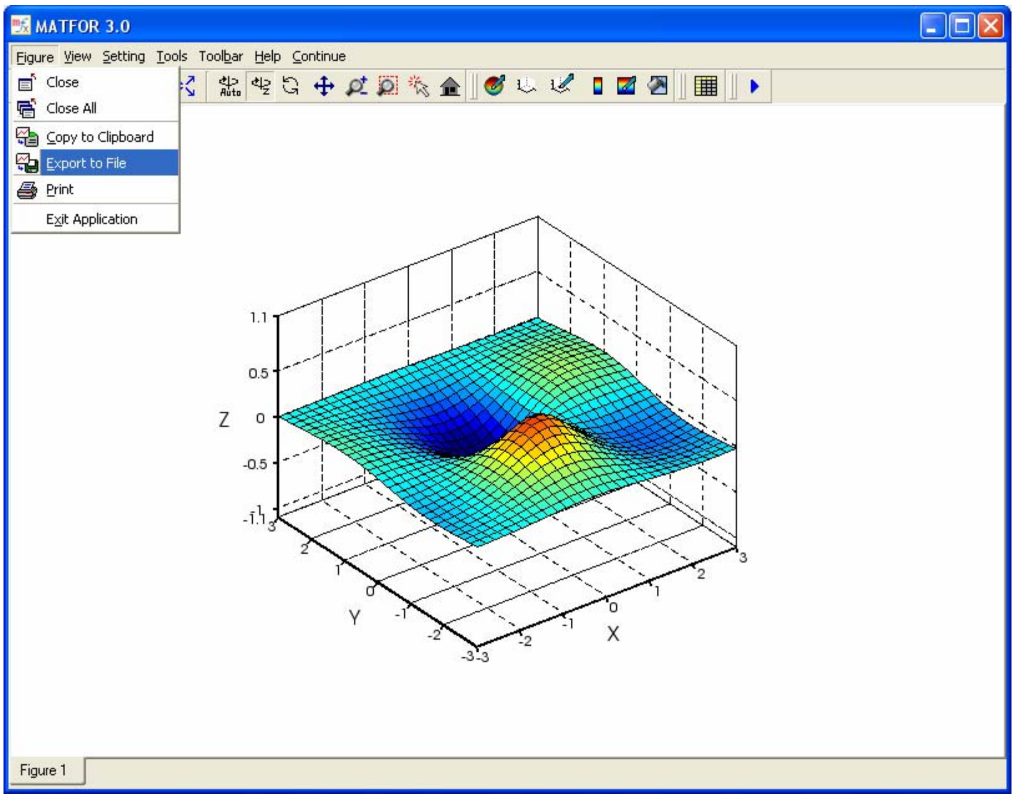
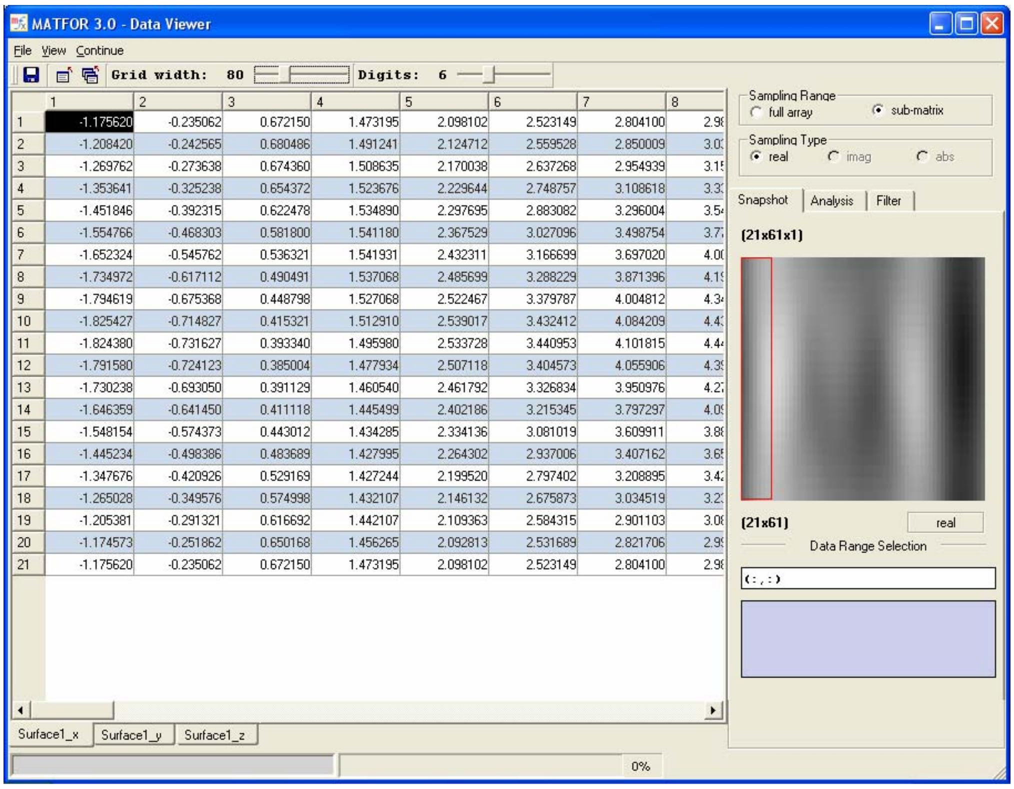


Figure 4.7.3 Capture displaying graph

4.8 MATFOR Data Viewer

MATFOR Data Viewer (as shown in Figure 4.8) 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.

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



Grid width: 80 Digits: 6

	1	2	3	4	5	6	7	8
1	-1.175620	-0.235062	0.672150	1.473195	2.098102	2.523149	2.804100	2.9
2	-1.208420	-0.242565	0.680486	1.491241	2.124712	2.559528	2.850009	3.0
3	-1.269762	-0.273638	0.674360	1.508635	2.170038	2.637268	2.954939	3.1
4	-1.353641	-0.325238	0.654372	1.523676	2.229644	2.748757	3.108618	3.3
5	-1.451846	-0.392315	0.622478	1.534890	2.297695	2.883082	3.296004	3.5
6	-1.554766	-0.468303	0.581800	1.541180	2.367529	3.027096	3.498754	3.7
7	-1.652324	-0.545762	0.536321	1.541931	2.432311	3.166699	3.697020	4.0
8	-1.734972	-0.617112	0.490491	1.537068	2.485699	3.288229	3.871396	4.1
9	-1.794619	-0.675368	0.448798	1.527068	2.522467	3.379787	4.004812	4.3
10	-1.825427	-0.714827	0.415321	1.512910	2.539017	3.432412	4.084209	4.4
11	-1.824380	-0.731627	0.393340	1.495980	2.533728	3.440953	4.101815	4.4
12	-1.791580	-0.724123	0.385004	1.477934	2.507118	3.404573	4.055906	4.3
13	-1.730238	-0.693050	0.391129	1.460540	2.461792	3.326834	3.950976	4.2
14	-1.646359	-0.641450	0.411118	1.445499	2.402186	3.215345	3.797297	4.0
15	-1.548154	-0.574373	0.443012	1.434285	2.334136	3.081019	3.609911	3.8
16	-1.445234	-0.498386	0.483689	1.427995	2.264302	2.937006	3.407162	3.6
17	-1.347676	-0.420926	0.529169	1.427244	2.199520	2.797402	3.208895	3.4
18	-1.265028	-0.349576	0.574998	1.432107	2.146132	2.675873	3.034519	3.2
19	-1.205381	-0.291321	0.616692	1.442107	2.109363	2.584315	2.901103	3.0
20	-1.174573	-0.251862	0.650168	1.456265	2.092813	2.531689	2.821706	2.9
21	-1.175620	-0.235062	0.672150	1.473195	2.098102	2.523149	2.804100	2.9

Sampling Range: full array [✓ sub-matrix]

Sampling Type: [✓ real] imag abs

(21x61x1)

(21x61) real

Data Range Selection

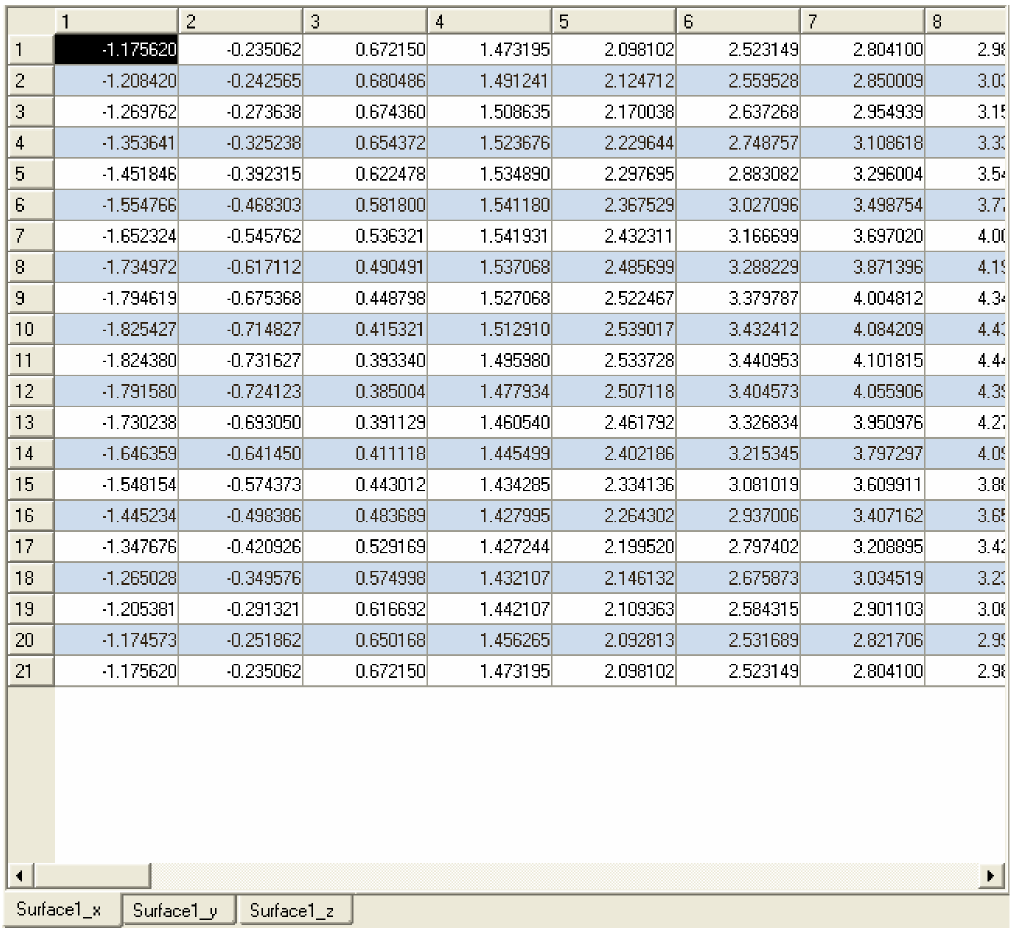
(:,:)

0%

Figure 4.8 MATFOR Data Viewer

4.8.1 Matrix Table

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



	1	2	3	4	5	6	7	8
1	-1.175620	-0.235062	0.672150	1.473195	2.098102	2.523149	2.804100	2.9
2	-1.208420	-0.242565	0.680486	1.491241	2.124712	2.559528	2.850009	3.0
3	-1.269762	-0.273638	0.674360	1.508635	2.170038	2.637268	2.954939	3.1
4	-1.353641	-0.325238	0.654372	1.523676	2.229644	2.748757	3.108618	3.3
5	-1.451846	-0.392315	0.622478	1.534890	2.297695	2.883082	3.296004	3.5
6	-1.554766	-0.468303	0.581800	1.541180	2.367529	3.027096	3.498754	3.7
7	-1.652324	-0.545762	0.536321	1.541931	2.432311	3.166699	3.697020	4.0
8	-1.734972	-0.617112	0.490491	1.537068	2.485699	3.288229	3.871396	4.1
9	-1.794619	-0.675368	0.448798	1.527068	2.522467	3.379787	4.004812	4.3
10	-1.825427	-0.714827	0.415321	1.512910	2.539017	3.432412	4.084209	4.4
11	-1.824380	-0.731627	0.393340	1.495980	2.533728	3.440953	4.101815	4.4
12	-1.791580	-0.724123	0.385004	1.477934	2.507118	3.404573	4.055906	4.3
13	-1.730238	-0.693050	0.391129	1.460540	2.461792	3.326834	3.950976	4.2
14	-1.646359	-0.641450	0.411118	1.445499	2.402186	3.215345	3.797297	4.0
15	-1.548154	-0.574373	0.443012	1.434285	2.334136	3.081019	3.609911	3.8
16	-1.445234	-0.498386	0.483689	1.427995	2.264302	2.937006	3.407162	3.6
17	-1.347676	-0.420926	0.529169	1.427244	2.199520	2.797402	3.208895	3.4
18	-1.265028	-0.349576	0.574998	1.432107	2.146132	2.675873	3.034519	3.2
19	-1.205381	-0.291321	0.616692	1.442107	2.109363	2.584315	2.901103	3.0
20	-1.174573	-0.251862	0.650168	1.456265	2.092813	2.531689	2.821706	2.9
21	-1.175620	-0.235062	0.672150	1.473195	2.098102	2.523149	2.804100	2.9

Figure 4.8.1 Matrix Table

The cells in the Matrix Table have different colors. Each cell color is interpreted differently. The default color (usually white) is used to represent entries in the odd rows whereas the sky blue color is used to represent entries in the even rows.

Through the Filter Panel, you can also define new colors for cells in order to emphasize a specific range of entries. Descriptions on using the Filter Panel can be found in Section 4.8.7.

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

4.8.2 Menu

The two menus in the Data Viewer are **File Menu** and **View Menu**, as illustrated in Figure 4.8.2.1 and Figure 4.8.2.2.

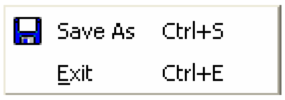


Figure 4.8.2.1 File Menu

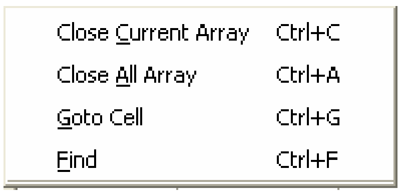


Figure 4.8.2.2 View Menu

You may notice that MATFOR Data Viewer does not have a file opening function. This is because it is not an independent application as all of the data is input passed from MATFOR Graphics Viewer or functionally generated.

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

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

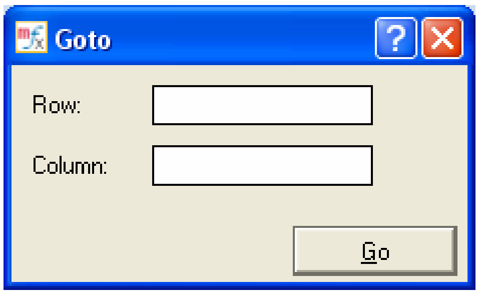
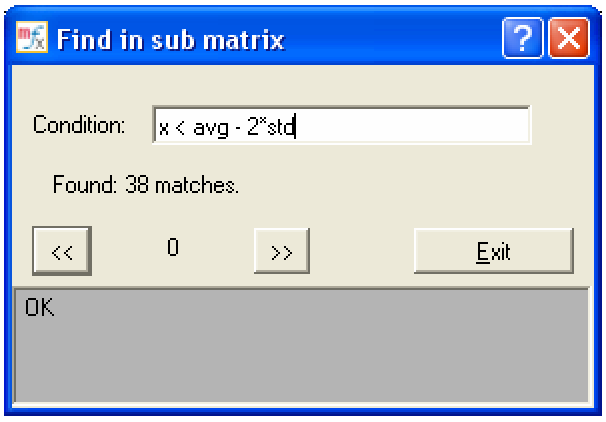


Figure 4.8.2.3 Goto Cell dialog box

The Find function enables you to find the entries that satisfy the condition you input. The format of the condition is similar to the one used in filtering. Refer to **Section 4.8.7 Filter Panel**.



x < avg - 2*std

Found: 38 matches.

OK

Figure 4.8.2.4 Find dialog box

4.8.3 Toolbar

MATFOR Data Viewer also provides you some quick buttons on the Toolbar for using the menu functions, as shown in Figure 4.8.3.

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

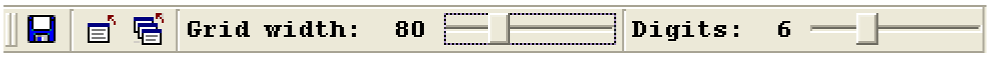

Grid width: 80
Digits: 6

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, it requires you to specify the range of entries you are retrieving. The range is in the following format:

```
(x-range, y-range, z-range)
```

For example, if the range selection is (:, :, 3), it will display 900 entries of the sub-matrix (1:30, 1:30, 3).

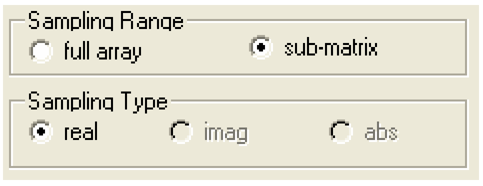

Sampling Range
○ full array ● sub-matrix
Sampling Type
● real ○ imag ○ abs

Figure 4.8.4 Range and type sampling

4.8.5 Snapshot Panel

The Snapshot Panel, as shown in Figure 4.8.5, 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**. In other words, the darker the cell color, the higher the value of the corresponding entry.

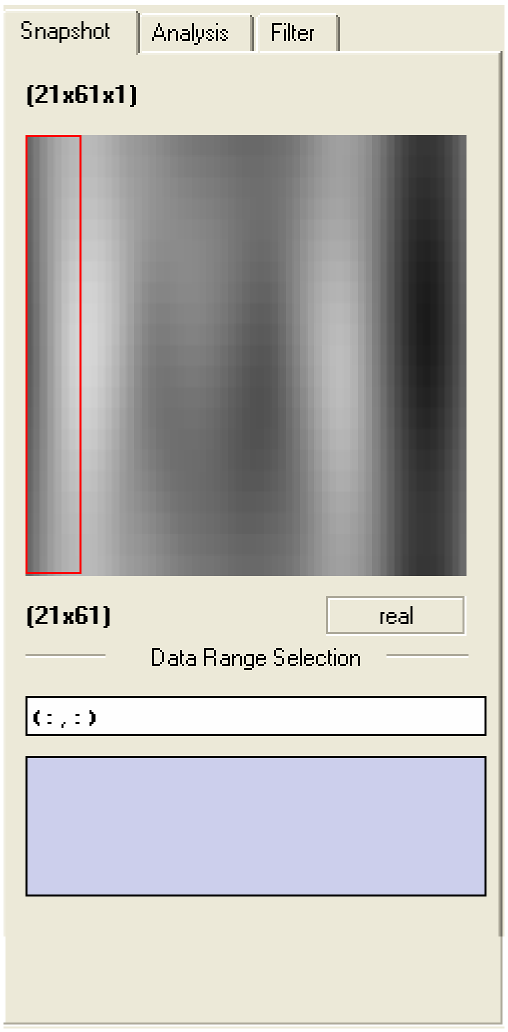


Figure 4.8.5 Snapshot Panel

In the snapshot window, the **range** is defined by setting the upper-boundary to be: average + 3 times the standard deviation, and setting the lower-boundary to be: average – 3 times the standard deviation. The values that exceed the upper-boundary are all treated as the maximum value and the values that are less than the lower-boundary are treated as the minimum value. The range is further divided into 256 shades, or levels, of darkness and the cells are drawn accordingly.

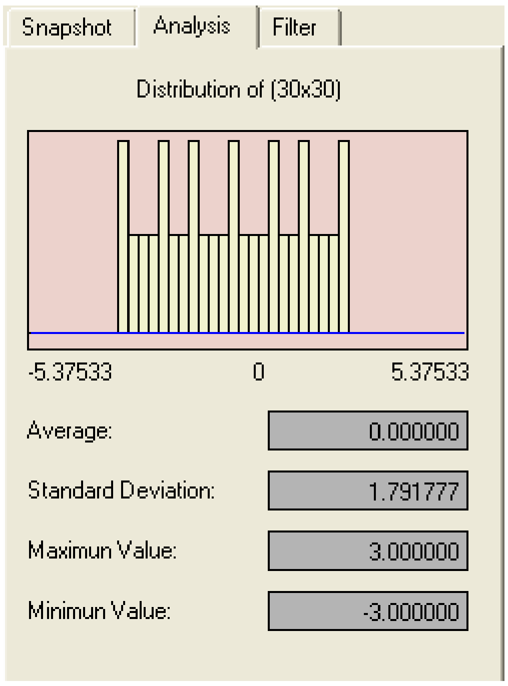
The string (30x30x1) above the snapshot window specifies the shape of the array being examined. The string (30x30) right below the snapshot window shows the size of the dimensions of displaying array data.

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 shown in Figure 4.8.6.



Average:	0.000000
Standard Deviation:	1.791777
Maximun Value:	3.000000
Minimun Value:	-3.000000

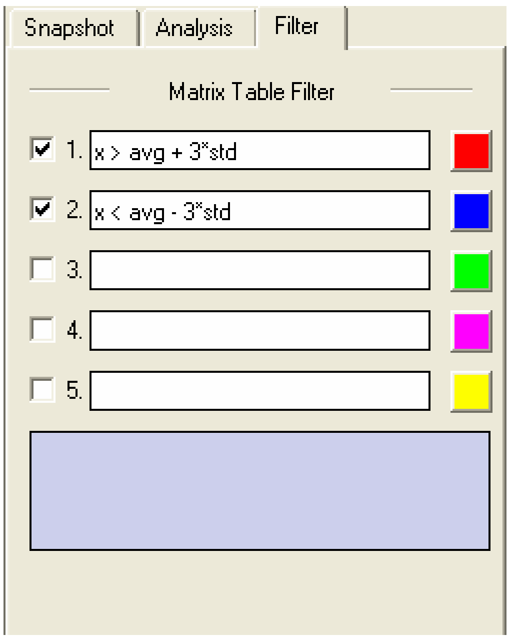
Figure 4.8.6 Analysis Panel

4.8.7 Filter Panel

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

Use '*X* or *x* to represent the data that is 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 <, >, `<=,` and >=. The equal sign is not supported.



☑ 1. x > avg + 3*std 🟥
☑ 2. x < avg - 3*std 🟦
☐ 3. 🟩
☐ 4. 🟪
☐ 5. 🟨

Figure 4.8.7 Filter panel

4.8.8 Status Bar

The Status Bar located at the bottom of the Data Viewer contains two parts. The one on the right-hand side shows the system status and the one on the left-hand side shows the progress status. The system status bar displays the value of a selected cell. It also shows messages of any abnormal system behaviors.

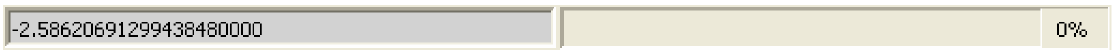


Figure 4.8.8 Status bar

Visualization Methods 5

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

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

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

5.1 Linear Graph

The data used for plotting linear graph in two-dimensional and three-dimensional space are the coordinates in vector form.

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

5.1.1 Two-dimensional Linear Graph

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

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

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

Example 5.1.1 Two-dimensional linear plot

Create a new figure window and plot the cosine graph. The line color is set as red when you specify the optional argument as ‘r’.

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

mfFigure(1);
mfPlot(x, y, "r");
mfTitle("Cosine graph" );
```

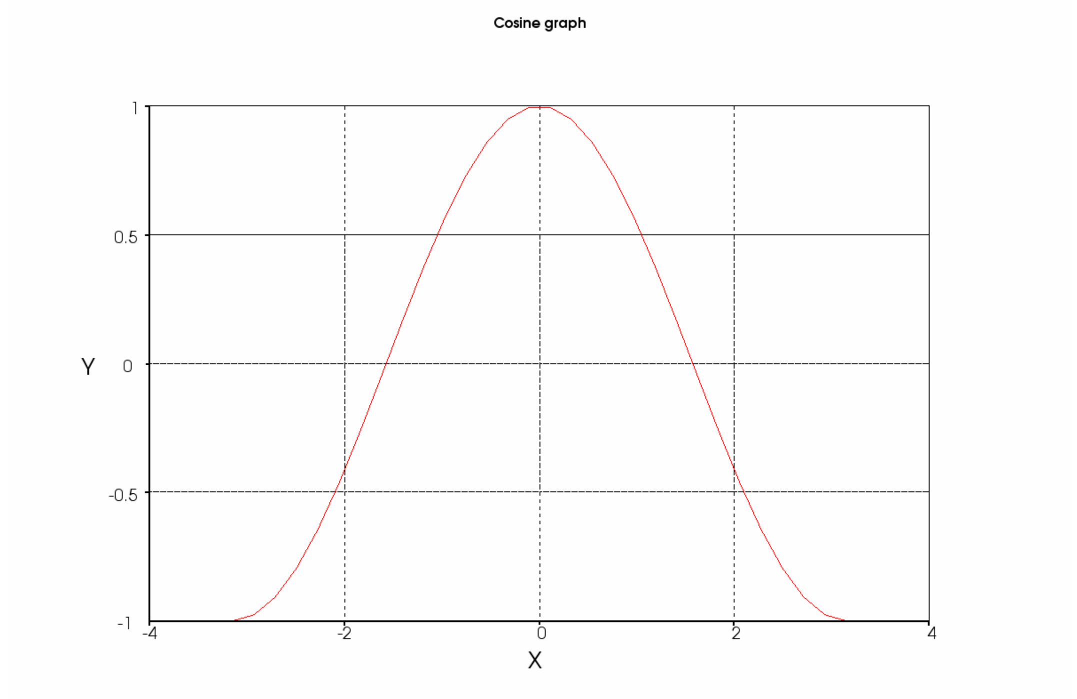


Figure 5.1.1.1 Cosine graph with red line

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

```
mfFigure(2);
mfPlot(x, y, "go-");
mfTitle("Cosine graph");
```

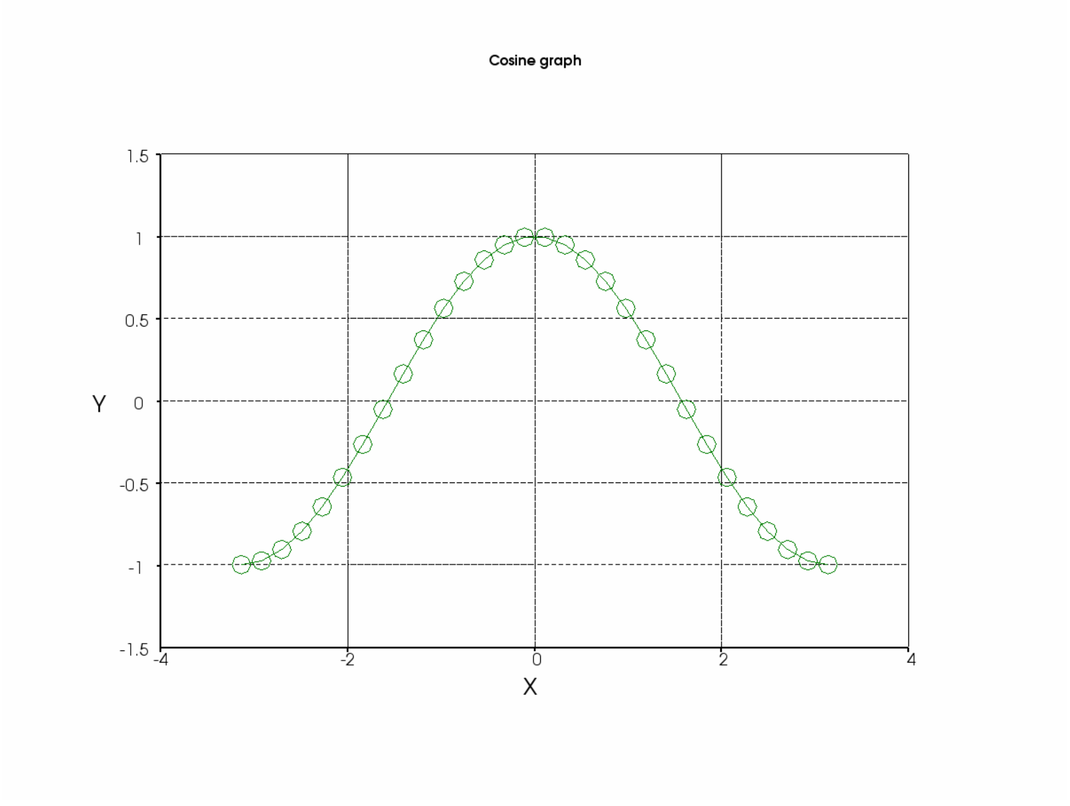


Figure 5.1.1.2 Cosine graph with green line and circle markers

5.1.2 Three-dimensional Linear Graph

MATFOR provides the functions `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 these three types 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.

```
// *******************************************************************
//  PLOT3
// *******************************************************************
mfFigure("plot3");
h = mfPlot3(x, y, z);
mfAxis(-30,30,-30,30,0,30);
mfCamZoom(1.4);

// *******************************************************************
//  TUBE
// *******************************************************************
mfFigure("tube");
h = mfTube(x, y, z) ;
mfAxis(-30,30,-30,30,0,30);
mfCamZoom(1.4);

// *******************************************************************
//  RIBBON
// *******************************************************************
mfFigure("ribbon");
h = mfRibbon(x, y, z) ;
mfAxis(-30,30,-30,30,0,30);
mfCamZoom(1.4);
```

The results show in the following figures.

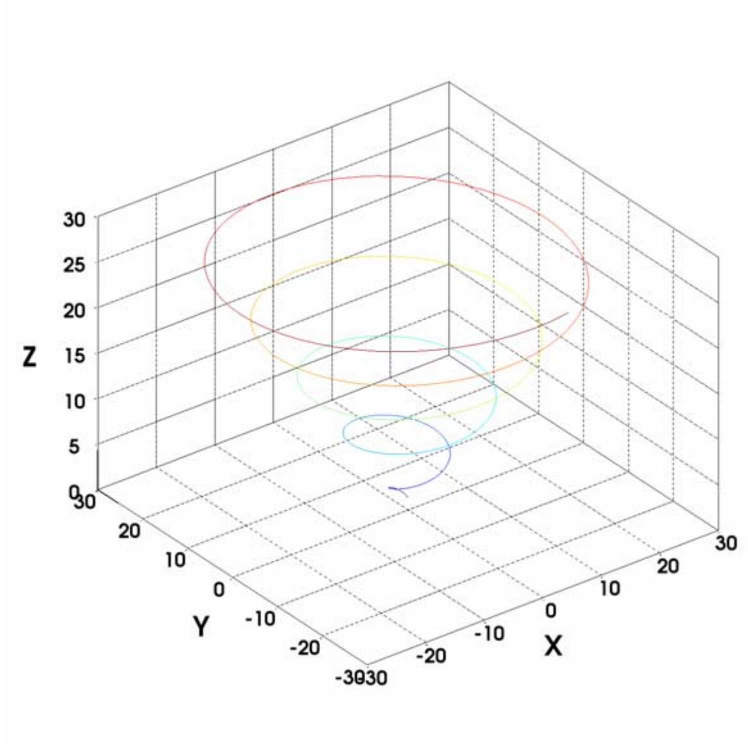


Figure 5.1.2.1 Line graph

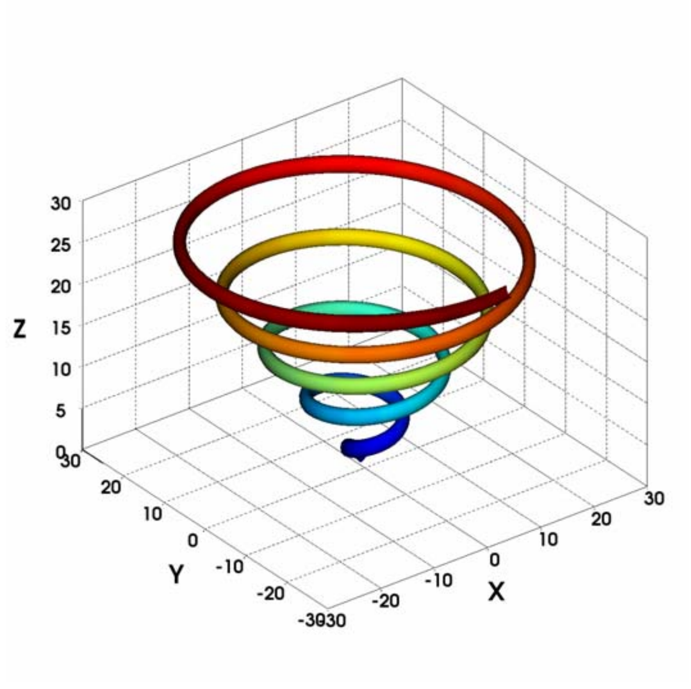


Figure 5.1.2.2 Tube graph

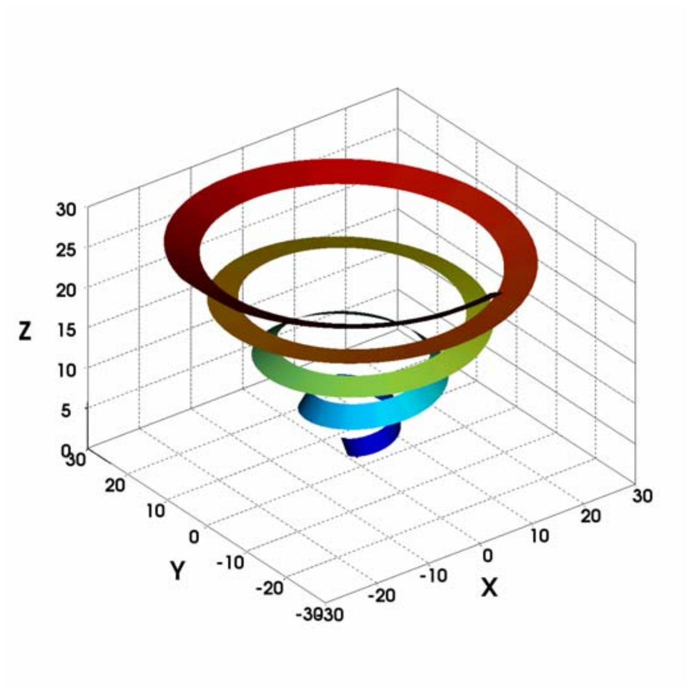


Figure 5.1.2.3 Ribbon graph

5.2 Surface Plot

In MATFOR, quadrilateral grid data can be plotted using various representations, such as surface graph, mesh graph, and with 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 functions `mfSurf`, `mfMesh`, `mfSurfc,` and `mfMeshc`.

Example 5.2.1 Surface plots using different methods

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

```
mfFigure('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 functions `mfSurfc` and `mfMeshc`, a two-dimensional contour plot is added to the surface plot.

```
// ******************************************************************
//  surf
// ******************************************************************
mfSubplot(2, 2, 1);
mfTitle("surf");
h = mfSurf(x, y, z);
mfCamZoom(1.5);

// ******************************************************************
//  mesh
// ******************************************************************
mfSubplot(2, 2, 2);
mfTitle("mesh");
h = mfMesh(x, y, z);
mfCamZoom(1.5);

// ******************************************************************
//  surfc
// ******************************************************************
mfSubplot(2, 2, 3);
mfTitle("surfc");
h = mfSurfc(x, y, z);
mfCamZoom(1.5);

// ******************************************************************
//  meshc
// ******************************************************************
mfSubplot(2, 2, 4);
mfTitle("meshc");
h = mfMeshc(x, y, z);
mfCamZoom(1.5);
```

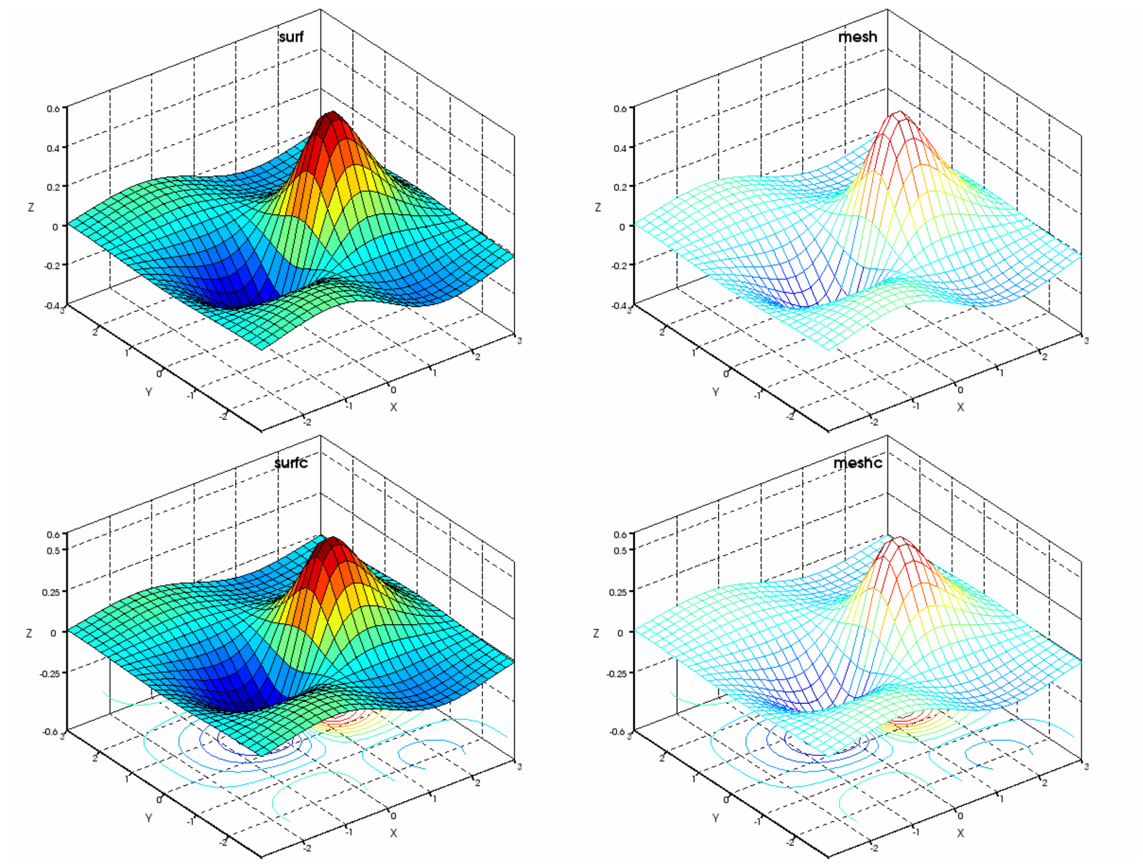


Figure 5.2.1 Surface plots

5.2.2 Contour Plot

Contour plots are lines or surfaces of constant scalar values. They can be drawn in two-dimensional or three-dimensional space using functions `mfContour`, `mfContour3`, `mfSolidContour`, and `mfSolidContour3`. The function `mfOutline` 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'`.

```
mfFigure('contour')
```

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

```
// *****************************************************************
//  contour3
// *****************************************************************
mfSubplot(2, 2, 1);
mfTitle("contour3 with outline");
h = mfContour3(x, y, z);
mfHold("on");
h = mfOutline(x, y, z);
mfDrawMaterial(h, "edge", "color", mfV(0, 0, 1));
mfCamZoom(1.5);


// *****************************************************************
//  contour
// *****************************************************************
mfSubplot(2, 2, 2);
mfTitle("contour");
h = mfContour(x, y, z);
mfAxis("equal");
mfCamZoom(1.5);

// *****************************************************************
//  solid contour3
// *****************************************************************
mfSubplot(2, 2, 3);
mfTitle("solidcontour3 with outline");
h = mfSolidContour3(x, y, z);
mfHold("on");
h = mfOutline(x, y, z);
mfDrawMaterial(h, "edge", "color", mfV(0, 0, 1));
mfCamZoom(1.5);

// *****************************************************************
//  solid contour
// *****************************************************************
mfSubplot(2, 2, 4);
```

```
mfTitle("solidcontour");
h = mfSolidContour(x, y, z);
mfAxis("equal");
mfCamZoom(1.5);
```

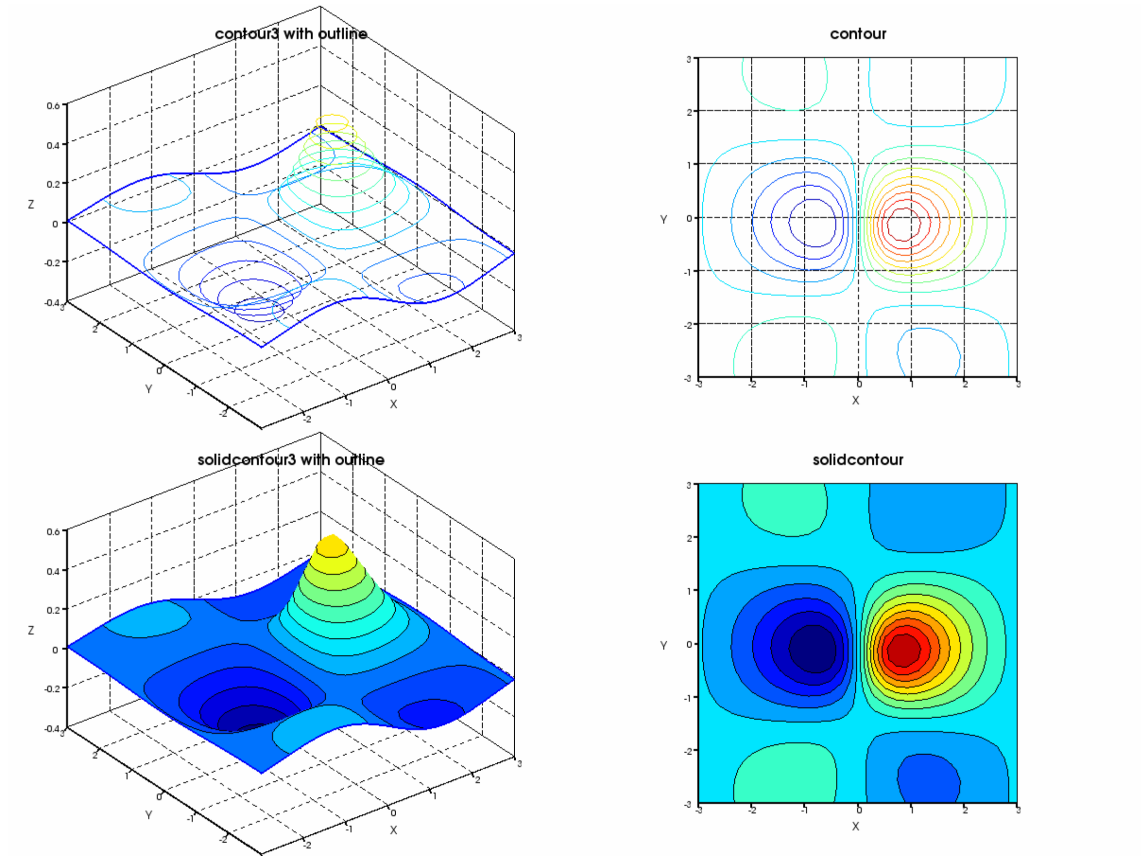


Figure 5.2.2 Contour plots

5.2.3 Pseudocolor Plot

Using the pseudocolor plotting function `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 function `mfSurf`.

Example 5.2.3 Pseudocolor plot

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

```
mfFigure('pcolor')
```

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

```
// ******************************************************************
//  pcolor
// ******************************************************************
mfSubplot(2, 2, 1);
mfTitle("pcolor");
h = mfPColor(x, y, z);
mfAxis("equal");
mfCamZoom(1.5);

// ******************************************************************
//  pcolor w/o edge
// ******************************************************************
mfSubplot(2, 2, 2);
mfTitle("pcolor w edge");
h = mfPColor(x, y, z);
mfDrawMaterial(h, "edge", "visible", "on");
mfAxis("equal");
mfCamZoom(1.5);

// ******************************************************************
//  pcolor with interp
// ******************************************************************
mfSubplot(2, 2, 3);
mfTitle("solid with interp and contour");
h = mfPColor(x, y, z);
mfDrawMaterial(h, "surf", "smooth", "on");
mfDrawMaterial(h, "edge", "visible", "off");
mfAxis("equal");
mfCamZoom(1.5);

// ******************************************************************
//  pcolor with contour
// ******************************************************************
mfSubplot(2, 2, 4);
mfTitle("solid with interp and contour");
h = mfPColor(x, y, z);
mfDrawMaterial(h, "surf", "smooth", "on");
mfDrawMaterial(h, "edge", "visible", "off");
mfHold("on");
h = mfContour(x, y, z);
mfDrawMaterial(h, "edge", "colormap", "off");
mfAxis("equal");
mfCamZoom(1.5);
```

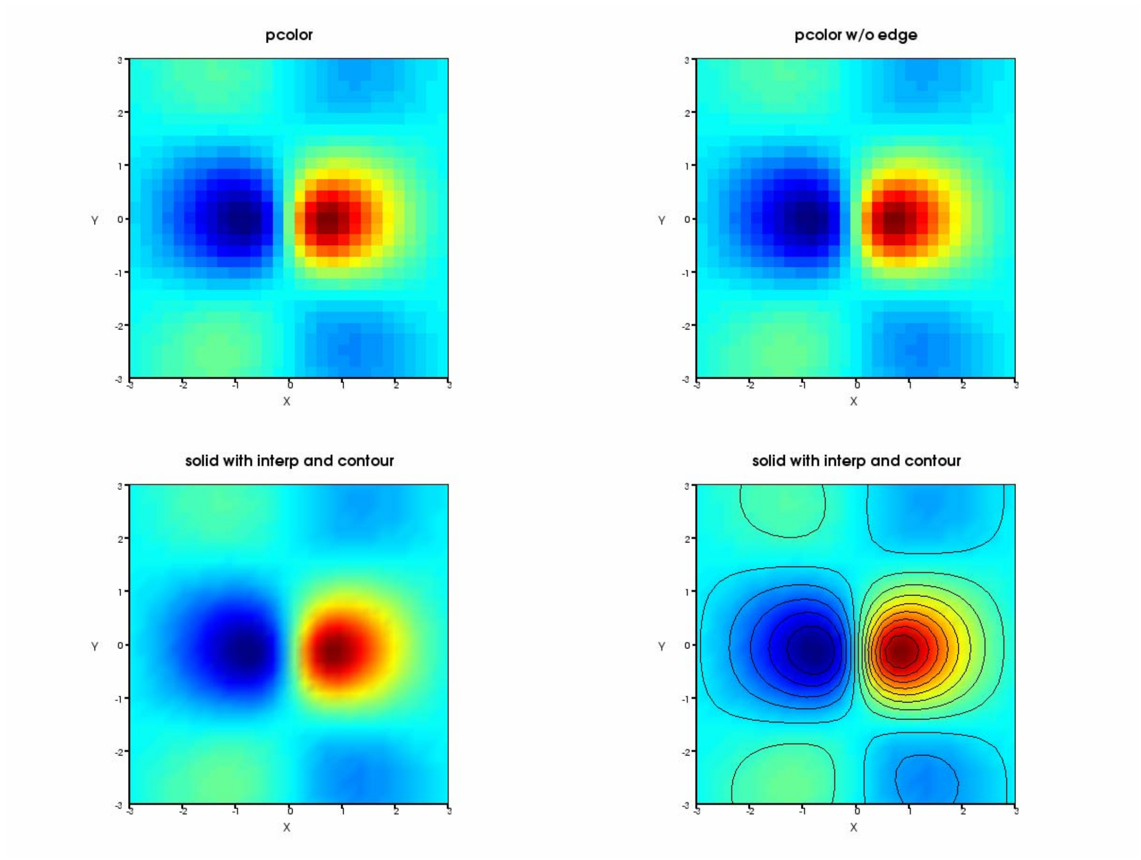


Figure 5.2.3 Pseudocolor plots

5.3 Volume Rendering

This section covers the functions that visualize volumetric data in the representation of surface, mesh, sliced-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 shall demonstrate an example that displays a portion of the nose on a missile using various representations to explore different aspects of the application. The example loads data from hdd.mfb, which 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)

Displaying the data as surface plot or contour plot would give you different perspectives of the application.

Example 5.3.1 Surface plots of the volumetric data

First, we simply use function `mfSurf` with a transparent shading to visualize the volumetric data as mesh grid.

```
mfFigure("Mesh");
mfTitle("Mesh");
h = mfSurf(x, y, z);
mfDrawMaterial(h, "surf", "smooth", "on", "trans", 50);
mfDrawMaterial(h, "edge", "visible", "on", "colormap", "on");
mfView(30, 45);
mfAxis("off");
mfCamZoom(1.8);
mfCamPan(0, -70);
```

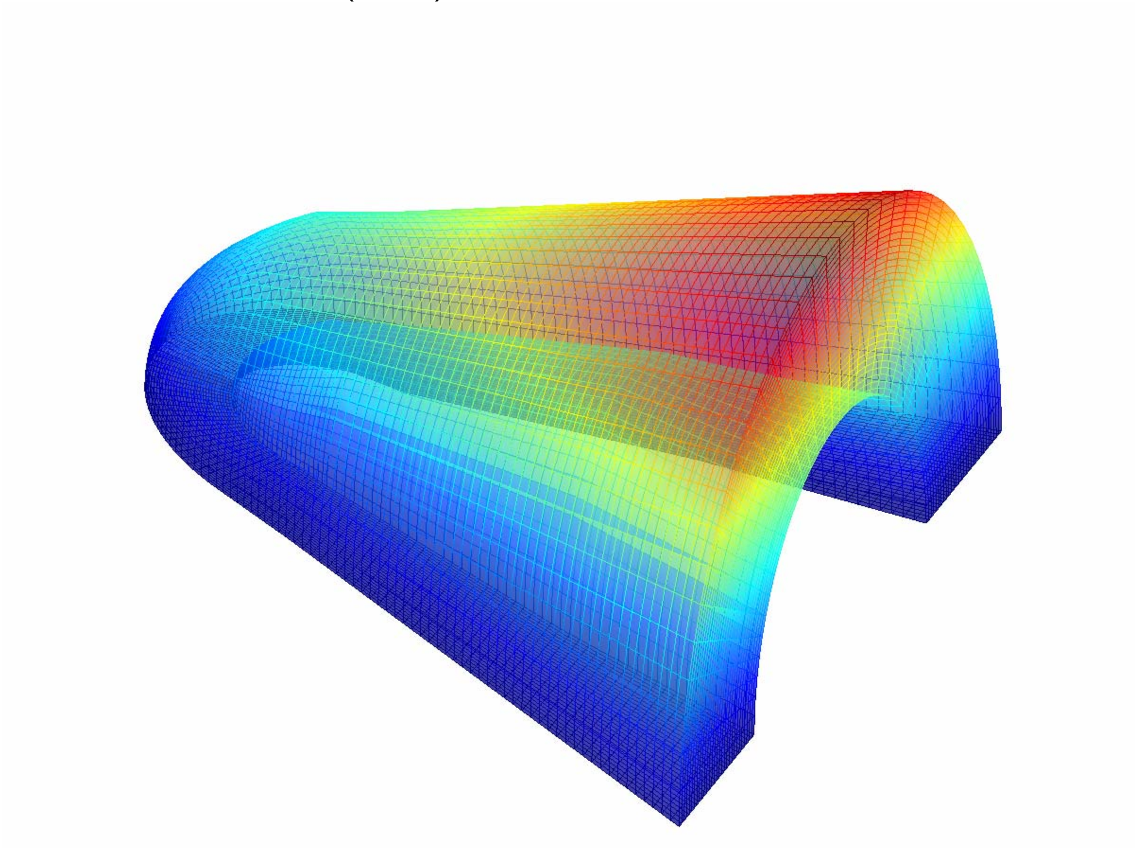


Figure 5.3.1.1 Mesh Plot of the volumetric data

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 function `mfSolidContour3` with field data mach1, and data set (x2, y2 and z2) using function `mfSurf` with field data mach2.

```
mfFigure("mach");
mfTitle("mach");

h = mfSolidContour3(x1, y1, z1, mach1);
mfHold("on");

h = mfSurf(x2, y2, z2, mach2);
mfDrawMaterial(h, "surf", "smooth", "on", "trans", 50);
mfDrawMaterial(h, "edge", "visible", "on", "colormap", "on");

mfColorbar("on");
mfView(30, 45);
mfAxis("off");
mfCamZoom(1.8);
mfCamPan(0, -70);
```

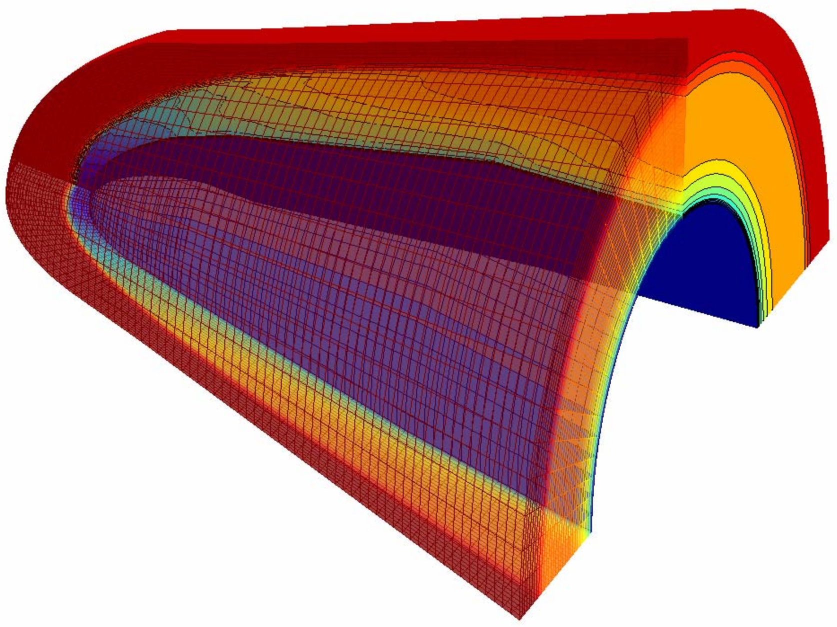


Figure 5.3.1.2 Solid contour and mesh plot of the volumetric data

5.3.2 Sliced-planes

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

Using function `mfSliceXYZ`, you can select any orthogonal sliced-plane along x, y, and z directions to be displayed. Function `mfSlicePlane` allows you to cut a sliced-plane along the arbitrary direction.

Function `mfSliceIJK` displays sliced-planes along i, j, and k which are the index of the matrices that specify the coordinates. It has the following syntax:

```
mfSliceIJK(x, y, z, i, j, k)
```

In example 5.3.2, we shall display several sliced-planes of the missile node object by using function `mfSliceIJK`.

Example 5.3.2 Sliced-planes of the volumetric data

Plot a sliced-plane along index of mfArray `x`, another sliced-plane along index of mfArray `y`, and five sliced-planes along index of mfArray `z`.

```
mfFigure("slice mach");
mfTitle("slice mach");
h = mfSliceIJK(x, y, z, mach, m, 1, mfLinspace(1, k, 5) );
mfDrawMaterial(h, "surf", "smooth", "on");
mfDrawMaterial(h, "edge", "visible", "off", "colormap", "on");

mfHold("on");
mfColorbar("on");
mfView(30, 45);
mfAxis("off");
mfCamZoom(1.8);
mfCamPan(0, -70);
```

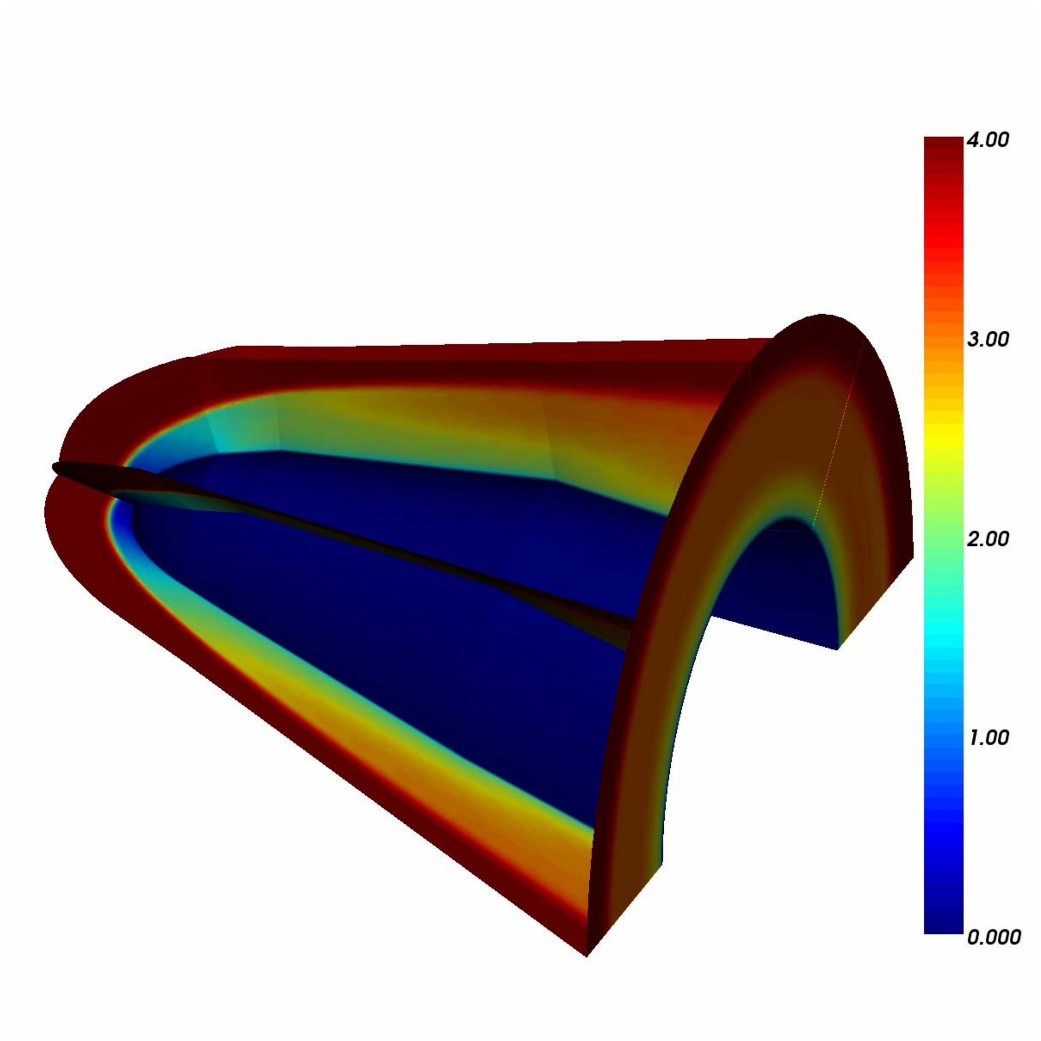


Figure 5.3.2 Show sliced-planes of the volumetric data

5.3.3 Isosurface

Function `mfIsoSurface` creates 3-D graphs composed of isosurface data from the volumetric data.

Example 5.3.3 combines a few representations of the missile nose using functions `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 Isosurface plots of the volumetric data

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

```
mfFigure("isosurface mach");
mfTitle("isosurface mach");
h = mfIsoSurface(x1, y1, z1, mach1, mfLinspace(1, 4, 6) );
mfHold("on");
h = mfIsoSurface(x3, y3, z3, mach3, mfLinspace(1, 4, 6) );
h = mfOutline(x, y, z);
mfColorbar("on");
mfView(30, 45);
mfAxis("off");
mfCamZoom(1.8);
mfCamPan(0, -70);
```

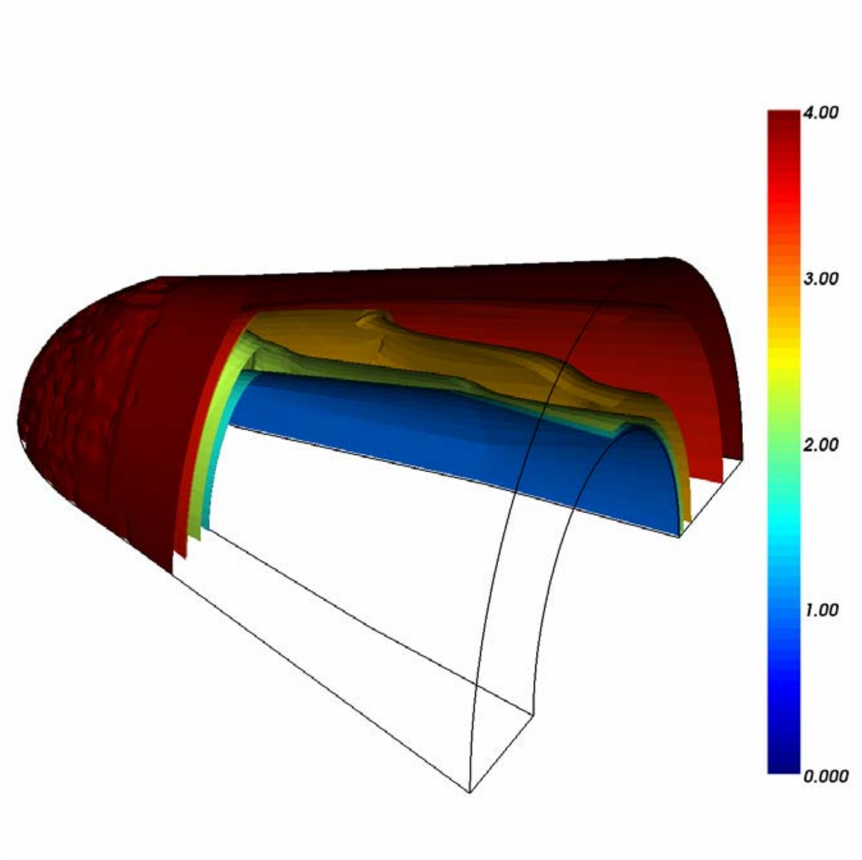


Figure 5.3.3 Isosurface plot of the volumetric data

5.4 Vector Field

Vector set in two-dimensional and three-dimensional space can be represented in quivers or streamlines using functions `mfQuiver` and `mfStreamlLine`. The streamlines are plotted by identifying the vector set, as well as their corresponding starting points.

5.4.1 Quiver and Streamline

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

Example 5.4.1 Quiver and streamline representations of the volumetric data

Start by generating a set of mesh grid data.

```
a = mfLinspace(-2, 1.6, 4);
b = mfLinspace(-2, 1, 3);
c = mfLinspace(-2, 1.84, 5);
mfMeshgrid(mfOut(x, y, z), a, b, c);
u = mfOnes(3, 4, 5);
v = 0.4*mfPow(z,2);
w = mfExp(0.5*x);
```

Plot the quivers and streamlines using the data set created. You may notice that the streamlines are plotted by identifying the starting points of the data.

```
mfQuiver3(x, y, z, u, v, w);
mfHold("on");
mfStreamLine(x,y,z,u,v,w, mfV(-1.2,0.0,0.2), mfV(-1.2,0.5,0.0),
        mfV(-2.0,-1.0,-2.0));
mfView(-30,50);
```

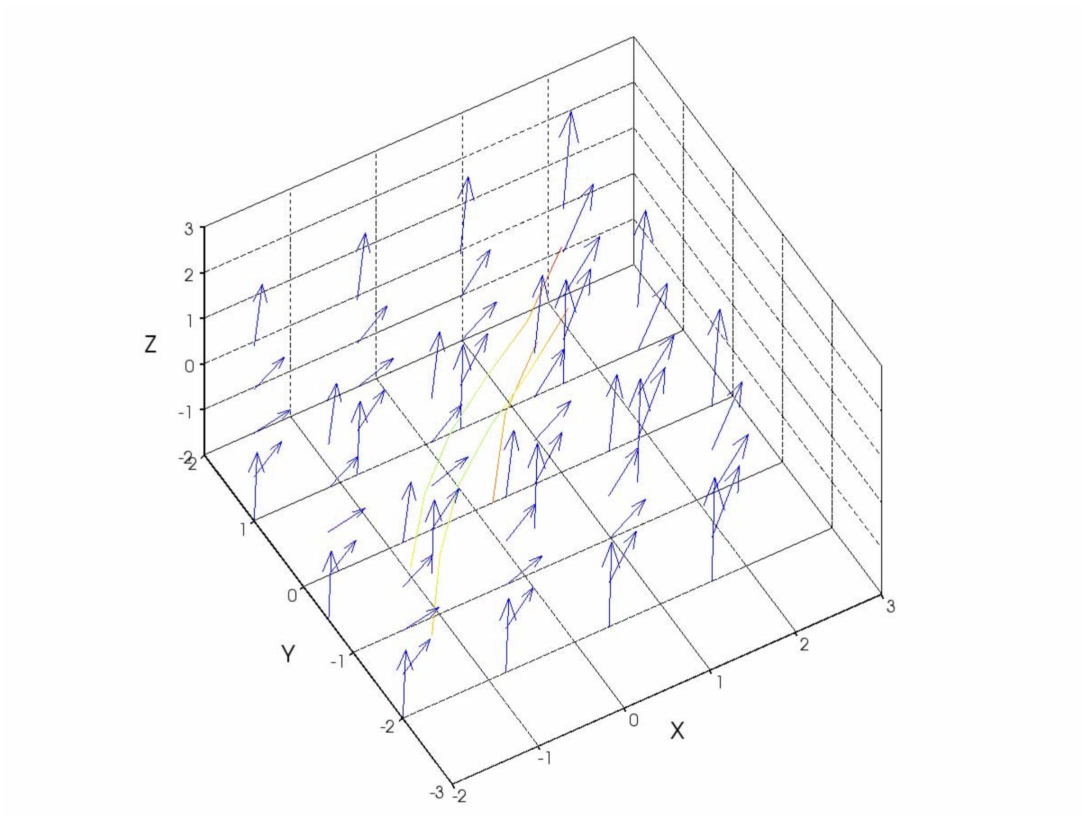


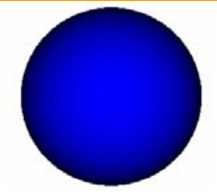
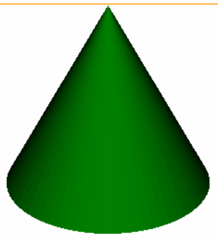
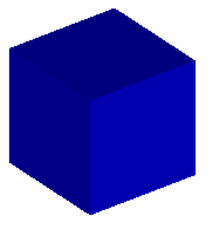
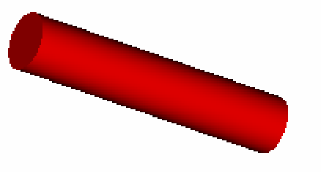
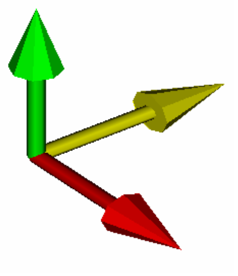
Figure 5.4.1 Plot quivers

5.5 Elementary 3-D Objects

MATFOR provides you with a set of elementary 3-D objects to display on the plot space. In general, the elements are plotted with given coordinates, sizes, and colors. When plotting molecules, the connectivity between the ball objects should also be specified.

5.5.1 Primitives

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

fSphere	
mfCone	
mfCube	
mfCylinder	
mfAxisMark	

5.5.2 Molecule

The construction of a molecule object is a little bit different from the constructions of 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 each of the different atoms.

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 = pos_data( MF_COL, mfI(2,4));

atom = pos_data( MF_COL, 1);
```

Define the radius for each atom type.

```
rad = atom;

mfGDisplay(atom > 14, "atom > 14");

rad(atom > 14 ) = 14;
rad = 0.1*rad;

color = mfOnes(s, 3) * 0.7;
for(i = 1; i <= s; i++)
{
        int weight = (int) atom(i).ToDouble();
  // C
        if (weight==12)
                color(i, MF_COL) = mfV(0.8, 0.8, 0.8);
        // N
        else if (weight==14)
                color(i, MF_COL) = mfV(0.2, 0.2, 1.0);
        // O
        else if (weight==16)
                color(i, MF_COL) = mfV(0.9, 0.0, 0.0);
  // S
        else if (weight==32)
                color(i, MF_COL) = mfV(1.0, 1.0, 0.0);
}
```

And finally, define the radius and color of the sticks to be connected between the atoms and then plot the structured protein.

```
stick_rad = 0.1 * mfOnes(s, 1);
stick_col = mfOnes(s, 3);

mfMolecule(loc, conn, rad, color, stick_rad, stick_col, mf(6));
```

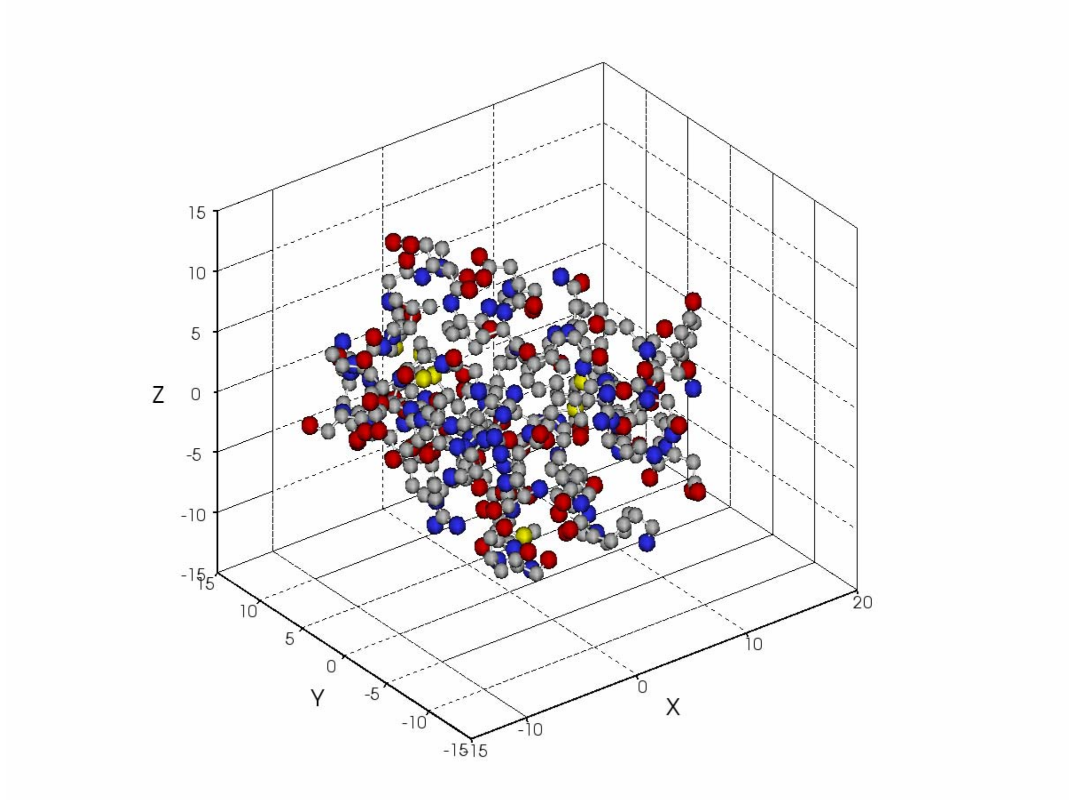


Figure 5.5.2 Structured protein graph

5.6 Unstructured Mesh

An arbitrary mesh, or surface, is usually represented as 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 connectivity between the vertex. In MATFOR, unstructured mesh can be visualized by using functions `mfTriSurf`, `mfTriMesh`, and `mfTriContour`.

The triangular representation is the most often-used face connectivity. 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. The resulting graphics objects are displayed using functions `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;

    double xyz_val[] = {
            -x, x, -x, x, 0.0, 0.0, 0.0, 0.0, z, -z, z, -z,
            0.0, 0.0, 0.0, 0.0, z, z, -z, -z, x, x, -x, -x,
            z, z, -z, -z, x, -x, x, -x, 0.0, 0.0, 0.0, 0.0 };
    double tri1_val[] = {
            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 };

    xyz = mfArray(xyz_val, 12, 3);
    tri1 = mfArray(tri1_val, 20, 3);
    c = mfPow(mfS(xyz, MF_COL, 1),2) + mfPow(mfS(xyz, MF_COL,
2),2) - mfPow(mfS(xyz, MF_COL, 3),2);

    mfSubplot(1, 2, 1);
    h = mfTriSurf(tri1, xyz, c);
    mfAxis("equal");
    mfSubplot(1, 2, 2);
    h = mfTriMesh(tri1, xyz, c);
    mfDrawMaterial(h, "surf", "visible", "off");
    mfAxis("equal");
    mfViewPause();
```

Perform subdivision three times. Draw the polygonal object during each iteration.

```
for(j = 1; j <= 3; j++)
{
        int L = 1;
        for(i = 0; i < (j-1); i++) {L *= 4;}
        L *= 20;

        for(i = 1; i <= L; i++)
        {
                p1 = tri1(i, 1);
                p2 = tri1(i, 2);
                p3 = tri1(i, 3);
                p1xyz = mfHCat(xyz(p1, 1), xyz(p1, 2), xyz(p1, 3));
                p2xyz = mfHCat(xyz(p2, 1), xyz(p2, 2), xyz(p2, 3));
                p3xyz = mfHCat(xyz(p3, 1), xyz(p3, 2), 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)
                {
                        tri2 = mfVCat(
                                mfHCat(p1, p12, p13),
                                mfHCat(p2, p23, p12),
                                mfHCat(p3, p23, p13),
                                mfHCat(p12,p23, p13));
                }
                else
                {
                        tri2 = mfVCat(tri2,
                                mfHCat(p1, p12, p13),
                                mfHCat(p2, p23, p12),
                                mfHCat(p3, p23, p13),
                                mfHCat(p12,p23, p13));
                }

                xyz = mfVCat(xyz, p12xyz, p23xyz, p13xyz);

        }
        tri1 = tri2;
        //tri2 = mf();
```

```
            c = mfPow(xyz(MF_COL, 1),2) + xyz(MF_COL, 2) -
mfPow(xyz(MF_COL, 3),2);
            mfSubplot(1, 2, 1);
            h = mfTriSurf(tri1, xyz, c);
            mfAxis("equal");
            mfSubplot(1, 2, 2);
            h = mfTriMesh(tri1, xyz, c);
            mfDrawMaterial(h, "surf", "visible", "off");
            mfAxis("equal");
            mfViewPause();
    }
```

The function `normalize` calculates the normalized cross product of two vectors.

```
mfArray normalize(mfArray v)
{
    mfArray d, out;

    out = mfOnes(1, 3);
    d = mfSqrt(v(1, 1)*v(1, 1) + v(1, 2)*v(1, 2) + v(1, 3)*v(1, 3));
   out(1, 1) = v(1, 1) / d;
   out(1, 2) = v(1, 2) / d;
    out(1, 3) = v(1, 3) / d;

   return out;
}
```

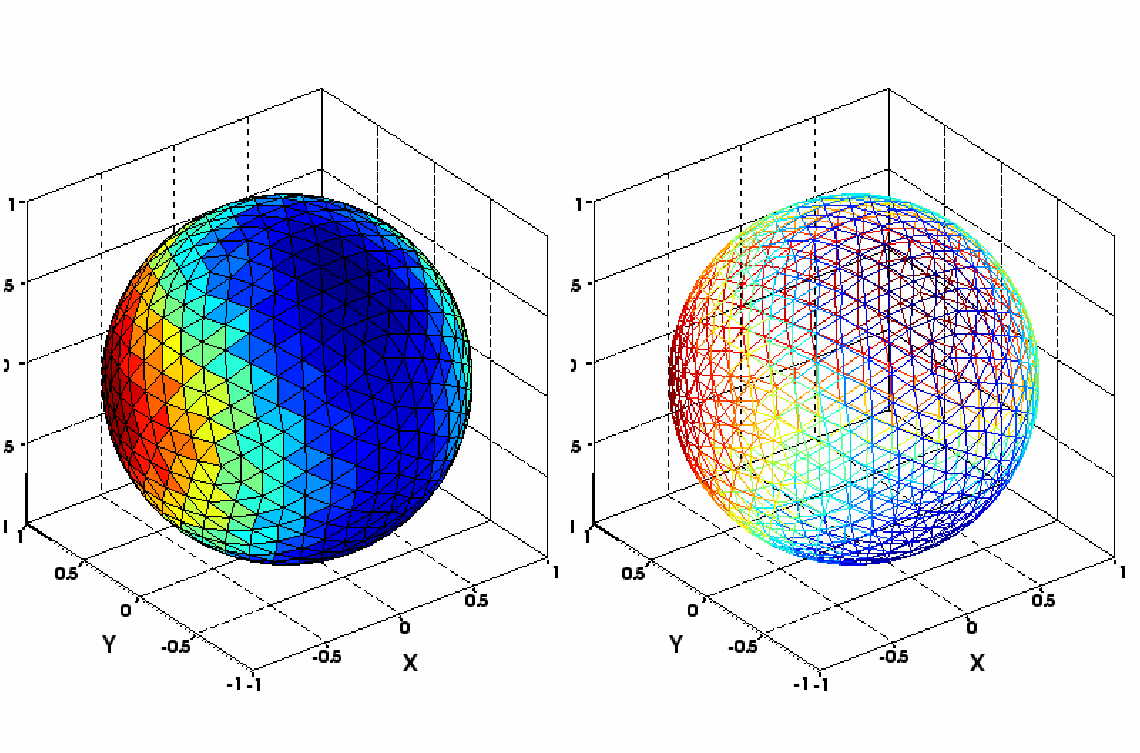


Figure 5.6.1.1 Surface and mesh plot of the polygonal object

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 Displaying the polygonal object using contour representation

Contour representation of the polygonal object:

```
mfSubplot(1, 2, 1);
h = mfTriSurf(tri1, xyz, c);
mfDrawMaterial(h, "edge", "visible", "off");
mfDrawMaterial(h, "surf", "smooth", "on", "ambient", 50, "diffuse",
20);
mfAxis("equal");
mfSubplot(1, 2, 2);
h = mfTriContour(tri1, xyz, c);
mfAxis("equal");
mfViewPause();
```

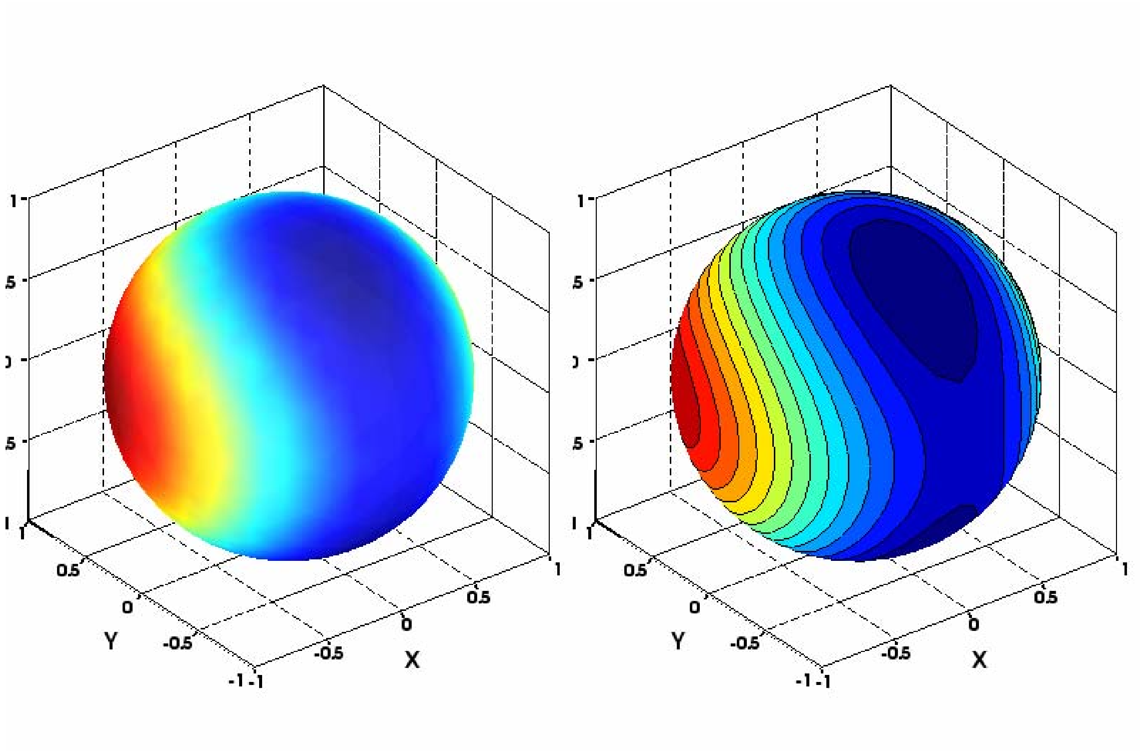


Figure 5.6.2 Contour plot of the polygonal object

5.7 Unstructured Grids

An arbitrary solid model can be represented by unstructured grids that include a set of vertices and a cell set. Vertices are the coordinates in space and cell set specifies the cell connectivity between the vertex. In MATFOR, unstructured grids can be visualized by using functions `mfTetSurf`, `mfTetMesh`, `mfTetContour`, and `mfTetIsosurface`.

MATFOR supports four kinds of the cell connectivity representations, as shown in Figure 5.7.1.

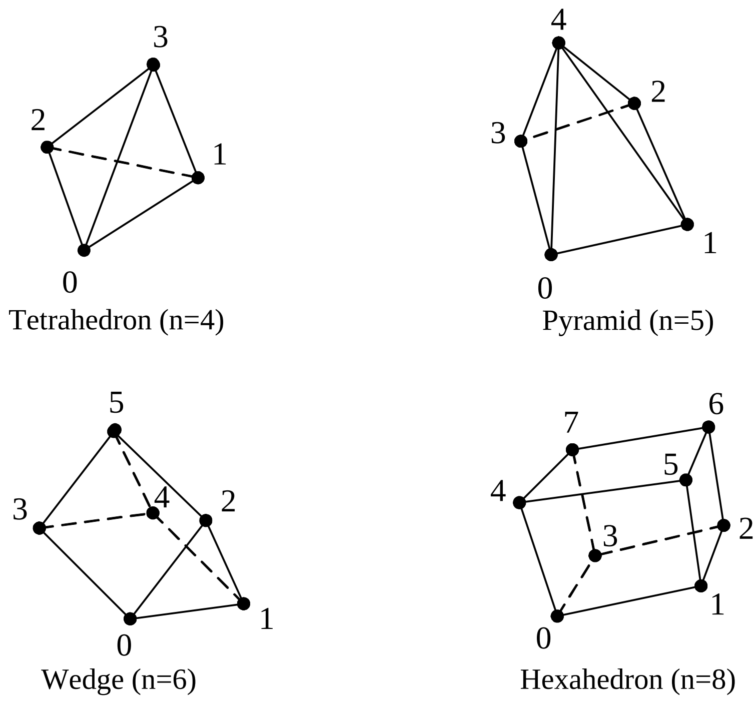


Figure 5.7.1 cell connectivity representations

In this section, we shall demonstrate an example of applying a force to the back of the L-shape steel board, which has a fix constrain around the hole on the bottom of the steel board. The applying force is represented with a cone and the fix constraint is represented by cylinder.

5.7.1 Surface, Contour and Iso-surface plots of unstructured 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 = node(":", "1:3");
dxyz = node(":", "4:6") * 10;
x = xyz(":", 1);
y = xyz(":", 2);
```

```
z = xyz(":", 3);
dx = dxyz(":", 1);
dy = dxyz(":", 2);
dz = dxyz(":", 3);
d_norm = mfSqrt( dx*dx + dy*dy + dz*dz );

sx = sxyz(":", 1) * 1.0e-3;
sy = sxyz(":", 2) * 1.0e-3;
sz = sxyz(":", 3) * 1.0e-3;
sxy = sxyz(":", 4) * 1.0e-3;
syz = sxyz(":", 5) * 1.0e-3;
sxz = sxyz(":", 6) * 1.0e-3;

m1 = mfSize(node, 1);
m2 = mfSize(sxyz, 1);
if (m1>m2)
{
        sx = mfVCat(sx, mfZeros(m1-m2, 1));
        sy = mfVCat(sy, mfZeros(m1-m2, 1));
        sz = mfVCat(sz, mfZeros(m1-m2, 1));
        sxy = mfVCat(sxy, mfZeros(m1-m2, 1));
        syz = mfVCat(syz, mfZeros(m1-m2, 1));
        sxz = mfVCat(sxz, mfZeros(m1-m2, 1));
}

s_norm = mfSqrt( sx*sx + sy*sy + sz*sz );
```

Displays polyhedrons defined by a cell matrix.

```
// *******************************************************************
//  Grid & Mesh
// *******************************************************************
mfFigure("Grid");

// *******************************************************************
//  Grid
// *******************************************************************
mfSubplot(1, 2, 1);
mfTitle("Grid");

// draw structure element
h = mfTetSurf( elem, x, y, z);
mfDrawMaterial(h, "surf", "colormap", "off");
mfHold("on");

// draw cylinder
h = mfCylinder(mfV(-5.8, 0.0, 0.25), 0.95, 1.5, mfV(0, 0, 1));
```

```
// draw cone
h = mfCone(mfV(1.5, 0.0, 5.0), 0.5, 2, mfV(0, 0, 1));
mfObjOrigin(h, 1.5, 0.0, 5.0);
mfObjOrientation(h, 0, -90, 0);

// draw force annotation
h = mfAnnotation("Force=10000NT", mfV(2.5,0.0,5.0), mfV(1, 0, 0) );
mfGSet(h, "offset", mfV(-50, 10));

// draw constraint annotation
h = mfAnnotation("Fix Constraint", mfV(-5.8,0.0,1), mfV(1, 0, 0) );
mfGSet(h, "offset", mfV(-50, 10));

mfAxis("equal");
mfAxis("off");
mfCamZoom(1.3);

// ******************************************************************
//  Mesh
// ******************************************************************
mfSubplot(1, 2, 2);
mfTitle("Mesh");
// draw structure element
h = mfTetSurf( elem, x, y, z);
mfDrawMaterial(h, "surf", "colormap", "off", "trans", 80 );
mfHold("on");

// draw cylinder
h = mfCylinder(mfV(-5.8, 0.0, 0.25), 0.95, 1.5, mfV(0, 0, 1));

// draw cone
h = mfCone(mfV(1.5, 0.0, 5.0), 0.5, 2, mfV(0, 0, 1));
mfObjOrigin(h, 1.5, 0.0, 5.0);
mfObjOrientation(h, 0, -90, 0);

// draw force annotation
h = mfAnnotation("Force=10000NT", mfV(2.5,0,5), mfV(1, 0, 0) );
mfGSet(h, "offset", mfV(-50, 10));

// draw constraint annotation
h = mfAnnotation("Fix Constraint", mfV(-5.8,0.0,1), mfV(1, 0, 0) );
mfGSet(h, "offset", mfV(-50, 10));

mfAxis("equal");
mfAxis("off");
mfCamZoom(1.3);
```

```
mfViewPause();
```

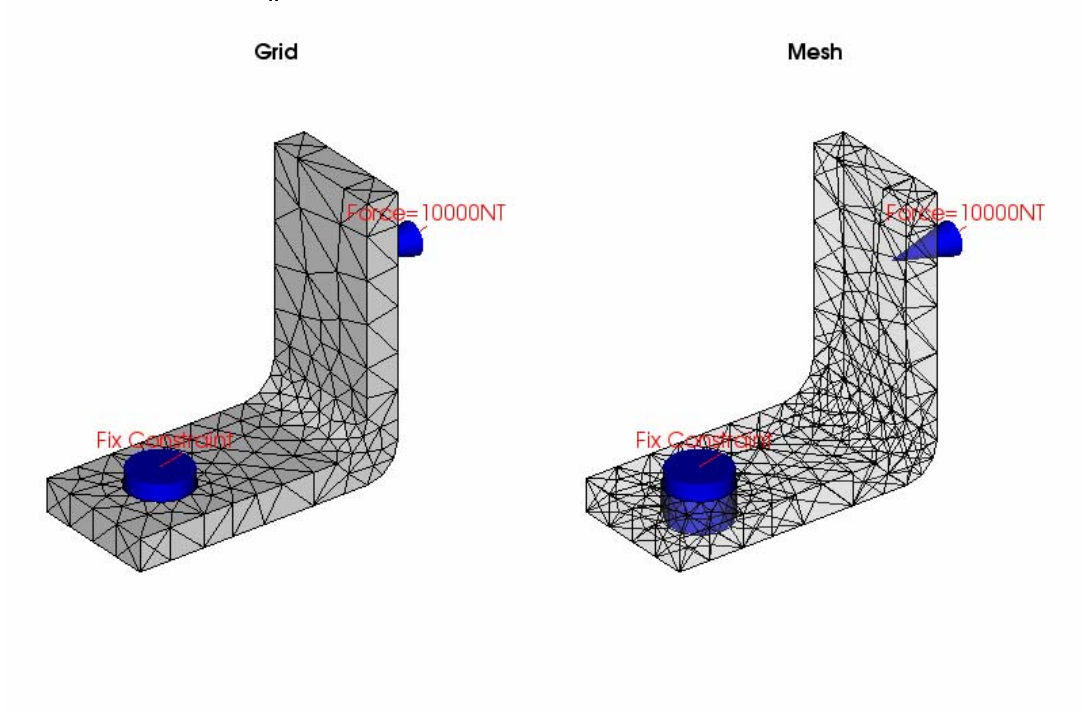


Figure 5.7.1.1 Display the grid and mesh plot of the L-shape steel board

Illustrate the deformation and displacement vectors of the steel board after the is force applied.

```
// *****************************************************************
//  Displacement
// *****************************************************************
mfFigure("Displacement");

// *****************************************************************
//  Deformation
// *****************************************************************
mfSubplot(1, 2, 1);
mfTitle("Deformation");
h = mfTetSurf( elem, x, y, z, d_norm );
mfDrawMaterial(h, "surf", "visible", "off");
mfDrawMaterial(h, "edge", "colormap", "on", "trans", 90 );
mfHold("on");
h1 = mfTetSurf( elem, xyz, d_norm);
mfDrawMaterial(h1, "edge", "visible", "off");
```

```
mfColorbar("vert");
mfAxis("equal");
mfAxis("off");
mfAxis(-8,2,-4,4,-2,9);
mfCamZoom(1.05);

// ***************************************************************
//  Displacement vector
// ***************************************************************
mfSubplot(1, 2, 2);
mfTitle("Displacement vector");

// draw structure element
h = mfTetSurf( elem, x, y, z, d_norm);
mfDrawMaterial(h, "surf", "colormap", "on", "trans", 90 );
mfDrawMaterial(h, "edge", "visible", "off", "trans", 90 );
mfHold("on");

h2 = mfQuiver3(x, y, z, dx*0, dy*0, dz*0);
mfDrawMaterial(h2, "edge", "colormap", "on");

mfColorbar("vert");
mfAxis("equal");
mfAxis("off");
mfViewPause();


for(i = 1; i <= 20; i++)
{
        mfGSet(h1, "xyz", xyz + i / 20.0 * dxyz);
        mfGSet(h2, "udata", dx*i/20, "vdata", dy*i/20, "wdata",
        dz*i/20);
        mfDrawNow();
}
```

```
mfViewPause();
```

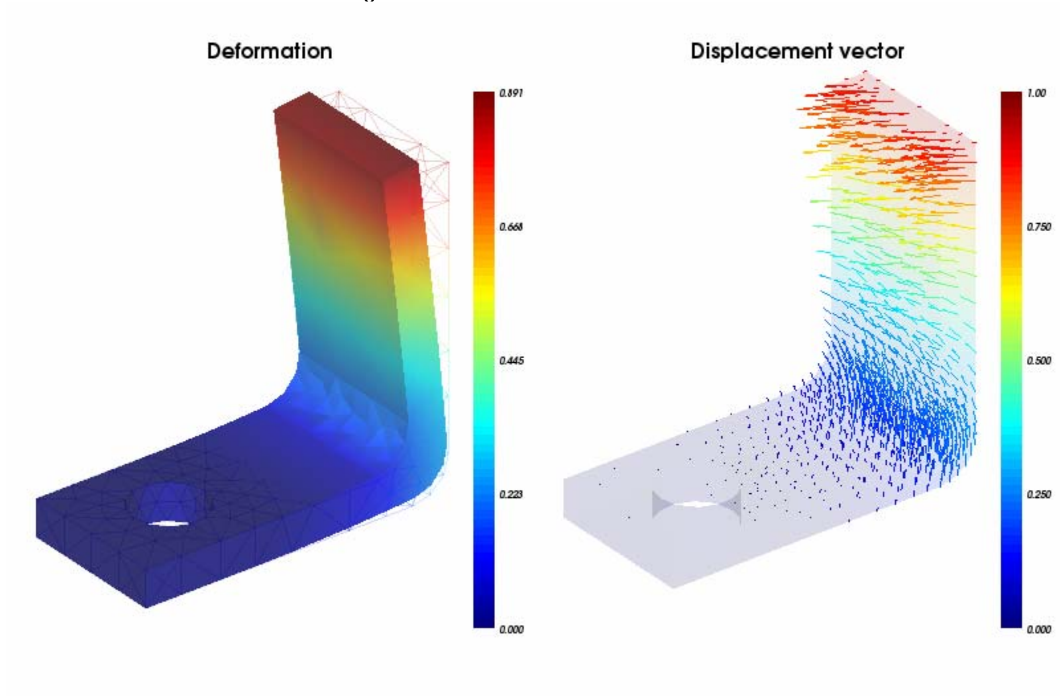


Figure 5.7.1.2 Display the deformation and displacement vector of the L-shape steel board

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.

We represent the shear stress using vectors as shown in the figure on the right.

```
// ******************************************************************
//  Shear stress
// ******************************************************************
mfFigure("Shear stress");

// ******************************************************************
//  Shear value
// ******************************************************************
mfSubplot(1, 2, 1);
mfTitle("Shear stress");
h = mfTetSurf( elem, x+dx, y+dy, z+dz, s_norm);

mfColormapRange(0.0, 2.0);
```

```
mfAxis("equal");
mfAxis("off");
mfCamZoom(1.05);
mfColorbar("vert");

// ****************************************************************
//  Shear vector
// ****************************************************************
mfSubplot(1, 2, 2);
mfTitle("Shear stress vector");

// draw structure element
h = mfTetSurf( elem, x+dx, y+dy, z+dz, d_norm);
mfDrawMaterial(h, "surf", "colormap", "off", "trans", 90 );
mfDrawMaterial(h, "edge", "visible", "off", "trans", 90 );
mfHold("on");

h1 = mfQuiver3(x+dx, y+dy, z+dz, sx*0, sy*0, sz*0, 0.2);
mfDrawMaterial(h1, "edge", "colormap", "on" );

mfColormapRange(0.0, 2.0);
mfColorbar("vert");
mfAxis("equal");
mfAxis("off");

mfViewPause();

for(i = 1; i<=20; i++)
{
        mfGSet(h1, "udata", sx*i/20, "vdata", sy*i/20, "wdata",
        sz*i/20);
        mfDrawNow();
}
```

```
mfViewPause();
```

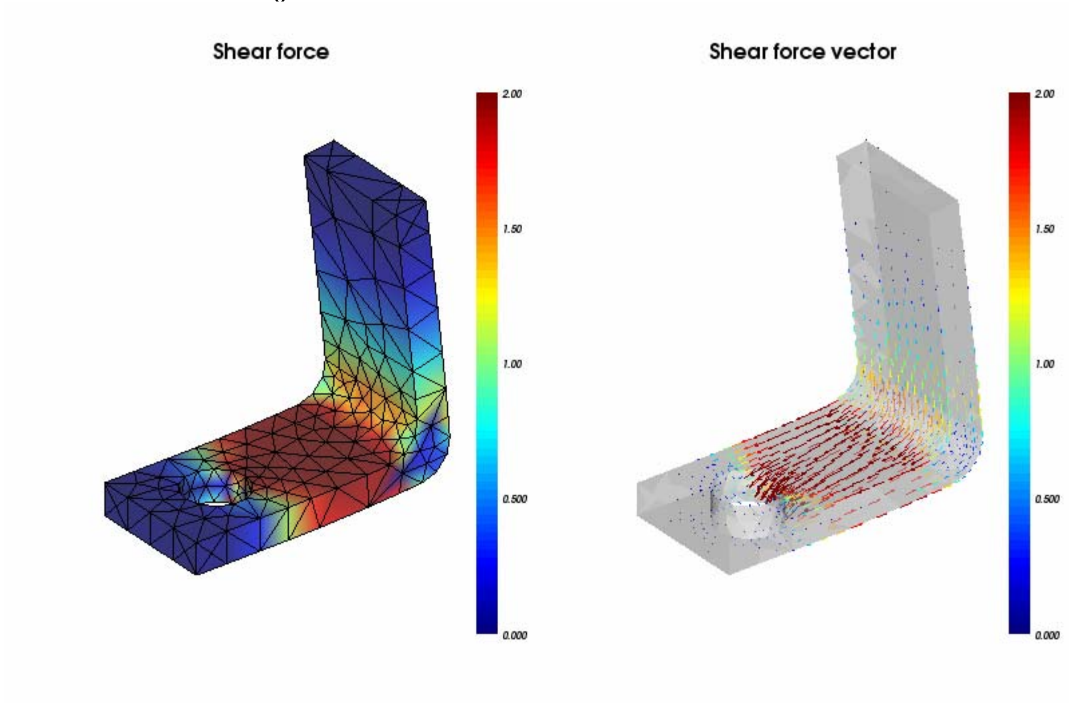


Figure 5.7.1.3 Display the shear force and the shear force vectors of the L-shape steel board

Draw contour on the surface with labels to show the iso-value of each impacted area of the shear stress.

The figure on the right is similar to the one on the left, except it only displays the contour lines around each area with its vector values.

```
// ******************************************************************
//  Shear contour
// ******************************************************************
mfFigure("Shear contour");

// ******************************************************************
//  Shear contour
// ******************************************************************
mfSubplot(1, 2, 1);
mfTitle("Shear stress contour");
```

```
// draw structure element
h = mfTetContour( elem, x+dx, y+dy, z+dz, s_norm);
mfGSet(h, "iso", mfLinspace(0, 2, 11), "label", "on");
mfDrawMaterial(h, "surf", "ambient", 20, "diffuse", 80);

mfColormapRange(0.0, 2.0);
mfColorbar("vert");
mfAxis("equal");
mfAxis("off");

// ****************************************************************
//  Shear contour
// ****************************************************************
mfSubplot(1, 2, 2);
mfTitle("Shear stress contour + vector");

// draw structure element
h = mfTetContour( elem, x+dx, y+dy, z+dz, s_norm);
mfGSet(h, "iso", mfLinspace(0, 2, 11));
mfDrawMaterial(h, "surf", "ambient", 20, "diffuse", 80, "trans", 80 );
mfDrawMaterial(h, "edge", "colormap", "on" );

mfHold("on");
h = mfQuiver3(x+dx, y+dy, z+dz, sx, sy, sz, 0.2);
mfDrawMaterial(h, "edge", "colormap", "on" );

mfColormapRange(0.0, 1.0);
mfAxis("equal");
mfAxis("off");
mfCamZoom(1.05);
mfColorbar("vert");

mfViewPause();
```

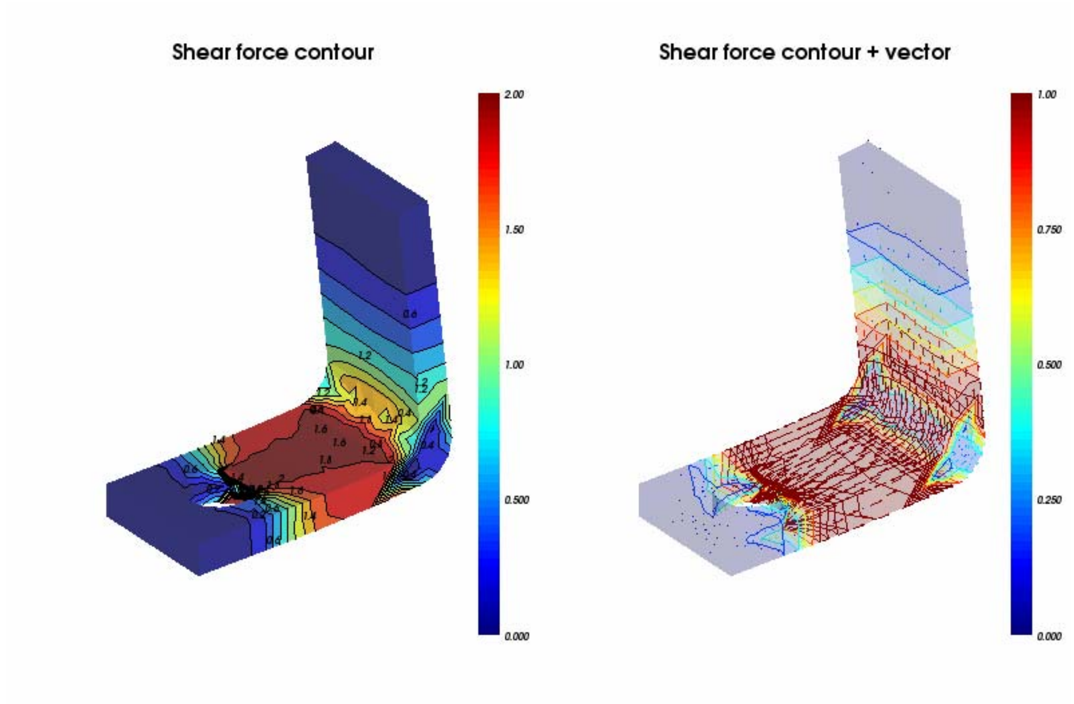


Figure 5.7.1.4 Display the shear force contour and vector of the L-shape steel board

Finally, show iso-surfaces of the shear stress on sliced-planes. There are six in the left-hand figure and eleven in the right-hand figure.

```
// *******************************************************************
//  Shear iso-surface
// *******************************************************************
mfFigure("Shear iso-surface");

// *******************************************************************
//  Shear iso-surface
// *******************************************************************
mfSubplot(1, 2, 1);
mfTitle("Shear iso-surface");

// draw structure element
h = mfTetContour( elem, x+dx, y+dy, z+dz, s_norm);
mfGSet(h, "iso", mfLinspace(0, 2, 6), "label", "on");
mfDrawMaterial(h, "surf", "ambient", 20, "diffuse", 80,     "trans", 80 );

mfDrawMaterial(h, "edge", "colormap", "on" );
```

```
    mfHold("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));
    mfDrawMaterial(h, "surf", "ambient", 20, "diffuse", 80);

    mfColormapRange(0.0, 1.0);
    mfColorbar("vert");
    mfAxis("equal");
    mfAxis("off");

    // ****************************************************************
    //  Shear iso-surface + contour
    // ****************************************************************
    mfSubplot(1, 2, 2);
    mfTitle("Shear iso-surface");

    // draw structure element with contour lines
    h = mfTetContour( elem, x+dx, y+dy, z+dz, s_norm);
    mfGSet(h, "iso", mfLinspace(0, 2, 11), "label", "on");
    mfDrawMaterial(h, "surf", "ambient", 20, "diffuse", 80, "trans", 80 );

    mfDrawMaterial(h, "edge", "colormap", "on" );

    mfHold("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));
    mfDrawMaterial(h, "surf", "ambient", 20, "diffuse", 80);

    mfColormapRange(0.0, 1.0);
    mfAxis("equal");
    mfAxis("off");
    mfCamZoom(1.05);
    mfColorbar("vert");

    mfViewPause();
```

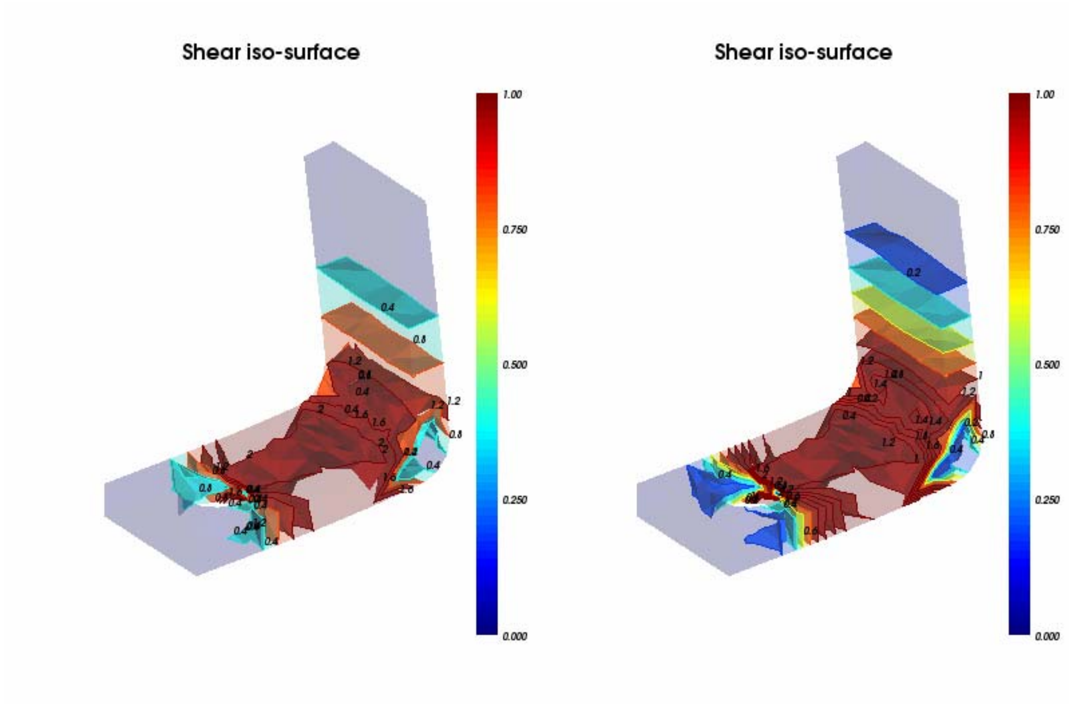


Figure 5.7.1.5 Display the shear iso-surfaces of the L-shape steel board

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-dimension and tetrahedrons in three-dimension from a set of input points.

5.8.1 Two-dimensional Delaunay

Function `mfDelaunay` takes a set of input points and plot the triangular mesh.

Function `mfGetDelaunay` generates the triangular mesh as output that can be used by `mfTriSurf` or `mfTriMesh` for plotting.

Example 5.8.1 Using 2-D Delaunay triangulation

Prompts you to insert the number of points to generate randomly:

```
t = mfInputValue("Input number of random points to generate", 30);
n = (int)t.ToDouble();

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 function `mfDelaunay`. To make the points more obvious, we shall plot them on top of the polygons using function `mfPlot`.

```
mfFigure("Delaunay 2D");
mfTitle("Delaunay 2D");
mfSubplot(1, 2, 1);
h = mfDelaunay( x, y );
mfHold("on");
h = mfPlot( x, y, "ob" );
mfAxis("equal");
```

Then, we shall use a different way to perform Delaunay triangulation on the points using function `mfGetDelaunay`.

```
mfSubplot(1, 2, 2);
mfTitle("Delaunay 2D surface");
tri = mfGetDelaunay( x, y );
h = mfSphere(mfHCat(x, y, z), 0.2, mfV(0, 0, 1) );
mfHold("on");
h = mfTriSurf(tri, x, y, z);
mfDrawMaterial(h, "surf", "trans", 50, "smooth", "on" );
mfAxis("equal");
```

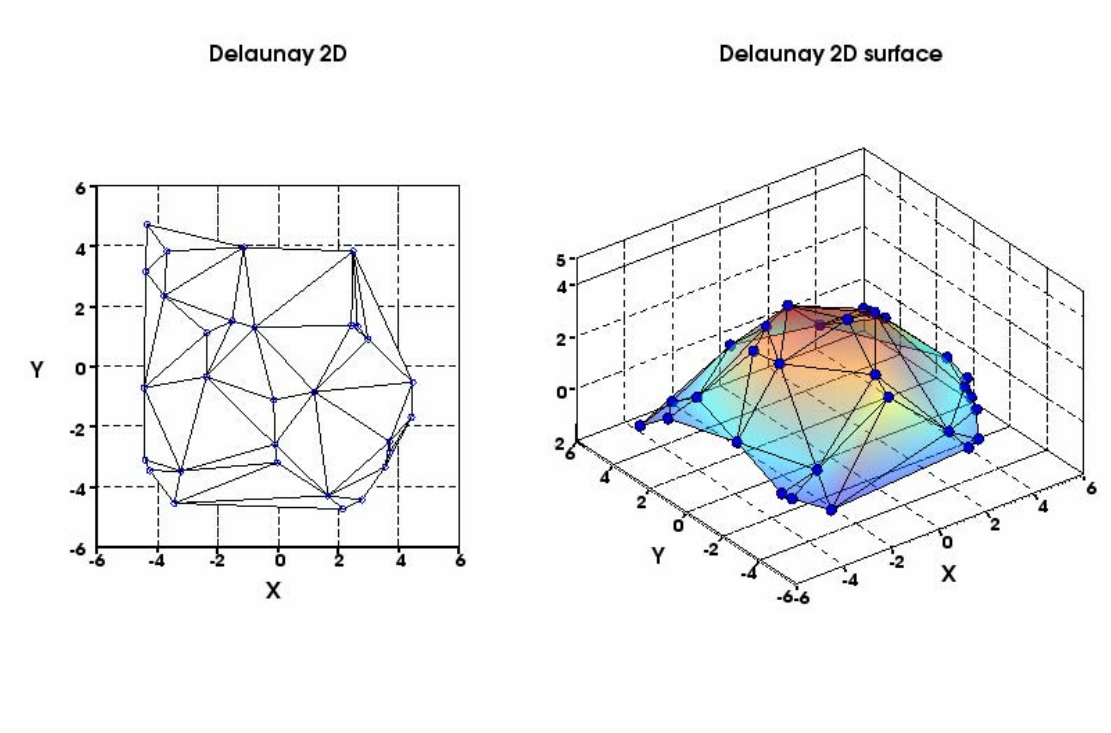


Figure 5.8.1.1 Delaunay 2D and Delaunay 2D surface

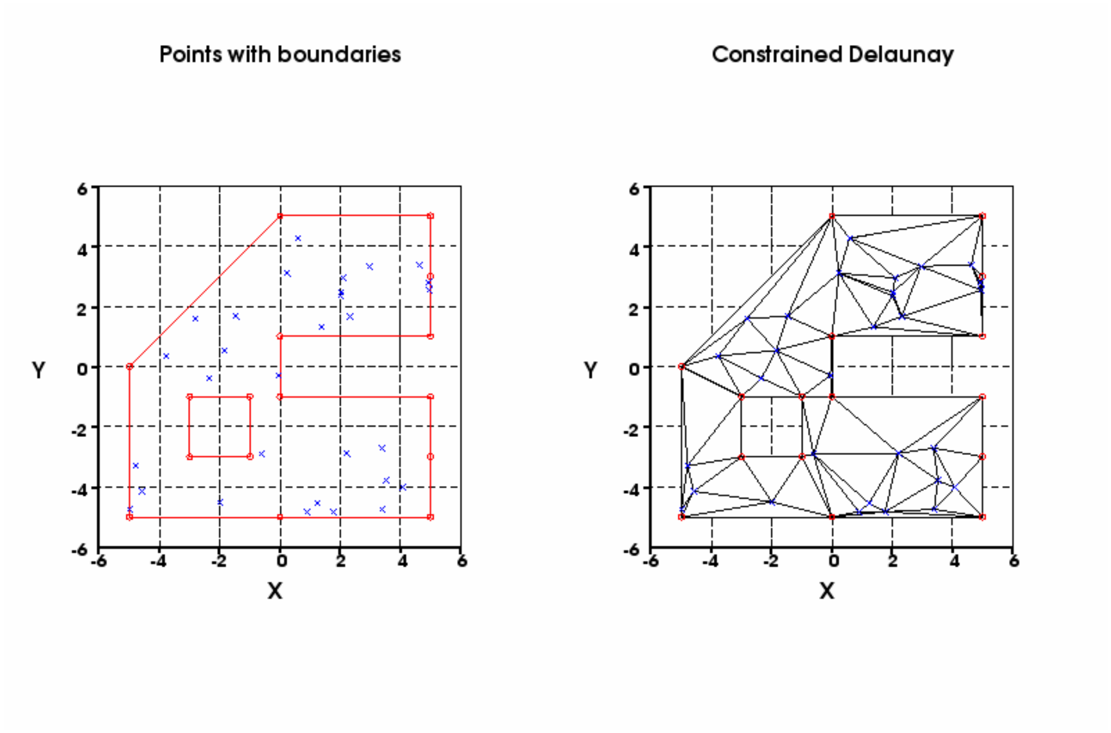


Figure 5.8.1.2 Show the points within boundary and constrained Delaunay

5.8.2 Three-dimensional Delaunay

Function `mfDelaunay3` takes a set of input points and plot the tetrahedron, whereas function `mfGetDelaunay3` generates the triangular mesh as output that can be used by `mfTetSurf` or `mfTetMesh` for plotting.

Example 5.8.2 Using 3-D Delaunay triangulation

Prompts you to insert the number of points to generate randomly:

```
t = mfInputValue("Input number of random points to generate", 30);
n = (int) t.ToDouble();
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 function `mfDelaunay3`.

```
mfFigure("Delaunay 3D");
mfSubplot(1, 2, 1);
mfTitle("Distribution");
h = mfSphere( xyz, 0.2, mfV(0, 0, 1) );

mfSubplot(1, 2, 2);
mfTitle("Delaunay 3D");
h = mfDelaunay3( xyz );
mfDrawMaterial(h, "surf", "trans", 50);
mfHold("on");
h = mfSphere( xyz, 0.2, mfV(0, 0, 1) );

mfViewPause();
```

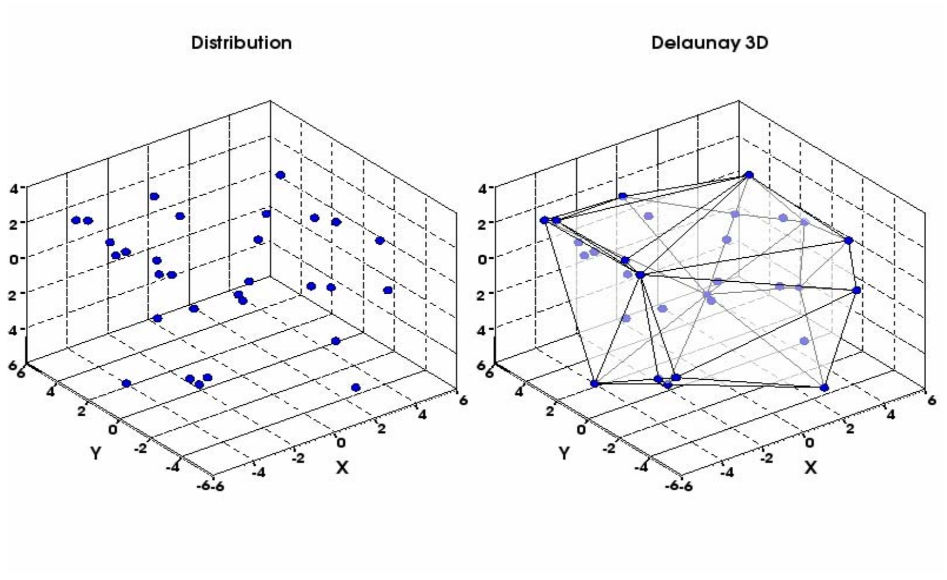


Figure 5.8.2 Display the distributions and Delaunay 3D of the points

Index

A

access mfArray 28

adjust viewpoint 94

analysis panel 127

animation 115

arithmetic operators 57

array terminology 11

ascii 92

axis object 98

B

bounds 12

C

colorbar 104

colormap 100

colormap editor 101

conformance 12

contour 156, 158

contour plot 138

create 3-D models 89

create and initialize mfArray 16

D

Data Viewer 9, 120

Data Viewer menu 123

Data Viewer toolbar 124

declare an mfArray 16

Delaunay Triangulation 169

determine optimal binomial 76

directory structure 13

Display 37

display 3-D objects 94

display mfArray data 37

displaying mode 98

documentation and examples 14

documentations 14

E

element subscripts 28

elementary 3-D objects 149

elmat.lib 21

Exchange binary data between MATLAB and MATFOR 44

extent 48

F

fgl 7

figure windows 84

file I/O 40

filter panel 127

fml 7

function calls 11

G

generate data 89

GetDims 48

GetM() 48

GetN() 48

GetZ() 48

graph annotation 113

Graphics Library 8

Graphics Viewer 8, 84

H

H() 56

HCat() 57

I

image exporting 120

inquiry procedures 45

installation 12

IsComplex() 46

IsEmpty() 46

IsLogical() 46

isosurface 146

iso-surface 158

IsReal() 46

IsString() 46

L

LDiv() 57

least square operations 76

linear graph 131

load data 92

logical inquiry 46

logical operations 50

M

MATFOR parameters 66

matrix division 62

matrix inverse 69

matrix operators and functions 60

matrix table 121

menu 87

MF_COLON 66

MF_E 67

MF_EMPTY 66

MF_EPS 66

MF_INF 67

MF_NAN 67

MF_NINF 67

MF_PI 66

MF_REALMAX 67

MF_REALMIN 67
mfAll 50, 51
mfAny 50, 51
mfArray 7, 15
mfArray creating procedures 30
mfArray I/O 36
mfArray member function/operator 30
mfArray operator 30
mfArray operators 55
mfb 92
mfColon 20
mfCone 150
mfContour 138
mfContour3 138
mfCreateRealMatrix 20
mfCube 150
mfCylinder 150
mfDelaunay 169
mfDelaunay3 172
mfEig 73
mfEye 21
mfFind 51
mfGetCurrentDraw 116
mfGetDelaunay 169
mfHC 20
mfInv 69
mfIsoSurface 146
mfLinspace 21, 90
mfLoad 42
mfLoad.m 43
mfLoadAscii 42, 92
mfMagic 21
mfMesh 136
mfMeshc 136
mfMeshgrid 21
mfOnes 21
mfOutline 146
mfPColor 140
mfPlot 131
mfPlot3 133
mfQuiver 148
mfRand 21
mfRepmat 21
mfReshape 21
mfRibbon 133
mfSave 41
mfSave.m 43
mfSize 48
mfSliceIJK 145
mfSlicePlane 145
mfSliceXYZ 145
mfSolidContour 138
mfSolidContour3 .... 138, 144
mfSphere 150

mfStreamlLine..148

mfSurf.....................................136, 143, 144

mfSurfc..136

mfTetContour..157

mfTetIsosurface.......................................157

mfTetMesh.......................................157, 172

mfTetSurf...157, 172

mfTriContour..153

mfTriMesh..153, 169

mfTriSurf...153, 169

mfTube...133

mfVC...20

mfWindowCaption.....................................84

mfWindowPos..84

mfWindowSize...84

mfZeros..21

molecule...150

msAnnotation..114

msAxis...82, 99

msAxisGrid...99

msAxisWall...99

msBackGroundColor..................................82

msCamPan...96

msCamProj..98

msCamZoom..97

msColorbar..104

msColormap...100

msDisplay...37

msDrawMaterial......................................105

msExportImage..120

msFigure...82

msGSet..116

msMeshgrid...89

msOutline..138

msPlot...84

msRecordEnd...118

msRecordStart..118

msSaveAscii...41

msShading...82

msSubplot..85

msSurf...116

msText...114

msTitle...113

msView..94

msViewPause.......................................40, 116

msXLabel...113

msYLabel...113

msZLabel...113

Mul()...56

Multi-dimension subscript..........................29

N

Numerical Library...8

O

orthographic...98

P

perspective 98

plot data 81

Pow() 56

primitives 149

procedure naming conventions 10

procedures with “mf” as prefix 10

pseudocolor plot 140

Q

quiver 148

R

rank 11

RDiv() 57

recording 115

regression model 71

relational operators 58

rescale objects 97

S

sample type 125

Scalar subscript 28

shading 100

shape 12, 48

shift objects 95

size 12, 48

sliced-planes 145

snapshot panel 125

status bar 128

streamline 148

subplots 85

surface 143, 153, 158

surface plot 136

T

T() 56

technical support 14

text annotation 114

texture 100

texture mapping 109

three-dimensional Delaunay 172

three-dimensional linear graph 133

toolbar 87

two-dimensional Delaunay 169

two-dimensional linear graph 131

U

unstructured grids 157

unstructured mesh 152

V

VCat() 57

vector field 148

visualization 81

volume rendering 142

W

window frame 84