Chapter One
Introduction to Matlab

1.1 Introductions

The name MATLAB stands for MATrix LABoratory. MATLAB was written originally to provide easy access to matrix software developed by the LINPACK (Linear System Package) and EISPACK (Eigen System Package) projects.

MTLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming environment. Furthermore, MATLAB is a modern programming language environment: it has sophisticated ***data structure***, contains built-in editing and debugging tools, and supports ***object-oriented*** programming. These factors make MATLAB an excellent tool for teaching and research.

MATLAB has many advantages compared to conventional computer languages (e.g. C, FORTRAN) for solving technical problems. MATLAB is an interactive system whose basic data element is an ***array*** that does not require dimensioning. The software package has been commercially available since 1084 and is now considered as a standard tool at most universities and industries worldwide.

It has powerful built-in routines that enable a very wide variety of computations. It also has easy to use graphics commands that make the visualization of results immediately available. Specific applications are collected in packages referred to as toolbox. There are toolboxes for signal processing, symbolic computation, control theory, simulation, optimization, and several other fields of applied science and engineering. Typical uses of Matlab include:

1. Math and Computation.
2. Algorithm development.
3. Modeling, simulation and prototyping.
4. Data analysis, exploration, and visualization.
5. Scientific and engineering graphics.
6. Application development, including graphical user interface building.

1.2 Strengths of Matlab

1. MATLAB may behave as a calculator or as a programming language.
2. MATLAB combine nicely calculation and graphic plotting.
3. MATLAB is relatively easy to learn.
4. MATLAB is interpreted (not compiled), errors are easy to fix.
5. MATLAB is optimized to be relatively fast when performing matrix operations.
6. MATLAB does have some object-oriented elements.

1.3 Weaknesses of Matlab

1. MATLAB is not general purpose programming language such as C, C++, or FORTRAN.
2. MATLAB is designed for scientific computing and is not well suitable for other applications.
3. MATLAB is an interpreted language, slower than a compiled language such as C++
4. MATLAB commands are specific for MATLAB usage. Most of them do not have a direct equivalent with other programming language commands.

1.4 Simple Math:

The arithmetic operations that we can do are:

Table 1.1 Basic Mathematical Operations

Operation	Symbol	Example
Addition , a+b	+	5+3
Subtraction, a-b	-	5.05-3.111
Multiplication, a*b	*	0.124*3.14
Left division , a\b	\	5\3
Right division , b/a	/	3/5 = 5\3
Exponentiation, a^b	^	5^2

As an example of a simple interactive calculation, just type the expression you want to evaluate. For example, let's suppose you want to calculate the expression, 1+2*3.

You type it at the prompt (>>) as follows:

Ex1:

```
>> 1+2*3
ans =
7
```

You will have noticed that if you do not specify an output variable, MATLAB uses a default variable ans short for answer, to store the results of the current calculation. Note that the variable ans is created (o overwritten, if it is already existed). To avoid this, you may assign a value to a variable or output argument name. For example:

Ex2:

```
>> 1+2*3
```

```
ans =
 7
```

Note: we will not mention Enter key any more for later examples as it was clear stated.

Ex3:

```
>> x=7
x =
     7
>> 4*x
ans =
    28
```

1.5 Creating MATLAB Variables

MATLAB variables are created with an assignment statement. The syntax of variable assignment is: variable name a value (or an expression)

For example,

>>x=expression

Where expression is a combination of numerical values, mathematical operators, variables, and function calls. On the other words, expression can involve:

Manual entry

Built-in functions (Ready Matlab functions).

User-defined functions

Once a variable has been created, it can be assigned. In addition, if you do not wish to see the intermediate results, you can suppress the numerical output by putting a semicolon (;) at the end of the line. The the sequence of commands looks like this:

Ex5:

```
>> t=5;
>> t=t+1
t =
6
```

1.6 Error Messages

If we enter an expression incorrectly, MATLAB will return an error message. For example, in the following, we left out the multiplication sign, *, in the following expression:

```
>> x=10;
>> 5x
```

??? 5x
Error: Unexpected MATLAB expression.

1.7 Making Corrections

To make corrections, we can of course retype the expressions. But if the expression is lengthy, we make more mistakes by typing a second time. A previously typed commands can be recalled with the up-arrow key ↑. When the command is displayed at the command prompt, it can be modified if needed and executed.

1.8 Controlling the Hierarchy of Operations or Precedence

Let's consider the previous arithmetic operation, but now we will include *parentheses*. For example, 1+2*3 will become (1+2)*3

```
Ex 6:
>> (1+2)*3
ans =
        9
```

While we have seen that:

```
>> 1+2*3
ans =
        7
```

Ex 7:Find the result of

$$\frac{1}{2+3^2}+\frac{4}{5}*\frac{6}{7}$$

```
>> 1/(2+3^2)+4/5*6/7
ans =
        0.7766
```

(but, if parenthesis are missing)
1/2+3^2+4/5*6/7

```
ans =
        10.1857
```

Table 1.2 Hierarchy of arithmetic operations

Precedence	**Mathematical operations**
First	The contents of the parentheses are evaluated first, starting from the innermost parentheses and working outward.
Second	All exponentials are evaluated, working from left to right.
Third	All multiplications and divisions are evaluated, working from left to right.
Fourth	All additions and subtractions are evaluated, starting from left to right

1.9 Entering Multiple Statements per Line

It is possible to enter multiple statements per line. Use commas (,) or semicolons (;) to enter more than one statement at once. Commas (,) allow multiple statements per line without suppressing output.

```
Ex8:
>>a=7; b=cos(a); c=cosh(a);
b=
0.6570
c=
548.3170
```

1.10 Miscellaneous Commands

Here are few additional useful general commands:
To clear the command window, type **clc** (variable not cleared from workspace)
To abort a MATLAB computation, type ctrl+c
To clear all variables from workspace window, type clear or Clear All.
Display command can show any text in command window, (display 'Hello').
Who command gives a list of all variables stored in memory.
Whos give more details which include size, space allocation, and class of the variables.

Also at the start of Matlab some variables have a value so that we can use them easily. Those values can be changed but it is not wise to do it. Those variables are:

Table 1.3 Special variable Values

ans	The default variable name used for results
pi	3.14
eps	The smallest possible number such that, when added to
one	Creates a number greater than one on the computer
flops	Count of floating point operations. (not used in ver. 6)

inf	Stands for infinity (e.g.:1/0)
NaN	Not a number (e.g. 0/0)
I (and) j	i=j=√
nargin	Number of function input arguments used
nargout	Number of function output arguments used
realmin	The smallest usable positive real number
realmax	The largest usable positive real number

Chapter Two
Mathematical Functions

1.1 Introduction

MATLAB offers many predefined mathematical functions for technical computing which contains a large set of mathematical functions.

Typing help ***elfun*** and help ***specfun*** calls up full lists of ***elementary*** and ***special*** functions respectively.

There is a long list of mathematical functions that are built into MATLAB. These functions are called ***built-ins***. Many standard mathematical functions, such as sin(x), cos(x), tan(x), e^x, ln(x), are evaluated by the function sin, cos, tan, exp, and log respectively in MATLAB. **Table 2.1** lists some commonly used functions, where variables x and y can be numbers, vectors, or matrices.

Table 2.1 Elementary Functions

Matlab Name	**Comment**
cos(x)	Cosine
sin(x)	Sine
tan(x)	Tangent
atan(x)	Arc tangent
acos(x)	Arc cosine
asin(x)	Arc sine
abs(x)	Absolute value
sign(x)	Signum function
max(x)	Maximum value
min(x)	Minimum value
ceil(x)	Round towards +∞
floor(x)	Round towards -∞
exp(x)	Exponential

round(x)	Round to nearest integer
sqrt(x)	Square tool
rem(x)	Remainder after division
log(x)	Natural logarithm
log10(x)	Common logarithm
angle(x)	Phase angle
conj(x)	Complex conjugate

Example

We illustrate here some typical examples which related to the elementary functions previously defined. As a first example, the value of the expression $y=e^{-a}\sin(x)+10\sqrt{}$ for a =5, x=2, and y=8 is computed by

```
>> a=5;x=2;y=8
y =
     8
>> y=exp(-a)*sin(x)+10*sqrt(y)
y =
28.2904
```

The subsequent examples are:

```
>> log(142)
ans =
    4.9558
>> log10(142)
ans =
2.1523
```

Note the difference between the natural logarithm log(x) and the decimal logarithm (base 10) log10(x)

To calculate $\sin(\pi/4)$ and e^{10}, we enter the following commands in MATLAB,

```
>> sin(pi/4)
ans =
    0.7071
>> exp(10)
```

```
ans =
2.2026e+004
```

1.2 Basic Plotting:

1.2.1 Creating Simple Plots

The basic MATLAB graphing procedure, for example in 2D, is to take a vector of x-coordinates, $\mathbf{x}=(x_1; \ldots..; x_n)$ and a vector of y-coordinates, $\mathbf{y}= (y_1;\ldots; y_n)$, locate the points $(x_i; y_i)$, with i=1; 2; …; n and then join them by straight lines. You need to prepare *x* and *y* in an identical array form; namely, *x* and *y* are both row arrays and column arrays of the ***same length.***

The MATLAB command to plot a graph is plot(x,y). The vectors x=(1; 2; 3; 4; 5; 6) and y=(3;-1;2;4;5;1) produce the picture shown in figure 1.1.

```
>> x=[1 2 3 4 5 6];
>> y=[3 -1 2 4 5 1];
>> plot (x,y)
```

Note: the plot function has different forms depending on the input arguments. If y is a vector plot (y) produces a piecewise linear graph of the elements of y versus the index of the elements of y. If we specify two vectors, as mentioned above, plot (x,y) produces a graph of y versus x.

For example, to plot the function sin (*x*) on the interval [0,2π], we first create a vector of *x* values ranging from 0 to 2π, then compute the *sine* of these values, and finally plot the result:

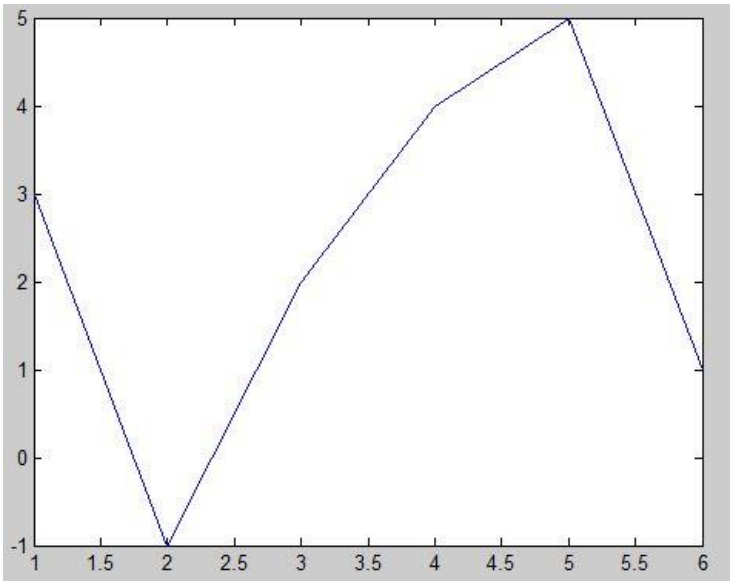


Figure 1.1 Pot for the vectors x and y

```
>> x=0:pi/100:2*pi;
>> y=sin(x);
>> plot(x,y);
```

Notes:
0:pi/100:2*pi yields a vector that:
Start at 0,
Takes steps (or increments) of π/100,
Stops when 2π is reached.
If you omit the increment, MATLAB automatically increments by 1.

1.2.2 Adding Titles, Axis Labels, and Annotations

MATLAB enables you to add axis labels and titles. For example, using the graph from the previous example, add x-axis and y-axis labels.

Now label the axis and add a title. The character \pi creates the symbol π. An example of 2D plot is shown in Figure 1.2.

```
>> x=0:pi/100:2*pi;
>>y=sin(x);
>>plot(x,y);
>>xlabel('x=0:2\pi')
>>ylabel('sine of x')
>>title('Plot of the sine function')
```

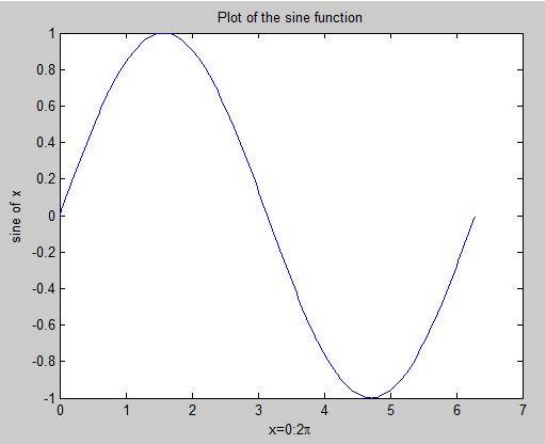


Figure 1.2 Plot of the Sine Function

The color of a single curve is, by default, blue, but other colors are possible. The desired color is indicated by a third argument. For example, red is selected by plot(*x*, *y*, *'r'*). Note the single quotes, ‘’, around r.

1.2.3 Multiple data sets in one plot

Multiple (*x*;*y*) pairs arguments create multiple graphs with a single call to plot. For examples, these statements plot three related functions of x:y_1=2 cos(x), y_2=cos(x), and y_3=0.5*cos(x), in the interval $0 \leq x \leq 2\pi$.

```
>> x=0:pi/100:2*pi;
>>y1=2*cos(x);
>>y2=cos(x);
>>y3=0.5*cos(x);
>>plot(x,y1,'--',x,y2,'-',x,y3,':')
>>xlabel('0\leq x\leq 2\pi')
>>ylabel('Cosine functions')
>>legend('2*cos(x)','cos(x)','0.5*cos(x)')
>>title('Typical example of multiple plots')
>>axis([0 2*pi -3 3])
```

The result of multiple data sets in one graph plot is shown in Figure 1.3.

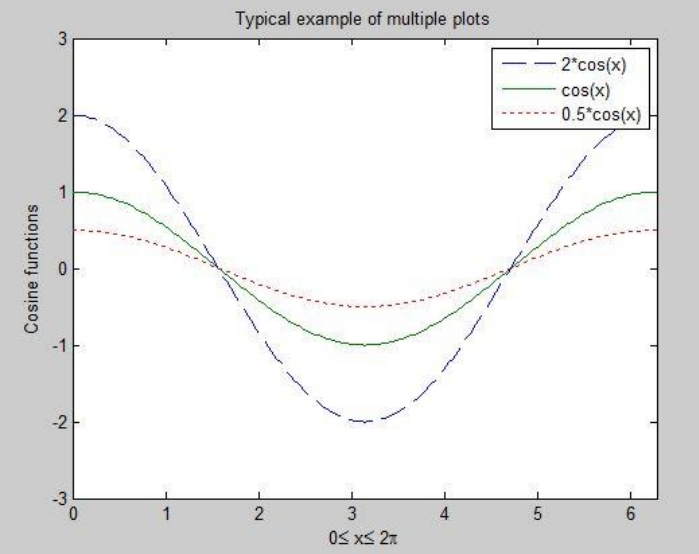


Figure 1.3 Typical example of multiple plots

By default, MATLAB uses line style and color to distinguish the data sets plotted in the graph. However, you can change the appearance of these graphic components or add annotations to the graph to help explain your data for presentation.

1.2.4 Specifying Line styles and Colors

It is possible to specify line styles, colors, and markers (e.g., circles, plus signs, ….) using the plot command:

Plot (x,y,'style_clor_marker')
Where style color marker is a triplet of values from Table 1.4.
To find additional information, type help plot or doc plot.

Table 2.2 Attributes for Plot

Symbol	Color	Symbol	Line Style	Symbol	Marker
k	Black	-	Solid	+	Plus sign
r	Red	--	Dashed	o	Circle
b	Blue	…	Dotted	8	Asterisk
g	Green	-.	Dash-dot	.	Point
c	Cyan	None	No line	X	Cross
m	Magenta			S	Square
y	Yellow			D	Diamond

Chapter Three
Vectors and Matrices

3.1 Introduction

Arrays: an array is a mathematical structure that has a collection of numerical **elements**, each of these elements referenced by an **index** which represents the location of that element within the array. This index has integer sequence and any element can be accessed by this sequence.

A **vector** is one dimension array, it can be either **row vector** which consists of one row and several columns or (n*1) elements size. **Column vector** is also one dimensional array but with one column and several rows or (1*n) elements size.

A **matrix** is normally a two dimensional array with (n*m) elements in which n represent the number of rows while m represents the number of columns. In some cases when (n=m) then such a matrix is called a square matrix.

3.2 Defining and Entering Vectors and Matrices Elements

There are several methods to enter elements to vectors and matrices in Matlab:

1. Entering an explicit list of elements.
2. Loading data from external file (or importing from an application such as excel).
3. Generating elements using function (such as (rand) function for random data).
4. Building from other arrays elements.

Array elements in rows are represented by semi columns (;), while columns in either spaces or commas (,) in between.

3.3 Matrix Generation

Matrices are fundamental to MATLAB. Therefore, we need to become familiar with matrix generation and manipulation. Matrices can be generated in several ways.

3.3.1 Entering a vector

A vector is a special case of matrix. The purpose of this section is to show how to create vectors and matrices in MATLAB. As discussed earlier, an array of dimension 1*n is called a row vector, whereas an array of dimension m*1 is

called a column vectors. The spaces or by commas. For example, to enter a row vector, v, type.

```
>> v=[1 4 7 10 13]
v =
    1     4     7    10    13
```

Column vectors are created in a similar way; however, semicolon (;) must separate the components of a column vector,

```
>> w=[1;4;7;10;13]
W =
    1
    4
    7
   10
   13
```

On the other hand, a row vector is converted to a column vector using the transpose operator.

The transpose operation is denoted by an apostrophe or a single quote(‘).

```
>> w=v'
w =
    1
    4
    7
   10
   13
```

Thus, v(1) is the first element of vector v, v(2) its second element, and so forth. Furthermore, to access blocks of elements, we use MATLAB’s colon notation (:). For example, to access the first three elements of v, we write,

```
>> v(1:3)
ans =
    1     4     7
```

Or, all elements from the third through the last elements,

```
>> v(3:end)
ans =
    7    10    13
```

Where end signifies the last element in the vector. If v is a vector, writing >> v(:) produces a column vector, whereas writing >> v(1:end) produces a row vector.

3.3.2 Entering a matrix

A matrix is an array of numbers. To type a matrix into MATLAB you must
Begin with a square bracket, [.
Separate elements in a row with spaces or commas (,).
Use a semicolon (;) to separate rows.
End the matrix with another square bracket,].
Here is a typical example. To enter a matrix A, such as,
A=[ ]

```
>> A=[1 2 3;4 5 6;7 8 9]
A =
   1   2   3
   4   5   6
   7   8   9
```

Note that the use of semicolons (;) here is different from their use mentioned earlier to suppress output or to write multiple commands in a single line.

Once we have entered the matrix, it is automatically stored and remembered in the workspace. We can then view a particular element in a matrix by specifying its location. We write,

```
>> A(2,1)
ans =
    4
```

A(2,1) is an element located in the second row and first column. Its value is 4.

3.4 Matrix Indexing

We select elements in a matrix just as we did for vectors, but now we need two indices. The element of row *i* and column *j* of the matrix A is denoted by A(i,j). Thus, A(i,j) in MATLAB refers to the element A_{ij} of matrix A. the *first* index is the *row* number and the *second* index is the *column* number. For example, A(1,3) is an element of *first* row and *third* column. Here, A(1,3)=3.

Correcting any entry is easy through indexing. Ere we substitute A(3,3)=9 by A(3,3)=0. The result is

```
>> A(3,3)=0
A =
   1   2   3
   4   5   6
   7   8   0
```

Single elements of a matrix are accessed as A(i,j), where $i \geq 1$ and $j \geq 1$. Zero or negative subscripts are not supported in MATLAB.

3.4.1 Colon Operator

The colon operator will prove very useful and understanding how it works is the key to efficient and convenient usage of MATLAB. It occurs in several different forms.

Often we must deal with matrices or vectors that are too large to enter one element at a time. For example, suppose we want to enter a vector x consisting of points (0; 0:1; 0:2; 0:3; …; 5). We can use the command

```
>>x=0:0.1:5;
```

The row vector has 51 elements.

3.4.2 Linear Spacing

On the other hand, there is a command to generate linearly spaced vectors: **linspace**. It is similar to the colon operator (:), but gives direct control over the number of points. For example,

Y=linspace(a,b)

Generates a row vector y of 100 points linearly spaced between and including a and b.

Y=linspace(a,b,n)

Generates a row vector y of n points linearly spaced between and including a and b. this is useful when we want to divide an interval into a number of subintervals of the same length.

For example,

>>theta=linspace(0,2*pi,101)

Divides the interval [0; 2π] into 100 equal subinterval, then creating a vector of 101 elements.

3.4.3 Colon Operator in a Matrix

The colon operator can also be used to pick out a certain row or column. For example, the statement A(m::n,k:l) specifies rows m to n and column k to l. subscript expressions refer to portions of a matrix. For example,

```
>> A(2,:)
ans =
   4    5    6
```

Is the second row elements of A.

```
>> A(:,2:3)
ans =
   2    3
   5    6
   8    0
```

The colon operator can also be used to extract a sub-matrix from a matrix A.

A(:,2:3) is a sub-matrix with the last two columns of A.

A row or a column of a matrix can be deleted by setting it to a null vector, [].

```
>> A(:,2)=[]
A =
   1    3
   4    6
   7    0
```

3.4.4 Creating a Sub-matrix

To create a submatrix B consisting of rows 2 and 3 and columns 1 and 2 of the matrix A, do the following

```
>>  B=A([2 3],[1 2])
B =
      4    6
      7    9
```

To interchange rows 1 and 2 of A, use the vector of row indices together with the colon operator.

```
>>  C=A([2 1 3],:)
    C =
        4    6
        1    3
        7    9
```

It is important to note that the colon operator (:) stands for all columns or all rows. To create a vector version of matrix A, do the following

```
>> A(:)
  ans =

     1
     4
     7
     3
     6
     9
```

The submatrix comprising the intersection of rows p to q and columns r to s is denoted by A(p:q,r:s).
As a special case, a colon (:) as the row or column specifier covers all entries in that row or column; thus

A(:,j) is the j^{th} column of A, while
A(i,;) is the i^{th} row, and
A(end,:) picks out the last row of A.

The keyword **end**, used in A(end,:), denotes the last index in the specified dimension. Here are some examples.

```
>> A
A =
    1    3
    4    6
    7    0
```

?
?
?

3.4.5 Deleting Row or Column

To delete a row or column of a matrix, use the empty vector operator,[].

```
>> A(3,:)=[]
A =
    1    3
    4    6
```

Third row of matrix A is now deleted. To restore the third row, we use a technique for creating a matrix.

```
>> A=[A(1,:);A(2,:);[7 8 0]]
A =
    1    2    3
    4    5    6
    7    8    0
```

Matrix A is now restored to its original form.

3.5 Dimension

To determine the dimensions of a matrix or vector, use the command size. For example, means 3 rows and 3 columns.

```
>> size(A)
ans =
    3    3
```

Or more explicitly with,
>>[m,n]=size(A)

3.7 Transposing a Matrix

The transpose operation is denoted by an apostrophe or a single quote ('). It flips a matrix about its main diagonal and it turns a row vector into a column vector. Thus,

```
>> A'
ans =
    1    4    7
    2    5    8
    3    6    0
```

By using linear algebra notation, the transpose of $m \times n$ real matrix A is the $n \times m$ matrix that results from interchanging the rows and columns of A. the transpose matrix is denoted A^T.

3.8 Concatenating Matrices

Matrices can be made up of sub-matrices. Here is an example. First, let's recall our previous matrix A.

A=[ ]

Then new matrix B will be,

```
>>  B=[A 10*A; -A [1 0 0; 0 1 0; 0 0 1]]
B =
    1    2    3   10   20   30
    4    5    6   40   50   60
    7    8    0   70   80    0
   -1   -2   -3    1    0    0
   -4   -5   -6    0    1    0
   -7   -8    0    0    0    1
```

3.9 Matrix Generators

MATLAB provides functions that generate elementary matrices. The matrix of zeros, the matrix of ones, and the identity matrix are returned by the functions zros, ones, and eye respectively.

Table 3.1 Elementary Matrices

eye(m, n)	Returns an m by n matrix with 1on the main diagonal
eye(n)	Returns an n by n square identity matrix
Zeros(m, n)	Returns an m by n matrix of zeros
Ones(m, n)	Returns an m by n matrix of ones
Diag(A)	Extract the diagonal of matrix A
Rand(m, n)	Returns an m by n matrix of random numbers

For a complete list of elementary matrices and matrix manipulations, type ***help elmat*** or ***doc elmat***. Here are some examples.

1.
```
>> b=ones(3,1)
b =
   1
   1
   1
```

Equivalently, we can define b as>>b=[1;1;1]

2.
```
>> eye(3)
ans =
     1     0     0
     0     1     0
     0     0     1
```

3.
```
>> c=zeros (2,3)
c =
     0     0     0
     0     0     0
```

In addition, it is important to remember that the three elementary operations of addition (+), subtraction (-), and multiplication (*) apply also to matrices whenever the dimensions are compatible.
Two other important matrix generation functions are rand and randn, which generate matrices of (pseudo-) random numbers using the same syntax as eye.
In addition, matrices can be constructed in a block form. With C defined by C=[1 2; 3 4], we may create a matrix D as follows

```
>> C=[1 2; 3 4]
C =
     1     2
     3     4
```

```
>> D=[C zeros(2); ones(2) eye(2)]
D =
     1     2     0     0
     3     4     0     0
     1     1     1     0
     1     1     0     1
```

3.10 Array Operations

MATLAB has two different types of arithmetic operations: matrix arithmetic operations and array arithmetic operations. We have seen matrix arithmetic operations in the previous lab. Now, we are interested in array operations.

3.10.1 Matrix Arithmetic Operations

As we mentioned earlier, MATLAB allows arithmetic operations: +, -, *, and ^ to be carried out on matrices. Thus,

A+B or B+A is valid if A and B are of the same size.
A*B is valid if A's number of column equals B's number of rows
A^2 is valid if A is square and equals A*A
α*A or A*α multiplies each element of A by α

3.10.2 Array Arithmetic Operations

On the other hand, array arithmetic operations or array operations for short, are done element-by-element. The period character, (.), distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition (+) and subtraction (-), the character pairs (+) and (-) are not used. The list of array operators is shown below in Table 3.2.

Table 3.2 Array Operators

.*	Element-by-element multiplication
./	Element-by-element division
.^	Element-by-element exponentiation

If A and B are two matrices of the same size with elements $A=[a_{ij}]$ and $B=[b_{ij}]$, then the command , **>>C=A.*B**
Produces another matrix C of the same size with elements $c_{ij}=a_{ij}b_{ij}$. For example, using the same 3 × 3 matrices,

```
>> A=[      ], B=[      ]
A =
    []
B =
    []
```

We have,

```
>> C= A.*B
C = []
```

To raise a scalar to a power, we use for example the command $10^{\wedge 2}$. If we want the operation to be applied to each element of a matrix, we use $.^{\wedge 2}$. for example, if we want to produce a new matrix whose elements are the square of the elements of the matrix A, we enter

```
>> A=[1 2 3; 4 5 6]
A =
    1    2    3
    4    5    6
>> A.^2
ans =
    1    4    9
   16   25   36
```

The relations below summarize the above operations. To simplify, let's consider two vectors U and V with elements $U=[u_i]$ and $V=[v_j]$.

U: *V produces $[u_1v_1\ u_2v_2 \ldots \ldots \ldots u_nv_n]$

U:/V produces $[u_1/v_1\ u_2/v_2 \ldots \ldots \ldots u_n/v_n]$

U: ^V produces $[u^{v1}\ u^{v2} \ldots \ldots \ldots u^{vn}]$

Table 3.3 Summary of Matrix and Array Operations

Operation	Matrix	Array
Addition	+	+
Subtraction	-	-
Multiplication	*	.*
Division	/	./
Left division	\	.\
Exponential	^	.^

3.11 Solving Linear Equation

One of the problems encountered most frequently in scientific computation is the solution of systems of simultaneous linear equations. With matrix notation, a

Ax=b

Where there are as many equations as unknown. A is a given square matrix of order n, b is a given column vector of n components, and x is an unknown column vector of n components.

In linear algebra we learn that the solution to $\boldsymbol{Ax=b}$ can be written as $\boldsymbol{x=A^{-1}}$ is the inverse of A.

For example, consider the following system of linear equations

$\boldsymbol{x+2y+3z=1}$

$\boldsymbol{4x+5y+6z=1}$

$\boldsymbol{7x+8y=1}$

The coefficient matrix A is

A=[ ], and the vector b=[ ]

With matrix notation, a system of simultaneous linear equations is written

$\boldsymbol{Ax=b}$

This equation can be solved for x using linear algebra. The result is $\boldsymbol{x=A^{-1}b.}$

There are typically two ways to solve for x in MATLAB:

1. Use the matrix inverse, inv.

```
>> A=[1 2 3; 4 5 6; 7 8 0];
>> b=[1;1;1];
>> x=inv(A)*b
x =
   -1.0000
    1.0000
   -0.0000
```

2. The second one is to use the backslash (n) operator. The numerical algorithm behind this operator is computationally efficient. This is a numerically reliable way of solving system of linear equations by using a well-known process of Gaussian elimination.

```
>> A=[1 2 3;4 5 6;  7 8 0]
>> b=[1;1;1];
>> x= A\b
x =
   -1.0000
    1.0000
   -0.0000
```

This problem is at the heart of many problems in scientific computation. Hence it is important that we know how to solve this type of problem efficiently.

3.12 Matrix Inverse

Let's consider the same matrix A.

A= [ ]

Calculating the inverse of A manually is probably not a pleasant work. Here the hand-calculation of A^{-1} gives as final result:

A^{-1} = [ ]

In MATLAB, however, it becomes as simple as the following commands:

```
>> A=[1 2 3;4 5 6;7 8 0];
>> inv(A)
ans =
   -1.7778    0.8889   -0.1111
    1.5556   -0.7778    0.2222
   -0.1111    0.2222   -0.1111
```

Which is similar to:

A^{-1}=[ ]

And the determinant of A is

```
>> det(A)
ans =
   27.0000
```

Solved Examples

Example (1): write MATLAB code to create a row vector with 5 elements.

```
>> A=[2 5 6 2 7]
A =
    2     5     6     2     7
```

Example (2): write MATLAB code to create a column vector with 6 elements.

```
>> B=[6;3;1;0;1;5]
B =
     6
     3
     1
     0
     1
     5
```

Example (3): Write MATLAB code to represent a row vector with 3 elements of sin(30), sin(60), and sin(90)

```
>>  X=[sin(30) sin(60) sin(90)]
X =
  -0.9880   -0.3048    0.8940
```

Example (4): Build column vector elements from a(3*1) vector and (2*1) vector

```
>> x=[3;1;5];
>> y=[-6;0];
>> z=[x;y]
z =
     3
     1
     5
    -6
     0
```

Example (5): From the previous example, find the transpose of (z).

```
>> z'
ans =
    3     1     5    -6     0
```

Example (6): Find the log of each of the 3 elements in a row vector.

```
>> v=[2 5 6];
>> log(v)
ans =
   0.6931    1.6094    1.7918
```

Example (7): Generate a row vector with element values from 1 to 10.

```
>> G=[1:10]
G =
    1     2     3     4     5     6     7     8     9    10
```

Example (8): Generate randomly a column vector with 4 element values between 0 and 10.

```
>> K=rand(4,1)*10
K =
   8.1472
   9.0579
   1.2699
   9.1338
```

Example (9): Find the transpose for A(3*4) matrix.

```
>> A=[4 6 0; 2 -1 0; 8 0 1;7 -3 1]
A =
    4     6     0
    2    -1     0
    8     0     1
    7    -3     1
```

```
>> A'
ans =
    4    2    8    7
    6   -1    0   -3
    0    0    1    1
```

Example (10): display the seventh element in a10 elements vector.

```
>> V=[6 1 2 4 5 0 -4 8 0 1];
>> V(7)
ans =
   -4
```

Example (11): Display from the second to the sixth elements of the previous example.

```
>> V(2:6)
ans =
    1    2    4    5    0
```

Example (12): Display the element B(3,4) in a 5*5 matrix.

```
>> B=[2 5 0 -1 4;; 6 0 0 1 -4;;3 0 1 0 1;;7 1 0 8 3;;1 0 -4 1 2]
B =
    2    5    0   -1    4
    6    0    0    1   -4
    3    0    1    0    1
    7    1    0    8    3
    1    0   -4    1    2
>> B(3,4)
ans =
    0
```

Example (13): Display the elements from row (2) Column (4) to row (3) Column (5) in a 2D matrix with elements (5*5)?

```
>> B=[2 5 0 -1 4;; 6 0 0 1 -4;;3 0 1 0 1;;7 1 0 8 3;;1 0 -4 1 2]
B =
    2    5    0   -1    4
    6    0    0    1   -4
    3    0    1    0    1
    7    1    0    8    3
    1    0   -4    1    2
>> B(2:3,4:5)
ans =
    1   -4
    0    1
```

Example (14): Add 10 to the element 5 in an 8 elements vector.

```
>> A=[2 0 1 2 -4 3 7 5]
A =
    2    0    1    2   -4    3    7    5
>> A(5)=A(5)+10
A =
    2    0    1    2    6    3    7    5
```

Example (15): Subtract 4 from the elements of previous examples.

```
>> A=A-4
A =
   -2   -4   -3   -2    2   -1    3    1
```

Example (16): Add the 6 elements column vector A to column vector B and put result in vector C.

```
>> A=[1;3;5;0;6], B=[5;-1;0;-3;4]
A =
    1
    3
    5
    0
    6
```

```
B =
     5
    -1
     0
    -3
     4
>> C=A+B
C =
     6
     2
     5
    -3
    10
```

Example (17): Add two (3*4) matrices and put result into new matrix.

```
>> X=[3 2 1;6 1 7;0 1 0;6 -4 2]
X =
     3     2     1
     6     1     7
     0     1     0
     6    -4     2
>> Y=[5 2 2;-5 2 0;3 -3 2;0 1 0]
Y =
     5     2     2
    -5     2     0
     3    -3     2
     0     1     0
>> Z=X+Y
Z =
     8     4     3
     1     3     7
     3    -2     2
     6    -3     2
```

Example (18): Subtract (2,5) from the Y matrix of the previous example.

```
>> Y-2.5
ans =
   2.5000   -0.5000   -0.5000
  -7.5000   -0.5000   -2.5000
   0.5000   -5.5000   -0.5000
  -2.5000   -1.5000   -2.5000
```

Example (19): Multiply row vector elements with 5 elements by another 5 row vector elements.

```
>> E=[5 1 2 1 4], F=[3 1 0 2 1]
E =
    5    1    2    1    4
F =
    3    1    0    2    1
>> J=E.*F
J =
   15    1    0    2    4
```

Example (20): Multiply matrix A(2*3) by B(2*3) using the special element-wise operation.

```
>> A=[8 1 2;5 3 4], B=[3 2 0;4 5 1]
A =
    8    1    2
    5    3    4
B =
    3    2    0
    4    5    1
>> C=A.*B
C =
   24    2    0
   20   15    4
```

Example (21): Display number of rows and columns for the previous example matrices.

```
>> size(A)
ans =
    2    3
>> size(B)
ans =
    2    3
>> size(C)
ans =
1    3
```

Example (22): If A and B are two 5 elements row vectors, find the solution of X=B/A

```
>> A=[2 2 2 2 2]
A =
    2    2    2    2    2
>> B=(2:2:10)
B =
    2    4    6    8   10
>> X=B/A
X =
   3.0000
>> X=A/B
X =
   0.2727
>> X=A\B
X =
    1    2    3    4    5
    0    0    0    0    0
    0    0    0    0    0
    0    0    0    0    0
    0    0    0    0    0
```

Example (23): Delete element no.3 from a 5 elements row vector

```
>> V=[4 1 5 6 3]
V =
    4    1    5    6    3
```

```
>> V(3)=[]
V =
    4    1    6    3
```

Example (24): Delete the 2nd column from a 3*4 matrix.

```
>> X=[5 1 7 0;8 2 0 0;4 2 1 6]
X =
    5    1    7    0
    8    2    0    0
    4    2    1    6

>> X(:,2)=[]
X =
    5    7    0
    8    0    0
    4    1    6
```

Example (25): Find the inverse of Matrix (4*4).

```
>> A=[5 2 1 0;7 1 0 -2;6 0 -3 1; 1 0 3 1]
A =
    5    2    1    0
    7    1    0   -2
    6    0   -3    1
    1    0    3    1
>> inv(A)
ans =
   -0.0451    0.0902    0.0827    0.0977
    0.6316   -0.2632   -0.1579   -0.3684
   -0.0376    0.0752   -0.0977    0.2481
    0.1579   -0.3158    0.2105    0.1579
```

Example (26): Prove that the theory of ($A*A^{-1}$) is true.

```
>> B=[2 1 1;1 2 2;2 1 2]
B =
    2    1    1
    1    2    2
```

```
     2     1     2
>> Binv=inv(B)
Binv =
    0.6667   -0.3333         0
    0.6667    0.6667   -1.0000
   -1.0000         0    1.0000
>> V=B*Binv
V =
     1     0     0
     0     1     0
     0     0     1
```

Chapter Four

Introduction to Programming in MATLAB

4.1Introduction

So far in these lab session, all the commands were executed in the command window. The problem is that the commands entered in the Command Window cannot be saved and executed again for several times. Therefore, a different way of executing repeatedly commands with MATLAB is:

1. To create a file with a list of commands.
2. Save the file, and
3. Run the file.

If needed, corrections or changes can be made to the commands in the file. The files that are used for this purpose are called script files or scripts for short.

This section covers the following topics:

- M-File Scripts
- M-File functions

4.2 M-File Scripts

A script file is an external files that contains a sequence of MATLAB statements. Script files have a filename extension .m and are often called M-files. M-files can be scripts that simply execute a series of MATLAB statements, or they can be functions that can accept arguments and can produce one or more outputs.

4.2.1 Examples

Here are two simple scripts.

Example 1

Consider the system of equations:

$$x + 2y + 3z = 1$$

$$3x + 3y + 4z = 1$$

$$2x + 3y + 3z = 2$$

Find the solution x to the system of equations.

Solution:

Use the MATLAB editor to create a file: file ⟶new ⟶M-file.

Enter the following statements in the file:

```
A=[1 2 3; 3 3 4; 2 3 3];
b= [1;1;2];
x=A\b
```

```
>> exa33
x =
   -0.5000
    1.5000
   -0.5000
```

When execution completes, the variables (A, b, and x) remain in the workspace. To see a listing of them, enter **whos** at the command prompt.

Note: The MATLAB editor is both a text editor specialized for creating M-files and a graphical MATLAB debugger. The MATLAB editor has numerous menus for tasks such as saving, viewing, and debugging. Because it performs some simple checks and also uses color to differentiate between various elements of codes, this text editor is recommended as the tool of choice for writing and editing M-files. There is another way to open the editor:

>>edit

or

>>edit filename.m

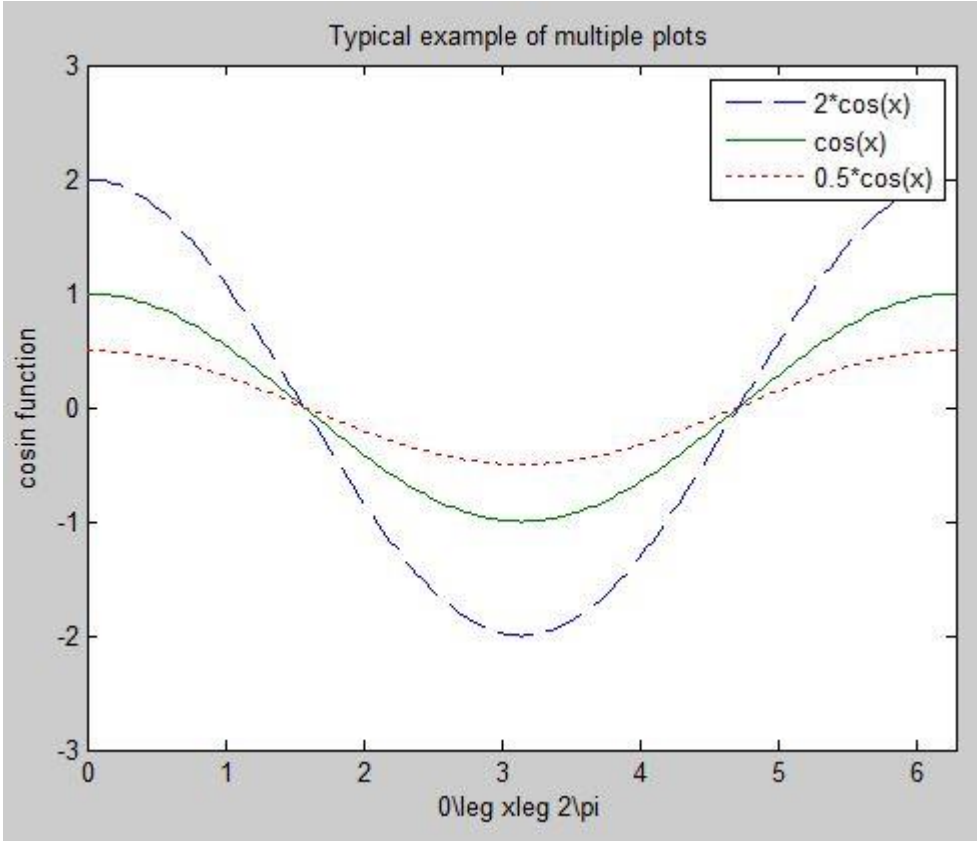
to open filename.m.

Example 2

Plot the following cosine functions, y1 = 2 cos(x), y2 = cos(x), and y3=0.5 * cos (x), in the interval $0 \leq x \leq 2\pi$. This example has been presented in previous chapter. Here we put the commands in a file.

Create a file, say example2.m, which contains the following commands:

```
x=0:pi/100:2*pi;
y1=2*cos(x);
y2=cos(x);
y3=0.5*cos(x);
plot(x,y1,'--',x,y2,'-',x,y3,':'), xlabel('0\leg xleg
2\pi'),ylabel('cosin function')
legend('2*cos(x)','cos(x)','0.5*cos(x)')
title('Typical example of multiple plots')
axis([0 2*pi -3 3])
```



Run the file by typing example2 in the Command Window.

4.2.2 Script side-effects

All variables created in a script file are added to the workspace. This may have undesirable effects, because:

Variables already existing in the workspace may be overwritten.

The execution of the script can be affected by the state variables in the workspace.

As a result, because scripts have some undesirable side-effects, it is better to code any complicated applications using rather function M-file.

4.3 M-File Functions

As mentioned earlier, functions are programs (or routines) that accept input arguments and return output arguments. Each M-file function (or function or M-file for short) has its own area of workspace, separated from the MATLAB base workspace.

4.3.1 Anatomy of M-file Function

This simple function shows the basic parts of an M-file.

```
Function f= factorial (n) (1)
Function f= factorial (n)                    (1)
% FACTORIAL (N) returns the factorial of N.  (2)
% compute a factorial value.                 (3)
F= prod (1:n);                               (4)
```

The first line of a function M-file starts with the keyword function. It gives the function name and order of arguments. In the case of function factorial, there are up to one output argument and one input argument. Table 4.1 summarizes the M-file function.

As an example, for n=5, the result is,

```
>> f=factorial(5)
f =
   120
```

Both functions and scripts can have all of these parts, except for the function definition line which applies to function only.

Table 4.1: Anatomy of a M-file functions

Part no.	M-file element	Description
(1)	Function definition line	Define the function name, and the number and order of input and output arguments
(2)	H1 line	One line summary description of the program, displayed when you request Help
(3)	Help text	A more detailed description of the program
(4)	Function body	Program code that performs the actual computations

In addition, it is important to note that function name must be no longer than the maximum of 63 characters. Furthermore, the name of the text file that you save will consist of the function name with the extension .m. thus, the above example file would be factorial.m

Table 4.2 summarizes the differences between scripts and functions.

Table 4.2 Differences between scripts and functions.

SCRIPTS	FUNCTIONS
- Do not accept input arguments or return output arguments	- Can accept input arguments and return output arguments
- Store variables in a workspace that is shared with other scripts	- Store variables in a workspace internal to the function
- Are useful for automating a series of commands	- Are useful for extending the MATLAB language for your application.

4.3.2 Input and Output Arguments

As mentioned above, the input arguments are listed inside parenthesis following the function name. The output arguments are listed inside the brackets on the left side. They are used to transfer the output from the function file. The general form looks like this

Function [outputs] = function_name(inputs)

Function file can have none, one, or several output arguments. Table 4.3 illustrate some possible combinations of input and output arguments.

Table 4.3 Example of input and output arguments.

Function C=FtoC(F)	one input argument and one output argument
Function area =TrapArea(a,b,n)	Three inputs and one output
Function [n,d]=motion (v,angle)	Two inputs and two outputs

4.4 Input to a Script File

When a script file is executed, the variables that are used in the calculations within the file must have assigned values. The assignments of a value to a variable can be done in three ways.

1. The variable is defined in the script file.
2. The variable is defined in the command prompt.
3. The variable is entered when the script is executed.
   We have already seen the two first cases. Here, we will focus our attention on the third one. In this case, the variable is defined in the script file. When the file is executed, the user is prompted to assign a value to the variable in the command prompt. This is done by using the input command. Here is an example.

```
% This script file calculates the average of points.
% scored in three games
% The point from each game are assigned to a variable
% by using the 'input' command

Game 1=input ('Enter the points scored in the first game');
Game 2=input ('Enter the points scored in the second game');
Game 3=input ('Enter the points scored in the third game');
Average = (game1+game2+game3)/3
```

The following shows the second prompt when this script file (saved as example3) is executed.

```
>>example3
>> Enter the points scored in the first game 15
>> Enter the points scored in the second game 23
>> Enter the points scored in the third game 10
Average =
16
```

4.5 Output Commands

A discussed before, MATLAB automatically generates a display when commands are executed. In addition to this automatic display, MATLAB has several commands that can be used to generate displays or outputs.

Tow commands that are frequently used to generate output are ***disp*** and ***fprintf***.

The main differences between these two commands can be summarized as follows.

Table 4.4: *disp* and *fprintf* commands

disp	Simple to use Provide limited control over the appearance of output
fprintf	Slightly more complicated than *disp* Provide total control over the appearance of output

Chapter Five

Control Flow and Operators

5.1 Introduction

MATLAB is also a programming language. Like other computer programming languages, MATLAB has some decision making structures for control of command execution. These decision making or control flow structures include for loops, while loops, and if-else-end constructions. Control flow structures are often used in script M-files and function M-files. By creating a file with the extension. M, we can easily write and run programs. We do not need to compile the program since MATLAB is an interpretative (not compiled) language. MATLAB has thousands of functions, and you can add your own using m-files. MATLAB provides several tools that can be used to control the flow of a program (script or function). In a simple program as shown in the previous chapter, the commands are executed one after the other. Here we introduce the flow control structure that make possible to skip commands or to execute specific group of commands.

5.2 Control Flow

MATLAB has four control flow structures: the if statement, the for loop, the while loop, and the switch statement.

5.2.1 The "if …end" structure

MATLAB supports the variants of "if" construct.

if….end

if...else…end

if…elseif…else…end

The simplest form of the if statement is

if. expression

statement

```
end
```

here are some examples based on the familiar quadratic formula.

1-
```
discr = b*b - 4*a*c;
if discr < 0
disp('Warning: discriminant is negative, roots are imaginary');
end
```

2-
```
discr = b*b - 4*a*c;
if discr < 0
disp('Warning: discriminant is negative, roots are imaginary');
else
disp('Roots are real, but may be repeated')
end
```

3-
```
discr = b*b - 4*a*c;
if discr < 0
disp('Warning: discriminant is negative, roots are imaginary');
elseif discr == 0
disp('Discriminant is zero, roots are repeated')
else
disp('Roots are real')
end
```

It should be noted that:

1. elseif has no space between else and if (one word).
2. no semicolon (;) is needed at the end of lines containing if, else, end
3. Indentation of if block is not required, but facilitate the reading.
4. The end statement is required.

5.2.2 Relational and logical operators

A relational operator compares two numbers by determining whether a comparison is true or false. Relational operators are shown in Table 5.1.

Table 5.1: Relational and logical operators

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
»=	Not equal to
&	AND operator
j	OR operator
»	NOT operator

Note that the "equal to" relational operator consists of two equal signs (==) (with no space between them), since = is reserved for the assignment operator.

5.2.3 The "for...end'' loop

In the for ... end loop, the execution of a command is repeated at a fixed and predeter-

mined number of times. The syntax is

for variable = expression

statements

end

Usually, expression is a vector of the form i:s:j. A simple example of for loop is

for ii=1:5

x=ii*ii

end

It is a good idea to indent the loops for readability, especially when they are nested. Note that MATLAB editor does it automatically.

Multiple for loops can be nested, in which case indentation helps to improve the readability. The following statements form the 5-by-5 symmetric matrix A with (*i; j*) element

i/j for j > i:

```
n = 5; A = eye(n);
for j=2:n
      for i=1:j-1
    A(i,j)=i/j;
   A(j,i)=i/j;
  end
end
```

This loop is used when the number of *passes* is not specified. The looping continues until a
stated condition is satisfied. The while loop has the form:

```
while expression
     statements
end
```

The statements are executed as long as expression is true.

```
x = 1
while x <= 10
   x = 3*x
end
```

It is important to note that if the condition inside the looping is not well defined, the looping will continue *indefinitely*. If this happens, we can stop the execution by pressing Ctrl-C.

5.2.5 Other flow structures

The break statement. A while loop can be terminated with the break statement, which passes control to the first statement after the corresponding end. The break statement can also be used to exit a for loop.
The continue statement can also be used to exit a for loop to pass immediately to the next iteration of the loop, skipping the remaining statements in the loop. Other control statements include return, continue, switch, etc. For more detail about these commands, consul MATLAB documentation.

5.2.6 Operator precedence

We can build expressions that use any combination of arithmetic, relational, and logical operators. Precedence rules determine the order in which MATLAB evaluates an expression. We have already seen this in the \Tutorial Lessons". Here we add other operators in the list. The precedence rules for MATLAB are shown in this list (Table 5.2), ordered from highest (1) to lowest (9) precedence level. Operators are evaluated from left to right.

Table 5.2: Operator precedence

Precedence	Operator
1	Parentheses ()
2	Transpose (:´), power (.^), matrix power (^)
3	Unary plus (+), unary minus (-), logical negation (~)
4	Multiplication (. *), right division (./), left division (.\),matrix multiplication (*), matrix right division (/),matrix left division (\)
5	Addition (+), subtraction (*i*)
6	Colon operator (:)
7	Less than (<), less than or equal to (≤), greater (>),greater than or equal to (≥), equal to (==), not equal to (~=)
8	Element-wise AND, (&)
9	Element-wise OR, (*i*)

5.3 Saving output to a file

In addition to displaying output on the screen, the command fprintf can be used for writing the output to a File. The saved data can subsequently be used by MATLAB or other software. To save the results of some computation to a file in a text format requires the following steps:

1. Open a file using fopen

2. Write the output using fprintf

3. Close the file using fclose

Here is an example (script) of its use.

```
% write some variable length strings to a file
op = fopen('weekdays.txt','wt');
fprintf(op,'Sunday\nMonday\nTuesday\nWednesday\n');
fprintf(op,'Thursday\nFriday\nSaturday\n');
fclose(op);
```

This file (weekdays.txt) can be opened with any program that can read .txt file.

Solved Examples:

Example(1): Given a vector with 10 element, check these elements for numbers grater than or equal 5 then change the element to be 100, if the element value is less than 5 then turn this element to (0), otherwise changed the element to -100?

```
a=[1 4 5 1 7 9 2 0 3 8]
for i=1:10
   ifa(i)<5
      k(i)=0;
 elseif a(i)>=5
   k(i)=100;
 else
  k(i)=-100
  end
end
a
k
```

```
a=
1 4 5 1 7 9 2 0 3 8
K=
0 0 100 0 100 100 0 0 0 100
```

Example (2): Write Matlab code to find number between 1 and 3 with incrementing value of 0

```
clc
for i= 1:0.5:3
end
```

Example (3): calculate the square root of the summation of odd numbers between (1-50)?

```
Sum=0;
for i=1:2:50
sum=sum+i;
end
sum
```

Example (4):Given a 3 by 3 matrix, write necessary code to change non diagonal elements to zero?

```
clc
M=[5 1 2;3 1 5;6 2 1]
for i=1:3
  for i=1:3
      if i~=j     if i not equal j
         M(i,j)=0;  put 0 in non-diagonal elements
       end
  end
end
M
```

Chapter 6

Debugging M-Files

6.1 Introduction

This section introduces general techniques for finding errors in M-files. Debugging is the process by which you isolate and fix errors in your program or code.

Debugging helps to correct two kinds of errors:

- Syntax errors - For example omitting a parenthesis or misspelling a function name.

- Run-time errors - Run-time errors are usually apparent and difficult to track down.

They produce unexpected results.

6.2 Debugging process

We can debug the M-files using the Editor/Debugger as well as using debugging functions from the Command Window. The debugging process consists of

-Preparing for debugging

- Setting breakpoints

-Running an M-le with breakpoints

-Stepping through an M-¯le

- Examining values

- Correcting problems

- Ending debugging

6.2.1 Preparing for debugging

Here we use the Editor/Debugger for debugging. Do the following to prepare for debugging:

- Open the file

- Save changes

- Be sure the file you run and any files it calls are in the directories that are on the search path.

6.2.2 Setting breakpoints

Set breakpoints *to pause* execution of the function, so we can examine where the problem might be. There are three basic types of breakpoints:
- A standard breakpoint, which stops at a specified line.
- A conditional breakpoint, which stops at a specified line and under specified conditions.
- An error breakpoint that stops when it produces the specified type of warning, error,
NaN, or infinite value.

You cannot set breakpoints while MATLAB is busy, for example, running an M-file.

6.2.3 Running with breakpoints

After setting breakpoints, run the M-file from the Editor/Debugger or from the Command Window. Running the M-file results in the following:
The prompt in the Command Window changes to
K>>
indicating that MATLAB is in debug mode.
The program pauses at the first breakpoint. This means that line will be executed when you continue. The pause is indicated by the green arrow.
In breakpoint, we can examine variable, step through programs, and run other calling functions.

6.2.4 Examining values

While the program is paused, we can view the value of any variable currently in the workspace. Examine values when we want to see whether a line of code has produced the expected result or not. If the result is as expected, step to the next line, and continue running. If the result is not as expected, then that line, or the previous line, contains an error. When we run a program, the current workspace is shown in the Stack field. Use who or whose to list the variables in the current workspace.

Viewing values as datatips

First, we position the cursor to the left of a variable on that line. Its current value appears. This is called a datatip, which is like a tooltip for data. If you have trouble getting the datatip to appear, click in the line and then move the cursor next to the variable.

6.2.5 Correcting and ending debugging

While debugging, we can change the value of a variable to see if the new value produces expected results. While the program is paused, assign a new value to the variable in the Command Window, Workspace browser, or Array Editor. Then continue running and stepping through the program.

6.2.6 Ending debugging

After identifying a problem, end the debugging session. It is best to quit debug mode before editing an M-file. Otherwise, you can get unexpected results when you run the file. To end debugging, select Exit Debug Mode from the Debug menu.

6.2.7 Correcting an M-file

To correct errors in an M-file,

- Quit debugging

- Do not make changes to an M-file while MATLAB is in debug mode

-Make changes to the M-file

- Save the M-file

- Clear breakpoints

Chapter Seven

Mathematical Equation in Matlab

7.1 Differential Equation

7.1.1 First order Differential Equation

In Mathematical, 1st order diff. eq. is written as dy/dx or y' …Matlab can solve defined or undefined equations symbolically; Command used to solve such equations is dsolve, while D*y* is used within in order to represent the Derivative function.

Example (1): Find the solution for x in the following first order eq.:xy'-y=1?

```
>> dsolve('x*Dy-y=1','x')

 ans =

C2*x - 1
```

Example (2): Find the solution for x in: y'+y=sin(x)/2

```
>> dsolve('Dy+y=sin(x)/2','x')

ans =

sin(x)/4 - cos(x)/4 + C4/exp(x)
```

Example (3): Find the solution for: y'+y= cos(t)

```
>> dsolve('Dy+y=cos(t)')

ans =

cos(t)/2 + sin(t)/2 + C6/exp(t)
```

Example (4): Find the solution for: y'=x+y where y(0)=c

```
 >> dsolve('Dy=x+y','y(0)=c','x')

ans =

exp(x)*(c + 1) - x – 1
```

7.1.2 Second order Differential Equation

In Mathematical, second order derivative may written as y"or d2y/dx. In Matlab, it's written as D2y(Thus the Higher is the same, for example D12y)…

Example (5): Solve the following second order equation: 3y" -4y'+y=cos(x).

```
>> dsolve('3*D2y-4*Dy+y=cos(x)','x')

ans =

C2*exp(x) - sin(x)/5 - cos(x)/10 + C3*exp(x/3)
```

Similar to 1 st. order differential equations, we may have initial conditions for y and y"

Example (6): Find the solution for the previous example (5), with initial condition values are y(0)=3 and Dy(0)=-3

```
>> dsolve('3*D2y-4*Dy+y=cos(x)','y(0)=3','Dy(0)=-3','x')

ans =

(177*exp(x/3))/20 - cos(x)/10 - (23*exp(x))/4 - sin(x)/5
```

Example (7): Solve the equation 8.d3y/dx3-5.dy/dx+3y=x3,for x, with initial values of (d3y/dx3 (0) =4) and y (1) =-2

(Try to solve it)

7.2 Representing Functions

in order to represent a function in Matlab, inline function is to be used.

Example (1): represent and find the value of function x^2+5x-2,if x=3

```
>> f=inline ('x^2+5*x-2')

f =

    Inline function:

    f(x) = x^2+5*x-2

>> f(3)

ans =

   22
```

We can also represent and find valve of a more than one variable function.

Example (2): represent and find the value of the following two variable function:

g(x,y)= x^3-4x^2+3xy-7,with value x=4,y=6.

```
g=inline('x^3-4*x^2+3*x*y-7')

g =

    Inline function:

    g(x,y) = x^3-4*x^2+3*x*y-7

>> g(4,6)

ans =

   65
```

To define a function as a vector, vectorize is used along with the function in order to give more than one point at the same time to evaluate that function.

Example (3): represent and evaluate the following function at the point 2, 5, and 8 f(x) =3x^2+2x-5.(Hint: in this case vectorize is to be used).

```
>> f=inline(vectorize('3*x^2+2*x-5'))
```

```
f =

    Inline function:

    f(x) = 3.*x.^2+2.*x-5

>> f([2 5 8])

ans =

   11    80   203
```

7.3 Limits

To compute limits using Matlab, limit function is used to evaluate it, when x goes to some value.

Example (1): Find the limit of the function: lim x→3((x^2-5))/(x-1)). (x goes to 3).

```
>> syms x

>> limit((x^2-5)/(x-1),x,3)

ans =

2
```

7.4 Differentiation

To perform differentiation in Matlab, ***diff*** function is used.

Example (1): Find the differentiation of the function: $3x^2$+1

(Here also, either ***syms*** is used to define variable(s) or (‘… ‘) used without ***syms).***

```
>> syms x

>> diff(3*x^2+1)

ans =

6*x
```

Or

```
diff('3*x^2+1')

ans =

6*x
```

Example (2): Solve the diff. of: 2 sin (x^2)/ cos(x).

```
>> diff('2*sin(x)/cos(x)')

ans =

(2*sin(x)^2)/cos(x)^2 + 2
```

The n^{th}. Derivative of (f) is written in the form ***diff(f,n).*** So if n=2 then the second derivative is required and so on…

Example (3): Find the 1 st, 2nd. And 3rd. derivative of the function: $2x^3$-$4x^2$-3x.

```
>> diff('2*x^3-4*x^2+3*x')

ans =

6*x^2 - 8*x + 3

>> diff('2*x^3-4*x^2+3*x',2)

ans =

12*x - 8

>> diff('2*x^3-4*x^2+3*x',3)

ans =

12
```

7.4.1 Partial derivative

Matlab can compute partial derivative of a give a given expression with respect to some variable … in this case this variable has to be specified within the ***diff*** function.

Example (4): Find the partial derivative of the equation *f(x,y)*=x^3-$2y^2$+sin(x)-2cos(y) ,in terms of x, then for y.

```
>> syms x y
```

```
>> diff('x^3-3*y^2+sin(x)-2*cos(y)',x)

ans =

cos(x) + 3*x^2

>> diff('x^3-3*y^2+sin(x)-2*cos(y)',y)

ans =

2*sin(y) - 6*y
```

We can also find the nth. Derivative as we mentioned before for any equation variable

Example (5): Find the third derivation (d^3fx^3) of the equation:

$d^3\ fx^3 = x^3 - 3y^2$+sin(x)-2 cos(y), in terms of x, then for y.

```
>> syms x y

>> diff(x^3-3*y^2+sin(x)-2*cos(y),x,3)

ans =

6 - cos(x)

>> syms x y

>>  diff(x^3-3*y^2+sin(x)-2*cos(y),y,3)

ans =

-2*sin(y)
```

7.5 Integration

Matlab can find both definite integrals. ***Int*** command is used to compute the symbolic *f* for x. i.e. int(*f*) → $\int f(x)dx$, or int(*f*,a,b)= $\int$.

Example (1): Find the definite and indefinite integration (with limits 2,5) for: $\int$.

```
>>  int('x^3+2')

ans =

(x*(x^3 + 8))/4
```

```
>>  int('x^3+2',2,5)

ans =

633/4

>> 633/4

ans =

  158.2500
```

Example (2): Find the definite integration of ∫.

```
>> int('cos(x)/sin(x)')

ans =

log(tan(x)) - log(tan(x)^2 + 1)/2
```

Finally, remember we used **log10** for common base of 10, and **log** 10 represent natural logarithm (ln) before.

But in case of integration we must use (**ln**) for Natural Logarithm instead of log.

Example (3): Find the definite integration with limit (1,3) for ∫.

```
>> int('ln(x)+2',1,3)

ans =

log(27) + 2

>> log(27) + 2

ans =

    5.2958
```

Example (4): Find the indefinite integration of ∫.

```
>> int('2*ln(x)+2*x/(x-3)')

ans =


6*log(x - 3) + 2*x*log(x)
```

Chapter Eight

Graph Plot in Matlab

Computer Graphics (CG) is one of the most important uses in all fields. Graph plotting considered one of implementations of CG.

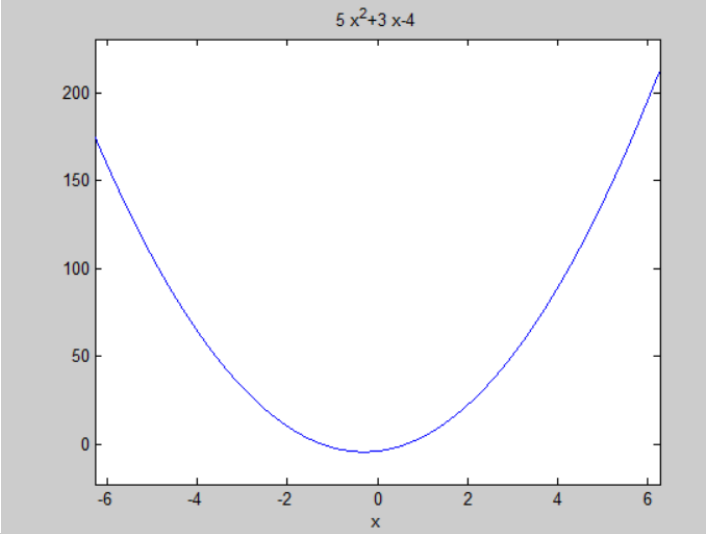
Matlab provides many methods to deal and generate plotted. Graphs representing mathematical and statistical data. When plotting a graph a special Graphic window displayed

8.1 Plotting symbolic functions

In order to plot a function graph, **ezplot** command (easy plot) is used to plot.

Example(1): Plot the function : $y=5x^2+3x-4$

```
>> ezplot('5*x^2+3*x-4')
```



Or, we can plot the same

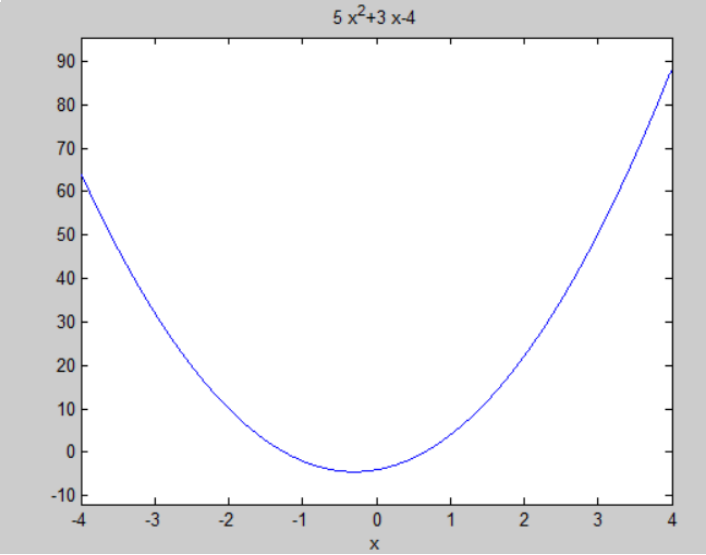
plot with different axis scale

In case we define value to it

For example between (-4 and 4)

To the independent variable.

```
>> ezplot('5*x^2+3*x-4',[-4:4])
```



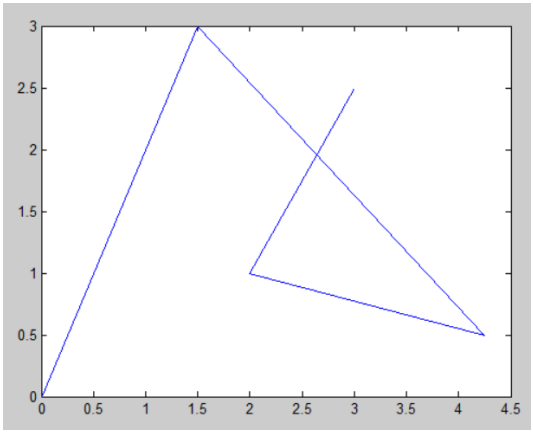
8.2 X-Y plots (vectors)

Using vectors to define Cartesian x, y data pairs with same number of elements in both axis. Let x vector be the x-coordinate with (X_1, X_2,… X_n) elements, and y vector be the y-coordinate with (Y_1, Y_2,… Y_n) elements then the command **plot (x,y)** will generate the graph of these pair of elements.

Example (2): Plot the graph of two dimensional for the following x, y

Coordinate : (0,0),(1.5,3),(4.25,0.5),(2,1),(3,2.5)?

```
>> x=[0 1.5 4.25 2 3];
>> y=[0 3 0.5 1 2.5];
>> plot(x,y)
```



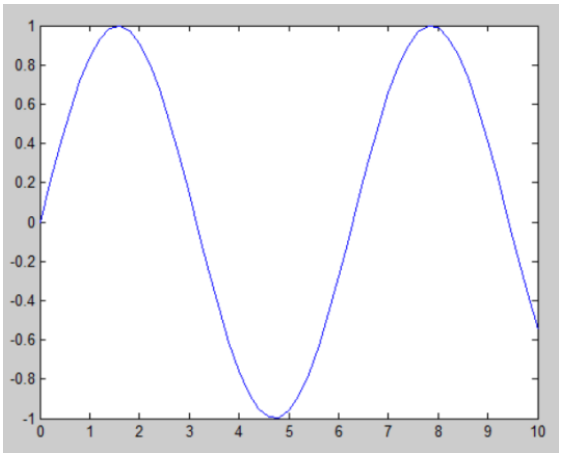
Matlab can plot different functions

Such as sin,cos,log …etc.

Example (3): Plot sin function for x values 0 to 10 with increasing value of 0.2?

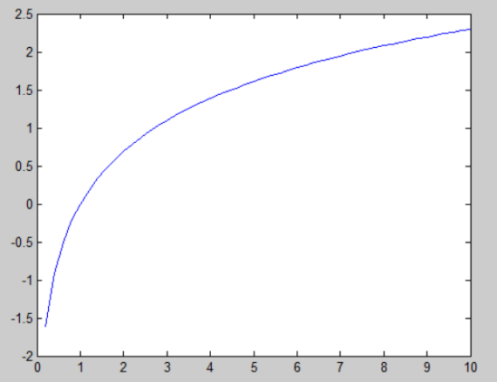
```
>> x=[0:0.2:10];
>> y=sin(x);
 <<plot(x,y)
```

Notice that the x-axis scale according to x vector values, while the depending y-axis scales according to function given.



Example (4): For the same x values in previous example, plot the function:y=log(x)?

```
>> x=[0:0.2:10];
>> y=log(x);
>> plot(x,y)
```



Decorating the graph:

Matlab has the ability to display the output graph in various forms. These decorating can be done either from menu bar of the graph window, or by using items within the plot command, such as:

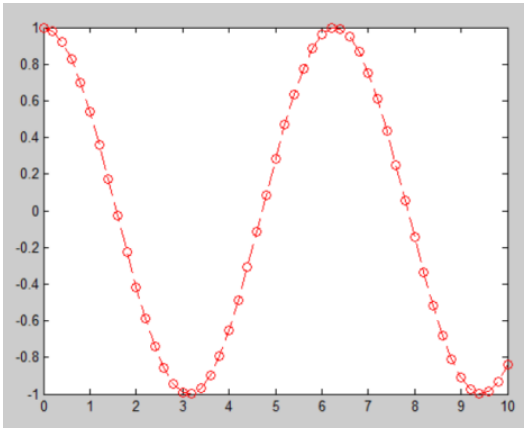
Character colors : K(black), g(green), r(red), b(blue)…

,Character symbol: **.** (Point), o (circle), ***** (star),..etc.

Character line style: -(solid), **:**(doted), **-.**(dashdot), --(dashed) …etc

Example (5): Plot function y=cos(x), with red color , circle and dotted curve?

```
>> y=cos(x);
>> plot (x, y, 'ro--')
```



We can also control the plot axis by defining the minimum and maximum value for x and y axis by using **axis ([x.min x.max y.min y.max]).**

For example >>axis([-4 4 -10 10])

Plots of parametrically defined curves can also be made as in the following example.

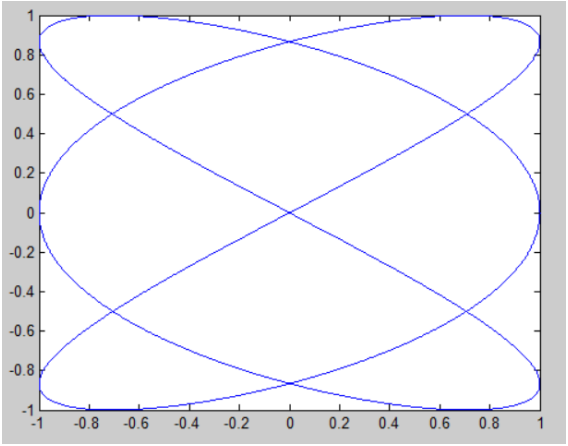
Example (6):

>> t=0:0.001:2*pi;

>> x=cos(3*t);

>> y=sin(2*t);

>> plot(x,y)



To give titles to axis we can either use the plot window menu bar then choose insert:

X label, Y label and Title, or use direct command as in the example below.

Example (6) : Plot the function y=sin(x^2) with x-axis ,y-axis label title Time and Amplitude respectively?

>> y=sin(x.^2);

>> plot(x,y);

>> xlabel('Time')

>> ylabel('Amplitude')

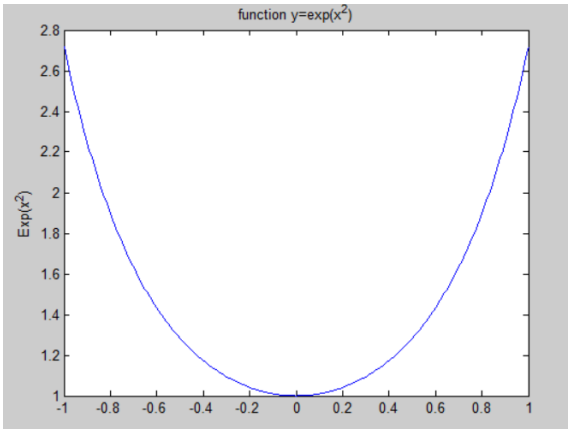
Notice that we used sin(x.^2) to define the function in this case.

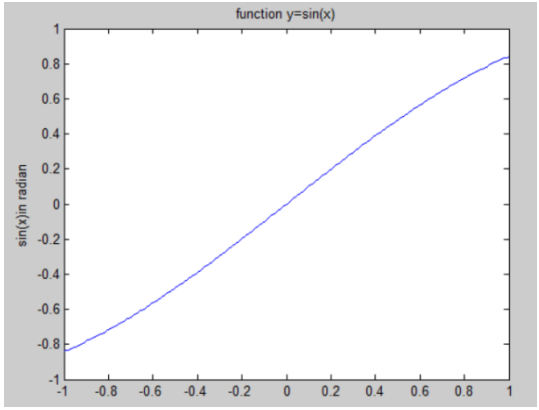
8.3 Multiple plots

Another Matlab feature is to plot more one graph without removing each other by using figure, command. Also more than one graph can be plotted in the same plot window:

Example (8): plot functions y=sin(x) and y=exp(x.^2) and keep both display for x value (-1 to 1 with increasing point 0.01)?

```
>>  x=[-1:0.01:1];
>> y=exp(x.^2);
>> plot(x,y);
>> ylabel('Exp(x^2)')
>> title('function y=exp(x^2)')
>> figure
>> x=[-1:0.01:1];
>> y=sin(x);
>> plot(x,y)
>> ylabel('sin(x)in radian')
>> title('function y=sin(x)')
```

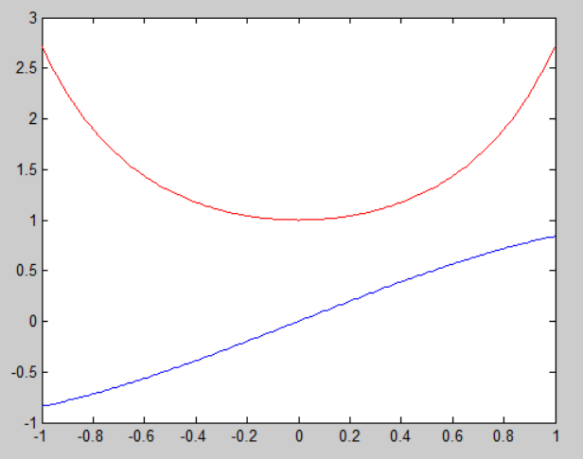




Using figure command enables to display more than one plot at the same time in two graph windows.

Now to plot two or in general multiple plots in one graph , **hold on** command is used. However, the axes may be rescaled according to this as see in the example below:

```
>> X=[-1:0.01:1];
>> Y=exp(X.^2);
>> plot(X,Y,'r')
>> hold on
>> X=[-1:0.01:1];
>> Y=sin(X);
>> plot(X,Y,'b')
```



This situation remains active until switched off by hold off

8.4 Subplot

To split the graph plot window to several or multiple (n * m) array of smaller windows

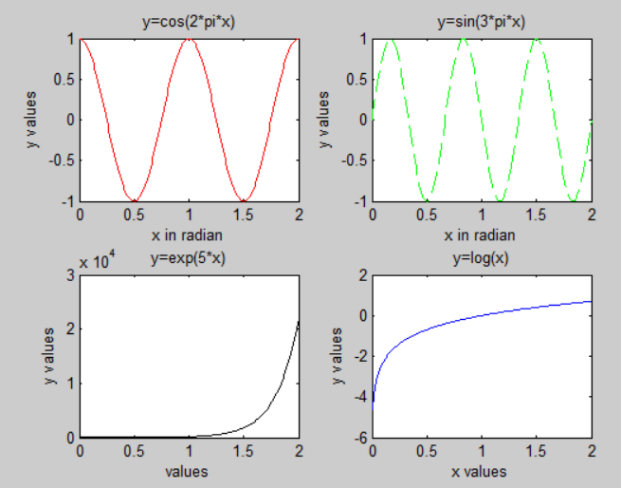
Another Matlab command is used which is **subplot**

The general format is subplot (nmp), where n and m are the number of smaller window plots or the rows and columns, p is the sequence of the plot within the whole graph. Properties such as hold on, gird, color or others will work on each individual subplots.

Example (9): Suppose we want to plot 4 plots (2 *2) in one graph, then the code could be as in below:

```
>>  x=0:0.01:2;
>> subplot (221),plot(x,cos(2*pi*x),'r') Defining subplot(1)
>> xlabel ('x in radian')
>> ylabel ('y values')
>> title ('y=cos(2*pi*x)')
>> subplot (222),plot(x,sin(3*pi*x),'g--') Defining subplot(2)
```

```
>> xlabel ('x in radian')
>> ylabel ('y values')
>> title ('y=sin(3*pi*x)')
>> subplot (223)
>> plot(x,exp (5*x),'ko')
>> subplot (223)
>> plot (x,exp(5*x),'k-')
>> xlabel ('values')
>> ylabel ('y values')
>> title ('y=exp(5*x)')
>> subplot (224)
>> plot (x,log(x),'b')
>> xlabel ('x values')
>> ylabel ('y values')
>> title ('y=log(x)')
```



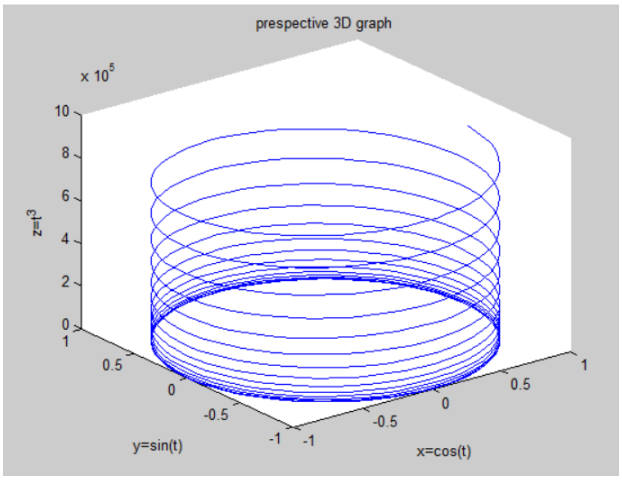
8.5 Three Dimensional Plots

Matlab can produce 3 dim. Plots by simply using command **plot.3** that will turn curves of 2 dim. Into 3 dim.

To plot a 3D graph we need to define the x, y, and z axis in order to make the graph. As mentioned before, the number of elements of each of the three vectors must be the same and are usually defined parametrically

Example (10): Plot a 3D graph if the value of t were (t=0.01:0.1:30*pi), and the three vectors of x =cos(t), y =sin(t), and z= t^3.

```
>> t=0.01:0.1:30*pi;
>>  x=cos(t);
>>  y=sin(t);
>> z=t.^3;
>> plot3(x,y,z)
>>  xlabel('x=cos(t)')
>> ylabel('y=sin(t)')
>>  zlabel('z=t^3')
>> title('prespective 3D graph')
```

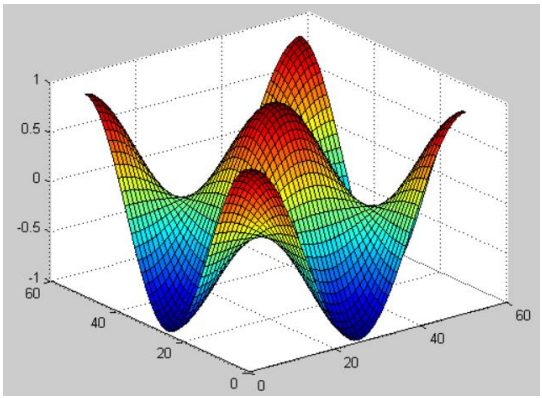


8.6 Surface

We can plot the surface plot of square matrix elements forming also a 3D graph using **surf** command as illustrated in the following example.

Example (11): Given a vector v=[0:0.02:1], plot the surface of y´ *y matrix, when y=cos(v*2*pi)?

```
>> v=0:0.02:1;
>> y=cos(v*2*pi);
>> m=y'*y;
>> surf(m)
```

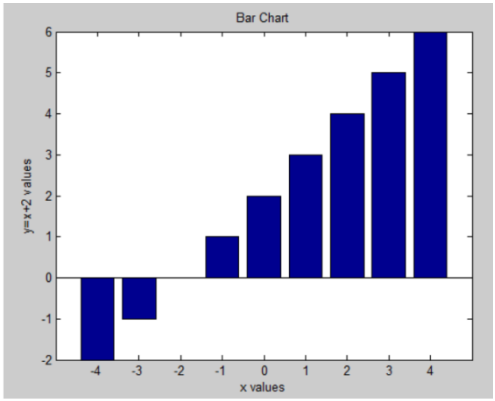


8.7 Bar chart plot

We can also plot bar chart or graph instead of curves using **bar** command as follows:

Let x=-4 to 4 and y= x+2, by using bar command we get.

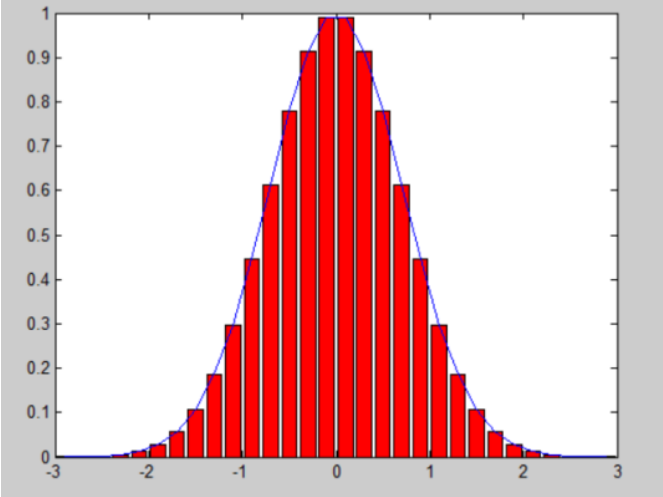
```
>>  x=-4:4;
>>  y=x+2;
>>  bar(x,y)
>>  xlabel('x values')
>> ylabel('y=x+2 values')
>> title('Bar Chart')
```



We may also combine two or more different plotting styles in one graph window:

```
>> x=-2.9:0.2:2.9;
>> bar(x,exp(-x.*x),'r')
>> hold on
>> plot(x,exp(-x.^2))
```

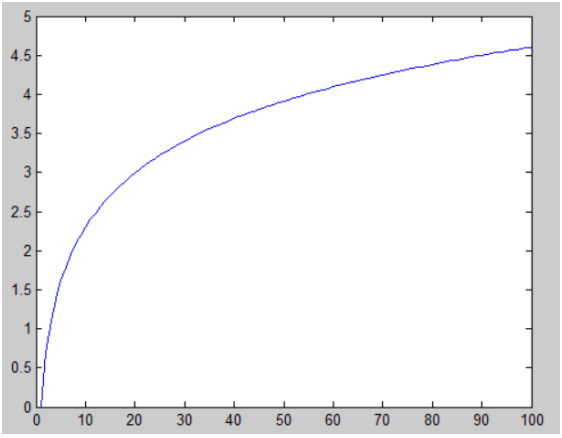
Notice the use of hold on command to enable plotting more than on plot on the same graph window as mentioned before



Simple Problems

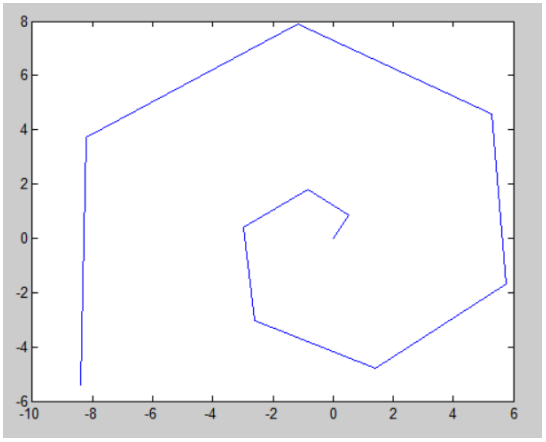
1. Plot the log of the values from 1 to 100?

```
>> x=1:100;
>> y=log(x);
>> plot(x,y)
```



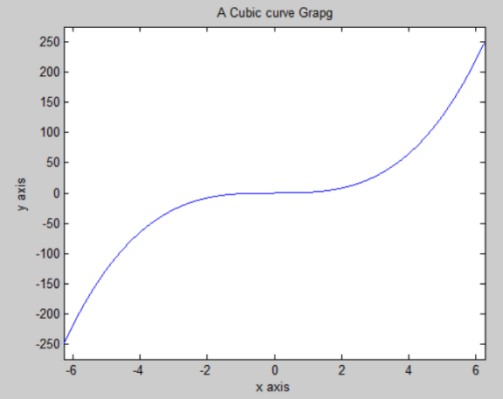
2. plot the parametric **curve** x=t.cos(t), y=t.sin(t) for 0<=t<=10

```
>> t=0:10;
>> x=t.*cos(t);
>> y=t.*sin(t);
>> plot(x,y)
```



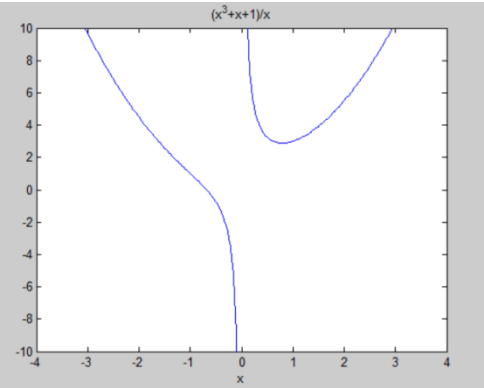
3. plot the cubic curve y=x^3 , label the axis and put a title "A Cubic curve Graph".

```
>> ezplot('x^3')
>> xlabel('x axis')
>> ylabel('y axis')
>> title(' A Cubic curve Grapg')
```



4. plot y=(x^3+x+1)/x,for -4=<x<=4 and -10=<y<=10

```
>> ezplot('(x^3+x+1)/x')
>> axis([-4 4 -10 10])
```



5. plot y=ln(x+1) and y=1-x^2 on the same window for -2<x<6 and -4<y<4

```
>> ezplot('log(x+1)')
>> subplot(211)
>> ezplot('log(x+1)')
>> axis([-2 6 -4 4])
>> subplot(212)
>> ezplot('1-x^2')
>> axis([-2 6 -4 4])
```