EXPERIMENT-ONE

INTRODUCTION

1- What Is MATLAB ?

MATLAB® is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation.

The name MATLAB stands for MATrix LABoratory. MATLAB was originally written to provide easy access to matrix software. MATLAB has evolved over a period of years with input from many users. In university environments, it is the standard instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of application-specific solutions called *toolboxes*. Very important to most users of MATLAB, toolboxes allow you to *learn* and *apply* specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

2- The MATLAB System

The MATLAB system consists of five main parts:

2.1- Development Environment: This is the set of tools and facilities that help you use MATLAB functions and files. Many of these tools are Graphical User Interfaces (GUI). It includes the MATLAB desktop and Command Window, a command history, and browsers for viewing help, the workspace, files, and the search path.

2.2- The MATLAB Mathematical Function Library: This is a vast collection of computational algorithms ranging from elementary functions like sum, sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms.

2.3- The MATLAB Language: This is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows both "programming in the small" to rapidly create quick and dirty throw-away programs, and "programming in the large" to create complete large and complex application programs.

2.4- Handle Graphics: This is the MATLAB graphics system. It includes high-level commands for two-dimensional and three-dimensional data visualization, image processing, animation, and presentation graphics. It also includes low-level commands that allow you to fully customize the appearance of graphics as well as to build complete graphical user interfaces on your MATLAB applications.

2.5- The MATLAB Simulink: This is a software for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates. Simulink enables you to posea question about a system, model it, and see what happens.

3- Starting, Quitting MATLAB and Opening M-files

3.1- Starting MATLAB: On a Microsoft Windows platform, to start MATLAB, double-click the MATLAB shortcut icon on your Windows desktop. After starting MATLAB, the MATLAB desktop opens.

3.2- Quitting MATLAB: To end your MATLAB session, select **Exit MATLAB** from the **File** menu in the desktop, or type quit in the Command Window, or from the closed bottom in the upper right corner of the command window.

3.2- Opening M-files: M-files are the files that we will write our only programs on it. From the MATLAB command window select the **File** menu and choose **New/M-file**. This action opens a Notepad (Untitled) window. You can regard this as a 'scratch pad' in which to write programs. From the M-file **save** menu you can save your program after typing any name instead of Untitled. Also you can run this program by choosing **Run** from the M-file **Debug** menu (or press F5 key). The results will appear on the MATLAB Desktop.

4- MATLAB Desktop

When you start MATLAB, the MATLAB desktop appears, 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 Fig-1, your desktop looks by opening, closing, moving, and resizing the tools in it. You can also move tools outside of the desktop or return them back inside the desktop (docking). All the desktop tools provide common features such as context menus and keyboard shortcuts. You can specify certain characteristics for the desktop tools by selecting **Preferences** from the **File** menu. For example, you can specify the font characteristics for Command Window text. For more information, click the **Help** button in the **Preferences** dialog box. Although your Launch Pad may contain different entries, you can change the way you want the desktop appearance.

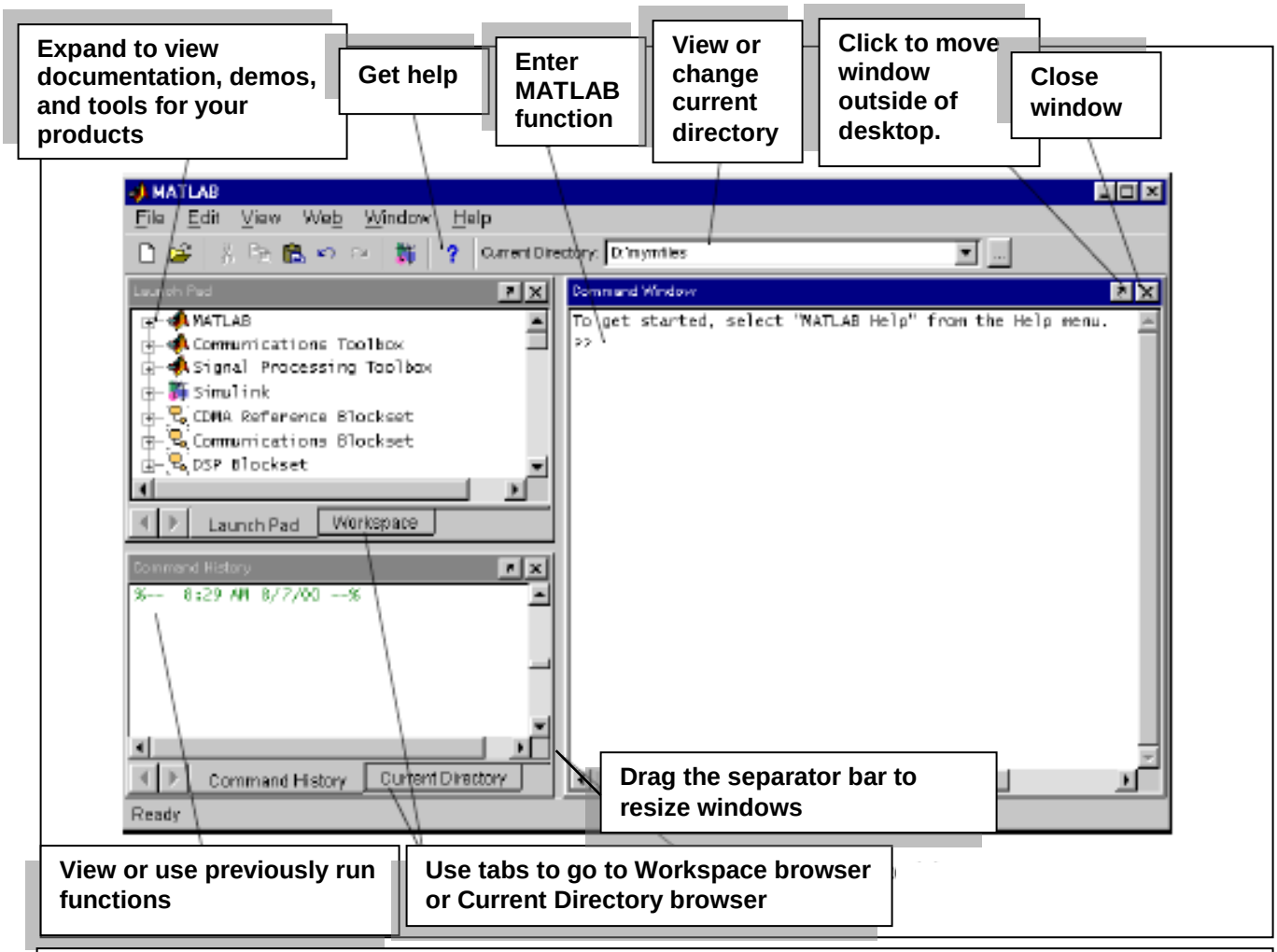


Fig-1: The MATLAB Desktop

Usually the following tools are appeared in the MATLAB's desktop:

- The Command Window.
- The Command History.
- The Launch Pad.
- Workspace Browser.
- The Array Editor.
- Editor/Debugger.

4.1- <u>Command Window:</u> Use the **Command Window** to enter variables and run functions and M-files and controlling input and output data. The command window is shown in Fig-2.

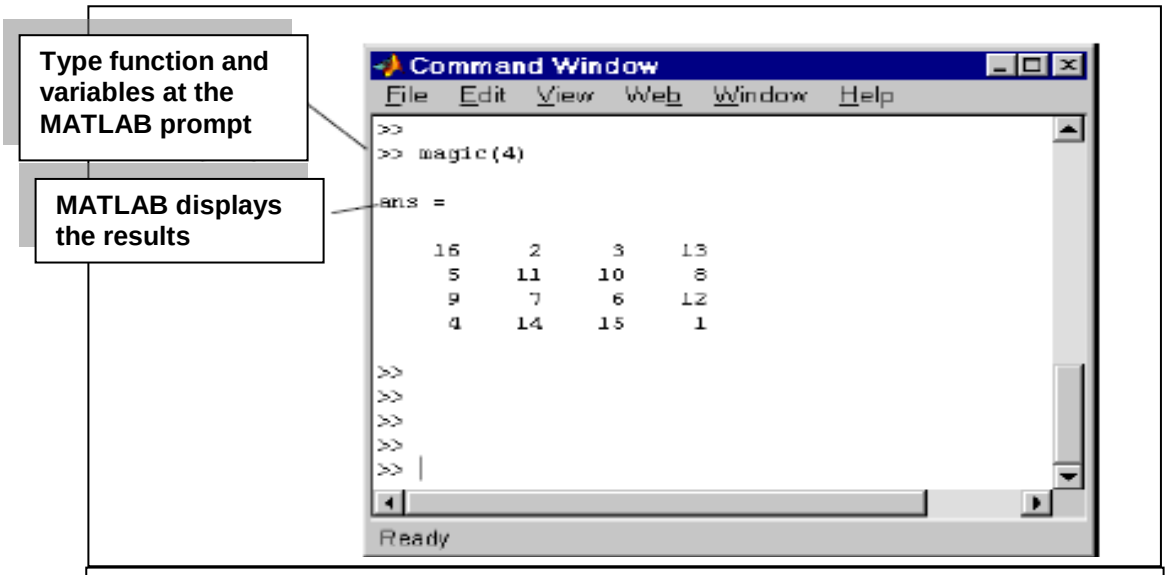


Fig-2: Command Window

4.2- <u>Command History:</u> Lines you enter in the Command Window are logged in the **Command History** window shown in Fig-3. In the Command History, you can view previously used functions, and copy and execute selected lines.

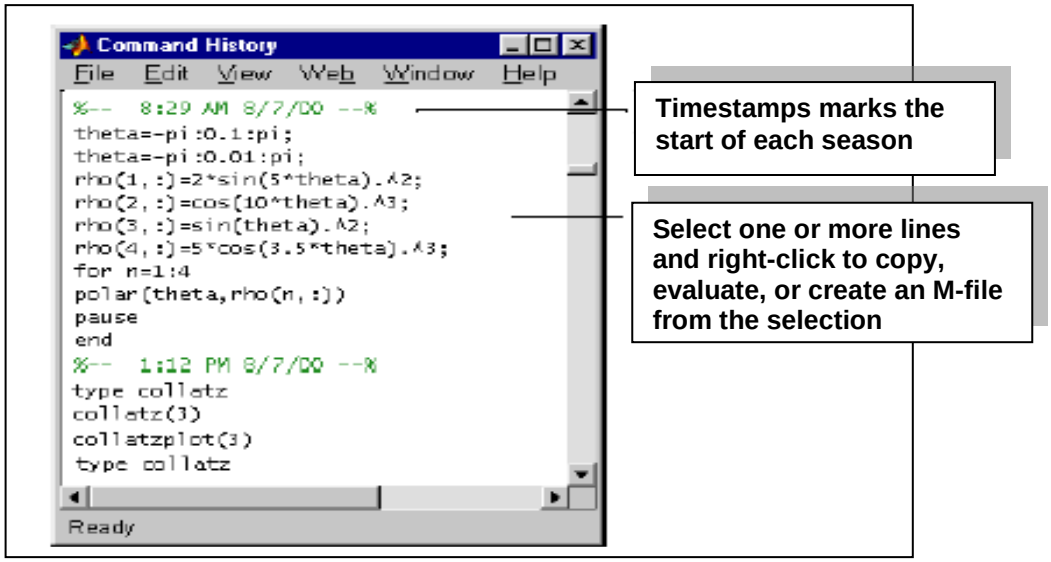


Fig-3: Command History

4.3- Launch Pad: MATLAB's **Launch Pad** (shown in Fig-4) provides easy access to tools, demos, and documentation.

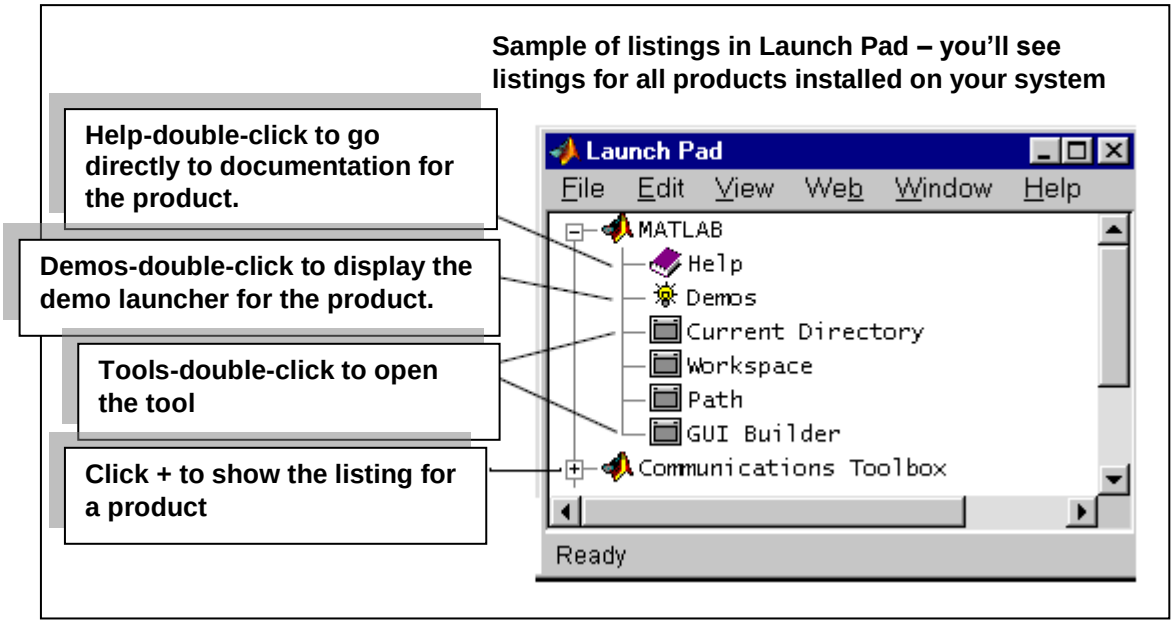


Fig-4: Launch Pad

4.4- Workspace Browser: The MATLAB workspace 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 (shown in Fig-5), or use the functions who and whos.

To delete variables from the workspace, select the variable and select **Delete** from the **Edit** menu. Alternatively, use the clear function. The workspace is not maintained after you end the MATLAB session. To save the workspace to a file that can be read during a later MATLAB session, select **Save Workspace As** from the **File** menu, or use the save function. This saves the workspace to a binary file called a MAT-file, which has a .mat extension. There are options for saving to different formats. To read in a MAT-file, select **Import Data** from the **File** menu, or use the load function.

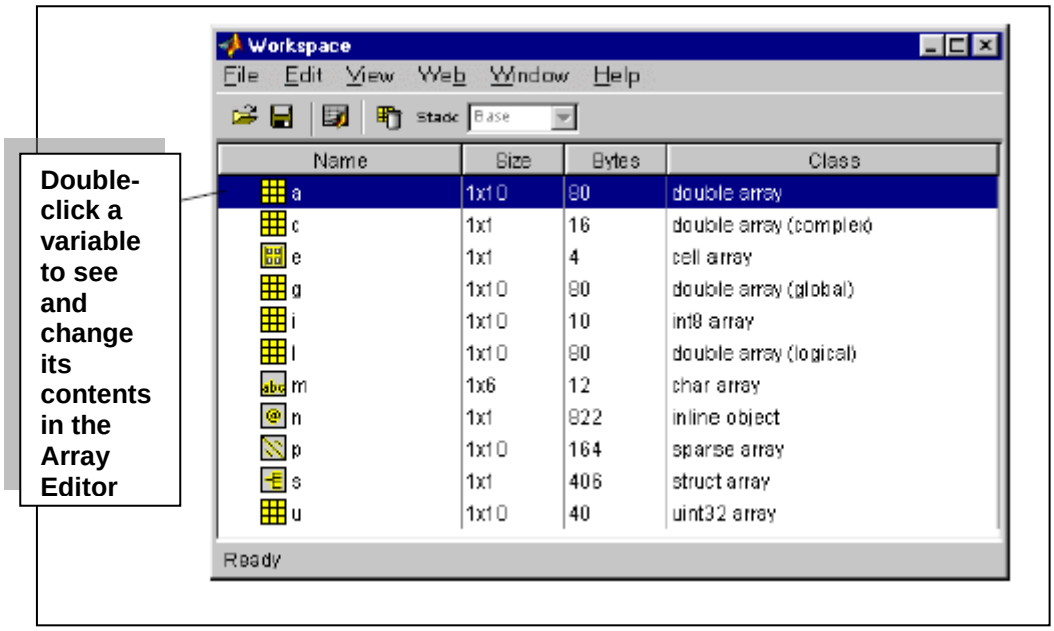


Fig-5: The Workspace Browser

4.5- Array Editor: Double-click on a variable in the Workspace browser to see it in the Array Editor shown in Fig-6. Use the Array Editor to view and edit a visual representation of one- or two-dimensional numeric arrays, strings, and cell arrays of strings that are in the workspace.

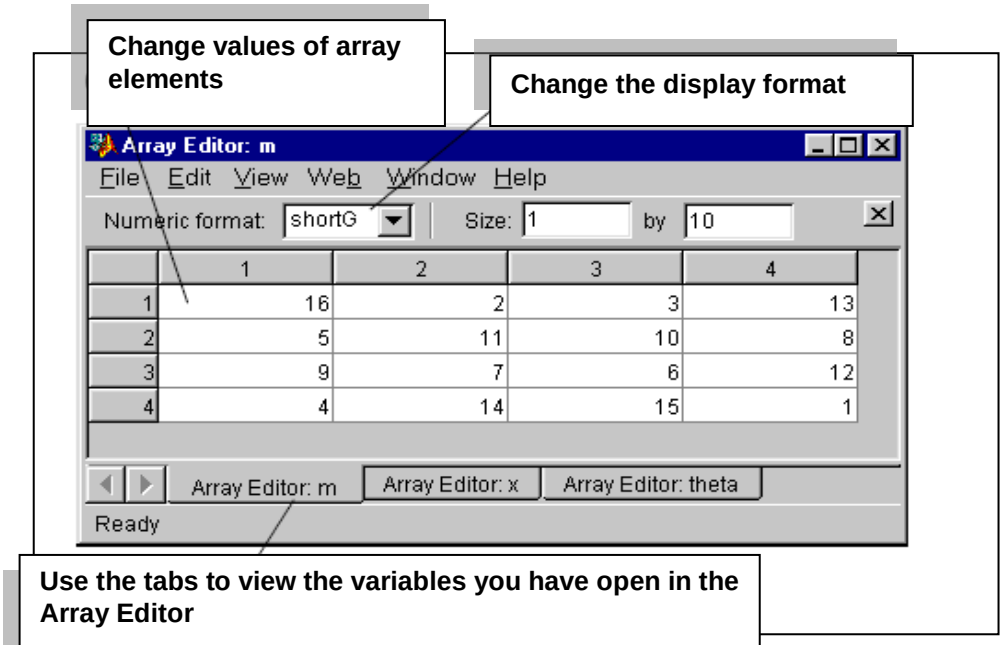


Fig-6: The Array Editor

4.6- <u>Editor/Debugger:</u> Use the Editor/Debugger to create and debug M-files, which are programs you write to run MATLAB functions. The Editor/Debugger (shown in Fig-7) provides a graphical user interface for basic text editing, as well as for M-file debugging. You can use any text editor to create M-files, such as Emacs, and can use preferences (accessible from the desktop **File** menu) to specify that editor as the default. If you use another editor, you can still use the MATLAB Editor/Debugger for debugging, or you can use debugging functions, such as dbstop, which sets a breakpoint. If you just need to view the contents of an M-file, you can display it in the Command Window by using the type function.

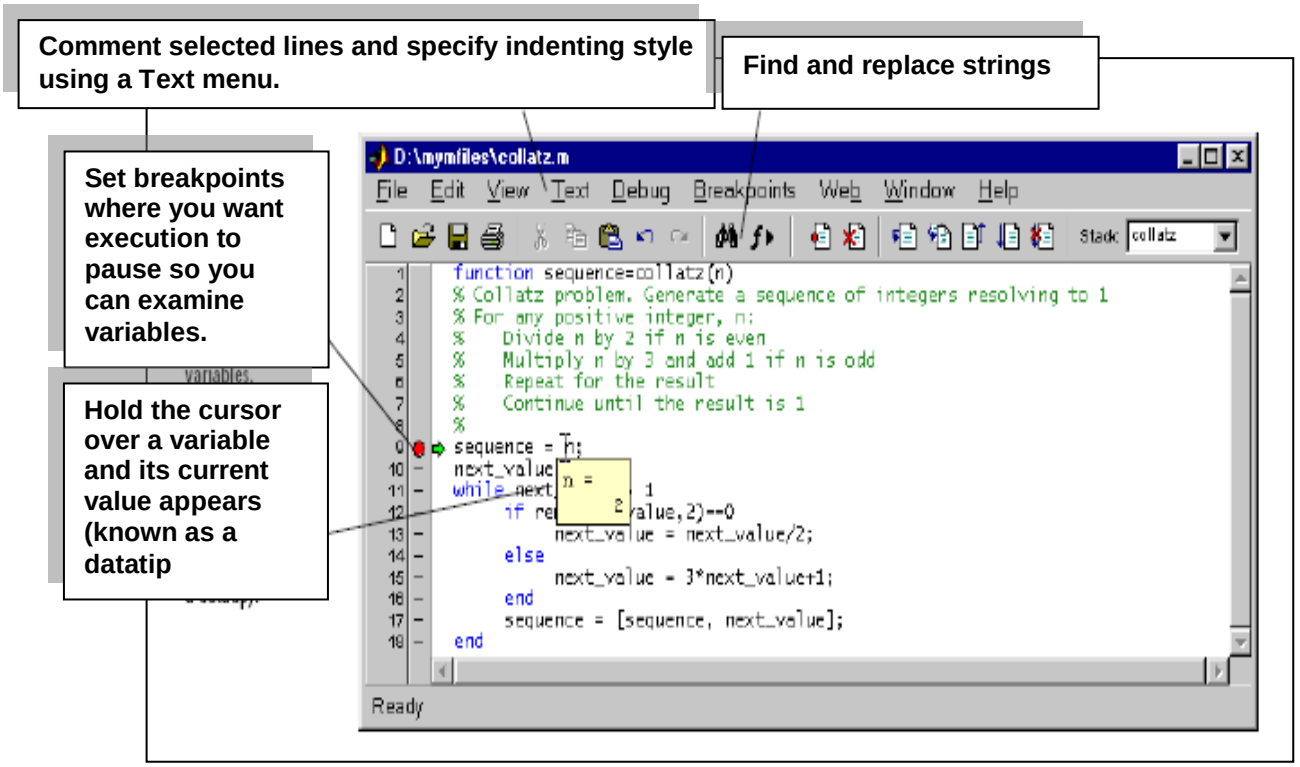


Fig-7: The Edditor/Debugger

5- Controlling Command Window Input and Output

So far, you have been using the MATLAB command line, typing commands and expressions, and seeing the results printed in the Command Window. This section describes how to:

- Control the appearance of the output values (format command).
- Suppress output from MATLAB commands.
- Enter long commands at the command line.
- Edit the command line.

5.1- The format Command: The format command controls the numeric format of the values displayed by MATLAB. The command affects only how numbers are displayed, not how MATLAB computes or saves them. Here are the different formats, together with the resulting output produced from a vector x with components of different magnitudes.

```
>> x = [4/3       1.2345e-6]
>> format short            % Scaled fixed point format with 5 digits (default display).
>> x       < Enter >
 x =
    1.3333       0.0000

>> format short e          % Floating point format with 5 digits plus exponent.
>> x       < Enter >
 x =
    1.3333e+000     1.2345e-006

>> format long             % Scaled fixed point format with 15 digits.
>> x       < Enter >
x =
   1.33333333333333        0.00000123450000

>> format long e           % Floating point format with 15 digits.
>> x       < Enter >
 x =
    1.333333333333333e+000        1.234500000000000e-006
>> format bank             % Fixed format for dollars and cents.
>> x       < Enter >
 x =
    1.33     0.00

>> format rat              % Approximation by ratio of small integers.
>> x       < Enter >
x =
    4/3     1/810045

>> format hex              % Hexadecimal format.
>> x       < Enter >
x =
    3ff5555555555555        3eb4b6231abfd271
```

If the largest element of a matrix is larger than 10^3 or smaller than 10^{-3}, MATLAB applies a common scale factor for the short and long formats. In addition to the format commands shown above, format compact suppresses many of the blank lines that appear in the output. This lets you view more information on a screen or window.

5.2- Suppressing Output: If you simply type a statement and press **Return** or **Enter**, MATLAB automatically displays the results on screen. However, if you end the line with a semicolon, MATLAB performs the computation but does not display any output. This is particularly useful when you generate large matrices. For example,

```
>> A = 100;       < Enter >
>>
```

5.3- Entering Long Command Lines: If a statement does not fit on one line, use three periods, ..., followed by **Return** or **Enter** to indicate that the statement continues on the next line. For example,

```
>> s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...      < Enter >
    - 1/8 + 1/9 - 1/10 + 1/11 - 1/12;                < Enter >
>>
```

Blank spaces around the = , + , and - signs are optional, but they improve readability.

5.4- Command Line Editing: Various arrow and control keys on your keyboard allow you to recall, edit, and reuse commands you have typed earlier. For example, suppose you mistakenly enter

```
>> rho = (1 + sqt(5))/2      < Enter >
>> Undefined function or variable 'sqt'.
```

You have misspelled sqrt. MATLAB responds with. So instead of retyping the entire line, simply press the ↑ key. The misspelled command is redisplayed. Use the ← key to move the cursor over and insert the missing r. Repeated use of the ↑ key recalls earlier lines. Typing a few characters and then the ↑ key finds a previous line that begins with those characters. You can also copy previously executed commands from the Command History.

The list of available command line editing keys is different on different computers. Experiment to see which of the following keys is available on your machine.

↑	**Ctrl+p**	Recall previous line
↓	**Ctrl+n**	Recall next line
←	**Ctrl+b**	Move back one character
→	**Ctrl+f**	Move forward one character

Ctrl+→	**Ctrl+r**	Move right one word
Ctrl+←	**Ctrl+l**	Move left one word
Home	**Ctrl+a**	Move to beginning of line
End	**Ctrl+e**	Move to end of line
Esc	**Ctrl+u**	Clear line
Del	**Ctrl+d**	Delete character at cursor
Backspace	**Ctrl+h**	Delete character before cursor
	Ctrl+k	Delete to end of line

Exercises:

1. What is MATLAB and what are its applications?
2. How many parts is the MATLAB system consists of? What are they?
3. What are the main parts of the MATLAB desktop? List the advantage of each part.
4. How can we control the appearance of the output values? And how can we suppress output from MATLAB commands.
5. Apply the format commands on a two numbers (A & B) on the command window.
6. Build the following program in M-file. Run the program and get the results;

```
% This is a test program
disp ('The variables value are')
x=3.05
y=5.6e-2
a=x/y;
b=x*y;
c=x\y;
res1=sqrt(a)+(b/c)
res2=res1+(c^2)
disp ('The program ends here')
```

EXPERIMENT-TWO

MATLAB EXPRESSIONS

Like most other programming languages, MATLAB provides *mathematical expressions*, but unlike most programming languages, these expressions involve entire matrices. The building blocks of expressions are:

- Variables
- Numbers
- Operators
- Functions

1- Variables

MATLAB does not require any type declarations or dimension statements. When MATLAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage. If the variable already exists, MATLAB changes its contents and, if necessary, allocates new storage. For example,

```
>> num_students = 25
```

creates a 1-by-1 matrix named num_students and stores the value 25 in its single element.

Variable names consist of a letter, followed by any number of letters, digits, or underscores. MATLAB uses only the first 31 characters of a variable name.
MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. A and a are *not* the same variable. To view the matrix assigned to any variable, simply enter the variable name.

2- Numbers

MATLAB uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, for numbers. *Scientific notation* uses the letter e to specify a power-of-ten scale factor. *Imaginary numbers* use either i or j as a suffix. Some examples of legal numbers are

3	-99	0.0001
9.6397238	1.60210e-20	6.02252e23
1i	-3.14159j	3e5i

All numbers are stored internally using the *long* format specified by the floating-point standard. Floating-point numbers have a finite *precision* of roughly 16 significant decimal digits and a finite *range* of roughly 10^{-308} to 10^{+308}. Several special numbers provide values of useful constants.

pi	π = 3.14159265…
i	Imaginary unit, $\sqrt{-1}$
j	Same as i
eps	Floating-point relative precision, 2^{-52} = 2.2204e-016
realmin	Smallest floating-point number, 2^{-1022} = 2.2251e-308
realmax	Largest floating-point number, 2^{1023} = 1.7977e+308
Inf	Infinity
NaN	Not-a-number

Infinity is generated by dividing a nonzero value by zero, or by evaluating well defined mathematical expressions that *overflow*, i.e., exceed realmax. Not-a-number is generated by trying to evaluate expressions like 0/0 or Inf-Inf that do not have well defined mathematical values.

The function names are not reserved. It is possible to overwrite any of them with a new variable, such as

```
>> eps = 1.e-6
```

and then use that value in subsequent calculations. The original function can be restored with

```
>> clear eps
```

3- Operators

Expressions use familiar arithmetic operators and precedence rules.

( )	Specify evaluation order
'	Complex conjugate or matrix transpose.
^	Power

\ Left division (i.e. $\frac{x}{y} = y \setminus x$)

/ Right Division (i.e. $\frac{x}{y} = x / y$)

* Multiplication

\- Subtraction

\+ Addition

4- Functions

MATLAB provides a large number of standard elementary mathematical functions, including abs, sqrt, exp, and sin. Taking the square root or logarithm of a negative number is not an error; the appropriate complex result is produced automatically. MATLAB also provides many more advanced mathematical functions, including Bessel and gamma functions. Most of these functions accept complex arguments. For a list of the elementary mathematical functions, type

```
>> help elfun
```

For a list of more advanced mathematical and matrix functions, type

```
>> help specfun
>> help elmat
```

So we can classify the elementary mathematical functions into:

- Trigonometric Functions.
- Exponential Functions.
- Complex Functions.
- Rounding and remainder Functions.
- User functions

4.1- Trigonometric Functions:

Sin(x) Sine of x.

Sinh(x) Hyperbolic sine (i.e. $\frac{e^{x} - e^{-x}}{2}$)

asin(x)	Inverse sine of x
asinh(x)	Inverse hyperbolic sine (i.e. $\ln(x+\sqrt{x^2+1})$)
cos(x)	Cosine of x.
cosh(x)	Hyperbolic cosine (i.e. $\frac{e^x+e^{-x}}{2}$)
acos(x)	Inverse cosine of x.
acosh	Inverse hyperbolic cosine (i.e. $\ln(x+\sqrt{x^2-1})$)
tan(x)	Tangent x.
tanh(x)	Hyperbolic tangent of x.
atan(x)	Inverse tangent of x.
atan2(y,x)	Four quadrant inverse tangent (i.e. $\tan^{-1}(\frac{y}{x})$)
atanh(x)	Inverse hyperbolic tangent (i.e. $\frac{1}{2}\ln(\frac{1+x}{1-x})$)
sec(x)	Secant x.
sech(x)	Hyperbolic secant x.
asec(x)	Inverse secant x.
asech(x)	Inverse hyperbolic secant x.
csc(x)	Cosecant x.
csch(x)	Hyperbolic cosecant x.
acsc(x)	Inverse cosecant x.
acsch(x)	Inverse hyperbolic cosecant x.
cot(x)	Cotangent x.
coth(x)	Hyperbolic cotangent x.
acot(x)	Inverse cotangent x.
acoth(x)	Inverse hyperbolic cotangent x.

4.2- Exponential Functions:

exp(x)	Exponential.
Log(x)	Natural logarithm.
log10(x)	Common (base 10) logarithm.
log2(x)	Base 2 logarithm and dissect floating point number.
pow2(x)	Base 2 power and scale floating point number.
realpow(x)	Power that will error out on complex result.
reallog(x)	Natural logarithm of real number.
realsqrt(x)	Square root of number greater than or equal to zero.

sqrt(x)	Square root.
nextpow2(x)	Next higher power of 2.

4.3- Complex Functions:

abs(x)	Absolute value.
angle(x)	Phase angle.
complex(x)	Construct complex data from real and imaginary parts.
conj(x)	Complex conjugate.
imag(x)	Complex imaginary part.
real(x)	Complex real part.
unwrap(x)	Unwrap phase angle.
isreal(x)	True for real array.
cplxpair(x)	Sort numbers into complex conjugate pairs.

4.4- Rounding and remainder Functions:

fix(x)	Round towards zero.
floor(x)	Round towards minus infinity.
ceil(x)	Round towards plus infinity.
round(x)	Round towards nearest integer.
mod(x,y)	Modulus (signed remainder after dividing x over y).
rem(x,y)	Remainder after division.
sign(x)	Signum.

Note: mod(x,y) and rem(x,y) are equal if x and y have the same sign, but differ by y if x and y have different signs.

4.5- User functions:

Some of the functions, like sqrt and sin, are *built-in*. They are part of the MATLAB core so they are very efficient, but you may want to implement your only function in M-files and even modify it if you want. The form that you use it to build your only function in M-file is:

function **output variables** = **function_name** (**input_variables**)
output variables = any relation with the **input variables**

Note that the function name will be the M-file name.

Exercises:

1. Use MATLAB to evaluate the following expressions:

(a) $2^{2\times3} + 10^{-5}$
(b) $1.5 \times 10^{-4} + 2.5 \times 10^{-2}$
(c) $1000\,(1 + 0.15 / 12)^{60}$
(e) $(0.0000123 + 5.678 \times 10^{-3}) \times 0.4567 \times 10^{-4}$

2. Translate the following into MATLAB statement (make x=3.0024):
(a) Round x towards zero and store the result in i.
(b) Round x towards minus infinity and store the result in j.
(c) Round x towards plus infinity and store the result in k.
(d) Round x towards nearest integer and store the result in m.

3. Write an m-file program to calculate x_1 and x_2, where

$$x_1, x_2 = \frac{-b \mp \sqrt{b^2 - 4ac}}{2a}$$

Find the solution for :

(a) a = 2, b = -10, c = 12.
(b) a = 1, b = 2, c = 1.
(c) a = 6, b = 5, c = 8.

4. The steady-state current I flowing in a circuit that contains a resistance R = 5, capacitance C = 10, and inductance L = 4 in series is given by:

$$I = \frac{E}{\sqrt{R^2 + (2\pi\omega L - \frac{1}{2\pi\omega C})^2}}$$

Where E = 2 and ω = 2 are the input voltage and angular frequency respectively. Compute the value of I.

5. Find the value of the following by using a simplified form of function in one m-file program:

(a) $\ln(2\pi + \sqrt{4\pi^2 - 1})$

(b) $\frac{1}{2}\ln(\frac{1+1.5\pi}{1-1.5\pi})$

(c) $2^{15} + \frac{e^{2.5\pi} + e^{-2.5\pi}}{2}$

(d) $\tan^{-1}(\frac{4.3\pi}{5.4\pi}) + e^{2.3}$

6. Write a program to calculate sum and average of three numbers in functions called ("Summation") and ('Average") respectively. Enter the numbers in the command window, call the functions and find the result.

EXPERIMENT-THREE

VECTORS, MATRICES and ARRAYS

1- VECTORS

In MATLAB we can make a row vector (X) by separating the elements of a row with blanks or commas and surrounding the entire list of elements with square brackets, [ ].

```
>> X= [3 6 4]
```

or

```
>> X= [3,6,4]
```

MATLAB displays the vector:

```
X =
    3    6    4
```

Also we can use the colon operator, :, to perform a row vector . The expression 1:10 is an increased row vector containing the integers from 1 to 10

```
>> X= 1:10

X=
   1 2 3 4 5 6 7 8 9 10
```

and

```
>> C= 0:pi/4:pi

C=
   0    0.7854    1.5708    2.3562    3.1416
```

or a decreased row vector:

```
>> S= 100:-7:50
S=
  100  93  86  79  72  65  58  51
```

If you want to make X as a column vector, separate the elements of a row with semicolons, ; , and surrounding the entire list of elements with square brackets, [ ].

```
>> X= [4; 6; 7]

X =
    4
    6
    7
```

2- Matrices

The best way for you to get started with MATLAB is to learn how to handle matrices. So you have only to follow a few basic conventions:

- Separate the elements of a row with blanks or commas.
- Use a semicolon, ; , to indicate the end of each row.
- Surround the entire list of elements with square brackets, [ ].

As an example, if we enter in the command window the matrix A:

```
>> A=[2 3 4; 7 5 8;14 23 55]
```

MATLAB displays the matrix you just entered.

```
A =
    2    3    4
    7    5    8
   14   23   55
```

Once you have entered the matrix, it is automatically remembered in the MATLAB workspace. You can refer to it simply as A. Now you have A in the workspace.

2.1- Arithmetic operations on matrices:

The function sum enabled you to take the sum along any row or column, or along either of the two main diagonals. Let's verify that using MATLAB, If we want the sum of each column in matrix A above:

```
>> sum(A)
```

MATLAB replies with

```
ans =

   23    31    67
```

So You have computed a row vector containing the sums of the columns of A. How about the row sums? MATLAB has a preference for working with the columns of a matrix, so the easiest way to get the row sums is to transpose the matrix, compute the column sums of the transpose, and then transpose the result. The transpose operation is denoted by an apostrophe or single quote, '.
It flips a matrix about its main diagonal and it turns a row vector into a column vector. So

```
>> A'
```

produces

```
ans =
    2    7   14
    3    5   23
    4    8   55
```

and

```
>> sum(A')'
```

produces a column vector containing the row sums

```
ans =

    9
   20
   92
```

The sum of the elements on the main diagonal is easily obtained with the help of the diag function, which picks off that diagonal.

```
>> diag(A)
```

produces

```
ans =

    2
    5
   55
```

and

```
>> sum(diag(A))
```

produces

```
ans =
   62
```

The other diagonal, the so-called *anti diagonal*, is not so important mathematically, so MATLAB does not have a ready-made function for it. But a function originally intended for use in graphics, fliplr, flips a matrix from left to right. So the sum of the anti diagonal is:

```
>> sum(diag(fliplr(A)))

ans =

   23
```

2.2- **Subscripts:**

The element in row i and column j of A is denoted by A(i,j). For example, A(3,2) is the number in the third row and second column. For our matrix (A), A(3,2) is 23. So it is possible to compute the sum of the elements in the third column of A by typing

```
>> A(1,3)+A(2,3)+A(3,3)
```

This produces

```
ans =

   67
```

But this is not an efficient way. The following subsection will produce the best way for this case.

2.3- The Colon Operator:

The colon operator is extremely powerful, and provide for very efficient ways of handling matrices. Subscript expressions involving colons refer to portions of a matrix. A(1:k,j) is the first k elements of the jth column of A. So:

```
>> sum(A(1:3,3))
```

computes the sum of the third column.

```
ans =
   67
```

But there is a better way. The colon by itself refers to *all* the elements in a row or column of a matrix and the keyword end refers to the *last* row or column. So

```
>> sum(A(:,end))
```

computes the sum of the elements in the last column of A.

```
ans =
   67
```

Other advantages of the colon operator will be listed in the following examples:

```
>> A(2:3,1:2)
```

this mean: return second and third rows, first and second columns of A.

```
ans =

    7     5
   14    23
```

```
>> A(3,:)
```

this mean: return the third row of A.

```
ans =
   14   23   55
```

```
>> A(1:2,1:2) = ones(2)
```

this mean: replace first and second rows and columns of A by a square matrix of 1's.

```
ans =
    1    1    4
    1    1    8
   14   23   55
```

2.4- Deleting Rows and Columns:

You can delete rows and columns from a matrix using just a pair of square brackets. Start with

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

```
D =
    4    6    8    7
    3    4    5    1
    2    3    8    9
    5    3    1    9
>>X = D;
```

Then, to delete the second column of X, use

```
>> X(:,2) = [ ]
```

This changes X to

```
X =
    4    8    7
    3    5    1
    2    8    9
    5    1    9
```

If you delete a single element from a matrix, the result isn't a matrix anymore. So, expressions like

```
>> X(1,2) = []
```

result in an error. However, using a single subscript deletes a single element, or sequence of elements, and reshapes the remaining elements into a row vector. So

```
>> X(2:2:10) = []
```

results in

```
X =
     4  2  8  8  7  9  9
```

2.5- Concatenation:

Concatenation is the process of joining small matrices to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, [], is the concatenation operator. For an example, start with the 2- by-2 square matrix, C,

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

C =

    1    2
    3    4
```

and form the matrix B:

```
>> B = [C C+2; C+4 C+6]
```

The result is an 4-by-4 matrix, obtained by joining the four sub matrices.

```
B =

    1    2    3    4
    3    4    5    6
    5    6    7    8
    7    8    9   10
```

2.6- Matrices Linear Algebra:

The mathematical operations defined on matrices are the subject of *linear algebra*. Let:

```
A =
   16 3 2 13
    5 10 11 8
    9  6 7 12
    4 15 14 1
```

provides several examples that give a taste of MATLAB matrix operations. You've already seen the matrix transpose, A'. Adding a matrix to its transpose produces a *symmetric* matrix.

```
>> A + A'
ans =

32 8 11 17
8 20 17 23
11 17 14 26
17 23 26 2
```

The multiplication symbol, *, denotes the *matrix* multiplication involving inner products between rows and columns. Multiplying the transpose of a matrix by the original matrix also produces a symmetric matrix.

```
>> A'*A
ans =

378 212 206 360
212 370 368 206
206 368 370 212
360 206 212 378
```

The determinant of this particular matrix happens to be zero, indicating that the matrix is *singular*.

```
>> d = det(A)
d =
   0
```

Since the matrix is singular, it does not have an inverse. If you try to compute the inverse with

```
>> X = inv(A)
```

you will get a warning message "warning: Matrix is close to singular or badly scaled".

```
>> e = eig(A)

e =
34.0000
8.0000
0.0000
-8.0000
```

One of the eigen values is zero, which is another consequence of singularity.

2.7- Generating Matrices:

MATLAB provides four functions that generate basic matrices of size (R×C):

Zeros(R,C)	All the elements of the matrix are zeros.
Ones(R,C)	All the elements of the matrix are ones.
Rand(R,C)	Uniformly distributed random elements.
Randn(R,C)	Normally distributed random elements.

and here are some examples.

```
>> Z = zeros(2,4)

Z =
   0 0 0 0
   0 0 0 0

>> F = 5*ones(3,3)

F =
   5 5 5
   5 5 5
   5 5 5
>> N = fix(10*rand(1,10))

N =
   4 9 4 4 8 5 2 6 8 0
```

```
>> R = randn(4,4)
R =
    1.0668   0.2944   -0.6918   -1.4410
    0.0593  -1.3362    0.8580    0.5711
   -0.0956   0.7143    1.2540   -0.3999
   -0.8323   1.6236   -1.5937    0.6900
```

3- Arrays

Informally, the terms *matrix* and *array* are often used interchangeably. More precisely, a *matrix* is a two-dimensional numeric array that represents a *linear transformation*.

3.1- Array operators:

Arithmetic operations on arrays are done element-by-element. This means that addition and subtraction are the same for arrays and matrices, but that multiplicative operations are different. MATLAB uses a dot, or decimal point, as part of the notation for multiplicative array operations. The list of operators includes:

+	Addition
-	Subtraction
.*	Element-by-element multiplication
./	Element-by-element division
.\	Element-by-element left division
.^	Element-by-element power
.'	Unconjugated array transpose

As an example: enter the following statements at the command line

```
>> a = [2  4  8];
>> b = [4  2  2];
>> a . * b        <ENTER>
ans =
       8   8  16
>> a . / b        <ENTER>
ans =
```

```
      0.5   2   4
>> a . ^ b       <ENTER>
ans =
      16   16   64
```

A common application of element-by-element multiplication is in finding the *scalar product* (also called the *dot product*) of two vectors **a** and **b**, which is defined as:

$$\mathbf{a}.\mathbf{b} = \sum_{\mathbf{i}} \mathbf{a_i} \mathbf{b_i}$$

and in MATLAB can be represented as:

```
>> Sum ( a .* b )
ans =
      32
```

3.2- Array tables:

Array operations are useful for building tables. Suppose n is the column vector

```
>> n = (0:8)';
```

Then

```
>> pows = [n n.^2 2.^n]
```

builds a table of squares and powers of two.

```
pows =
      0 0 1
      1 1 2
      2 4 4
      3 9 8
      4 16 16
      5 25 32
      6 36 64
      7 49 128
      8 64 256
```

The elementary math functions operate on arrays element by element. So format short g

```
>> x = (1:0.1:2)';
>> logs = [x log10(x)]
```

builds a table of logarithms.

```
logs =
    1.0   0
    1.1   0.04139
    1.2   0.07918
    1.3   0.11394
    1.4   0.14613
    1.5   0.17609
    1.6   0.20412
    1.7   0.23045
    1.8   0.25527
    1.9   0.27875
    2.0   0.30103
```

Exercises:

1. Let C be any 4×4 matrix. Write some statements to find :

a) Sum of each columns.
b) Sum of each rows.
c) Sum of the main diagonal elements.
d) Sum of the anti diagonal elements.
e) Sum of the third row.

2. Set up any 3×3 matrix D. Write some statements to convert D into a row vector X contains:

a) The odd elements of D.
b) The even elements of D.
c) The first and the last elements of D.
d) The three last elements of D.

3. If A and B are 2×2 matrices. Find a matrix C such that:

a) $\mathbf{C = A^T + B}$

b) $\mathbf{C} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{A+1} & \mathbf{B+2} \end{bmatrix}$

c) $\mathbf{C = AB / (A+B)}$

d) $\mathbf{C = A - A^{-1}B}$

4. If A is 4×4 normally distributed random matrix and I is 4×4 identity matrix. Proof that:

a) $\mathbf{A^{-1} A = I}$
b) A is not a singular matrix (use two different solutions).
c) $\mathbf{|A| - |I| \, |A| = 0}$
d) $\mathbf{A^{-1} A = A A^{-1}}$

5. Solve the equations below :

$$\mathbf{2x - y + z = 4}$$
$$\mathbf{x + y + z = 3}$$
$$\mathbf{3x - y - z = 1}$$

Hint : The solution of the equation AX=B is: $X=A^{-1}B$. Where A is the variables coefficient matrix, X is the variables column vector and B is the constants column vector.

6. If X and Y are two row vectors. Use the array operations to find:

a) $\sum_{i=1}^{10} X_i Y_i$ **b)** $\sum_{i=1}^{10} X_i^{Y_i}$ **c)** $\sum_{i=1}^{10} 4 X_i^3 + \sum_{i=1}^{10} 5 Y_i^2$ **d)** $\sum_{i=1}^{10} [6(\frac{X_i}{Y_i}) - 2(X_i^{Y_i})]$

7. Build the following table by using arrays where the table below converts the power to its value in decibels (dB) according to the relation:

$$\mathbf{G[dB] = 10 \log(P)}$$

Where the function log is the logarithm to base 10.

P	G[dB]
2	
1	
0.5	
0.1	
$\mathbf{10^{-3}}$	

EXPERIMENT-FOUR

FLOW CONTROL

MATLAB has several flow control constructs:

- if statement
- if- else statement
- if- elseif statement
- switch and case statements.
- for loops
- while loops
- continue statement
- break statement

1- if statement

The if statement evaluates a logical expression and executes a group of statements when the expression is *true*. The if statement form is:

```
if condition
   statement
end
```

or in simplest form we can put it in a single line:

```
if condition  statement, end
```

where *condition* is usually a logical *expression* , i.e. an expression containing a *relational operator,* and which is either *true* or *false.* These *relational operators* are:

```
<     less than
<=    less than or equal
==    equal
~=    not equal
>     greater than
```

>= greater than or equal

If *condition* is true, statement executed, but if condition is false, nothing happens. More complicated *logical expressions* can be constructed using the three *logical operators*:

& and
| or
~ not

Example: The quadratic equation:

$$ax^2 + bx + c = 0$$

has equal roots, given by $-b/2a$, provided that $b^2 - 4ac=0$ and $a \neq 0$. This translates to the following MATLAB statements:

```
a = input('enter the value of a:');
b = input('enter the value of b:');
c = input('enter the value of c:');
if (b^2 -4*a*c == 0) & (a ~= 0)
   disp('The quadratic equation has equal roots')
   x = -b / (2*a)
end
```

After running this program you will see that the command window wants you to enter a value for the variables a and b according to the input function that you put it in the first and second lines of the program. Then the command window will gives the result if the condition is satisfied.

condition may be a vector or matrix, so it is important to understand how relational operators and if statements work with matrices. When you want to check for equality between two variables, you might use

```
if A == B, ...
```

This is legal MATLAB code, and does what you expect when A and B are scalars. But when A and B are matrices, A == B does not test *if* they are equal, it tests *where* they are equal; the result is another matrix of 0's and 1's showing element-by-element equality. In fact, if A and B are not the same size, then A == B is an error.

The proper way to check for equality between two variables is to use the isequal function,

```
if isequal(A,B), ...
```

Several functions are helpful for reducing the results of matrix comparisons to scalar conditions for use with if, including

Isequal	True if arrays are numerically equal
Isempty	True for empty array.
all	True if all elements of a vector are nonzero.
any	True if any element of a vector is nonzero.

2- if-else statement

if-else statement keywords provide for the execution of alternate groups of statements. An end keyword, which matches the if, terminates the last group of statements. The groups of statements are delineated by the four keywords – no braces or brackets are involved. The basic form of if-else statement for use in a program file is:

```
if condition
   statement-1
else
   statement-2
end
```

or in simplest form :

```
if condition  statement-1, else statement-2,end
```

Example: the same example above will be repeated using if-else statement:

```
a = input('enter the value of a:');
b = input('enter the value of b:');
c = input('enter the value of c:');
if (b^2 -4*a*c == 0) & (a ~= 0)
```

```
    disp('The quadratic equation has equal roots')
    x = -b / (2*a)
else
    disp('The quadratic equation did not have equal roots')
end
```

3- if-elseif statement

The if-elseif statement executes groups of statements based on different expressions. So if our comparison contains many statements for many conditions then we must use the if-elseif statement. The basic form of this statement is:

```
if condition-1
   statement-1
elseif condition-2
   statement-2
.
.
.
elseif condition-N
   statement-N
else
  statement-N+1
end
```

Example: Suppose the random bank offers 9% interest on balances of less than $5000, 12% for balances of $5000 or more but less than $10000, and 15% for balances of $10000 or more. The following program calculates a customer's new balance after one year according to this scheme:

```
bal = input (' Enter bank balance:' );
if bal < 5000
   rate = 0.09;
elseif bal < 10000
```

```
  rate=0.12;
else
  rate = 0.15;
end
newbal = bal + rate * bal;
format bank
disp ('New balance is:')
disp (newbal)
```

4- switch and case statements

The switch statement executes groups of statements based on the value of a variable or expression. The keywords case and otherwise delineate the groups. Only the first matching case is executed. There must always be an end to match the switch. The general form of a while statement is:

```
switch variable
   case case_number_1,
      statement-1
   case case_number_2,
      statement-2
   .
   .
   .
   case case_number_N,
      statement-N
    otherwise,
      statement-N+1
 end
```

So the statements following the case statement are executed when the case number matches the variable value entry with the switch statement.

EXPERIMENT-FOUR: FLOW CONTROL

Note: Unlike the C language switch statement, MATLAB's switch does not fall through. If the first case statement is true, the other case statements do not execute. So, break statements are not required.

Example: Write a script file to enter an integer random numbers from 1 to 10 in a (3×3) matrix (named A). Find the requirements below depending on your entry from 1 to 4:

1. The transpose of matrix A.
2. The determinant of matrix A.
3. The inverse of matrix A.
4. The eigen values of matrix A

```
A=fix(rand(3,3)*10);
disp('your matrix is:')
A
n=input('Enter your choice from 1 to 4:')
switch n
   case 1
      disp('The transpose of matrix A is:')
      A'
   case 2
      disp('The determinant of matrix a is:')
      det(A)
   case 3
      disp('The inverse of matrix A is:')
      inv(a)
   case 4
      disp('The eigen values of matrix A is:')
      eig(a)
   otherwise
      disp('wrong number, enter another number')
end
```

5- for loops

The for loop repeats a group of statements a fixed, predetermined number of times. A matching end delineates the statements. The general form of a for statement is:

```
for variable = x : s : y,
    statement, ..., statement
end
```

The variable and then the following statements, up to the end, are executed. The expression of the form x : s : y are being to be a vector represents the beginning, x, and the end, y, of the loop by a step of, s. Note that you may not want s if the step size was 1.

Some examples will be listed here to get a brief knowledge a bout for loops:

Example: Suppose we want to find the factorial of n (n!):

```
n = input ('Enter any number:');
f=1;
for i = 1:n
   f = f * i ;
end
disp ('The factorial of this number is:')
f
```

Example: Find 100 values of x and y obtained from the difference equations:

$$\mathbf{x_{k+1} = y_k\,(1 + \sin(0.7x_k)) - 1.2\sqrt{|\,x_k\,|}}$$

$$\mathbf{y_{k+1} = 0.21 - x_k}$$

starting with $x_o=y_o=0$.

```
x(1)=0;y(1)=0;
for k=1:10
   x(k+1)=y(k)*(1+sin(0.7*x(k)))-1.2*sqrt(abs(x(k)));
   y(k+1)=0.21-x(k);
end
```

Note: MATLAB did not accept zero indices like x(0) or y(0) as an example.

6- while loop

The while loop repeats a group of statements an indefinite number of times under control of a logical condition. A matching end delineates the statements. The general form of a while statement is:

```
while expression
   statements
end
```

So the statements are executed while the real part of the expression has all non-zero elements. The expression is usually the result of a logical expressions (==, <, >, <=, >=, or ~=).

Example: Write a script file to find a solution for the polynomial ($x^3 + x - 3=0$) by using Newton's method. Give an initial guess to x and stop the program either when the absolute value of y(x) is less than 10^{-8}, or after 20 steps.

Hint: Newton's method used to solve a general equation y(x)=0 by repeating the assignment:

$$\mathbf{x_{k+1} = x_k - \frac{y(x_k)}{y'(x_k)}}$$

where $\mathbf{y'(x_k)}$ (i.e. $\frac{dy}{dx}$) is the first derivative of $\mathbf{y(x_k)}$. The process continues until $\mathbf{y(x_k)}$ is close enough to zero.

The solution of this problem will be as follow:

```
% Newtons Method
steps=0;
x=input('initial guess:')
y=x^3+x-3;
e=1e-8;
while(abs(y)>=e)&(steps<20)
```

```
    y=x^3+x-3;
    y_dash=3*x^2+1;
    x=x-(y/y_dash);
    steps=steps+1;
    disp([x y])
end
```

Note that there are two conditions that will stop the while loop: convergence, or the completion of 20 steps. Otherwise the script could run indefinitely.
Here is a sample run (with format long), starting with initial guess of x = 1.

```
x =
     1
  1.25000000000000  -1.00000000000000
  1.21428571428571   0.20312500000000
  1.21341217578282   0.00473760932945
  1.21341166276241   0.00000277908667
  1.21341166276223   0.00000000000096
```

7- continue statement

The continue statement passes control to the next iteration of the for or while loop in which it appears, skipping any remaining statements in the body of the loop. In nested loops, continue passes control to the next iteration of the for or while loop enclosing it.

Example: Write a script file to print the even elements in matrix A. Where:

$$\mathbf{A} = \begin{bmatrix} \mathbf{23} & \mathbf{11} & \mathbf{12} & \mathbf{34} \\ \mathbf{42} & \mathbf{56} & \mathbf{2} & \mathbf{9} \\ \mathbf{77} & \mathbf{82} & \mathbf{52} & \mathbf{21} \\ \mathbf{12} & \mathbf{10} & \mathbf{33} & \mathbf{2} \end{bmatrix}$$

```
a=[23 11 12 34;42 56 2 9;77 82 52 21;12 10 33 2];
for i=1:4
   for j=1:4
      if rem(a(i,j),2)~=0
         continue
      end
      disp(a(i,j))
   end
end
```

8- break statement

The break statement lets you exit early from a for or while loop. In nested loops, break exits from the innermost loop only.

Example: Write a script file to find a solution for the exponential series below.

$$\mathbf{e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...+ \frac{x^n}{n!}}$$

Make the output precision be: 0.0001

Hint: The program will stopped when the value of $(\frac{\mathbf{x^n}}{\mathbf{n!}})$ reached to 0.0001 even when the counter not reached
its final value.

```
% This script is used to find the solution of the exponential series exp(x)
x=input('Enter the value of x:')
n=input('Enter the highest exponent (n):')
s=0;
for i=1:n
   f=1;
   for j=1:i;
      f=f*j;
```

```
    end
    f1=f;
    e=x^i/f1;
    if e<0.0001
       break
    end
    s=s+e;
end
disp('the result is:')
s
```

Exercises:

1. If C and F are the Celsius and Fahrenheit temperature respectively, the formula for conversion from Celsius to Fahrenheit is:

$$\mathbf{F = (9C / 5) + 32}$$

Write a script which will ask you for the Celsius temperature and display the equivalent Fahrenheit one with the following comments:

"Cold" when $F \leq 41$.
"Nice" when $41 < F \leq 77$.
"Hot" when $F > 77$.

2. Set up any 4×4 matrices A & B. Write some statements to execute the following statements:

a) A+B if A = B.
b) A^2+B^2 if |A| > |B|.
c) $\sqrt{A} + (\sqrt{B})^3$ if all the eigen values of A are nonzero.

3. Write a program to compute the below functions depending on your entry from 1 to 3:

1. **x(t) = sin(t)+tan(t)**
2. **x(t) = cosh(t)**
3. **x(t) = $\tan^{-1}$(4t)**

Use the interval $-2\pi \le t \le 2\pi$ in steps of $\pi/2$.

4. Write a script file to find **y** with respect to all variables:

a) $$y = \frac{n!}{(n-r)!}$$

b) $$y = \sum_{k=1}^{100} \frac{3x_k}{(2x_k + x_k^2)}$$

5. When a resistor (R), capacitor (C) and battery (V) are connected in series, a charge Q builds up on the capacitor according to the formula:

$$Q(t) = CV\,(1-e^{-t/RC})$$

If there is no charge on the capacitor at time t=0. The problem is to monitor the charge on the capacitor every 0.1 seconds in order to detect when it reaches a level of 8 units, given that V=9, R=4 and C=1. Write a program which displays the time and charge every 0.1 seconds until the charge first exceeds 8 units (i.e. the last charge displayed must exceed 8).

6. A square wave of period T may be defined by the function

$$f(t) = \begin{cases} 1 & (0 < t < T) \\ -1 & (-T < t < 0) \end{cases}$$

The Fourier series for f(t) is given by:

$$F(t) = \frac{4}{\pi} \sum_{k=0}^{\infty} \frac{1}{2k+1} \sin\left[\frac{(2k+1)\pi t}{T}\right]$$

It is of interest to know how many terms are needed for a good approximation to this infinity sum. Taking T=1, write a program to compute and display the sum to n terms of the series for t from 0 to 1 in steps of 0.1, say. Run the program for different values of n, e.g. 1, 3, 6, etc.

7. Write a program to compute a table of the function

$$\mathbf{f(x) = x\sin\left[\frac{\pi(1+20x)}{2}\right]}$$

over the closed interval [-1,1] using increments in x of (a) 0.2 (b) 0.1 and (c) 0.01.

8. One of the fastest series for (π/4) is:

$$\mathbf{\frac{\pi}{4} = 6\tan^{-1}\left[\frac{1}{8}\right] + 2\tan^{-1}\left[\frac{1}{57}\right] + \tan^{-1}\left[\frac{1}{239}\right]}$$

Use the series below to compute $\tan^{-1}(x)$:

$$\mathbf{\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \ldots.. + \frac{x^{99}}{99}}$$

9. Find 14 values of **S** and **R** obtained from the difference equations:

$$\mathbf{S_{k+1} = R_k\,(1/\cos(0.3S_k))}$$

$$\mathbf{R_{k+1} = 0.4S_k + S_k^2}$$

starting with S_o=R_o=1.

10. Write a script file to find a solution for the polynomial ($\boldsymbol{x^4+2x^2+4x-5=0}$) by using Newton's method. Give an initial guess to x and stop the program either when the absolute value of f(x) is less than 10^{-5}, or after 100 steps.

11. Write a script file to print the odd elements in matrix B. Where:

$$\mathbf{B} = \begin{bmatrix} 3 & 1 & 32 & 54 \\ 9 & 44 & 20 & 98 \\ 72 & 12 & 55 & 31 \\ 87 & 90 & 23 & 2 \end{bmatrix}$$

12. Write a script file to find a solution for the exponential series below.

$$\mathbf{\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots.. + (-1)^{n+1}\cdot\frac{x^{2n-1}}{(2n-1)!}}$$

Input x and make the output precision be: 10^{-6}

EXPERIMENT-FIVE

GRAPHICS

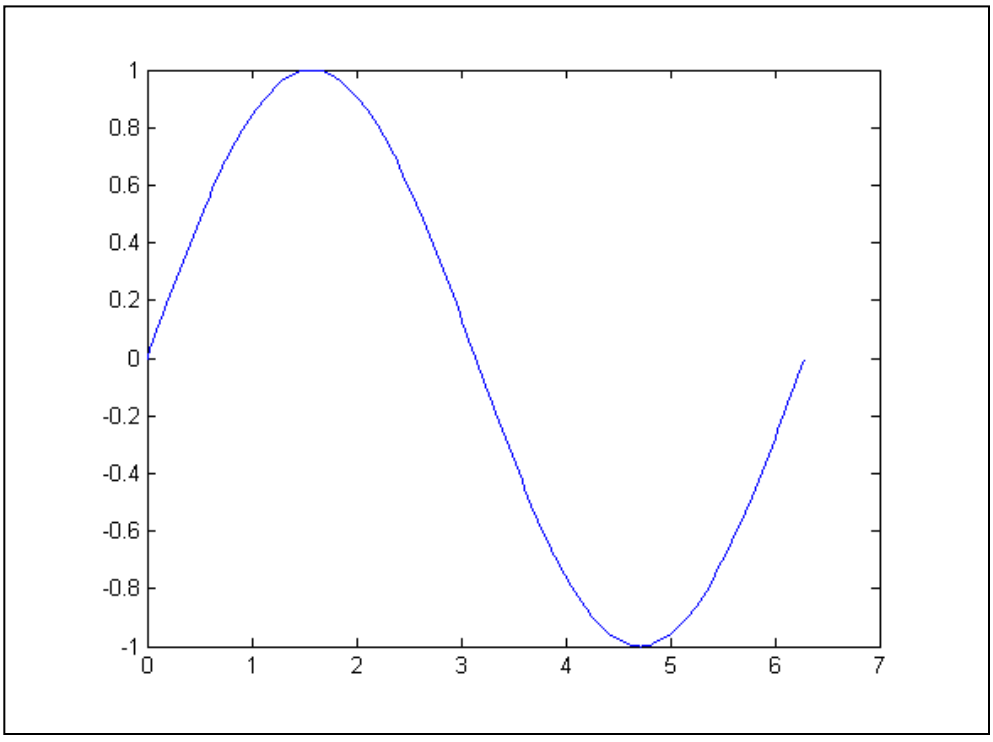
MATLAB has extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs. This chapter describes a few of the most important graphics functions and provides examples of some typical applications.

1- Creating a Plot

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

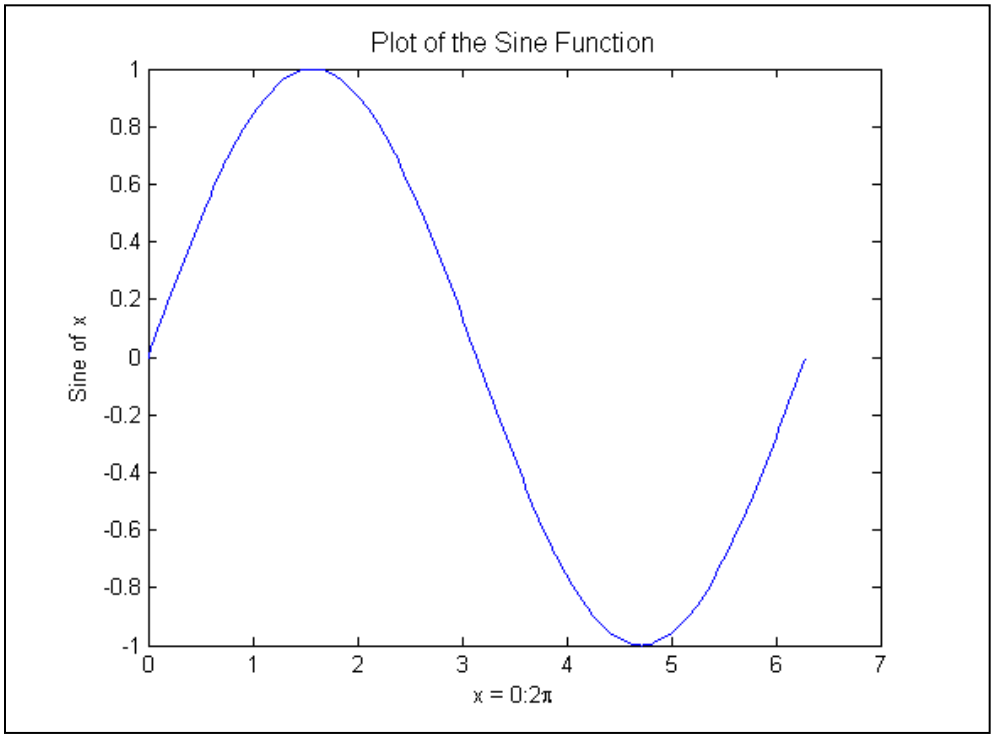
For example, these statements use the colon operator to create a vector of x values ranging from zero to 2π, compute the sine of these values, and plot the result.

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



Now label the axes and add a title. The characters \pi create the symbol π in the plot.

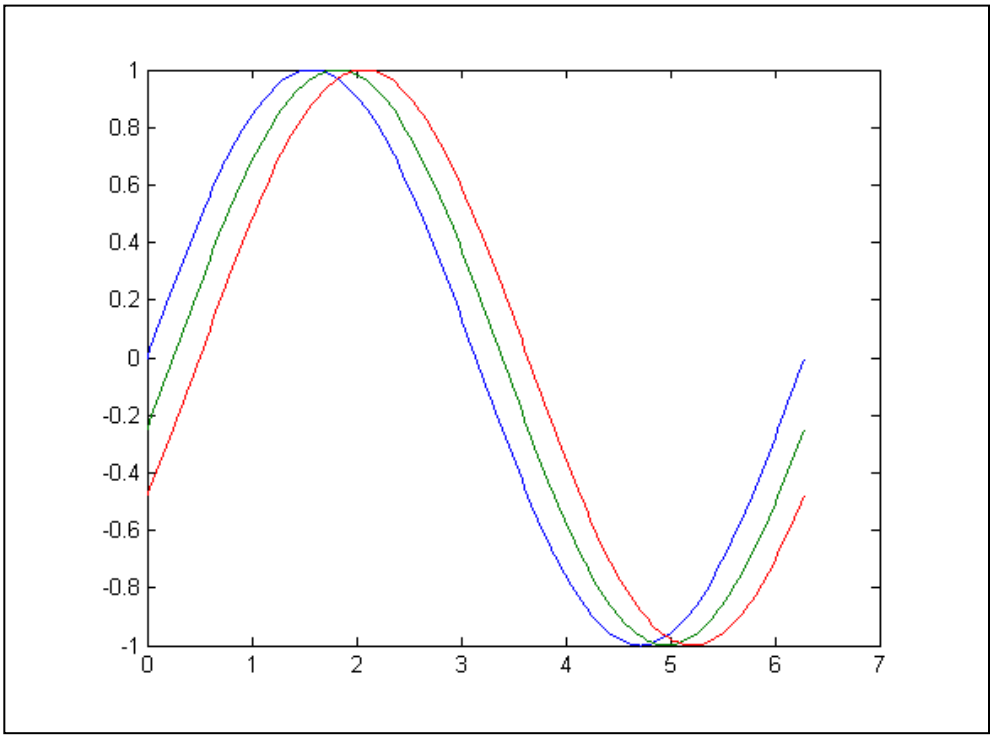
```
xlabel('x = 0:2\pi')
ylabel('Sine of x')
title('Plot of the Sine Function')
```



2- Multiple Data Sets in One Graph

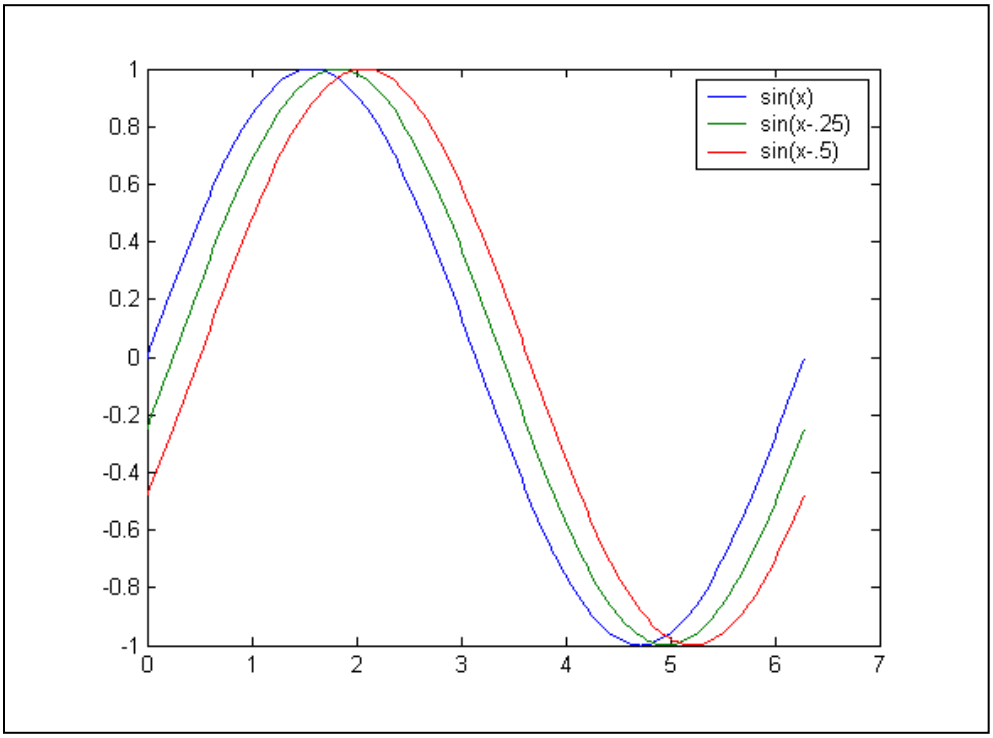
Multiple x-y pair arguments create multiple graphs with a single call to plot. MATLAB automatically cycles through a predefined (but user settable) list of colors to allow discrimination between each set of data. For example, these statements plot three related functions of x, each curve in a separate distinguishing color.

```
x = 0:pi/100:2*pi;
y = sin(x);
y2 = sin(x-0.25);
y3 = sin(x-0.5);
plot(x,y,x,y2,x,y3)
```



The legend command provides an easy way to identify the individual plots.

legend('sin(x)','sin(x-.25)','sin(x-.5)')



3- Specifying Line Styles and Colors

It is possible to specify color, line styles, and markers (such as plus signs or circles) when you plot your data using the plot command.

```
plot(x,y,'color_style_marker')
```

color_style_marker is a string containing from one to four characters (enclosed in single quotation marks) constructed from a color, a line style, and a marker type:

- Color strings are 'c', 'm', 'y', 'r', 'g', 'b', 'w', and 'k'. These correspond to cyan, magenta, yellow, red, green, blue, white, and black.
- Linestyle strings are '-' for solid, '--' for dashed, ':' for dotted, '-.' for dash-dot, and 'none' for no line.
- The marker types are '+', 'o', '*', and 'x' and the filled marker types 's' for square, 'd' for diamond, '^' for up triangle, 'v' for down triangle, '>' for right triangle, '<' for left triangle, p' for pentagram, 'h' for hexagram, and none for no marker.

For example,

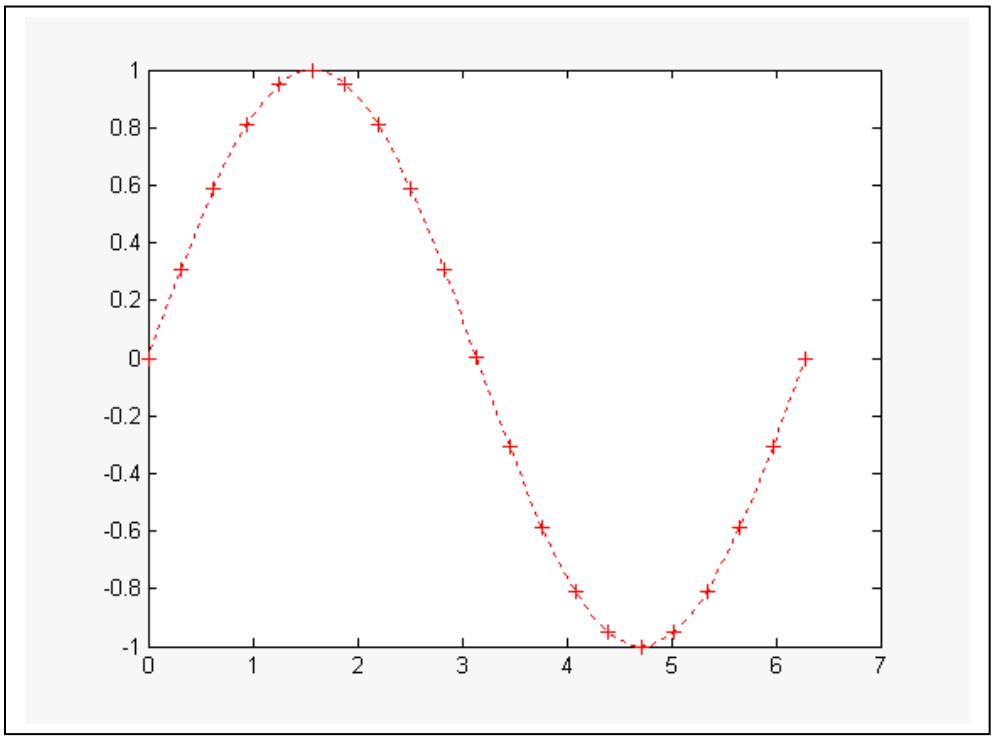
```
plot(x,y,'ko')
```

plots black circles at each data point, but does not connect the markers with a line. The statement

```
plot(x,y,'r:+')
```

plots a red dotted line and places plus sign markers at each data point. You may want to use fewer data points to plot the markers than you use to plot the lines. This example plots the data twice using a different number of points for the dotted line and marker plots.

```
x1 = 0:pi/100:2*pi;
x2 = 0:pi/10:2*pi;
plot(x1,sin(x1),'r:',x2,sin(x2),'r+')
```



4- Imaginary and Complex Data

When the arguments to plot are complex, the imaginary part is ignored *except* when plot is given a single complex argument. For this special case, the command is a shortcut for a plot of the real part versus the imaginary part. Therefore,

```
plot(Z)
```

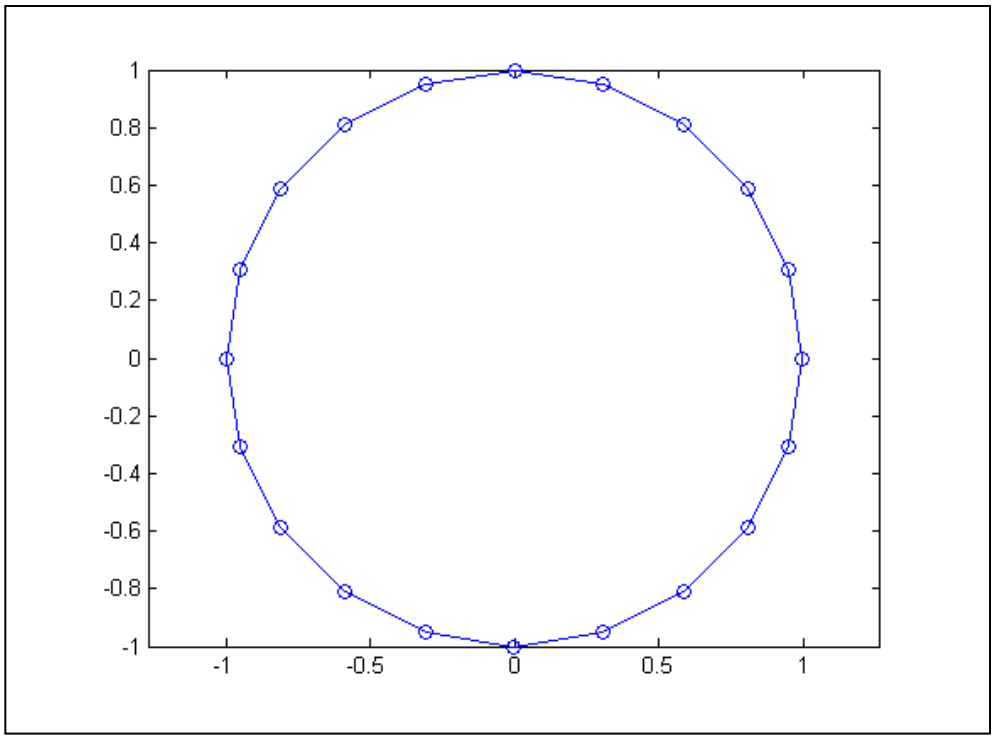
where Z is a complex vector or matrix, is equivalent to

```
plot(real(Z),imag(Z))
```

For example,

```
t = 0:pi/10:2*pi;
plot(exp(i*t),'-o')
axis equal
```

draws a 20-sided polygon with little circles at the vertices. The command, axis equal, makes the individual tick mark increments on the *x*- and *y*-axes the same length, which makes this plot more circular in appearance.



5- Adding Plots to an Existing Graph

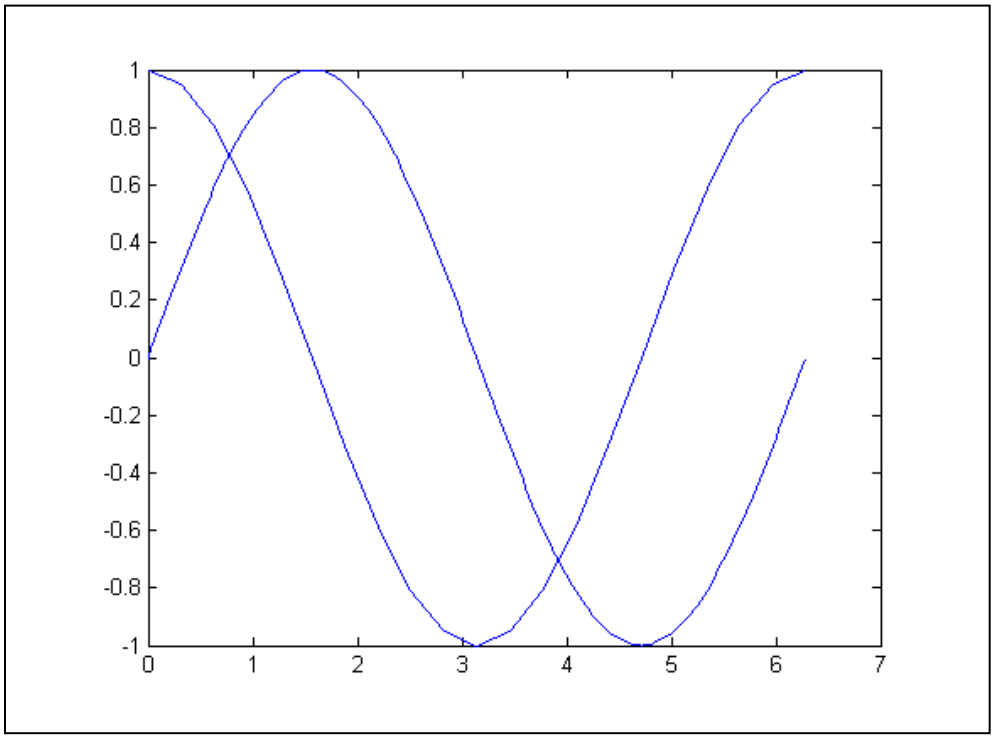
The hold command enables you to add plots to an existing graph. When you type

```
hold on
```

MATLAB does not replace the existing graph when you issue another plotting command; it adds the new data to the current graph, rescaling the axes if necessary. For example, these statements first create a contour plot of the peaks function, then superimpose a pseudocolor plot of the same function.

```
x = 0:pi/100:2*pi;
y = 0:pi/10:2*pi;
plot(x,sin(x))
hold on
plot(y,cos(y))
hold off
```

The hold on command causes the sin(x) plot to be combined with the cos(y) plot in one figure.



6- Figure Windows

Graphing functions automatically open a new figure window if there are no figure windows already on the screen. If a figure window exists, MATLAB uses that window for graphics output. If there are multiple figure windows open, MATLAB targets the one that is designated the "current figure" (the last figure used or clicked in).

To make an existing figure window the current figure, you can click the mouse while the pointer is in that window or you can type

```
figure(n)
```

where n is the number in the figure title bar. The results of subsequent graphics commands are displayed in this window.

To open a new figure window and make it the current figure, type

```
figure
```

7- Controlling the Axes

The axis command supports a number of options for setting the scaling, orientation, and aspect ratio of plots. You can also set these options interactively.

7.1- Setting Axis Limits:

By default, MATLAB finds the maxima and minima of the data to choose the axis limits to span this range. The axis command enables you to specify your own limits

```
axis([xmin xmax ymin ymax])
```

or for three-dimensional graphs,

```
axis([xmin xmax ymin ymax zmin zmax])
```

Use the command `axis auto` to re-enable MATLAB's automatic limit selection.

7.2- Setting Axis Aspect Ratio:

`axis` also enables you to specify a number of predefined modes. For example,

```
axis square
```

makes the *x*-axes and *y*-axes the same length and

```
axis equal
```

makes the individual tick mark increments on the *x*- and *y*-axes the same length. This means

```
plot(exp(i*[0:pi/10:2*pi]))
```

followed by either `axis square` or `axis equal` turns the oval into a proper circle.

```
axis auto normal
```

returns the axis scaling to its default, automatic mode.

7.3- Setting Axis Visibility:

You can use the axis command to make the axis visible or invisible.

```
axis on
```

makes the axis visible. This is the default.

```
axis off
```

makes the axis invisible.

7.4- Setting Grid Lines:

The grid command toggles grid lines on and off. The statement

```
grid on
```

turns the grid lines on and

```
grid off
```

turns them back off again.

8- Multiple Plots in One Figure

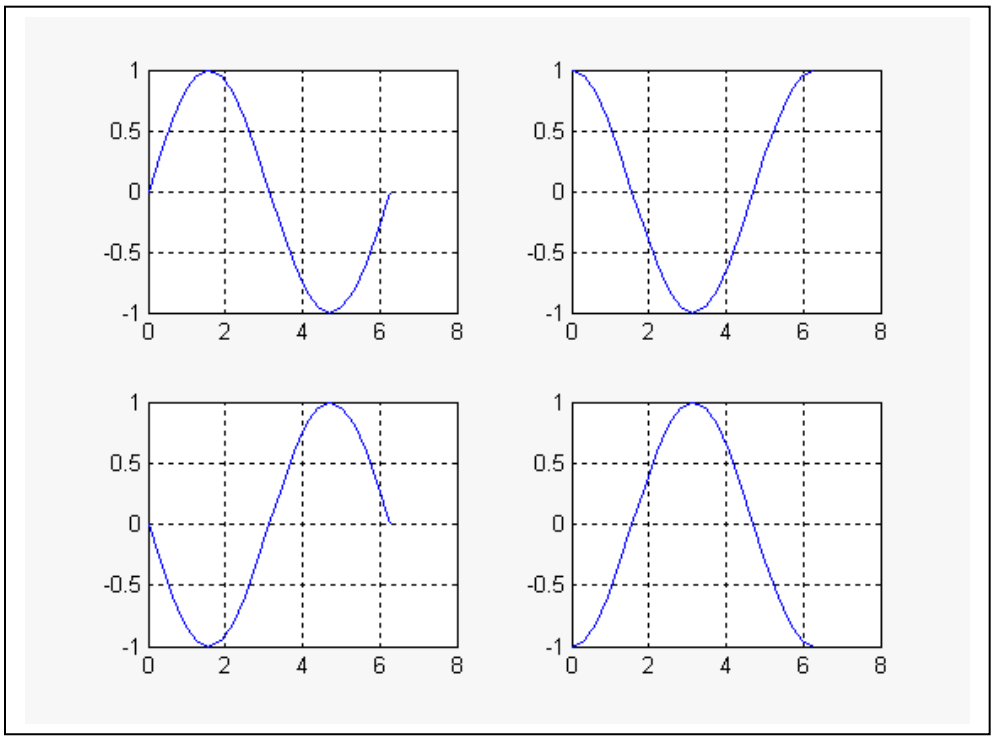
The subplot command enables you to display multiple plots in the same window or print them on the same piece of paper. Typing

```
subplot(m,n,p)
```

partitions the figure window into an m-by-n matrix of small subplots and selects the pth subplot for the current plot. The plots are numbered along first the top row of the figure window, then the second row, and so on. For example, these statements plot data in four different subregions of the figure window.

```
t = 0:pi/10:2*pi;
subplot(2,2,1),plot(t,sin(t)),grid
subplot(2,2,2),plot(t,cos(t)),grid
```

```
subplot(2,2,3),plot(t,sin(t+pi)),grid
subplot(2,2,4),plot(t,cos(t+pi)),grid
```



9- 3-D Plots

MATLAB has a variety of functions for displaying and visualizing data in 3-D, either as lines in 3-D (plot3 function), or as a wire frame (mesh function) and surfaces (surf function). This section provides a brief overview.

9.1- plot3 function:

The function plot3 is the 3-D version of plot. The command

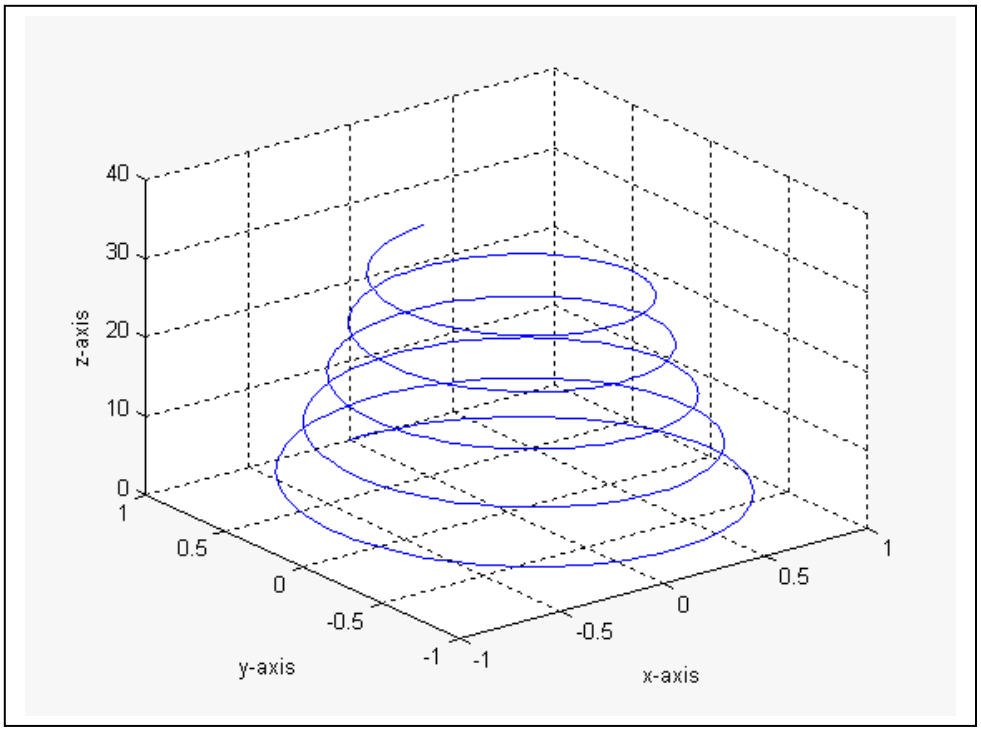
plot3 (x, y, z)

draws 2-D projection of a line in 3-D through the points whose coordinates are the elements of the vectors x, y, and z.

For example:

```
t = 0 : pi/50 : 10*pi ;
plot3 (exp(-0.02*t).* sin(t) , exp(-0.02*t).* cos(t) , t), …
xlabel ('x-axis'),ylabel ('y-axis') , zlabel ('z-axis') , grid
```

produce the inwardly spiraling helix shown below:



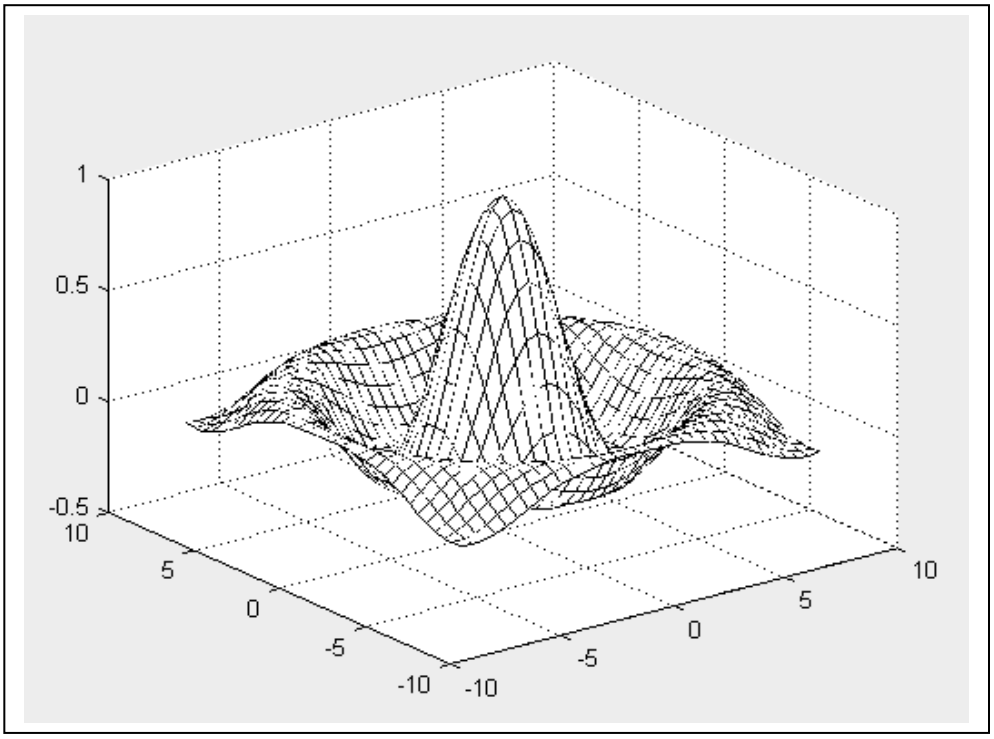
9.2- Visualizing functions of two variables (mesh function):

mesh function enable you to plot a 3-D mesh surface. The instruction mesh(z) displays a function of two variables, $z = f(x,y)$ after generating X and Y matrices consisting of repeated rows and columns, respectively, over the domain of the function.

The meshgrid function transforms the domain specified by a single vector or two vectors x and y into matrices X and Y for use in evaluating functions of two variables. The rows of X are copies of the vector x and the columns of Y are copies of the vector y.

The following example evaluates and graphs the two-dimensional sinc function, sin(r)/r, between the x and y directions. R is the distance from origin, which is at the center of the matrix. Adding eps (a MATLAB command that returns the smallest floating-point number on your system) avoids the indeterminate 0/0 at the origin.

```
[X,Y] = meshgrid(-8:0.5:8, -8:0.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
mesh(Z)
```

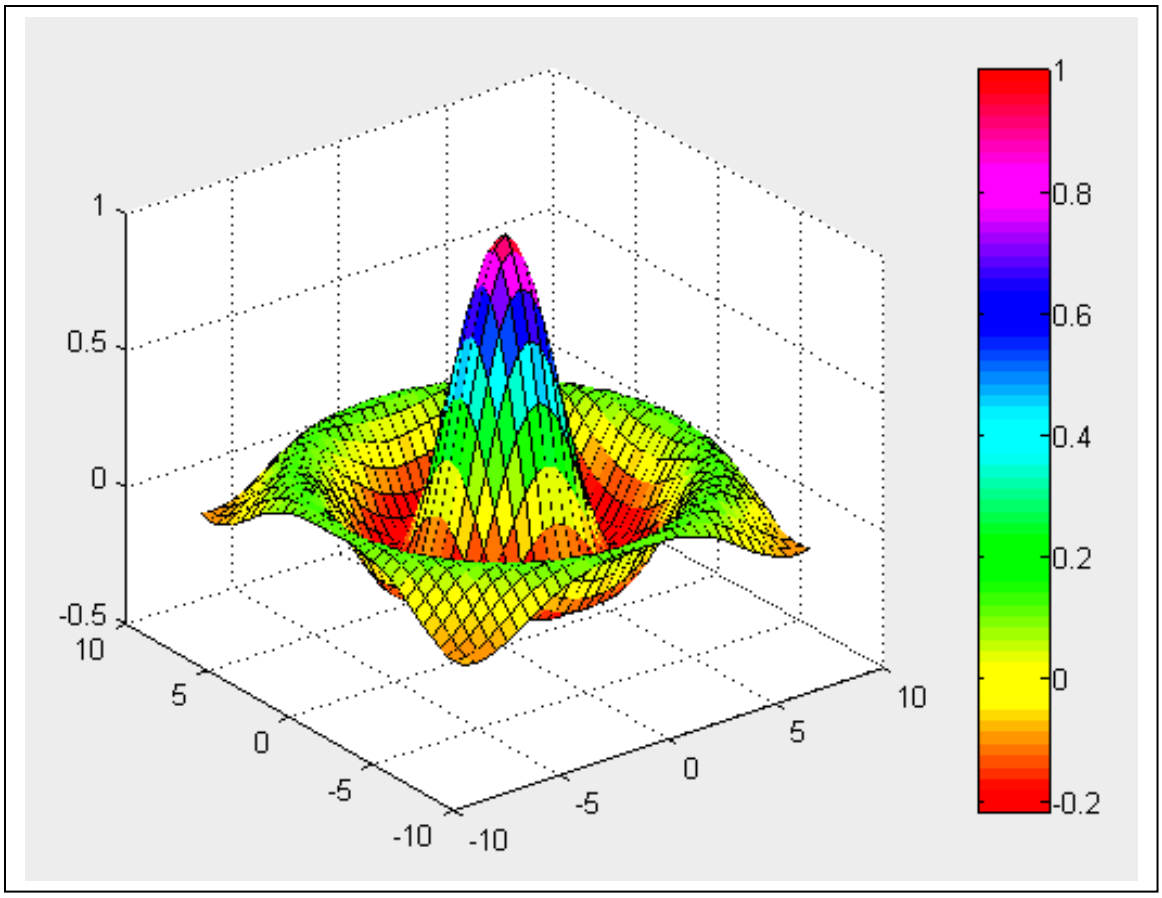


9.3- Colored Surface Plots (surf function):

A surface plot is similar to a mesh plot except the rectangular faces of the surface are colored. The color of the faces is determined by the values of Z and the colormap (a colormap is an ordered list of colors). These statements graph the sinc function as a surface plot, select a colormap, and add a color bar to show the mapping of data to color.

```
[X,Y] = meshgrid(-8:0.5:8, -8:0.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
surf(X,Y,Z)
```

colormap hsv
colorbar



Exercises:

1. Plot the following functions. Grid the plots, label the axis, and put a suitable title on the graphs.

a) y = tan(x) **$-3\pi/2 \le x \le 3\pi/2$** **step $\pi/100$.**
b) y = sinc(x) **$-4\pi \le x \le 4\pi$** **step $\pi/20$.**
c) y = e^x **$0 \le x \le 4$** **step 0.1.**
d) y = $\sin^{-1}(x)$ **$-\pi/2 \le x \le \pi/2$** **step $\pi/10$.**

2. Split the plotting window into four windows and place the plots you obtained in question-1 on each window.
3. Plot the inverse hyperbolic sine function and the inverse hyperbolic cosine function in the same graph (ranges from -10π to 10π in steps of π/10). Use a red dashed line for the first function and yellow plus line for the second function.
4. plot the below function in the range 0 to 3π in steps of π/20.

$$y(x)=\begin{cases}\sin(x) & \sin(x)\geq 0\\ 0 & \sin(x)\leq 0\end{cases}$$

5. Generate 10 normally distributed random points in 3-D space, and join them with lines in one 3-D graph.
6. Plot the surface $\mathbf{z = x^2 + y^2}$ with a finer mesh (of 0.25 units in each direction),using

   [x,y] = meshgrid (0:0.25:5 , 0:0.25:5).

7. The initial heat distribution over a steel plate is given by the function:

$$u(x,y)=80y^2e^{-x^2-0.3y^2}$$

   Plot the surface ***u*** over the grid defined by:

$$-2.1\leq x\leq 2.1\ ,\ -6\leq y\leq 6$$

   where the grid width is 0.15 in both directions.

10- MATLAB Commands Review

axis Sets the axis limits for both 2-D and 3-D plots. Axis supports the arguments equal and square, which makes the current graphs aspect ratio 1.

contour Plots contour lines of a surface.

clear Clears all variables from the workspace.

clf Clears figure.

for Runs a sequence of commands a given number of times.

getframe Returns the pixel image of a movie frame.

help Online help.

hold on(off) Holds the plot axis with existing graphics on, so that multiple figures can be plotted on the same graph (release the hold of the axes).

if Conditional evaluation.

length Gives the length of an array.

load Loads data or variable values from previous sessions into current MATLAB session.

linspace Generates an array with a specified number of points between two values.

meshgrid Makes a 2-D array of coordinate squares suitable for plotting surface meshes.

mesh Plots a mesh surface of a surface stored in a matrix.

meshc The same as mesh, but also plots in the same figure the contour plot.

min Finds the smallest element of an array.

max Finds the largest element of an array.

mean Finds the mean of the elements of an array.

moviein Creates the matrix that contains the frames of an animation.

movie Plays the movie described by a matrix M.

orient Orients the current graph to your needs.

plot Plots points or pairs of arrays on a 2-D graph.

plot3 Plots points or array triples on a 3-D graph.

polar Plots a polar plot on a polar grid.

pol2cart Polar to Cartesian conversion.

print Prints a figure to the default printer.

quit or **exit** Leave MATLAB program.

rand Generates an array with elements randomly chosen from the uniform distribution over the interval [0, 1].

randn Generates an array with elements randomly chosen from the normal distribution function with zero mean and standard deviation 1.

subplot Partitions the graphics window into sub-windows.

save Saves MATLAB variables.

std Finds the standard deviation of the elements of an array.

stem Plots the data sequence as stems from the *x*-axis terminated with circles for the data value.

view Views 3-D graphics from different perspectives.

who Lists all variables in the workspace.

xlabel, ylabel, zlabel, title Labels the appropriate axes with text and title.

(x>=x1) Boolean function that is equal to 1 when the condition inside the arenthesis is satisfied, and zero otherwise.

LECTURE-SEVEN

FUNCTIONS

1- Functions

A function is a group of statements that together perform a task. In MATLAB, functions are defined in separate files. The name of the file and of the function should be the same.

Functions operate on variables within their own workspace, which is also called the **local workspace**, separate from the workspace you access at the MATLAB command prompt which is called the **base workspace**.

Functions can accept more than one input arguments and may return more than one output arguments.

Syntax of a function statement is:

```
function [out1,out2, ..., outN] = myfun(in1,in2,in3, ..., inN)
….
end
```

Example: The following function named `mymax` should be written in a file named `mymax.m`. It takes five numbers as argument and returns the maximum of the numbers. Create a function file, named `mymax.m` and type the following code in it:

```
function max = mymax(n1, n2, n3, n4, n5)
% This function calculates the maximum of the
% five numbers given as input
max =  n1;
if(n2 > max)
   max = n2;
end
if(n3 > max)
   max = n3;
end
if(n4 > max)
   max = n4;
end
if(n5 > max)
   max = n5;
end
```

The first line of a function starts with the keyword `function`. It gives the name of the function and order of arguments. In our example, the `mymax` function has five input arguments and one output argument.
The comment lines that come right after the function statement provide the help text. These lines are printed when you type:

```
help mymax
```

MATLAB will execute the above statement and return the following result:

```
This function calculates the maximum of the
five numbers given as input
```

You can call the function as:

```
mymax(34, 78, 89, 23, 11)
```

MATLAB will execute the above statement and return the following result:

```
ans =
     89
```

2- Multiple output arguments

In its general form MATAB functions can return more than one output argument. The default value assigned by the first argument as a return value for the function. But still other arguments can be assigned to variables in the main program.

Example: Let us write a function named quadratic that would calculate the roots of a quadratic equation. The function would take three inputs, the quadratic co-efficient, the linear co-efficient and the constant term. It would return the roots. The function file quadratic.m contains the following:

```
function [x1,x2] = quadratic(a,b,c)
%this function returns the roots of
% a quadratic equation.
% It takes 3 input arguments
% which are the co-efficients of x2, x and the
%constant term
% It returns the roots
d = sqrt(b^2 - 4*a*c); % the discriminant
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end   % end of quadratic
```

calling the function form the command prompt as:

```
quadratic(2,4,-4)
```

MATLAB will execute the above statement and return the following result:

```
ans =
      0.7321
```

This answer is the value of the first argument x1. To assign both output arguments to variables in the base workspace of MATLAB we can call the function as:

```
[p,q] = quadratic(2,4,-4)
```

MATLAB will execute the above statement and return the following result:

```
p =
    0.7321

q =
   -2.7321
```

In this case the value of the first output argument `x1` is assigned to the first variable `p`, and the value of the second output argument `x2` is assigned to the second variable `q`.The same way we can assign only the second output argument to a variable as:

```
[~,q] = quadratic(2,4,-4)
```

MATLAB will execute the above statement and return the following result:

```
q =
   -2.7321
```

3- Primary and Sub-Functions

Any function other than an anonymous function must be defined within a file. Each function file contains a required primary function that appears first and any number of optional sub-functions that comes after the primary function and used by it.
Primary functions can be called from outside of the file that defines them, either from command line or from other functions, but sub-functions cannot be called from command line or other functions, outside the function file.
Sub-functions are visible only to the primary function and other sub-functions within the function file that defines them.

Example: In this case, the function file `quadratic.m` will contain the primary function `quadratic` and the sub-function `dis`, which calculates the discriminant.
Create a function file quadratic.m and type the following code in it:

```
function [x1,x2] = quadratic(a,b,c)

%this function returns the roots of
% a quadratic equation.
% It takes 3 input arguments
% which are the co-efficients of x2, x and the
%constant term
% It returns the roots
d = disc(a,b,c);
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end   % end of quadratic

function dis = disc(a,b,c)
%function calculates the discriminant
dis = sqrt(b^2 - 4*a*c);
end   % end of sub-function
```

You can call the above function from command prompt as:

```
quadratic(2,4,-4)
```

MATLAB will execute the above statement and return the following result:

```
ans =
      0.7321
```

4- Nested Functions

You can define functions within the body of another function. These are called nested functions. A nested function contains any or all of the components of any other function. Nested functions are defined within the scope of another function and they share access to the containing function's workspace.

A nested function follows the following syntax:

```
function x = A(p1, p2)
...
B(p2)
   function y = B(p3)
   ...
   end
...
end
```

Example: Let us rewrite the function quadratic, from previous example, however, this time the disc function will be a nested function.
Create a function file quadratic2.m and type the following code in it −

```
function [x1,x2] = quadratic2(a,b,c)
function disc  % nested function
d = sqrt(b^2 - 4*a*c);
end   % end of function disc

disc;
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end   % end of function quadratic2
```

You can call the above function from command prompt as:

```
quadratic2(2,4,-4)
```

MATLAB will execute the above statement and return the following result:

```
ans =
     0.73205
```

5- Private Functions

A private function is a primary function that is visible only to a limited group of other functions. If you do not want to expose the implementation of a function(s), you can create them as private functions.
Private functions reside in subfolders with the special name private.
They are visible only to functions in the parent folder.
Example: Let us rewrite the quadratic function. This time, however, the disc function calculating the discriminant, will be a private function.
Create a subfolder named private in working directory. Store the following function file disc.m in it:

```
function dis = disc(a,b,c)
%function calculates the discriminant
dis = sqrt(b^2 - 4*a*c);
end      % end of sub-function
```

Create a function quadratic3.m in your working directory and type the following code in it:

```
function [x1,x2] = quadratic3(a,b,c)

%this function returns the roots of
% a quadratic equation.
% It takes 3 input arguments
% which are the co-efficient of x2, x and the
%constant term
% It returns the roots
d = disc(a,b,c);

x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end      % end of quadratic3
```

You can call the above function from command prompt as:

```
quadratic3(2,4,-4)
```

MATLAB will execute the above statement and return the following result:

```
ans =  0.73205
```

6- Global Variables

Global variables can be shared by more than one function. For this, you need to declare the variable as global in all the functions.

If you want to access that variable from the base workspace, then declare the variable at the command line.

The global declaration must occur before the variable is actually used in a function. It is a good practice to use capital letters for the names of global variables to distinguish them from other variables.

Example: Let us create a function file named average.m and type the following code in it

```
function avg = average(nums)
global TOTAL
avg = sum(nums)/TOTAL;
end
```

Create a script file and type the following code in it:

```
global TOTAL;
TOTAL = 10;
n = [34, 45, 25, 45, 33, 19, 40, 34, 38, 42];
av = average(n)
```

When you run the file, it will display the following result:

```
av =
      35.500
```

7- Anonymous Functions

An anonymous function is like an inline function in traditional programming languages, defined within a single MATLAB statement. It consists of a single MATLAB expression and any number of input and output arguments.

You can define an anonymous function right at the MATLAB command line or within a function or script.

This way you can create simple functions without having to create a file for them.

The syntax for creating an anonymous function from an expression is:

```
f = @(arglist)expression
```

Example: In this example, we will write an anonymous function named power, which will take two numbers as input and return first number raised to the power of the second number. Create a script file and type the following code in it:

```
power = @(x, n) x.^n;
result1 = power(7, 3)
result2 = power(49, 0.5)
result3 = power(10, -10)
result4 = power (4.5, 1.5)
```

When you run the file, it displays:

```
result1 =
      343
result2 =
      7
result3 =
      1.0000e-10
result4 =
      9.5459
```

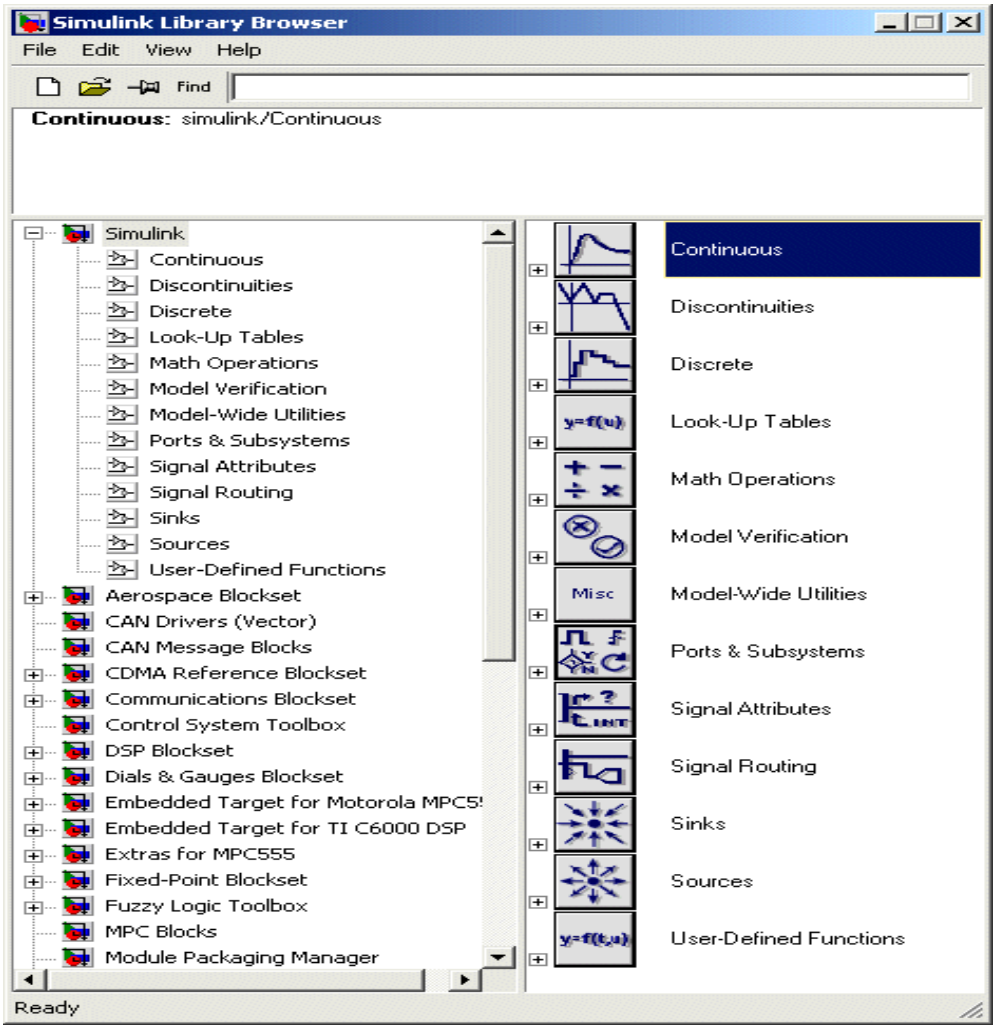
EXPERIMENT-SIX
SIMULINK

Simulink is a software package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates. For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. With this interface, you can draw the models just as you would with pencil and paper (or as most textbooks depict them). This is a far cry from previous simulation packages that require you to formulate differential equations and difference equations in a language or program. Simulink includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors. You can also customize and create your own blocks.

1- Starting Simulink

To start Simulink, you must first start MATLAB. Consult your MATLAB documentation for more information. You can then start Simulink in two ways:

- Click the Simulink icon  on the MATLAB toolbar.
- Enter the simulink command at the MATLAB prompt.



The Library Browser displays a tree-structured view of the Simulink block libraries installed on your system. You can build models by copying blocks from the Library Browser into a model window. Some of the important library blocks and its purposes are illustrated in the following subsections. Also you can apply the other blocks depending on your requirements.

1.1- Sinks library:

The *Sinks* library contains blocks that display or write block output.

Block Name	Purpose
Display	Show the value of the input.
Outport	Create an output port for a subsystem or an external output.
Scope, Floating Scope	Display signals generated during a simulation.
Stop Simulation	Stop the simulation when the input is nonzero.
Terminator	Terminate an unconnected output port.
To File	Write data to a file.
To Workspace	Write data to a variable in the workspace.
XY Graph	Display an X-Y plot of signals using a MATLAB figure window.

1.2- Sources library:

The *Sources* library contains blocks that generate signals.

Block Name	Purpose
Band-Limited White Noise	Introduce white noise into a continuous system.
Chirp Signal	Generate a sine wave with increasing frequency.
Clock	Display and provide the simulation time.
Constant	Generate a constant value.
Digital Clock	Generate simulation time at the specified sampling interval.
From File	Read data from a file.
From Workspace	Read data from a variable defined in the workspace.
Ground	Ground an unconnected input port.
Inport	Create an input port for a subsystem or an external input.
Pulse Generator	Generate pulses at regular intervals.
Ramp	Generate a constantly increasing or decreasing signal.
Random Number	Generate normally distributed random numbers.
Repeating Sequence	Generate a repeatable arbitrary signal.
Signal Builder	Generate an arbitrary piecewise linear signal.
Signal Generator	Generate various waveforms.
Sine Wave	Generate a sine wave.
Step	Generate a step function.
Uniform Random Number	Generate uniformly distributed random numbers.

1.3- Continuous library:

The Continuous library contains blocks that model linear functions.

Block Name	Purpose
Derivative	Output the time derivative of the input.
Integrator	Integrate a signal.
State-Space	Implement a linear state-space system.
Transfer Fcn	Implement a linear transfer function.
Transport Delay	Delay the input by a given amount of time.
Variable Transport Delay	Delay the input by a variable amount of time.
Zero-Pole	Implement a transfer function specified in terms of poles and zeros.

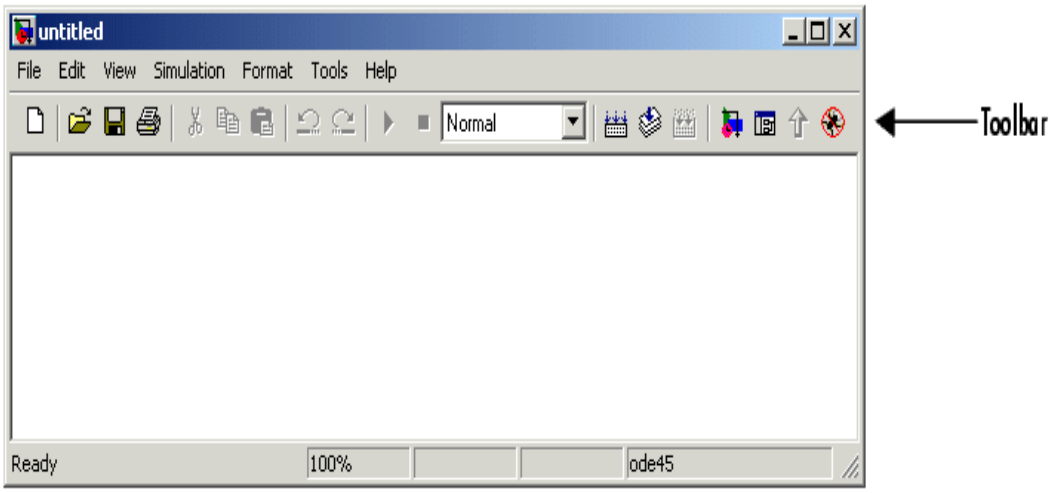
1.4- Discrete library:

The *Discrete* library contains blocks that represent discrete-time functions.

Block Name	Purpose
Discrete Filter	Implement IIR and FIR filters.
Discrete State-Space	Implement a discrete state-space system.
Discrete Transfer Fcn	Implement a discrete transfer function.
Discrete Zero-Pole	Implement a discrete transfer function specified in terms of poles and zeros.
Discrete-Time Integrator	Perform discrete-time integration of a signal.
First-Order Hold	Implement a first-order sample-and-hold.
Memory	Output the block input from the previous time step.
Unit Delay	Delay a signal one sample period.
Zero-Order Hold	Implement zero-order hold of one sample period.

2 Creating a New Model

To create a new model, click the **New** button on the Library Browser's toolbar (Windows only) or choose **New** from the library window's **File** menu and select **Model**. You can move the window as you do other windows. The figure below shows the new model that you open it.



3- Connecting Blocks

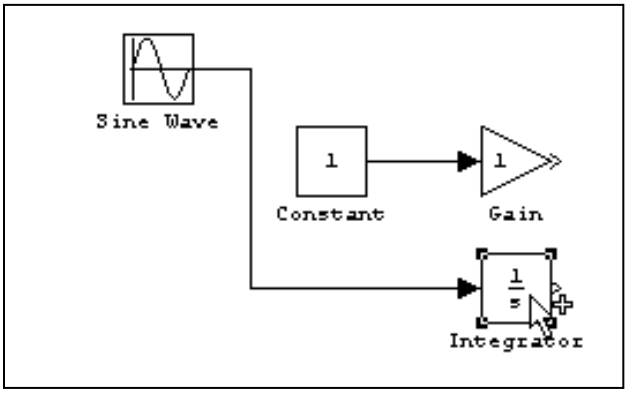
Simulink block diagrams use lines to represent pathways for signals among blocks in a model. Simulink can connect blocks for you or you can connect the blocks yourself by drawing lines from their output ports to their input ports.

3.1- Connecting Two Blocks:

To auto connect two blocks:

1. Select the source block.

2. Hold down **Ctrl** and left-click the destination block.

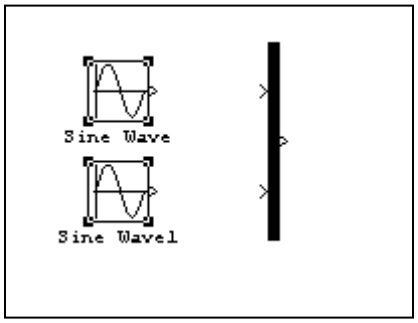


3.2- Connecting Groups of Blocks:

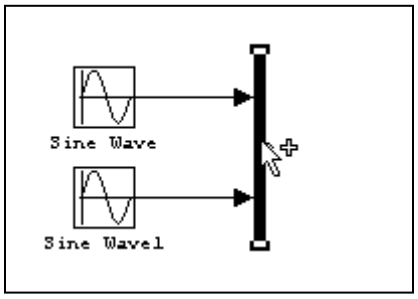
Simulink can connect a group of source blocks to a destination block or a source block to a group of destination blocks.

To connect a group of source blocks to a destination block:

1. Select the source blocks.



2. Hold down **Ctrl** and left-click the destination block.

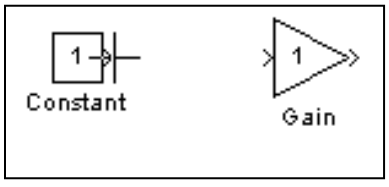


In the same way we can connect a source block to a group of destination blocks.

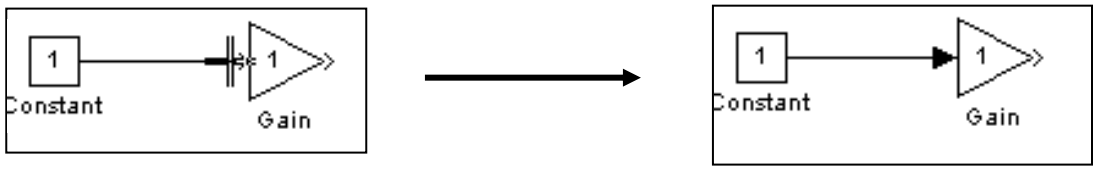
4- Drawing a Line Between Blocks

To connect the output port of one block to the input port of another block:

1. Position the cursor over the first block's output port. It is not necessary to position the cursor precisely on the port. The cursor shape changes to crosshairs.



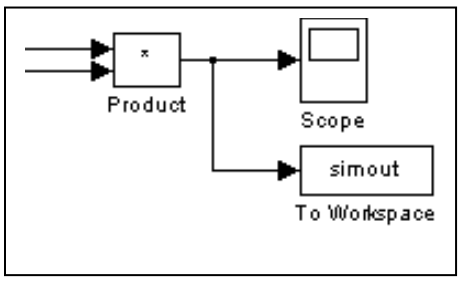
3. Press and hold down the mouse button.
4. Drag the pointer to the second block's input port. The cursor shape changes to double crosshairs. Release the mouse button. Simulink replaces the port symbols by a connecting line with an arrow showing the direction of the signal flow.



Simulink draws connecting lines using horizontal and vertical line segments. To draw a diagonal line, hold down the **Shift** key while drawing the line.

We can also draw a branch lines. A *branch line* is a line that starts from an existing line and carries its signal to the input port of a block. Both the existing line and the branch line carry the same signal. Using branch lines enables you to cause one signal to be carried to more than one block.

In this example, the output of the Product block goes to both the Scope block and the To Workspace block.



To add a branch line, follow these steps:

1. Position the pointer on the line where you want the branch line to start.
2. While holding down the **Ctrl** key, press and hold down the left mouse button.
3. Drag the pointer to the input port of the target block, then release the mouse button and the **Ctrl** key.

You can also use the right mouse button instead of holding down the left mouse button and the **Ctrl** key.

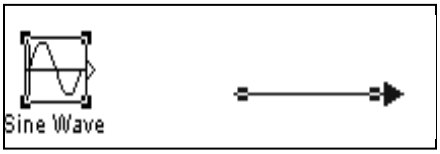
To draw a line segment, you draw a line that ends in an unoccupied area of the diagram. An arrow appears on the unconnected end of the line. To add another line segment, position the cursor over the end of the segment and draw another segment. Simulink draws the segments as horizontal and vertical lines. To draw diagonal line segments, hold down the **Shift** key while you draw the lines.

5- Selecting Objects

Many model building actions, such as copying a block or deleting a line, require that you first select one or more blocks and lines (objects).

5.1- Selecting One Object:

To select an object, click it. Small black square "handles" appear at the corners of a selected block and near the end points of a selected line. For example, the figure below shows a selected Sine Wave block and a selected line.

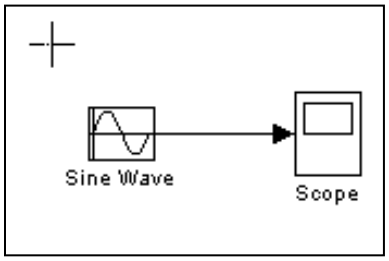


When you select an object by clicking it, any other selected objects are deselected.

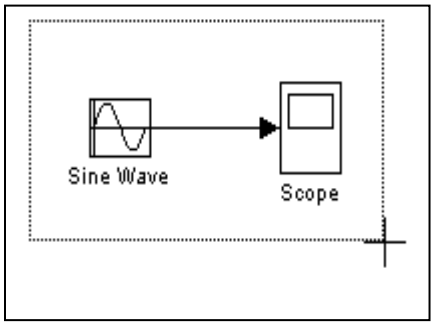
5.2- Selecting Multiple Objects Using a Bounding Box:

An easy way to select more than one object in the same area of the window is to draw a bounding box around the objects:

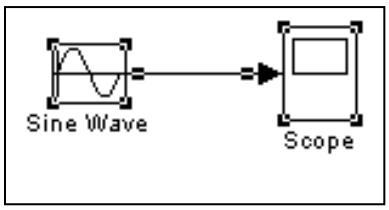
1. Define the starting corner of a bounding box by positioning the pointer at one corner of the box, then pressing and holding down the mouse button. Notice the shape of the cursor.



2. Drag the pointer to the opposite corner of the box. A dotted rectangle encloses the selected blocks and lines.



3. Release the mouse button. All blocks and lines at least partially enclosed by the bounding box are selected.



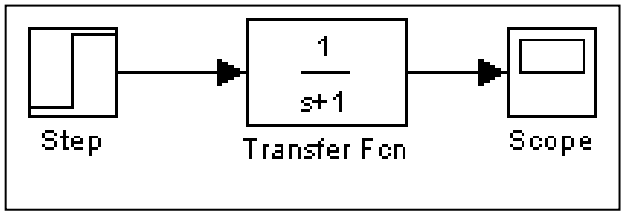
6- Viewing Output Trajectories

Output trajectories from Simulink can be plotted using one of three methods:

- Feed a signal into either a Scope or an XY Graph block.
- Write output to return variables and use MATLAB plotting commands.
- Write output to the workspace using To Workspace blocks and plot the results using MATLAB plotting commands.

6.1- Using the Scope Block:

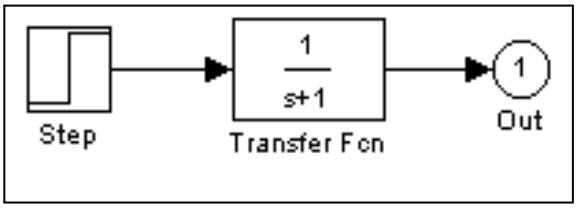
You can use display output trajectories on a Scope block during a simulation. This simple model shows an example of the use of the Scope block.



The display on the Scope shows the output trajectory. The Scope block enables you to zoom in on an area of interest or save the data to the workspace. The XY Graph block enables you to plot one signal against another.

6.2- Using Return Variables:

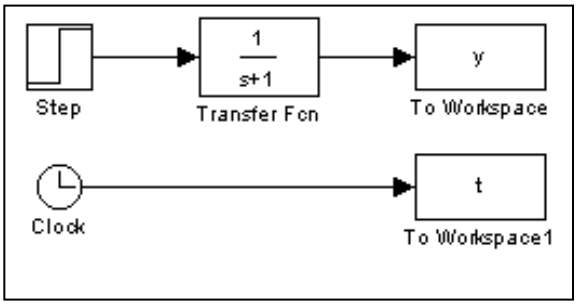
By returning time and output histories, you can use MATLAB plotting commands to display and annotate the output trajectories.



The block labeled Out is an Outport block from the Signals & Systems library.

6.3- Using the To Workspace Block:

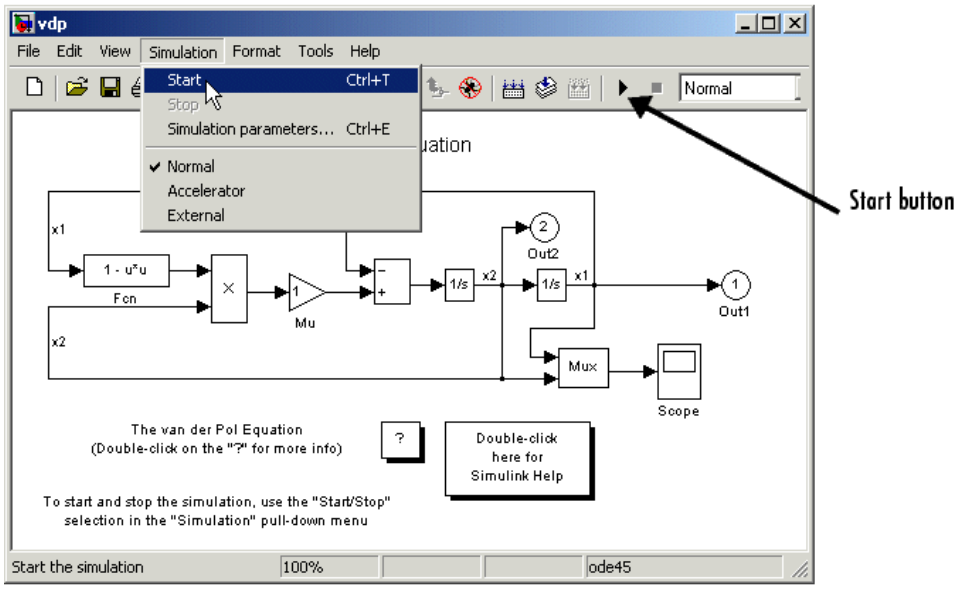
The To Workspace block can be used to return output trajectories to the MATLAB workspace. The model below illustrates this use.



The variables y and t appear in the workspace when the simulation is complete. You store the time vector by feeding a Clock block into a To Workspace block. You can also acquire the time vector by entering a variable name for the time on the **Workspace I/O** pane of the **Simulation Parameters** dialog box, for menu-driven simulations. The To Workspace block can accept an array input, with each input element's trajectory stored in the resulting workspace variable.

7- Starting and Stopping a Simulation

To start execution of a model, select **Start** from the model editor's **Simulation** menu or click the **Start** button on the model's toolbar. You can also use the keyboard shortcut, **Ctrl+T**, to start the simulation.



While the simulation is running, a progress bar at the bottom of the model window shows how far the simulation has progressed. A **Stop** command replaces the **Start** command on the **Simulation** menu. A **Pause** command appears on the menu and replaces the **Start** button on the model toolbar.

8- Modeling Systems

One of the most confusing issues for new Simulink users is how to model systems. Here are some examples that might improve your understanding of how to model systems.

Example-1: Converting Celsius to Fahrenheit

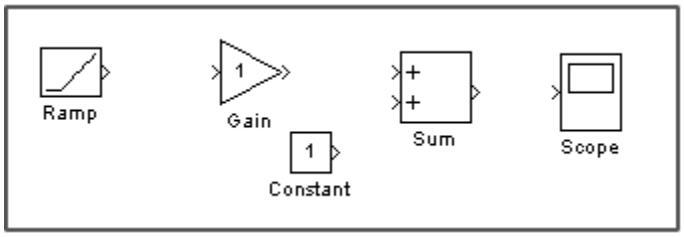
To model the equation that converts Celsius temperature to Fahrenheit

TF = 9/5(TC) + 32

First, consider the blocks needed to build the model.

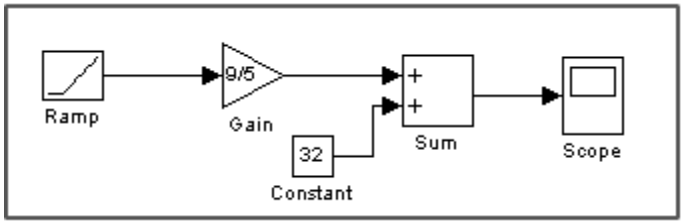
- A Ramp block to input the temperature signal, from the Sources library
- A Constant block to define a constant of 32, also from the Sources library

- A Gain block to multiply the input signal by 9/5, from the Math library
- A Sum block to add the two quantities, also from the Math library
- A Scope block to display the output, from the Sinks library



Next, gather the blocks into your model window, assign parameter values to the Gain and Constant blocks by opening (double-clicking) each block and entering the appropriate value. Then, click the **Close** button to apply the value and close the dialog box.

Now, connect the blocks.



The Ramp block inputs Celsius temperature. Open that block and changes the **Initial output** parameter to 0. The Gain block multiplies that temperature by the constant 9/5. The Sum block adds the value 32 to the result and outputs the Fahrenheit temperature.

Open the Scope block to view the output. Now, choose **Start** from the **Simulation** menu to run the simulation. The simulation runs for 10 seconds.

Example-2: Modeling a Simple Continuous System

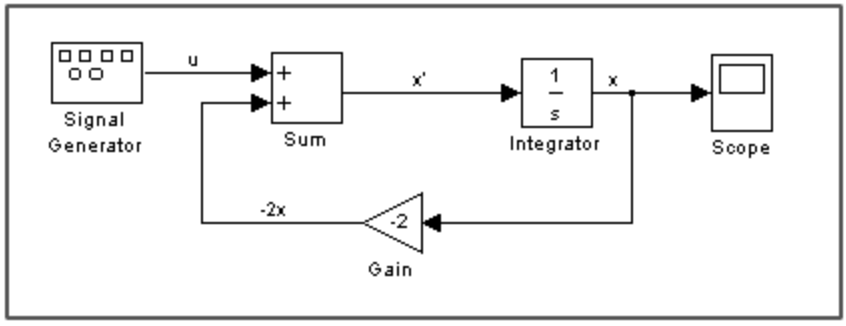
To model the differential equation

$$x'(t) = -2x(t) + u(t)$$

where *u(t)* is a square wave with an amplitude of 1 and a frequency of 1rad/sec. The Integrator block integrates its input *x´* to produce *x*. Other blocks needed in this model include a Gain block and a Sum block. To generate a square wave, use a Signal Generator

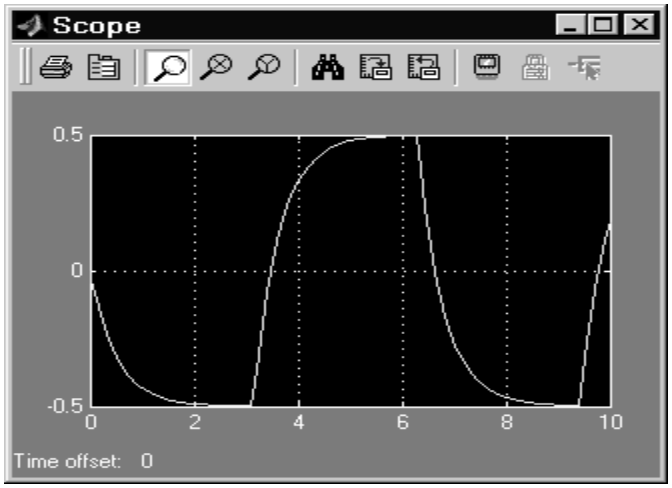
block and select the Square Wave form but change the default units to radians/sec. Again, view the output using a Scope block. Gather the blocks and define the gain.

In this model, to reverse the direction of the Gain block, select the block, then use the **Flip Block** command from the **Format** menu. To create the branch line from the output of the Integrator block to the Gain block, hold down the **Ctrl** key while drawing the line.



An important concept in this model is the loop that includes the Sum block, the Integrator block, and the Gain block. In this equation, *x* is the output of the Integrator block. It is also the input to the blocks that compute *x´*, on which it is based. This relationship is implemented using a loop.

The Scope displays *x* at each time step. For a simulation lasting 10 seconds, the output looks like this:



The equation you modeled in this example can also be expressed as a transfer function. The model uses the Transfer Fcn block, which accepts *u* as input and outputs *x*. So, the block implements *x/u*. If you substitute *sx* for *x´* in the above equation, you get

$$sx = -2x + u$$

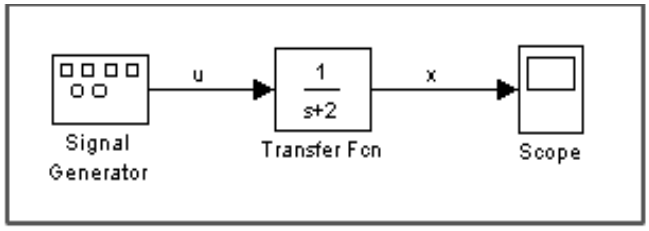
Solving for *x* gives

$$x = u/(s+2)$$

or,

$$x/u = 1/(s+2)$$

The Transfer Fcn block uses parameters to specify the numerator and denominator coefficients. In this case, the numerator is 1 and the denominator is s+2. Specify both terms as vectors of coefficients of successively decreasing powers of s. In this case the numerator is [1] (or just 1) and the denominator is [1 2]. The model now becomes quite simple.



The results of this simulation are identical to those of the previous model.

Example-3: Modeling a Simple Control System

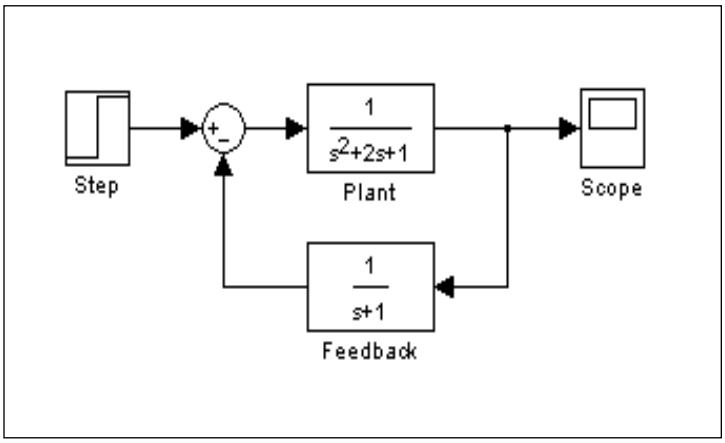
In this example we will model a simple control system and finding its time response according to these specifications:

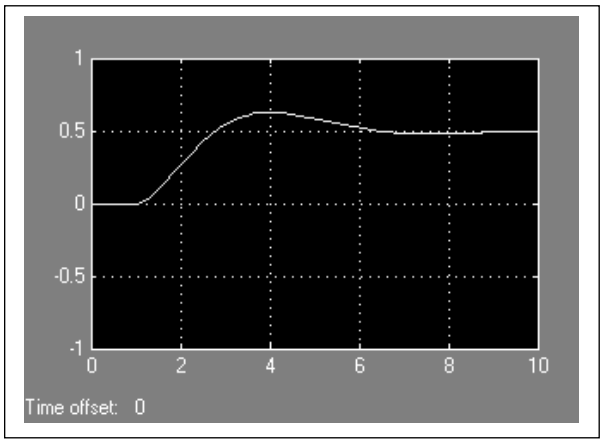
The Plant Transfer function **(G(s))=** $\dfrac{\mathbf{1}}{\mathbf{s^2+2s+1}}$.

The Feedback Transfer function **(H(s))=** $\dfrac{\mathbf{1}}{\mathbf{s+1}}$.

Note that to find the response for any control system, a step input is usually used.

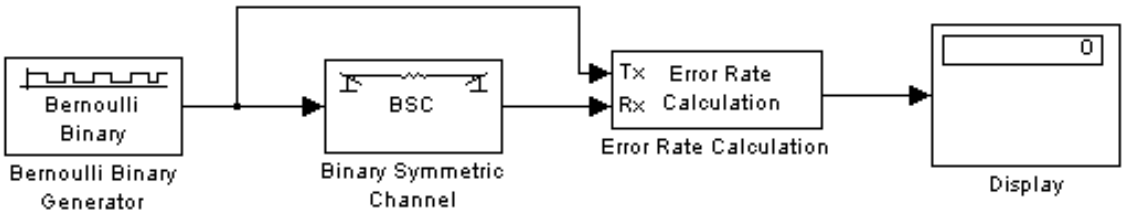
Assign parameter values to the blocks by opening (double-clicking) each block and entering the appropriate value. Then, click the **Close** button to apply the value and close the dialog box. Now, connect the blocks and view the response from the Scope block:





Example-4: Building a Channel Noise Model:

This section shows how to build a simple model of a communication system. The model, shown in the following figure, contains the most basic elements of a communication system: a source for the signal, a channel with noise, and means of detecting errors caused by noise.



We encourage you to build the model for yourself, as this is the best way to learn how to use the Communications Blockset.

Overview of the Model

The channel noise model generates a random binary signal, and then switches the symbols 0 and 1 in the signal, according to a specified error probability, to simulate a channel with noise. The model then calculates the error rate and displays the result. The model contains the following components.

1. Source: The source for the signal in this model is the Bernoulli Binary Generator block, which generates a random binary sequence. You can get this block from the Data Sources sublibrary of the Comm Sources library.

2. **Channel:** The Binary Symmetric Channel block simulates a channel with noise. The block introduces random errors to the signal by changing a 0 to a 1 or the reverse, with a probability specified by the **Error probability** parameter in the block's mask. You can get this block from the Channels library.

3. **Error Rate Calculation:** The Error Rate Calculation block calculates the error rate of the channel. The block has two input ports, labeled Tx, for the transmitted signal, and Rx, for the received signal. The block compares the two signals and checks for errors. The output of the block is a vector with three entries:

   - Bit error rate, which you expect to be approximately 0.01, since this is the probability of error in the channel
   - Number of errors
   - Total number of bits that are transmitted

You can get this block from the Comm Sinks library.

4. **Display:** The Display block displays the output of the Error Rate Calculation block. You can get this block from the Simulink Sinks library.

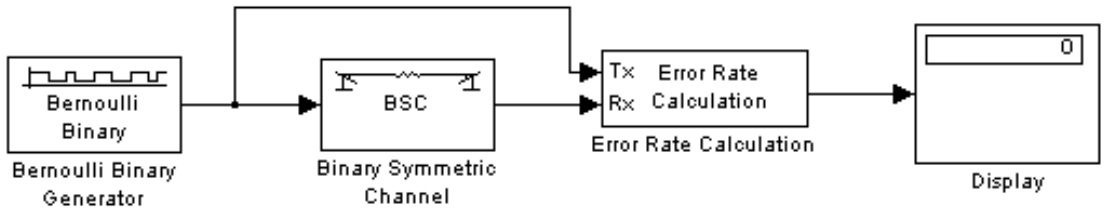
Setting Parameters in the Channel Noise Model

To set block parameters in the channel noise model, do the following:

1. Double-click the Binary Symmetric Channel block and set **Error probability** `0.01`.
   Clear the box next to **Output error vector**. This removes the block's lower output port, which is not needed for this model.
2. Double-click the Error Rate Calculation block and set **Output data** to **Port** to create an output port for the block. Select the box next to **Stop simulation** (This causes the simulation to stop after the target number of errors occurs or the maximum number of symbols is reached).

Connecting the Blocks

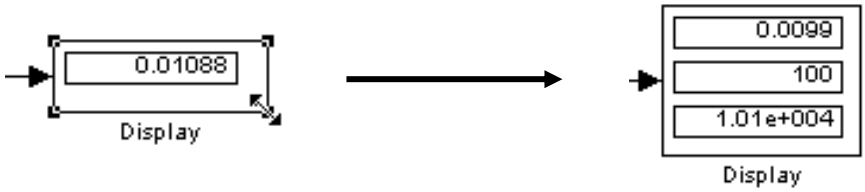
Next, connect the blocks as shown in the following figure. Make sure to connect the arrow from the Binary Symmetric Channel block to the input port labeled Rx on the Error Rate Calculation block.



The upper line leading from the Bernoulli Binary Generator block to the Error Rate Calculation block, shown in the following figure, is called a *branch line*. Branch lines carry the same signal to more than one block.

Running the Model

To run the model, select **Start** from the **Simulation** menu. After a few seconds, the model will stop automatically. To see all three boxes in the Display block, you must enlarge the block slightly by Selecting the Display block and move the mouse pointer to one of the lower corners of the block, so that a diagonal arrow appears on the corner, as shown.



The Display block displays the following information:

- The bit error rate
- The number of errors
- The total number of bits that are transmitted

Exercises:

1. If a stone is thrown vertically with an initial speed **u**, its vertical displacement **s** after a time **t** has elapsed is given by the formula:

$$\mathbf{s(t) = ut - gt^2/2}$$ (Air resistance has been ignored)

Model this equation with a simulink diagram to obtain a plot for the vertical displacement **s** with time **t**. Where g=9.8 , u=40.

Hints:

First, consider the blocks needed to build the model.

- A Ramp block to input the time signal t, from the Sources library.
- A Math function block (double click on it and select square) to get t^2, from the Math library.
- A Gain block to multiply the input signal by u, from the Math library.
- A Gain block to multiply the square of the input signal by g/2, from the Math library.
- A Sum block to subtract the two quantities, also from the Math library.
- A Scope block to display the output, from the Sinks library.

Next, gather the blocks into your model window. Note that the output will not display a cleared output so right click on the display and select autoscale.

2. If the exact number of bacteria at time t is given by the formula:

$$\mathbf{N(t) = 1000\ e^{rt}}$$

Where r (growth rate per hour)=0.01 and t (in hours) is a ramp input

Model this formula in a simulink diagram to obtain a plot for the bacteria growth (N) with time (t).

Hint:

The blocks needed to build the model.

- A Ramp block to input the time signal t, from the Sources library.
- A Gain blocks to multiply r with t, from the Math library.
- A Math function block to get the exp, from the Math library.
- A Gain blocks to multiply the result×1000, from the Math library.
- A Scope block to display the output, from the Sinks library.

3. Model the differential equation

$$x''(t) = -5x'(t) + 2x(t) + u(t)$$

Where *u(t)* is a square wave with an amplitude of 1 and a frequency of 1rad/sec. View the output using a Scope block.. **Hint:** See example-2, page (6-13).

4. Repeat problem-3 using the Transfer Fcn block. **Hint:** See example-2, page (6-14).

5. Model the differential equation

$$x''(t) = x'(t)x(t) - 2x(t) + u(t)$$

where *u(t)* is a sine wave with an amplitude of 1 and a frequency of 10 rad/sec. View the output using a Scope block.. **Hint:** See example-2, page (6-13).

6. Model the following control system and find its time response according to these specifications:

Plant Transfer function **(G(s))=** $\frac{\mathbf{2}}{\mathbf{s^2+3s+1}}$, Feedback Transfer function **(H(s))=** $\frac{\mathbf{2}}{\mathbf{s+2}}$, A forward gain **(K)** = **0.1**, **1**, and **10**.

Find the response for this system for each value of the gain (K). Note that to find the response for any control system, a step input is usually used. **Hint:** See example-3.

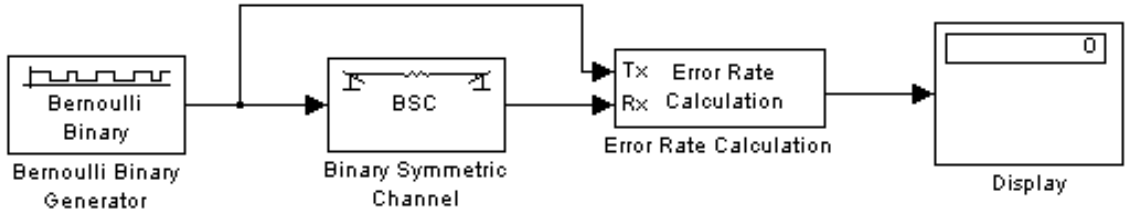
7. Model the following control system having a plant transfer function **(G(s))=** $\frac{\boldsymbol{\omega_n^2}}{\mathbf{s^2+2\zeta\omega_n s+\omega_n^2}}$ and unity feedback transfer function (H(s)). Find the time response according to:

a) $\zeta = 0$ and $\omega_n = 2$.

b) $\zeta = 0.1$ and $\omega_n = 4$.

c) $\zeta = 1$ and $\omega_n = 8$.

8. Build the following communication system model. The model, shown in the following figure, contains the most basic elements of a communication system: a source for the signal, a channel with noise, and means of detecting errors caused by noise.



Set the Error probability for the Binary Symmetric Channel Block `0.01, 0.05, 0.1` and `1.0,` and the sampling rate for Bernoulli Binary Generator Block 0.001.

Connect the blocks and display the following information for each value of Error probabilities:

- The bit error rate.
- The number of errors.
- The total number of bits that are transmitted.

Hint: See example-4, page (6-16).

9. Repeat problem-7 but replace the Binary Symmetric Channel Block by the AWGN Channel Block with signal to noise ratio (SNR) equal to 100, 1000, and 10000 dB.

Hint: You can get the AWGN Channel Block from the Channels sublibrary in Communications blockset library and it looks like the figure below. For more information, see example-4, page (6-16).