MATLAB is a software package for high-performance mathematical computation, visualization, and programming environment. It provides an interactive environment with hundreds of built-in functions for technical computing, graphics, and animations.

MATLAB stands for **Matrix Laboratory**. MATLAB was written initially to implement a simple approach to matrix software developed by the **LINPACK** (Linear system package) and **EISPACK** (Eigen system package) projects.

MATLAB is a modern programming language environment, and it has refined data structures, includes built-in editing and debugging tools, and supports object-oriented programming.

MATLAB is **Multi-paradigm**. So, it can work with multiple types of programming approaches, such as Functional, Object-Oriented, and Visual.

Besides an environment, MATLAB is also a programming language.

As its name contains the word Matrix, MATLAB does its' all computing based on mathematical matrices and arrays. MATLAB's all types of variables hold data in the form of the array only, let it be an integer type, character type or String type variable.

MATLAB's built-in functions provide excellent tools for linear algebra computations, data analysis, signal processing, optimization, numerical solution of ordinary differential equations **(ODEs)**, quadrate, and many other types of scientific calculations.

Most of these functions use state-of-the-art algorithms. These are numerous functions for 2-D and 3-D graphics, as well as for animations.

MATLAB supports an external interface to run those programs from within MATLAB. The user is not limited to the built-in functions; he can write his functions in the MATLAB language.

There are also various optional "**toolboxes**" available from the developers of MATLAB. These toolboxes are a collection of functions written for primary applications such as symbolic computations, image processing, statistics, control system design, and neural networks.

The necessary building components of MATLAB are the **matrix**. The fundamental data type is the array. Vectors, scalars, real matrices, and complex matrices are all automatically handled as special cases of the primary data type. MATLAB loves matrices and matrix functions. The built-in functions are optimized for vector functions. Therefore, Vectorised commands or codes run much faster in MATLAB.

**Development Environment**

This is the set of tools and facilities that help you use MATLAB operations and files. Many of these tools are the graphical user interface. It involves the MATLAB desktop and command window, a command history, an editor and debugger, and browsers for considering help, the workspace, reports, and the search path.

**MATLAB Mathematical Function Library**

This is a vast compilation of computing design ranging from basic functions, like sum, sine, cosine, and complex mathematic, to more sophisticated features like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms.

**MATLAB Language**

This is a high level matrix/array language with control flow statement, function, data structure, input/output, and object-oriented programming characteristics. It allows both "programming in the small" to create quick and dirty throw-away programs rapidly and "programming in the large" to create large and complex application functions.

**Graphics**

MATLAB has extensive facilities for displaying vector and matrices as graphs, as well as annotating and printing these graphs. It contains high-level structures for two-dimensional and three-dimensional data visualization, image processing, animation, and presentation graphics. It also involves low-level structures that allow us to customize the display of graphics fully as well as to build complete graphical user interfaces on our MATLAB applications.

**MATLAB External Interfaces/API**

This is a library that allows us to write C and FORTRAN programs that interact with MATLAB. It contains facilities for calling routines from MATLAB (dynamic linking), calling MATLAB as a computational engine, and for reading and writing MAT-files.

## **MATLAB Features**

As there are numerous features to describe, but here, we will focus on some of the key features:

* It is designed for numerical as well as symbolic computing.
* It's a high-level language used mainly for engineering and scientific computing.
* It works within a Desktop environment providing full features for iterative exploration, design, and problem-solving.
* Creation of custom plots for visualizing data and tools, with the help of built-in Graphics.
* Specific applications are designed to work with any particular type of problems, such as data classification, control system design and tuning, signal analysis.
* Provides several add-on toolboxes to build a wide range of engineering, scientific, and custom user interface applications.
* Provide interfaces to work with other programming languages such as C, C++, Java, .NET, Python, SQL, and Hadoop.

## **MATLAB Desktop**

The main tools within or accessible from the MATLAB desktop are

* Command Window
* Command History Window
* Start Button
* Documents Window, containing the Editor/Debugger and the Array Editor
* Figure Windows
* Workspace Browser
* Help Browser
* Path Browser

## **Initializing Variables in Assignment Statement**

The simplest method to initialize a variable is to assign it one or more value in an assignment statement.

An assignment statement has the standard form

                  var = expression;

where var is the name of the variables and expression is a scalar constant, an array, or a combination of constants, other variables, and mathematical operations (+, -, etc.). The value of the expression is computed using the standard rules of mathematics, and the resulting values are saved in the named variable. The semicolon at the last of the statement is optional. If the semicolon is absent, the values assigned to var will be echoed in the command window. If it is present, nothing will be shown in the Command Window, even though the assignment has appeared.

Examples of initializing variables with assignment statements contain

                  var = 40i;  
                  var2 = var/5;  
                  x = 1; y = 2;  
                  array = [1 2 3 4];

The first example generates a scalar variable of type double and saves the imaginary number 40i in it.

The second example generates a scalar variable and saves the result of the expression var/5 in it.

The third example shows that multiple assignment statements can be placed on a single line, supported that they are divided by semicolons or commas.

The last example display that variables can also be initialized with arrays of data. Such arrays are build up using brackets ([]) and semicolons. All of the items of an array are listed in row order. In other words, the value in each row are recorded from left to the right, with the top-most row first, and the bottom-most row last. The single value within a row are separated by blank spaces or commas, and the rows themselves are divided by semicolons or newlines.

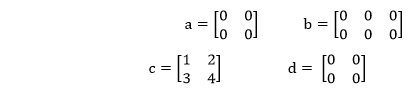
## **Initializing with Built-In Functions**

Arrays can also be initialize using built-in MATLAB function. For example, the function zero can be used to generate an all-zero array of any desired size. There are a various form of the zeros function. If the function has an individual scalar argument, it will develop a square array using the single arguments as both the number of rows and the number of columns. If the function has two scalar argument, the first arguments will be the number of rows, and the second arguments will be the number of the columns. Since the size function return two values including the number of row and column in an array, it can be combined with the zero function to create an array of zeros that is the same size of another array.

Some examples using the zeros function follow:

                  a = zeros(2);  
                  b = zeros(2,3);  
                  c = [1 2; 3 4];  
                  d = zeros(size(c));

These statements generate the following arrays:



Similarly, the **ones** function can be used to generate array including all ones, and the eye function can be used to generate arrays including **identity matrices**, in which all on-diagonal items are one, while all off-diagonal items are zero.

|  |  |
| --- | --- |
| zeros(n) | Creates a n x n matrix of zeros. |
| zeros(m,n) | Creates a m x n matrix of zeros |
| zeros(size(arr)) | Create a matrix of zeros of the same size as arr. |
| ones(n) | Creates a n x n matrix of ones. |
| ones(m,n) | Creates a m x n matrix of ones. |
| ones(size(arr)) | Creates a matrix of ones of the same size as arr. |
| eye(n) | Creates a n x n identity matrix. |
| eye(m,n) | Creates an m x n identity matrix. |
| length(arr) | Return the length of a vector, or the longest dimension of a 2-D array. |
| size(arr) | Return two values specifying the number of rows and columns in arr. |

# MATLAB Plotting

## **Creating Plotting**

MATLAB makes it easy to create plots. For example in 2D, is to take a vector of **a**- coordinates, **a** = (a1... an ), and a vector of **b**-coordinates, **b** = (b1...bn), locate the points (ai...bi), with i = 1, 2. . . n and then connect them by straight lines.

The MATLAB commands to plot a graph is plot (a, b).

The vectors a = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) and b = (0, 1, -1, 1, 0) produce the picture shown in figure.

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

>> b = [0, 1, -1, 1, 0];

>> plot(a, b)

## 

## **fplot()**

# MATLAB 2-D Plots

There are many specific graphics functions for 2-D plotting. They are used as alternatives to the plot **command** we have just discussed. MATLAB help desk lists more than 20 types of two-dimensional plots. Examples include stem plots, stair plots, bar plots, pie plots, and compass plots.

Here we are going to discuss the 2D plotting function in MATLAB.

## **MATLAB fplot()**

It is used to plot between the specific limit. The function must be of form y=f(x), where x is the vector whose specifies the limits, and y is the vector with the same size as x.

### **Syntax**

fplot(fun, limits) //   A function fun is plotted in between the limit specified

fplot(fun, limits, linespace) // It allows plotting fun with line specification

fplot(fun, limits, tol) //  It allows plotting with relative error tolerance 'tol'.If not specified default tolerance will be 2e-3 ie .2%  accuracy.

fplot(fun, limits, tol, linespace)//     It allows plotting with relative tolerance and line specification

### **Example**

f(t)=t sin t, 0≤t≤10π

fplot ('x.\*sin(x)',[0 10\*pi])

# Bar()

A bar plot is a plot in which each point is represented by a vertical bar or horizontal bar.

### **Syntax**

bar(y) // It creates a bar graph with one bar for each element in y.

bar (x, y) // This function creates a vertical bar plot, with the values in x used to label each bar and the values in y used to determine the height of the bar.

### **Example**

Create Bar Graph

r^2=2 sin 5t, 0≤t≤2π

y = r sin t

t=linspace (0, 2\*pi,200);

r=sqrt(abs(2\*sin(5\*t)));

y=r.\*sin(t);

bar (t, y)

axis ([0 pi 0 inf]);

# Pie()

This function creates a pie plot. This function determines the percentage of the total pie corresponding to each value of x, and plots pie slices of that size.

The optional array **explodes** controls whether or not individual pie slices are separated from the remainder of the pie.

### **Syntax**

pie(x) // It draws a pie chart with the data in x.

pie(x, explode) // It offsets a slice from the pie.explode is a vector or matrix of zeroes and non-zeroes corresponding to x.

### **Example**

World population by continents.

cont= **char**('Asia','Europe','Africa',....'N.America','S.America');

pop=[3332;696;694;437;307];

pie(pop)

**for** i=1:5,

    gtext (cont(i,:));

end

Title ('World Population (1992)',....'fontsize', 18)

# hist()

A histogram is a plot presenting the distribution of values within a data set. To develop a histogram, the range of values within the data set is split into evenly spaced bins, and the number of data values falling into each bin is determined.

### **Syntax**

n=hist(y) // It bins the elements in vector y into ten equally spaced containers and returns the number of items in each container as a row vector.

### **Example**

Histogram of 50 randomly distributed numbers between 0 and 1.

y=randn (50, 1);

hist (y)

# stem()

A two-dimensional stem plot shows data as lines extending from a baseline along the x-axis. A circle (the default) or another marker whose y-position represents the data value terminates each stem.

### **Syntax**

stem(Y) // It plots the data sequence Y as stems that extends from equally spaced and automatically created values along the x-axis. When Y is a matrix, stem plot, all items in a row against the same x value.

stem(X,Y) // It plot X versus the column of Y. X and Y are vectors or matrices of a similar size. X can be the row or a column vector, and Y is a matrix with length(X) rows.

stem(...,'fill') // It specifies whether to color the circle at the end of the stem.

stem(...,LineSpec) // It specifies the line style, marker symbol, and color.

h = stem(...) // It returns a vector of Stem objects in h.

### **Example**

f=e^-t/5 sint,0≤t≤2π

t=linspace (0, 2\*pi, 200);

f=exp (-.2\*t).\*sin(t);

stem(t, f)

## MATLAB 3D Plots

# plot3()

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

### **Syntax**

plot3(X1,Y1,Z1,...) // where X1, Y1, Z1 are vectors or matrices, plot one or more lines in 3-D space through the points whose points are the items of X1, Y1, and Z1.

plot3(X1,Y1,Z1,LineSpec,...) // It develop and shows all the lines described by the Xn,Yn,Zn,LineSpec quads, where LineSpec is a line specification that define line style, marker symbol, and color of the plotted lines.

plot3(...,'PropertyName',PropertyValue,...) // It sets properties to the specified property values for all Line graphics objects generated by plot3.

h = plot3(...) // It returns a column vector of handles to line graphics objects, with one handle per line.

### **Example**

Plot of a parametric space curve:

 x(t)=t,y(t)=t^2,z(t)=t^3.

0≤t≤1

t= linspace (0, 1,100);

x=t; y=t.^2; z=t.^3;

plot3(x, y, z), grid

xlabel ('x(t)=t')

ylabel ('y(t)=t2')

zlabel ('z(t)=t3')

# surfc()

surfc develop colored parametric surfaces specified by X, Y, and Z, with the color specified by Z or C.

### **Syntax**

surf(Z)

surf(X,Y,Z)

surf(X,Y,Z,C)

surf(...,'PropertyName',PropertyValue)

surf(axes\_handle,...)

surfc(...)

h = surf(...)

h = surfc(...)

hsurface = surf('v6',...), hsurface = surfc('v6',...)

### **Example**

Display contour plot under surface plot.

[X,Y] = meshgrid(1:0.5:10,1:20);

Z = sin(X) + cos(Y);

surfc(X,Y,Z)

# sphere()

The sphere function develops the x-, y-, and z-coordinates of a unit sphere for use with **surf** and **mesh**.

### **Syntax**

sphere // It generates a sphere consisting of 20-by-20 faces.

sphere(n) // It draws a surf plot of an n-by-n sphere in the current figure.

[X,Y, Z] = sphere(...) // It returns the coordinates of a sphere in three matrices that are (n+1)-by-(n+1) in size.

### **Example**

Generate and plot a sphere.

sphere(20)

axis('square')

or

[x,y,z]=sphere(20);

surf(x, y, z)

axis('square')

# cylinder()

cylinder creates x, y, and z coordinates of the unit cylinder. We can draw the cylindrical object using surf or mesh, or draw it immediately by not providing output arguments.

### **Syntax**

[X, Y, Z] = cylinder // It returns the x, y, and z coordinates of a cylinder with a radius similar to 1. The cylinder has 20 similar spaced points around its circumference.

[X, Y, Z] = cylinder(r) // It returns the x, y, and z coordinates of a cylinder using r to describe a profile curve. cylinder treats each component in r as a radius at equally spaced heights along with the unit height of the cylinder.

[X,Y,Z] = cylinder(r,n) // It returns the x, y, and z coordinates of a cylinder based on the profile curve described by vector r. The cylinder has n similar spaced points around its circumference.

cylinder(...) // with no output arguments, plot the cylinder using MATLAB surf.

### **Example**

r=sin?(3π z)+2

0≤z≤1,0≤θ≤2θ

z=[0: .02:1]';

r=sin(3\*pi\*z)+2;

cyclinder(r), axis square

# Multi-Dimensional Arrays in MATLAB

* Arrays with one more than two dimensions are called multi-dimensional arrays.
* Multi-dimensional arrays are created with more than two subscripts in MATLAB.
* For example:
  + Let's create a three-dimensional array using function ones (3, 8, 3).
  + This function creates a 3-by-8-by-3 array with a total of 3\*8\*3 = 72 elements.
  + The third subscript tells to create no. of sets of elements in rows and columns as per first & second subscripts.
* Let's have one more example:
* Here we use some more functions, and one of them is the **perms** function.
* The **perms** function returns all number of possible ways or permutations to arrange the elements of a matrix or vector in a different set of orders of a row vector.

# MATLAB Symbolic Mathematics

Symbolic mathematics defines doing mathematics on symbols (not numbers!). For example, a+a is 2a. The symbolic math function is in the Symbolic Math Toolbox in MATLAB. Toolboxes include related functions and are add-ons to MATLAB.

## **Symbolic Variables and Expressions**

MATLAB has type called **sym** for symbolic variables and expressions, and these work with string.

For example, to generate a symbolic variable **a** and perform the addition just defined, first, a symbolic variable will be created by passing the string **'a'** to the **sym** function:

>> a = sym('a');

>> a+a

ans =  2\*a

Symbolic variables can also store expressions. For example, the variables b and c save symbolic expressions:

>> b = sym('x^2');

>> c = sym('x^4');

All basic numerical operations can be performed on symbolic variables and expressions (e.g., add, subtract, multiply, divide, raise to a power, etc.).

Here are some examples:

>> c/b

ans =  x^2

>> b^3

ans =  x^6

>> c\*b

ans =  x^6

>> b + sym('4\*x^2')

ans =  5\*x^2

### **Simplification Functions**

Several functions work with expressions and simplify the terms. It is not all expressions that can be simplified, but the simplify function does whatever it can to simplify expressions containing gathering like terms.

For example:

>> x = sym('x');

>> myexpr = cos(x)^2 + sin(x)^2

myexpr =  cos(x)^2 sin(x)^2

>> simplify(myexpr)

ans =  1

The functions **collect, expand,** and **factor** works with polynomial expressions. The collect function collects coefficients.

For example:

>> x = sym('x');

>> collect(x^2+4\*x^3+3\*x^2)

ans =  4\*x^2+4\*x^3

The **expand** functions will multiply out terms, and element will do the reverse:

>> expand((x+2) \*(x-1))

ans =  x^2+x-2

>> factor(ans)

ans = (x+2)\*(x-1)

The **subs** function will substitute an equation for the symbolic variable in expressions.

For example:

>> myexp = x^3^ +3\*x^2^ -2

myexp =  x^ ^3^ +3\*x^2^ -2

>> x = 3;

>> subs (myexp, x)

ans = 52

With symbolic mathematics, MATLAB works by default with rational numbers, defining that results are kept in fractional forms. For example, performing the addition of 1/3+1/2 would usually result in a double value:

>> 1/3 + 1/2

ans =  0.8333

However, by making the function symbolic, the result is symbolic also. Any mathematical function (e.g., double) can modify that:

>> sym(1/3 + 1/2)

ans =  5/6

>> **double**(ans)

ans =0.8333

The **numden** functions will return the numerator and denominator of the symbolic expressions separately.

>> sym(1/3+1/2)

ans =  5/6

>> [n, d] = numden(ans)

n =  5

d =  6

>> [n, d] = numden((x^3^ +x^2)/x)

n =  x^2\*(x+1)

d =  x

# MATLAB Environment Programming

**Objective:** To study MATLAB environment and to familiarize with command window, history, workspace, current directory, figure window, edit window, shortcuts, helplines.

MATLAB is a particular computer program optimized to perform engineering and scientific calculations.

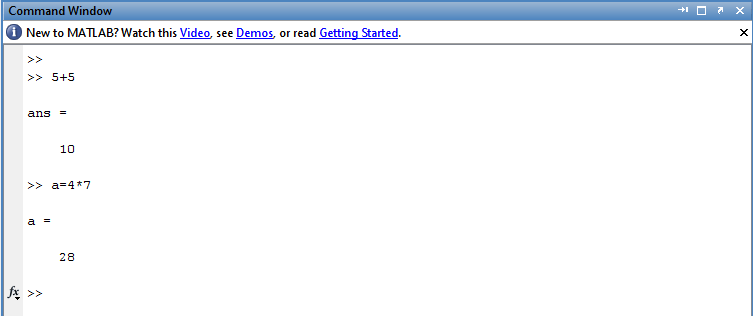
**Advantages:**

* Ease of use.
* Platform independence.
* Predefined functions.
* Device-independent plotting.
* GUI.
* MATLAB compiler.

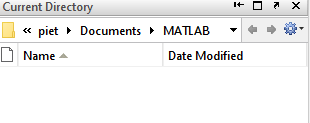
**Disadvantages:**Interpreted language.The cost is high.

### **MATLAB Environment**

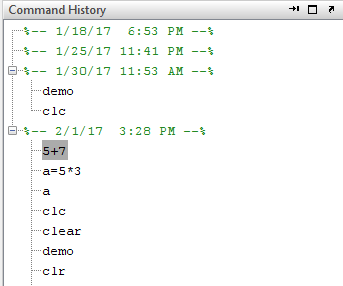
* **Command Window:** It is the space where commands may be entered.



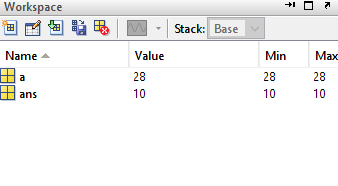
* **Figure Window:** It displays plots and graphs.
* **Edit Window:** It permits a user to create and modify MATLAB programs by creating new M files or to modify existing ones.
* **Current Directory Window:** It shows the path of the current directory.



* **Command History:** It displays a list of commands that the user has entered in the command window.



* **Workspace:** It is a collection of all the variables and array that can be used by MATLAB when a particular command, M file, or function is executed.



* **Help Files:** A user can get help from MATLAB through MATLAB documentation.

**Shortcuts of MATLAB:**

In MATLAB, if the statement is too long to type on a single line, it may be continued on successive lines by typing an ellipsis (...) at the end of the first line and then continuing on the next line.

Ex: a=1/2+3/2-2/3 ...+4/5-2/3;

**Shortcuts:**

* **clc:** clear command window.
* **clf:** clear contents of the current figure window
* **clear:** clears variables in workspace
* **abort:** (ctrl+C) For M files that appears running too long may contain an infinite loop that never terminates. To terminate, we use abort.
* **!:** It is a special character, after which any character or command will be sent to the operating system and executed as they had been types in operating system command prompt.
* **diary:** (diary filename)  
  After this command, a copy of all inputs and most of the outputs typed in the command window will be echoed in the diary file.
* **diary off:** It suspends input into the diary file.
* **diary on:** It resumes input again.
* **which:** It tells which version of a file is being executed and where it is located.

# MATLAB Control Statements

**Objective:** To study control structures (for, while, if, switch, break, continue, input/output functions, reading, and storing data).

**If:** If evaluates a logical expression and executes a group of statements based on the value of the expression.

**Syntax of If Statement**

if expression 1

statement1

elseif expression 2

statement 2

else

statement 3

end

### **Examples**

>> a=7

a =  7

>> **if** a>0

disp('a is positive');

elseif a<0

disp('a is negative')

**else**

disp('a is zero')

end

**Output:**

a is positive

**Switch, case, and otherwise:** Switch executes certain statements based on the value of a variable or expression. Its basic form is

**Syntax**

switch switch expression

case case expression

statements

case case expression

statements

.

.

.

otherwise

statements

end

An evaluated switch expression is a scaler or string. An evaluated case expression a scaler, a string, or a cell array of scaler or strings. The switch block tests each case is until one of the cases is true.

### **Examples**

Conditionally display different text depending on value entered at the command line.

 >> mynumber=input('enter a number')

enter a number -1

mynumber = -1

>> **switch** mynumber

**case** -1

disp('negative one')

**case** 0

disp('zero');

**case** 1

disp('positive one');

otherwise

disp('other value');

end

**Output:**

negative one

### **Example 2:**

>> result=52;

>> **switch**(result)

**case** 52

disp('result is 52')

**case** {52,78}

disp('result is 52 or 78')

end

**Output:**

result is 52

### **Example 3:**

>> [daynum, daystr] =weekday(date,'long','en\_US')

**switch**(daystr)

**case** 'monday'

disp('start of week')

**case** 'tuesday'

disp('day 2')

otherwise

disp('weekend')

end

**Output:**

weekend

# MATLAB Sine Wave Plot

**Objective:** To plot a sine wave of the frequency of 1KHz.

**Example:** Let's generate a simple continuous like sinusoidal signal with frequency FM=1KHz. In order to make it occur as a repetitive signal when plotting, a sampling rate of fs=500KHz is used.

fs= 500e3;

f= 1000;

nCyl=5;

t=0:1/fs:nCyl\*1/f;

x=sin(2\*pi\*f\*t);

plot(t,x)

title ('Continuous sinusoidal signal')

xlabel('Time(s)');

ylabel('Amplitude');

**Output:**

