MATLAB

MATLAB: is a high-level language and interactive environment that enables you to perform computationally intensive tasks faster than with traditional programming languages such as C, C++, and Fortran.

What is MatLab?

- Matlab is a computer program that combines computation and visualization power that makes it particularly useful for engineers.
- Matlab is an executive program, and a script can be made with a list of Matlab commands like other programming language.
- Matlab stands for MATrix LABoratory.

  The system is designed to make matrix computation particularly easy.

- **The Matlab environment allows the user to:**

  (1) manage variables (2) import and export data (3) perform calculations

  (4) generate plots (5) develop and manage files for use with Matlab

MATLAB Desktop

When you start MATLAB, the MATLAB desktop appears as shown in **Fig (1.1), containing tools (graphical user interfaces) for managing files, variables, and applications associated with MATLAB**.

The first time MATLAB starts, the desktop appears as shown in the following illustration, although your Launch Pad may contain different entries.

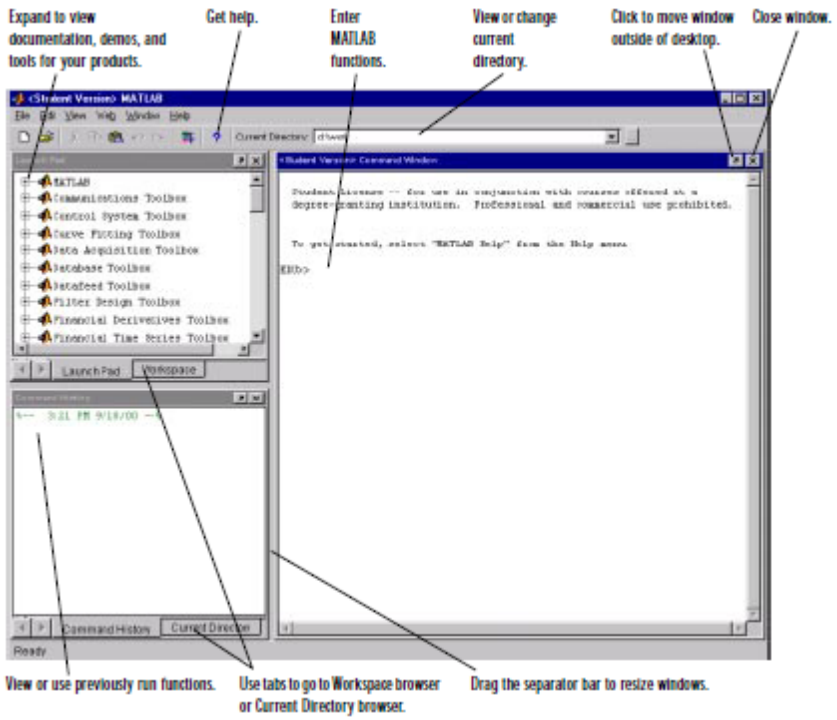


Fig (1.1): MATLAB Desktop

Desktop Tools

This section provides an introduction to MATLAB's desktop tools. You can also use MATLAB functions to perform most of the features found in the desktop tools. **The tools are:**

- **"Command Window"**
- **"Command History"**
- **"Workspace Browser"**
- **"Help Browser"**
- **"Current Directory Browser"**
- **"Editor/Debugger"**

Command Window

Command Window used to enter variables and run functions and M-files, as shown in **Fig (1.2).**

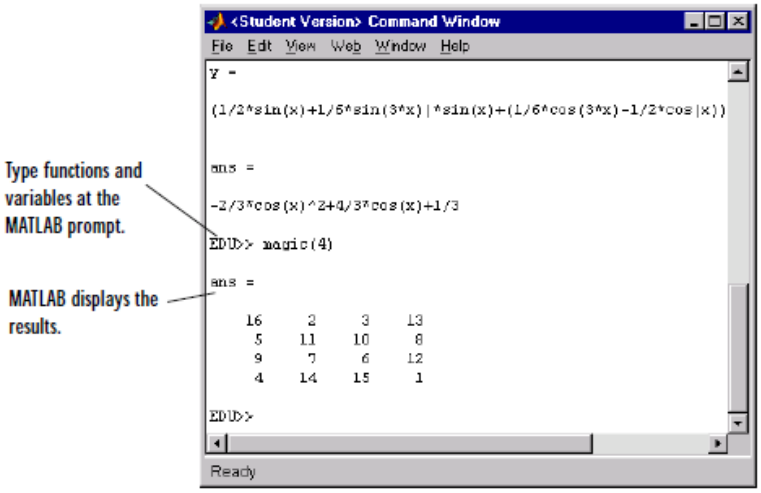


Fig (1.2): Command window page

Command History

Lines you enter in the Command Window are logged in the Command History window. In the Command History as shown in **Fig (1.3)**, **you can view previously used functions, and copy and execute selected lines.**

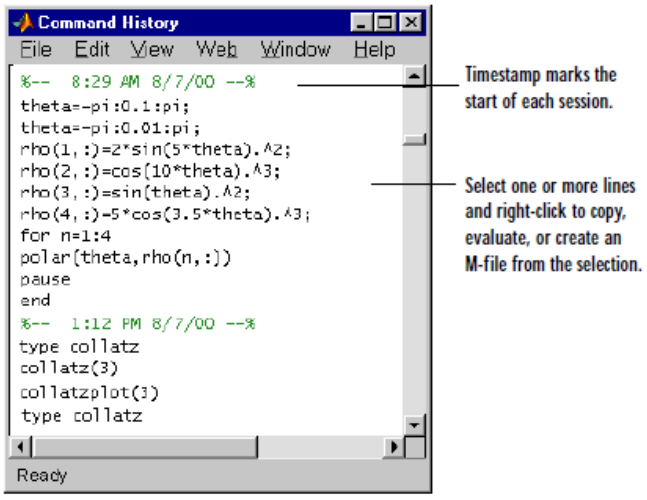


Fig (1.3): Command history window

Workspace Browser

The MATLAB workspace as shown in **Fig (1.4)** **consists of the set of variables (named arrays) built up during a MATLAB session and stored in memory**. You add variables to the workspace by using functions, running M-files, and loading saved workspaces. To view the workspace and information about each variable, use the Workspace browser, or use the functions **who** and **whos**.

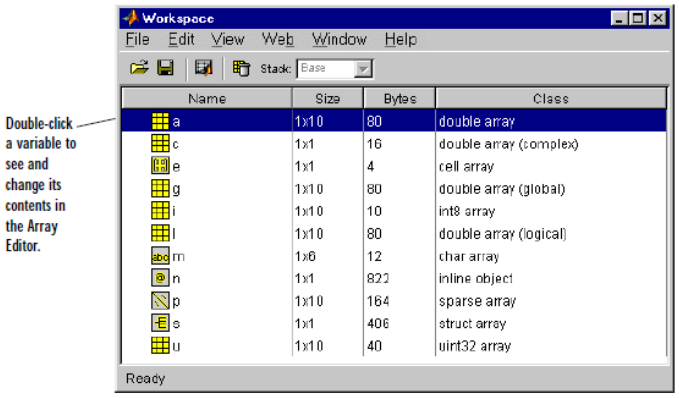


Fig (1.4): Command Window page

Table (1.1) list variable, vector, matrix, and string declaration in MATLAB:

Table (1.1): Variable, Vector, Matrix, and String Declaration

Data	Mathematical Representation	MATLAB Representation
Numerical variable (scalar)	**e.g.** $z=5$	**e.g.** z=5
Row vector	**e.g.** $x=1\ 5\ 7\ 2$	**e.g.** x=[1 5 7 2] **or** x=[1,5,7,2]
Column vector	**e.g.** $y=\begin{matrix}2\\9\\6\end{matrix}$	**e.g.** y=[2;9;6]
matrix	**e.g.** $m=\begin{matrix}1 & 5\\2 & 6\\8 & 1\end{matrix}$	**e.g.** m=[1 5; 2 6; 8 1] **or** m=[1,5;2, 6;8,1]
string	**e.g.** Hello	**e.g.** s='Hello'

Table (1.2) list some MATLAB commands:

Table (1.2): Some MATLAB commands

Command	Description
who	Lists the variables currently in the workspace
whos	Displays a list of the variables currently in the memory and their size together with information about their bytes and class
clc	Clear the Command Window
clear	Removes all variables from the memory (workspace)
clear x,z	Clears/ removes only variables x, and z from the memory (workspace)
help	Helps user to acquire topics, commands, and/or process
Help function	Helps user to acquire a specific topic, command, and/or process
quit	Exit from MATLAB

Notes: to see the relationship between the Command Window, Command History, and Workspace Browser see the following examples:

1- To declare the scalar (numerical variable) as shown in **Fig (1.5)** like **z=5** then in Command Window after >> write z=5 then press enter key and you can see that the value of z becomes 5 and the command z=5 enters to the Command History and in Workspace Window you can see the Name is z and its value 5

>>z=5

z=5

- To check the value of **z** write the command z and press enter key and you can see that the value of z =5 and the command z enters to the Command History

>>z

z=5

- Use command **who** then the message Your variables are: z appeared, also the command who enters to the Command History
- Use command **whos** then the table appeared which include the Name of variable is z and its Size is 1x1 (which means matrix of 1 row and 1 column) and the No. of Bytes which allocated from the memory to the variable z are 8 and the Class of variable z is double by default declaration, also the command whos enters to the Command History

Fig (1.5): Numerical variable (scalar) example

- **To clear the Command Window**, use the command **clc** or **Home→Clear Commands→Command Window** or from the down arrow (Show Command Window Actions) which appeared in the title bar of Command Window select Clear Command Window.
- **To clear the Workspace Browser window**, use the command **clear** or **Home→Clear Workspace** then all the variables are removed from the memory or from the down arrow (Show Workspace Actions) which appeared in the title bar of Workspace Window select Clear Workspace.

- **Use command clear z to remove variable z only** from the memory, while **clear x,z** used **to remove variables x and z** from the memory.
- **To clear the Command History**, use **Home→Clear Commands→Command History** or from the down arrow (Show Command History Actions) which appeared in the title bar of Command History Window select Clear Command History.
- Use **Home→Layout** to show or close any window or return to Default view

2- To declare the **row vector** as shown in **Fig (1.6)**, for example **x=[1 5 7 2]** or **x=[1,5,7,2]** and you can see that x=1 5 7 2 and in Workspace Window you can see the Name is x and its value [1,5,7,2]

```
>> x=[1 5 7 2]
   x =1 5 7 2
```

- To check the value of x write the command **x** and press enter key and you can see that **x =1 5 7 2**

```
>>x
   x =1 5 7 2
```

- Use command **who** then the message Your variables are: x appeared.
- Use command **whos** then the table appeared which include the Name of variable is x and its Size is 1x4 (which means matrix of 1 row and 4 columns) and the No. of Bytes which allocated from the memory to the vector x are 32 (8*4) and the Class of vector x is double by default declaration

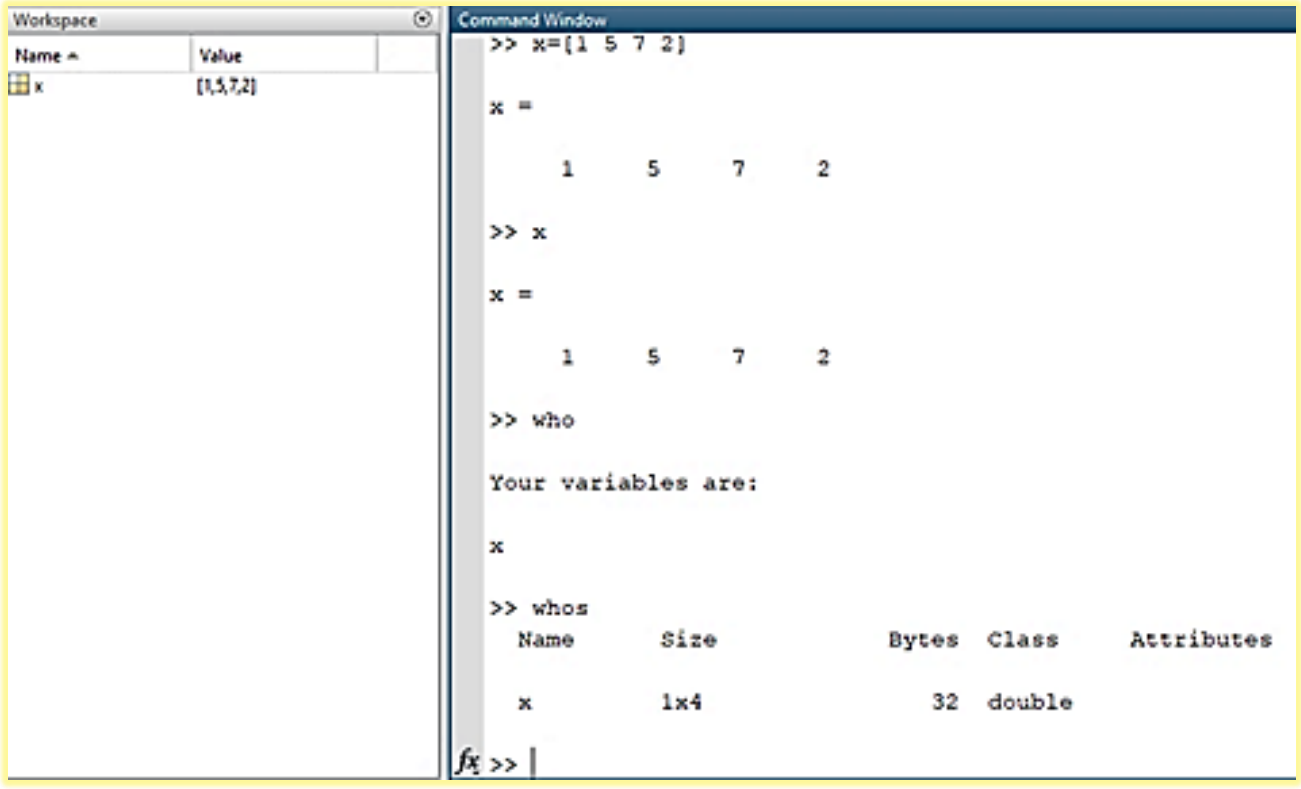


Fig (1.6): Row vector example

3- To declare the column vector as shown in **Fig (1.7)**, for example **y=[2;9;6]** and you can see that

```
y=
     2
     9
     6
```

and in Workspace Window you can see the Name is y and its value [2;9;6]

```
>> y=[2;9;6]

y=
     2
     9
     6
```

- Use command **whos** then the table appeared which include the Name of variable is y and its Size is 3x1 (which means matrix of 3 rows and 1 column) and the No. of Bytes which allocated from the memory to the vector y are 24 (8*3) and the Class of vector y is double by default declaration.

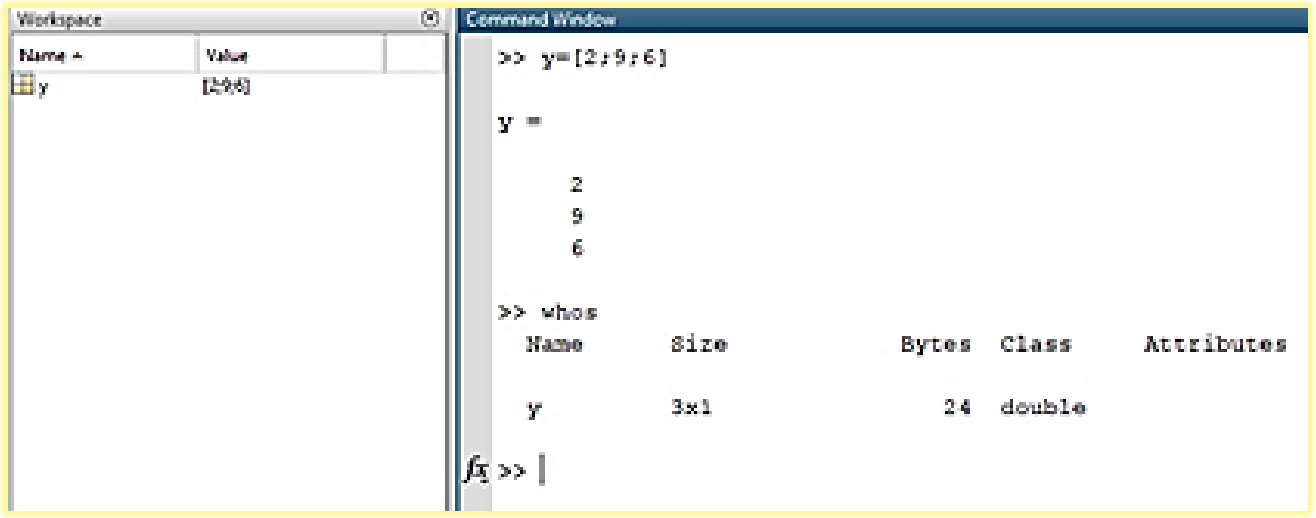


Fig (1.7): Column vector example

4- To define the matrix as shown in **Fig (1.8)**, for example **m=[1 5; 2 6; 8 1]** or **m=[1,5;2, 6;8,1]** and you can see that

```
m=
     1   5
     2   6
     8   1
```

and in Workspace Window you can see the Name is m and its value [1,5;2, 6;8,1]

>> **m=[1 5; 2 6; 8 1]**

```
m=
     1   5
     2   6
     8   1
```

- Use command **whos** then the table appeared which include the Name of variable is m and its Size is 3x2 (which means matrix of 3 rows and 2 columns) and the No. of Bytes which allocated from the memory to the matrix m are 48 (8*6) and the Class of matrix m is double by default declaration

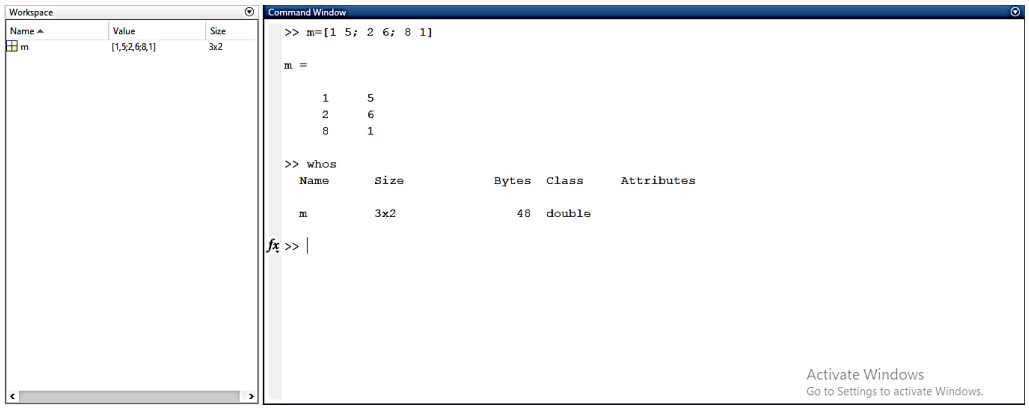


Fig (1.8): Matrix example

5- To declare the string as shown in **Fig (1.9)**, for example s='Hello' and you can see that **s='Hello'** and in Workspace Window you can see the Name is s and its value 'Hello'

- Use command **whos** then the table appeared which include the Name of string is s and its Size is 1x5 (which means matrix of 1 row and 5 columns (characters)) and the No. of Bytes which allocated from the memory to the string s are 10 (2*5) and the Class of string s is char by default declaration

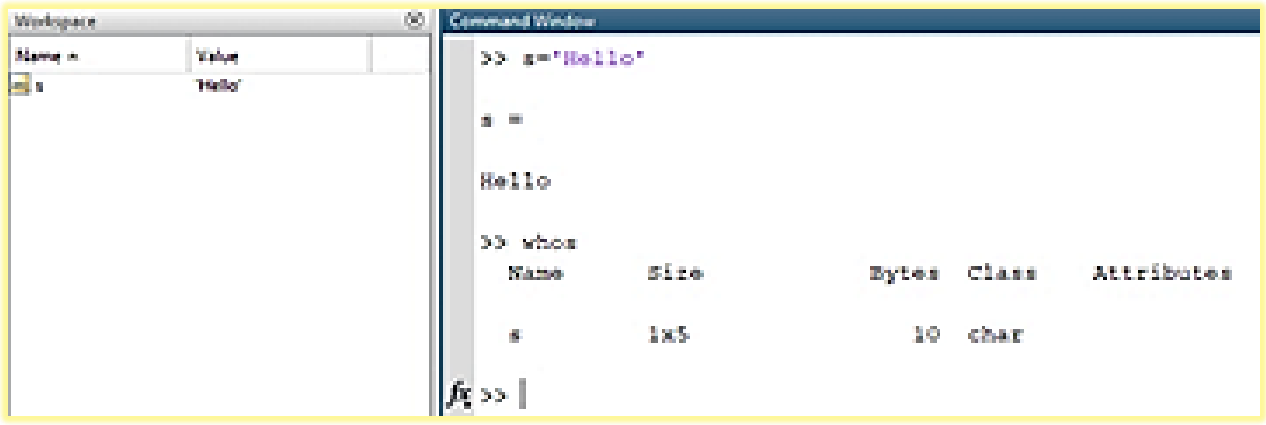


Fig (1.9): String example

Help Browser

Use the Help browser as shown in **Fig (1.10)** **to search and view documentation for all Math Works products**.

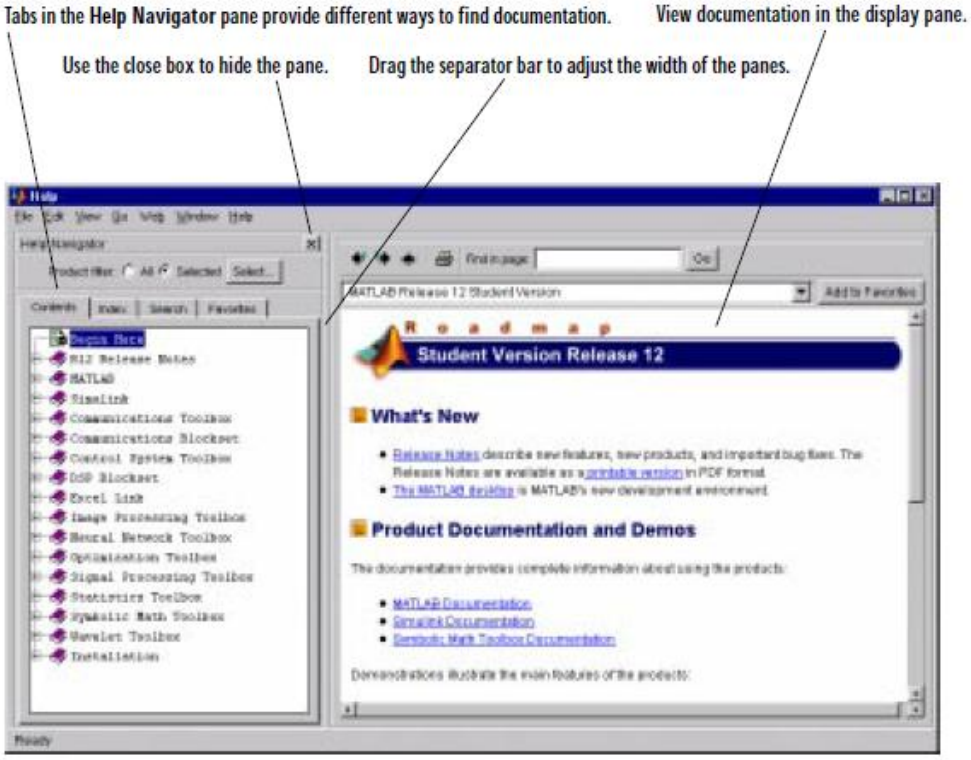


Fig (1.10): Help browser

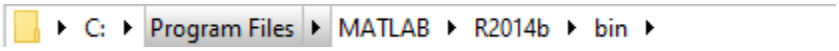
- For example, as shown in **Fig (1.11)**, command help sin to see the help for function sin

```
Command Window
>> help sin
 sin    Sine of argument in radians.
    sin(X) is the sine of the elements of X.

    See also asin, sind.

    Overloaded methods:
       codistributed/sin
       gpuArray/sin
       sym/sin

    Reference page in Help browser
       doc sin

fx >>
```

Fig (1.11): Command help sin

Current Directory Browser

MATLAB file operations use the current directory and the search path as reference points. Any file you want to run must either be in the current directory or on the search path. A quick way to view or change the current directory is by using the **Current Directory** field in the desktop toolbar as shown below.



To search for, view, open, and make changes to MATLAB-related directories and files, use the MATLAB Current Folder browser. As shown in **Fig (1.12)**

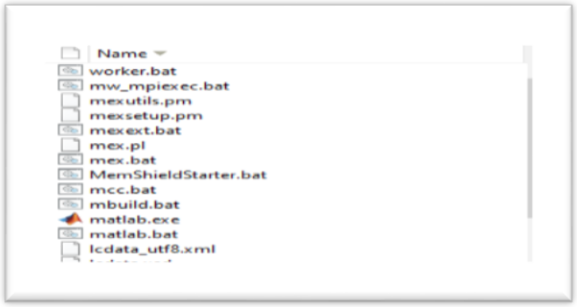


Fig (1.12): Current Folder browser

Editor/Debugger

Use the Editor/Debugger to create and debug M-files as shown in **Fig (1.13)**, **which are programs you write to run MATLAB functions. The Editor/Debugger provides a graphical user interface for basic text editing, as well as for M-file debugging.**

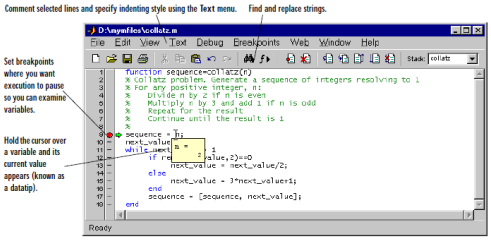


Fig (1.13): Editor/Debugger

- Use **Home→New Script** or **Editor->New** to write m file (program) in matlab then write your program as shown in **Fig (1.14)**.

  x=5

  y=3

  z=x+y

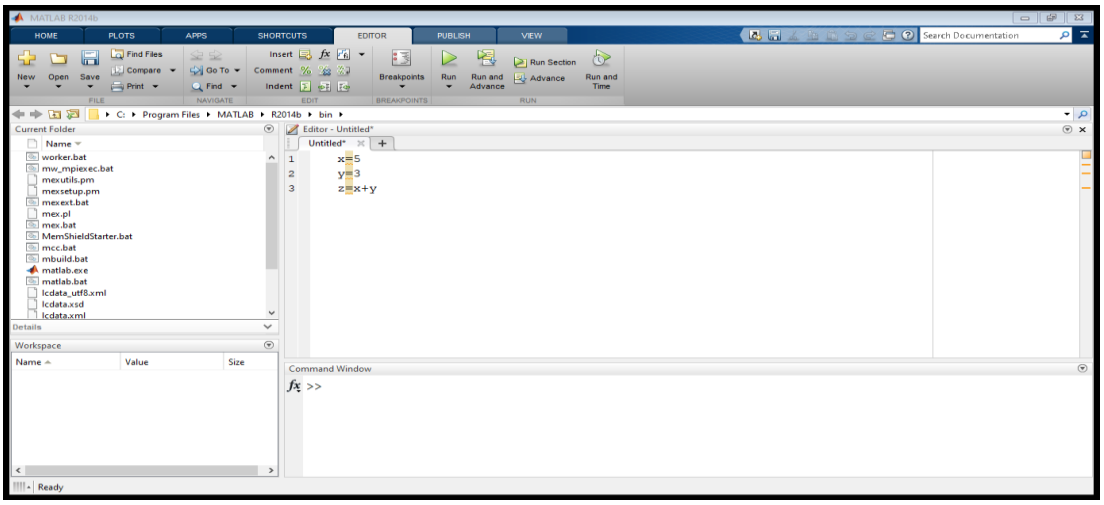


Fig (1.14): M-File Example

- To run your program, use **Editor→Run** then Save File Window appeared to save your file write for example the name of file test in File name: and you can see the type of file is MATLAB Code files **(*.m)** then press save as shown in **Fig (1.15)**

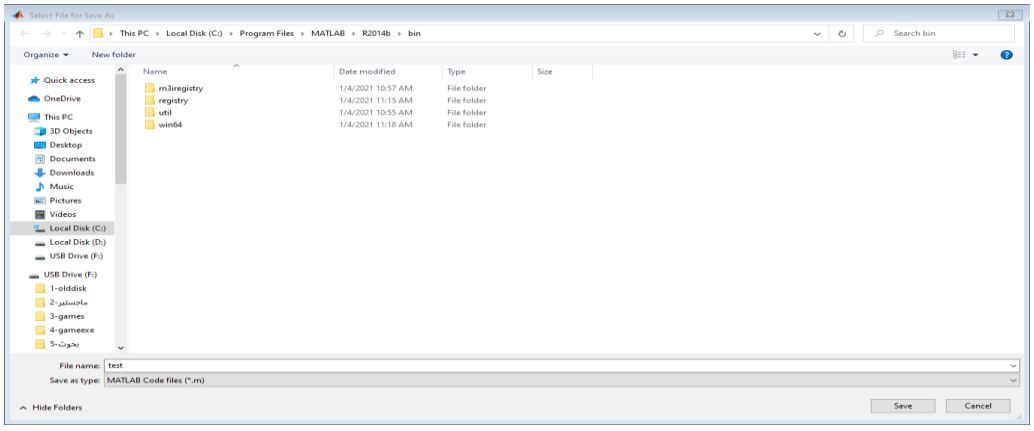


Fig (1.14): Save

Note: If you are in the current folder of matlab then may be the following message appeared if you don't have the permission to save in the current folder of the matlab and ask if you want to save in the document folder instead then choose Yes to save in the document folder or choose No then select another location and press Save again, then another message appeared to add the path and always choose this location then the previous message not appeared again because this path be the current directory or select change path instead to add the path temporary then always do the same steps to run the program. as shown in **Fig (1.15)**

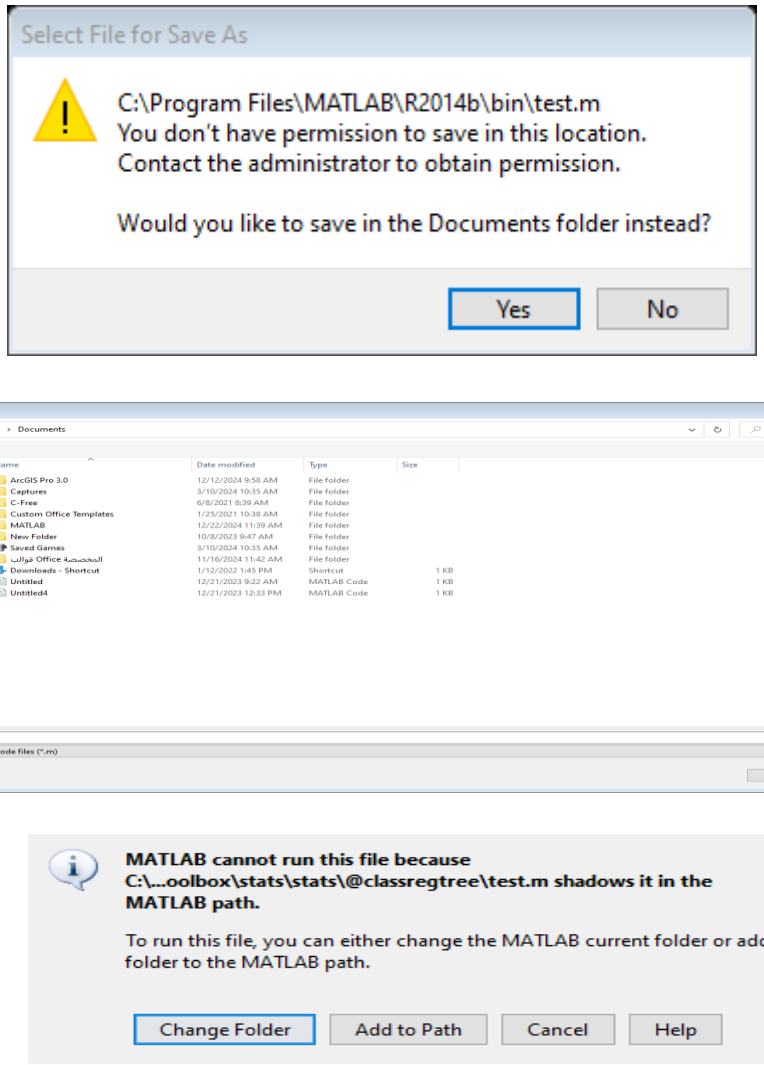


Fig (1.15): Steps to change folder

- In the Command window you can see the output of execution (run) the program test. As shown in **Fig (1.15)**

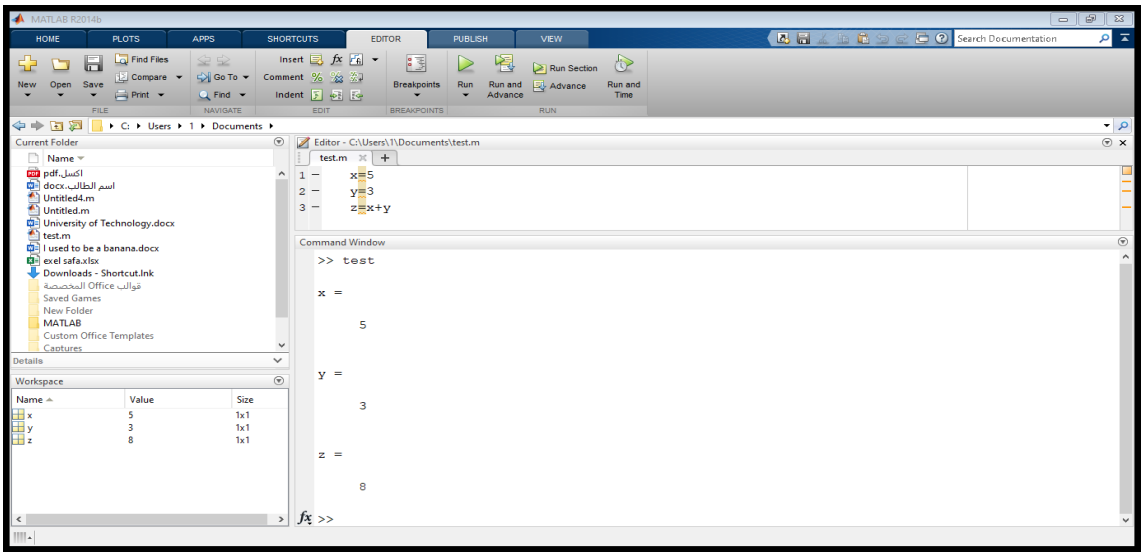


Fig (1.15): Run M-File program

- When you **don't want to see the output of execution any command** then put **;** at the end of this command. As shown in **Fig (1.16)**

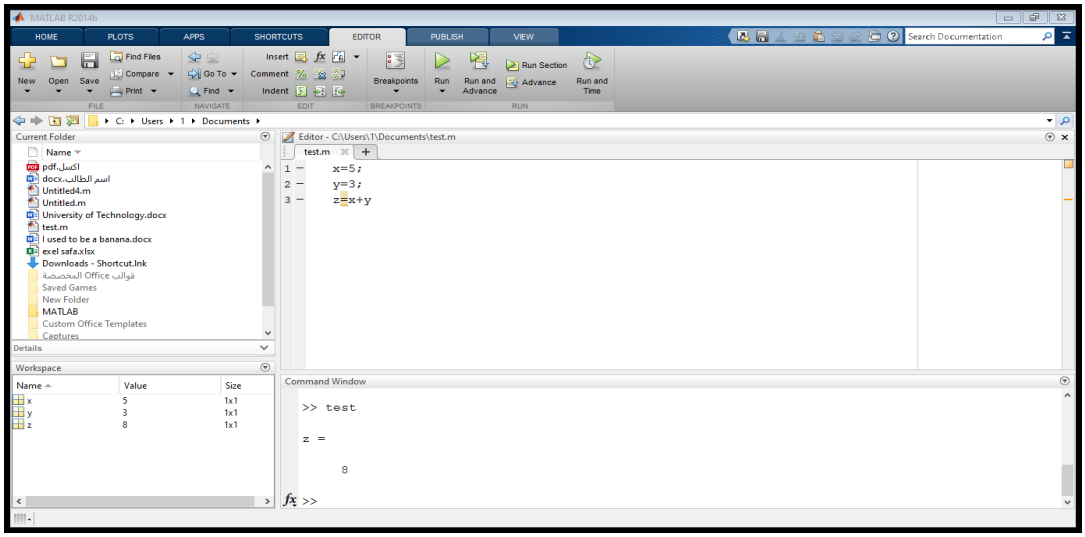


Fig (1.15): ; example

When you want **to write comment** or **to stop the execution of any command** in the program then put **%** at the left of the comment.

Numerical Variable (Scalar)

As a programmer, you will frequently want your program to "remember" a value. For example, if your program requests a value from the user, or if it calculates a value, you will want to remember it somewhere so you can use it later. The way your program remembers things is by using **variables**. Variables represent storage locations in the computer's memory.

Example: X=5

Format:

Use these format types to switch between different output display formats for floating-point variables. Table (2.1) list some MATLAB format commands:

Table (2.1): Format commands

Type	Description
short	Scaled fixed point format, with 4 digits after the decimal point. For example, `3.1416`.
long	Scaled fixed point format with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, `3.141592653589793`.
short e	Floating point format, with 4 digits after the decimal point. For example, `3.1416e+000`.
long e	Floating point format, with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, `3.141592653589793e+000`.
short g	Best of fixed or floating point, with 4 digits after the decimal point. For example, `3.1416`.
long g	Best of fixed or floating point, with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, `3.14159265358979`.
short eng	Engineering format that has 4 digits after the decimal point, and a power that is a multiple of three. For example, `3.1416e+000`.
long eng	Engineering format that has exactly 16 significant digits and a power that is a multiple of three. For example, `3.14159265358979e+000`.

```
>>format short

>>x=3.1415

        x=3.1415

>> format long

>> x

        x = 3.141500000000000

>> format shorte

>> x

        x = 3.1415e+00

>> format longe

>> x

        x = 3.141500000000000e+00

>> format short g

>> x

        x =  3.1415

>> format long g

>> x

        x = 3.1415

>> format short eng

>> x

        x =    3.1415e+000
```

>> format long eng

>> x

x = 3.14150000000000e+000

>> format short eng

>> x=3000000.678

x = 3.0000e+006

>> format long eng

>> x

x = 3.00000067800000e+006

>> x=3000.678

x = 3.00067800000000e+003

Arithmatic operations:

Table (2.2) list Arithmatic operations

Table (2.2): Arithmatic operations

Notation	Math. Representation	MATLAB Representation
Addition	**e.g.** $z = x + y$	**e.g.** >> $z = x + y$
Subtraction	**e.g.** $z = x - y$	**e.g.** >> $z = x - y$
Multiplication	**e.g.** $z = x \times y$	**e.g.** >> $z = x * y$
Division	**e.g.** $z = \frac{X}{Y}$	**e.g.** >> $z = x / y$ **(right division)**
	e.g. $z = \frac{Y}{X}$	**e.g.** >> $z = x \backslash y$ **(left division)**
Raise to the power	**e.g.** $z = x^a$	**e.g.** >> $z = x\text{^}a$
Nth Root	**e.g.** $z = \sqrt[N]{X}$	**e.g.** >> $z = x\text{^}(1/N)$

Examples:

```
>>x=5;
>>y=3;
>>z=x+y
      z=8
>>z=x-y
      z=2
>>z=x*y
      z=15
>>z=x/y
      z=1.6667
>>z=x\y
      z=0.6000
>>z=x^2
      z=25
>>z=x^0.5
      z=2.2316
```

Priorities in MATLAB

The MATLAB software is a mathematical tool. Therefore, it obeys all mathematical roles and condition. Regardless other roles, the priorities in MATLAB are come up to brackets first ( ), raising to the power ^, division, multiplication, and finally to addition and subtraction.

Example: write a program to find $z = \frac{x+x^2}{2xy}$ where x=5 and y=10

```
>> x = 5;
>> y = 10;
>> z = (x + x^2)/ (2 * x * y)
      z=0.3000
```

Built in Functions in MATLAB

`cos(x)`	Cosine	`abs(x)`	Absolute value
`sin(x)`	Sine	`ceil(x)`	Round towards $+\infty$
`tan(x)`	Tangent	`floor(x)`	Round towards $-\infty$
`acos(x)`	Arc cosine	`round(x)`	Round to nearest integer
`asin(x)`	Arc sine	`rem(x)`	Remainder after division
`atan(x)`	Arc tangent		
`exp(x)`	Exponential		
`sqrt(x)`	Square root		
`log(x)`	Natural logarithm		
`log10(x)`	Common logarithm		

>>x=90;

>>sin(x*3.14/180) % angle in radian

ans=1.0000

>>asin(1)*180/3.14

ans=90.0456

>> sind(x) % angle in degree

ans = 1

>> asind(1)

ans = 90

>>x=25;

>>exp(x)

ans= 7.2005e+10

>>sqrt(x)

ans=5

>>x=100;

>>log(x)

ans= 4.6052

>>log10(x)

ans=2

>>x=-25;

>>abs(x)

ans=25

>> x=6.7

x = 6.7000

>> ceil(x)

ans =7

>> floor(x)

ans = 6

>> round(x)

ans = 7

>> x=6.2

x = 6.2000

>> ceil(x)

ans =7

>> floor(x)

ans = 6

>> round(x)

ans = 6

>>x=-6.7;

>>ceil(x)

ans=-6

>>floor(x)

ans=-7

>>round(x)

ans=-7

>> x=5.234;

>> fix(x) %rounds the elements of X to the nearest integers towards zero.

ans = 5

>> x=5.834;

>> fix(x)

ans = 5

>> x=-5.834;

>> fix(x)

ans = -5

>> rem(5,2)

ans = 1

>> mod(5,2) % is the same as rem

ans = 1

Example : when x=20, y=25 find $z = \left|\sqrt[2]{y} - x\right|$

>>x=20;

>>y=25;

>>z=abs(sqrt(y)-x)

z=15

Example: transformation of mathematical equation to code when x=10, z=3, Q=90

$$y = \frac{\frac{x+z}{e^x} + \frac{\sin Q}{tan\ Q}}{\log x + \sqrt{\frac{x+2x}{3}}}$$

```
>>x=10;
>>z=3;
>>Q=90;
>>y=(   (x+z)/exp(x)  +sind(Q)/tand(Q) ) /  (  log(x)+sqrt((x+2*x)/3)    )
```

Example: transformation of mathematical equation to code when x=3, z=5, y=4, Q=3.1415

$$m = \frac{\frac{\cos Q + \sin Q}{e^x} + \frac{|x-z|}{y}}{\sqrt{z^2 + y^2 - x}}$$

```
>>x=3;
>>z=5;
>>y=4;
>>Q=3.1415;
>>m=(  (cos(Q)+sin(Q) ) /exp(x)+abs(x-z)/y) / sqrt(z^2+y^2-x)
```

Relational Operators

When generating code for the Embedded MATLAB Function block, the coder supports the relational operators (and their M-function equivalents) listed in **Table (2.3)**

Table (2.3): Relational operators

Relation	Operator Syntax	M Function Equivalent	Fixed-Point Support?
Less than	A<B	lt(A,B)	Y
Less than or equal to	A<=B	le(A,B)	Y
Greater than	A>B	gt(A,B)	Y
Greater than or equal to	A>=B	ge(A,B)	Y
Equal	A==B	eq(A,B)	Y
Not equal	A~=B	ne(A,B)	Y

Note: When the condition satisfied the answer becomes 1 otherwise 0

Examples:

```
>>a=6;

>>b=4;
>>a<b
        ans=0
>>lt(a,b)
        ans=0
>>a<=b
        ans=0
```

>>le(a,b)

ans=0

>>a>b

ans=1

>>gt(a,b)

ans=1

>>a>=b

ans=1

>>ge(a,b)

ans=1

>>a==b

ans=0

>>eq(a,b)

ans=0

>>a~=b

ans=1

>>ne(a,b)

ans=1

Data Types Available in MATLAB

MATLAB provides 15 fundamental data types. Every data type stores data that is in the form of a matrix or array. The size of this matrix or array is a minimum of 0-by-0 and this can grow up to a matrix or array of any size.

The following table shows the most commonly used data types in MATLAB –

Sr.No.	Data Type & Description
1	**int8** 8-bit signed integer
2	**uint8** 8-bit unsigned integer
3	**int16** 16-bit signed integer
4	**uint16** 16-bit unsigned integer
5	**int32** 32-bit signed integer
6	**uint32** 32-bit unsigned integer
7	**int64** 64-bit signed integer
8	**uint64** 64-bit unsigned integer
9	**single** single precision numerical data
10	**double** double precision numerical data
11	**logical** logical values of 1 or 0, represent true and false respectively
12	**char** character data (strings are stored as vector of characters)

13 **cell array**
array of indexed cells, each capable of storing an array of a different dimension and data type

14 **structure**
C-like structures, each structure having named fields capable of storing an array of a different dimension and data type

15 **function handle**
pointer to a function

16 **user classes**
objects constructed from a user-defined class

17 **java classes**
objects constructed from a Java class

Example

Create a script file with the following code –

Live Demo

```
str = 'Hello World!'
n = 2345
d = double(n)
un = uint32(789.50)
rn = 5678.92347
c = int32(rn)
```

When the above code is compiled and executed, it produces the following result –

str = Hello World!

n = 2345

d = 2345

un = 790

rn = 5678.9

c = 5679

Median: The middle number; found by ordering all data points and picking out the one in the middle (or if there are two middle numbers, taking the mean of those two numbers).

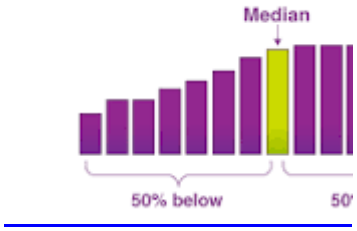
1, 3, 3, **6**, 7, 8, 9

Median = 6

1, 2, 3, **4**, **5**, 6, 8, 9

Median = (4 + 5) ÷ 2

= 4.5



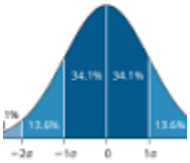
A = [13 22 76 44 90 12 16 13];

12 13 13 16 22 44 76 90

Median=(16+22)/2=38/2=19

Standard deviation

In statistics, the standard deviation is a measure of the amount of variation of the values of a variable about its mean.



Formula

$$\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}}$$

σ = population standard deviation

N = the size of the population

x_i = each value from the population

μ = the population mean

>>A = [13 22 76 44 90 12 16 13]; % A one-dimensional array (row vector)
>>mean(A) % Find the average value of vector A
ans = 35.7500

Vectors

■ A vector is an ordered list of numbers. You can enter a vector of any length in MATLAB by typing a list of numbers, separated by commas and/ or spaces, inside square brackets. For example:

>> **Z = [2,4,6,8]**

Z =

2 4 6 8

>> **Y = [4 -3 5 -2 8 1]**

Y =

4 -3 5 -2 8 1

■ Suppose that you want to create a vector of values running from 1 to 9. Here's how to do it without typing each number:

>> **X = 1: 9**

X =

1 2 3 4 5 6 7 8 9

The notation 1: 9 is used to represent a vector of numbers running from 1 to 9 in increments of 1. The increment can be specified as the middle of three arguments:

>> **X = 0 : 2 : 6**

X = 0 2 4 6

■ The elements of the vector X can be extracted as X(1), X(2), etc. For example:

>> **X(3)**

ans = 4

You can perform mathematical operations on vectors. For example, to square the elements of the vector X, type

>> **X.^2**

ans = 0 4 16 36

■ To change the vector X from a row vector to a column vector, put a prime (‘) after X:

>> X’

ans =

0
2
4
6

>> X = 6 : - 2 : 0

X = 6 4 2 0

Increments can be fractional or negative, for example, 0: 0.1: 1 or 100 : -1 : 0.

linspace command

A vector with **n** elements that are linearly (equally) spaced in which the first element is **xi** and the last element **xn** can be created using linspace command as follow:

Variable_Name = linspace(xi , xn , m)

Matlab Command	Command Abbreviation	Command Interpretation
>> x = linspace (n, m)	Linspace(1st number, last number)	Create a linearly spaced vector between n and m.
>> x = linspace (n, m, k)	Linspace (1st number, last number, number of elements)	Create a linearly spaced vector between n and m with k elements between them

Example:

>> m = linspace(1,4,5)

m = 1.0000 1.7500 2.5000 3.2500 4.0000

- A column vector is created the same way as the row vector, and the rows are separated by semicolons
- To input a matrix, you basically define a variable. For a matrix the form is: variable name = [#,#; #,#; #,#; ….]

Example

```
>> x=[0 0.25*pi 0.5*pi]
   x =
        0    0.7854    1.5708
>> y=[0; 0.25*pi; 0.5*pi]
   y =
             0
        0.7854
        1.5708
```

Examples: Arithmetic operations between row vector and scalar

```
>>x=[2 5 1 4]
      x=    2     5     1     4
>>z=x+2
      z=    4     7     3     6
>>z=x-2
      z=    0     3    -1     2
>>z=x*2
      z=    4    10     2    16
>>z=x/2   %unwanted
      z=    1.0000    2.5000    0.5000    2.0000

>> z=x\2     %unwanted
      z =
           0
      0.4000
           0
           0
>> z=x./2
      z =   1.0000    2.5000    0.5000    2.0000
>>z=x .\2
      z=    1.0000    0.4000    2.0000    0.5000
>>z=x ^2
Error using  ^
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.
>>z=x .^2
      z=    4    25     1    16
```

Examples: Arithmetic operations between two row vectors

```
>>x=[2 5 1 4]
      x=    2     5     1     4
>>y=[3 4 8 9]
      y=    3     4     8     9
>>z=x+y          % the two vectors must have the same no. of elements
      z=    5     9     9     13
>>z=x-y          % the two vectors must have the same no. of elements
      z=    -1    1     -7    -5
>>z=x*y
Error using  *
Inner matrix dimensions must agree.
>>z=x.*y
      z=    6     20    8     36
>> z=x/y   %unwanted
      z =   0.4118
>> z=x\y   %unwanted
      z =
            0     0     0     0
       0.6000  0.8000  1.6000  1.8000
            0     0     0     0
            0     0     0     0
>>z=x./y
      z=    0.6667  1.2500  0.1250  0.4444
>>z=x.\y
      z=    1.5000  0.8000  8.0000  2.2500
```

Multiplying a Matrix by Another Matrix

To perform matrix multiplication, the first matrix must have the same number of columns as the second matrix has rows. The number of rows of the resulting matrix equals the number of rows of the first matrix, and the number of columns of the resulting matrix equals the number of columns of the second matrix.

To multiply matrix (2x3) by matrix (3x2) the answer is matrix (2x2)

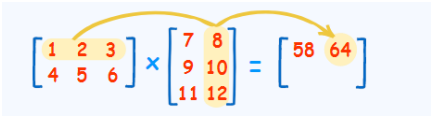
To work out the answer for the 1st row **and** 1st column**:**

Multiply matching members, then sum up:

$$(1, 2, 3) \bullet (7, 9, 11) = 1\times7 + 2\times9 + 3\times11 \quad = 58$$

We match the 1st members (1 and 7), multiply them, likewise for the 2nd members (2 and 9) and the 3rd members (3 and 11), and finally sum them up.

Want to see another example? Here it is for the 1st row and **2nd column**:



$$(1, 2, 3) \bullet (8, 10, 12) = 1\times8 + 2\times10 + 3\times12 \quad = 64$$

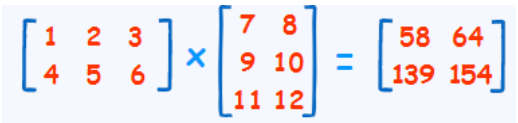
We can do the same thing for the **2nd row** and **1st column**:

$$(4, 5, 6) \bullet (7, 9, 11) = 4\times7 + 5\times9 + 6\times11 \quad = 139$$

And for the **2nd row** and **2nd column**:

$$(4, 5, 6) \bullet (8, 10, 12) = 4\times8 + 5\times10 + 6\times12 \quad = 154$$

And we get:



Examples: Arithmetic operations between row vector and column vector

```
>>x=[2 5 1 4]
      x=    2     5     1     4
>>y=[3; 4; 8; 9]
      y=    3
            4
            8
            9
>>z=x+y
      Error using + matrix dimensions must agree
>>z=x-y
      Error using - matrix dimensions must agree
>>z=x*y            % z(1x1)=x(1x4)*y(4x1)
      z=70         %  z=2*3+5*4+1*8+4*9=70
>>z=x.*y
      Error using .* matrix dimensions must agree
>>z=x/y
      Error using / matrix dimensions must agree
>>z=x\y
      Error using \ matrix dimensions must agree
>>z=x./y
      Error using  ./ matrix dimensions must agree
>>z=x.\y
      Error using .\ matrix dimensions must agree
```

Built in Functions

```
>>A = [ 13 22 76 44 90 12 16 13];      % A one-dimensional array (row vector)
>>mean(A)          % Find the average value of vector A
        ans = 35.7500
>>max(A)           % Find the largest elements among all elements in vector A
        ans = 90
>>min(A)           % Find the smallest elements among all elements in vector A
        ans = 12
>>sum(A)           % Adds all elements of vector A
        ans = 286
>>sort(A)          % Ascending arrangement
        ans = 12    13    13    16    22    44    76    90
>>median(A)        % Median value of vector A
        ans = 19
>>std(A)           % Standard deviation of vector A
        ans = 31.1895
```

Matrices

- Consider the 3x4 matrix

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

- It can be entered in MATLAB with the command

>> A = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12]

A =

1 2 3 4

5 6 7 8

9 10 11 12

- Note that the matrix elements in any row are separated by commas, and the rows are separated by semicolons. The elements in a row can also be separated by spaces.

Matrix addressing

-matrixname(row, column)

-colon may be used in place of a row or column reference to select the entire row or column

Example

>> A(2,3)

ans = 7

>> A(1,:)

ans = 1 2 3 4

>> A(:,1)

ans =

1

5

9

Matrices Commands

- zeros(n) – returns a nxn matrix of zeros
- zeros(m,n) – returns a mxn matrix of zeros
- ones(n) – returns a nxn matrix of ones
- ones(m,n) – returns a mxn matrix of ones
- size(A) – for a mxn matrix, returns the row vector [m,n] containing the number of rows and columns in matrix
- length(A) – returns the larger of the number of rows and columns in A

Examples:

>>a=zeros(3)

```
a=

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

>>a=zeros(3,2)

```
a=

     0     0
     0     0
     0     0
```

```
>>a=ones(3)

    a=

      1   1   1
      1   1   1
      1   1   1
```

```
>>a=ones(3,2)

    a=

      1   1
      1   1
      1   1
```

```
>>b=[2 4;5 6;8 1]

    b=

      2   4
      5   6
      8   1
```

```
>>size(b)

    ans=    3    2
```

```
>>length(b)

    ans=3
```

Built in Functions

```
>>A = [4 2 6; 5 1 3]
      A=
            4    2    6
            5    1    3
>>mean(A)          % perform the mean (average) for each column
      ans = 4.5000    1.5000    4.5000
>>median(A)        % perform the median for each column
      ans = 4.5000    1.5000    4.5000
>>sum(A)           % perform the sum for each column
      ans = 9     3     9
>>sort(A)          % perform the ascending sort for each column
      ans =
            4    1    3

            5    2    6
```

Arithmetic Operators

When generating code for the Embedded MATLAB Function block, the coder supports the arithmetic operators (and their M-function equivalents) listed in **Table (3.1).**

Table (3.1) Matrices arithmetic operators

Operation	Operator Syntax
Binary addition	**A+B**
Matrix multiplication	**A*B**
Array wise multiplication	**A.*B**
Matrix right division	**A/B**
Arraywise right division	**A./B**
Matrix left division	**A\B**
Arraywise left division	**A.\B**
Matrix power	**A^B**
Arraywise power	**A.^B**
Complex transpose	**A'**
Matrix transpose	**A.'**
Matrix concat	**[A B]**
Matrix index	**A(r,c)**

- If two matrices A and B are the same size, their (element-by-element) sum is obtained by typing A + B.
- You can also add a scalar (a single number) to a matrix; A + c adds c to each element in A.
- Likewise, A - B represents the difference of A and B, and A – c subtracts the number c from each element of A.
- If A and B are multiplicatively compatible, i. e., if A is n x m and B is m x l, then their product A* B is n x l. Recall that the element of A* B in the ith row and jth column is the sum of the products of the elements from the ith row of A times the elements from the jth column of B, i. e.,

$$(A*B)_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj}, \qquad 1 \le i \le n, \quad 1 \le j \le l.$$

- A' represents the conjugate transpose of A. (For more information, see the online help for ctranspose and transpose.)

Matrix Operations – Summary listed in **Table (3.2)**

Table (3.2) Matrix Operations – Summary

Operation	Command	Comment
Transpose	**B=A'**	Exchange rows with columns
Identity matrix	**eye(n)**	Return an nxn identity matrix
	eye(m.n)	Return an mxn matrix with ones on the main diagonal and zero elsewhere
Addition	**C=A+B**	
Subtraction	**C=A-B**	
Scalar Multiplication	**B=aA**	Where a is a scalar
Matrix Multiplication	**C=A*B**	
Matrix Inverse	**B=inv(A)**	A must be a square matrix
Matrix Rank	**C=rank(A)**	Return the rank of matrix A provides an estimate of the number of linearly independent rows or columns of a matrix A
Matrix Powers	**B=A.^2**	Square each element in the matrix
	C= A^2=A*A	Compute A*A, and A must be a square matrix
Determinant	**det(A)**	A must be a square matrix

Examples:

>>a=eye(3)

```
a=

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

>>a=eye(3,2)

```
a=

     1     0
     0     1
     0     0
```

>>k=[2 4;5 6;8 1]

k=

2 4
5 6
8 1

>> c=k'

c =

2 5 8

4 6 1

>>b=[1 4;2 5]

b=

1 4
2 5

>>d=det(b)

d= -3

det(b)=1*5-4*2

>>c=inv(b)

c=

-1.6667 1.3333

0.6667 -0.3333

inv(b) = 1/b = b^{-1} = adj(b) / det(b)

$$= \begin{bmatrix} 5 & -4 \\ -2 & 1 \end{bmatrix} / -3$$

Examples: Arithmetic operations between matrix and scalar

```
>>a=[2 7;0 -2]

    a=

        2     7
        0    -2
```

```
>>z=a+3

    z=

        5    10
        3     1
```

```
>>z=a-3

    z=

       -1     4
       -3    -5
```

```
>>z=a*3

    z=

        6    21
        0    -6
```

```
>>z=a/3    %unwanted

    z=

   0.6667         2.3333

        0        -0.6667
```

```
>> z=a\3      %unwanted
        Error using  \
        Matrix dimensions must agree.
>> z=a./3


z =

   0.6667    2.3333
        0   -0.6667
>>z=a.\3
      z=
         1.5000    0.4286
           Inf   -1.5000
>>z=a^3    %z=a*a*a
z =
    8    28
    0    -8
```

$$C=\begin{bmatrix}2 & 7\\ 0 & -2\end{bmatrix}*\begin{bmatrix}2 & 7\\ 0 & -2\end{bmatrix}*\begin{bmatrix}2 & 7\\ 0 & -2\end{bmatrix}$$

$$=\begin{bmatrix}4 & 0\\ 0 & 4\end{bmatrix}*\begin{bmatrix}2 & 7\\ 0 & -2\end{bmatrix}=\begin{bmatrix}8 & 28\\ 0 & -8\end{bmatrix}$$

```
>>z=a.^3
z=
        8   343
        0    -8
```

Examples: Arithmetic operations between two matrices

```
>>a=[2 7;0 -2]

    a=

     2     7
     0    -2

>>b=[1 4;2 5]

    b=

     1     4
     2     5

>>c=a+b
    c=
          3    11
          2     3

>>c=a-b
    c=
          1     3
         -2    -7

>>c=a*b
c =

    16    43
    -4   -10

>>c=a.*b
    c=
          2    28
          0   -10
```

$$C=\begin{bmatrix}2 & 7\\ 0 & -2\end{bmatrix}*\begin{bmatrix}1 & 4\\ 2 & 5\end{bmatrix}=\begin{bmatrix}16 & 43\\ -4 & -10\end{bmatrix}$$

>>c=a/b

c=

1.3333 0.3333

-1.3333 0.6667

c=a/b=a* 1/b = a*b^{-1} = a*inv(b) = a*adj(b) / det(b)

$$= \begin{bmatrix} 2 & 7 \\ 0 & -2 \end{bmatrix} * \begin{bmatrix} 5 & -4 \\ -2 & 1 \end{bmatrix} / -3 = -\frac{1}{3} * \begin{bmatrix} 2 & 7 \\ 0 & -2 \end{bmatrix} * \begin{bmatrix} 5 & -4 \\ -2 & 1 \end{bmatrix}$$

$$= -\frac{1}{3} * \begin{bmatrix} -4 & -1 \\ 4 & -2 \end{bmatrix} = \begin{bmatrix} \frac{4}{3} & \frac{1}{3} \\ \frac{-4}{3} & \frac{2}{3} \end{bmatrix}$$

>>c=a./b

c=

2.0000 1.7500

0 -0.4000

>>c=a\b

c=

4.0000 10.7500

-1.0000 -2.5000

c=a\b =1/a * b = a^{-1} *b= inv(a) *b =(adj(a) / det(a)) *b

$$= \left(\begin{bmatrix} -2 & -7 \\ 0 & 2 \end{bmatrix} / -4 \right) * \begin{bmatrix} 1 & 4 \\ 2 & 5 \end{bmatrix} = -\frac{1}{4} * \begin{bmatrix} -2 & -7 \\ 0 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 4 \\ 2 & 5 \end{bmatrix}$$

$$= -\frac{1}{4} * \begin{bmatrix} -16 & -43 \\ 4 & 10 \end{bmatrix} = \begin{bmatrix} \frac{16}{4} & \frac{43}{4} \\ \frac{-4}{4} & \frac{-10}{4} \end{bmatrix}$$

>>c=a.\b

c=

0.5000 0.5714

Inf -2.5000

>> **c=[a b]**

c=

2 7 1 4

0 -2 2 5

The "rand" Command

Command	Description	Example
rand	Generates a single random number between 0 and 1	>> rand ans=0.2311
rand(1,n)	Generates an n-element row vector of random numbers between 0 and 1	>>b=rand(1,4) b=0.6068 0.4860 0.8913 0.7621
rand(n)	Generates an nxn matrix with random numbers between 0 and 1	>>b=rand(3) b= 0.4565 0.4447 0.9218 0.0185 0.6154 0.7382 0.8214 0.7919 0.1763
rand(m,n)	Generates an mxn matrix with random numbers between 0 and 1	>>c=rand(2,4) c= 0.4057 0.9169 0.8936 0.3529 0.9355 0.4103 0.0.579 0.8132
Randperm(n)	Generates a row vector with n elements that are random permutation of integers 1 through n	>>randperm(8) ans=8 2 7 4 3 6 5 1

The "randi" Command

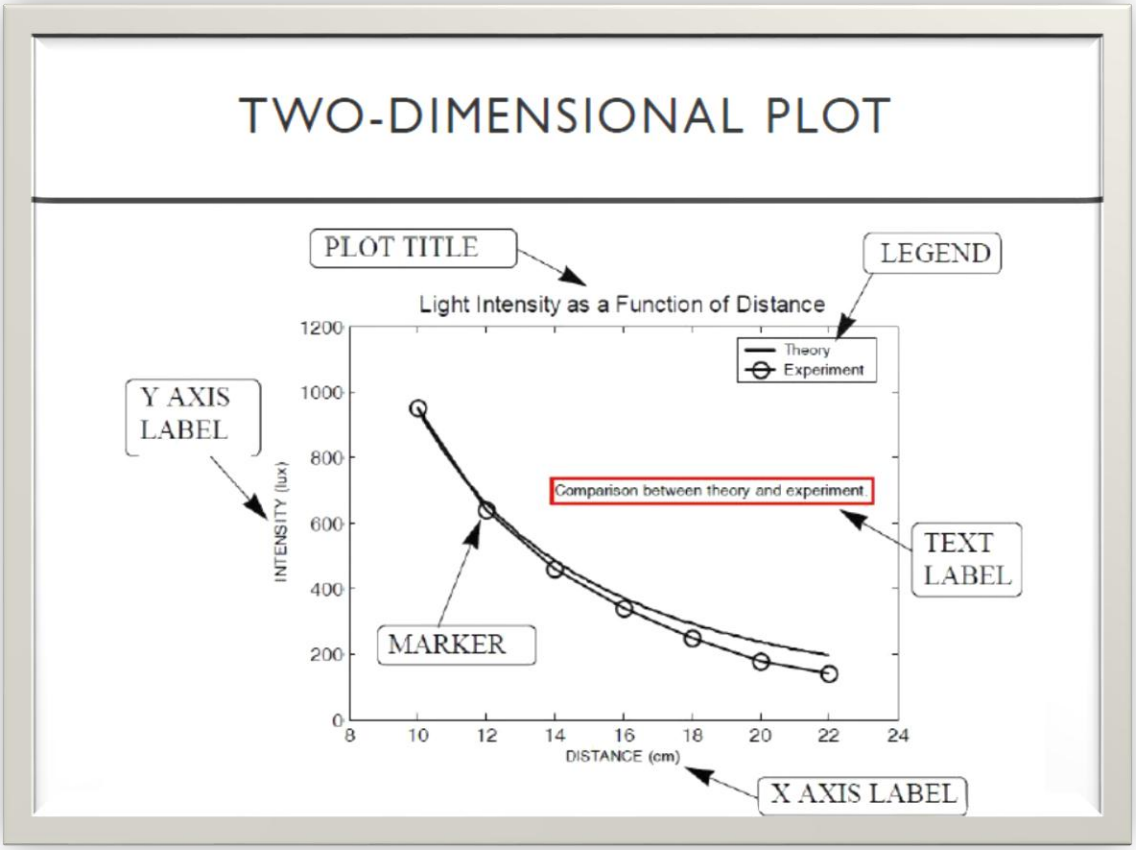
Command	Description	Example
randi(imax)	Generates a single random number between 1 and imax	>> a=randi(15) a=9
randi(imax,n)	Generates an nxn matrix with random integers between 1 and imax	>>b=randi(15,3) b= 4 8 11 14 3 8 1 15 8
randi(imax,m,n)	Generates an mxn matrix with random integers between 1 and imax	>>c=randi(15,2,4) c= 1 7 8 13 11 2 7 5

2D Plotting

•Another powerful feature of MatLab software is how to present data in graphical mode.

•MatLab deals with many built-in functions that can be used to create different plot styles.

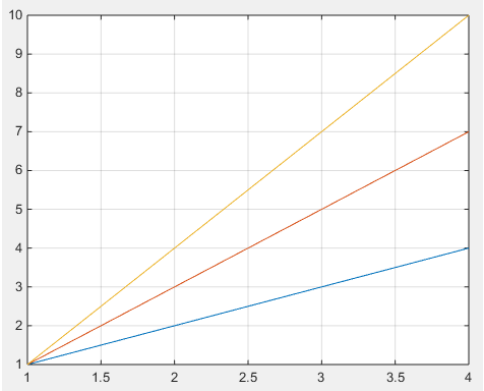
•This lecture describes how MatLab can be used to create and format two-dimensional plot and how plot characteristics can be changed according to the user demand.



Plotting Matrices

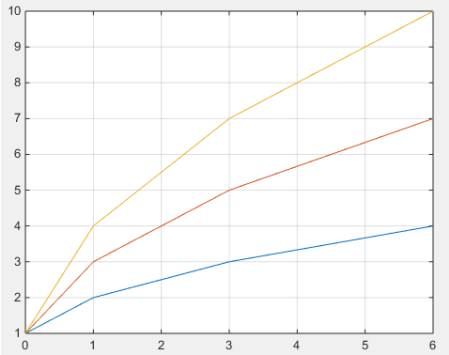
If one of the arguments to the plot command is a matrix, matlab will use the columns of the matrix to plot a set of lines, one line per column:

```
>> q = [1 1 1;2 3 4;3 5 7;4 7 10]
      q =
          1  1  1
          2  3  4
          3  5  7
          4  7  10
>> plot(q)
>> grid
```



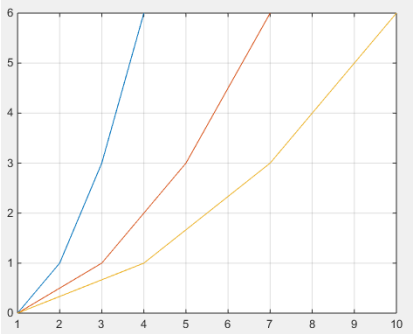
matlab plots the columns of the matrix q against the row index. You can also supply an *x* variable:

```
>> x = [0 1 3 6]
      x =
          0 1 3 6
>> plot(x,q)
>> grid
```



Here the *x* values are not uniformly spaced, but they are the same for each column of q. You can also plot a matrix of *x* values against a vector of *y* values (be careful: the *y* values are in the vector x):

```
>>plot(q,x)
>>grid
```



PLOT(X,Y) plots vector Y versus vector X. If X or Y is a matrix, then the vector is plotted versus the rows or columns of the matrix, whichever line up. If X is a scalar and Y is a vector, disconnected line objects are created and plotted as discrete points vertically at X.

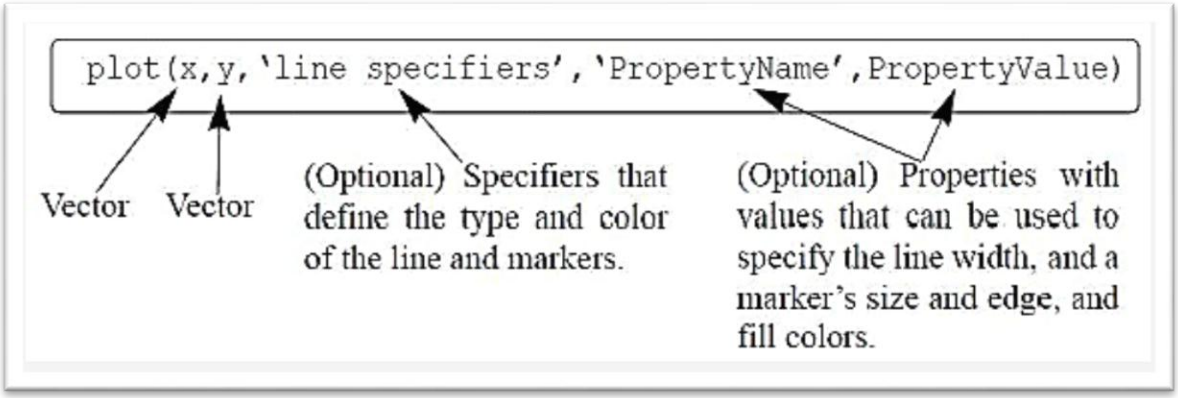
PLOT(Y) plots the columns of Y versus their index. If Y is complex, PLOT(Y) is equivalent to PLOT(real(Y), imag(Y)). In all other uses of PLOT, the imaginary part is ignored.

plot Specifications

•The *plot* command has additional options that used to specify the line properties.

•Color, line style marker, axis labels, title are options those can be configured for plotting figures.

•To make *plot* command works with such changes, we use:



Various line types, plot symbols and colors may be obtained with PLOT(X,Y,S) where S is a character string made from one element from any or all the following 3 columns in **Table (4.1).**

Table (4.1): Line Specifier

Line Color	Marker Type	Line Style
b blue	. point	- solid
g green	o circle	: dotted
r red	x x-mark	-. dashdot
c cyan	+ plus	-- dashed
y yellow	* star	(none) no line
m magenta	s square	
w white	d diamond	
k black	v triangle (down)	
	^ triangle (up)	
	< triangle (left)	
	> triangle (right)	
	p pentagram	
	h hexagram	

Notes to Be Considered

•To import specifiers inside the *plot* command, they have to be introduced as a string command.

•The specifiers can be defined in any order

•The number of specifier is optional, you can make one, two or more depending on what you need to define.

Example

```
>> plot ( x, y )                    %plot without specifier
>> plot ( x, y, 'r')                %plot with red color specifier
>> plot ( x, y, '- - r', '^' )      %plot with red, dashed-line, and marker specifier
>>plot(X,Y,'c+:')                  %plots a cyan dotted line with a plus at each data point
%plots blue diamond at each data point but does not draw any line.
>>plot(X,Y,'bd')
% to create a plot with a dark red line width of 2 points.
>>plot(X,Y,'LineWidth',2,'Color',[6 0 0])
```

xlabel, ylabel, zlabel

Each axes graphics object can have one label for the *x*-, *y*-, and *z*-axis. The label appears beneath its respective axis in a two-dimensional plot and to the side or beneath the axis in a three-dimensional plot.

```
>>xlabel ('string')          % labels the x-axis of the current axes.
>>ylabel ('string')          % labels the y-axis of the current axes.
>>zlabel ('string')          % labels the z-axis of the current axes.
```

title

To create or modify a plot's title from a GUI, use Insert Title from the figure menu. The title is located at the top and in the center of the axes.

```
>>title('string')
```

grid

Grid lines for 2-D and 3-D plots

```
>>  grid on
```

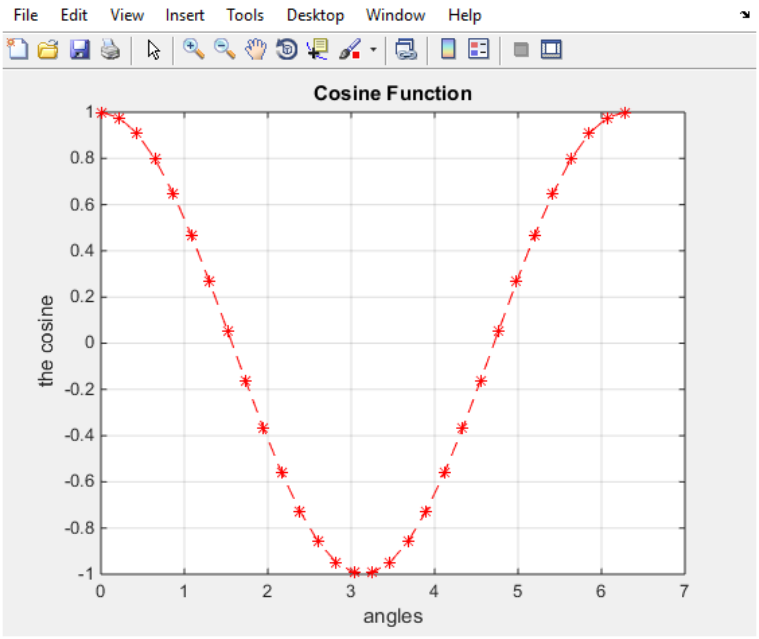
axis

Axis scaling and appearance

```
>>axis([xmin xmax ymin ymax zmin zmax])  % sets the x-, y-, and z-axis limits
```

Example

```
x = linspace (0, 2*pi, 30);        % a vector of 30 angles between 0 and 2pi
y = cos (x);          % a vector of cosine function for each angle defined by x
plot ( x, y, 'r--*')  % two-dim. plot with red, dashed-line type and * marker  specifier
grid on;                           % opens grid on the graphed background
xlabel ('angles')                  % gives a title for the x-axis
ylabel ('the cosine')              % gives and title for the y-axis
title ('Cosine Function')          % gives a title for the whole plot.
```



Property Name and Value

•To specify the thickness of the line, the size of the marker, and the color of the marker's edge and fill we do the follow:

```
plot(x,y,'-mo','LineWidth',2,'markersize',12,
     'MarkerEdgeColor','g','markerfacecolor','y')
```

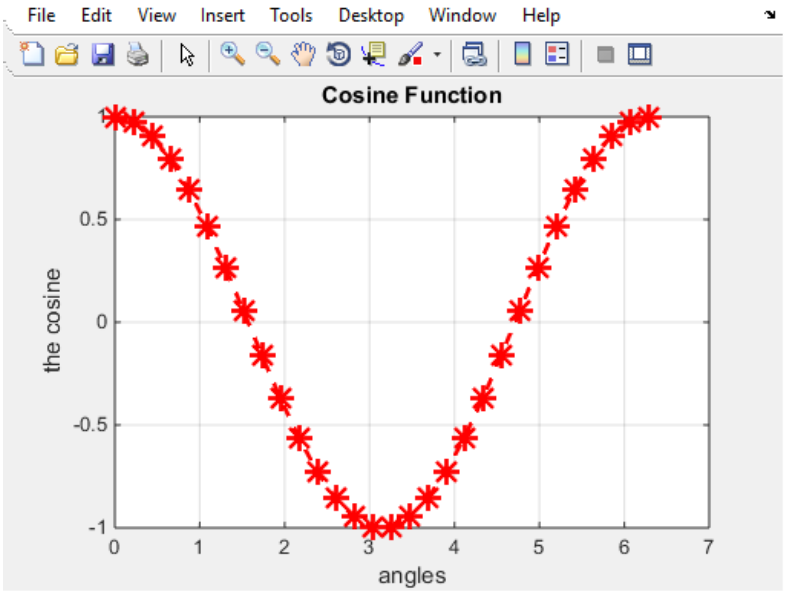
•All properties of the name and value have to be inside the *plot* command listed in **Table (4.2).**

Table (4.2): Line and Marker Property

Property Name	Description	Possible Property Values
LineWidth **(or linewidth)**	Specify the width of the line	A number in units of points (default 0.5)
MarkerSize **(or markersize)**	Specify the size of the marker	A number in units of points
MarkerEdgeColor **(Or markeredgecolor)**	Specify the color of the marker, or the color of the edge line for filled marker	Color specifies from the table above, typed as string
MarkerFaceColor **(Or markerfacecolor)**	Specify the color of the filling for filled marker	Color specifies from the table above, typed as string

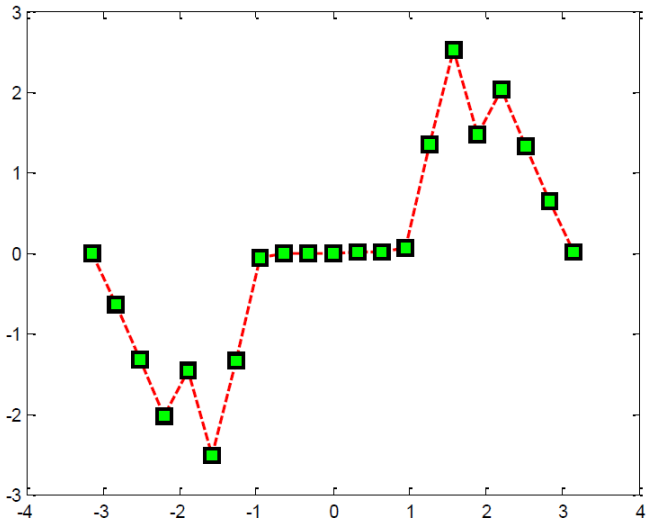
Example

```
x = linspace (0, 2*pi, 30);        % a vector of 30 angles between 0 and 2pi
y = cos (x);          % a vector of cosine function for each angle defined by x
% two-dimensional plot with red, dashed-line type and * marker specifier,
% and the line width = 2 and the marker size = 12
plot ( x, y, 'r--*','linewidth', 2,'markersize', 12 )
grid on;                           % opens grid on the graphed background
xlabel ('angles')                  % gives a title for the x-axis
ylabel ('the cosine')              % gives and title for the y-axis
title ('Cosine Function')          % gives a title for the whole plot.
```



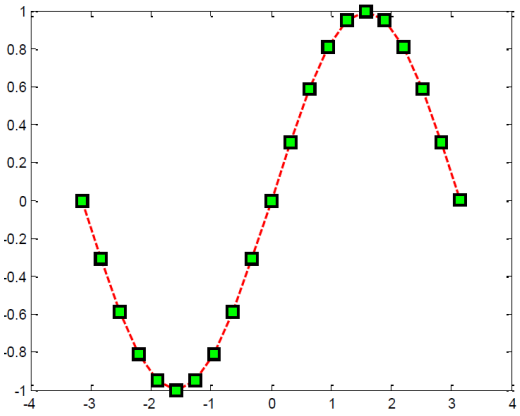
Example

```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'--rs','LineWidth',2,'MarkerEdgeColor','k', 'MarkerFaceColor','g','MarkerSize',10)
```



Example

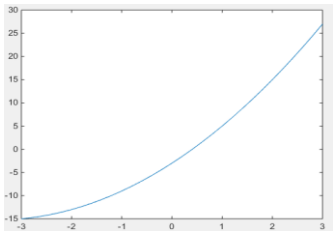
```
x = -pi:pi/10:pi;
y = sin(x) ;
plot(x,y,'--rs','LineWidth',2,'MarkerEdgeColor','k','MarkerFaceColor','g',...
'MarkerSize',10)
```



Example

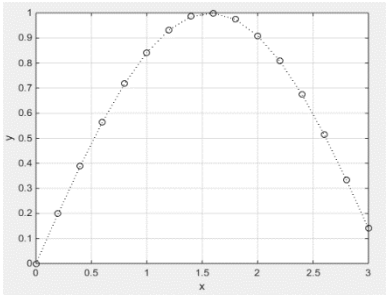
To plot the quadratic x^2+7x-3 from x equals −3 to 3 in steps of 0.2 we use the code

```
x = -3:0.2:3;
y = x.^2+7*x-3;
plot(x,y)
```



Example

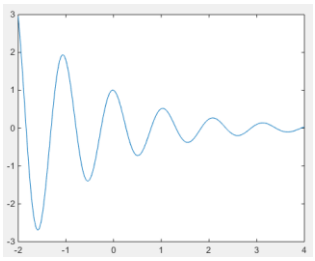
```
x = 0:0.2:pi;     % Create x-array
y = sin(x);       % Create y-array
plot(x,y,'k:o') % Plot x-y points with specified color
                  % and symbol ('k' = black, 'o' = circles)
grid on           % Display coordinate grid
xlabel('x')       % Display label for x-axis
ylabel('y')       % Display label for y-axis
```



Example

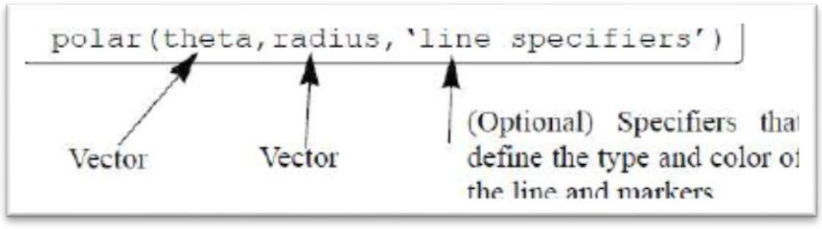
% A script file that creates a plot of the function: 3.5.^(-0.5*x).*cos(6x)

```
x=[-2:0.01:4];
y=3.5.^(-0.5*x).*cos(6*x);
plot(x,y)
```



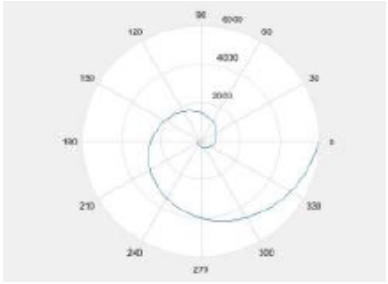
Polar plot

•*polar* plot used to plot a point identified by its value and the rotating angle as follow:



Example

```
x = linspace(pi,4*pi,100);
y = 3.*sin(0.5*x)+3.*x.^3;
polar(x,y)
```

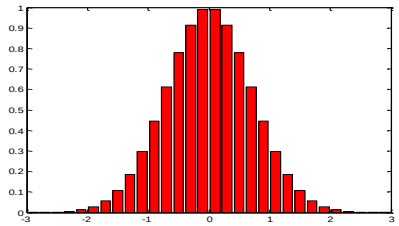


Example

Single Series of Data

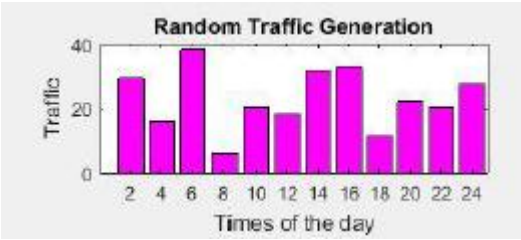
This example plots a bell-shaped curve as a bar graph and sets the colors of the bars to red.

```
x = -2.9:0.2:2.9;
bar(x,exp(-x.*x),'r')
```



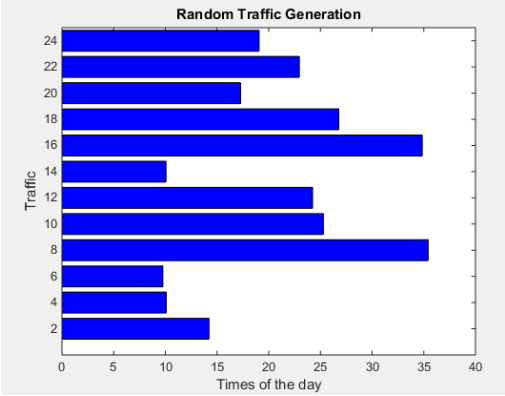
Example

```
x = [02 04 06 08 10 12 14 16 18 20 22 24];
y = 35*rand(1,12)+5;
bar(x,y,'m')
xlabel('Times of the day')
ylabel('Traffic')
title('Random Traffic Generation')
```



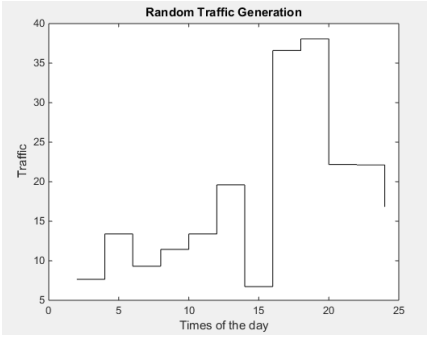
Example

```
x = [02 04 06 08 10 12 14 16 18 20 22 24];
y = 35*rand(1,12)+5;
barh(x,y,'b')
xlabel('Times of the day')
 ylabel('Traffic')
 title('Random Traffic Generation')
```



Example

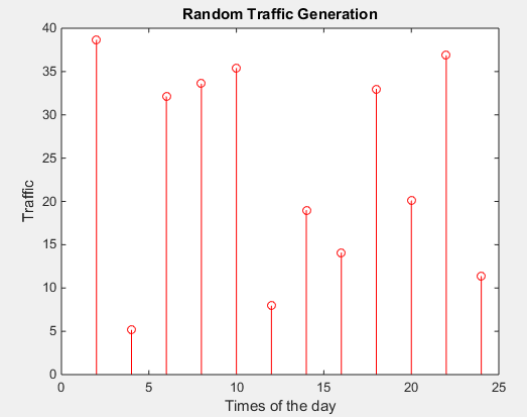
```
x = [02 04 06 08 10 12 14 16 18 20 22 24];
y = 35*rand(1,12)+5;
stairs(x,y,'k');
```



```
xlabel('Times of the day')
 ylabel('Traffic')
 title('Random Traffic Generation')
```

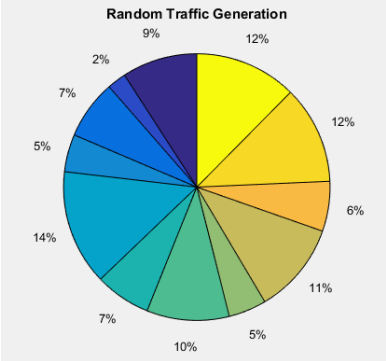
Example

```
x = [02 04 06 08 10 12 14 16 18 20 22 24];
y = 35*rand(1,12)+5;
stem(x,y,'r')
xlabel('Times of the day')
 ylabel('Traffic')
title('Random Traffic Generation')
```



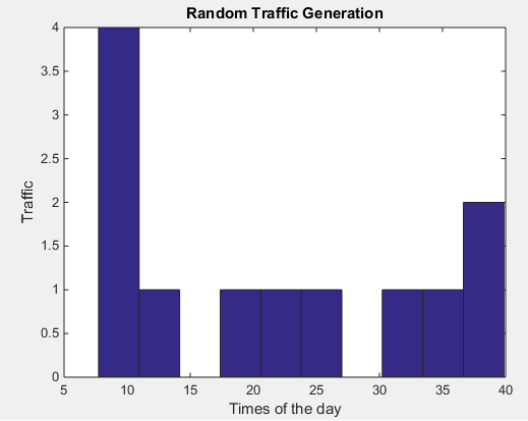
Example

```
x = [02 04 06 08 10 12 14 16 18 20 22 24];
y = 35*rand(1,12)+5;
pie(y)
xlabel('Times of the day')
 ylabel('Traffic')
 title('Random Traffic Generation')
```

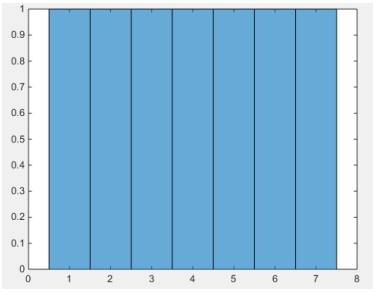


Example

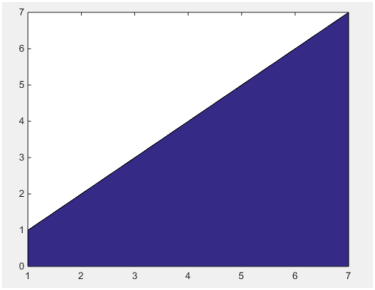
```
x = [02 04 06 08 10 12 14 16 18 20 22 24];
y = 35*rand(1,12)+5;
hist(y)
xlabel('Times of the day')
ylabel('Traffic')
title('Random Traffic Generation')
```



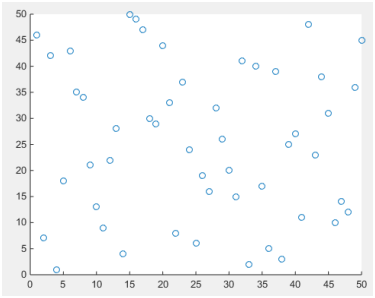
```
x =1:7;

histogram(x)
```



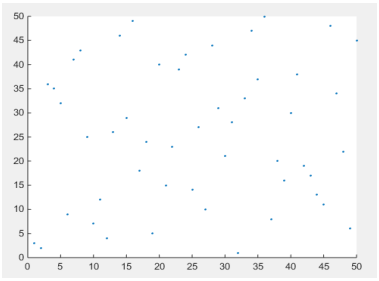
```
x =1:7;
y=7:-1:1;
area(x)
```



```
x =randperm(50);
y=randperm(50);
scatter(x,y)
```



```
x =randperm(50);
y=randperm(50);
scatter(x,y,5,'filled')
```



3D Plotting

plot3

The plot3 function displays a three-dimensional plot of a set of data points.

plot3(X1,Y1,Z1)

Examples

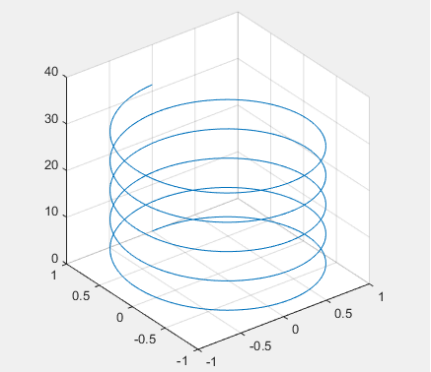
Plot a three-dimensional helix.

t = 0:pi/50:10*pi;

plot3(sin(t),cos(t),t)

grid on

axis square



Plotting Matrix Data

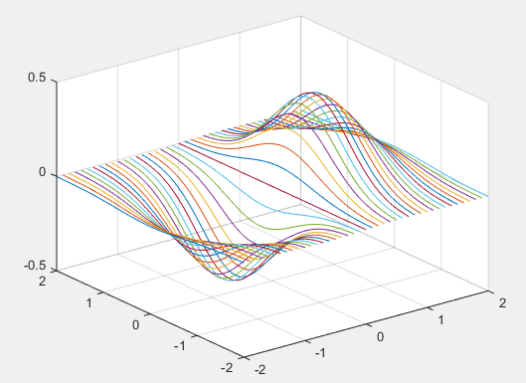
If the arguments to plot3 are matrices of the same size, lines obtained from the columns of X, Y, and Z are plotted. For example:

[X,Y] = meshgrid([-2:0.1:2]);

Z = X.*exp(-X.^2-Y.^2);

plot3(X,Y,Z)

grid on



Command	Description
meshgrid	Cartesian grid in 2-D/ 3-D space [X,Y] = meshgrid(xgv,ygv) replicates the grid vectors xgv and ygv to produce the coordinates of a rectangular grid (X, Y). The grid vector xgv is replicated numel(ygv) times to form the columns of X. The grid vector ygv is replicated numel(xgv) times to form the rows of Y. [X,Y] = meshgrid(gv) is equivalent to [X,Y] = meshgrid(gv,gv)
peaks	is a function of two variables, obtained by translating and scaling Gaussian distributions, which is useful for demonstrating mesh, surf, pcolor, contour, etc.
colormap hsv	colormap is a function that sets the Colormap property of a figure. An hsv colormap varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. The map is particularly useful for displaying periodic functions. You can use ***bone; colorcube; cool; copper; flag; gray; hot; Jet; lines; pink; prism; white; autumn; spring; summer; or winter*** instead of **hsv**

Mesh

mesh, meshc, and meshz create wireframe parametric surfaces specified by X, Y, and Z, with color specified by C.

A mesh is drawn as a surface graphics object with the viewpoint specified by view(3). The face color is the same as the background color (to simulate a wireframe with hidden-surface elimination), or none when drawing a standard see-through wireframe. The current colormap determines the edge color. The hidden command controls the simulation of hidden-surface elimination in the mesh, and the shading command controls the shading model.

Example

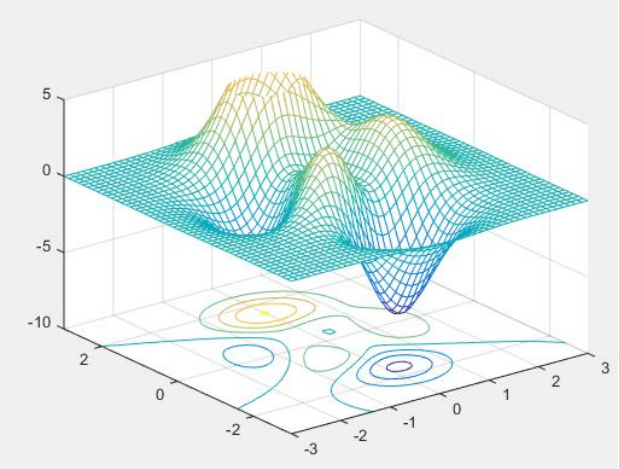
Produce a combination mesh and contour plot of the peaks surface:

[X,Y] = meshgrid(-3:.125:3);

Z = peaks(X,Y);

meshc(X,Y,Z);

axis([-3 3 -3 3 -10 5])



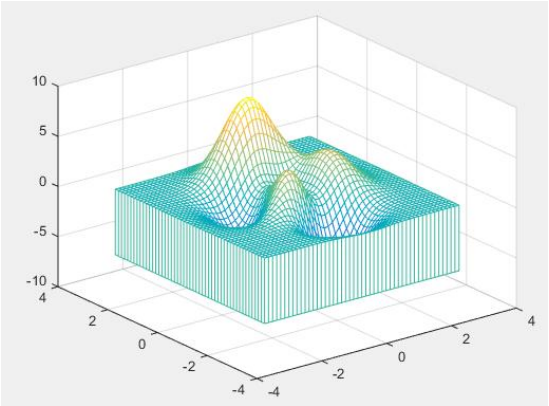
Example

Generate the curtain plot for the peaks function:

[X,Y] = meshgrid(-3:.125:3);

Z = peaks(X,Y);

meshz(X,Y,Z)



Example

surf, surfc

3-D shaded surface plot

surf(X,Y,Z,C) creates a shaded surface, with color defined by C. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

Example

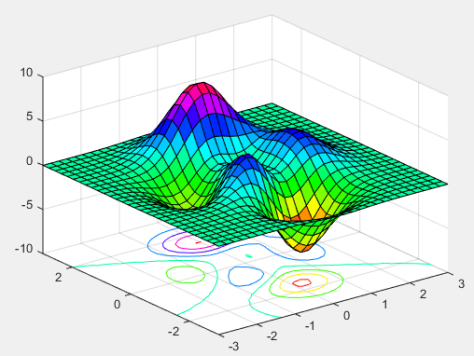
Display a surface plot and contour plot of the peaks surface.

[X,Y,Z] = peaks(30);

surfc(X,Y,Z)

colormap hsv

axis([-3 3 -3 3 -10 10])



pie3

pie3(X) draws a three-dimensional pie chart using the data in X. Each element in X is represented as a slice in the pie chart.

pie3(X,explode) specifies whether to offset a slice from the center of the pie chart. X(i,j) is offset from the center of the pie chart if explode(i,j) is nonzero. explode must be the same size as X.

Example

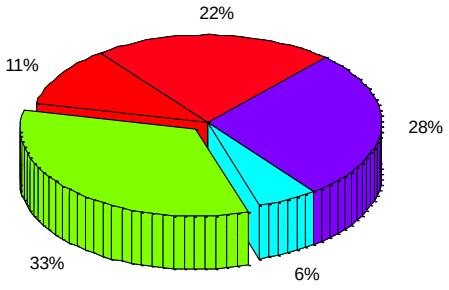
Offset a slice in the pie chart by setting the corresponding explode element to 1:

x = [1 3 0.5 2.5 2];

explode = [0 1 0 0 0];

pie3(x,explode)

colormap hsv



Multiple Figure Windows

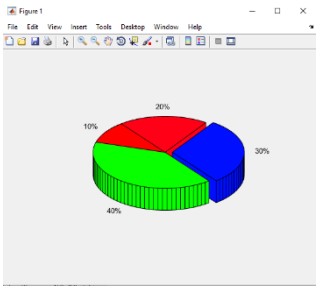
When plot or any other command that generates a plot is executed, the Figure Window opens (if not already open) and displays the plot. MATLAB labels the Figure Window as Figure 1. If the Figure Window is already open when the plot or any other command that generates a plot is executed, a new plot is displayed in the Figure Window (replacing the existing plot). Commands that format plots are applied to the plot in the Figure Window that is open. It is possible, however, to open additional Figure Windows and have several of them open (with plots) at the same time. This is done by typing the command figure. Every time the command figure is entered, MATLAB opens a new Figure Window. If a command that creates a plot is entered after a figure command, MATLAB generates and displays the new plot in the last Figure Window that was opened, which is called the active or current window.

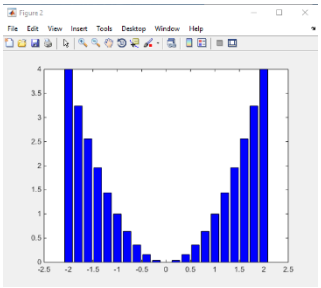
Example

Write a program to draw the following on multiple windows:

The 1st window includes cake chart contain the slices 1 4 3 2 and cut the 3rd slice. While the 2nd window draws full blue rectangle where $-2 \leq x \leq 2$ in step of 0.2 and the y is equal to x powered by 2.

```
x = [1 4 3 2];
explode = [0 0 1 0];
pie3(x,explode)
colormap hsv
figure
x = -2:0.2:2;
bar(x,x.^2,'b')
```





Putting Multiple Plots On The Same Page

Multiple plots can be created on the same page with the subplot command, which has the form:

subplot(m,n,p)

The command divides the Figure Window (and the page when printed) into ($m \times n$) rectangular subplots. The subplots are arranged like elements in an $m \times n$ matrix where each element is a subplot. The subplots are numbered from 1 through. The upper left subplot is numbered 1 and the lower right subplot is numbered. The numbers increase from left to right within a row, from the first row to the last. The command subplot(m,n,p) makes the subplot p current. This means that the next plot command (and any formatting commands) will create a plot (with the corresponding format) in this subplot. For example, the command subplot(3,2,1) creates six areas arranged in three rows and two columns as shown, and makes the upper left subplot current.

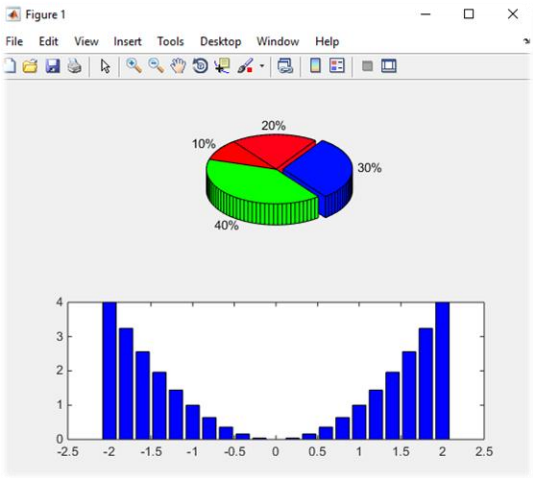
Example

Write a program to draw the following on the same window:

first draw cake chart contain the slices 1 4 3 2 and cut the 3rd slice.

then draw full blue rectangle where $-2 \leq x \leq 2$ in step of 0.2 and the y is equal to x powered by 2.

```
subplot(2,1,1)
x = [1 4 3 2];
explode = [0 0 1 0];
pie3(x,explode)
colormap hsv
subplot(2,1,2)
x = -2:0.2:2;
bar(x,x.^2,'b')
```



Conditional Statements

The if-elseif-else-end *Structure*

A conditional statement is a command that allows MATLAB to make a decision of whether to execute a group of commands that follow the conditional statement, or to skip these commands. In a conditional statement a conditional expression is stated. If the expression is true, a group of commands that follow the statement are executed. If the expression is false, the computer skips the group. The basic form of a conditional statement is if conditional expression consisting of relational operators (<;<=;>;>=;==;~=) and/or logical operators (&;|).

Examples:

```
if a < b
if c >= 5
if a == b
if a ~= 0
if (d<h)&(x>7)
if (x~=13)|(y<0)
```

See also **Table (6.1)**

Table (6.1): Some MATLAB if Representation

Mathematical Representation	MATLAB Representation
x in [2..7]	**If (x>=2) & (x<=7)**
x in (2..7)	**If (x>2) & (x<7)**
x in [2..7)	**If (x>=2) & (x<7)**
x in (2..7]	**If (x>2) & (x<=7)**
$2 \leq x \leq 7$	**If (x>=2) & (x<=7)**
$7 \geq x \geq 2$	**If (x>=2) & (x<=7)**

The if-elseif-else-end structure is shown in **Fig (6.1)**. The figure shows how the commands are typed in the program, and gives a flowchart that illustrates the flow, or the sequence, in which the commands are executed. This structure includes

two conditional statements (if and elseif) that make it possible to select one out of three groups of commands for execution. The first line is an if statement with a conditional expression. If the conditional expression is true, the program executes group 1 of commands between the if and the elseif statements and then skips to the end. If the conditional expression in the if statement is false, the program skips to the elseif statement. If the conditional expression in the elseif statement is true, the program executes group 2 of commands between the elseif and the else and then skips to the end. If the conditional expression in the elseif statement is false, the program skips to the else and executes group 3 of commands between the else and the end. It should be pointed out here that several elseif statements and associated groups of commands can be added.

In this way more conditions can be included. Also, the else statement is optional. This means that in the case of several elseif statements and no else statement, if any of the conditional statements is true the associated commands are executed; otherwise nothing is executed.

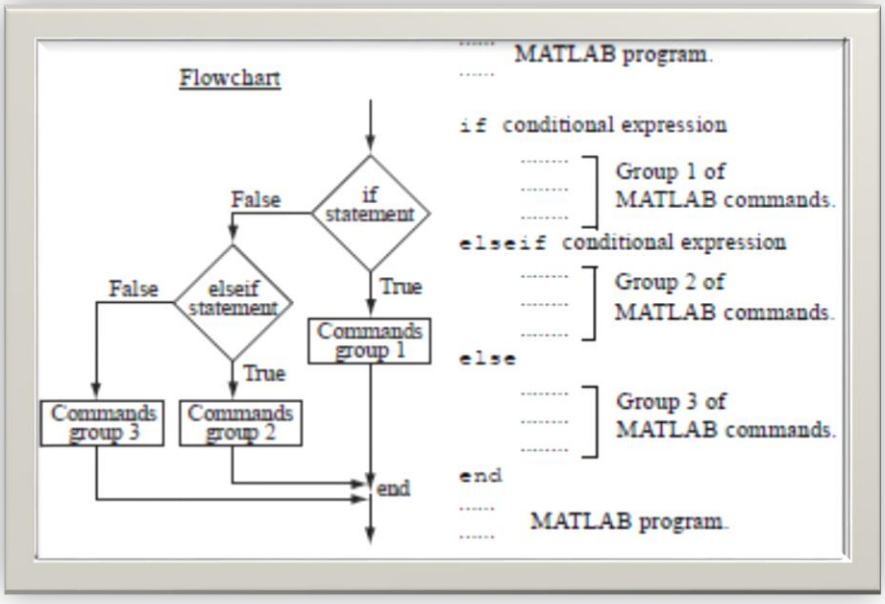


Fig (6.1): if-elseif-else-end structure

Example

Write a program to check the number when it is even or odd.

```
n=input('Enter the value of  number: ');

if (mod(n,2)==0)

  disp('even number')
else
 disp('odd number')
end

% The run of the program
>> test
Enter the value of number: 6
even number

% Another run of the program
>> test
Enter the value of number: 3
odd number
```

Example

Write a program to check the number when it is positive; negative; or zero.

```
n=input('Enter the value of  number: ');

if (n>0)

  disp('Positive number')
elseif (n<0)

  disp('Negative number')
else
 disp('Zero')
end
```

```
% The run of program
>> test
Enter the value of number: 6
Positive number

% Another run of program
>> test
Enter the value of number: 0
Zero

% Another run of program
>> test
Enter the value of number: -5
Negative number
```

Example

Write a program to do the following:

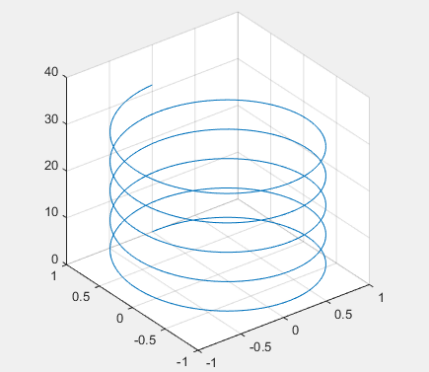
1-When the user enters number in [2..6] draw three-dimensional helix when angle changed from 0 to 10pi in steps of pi/50

2- When the user enters number equal 8 or 10 calculate the speed of the wave

3- When the user enters number equal 12 Welcome appeared on the screen.

```
n=input('Enter the value of number: ');

if (n>=2) & (n<=6)
   t = 0:pi/50:10*pi;
   plot3(sin(t),cos(t),t)
   grid on
   axis square
elseif (n==8) | (n==10)
   x=input('Enter wavelength: ');
   y=input('Enter frequency: ');
   speed=x*y;
   fprintf('Wave speed= %f\n',speed)
elseif (n==12)
    disp('Welcome')
end
```

% The run of program



>> test

Enter the value of number: 3

% Another run of program

>> test

Enter the value of number: 8

Enter wavelength: 5

Enter frequency: 4

Wave speed= 20.000000

% Another run of program

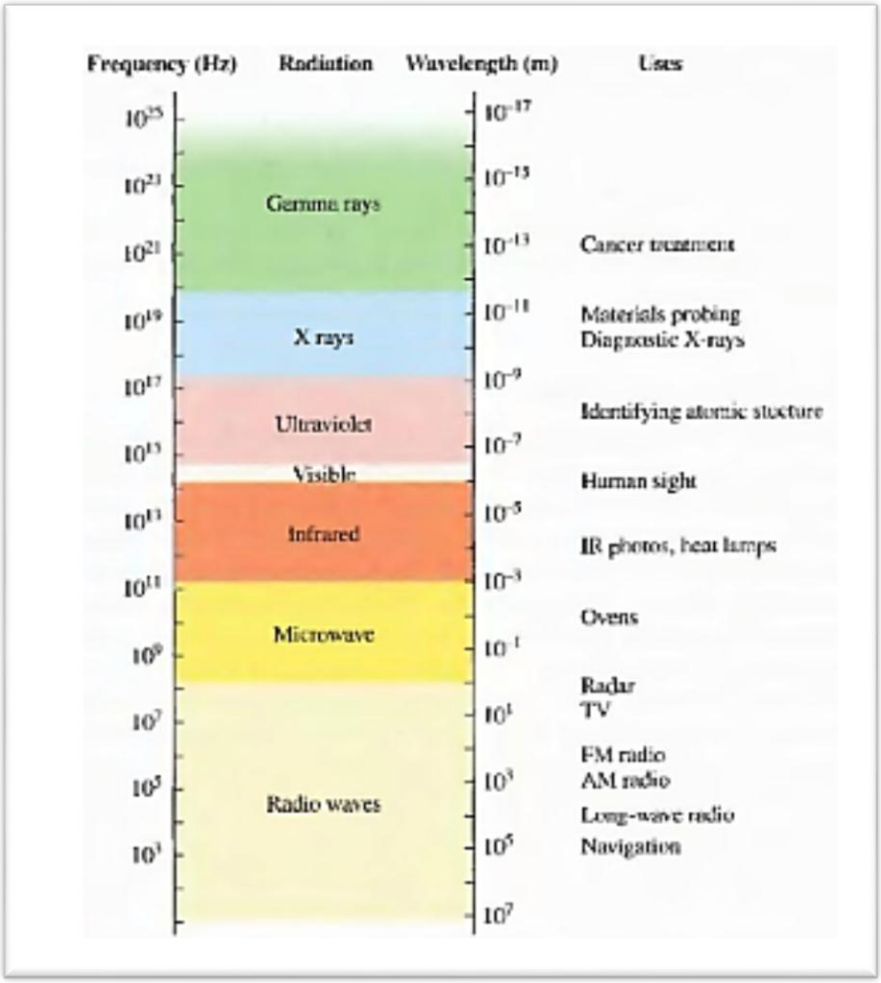
>> test

Enter the value of number: 12

Welcome

Example

Display the type of radiation according to the value of frequency




```
f=input('Enter the value of Frequency (Hz): ');

 if (f>=10) & (f<1e8)
      disp('Radio Waves')
 elseif (f>=1e8) & (f<1e11)
      disp('MicroWave')
 elseif (f>=1e11) & (f<1e14)
      disp('Infrared')
```

```
elseif (f>=1e14) & (f<1e15)
    disp('Visible')
elseif (f>=1e15) & (f<1e17)
    disp('UltraViolet')
elseif (f>=1e17) & (f<1e20)
    disp('X rays')
elseif (f>=1e20) & (f<1e25)
    disp('Gamma rays')
end
% The run of program
>> test
Enter the value of Frequency (Hz): 1e5
Radio Waves
% Another run of program
>> test
Enter the value of Frequency (Hz): 1e16
UltraViolet

% Another run of program
>> test
Enter the value of Frequency (Hz): 1e22
Gamma rays
```

The switch-case *Statement*

The switch-case statement is another method that can be used to direct the flow of a program. It provides a means for choosing one group of commands for execution out of several possible groups.

• The first line is the switch command, which has the form

`switch` switch expression

The switch expression can be a scalar or a string. Usually it is a variable that has an assigned scalar or a string. It can also be, however, a mathematical expression that includes pre-assigned variables and can be evaluated.

• Following the switch command are one or several case commands. Each has a value (can be a scalar or a string) next to it (value1, value2, etc.) and an associated group of commands below it.

• After the last case command there is an optional otherwise command followed by a group of commands.

• The last line must be an end statement.

The value of the switch expression in the switch command is compared with the values that are next to each of the case statements. If a match is found, the group of commands that follow the case statement with the match are executed. (Only one group of commands—the one between the case that matches and either the case, otherwise, or end statement that is next—is executed). As shown in **Fig (6.2).**

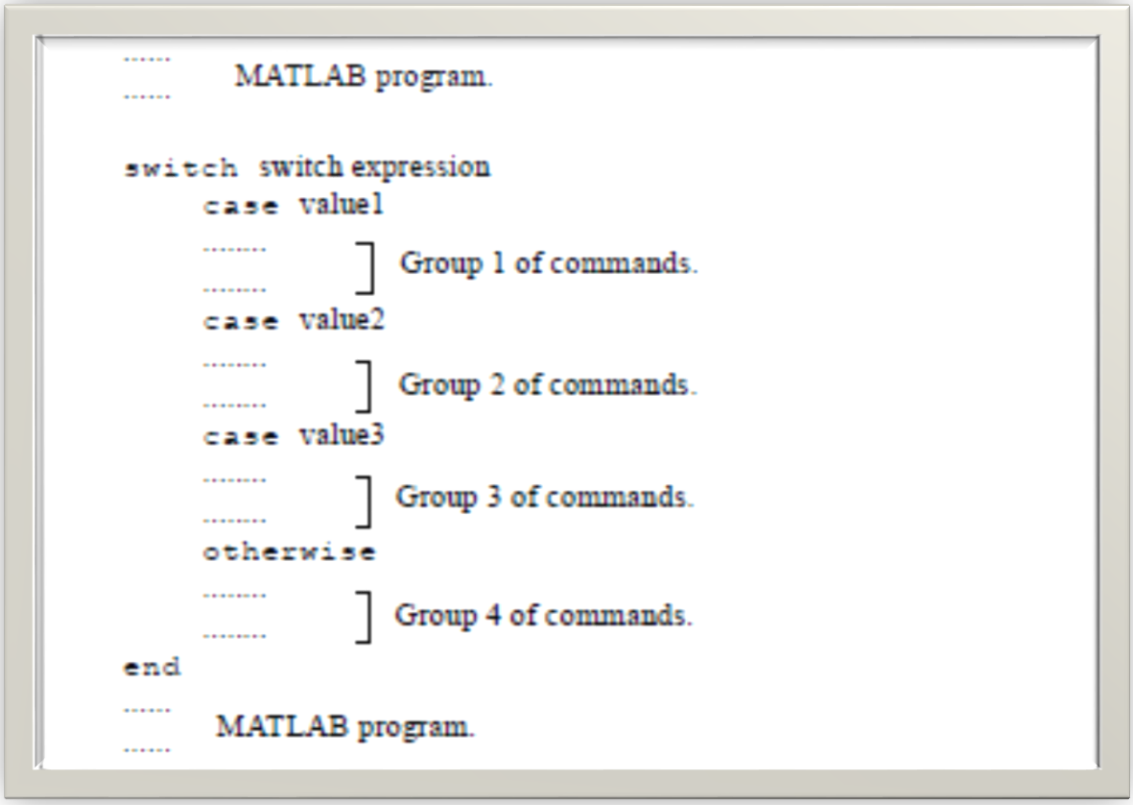


Fig (6.2): Switch structure

- If there is more than one match, only the first matching case is executed.
- If no match is found and the otherwise statement (which is optional) is present, the group of commands between otherwise and end is executed.
- If no match is found and the otherwise statement is not present, none of the command groups is executed.
- A case statement can have more than one value. This is done by typing the values in the form: {value1, value2, value3, ...}. (This form, which is not covered in this book, is called a cell array.) The case is executed if at least one of the values matches the value of switch expression.

Example

Write a program to do the following:

1-When the user enters H to draw three-dimensional helix when angle changed from 0 to 10pi in steps of pi/50

2- When the user enters S calculate the speed of the wave

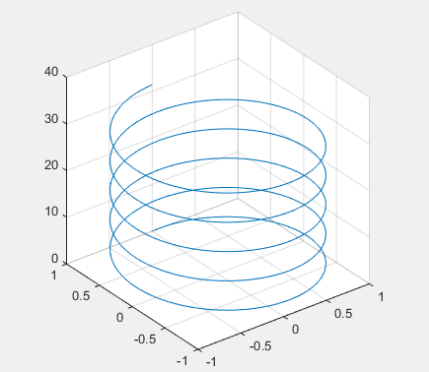
3- When the user enters A, Welcome appeared on the screen.

```
ch=input('Enter H to draw helix, enter S to calculate the speed of the wave, enter A to show welcome on the screen: ','s');
switch ch
case 'H'
  t = 0:pi/50:10*pi;
  plot3(sin(t),cos(t),t)
  grid on
  axis square
case 'S'
  x=input('Enter wavelength: ');
  y=input('Enter frequency: ');
  speed=x*y;
  fprintf('Wave speed= %f\n',speed)
case 'A'
   disp('Welcome')
end
```

% The run of program

>> test

Enter H to draw helix, enter S to calculate the speed of the wave, enter A to show welcome on the screen: W



% Another run of program

>> test

Enter H to draw helix, enter S to calculate the speed of the wave, enter A to show welcome on the screen: S

Enter wavelength: 5

Enter frequency: 4

Wave speed= 20.000000

% Another run of program

>> test

Enter H to draw helix, enter S to calculate the speed of the wave, enter A to show welcome on the screen: A

Welcome

Example

Write a program to do the following:

1-When the user enters 1 draw three-dimensional helix when angle changed from 0 to 10pi in steps of pi/50

2- When the user enters 2 calculate the speed of the wave

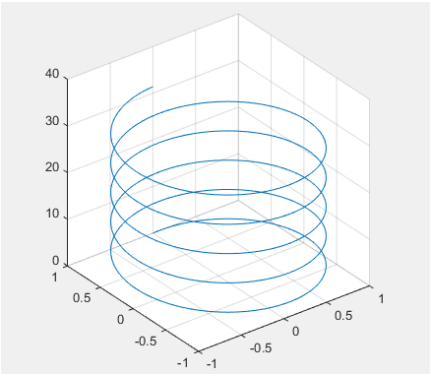
3- When the user enters 3, Welcome appeared on the screen.

```
n=input('Enter 1 to draw helix, enter 2 to calculate the speed of the wave, enter 3 to show welcome on the screen: ');
switch n
case 1
  t = 0:pi/50:10*pi;
  plot3(sin(t),cos(t),t)
  grid on
  axis square
case 2
  x=input('Enter wavelength: ');
  y=input('Enter frequency: ');
  speed=x*y;
  fprintf('Wave speed= %3.4f\n',speed)
case 3
   disp('Welcome')
end
```

% The run of program

>> test

Enter 1 to draw helix, enter 2 to calculate the speed of the wave, enter 3 to show welcome on the screen: 1



% Another run of program

>> test

Enter 1 to draw helix, enter 2 to calculate the speed of the wave, enter 3 to show welcome on the screen: 2
Enter wavelength: 5
Enter frequency: 4
Wave speed= 20.0000

% Another run of program

>> test

Enter 1 to draw helix, enter 2 to calculate the speed of the wave, enter 3 to show welcome on the screen: 3
Welcome

Loop

A loop is another method to alter the flow of a computer program. In a loop, the execution of a command, or a group of commands, is repeated several times consecutively. Each round of execution is called a pass. In each pass at least one variable, but usually more than one, or even all the variables that are defined within the loop, are assigned new values. MATLAB has two kinds of loops. In for-end loops the number of passes is specified when the loop starts. In while-end loops the number of passes is not known ahead of time, and the looping process continues until a specified condition is satisfied.

for-end *Loops*

In for-end loops the execution of a command, or a group of commands, is repeated a predetermined number of times. The form of a loop is shown in **Fig (7.1).**

• The loop index variable can have any variable name (usually i, j, k, m, and n are used, however, i and j should not be used if MATLAB is used with complex numbers).

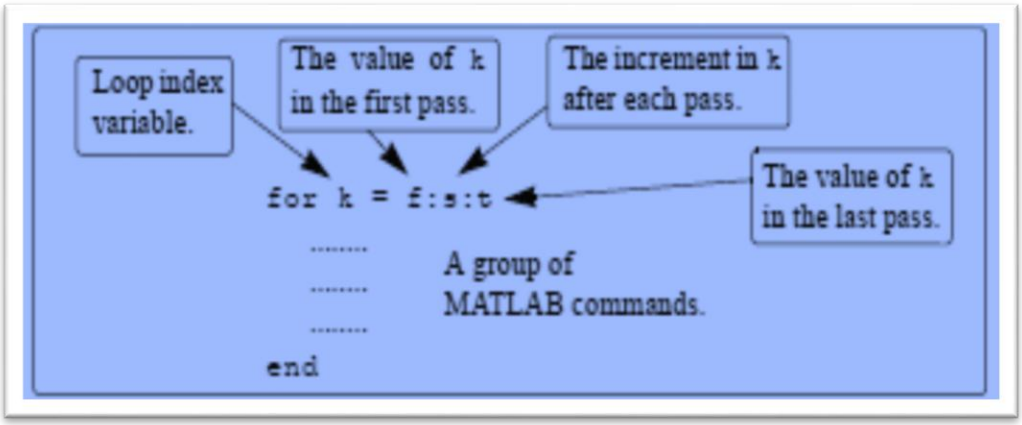


Fig (7.1): for-end loop

• In the first pass k = f and the computer executes the commands between the for and end commands. Then, the program goes back to the for command for the second pass. k obtains a new value equal to k = f + s, and the commands between the for

and end commands are executed with the new value of k. The process repeats itself until the last pass, where k = t. Then the program does not go back to the for, but continues with the commands that follow the end command. For example, if k = 1:2:9, there are five loops, and the corresponding values of k are 1, 3, 5, 7, and 9.

• The increment s can be negative (i.e.; k = 25:–5:10 produces four passes with k = 25, 20, 15, 10).

• If the increment value s is omitted, the value is 1 (default) (i.e.; k = 3:7 produces five passes with k = 3, 4, 5, 6, 7).

• If f = t, the loop is executed once.

• If f > t and s > 0, or if f < t and s < 0, the loop is not executed.

• If the values of k, s, and t are such that k cannot be equal to t, then if s is positive, the last pass is the one where k has the largest value that is smaller than t (i.e., k = 8:10:50 produces five passes with k = 8, 18, 28, 38, 48). If s is negative, the last pass is the one where k has the smallest value that is larger than t.

• In the for command k can also be assigned a specific value (typed as a vector).

Example: for k = [7 9 –1 3 3 5].

• The value of k should not be redefined within the loop.

• Each for command in a program *must* have an end command.

• The value of the loop index variable (k) is not displayed automatically. It is possible to display the value in each pass (which is sometimes useful for debugging) by typing k as one of the commands in the loop.

• When the loop ends, the loop index variable (k) has the value that was last assigned to it. A simple example of a for-end loop (in a script file) is:

```
for k=1:3:10
  x = k^2
end
```

When this program is executed, the loop is executed four times. The value of k in the four passes is k = 1, 4, 7, and 10, which means that the values that are assigned to x in the passes are x = 1, 16, 49, and 100, respectively. Since a semicolon is not typed at the end of the second line, the value of x is displayed in the Command Window at each pass. When the script file is executed, the display in the Command Window is:

```
x =     1
x =    16
x =    49
x =   100
```

Example: Sum of a series

Use a for-end loop in a script file to calculate the sum of the first *n* terms of the series: $\sum_{k=1}^{n} \frac{(-1)^k k}{2^k}$.

Solution

```
n=input('Enter number of terms: ' );
s=0;
for k=1:n
  s=s+(-1)^k*k/2^k;
end
fprintf('The sum of the series is: %f\n',s)
```

% The run of program

```
>> test
Enter number of terms: 3
The sum of the series is: -0.375000
```

Example

Calculate the series $sum = 5 - \frac{3!}{x^3} - \frac{5!}{x^5} - \frac{7!}{x^7}$

```
x=input('Enter x: ');
sum=5;
for i=3:2:7
  sum=sum- factorial(i)/ x^i;
end
fprintf('Sum= %f',sum)
```

% The run of program

```
>> test

Enter x: 2

Sum= -38.875000
```

Example

calculate the series $sum = 8 + \frac{5}{x^2} + \frac{10}{x^4} + \frac{15}{x^8}$

```
x=input('Enter x: ');
sum=8;
j=2;
for i=5:5:15
 sum=sum+i/x^j;
 j=j*2;
end
fprintf('Sum= %f\n',sum)
```

% The run of program

```
>> test
Enter x: 3
Sum= 8.681299
```

Example

calculate the series $sum = 8 - \frac{5}{x^2} + \frac{10}{x^4} - \frac{15}{x^8}$

```
x=input('Enter x: ');
sum=8;
j=2;
s=-1;
for i=5:5:15
  sum=sum+s* i/x^j;
 j=j*2;
s=-s;
end
fprintf('Sum= %f\n',sum)

% The run of program
>> test
Enter x: 3
Sum= 7.565615
```

while-end *Loops*

while-end loops are used in situations when looping is needed but the number of passes is not known in advance. In while-end loops the number of passes is not specified when the looping process starts. Instead, the looping process continues until a stated condition is satisfied. The structure of a while-end loop is shown in **Fig (7.2)**.

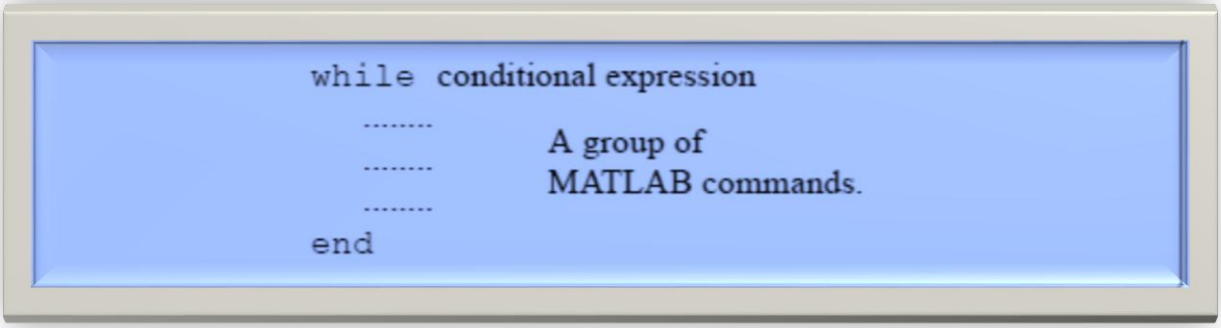


Fig (7.2): while-end loop

The first line is a while statement that includes a conditional expression. When the program reaches this line the conditional expression is checked. If it is false (0), MATLAB skips to the end statement and continues with the program. If the conditional expression is true (1), MATLAB executes the group of commands that follow between the while and end commands. Then MATLAB jumps back to the while command and checks the conditional expression. This looping process continues until the conditional expression is false.

Example **Taylor series representation of a function**

The function $f(x) = e^x$ can be represented in a Taylor series by $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$

Write a program in a script file that determines e^x by using the Taylor series representation. The program calculates e^x by adding terms of the series and

stopping when the absolute value of the term that was added last is smaller than 0.0001.Use a while-end loop, but limit the number of passes to 30. If in the 30th pass the value of the term that is added is not smaller than 0.0001, the program stops and displays a message that more than 30 terms are needed.

Use the program to calculate e^2, e^{-4}, and e^{21}.

Solution

The first few terms of the Taylor series are:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

A program that uses the series to calculate the function is shown next. The program asks the user to enter the value of x. Then the first term, an, is assigned the number 1, and an is assigned to the sum S. Then, from the second term on, the program uses a while loop to calculate the *n*th term of the series and add it to the sum. The program also counts the number of terms n. The conditional expression in the while command is true as long as the absolute value of the *n*th an term is larger than 0.0001, and the number of passes n is smaller than 30. This means that if the 30th term is not smaller than 0.0001, the looping stops.

```
x=input('Enter x: ' );
n=1; an=1; s=an;
while abs(an) >= 0.0001 & n <= 30
  an=x^n/factorial(n);
  s=s+an;
  n=n+1;
end
if n >= 30
 disp('More than 30 terms are needed')
else
 fprintf('exp(%3.4f) = %3.4f',x,s)
 fprintf('\nThe number of terms used is: %i\n',n)
end
```

% The run of program

>> test

Enter x: 4

exp(4.0000) = 54.5981

The number of terms used is: 18

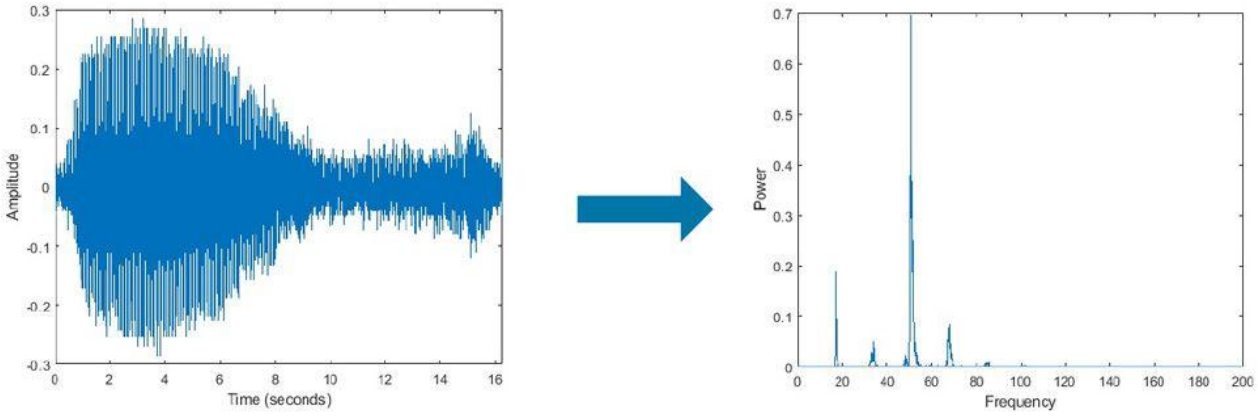
Signal Processing

In information theory, a signal is a codified message, that is, the sequence of states in a communication channel that encodes a message. In a communication system, a transmitter encodes a message to create a signal, which is carried to a receiver by the communication channel.

Signal processing involves converting or transforming data in a way that allows us to see things in it that are not possible via direct observation. Signal processing allows engineers and scientists to analyze, optimize, and correct signals, including scientific data, audio streams, images, and video.

Fast Fourier Transform (FFT)

A **Fast Fourier Transform (FFT)** is a highly optimized implementation of the discrete Fourier transform (DFT), which convert discrete signals from the time domain to the frequency domain. FFT computations provide information about the frequency content, phase, and other properties of the signal.



Blue whale moan audio signal decomposed into its frequency components using FFT.

A wave is a disturbance that travels or propagates from the place where it was created.

Wavelength: The distance travelled by a wave in one complete oscillation is called as wavelength. The SI unit of wavelength is meter (m) and it is denoted by a Greek letter lambda (λ).

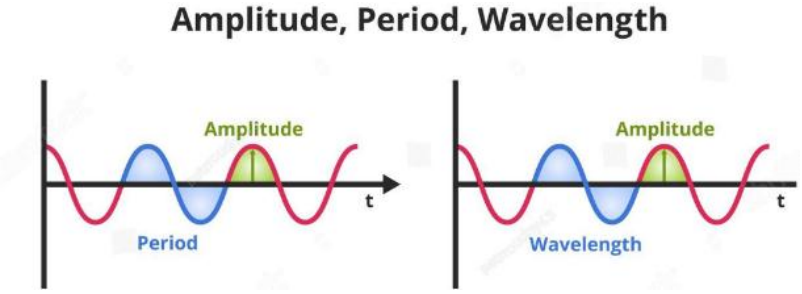
Frequency: The number of oscillations completed in one second is called as frequency of a wave. Frequency can also be described as the number of waves that pass a point in one second. **f = 1 / t**

Amplitude: distance between the resting position and the maximum displacement of the wave.

Displacement: is defined as the change in position of an object. It is a vector quantity and has a direction and magnitude.

The magnitude of displacement is defined as the shortest distance between the initial and final position of the object. For a particle in motion, the magnitude is either less than or equal to the distance travelled.

Period: time it takes for one wave cycle to complete.



Fourier Transforms

The Fourier transform is a mathematical formula that transforms a signal sampled in time or space to the same signal sampled in temporal or spatial frequency. In signal processing, the Fourier transform can reveal important characteristics of a signal, namely, its frequency components.

The Fourier transform is defined for a vector x with n uniformly sampled points by

$$y_{k+1} = \sum_{j=0}^{n-1} \omega^{jk} x_{j+1}.$$

$\omega = e^{-2\pi i/n}$ is one of the n complex roots of unity where i is the imaginary unit. For x and y, the indices j and k range from 0 to $n-1$.

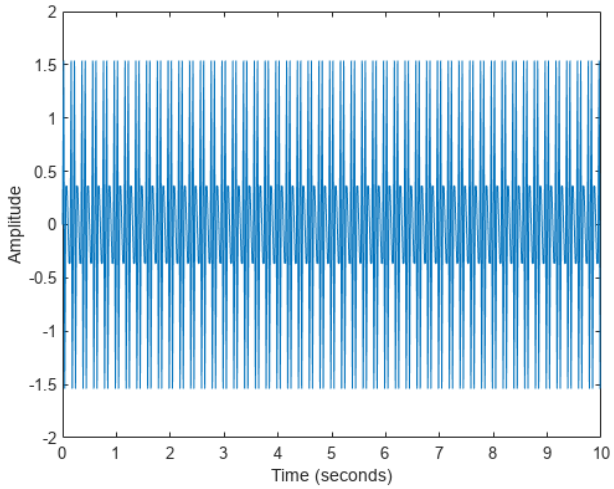
The fft function in MATLAB® uses a fast Fourier transform algorithm to compute the Fourier transform of data. Consider a sinusoidal signal x that is a function of time t with frequency components of 15 Hz and 20 Hz. Use a time vector sampled in increments of 1/50 seconds over a period of 10 seconds.

x = A sin (ωt + ϕ) or x = A cos (ωt + ϕ)

Here,

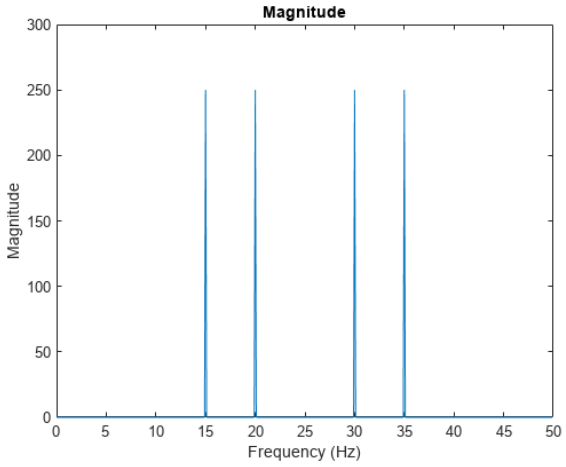
- x = displacement of wave (meter)
- A = amplitude
- ω = angular frequency (rad/s)
- t = time period
- ϕ = phase angle

```
Ts = 1/50;

t = 0:Ts:10-Ts;

x = sin(2*pi*15*t) + sin(2*pi*20*t);

plot(t,x)

xlabel('Time (seconds)')

ylabel('Amplitude')
```



Compute the Fourier transform of the signal, and create the vector f that corresponds to the signal's sampling in frequency space.

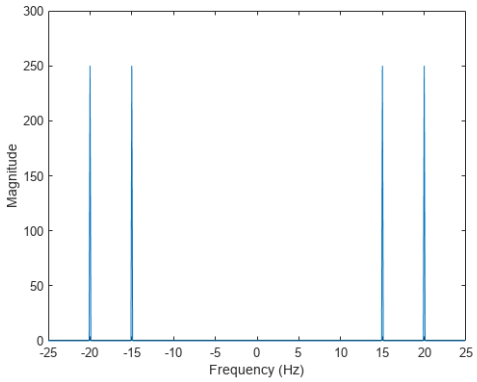
```
Ts = 1/50;

t = 0:Ts:10-Ts;

x = sin(2*pi*15*t) + sin(2*pi*20*t);

y = fft(x); %computes the discrete Fourier transform (DFT) of X

fs = 1/Ts;

f = (0:length(y)-1)*fs/length(y); %returns the size of the longest dimension of  vector

plot(f,abs(y)) %Plot the magnitude of the transformed signal as a function of frequency.

xlabel('Frequency (Hz)')

ylabel('Magnitude')

title('Magnitude')
```



The plot shows four frequency peaks, although the signal is expected to have two frequency peaks at 15 Hz and 20 Hz. Here, the second half of the plot is the mirror reflection of the first half. The discrete Fourier transform of a time-domain signal has a periodic nature, where the first half of its spectrum is in the positive frequencies and the second half is in the negative frequencies. The 30 Hz and 35 Hz frequency components in

the plot correspond to the –20 Hz and –15 frequency components. To better visualize this periodicity, you can use the fft shift function, which performs a zero-centered, circular shift on the transform.

```
Ts = 1/50;

t = 0:Ts:10-Ts;

x = sin(2*pi*15*t) + sin(2*pi*20*t);

y = fft(x);

fs = 1/Ts;

n = length(y);

fshift = (-n/2:n/2-1)*(fs/n);

% fftshift(y) rearranges a Fourier transform X by shifting
%the zero-frequency component to the center of the array.

yshift = fftshift(y);

plot(fshift,abs(yshift))

xlabel('Frequency (Hz)')

ylabel('Magnitude')
```



Noisy Signals

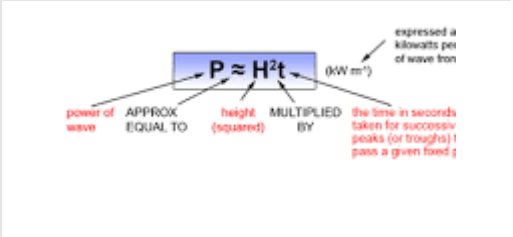
In scientific applications, signals are often corrupted with random noise, disguising their frequency components. The Fourier transform can process out random noise and reveal the frequencies. For example, create a new signal, xnoise, by injecting Gaussian noise into the original signal, x.

%Rng function controls the global stream, which determines how the rand, randi, randn, and %randperm **functions** produce a sequence of random numbers.

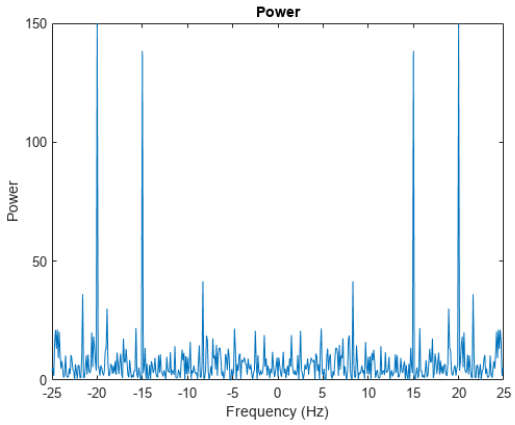
```
rng('default')
xnoise = x + 2.5*randn(size(t));  % returns a random scalar drawn from the standard normal distribution.
```

Signal power as a function of frequency is a common metric used in signal processing. Power is the squared magnitude of a signal's Fourier transform, normalized by the number of frequency samples. Compute and plot the power spectrum of the noisy signal centered at the zero frequency. Despite noise, you can still make out the signal's frequencies due to the spikes in power.

```
Ts = 1/50;
t = 0:Ts:10-Ts;
x = sin(2*pi*15*t) + sin(2*pi*20*t);
fs = 1/Ts;
n = length(x);
fshift = (-n/2:n/2-1)*(fs/n);
rng('default')
xnoise = x + 2.5*randn(size(t));
ynoise = fft(xnoise);
```



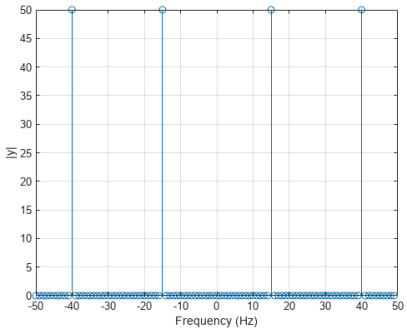
```
ynoiseshift = fftshift(ynoise);

power = abs(ynoiseshift).^2/n;

plot(fshift,power)

title('Power')

xlabel('Frequency (Hz)')

ylabel('Power')
```



Phase of Sinusoids

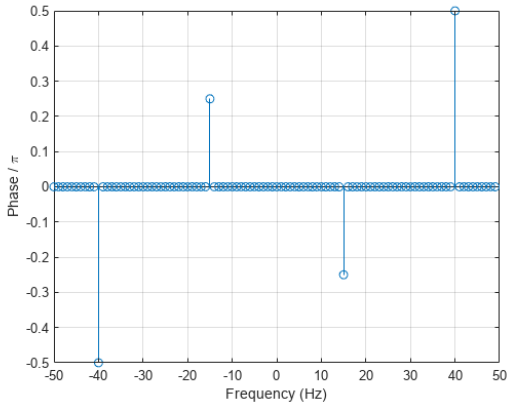
Using the Fourier transform, you can also extract the phase spectrum of the original signal. For example, create a signal that consists of two sinusoids of frequencies 15 Hz and 40 Hz. The first sinusoid is a cosine wave with phase $-\pi/4$, and the second is a cosine wave with phase $\pi/2$. Sample the signal at 100 Hz for 1 second.

```
fs = 100;
t = 0:1/fs:1-1/fs;
x = cos(2*pi*15*t - pi/4) - sin(2*pi*40*t);
%Compute the Fourier transform of the signal. Plot the magnitude of the transform as a
%function of frequency.

y = fft(x);

z = fftshift(y);

ly = length(y);

f = (-ly/2:ly/2-1)/ly*fs;

stem(f,abs(z))

xlabel('Frequency (Hz) ')

ylabel('|y|')

grid
```



Compute the phase of the transform, removing small-magnitude transform values. Plot the phase as a function of frequency.

```
fs = 100;

t = 0:1/fs:1-1/fs;

x = cos(2*pi*15*t - pi/4) - sin(2*pi*40*t);

y = fft(x);

z = fftshift(y);

ly = length(y);

f = (-ly/2:ly/2-1)/ly*fs;

tol = 1e-6;

z(abs(z) < tol) = 0;

theta = angle(z); % returns the phase angle in the interval [-π,π]

stem(f,theta/pi)

xlabel('Frequency (Hz)')

ylabel('Phase / \pi')

grid
```



Popular FFT algorithms include the Cooley-Tukey algorithm, prime factor FFT algorithm, and Rader's FFT algorithm. The most commonly used FFT algorithm is the Cooley-Tukey algorithm, which reduces a large Discrete Fourier Transform DFT into smaller DFTs to increase computation speed and reduce complexity. FFT has applications in many fields.

Signal Processing Filtering

In signal processing, a **filter** is a device or process that removes some unwanted components or features from a signal. Filtering is a class of signal processing, the defining feature of filters being the complete or partial suppression of some aspect of the signal. Most often, this means removing some frequencies or frequency bands. However, filters do not exclusively act in the frequency domain; especially in the field of image processing many other targets for filtering exist. Correlations can be removed for certain frequency components and not for others without having to act in the frequency domain. Filters are widely used in electronics and telecommunication, in radio, television, audio recording, radar, control systems, music synthesis, image processing, computer graphics, and structural dynamics.

There are many different bases of **classifying filters** and these overlap in many different ways; there is no simple hierarchical classification. Filters may be:

- non-linear or linear
- time-variant or time-invariant, also known as shift invariance. If the filter operates in a spatial domain then the characterization is space invariance.
- causal or non-causal: A filter is non-causal if its present output depends on future input. Filters processing time-domain signals in real time must be causal, but not filters acting on spatial domain signals or deferred-time processing of time-domain signals.
- analog or digital

- discrete-time (sampled) or continuous-time
- passive or active type of continuous-time filter
- infinite impulse response (IIR) or finite impulse response (FIR) type of discrete-time or digital filter.

Linear continuous-time filters

Linear continuous-time circuit is perhaps the most common meaning for filter in the signal processing world, and simply "filter" is often taken to be synonymous. These circuits are generally designed to remove certain frequencies and allow others to pass. Circuits that perform this function are generally linear in their response, or at least approximately so. Any nonlinearity would potentially result in the output signal containing frequency components not present in the input signal.

The modern design methodology for linear continuous-time filters is called network synthesis. Some important filter families designed in this way are:

- Chebyshev filter, has the best approximation to the ideal response of any filter for a specified order and ripple.
- Butterworth filter, has a maximally flat frequency response.
- Bessel filter, has a maximally flat phase delay.
- Elliptic filter, has the steepest cutoff of any filter for a specified order and ripple.

The difference between these filter families is that they all use a different polynomial function to approximate to the ideal filter response. This results in each having a different transfer function.

Another older, **less-used methodology is the image parameter method. Filters designed by this methodology are archaically called "wave filters".** Some important filters designed by this method are:

- Constant k filter, the original and simplest form of wave filter.
- m-derived filter, a modification of the constant k with improved cutoff steepness and impedance matching.

Signal Analysis methods

Frequency spectrum is limited but applications in mobile, wireless and satellite communications are becoming ever more diverse. Monitoring and intercepting signals across wide frequency ranges in different signal scenarios is very challenging. Comprehensive signal analysis of unknown and complex emissions also demands - great - effort. With thousands of signals occupying the frequency spectrum, the mission is to monitor all target signals, detect signals of interest and identify unknowns.

Efficient and advanced signal monitoring solutions enable surveillance of spectrum segments to detect emissions of interest which are then identified, classified and further processed (recorded, demodulated, decoded and analyzed).

- Signal surveillance: Observing (targeted monitoring) the occurrence and behavior of specific signals for situational awareness in specific signal scenarios. Surveillance systems must be able to measure unknown or “unfriendly” transmissions and extract information.
- Signal interception: Searching for, detecting, recording and reporting all signals of interest in a given scenario, often including content extraction with demodulators and decoders.
- Signal analysis: Determining technical signal parameters with automatic or manual measurements of live or recorded signals. This might also include using demodulators and decoders to resolve content from unknown signals.

Image Processing

Image is an array, or a matrix, of square pixels (picture elements) arranged in columns and rows

Image processing is the process of transforming an image into a digital form and performing certain operations to enhance or extract some useful information from it. The image processing system usually treats all images as 2D signals when applying certain predetermined signal processing methods.

Image representation

Image representation refers to how visual data is translated into a digital format that computers can interpret. It involves capturing information about the color, shape, texture, and other visual characteristics of an image and encoding them in a structured way. The chosen representation format determines how the image is stored, processed, and displayed. In Matlab, images are typically represented as matrices. A grayscale image is a 2D matrix where each element corresponds to the brightness of a pixel, ranging from 0 (black) to 255 (white). Color image are represented as 3D matrices, with the 3rd dimension corresponding to the RGB (Red, Green, Blue) color channels each having values ranging from 0 to 255.

Image Formats

Matlab supports varies images formats including JPEG, PNG, BMP, and TIFF. The imread function is used to import these images into the workspace, while imwrite is used to save processed images.

Table 1.1 Common image formats and their associated properties

Acronym	Name	Properties
GIF	Graphics interchange format	Limited to only 256 colours (8-bit); lossless compression
JPEG	Joint Photographic Experts Group	In most common use today; lossy compression; lossless variants exist
BMP	Bit map picture	Basic image format; limited (generally) lossless compression; lossy variants exist
PNG	Portable network graphics	New lossless compression format; designed to replace GIF
TIF/TIFF	Tagged image (file) format	Highly flexible, detailed and adaptable format; compressed/uncompressed variants exist

Image Filtering

Filtering is a technique for modifying or enhancing an image. For example, you can filter an image to emphasize certain features or remove other features. Image processing operations implemented with filtering include smoothing, sharpening, and edge enhancement.

The Purpose of smoothing is to reduce noise and improve the visual quality of the image. A variety of algorithms i.e. [linear] and [nonlinear] algorithms are used for filtering the images. Image filtering makes possible several useful tasks in image processing. A filter can be applied to reduce the amount of unwanted noise in a particular image as shown in fig. Another type of filter can be used to reverse the effects of blurring on a particular picture. Nonlinear filters have quite different behavior compared to linear filters. For nonlinear filters, the filter output or response of the filter does not obey the principles outlined earlier, particularly scaling and shift invariance. Moreover, a nonlinear filter can produce results that vary in a non-intuitive manner. Defected image Real Image Fig 1- A Defected image and real image after applying filtering

Fig 1- A Defected image and real image after applying filtering

Image Processing with MATLAB

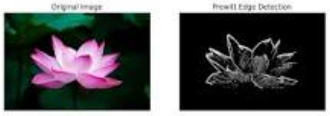
1. Importing (Image Acquisition) and exporting images.
2. Enhancing images.
3. Detecting edges and shapes.
4. Segmenting objects based on their color and texture.
5. Modifying objects' shape using morphological operations.
6. Measuring shape properties.
7. Performing batch analysis over sets of images

Image Acquisition

Import and visualize different image types in MATLAB. Manipulate images for streamlining subsequent analysis steps.

Image enhancement is the process of adjusting digital images so that the results are more suitable for display or further image analysis. For example, you can remove noise, sharpen, or brighten an image, making it easier to identify key features.

Edge detection is a fundamental image processing technique for identifying and locating the boundaries or edges of objects in an image.



Shape detection is an important part of Image Processing referring to modules that deal with identifying and detecting shapes of parts of image which differ in brightness, color or texture.

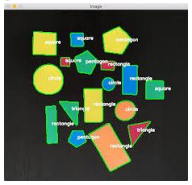
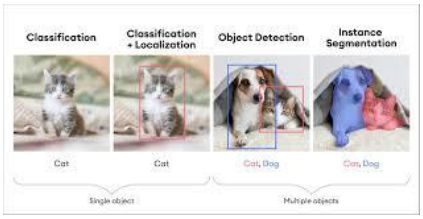
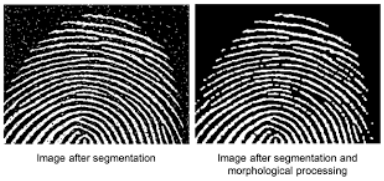


Image Segmentation in Image Processing: divide a digital image into meaningful parts, like partitioning it into different regions containing pixels with similar characteristics. This simplifies the image and allows for a more focused analysis of specific objects or areas of interest.



Morphological operations are a set of techniques used in image processing to analyze and modify the shape and structure of objects within an image. In a morphological operation, each pixel in the image is adjusted based on the value of other pixels in its neighborhood.

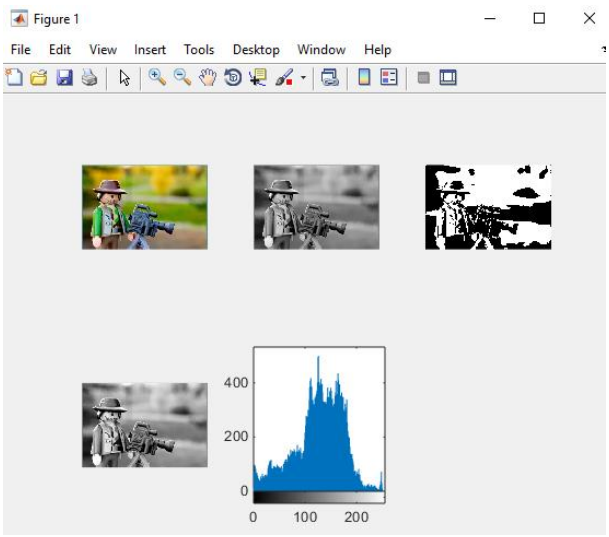


Shape measurement refers to the process of quantifying and describing the geometrical characteristics of objects, particularly their shape

The Image Batch Processor app enables you to process multiple images using the same function.

Example:

```
a = imread('cameraman.jpg');
subplot(2,3,1);
imshow(a);

b = rgb2gray(a);   % convert RGB image into grayscale
subplot(2,3,2);
imshow(b);

c = im2bw(a); %convert the image into binary
subplot(2,3,3);
imshow(c);

d = imadjust(b); % maps the intensity values in grayscale image b to new values in d
subplot(2,3,4);
imshow(d);

e = a;
e=rgb2gray(e);
subplot(2,3,5);
imhist(e);          %the histogram of the image
```



Example

```
A=imread('flowers.png');          %read in image
h=fspecial('motion',10,45);   %define motion filter
C=imfilter(A,h,'replicate');   %convolve motion
subplot(2,1,1)
imshow(A)         %display original image
subplot(2,1,2)
imshow(C,[])      %display filtered image
```

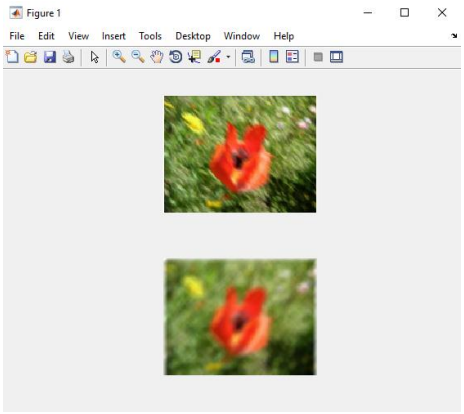


Image Transformation

Image transformation refers to the process of changing or distorting an image using operations such as rotation, scaling, shearing, reflection, or projection

Rotation a process in computer science that involves rotating an image by a specified angle around a given point. This operation is commonly used in image processing for tasks such as matching and alignment.

scaling is the process of resizing an image. We deal with the dimensions of an image. Scaling down deals with making image smaller while Scaling up refers to increase the size of image.

Shearing is a geometric augmentation that changes a form of an image along a specific axis to create a different perception angle.

Reflection transformation in image processing is a geometric operation that involves flipping an image across a specific axis. The reflection can be done horizontally, vertically, or diagonally, resulting in a mirrored version of the original image.

Projection: Representing an n-dimensional object into an n-1 dimension is known as projection. It is process of converting a 3D object into 2D object, we represent a 3D object on a 2D plane {(x,y,z)->(x,y)}. It is also defined as mapping or transforming of the object in projection plane or view plane.

Digital Image Processing system

In digital image processing, we will develop a system that whose input would be an image and output would be an image too. And the system would perform some processing on the input image and gives its output as a processed image. It is shown below.

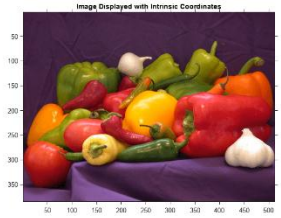
Now function applied inside this digital system that process an image and convert it into output can be called as transformation function.

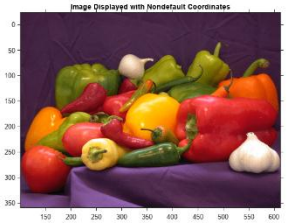
As it shows transformation or relation, that how an image1 is converted to image2.

Shift X- and Y-Coordinate Range of Displayed Image

This example shows how to specify a nondefault world coordinate system by changing the XData and YData properties of a displayed image.

```
%Read an image.

I = imread('peppers.png');
%Display the image using the intrinsic coordinate system, returning properties of the image in ax. Turn on the axis to display the coordinate system.

ax = imshow(I);
title('Image Displayed with Intrinsic Coordinates')
axis on
%Check the range of the x- and y-coordinates, which are stored
in the XData and YData properties of ax. The ranges match the dimensions of the image.

xrange = ax.XData
yrange = ax.YData
%Change the range of the x- and y-coordinates. This example shifts the image to the right by adding 100 to the x-coordinates and shifts the image up by subtracting 25 from the y-coordinates.

xrangeNew = xrange + 100;
yrangeNew = yrange - 25;
%Display the image, specifying the shifted spatial coordinates.

figure
axNew = imshow(I, 'XData',xrangeNew, 'YData',yrangeNew);
title('Image Displayed with Nondefault Coordinates');
axis on
```





Rotation Example

```
%% Step 1: Read Image
% Bring an image into the workspace.
original = imread('cameraman.tif');
imshow(original);
text(size(original,2),size(original,1)+15, ...
    'Image courtesy of Massachusetts Institute of Technology', ...
    'FontSize',7,'HorizontalAlignment','right');

%% Step 2: Resize and Rotate the Image

scale = 0.7;
J = imresize(original, scale); % Try varying the scale factor.

theta = 30;
distorted = imrotate(J,theta); % Try varying the angle, theta.
figure; imshow(distorted)
```