SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – I – Software Tools for Engineering Applications – SECA4001

		L	T	P	Credits	Total Marks
SECA4001	**SOFTWARE TOOLS FOR ENGINEERING APPLICATIONS**	3	0	0	3	100

COURSE OBJECTIVES

To understand and gain complete knowledge about the fundamentals of MATLAB programming.

To develop and translate mathematical concepts to MATLAB code.

To provide data analytic skills by processing and visualization of data's.

To design and develop Simulink and MATLAB models for specific engineering applications.

UNIT 1 INTRODUCTION TO MATLAB 9 Hrs.

Introduction To Matlab software-basic Features- Introduction to programming in MATLAB-M-File- Scripts-Input-Output commands - Creation and overwriting of Variables- Data types –Arithmetic, Relational & Logical operations - Example programs for operations-precedence of operators -Matrix generation and matrix arithmetic operation- Transposing a matrix - Concatenating matrices -Array Initialization and array arithmetic operations-Examples for Solving linear equations- Functions - User defined functions - passing arguments - using functions with vectors and matrices- cell arrays & structures - Strings - comparing - Concatenation.

UNIT 2 LOOPS AND CONTROL STATEMENTS 9 Hrs.

Control Flow & Decision statements- IF - IF ELSE - NESTED IF ELSE - SWITCH - TRY & CATCH - FOR -WHILE - NESTED FOR - FOR with IF statements, MATLAB program organization, Debugging methods- Setting and running with breakpoints – Examining values-Correcting and ending debugging, Example programs using the above commands. MISCELLANEOUS TOPICS: Date & Time Functions, Time Computations, File & Directory management.

UNIT 3 PLOTS IN MATLAB AND GUI 9 Hrs.

Basic 2D plots, Parametric & Implicit plots, subplot, LOG, LOG-LOG, SEMILOG-POLAR-COMET, exporting figures, HOLD, STEM, BAR, HIST, Interactive plotting, , axis labels, and annotations, Specifying line styles and colors, 3D plots – Mesh Surface- Contour –Plots with special graphics, View command, Plotting file data, Plotting from a function, GUI – Event based user interfaces, Matlab GUIDE, call back function, GUI controls, Example programs.

UNIT 4 ADVANCED MATHEMATICS APPLICATIONS 9 Hrs.

Fitting Curves to Data -Polynomials, Addition, multiplication and division of polynomials, Roots and derivative of a polynomial, curve fitting, polyfit, Interpolation, Extrapolation, Least squares, basic fitting interface, Complex Numbers, Adding , Subtracting and Multiplying Complex Numbers, Integration and Differentiation, Trapezoidal Rule, Calculus in Symbolic Math Toolbox.

UNIT 5 SIMULINK AND MATLAB APPLICATIONS 9 Hrs.

Simulink- Introduction, Block setting, Model annotation, solver, sinks library, sources, math operations library, user defined functions and look up table in Simulink, ports and subsystems, masked sub system, program controls in Simulink: FOR, WHILE, IF, CASE, Signal routing and logical's, Exporting Simulink data to Matlab, Applications –Modelling of a simple PID controller using SIMULINK, Plotting the Frequency response of FIR & IIR filters using MATLAB.

Max. 45 Hrs.

COURSE OUTCOMES

On completion of the course, student will be able to

CO1 - Recall and recollect the basic programming fundamentals

CO2 - Understand various array arithmetic procedures

CO3 - Analyze and develop different control structures using MATLAB

CO4 - Evaluate different interactive plotting methods

CO5 - Identify the need for GUI based operations for real time programming.

CO6 - Design and demonstrate applications based on communication systems, controllers etc.,

UNIT 1 INTRODUCTION

Introduction To Matlab software-basic Features- Introduction to programming in MATLAB-M-File- Scripts-Input-Output commands - Creation and overwriting of Variables- Data types –Arithmetic, Relational & Logical operations – Example programs for operations- precedence of operators -Matrix generation and matrix arithmetic operation- Transposing a matrix - Concatenating matrices -Array Initialization and array arithmetic operations-Examples for Solving linear equations- Functions - User defined functions - passing arguments - using functions with vectors and matrices- cell arrays & structures - Strings - comparing – Concatenation.

MATLAB® is a very powerful software package that has many built-in tools for solving problems and for graphical illustrations. The simplest method for using the MATLAB product is interactively; an expression is entered by the user and MATLAB immediately responds with a result. It is also possible to write programs in MATLAB, which are essentially groups ofcommands that are executed sequentially. This chapter will focus on the basics, including many operators and built-in functions that can be used in interactive expressions. Means of storing values, including vectors and matrices, will also be introduced.

MATrix LABoratory (MATLAB)

- Basically deals with interactive matrix calculations
- Special purpose computer program optimized to perform Engineering and Scientificcalculations
    - Has built in integrated development environment
- Supports different platform (windows 9x/NT/ 2000, Unix,etc.,)
- Has extensive library and built in functions for various field.
- MATLAB complier is an interpreter
- Includes tools that allow Graphical User Interface (GUI)
    - Visit : https://in.mathworks.com

GETTING INTO MAT LAB

MATLAB is a mathematical and graphical software package; it has numerical, graphical, and programming capabilities. It has built-in functions to do many operations, and there are toolboxes that can be added to augment these functions (e.g., for signal processing). There are versions available for different hardware platforms, and there are both professional and student editions. When the MATLAB software is started, a window is opened: the main part is the Command Window (see Figure 1.1). In the Command Window, there is a statement that says:

In the Command Window, you should see:

>>

The >> is called the prompt. In the Student Edition, the promptappears as: EDU>>

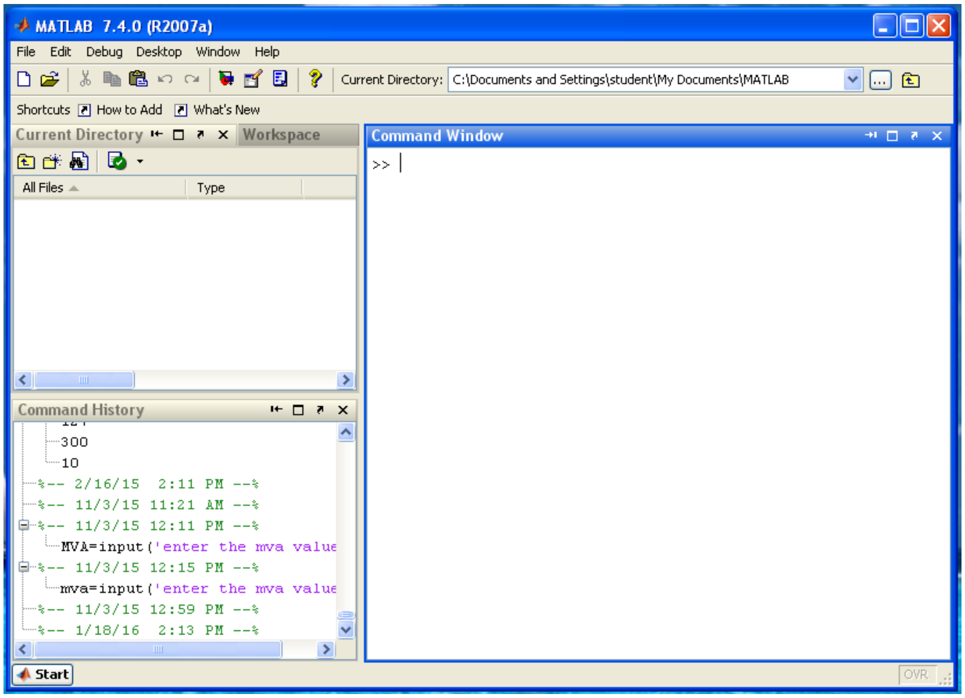


Figure 1.1 MATLAB Window

In the Command Window, MATLAB can be used interactively. At the prompt, any MATLAB command or expression can be entered, and MATLAB will immediately respond with the result. It is also possible to write ***programs*** in MATLAB, which are contained in ***script files*** or M-files. There are several commands that can serve as an introduction to MATLAB and allow you to get help:

info will display contact information for the product

demo has demos of several options in MATLAB

help will explain any command; **help help** will explain how help works

help browser opens a Help Window

lookfor searches through the help for a specific string (be aware that this can take a longtime)

To get out of MATLAB, either type **quit** at the prompt, or chooses File, then Exit MATLAB from the menu. In addition to the Command Window, there are several other windows that can be opened and may be opened by default. What is described here is the default layout for these windows, although there are other possible configurations. Directly above the Command Window, there is a pull-down menu for the Current Directory. The folder that is set as the Current Directory is where files will be saved. By default, this is the

Work Directory, but thatcan be changed.

To the left of the Command Window, there are two tabs for Current Directory Window and Workspace Window. If the Current Directory tab is chosen, the files stored in that directory are displayed. The Command History Window shows commands that have been entered, not just in the current session (in the current Command Window), but previously as well. This default configuration can be altered by clicking Desktop, or using the icons at the top-right corner of each window: either an ―x,‖ which will close that particular window; or a curled arrow, which in its initial state pointing to the upper right lets you undock that window. Once undocked, clicking the curled arrow pointing to the lower right will dock the window again.

VARIABLES AND ASSIGNMENT STATEMENTS

In order to store a value in a MATLAB session, or in a program, a ***variable*** is used. The Workspace Window shows variables that have been created. One easy way to create a variable isto use an ***assignment statement***. The format of an assignment statement is

variablename = expression

The variable is always on the left, followed by the ***assignment operator***, = (unlike in mathematics, the single equal sign does *not* mean equality), followed by an expression. The expression is evaluated and then that value is stored in the variable. For example, this is the wayit would appear in the Command Window:

```
>> mynum = 6

mynum
= 6
>>
```

Here, the ***user*** (the person working in MATLAB) typed mynum = 6 at the prompt, and MATLAB stored the integer 6 in the variable called *mynum*, and then displayed the result followed by the prompt again. Since the equal sign is the assignment operator, and does not mean equality, the statement should be read as ―mynum gets the value of 6‖ (*not* ―mynum equals 6‖). Note that the variable name must always be on the left, and the expression on the right. An error will occur if these are reversed.

```
>> 6 = mynum

??? 6 = mynum
      |
```

Error: The expression to the left of the equals sign is not a valid target for an assignment. Putting a semicolon at the end of a statement suppresses the output.For example,

>> res = 9 – 2;

\>>

This would assign the result of the expression on the right side, the value 7, to the variable *res*; it just doesn‘t show that result. Instead, another prompt appears immediately. However, at this point in the Workspace Window the variables *mynum* and *res* can be seen.

Note: In the remainder of the text, the prompt that appears after the result will not be shown. The spaces in a statement or expression do not affect the result, but make it easier to read. The following statement that has no spaces would accomplish exactly the same thing as the previous statement:

>> res = 9–2;

MATLAB uses a default variable named *ans* if an expression is typed at the prompt and it is not assigned to a variable. For example, the result of the expression 6 3 is stored in the variable *ans*:

>> 6 + 3

ans
= 9

This default variable is reused any time just an expression is typed at the prompt. A short-cut for retyping commands is to press the up-arrow, which will go back to the previously typed command(s). For example, if you decided to assign the result of the expression 6 3 to the variable *res* instead of using the default *ans*, you could press the up-arrow and then the left-arrow to modify the command rather than retyping the whole statement:

>> res = 6 + 3

res
= 9

This is very useful, especially if a long expression is entered with an error, and you want to go back to correct it. To change a variable, another assignment statement can be used that assigns the value of a different expression to it. Consider, for example, the following sequence of statements:

>> mynum = 3

mynum
= 3
\>>
mynum
= 4 + 2
mynum
=

6
>> mynum = mynum + 1

mynum
= 7

In the first assignment statement, the value 3 is assigned to the variable *mynum*. In the next assignment statement, *mynum* is changed to have the value of the expression 4 2, or 6. In the third assignment statement, *mynum* is changed again, to the result of the expression mynum 1. Since at that time *mynum* had the value 6, the value of the expression was 6 1, or 7. At that point, if the expression mynum 3 is entered, the default variable *ans* is used since the result of this expression is not assigned to a variable.

Thus, the value of *ans* becomes 10 but *mynum* is unchanged (it is still 7). Note that just typingthe name of a variable will display its value.

>> mynum + 3

a
n
s

=

1
0
>>mynum

m
y
n
u
m

=

7

INITIALIZING, INCREMENTING, AND DECREMENTING

Frequently, values of variables change. Putting the first or initial value in a variable is called ***initializing*** the variable. Adding to a variable is called ***incrementing***. For example, the statement

mynum = mynum + 1

increments the variable *mynum* by 1.

VARIABLE NAMES

Variable names are an example of ***identifier names***. We will see other examples of identifiernames, such as filenames, in future chapters. The rules for identifier names are:

- The name must begin with a letter of the alphabet. After that, the name can contain letters, digits, and the underscore character (e.g., value_1), but it cannot have a space. There is a limit to the length of the name; the built-in function **namelengthmax** tells howmany characters this is.
- MATLAB is case-sensitive. That means that there is a difference between upper- andlowercase letters. So, variables called *mynum*, *MYNUM*, and *Mynum* are all different.
- There are certain words called ***reserved words*** that cannot be used as variablenames. Names of built-in functions can, but should not, be used as variable names.

Additionally, variable names should always be ***mnemonic***, which means they should make some sense. For example, if the variable is storing the radius of a circle, a name such as -radius‖ would make sense; ‖x‖probably wouldn‘t. The Workspace Window shows the variables that have been created in the current Command Window and their values.

The following commands relate to variables:

- **who** shows variables that have been defined in this Command Window (this just showsthe names of the variables)
- **whos** shows variables that have been defined in this Command Window (this showsmore information on the variables, similar to what is in the Workspace Window)
- **clear** clears out all variables so they no longer exist
  clear *variablename* clears out a particular variable

If nothing appears when **who** or **whos** is entered, that means there aren‘t any variables! For example, in the beginning of a MATLAB session, variables could be created and then selectivelycleared (remember that the semicolon suppresses output):

```
>> who

>> mynum = 3;

>> mynum + 5;
```

>> who

Your variables are:
Ans   mynum
>> clear mynum

>> who

Your variables are:

ans

EXPRESSIONS

Expressions can be created using values, variables that have already been created, operators, built-in functions, and parentheses. For numbers, these can include operators such as multiplication, and functions such as trigonometric functions. An example of such an expression would be:

*>> 2 * sin(1.4)*
ans=
1.9709

The Format **Function and Ellipsis**

The ***default*** in MATLAB is to display numbers that have decimal places with four decimal places, as already shown. The **format** command can be used to specify the output format of expressions. There are many options, including making the format **short** (the default) or **long**. For example, changing the format to **long** will result in 15 decimal places. This will remain in effect until the format is changed back to **short**, as demonstrated with an expression and with the built-in value for **pi**.

>> format long

*>> 2 * sin(1.4)*

ans =
1.97089
945997
69
20
>>
pi

ans =

3.1415926
53589793
>> format short

*>> 2 * sin(1.4)*
ans =
1.9709

The **format** command can also be used to control the spacing between the MATLAB commandor expression and the result; it can be either **loose** (the default) or **compact**

>> format loose

>> 2^7
Ans=128

>> format compact

>> 2^7

Ans=128

Especially long expressions can be continued on the next line by typing three (or more) periods,which is the continuation operator, or the **ellipsis**. For example,
>> 3 + 55 – 62 + 4 – 5 .â•›.â•›.

+ 22 – 1

ans
= 16

BUILT-IN FUNCTIONS AND HELP

There are many, many built-in functions in MATLAB. The **help** command can be used to find out what functions MATLAB has, and also how to use them. For example, typing **help** at the prompt in the Command Window will show a list of help topics, which are groups of related functions. This is a very long list; the most elementary help topics are in the beginning.

For example, one of these is listed as **matlab\elfun**; it includes the elementary math functions. Another of the first help topics is **matlab\ops**, which shows the operators that can be used in expressions. To see a list of the functions contained within a particular help topic, type **help**

followed by the name of the topic. For example,
>> *help elfun*

will show a list of the elementary math functions. It is a very long list, and is broken into trigonometric (for which the default is radians, but there are equivalent functions that instead use degrees), exponential, complex, and rounding and remainder functions. To find out what a particular function does and how to call it, type **help** and then the name of the function. For example,

>> *help sin*

will give a description of the **sin** function.
To ***call*** a function, the name of the function is given followed by the ***argument(s)*** that are passed to the function in parentheses. Most functions then ***return*** value(s). For example, to find the absolute value of –4, the following expression would be entered:

>> *abs(–4)*

which is a ***call*** to the function **abs**. The number in the parentheses, the –4, is the ***argument***. Thevalue 4 would then be ***returned*** as a result. In addition to the trigonometric functions, the elfun help topic also has some rounding and remainder functions that are very useful. Some of these include **fix**, **floor**, **ceil**, **round**, **rem**, and **sign**. The **rem** function returns the remainder from a division; for example 5 goes into 13 twice with a remainder of 3, so the result of this expressionis 3:

>> *rem(13,5)*

ans
= 3
Another function in the elfun help topic is the **sign** function, which returns 1 if the argument ispositive, 0 if it is 0, and –1 if it is negative. For example,
>> *sign(–5)*

ans =
–1
>> *sign(3)*

ans
= 1

CONSTANTS

Variables are used to store values that can change, or that are not known ahead of time. Most languages also have the capacity to store ***constants***, which are values that are known ahead of time, and cannot possibly change. An example of a constant value would be **pi**, or , which is 3.14159…. In MATLAB, there are functions that return some of these constant values. Some of these include:

pi 3.14159….

i square root of 1
j square root of 1
inf infinity
NaN stands for -not a number‖; e.g., the result of 0/0

TYPES

Every expression, or variable, has a ***type*** associated with it. MATLAB supports many types of values, which are called ***classes***. A class is essentially a combination of a type and the operations that can be performed on values of that type. For example, there are types to store different kinds of numbers. For float or real numbers, or in other words numbers with a decimal place (e.g., 5.3), there are two basic types: **single** and **double**. The name of the type **double** is short for double precision; it stores larger numbers than **single**. MATLAB uses a ***floating point*** representation for these numbers. For integers, there are many integer types (e.g., **int8**, **int16**, **int32**, and **int64**). The numbers in the names represent the number of bits used to store values of that type. For example, the type **int8** uses eight bits altogether to store the integer and its sign. Since one bit is used for the sign, this means that seven bits are used to store the actual number. Each bit stores the number in binary (0‘s or 1‘s), and 0 is also a possible value, which means that 2 ^ 7 – 1 or 127 is the largest number that can be stored. The range of values that can be stored in **int8** is actually from –128 to 127. This range can be found for any type by passing the name of the type as a string (which means in single quotes) to the functions **intmin** and **intmax**. For example,

>> *intmin('int8')*

ans =
–128
>> *intmax('int8')*

Ans=127

The larger the number in the type name, the larger the number that can be stored in it. We will for the most part use the type **int32** when an integer type is required. The type **char** is used to store either single ***characters*** (e.g., ‗x‘) or ***strings***, which are sequences of characters (e.g., ‗cat‘). Both characters and strings are enclosed in single quotes. The type **logical** is used to store true/false values. If any variables have been created in the Command Window, they can be seen in the Workspace Window. In that window, for every variable, the variable name, value, and class (which is essentially its type) can be seen. Other attributes of variables can also be seen in the Workspace Window. Which attributes are visible by default depends on the version of MATLAB. However, when the Workspace Window is chosen, clicking View allows the user to choose which attributes will be displayed. By default, numbers are stored as the type **double** in MATLAB. There are, however, many functions that convert values from one type to another. The names of these functions are the same as the names of the types just shown. They can be used as functions to convert a value to that type. This is called ***casting*** the value to a different

type, or type casting. For example, to convert a value from the type **double**, which is the default,to the type **int32**, the function **int32** would be used. Typing the following assignment statement:

>> val = 6+3

would result in the number 9 being stored in the variable *val*, with the default type of **double**,which can be seen in the Workspace Window. Subsequently, the assignment statement

>> val = int32(val);

would change the type of the variable to **int32**, but would not change its value. If we insteadstored the result in another variable, we could see the difference in the types by using **whos**.

>> val = 6 + 3;

>> vali = int32(val);

>> whos

Name Size Bytes Class Attributes

val	1x1	8	double
vali	1x1	4	int32

One reason for using an integer type for a variable is to save space.

RANDOM NUMBERS

When a program is being written to work with data, and the data is not yet available, it is often useful to test the program first by initializing the data variables to ***random numbers***. There are several built-in functions in MATLAB that ***generate*** random numbers, some of which will be illustrated in this section. Random number generators or functions are not truly random. Basically, the way it works is that the process. starts with one number, called a ***seed***. Frequently, the initial seed is either a predetermined value or it is obtained from the built-in clock in the computer. Then, based on this seed, a process determines the next random number. Using that number as the seed the next time, another random number is generated, and so forth. These are actually called ***pseudo-random***; they are not truly random because there is a process that determines the next value each time. The function **rand** can be used to generate random real numbers; calling it generates one random real number in the range from 0 to 1. Thereare no arguments passed to the **rand** function. Here are two examples of calling the **rand** function:

>rand
Ans = 0.961

The seed for the **rand** function will always be the same each time MATLAB is started, unless the

state is changed, for example, by the following:

*rand('state',sum(100*clock))*

This uses the current date and time that are returned from the built-in **clock** function to set the seed. Note: this is done only once in any given MATLAB session to set the seed; the **rand** function can then be used as shown earlier any number of times to generate random numbers. Since **rand** returns a real number in the range from 0 to 1, multiplying the result by an integer Nwould return a random real number in the range from 0 to N. For example, multiplying by 10 returns a real in the range from 0 to 10, so this expression *rand*10*
would return a result in the range from 0 to 10. To generate a random real number in the range from *low* to *high*, first create the variables *low* and *high*. Then, use the expression rand*(high– low) low. For example, the sequence

>> low = 3;

>> high = 5;

>> rand(high–low)+low*

would generate a random real number in the range from 3 to 5.
However, in MATLAB, there is another built-in function that specifically generates randomintegers,
randint. Calling the function with **randint(1,1,N)** generates one random integer in the range from0 to N –

1. The first two arguments essentially specify that one random integer will be returned; the thirdargument gives the range of that random integer. For example,

>> randint(1,1,4)

generates a random integer in the range from 0 to 3. Note: Even though this creates random integers, the type is actually the default type **double**. A range can also be passed to the **randint** function. For example, the following specifies a random integer in the range from 1 to 20:

>> randint(1,1,[1,20])

VECTORS AND MATRICES
Vectors and ***matrices*** are used to store sets of values, all of which are the same type. A vector can be either a ***row vector*** or a ***column vector***. A matrix can be visualized as a table of values. The dimensions of a matrix are r × c, where r is the number of rows and c is the number of columns. This is pronounced -r by c.‖ If a vector has *n* elements, a row vector would have the dimensions 1 × *n*, and a column vector would have the dimensions *n* × 1. A ***scalar*** (one value) has the dimensions 1 × 1. Therefore, vectors and scalars are actually just subsets of matrices. All the values stored in these matrices are stored in what are called ***elements***. MATLAB is written to work with matrices; the name MATLAB is short for -matrix laboratory.‖ Forthis reason, it is very easy to create vector and matrix variables, and there are many operations and functions that can be used on vectors and matrices. A vector in MATLAB is equivalent to what is called a one-dimensional ***array*** in other languages. A matrix is equivalent to a two-dimensional array. Usually, even in MATLAB, some operations that can be performed on either vectors or matrices are referred to as ***array operations***. The term ***array*** also frequently is used to mean generically either a vector or a matrix.

CREATING ROW VECTORS
There are several ways to create row vector variables. The most direct way is to put the values that you want in the vector in square brackets, separated by either spaces or commas. For example, both of these assignment statements create the same vector v:
>> v = [1 2 3 4]

v =

1 2 3 4

>> v = [1,2,3,4]

v =

1 2 3 4

Both of these create a row vector variable that has four elements; each value is stored in aseparate element in the vector.

The Colon Operator and** Linspace **Function

If, as in the earlier examples, the values in the vector are regularly spaced, the ***colon operator***
can be used to ***iterate*** through these values. For
example, 1:5results in all the integers from 1 to 5:

>> vec = 1:5

vec =

1 2 3 4 5

Note that in this case, the brackets [ ] are not necessary to define the vector.
With the colon operator, a ***step value*** can also be specified with another colon,
in the form(first:step:last). For example, to create a vector with all integers from
1 to 9 in steps of 2:

>> nv = 1:2:9

nv =

1 3 5 7 9

Similarly, the **linspace** function creates a linearly spaced vector; **linspace(x,y,n)** creates a vector with n values in the inclusive range from x to y. For example, the following creates a vector withfive values linearly spaced between 3 and 15, including the 3 and 15:

>> ls = linspace(3,15,5)

ls =

3 6 9 12 15

Vector variables can also be created using existing variables. For example, a new vector is createdhere consisting first of all the values from *nv* followed by all values from *ls*:

>> newvec = [nv ls]

newvec =

1 3 5 7 9 3 6 9 12 15

Putting two vectors together like this to create a new one is called ***concatenating*** the vectors.

REFERRING TO AND MODIFYING ELEMENTS

A particular element in a vector is accessed using the name of the vector variable and the element number (or ***index***, or ***subscript***) in parentheses. In MATLAB, the indices start at 1. Normally, diagrams of vectors and matrices show the indices; for example, for the variable *newvec* created earlier the indices 1–10 of the elements are shown above the vector:

1	2	3	4	5	6	7	8	9	10
1	3	5	7	9	3	6	9	12	15

For example, the fifth element in the vector *newvec* is a 9.

>> newvec(5)

ans
= 9

A subset of a vector, which would be a vector itself, can also be obtained using the colon operator. For example, the following statement would get the fourth through sixth elements of the vector *newvec*, and store the result in a vector variable *b*:

>> b = newvec(4:6)

b =
7 9 3

Any vector can be used for the indices in another vector, not just one created using the colon operator. For example, the following would get the first, fifth, and tenth elements of the vector*newvec*:

>> newvec([1 5 10])

ans =
1 9 15
The vector [1 5 10] is called an ***index vector***; it specifies the indices in the original vector that arebeing referenced. The value stored in a vector element can be changed by specifying the index orsubscript. For example, to change the second element from the vector *b* to now store the value 11instead of 9:

>> b(2) = 11

b =

7 11 3

By using an index, a vector can also be extended. For example, the following creates a vectorthat has three elements. By then referring to the fourth element in an assignment statement, the vector is extended to have four elements.

>> rv = [3 55 11]

rv =

3 55 11

>> rv(4) = 2

rv =

3 55 11 2

If there is a gap between the end of the vector and the specified element, 0's are filled in. Forexample, the following extends the variable created earlier again:

>> rv(6) = 13

rv =

3 55 11 2 0 13

CREATING COLUMN VECTORS

One way to create a column vector is by explicitly putting the values in square brackets, separatedby semicolons:

>> c = [1; 2; 3; 4]

c

=

1

2

3

4

There is no direct way to use the colon operator described earlier to get a column vector. However, any row vector created using any of these methods can be ***transposed*** to get a column vector. In general, the transpose of a matrix is a new matrix in which the rows and columns are interchanged. For vectors, transposing a row vector results in a column vector, and transposing a column vector results in a row vector. MATLAB has a built-in operator, the apostrophe, to get a transpose.

>> r = 1:3;

>> c = r

c
=

1
2
3

CREATING MATRIX VARIABLES

Creating a matrix variable is really just a generalization of creating row and column vector variables. That is, the values within a row are separated by either spaces or commas, and the different rows are separated by semicolons. For example, the matrix variable *mat* is created by explicitly typing values:

>> mat = [4 3 1; 2 5 6]

mat =
4 3 1
2 5 6

There must always be the same number of values in each row. If you attempt to create a matrixin which there are different numbers of values in the rows, the result will be an error message;

for example:

>> mat = [3 5 7; 1 2]

??? Error using ==> vertcat

CAT arguments dimensions are not consistent. Iterators can also be used for the values on the rows using the colon operator; for example:

>> mat = [2:4; 3:5]

mat =
2 3 4
3 4 5

Different rows in the matrix can also be specified by pressing the Enter key after each row instead of typing a semicolon when entering the matrix values; for example:

>> newmat = [2 6 88

33 5 2]

newmat =

2	6	88
33	5	2

Matrices of random numbers can be created using the **rand** and **randint** functions. The first two arguments to the **randint** function specify the size of the matrix of random integers. For example, the following will create a 2 × 4 matrix of random integers, each in the range from 10to 30:

>> randint(2,4,[10,30])

ans =

29	22	28	19
14	20	26	10

For the **rand** function, if a single value *n* is passed to it, an *n* × *n* matrix will be created, or passing two arguments will specify the number of rows and columns:

>> rand(2)

ans =

0.2311	0.4860
0.6068	0.8913

>>
rand(1,3) ans = 0.7621

0.4565 0.018
5

MATLAB also has several functions that create special matrices. For example, the **zeros** function creates a matrix of all zeros. Like **rand**, either one argument can be passed (which will be both the number of rows and columns), or two arguments (first the number of rows and then the number of columns).

>> *zeros(3)*

ans =
0 0 0
0 0 0
0 0 0

>> *zeros(2,4)*

ans =
0 0 0 0
0 0 0 0

REFERRING TO AND MODIFYING MATRIX ELEMENTS

To refer to matrix elements, the row and then the column indices are given in parentheses (always the row index first and then the column). For example, this creates a matrix variable *mat*, and then refers to the value in the second row, third column of *mat*:

>> *mat = [2:4; 3:5]*

mat =
2 3 4
3 4 5

>> *mat(2,3)*

ans
= 5

It is also possible to refer to a subset of a matrix. For example, this refers to the first and second rows, second and third columns:

>> mat(1:2,2:3)

ans =

3 4

4 5

Using a colon for the row index means all rows, regardless of how many, and using a colon for the column index means all columns. For example, this refers to the entire first row:

>> mat(1,:)

ans =

2 3 4

and this refers to the entire second column:

>> mat(:, 2)

ans

= 3

4

If a single index is used with a matrix, MATLAB ***unwinds*** the matrix column by column. For example, for the matrix *intmat* created here, the first two elements are from the first column, and the last two are from the second column:

>> intmat = randint(2,2,[0 100])

intmat =

100 77

28 14

>> intmat(1)

ans

=

100

>> intmat(2)

ans

= 28

>> intmat(3)

ans

= 77
>> *intmat(4)*

ans
= 14

This is called ***linear indexing***. It is usually much better style when working with matrices to refer to the row and column indices, however. An individual element in a matrix can be modified by assigning a value.

>> *mat = [2:4; 3:5];*

>> *mat(1,2) = 11*

mat =

2	11	4
3	4	5

An entire row or column could also be changed. For example, the following replaces the entire second row with values from a vector:
>> *mat(2,:) = 5:7*

mat =

2	11	4
5	6	7

Notice that since the entire row is being modified, a vector with the correct length must be assigned. To extend a matrix, an individual element could not be added since that would mean there would no longer be the same number of values in every row. However, an entire row or column could be added. For example, the following would add a fourth column to the matrix:

>> *mat(:,4) = [9 2]'*

mat =

2	11	4	9
5	6	7	2

Just as we saw with vectors, if there is a gap between the current matrix and the row or column being added, MATLAB will fill in with zeros.

>> *mat(4,:) = 2:2:8*

mat =

2	11	4	9
5	6	7	2
0	0	0	0
2	4	6	8

DIMENSIONS

The **length** and **size** functions in MATLAB are used to find array dimensions. The **length** function returns the number of elements in a vector. The **size** function returns the number of rows and columns in a matrix. For a matrix, the **length** function will return either the number of rows or the number of columns, whichever is largest. For example, the following vector, *vec*, has four elements so its length is 4. It is a row vector, so the size is 1 × 4.

>> *vec = -2:1*

vec =

-2 -1 0 1

>> *length(vec)*

ans

= 4

>> *size(vec)*

ans =

1 4

For the matrix *mat* shown next, it has three rows and two columns, so the size is 3 × 2. The length is the larger dimension, 3.

>> *mat = [1:3; 5:7]'*

mat =

1 5

2 6

3 7

>> *size(mat)*

ans =

3 2

>> *length(mat)*

ans

= 3

>> [r c] = size(mat)

r
=
3
c =
2

Note: The last example demonstrates a very important and unique concept in MATLAB: the ability to have a vector of variables on the left-hand side of an assignment. The **size** function returns two values, so in order to capture these values in separate variables we put a vector of two variables on the left of the assignment. The variable *r* stores the first value returned, which is the number of rows, and *c* stores the number of columns.

MATLAB also has a function, **numel**, which returns the total number of elements in any array (vector or matrix):

>> vec = 9:–2:1

vec =
9 7 5 3 1

>> numel(vec)

ans
= 5

>> mat = randint(2,3,[1,10])

mat =
7 9 8
4 6 5

>> numel(mat)

ans
= 6

For vectors, this is equivalent to the length of the vector. For matrices, it is the product of the number of rows and columns. MATLAB also has a built-in expression **end** that can be used to refer to the last element in a vector; for example, v(end) is equivalent to v(length(v)). For matrices, it can refer to the last row or column. So, using **end** for the row index would refer to the last row. In this case, the element referred to is in the first column of the last row:

>> *mat = [1:3; 4:6]'*

mat =

1	4
2	5
3	6

>> *mat(end,1)*

ans
= 3
Using **end** for the column index would refer to the last column (e.g., the last column of the second row):

>> *mat(2,end)*

ans
= 5
This can be used only as an index.

CHANGING DIMENSIONS
In addition to the transpose operator, MATLAB has several built-in functions that change the dimensions or configuration of matrices, including **reshape**, **fliplr**, **flipud**, and **rot90**. The **reshape** function changes the dimensions of a matrix. The following matrix variable *mat* is 3 4, or in other words it has 12 elements.

>> *mat = randint(3,4,[1 100])*

mat =

14	61	2	94
21	28	75	47
20	20	45	42

These 12 values instead could be arranged as a 2 x 6 matrix, 6 x 2, 4 x3, 1x 12, or 12 x1. The **reshape** function iterates through the matrix columnwise. For example, when reshaping *mat* into a 2 6 matrix, the values from the first column in the original matrix (14, 21, and 20) are used first, then the values from the second column (61, 28, 20), and so forth.

>> *reshape(mat,2,6)*

ans =

14	20	28	2	45	47
21	61	20	75	94	42

The **fliplr** function ―flips‖ the matrix from left to right (in other words the left-most column, the first column, becomes the last column and so forth), and the **flipud** functions flips up to down. Note that in these examples *mat* is unchanged; instead, the results are stored in the default variable *ans* each time.

>> mat = randint(3,4,[1 100])

mat =

14	61	2	94
21	28	75	47
20	20	45	42

\>>
fliplr(mat)
ans =

94	2	61	14
47	75	28	21
42	45	20	20

\>>
mat
mat =

14	61	2	94
21	28	75	47
20	20	45	42

>> flipud(mat)

ans =

20	20	45	42
21	28	75	47
14	61	2	94

The **rot90** function rotates the matrix counterclockwise 90 degrees, so for example the value in the top-right corner becomes instead the top-left corner and the last column becomes the first row:

>> mat

mat =

14	61	2	94
21	28	75	47
20	20	45	42

>>
rot90(mat)
ans =

94	47	42
2	75	45
61	28	20
14	21	20

The function **repmat** can also be used to create a matrix; **repmat(mat,m,n)** creates a larger matrix, which consists of an $m \times n$ matrix of copies of mat. For example, here is a 2×2 random matrix:

>> intmat = randint(2,2,[0 100])

intmat =

100	7
7	
28	14

The function **repmat** can be used to replicate this matrix six times as a 3×2 matrix of the variable *intmat*.

>> repmat(intmat,3,2)

ans =

100	77	100	77
28	14	28	14
100	77	100	77
28	14	28	14
28	14	28	14

EMPTY VECTORS

An ***empty vector***, or, in other words, a vector that stores no values, can be created using empty square brackets:

>> evec = [ ]

evec

= [ ]

>> length(evec)

ans

= 0

Then, values can be added to the vector by ***concatenating***, or adding values to the existing vector. The following statement takes what is currently in *evec*, which is nothing, and adds a 4 to it.

>> evec = [evec 4]

evec

= 4

The following statement takes what is currently in *evec*, which is 4,
and adds an 11 to it.

>> evec = [evec 11]

evec =

4 11

This can be continued as many times as desired, in order to build a vector up from nothing. Empty vectors can also be used to ***delete elements from arrays***. For example, to remove the third element from an array, the empty vector is assigned to it:

>> vec = 1:5

vec =

1 2 3 4 5

>> vec(3) = [ ]

vec =

1 2 4 5

The elements in this vector are now numbered 1 through 4. Subsets of a vector could also be removed; for example:

>> vec = 1:8

vec =

1 2 3 4 5 6 78

>> vec(2:4) = [ ]

vec =

1 5 6 7 8

Individual elements cannot be removed from matrices, since matrices always have to have the same number of elements in every row.

>> mat = [7 9 8; 4 6 5]

mat =

7 9 8

4 6 5

>> mat(1,2) = [ ];

??? Indexed empty matrix assignment is not allowed. However, entire rows or columns could be removed from a matrix. For example, to remove the second column:

>> mat(:,2) = [ ]

mat =

7 8

4 5

CELL ARRAYS

It is a special MATLAB array whose elements are cells, container that can hold other MATLAB arraysCell array contains data structures instead of data

C{1,1} =

1 2 3

4 5 6

7 8 3

C{2,1} =

3.0000 + 4.0000i -5.0000 + 0.0000i

2.0000 + 0.0000i 0.0000 -10.0000i

C{1,2} =

this is a textC{2,2} =

[]

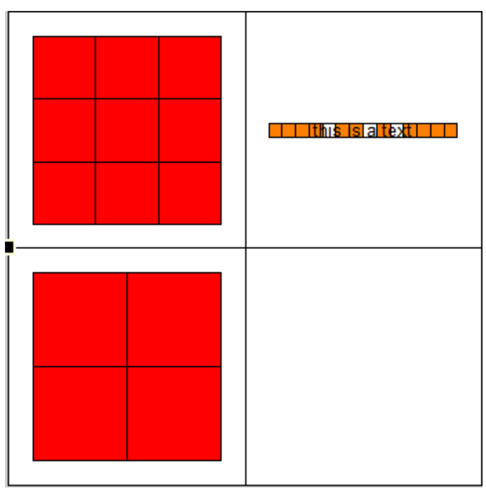


Figure 1.2 Cell Array

TO CREATE CELL ARRAY

Using assignment statement Assignment with content indexing

```
C{1,1} = [ 1 2 3; 4 5 6;
7 8 9];
C{1,2} = ‘ This is a
text’; C{2,1} = [
3+4*I -5; 2 -10*i];
C{2,2} = [ ];
```

Assignment with cell indexing

```
C(1,1) = {[ 1 2
3; 4 5 6; 7 8 9]};
C(1,2) = {‘ This is a text’};

C(2,1) = {[ 3+4*I -5; 2 -10*i]};

C(2,2) = {[ ]};
```

Pre-allocating cell array with cell function

B=cell(2,3)

This will create an empty 2X2 cell array. Once created using assignment statement the cells canbe filled with values.

```
>>
B=c
ell(2
,3)B
=
   []    []    []

   []    []    []
```

Use { } as cell constructor

Individual cell values can be created separated by commaB = {[1 2],17,[2;3];3-4*I, ‘hello’, eye(3)}

```
>> B = {[1 2],17,[2;3];3-4*i,
'hello', eye(3)}B =
        [1x2 double]    [   17]
                        [2x1 double]
```

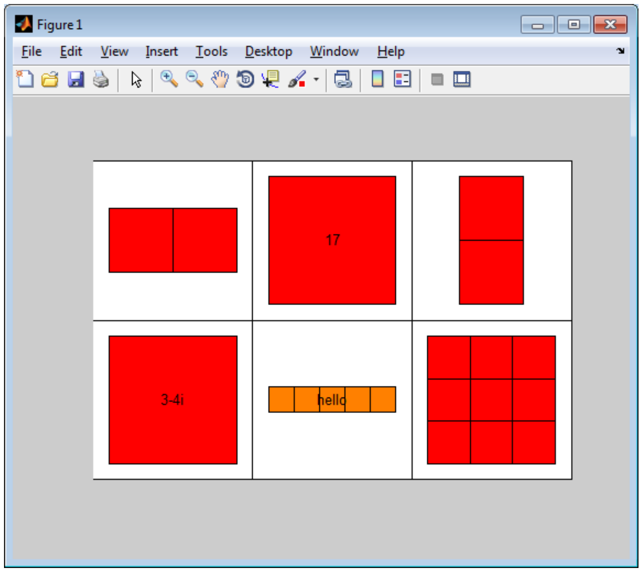


Figure 1.3 Cell Plot

```
[3.0000 - 4.0000i]        'hello'    [3x3 double]
```

>> **cellplot(B)**

TO EXTENT CELL ARRAY

The existing cell array can be extended by assignment statement

```
>> B{4,4}=5B =
        [1x2 double]    [   17]    [2x1
   double]              [] [3.0000 -
   4.0000i]               'hello'    [3x3
   double]                []
              []                   []     []                                            []
```

[] [] [] [5]

TO DELETE THE CONTENT OF CELL ARRAY

```
>>
B(4,:
) = [
]B =
[1x2 double]        [   17]    [2x1 double]    []

[3.0000 - 4.0000i]        'hello'    [3x3 double]    []

[]                  []    []                      []
```

The fourth row is deleted

There are several methods of displaying cell arrays. The **celldisp** function displays all elements ofthe cell array:

>> celldisp(cellrowvec)

```
cellrowvec{1
} =23
cellrowvec{2
} = a
cellrowvec{
3} =
1          3          5          7
cellrowvec{4
} = hello
```

The function **cellplot** puts a graphical display of the cell array in a Figure Window; however, it isa high- level view and basically just displays the same information as typing the name of the variable (e.g., it wouldn't show the contents of the vector in the previous example). Many of the functions and operations on arrays that we have already seen also work with cell arrays. For example, here are some related to dimensioning:

>> length(cellrowvec)

ans

= 4
>> *size(cellcolvec)*

ans =
4 1
>> *cellrowvec{end}*

ans
=
hello

It is not possible to delete an individual element from a cell array. For example, assigning anempty vector to a cell array element does not delete the element, it just replaces it with the emptyvector:

>> *cellrowvec*

mycell =
[23] [1x5 double] ‗hello'

>> *length(cellrowvec)*

ans
= 4
>> *cellrowvec{2} = [ ]*

mycell =
[23]â [] [1x5 double] ‗hello‘

>> *length(cellrowvec)*

ans
= 4
However, it is possible to delete an entire row or column from a cell array by assigning the empty vector (**Note**: use parentheses rather than curly braces to refer to the row or column):

>> *cellmat*

mycellmat =

[23] _a‗

[1x5 double] _hello‗

>> *cellmat(1,:) = []*

mycellmat =

[1x5 double] _hello‗

Storing Strings in Cell Arrays

One good application of a cell array is to store strings of different lengths. Since cell arrays can store different types of values in the elements, that means strings of different lengths can be stored in the elements.

>> *names = {'Sue', 'Cathy', 'Xavier'}*

names =

_Sue‗ _Cathy‗ _Xavier‗

This is extremely useful, because unlike vectors of strings created using **char** or **strvcat**, these strings do not have extra trailing blanks. The length of each string can be displayed using a **for** loop to loop through the elements of the cell array:

>> *for i =*
1:length(names)
disp(length(names
*{i}))*end

It is possible to convert from a cell array of strings to a character array, and vice versa. MATLAB has several functions that facilitate this. For example, the function **cellstr** converts from a character array padded with blanks to a cell array in which the trailing blanks have been removed.

>> *greetmat = char('Hello','Goodbye');*

>> *cellgreets = cellstr(greetmat)*

cellgreets =

_Hello‗

_Goodbye‗

The **char** function can convert from a cell array to a character matrix:

>> *names = {'Sue', 'Cathy', 'Xavier'};*

>> *cnames = char(names)*

>> *size(cnames)*

ans =

3 6

The function **iscellstr** will return logical true if a cell array is a cell array of all strings, or logicalfalse if not.

>> *iscellstr(names)*

ans

= 1

>> *iscellstr(cellcolvec)*

ans

= 0

We will see several examples of cell arrays containing strings of varying lengths in the coming chapters, including advanced file input functions and customizing plots.

STRUCTURE ARRAY

A cell array is a data type in which there is a single name for the whole data structure.

A Structure is a data type in which each individual element has a name. The individual elements of astructure are known as fields.

CREATE STRUCTURE ARRAY

A Field at a time using assignment statement

Assignment Statement

```
>> student.name='ram';

>> student.regno='3513110';

>> student.add='1st street';

>> student.city='Chennai';

>> student.zip='600119';
```

These assignment statement will create a structure named student with fields – name, regno,add,city,zip

To add another database

```
>> student(2).name='shiva';
```

Using struct function

```
>> student_database=struct('name', 'sathya','regno',
[3513120])student_database =
name: 'sathya' regno:
3513120
```

Will create a structure named student_database with fields name and regno

```
>> student2_database(1000)=struct('name',[ ],'regno',[
],'add',[ ])student2_database =
1x1000 struct array
   with fields:name
regno
add
```

Will create a structure named student2_database with fields name, regno and add

TO ADD FIELDS TO STRUCTURE

```
>> student(2).mark = [ 88 80 76 90 78 81 99]

student =
```

1x2 struct array
with fields:
name
regno
add
city
zip
mark

The field mark is added to the structure named student

TO REMOVE FIELDS TO STRUCTURE

>> student =
rmfield(student,'zip')student =
1x2 struct array
with fields:
name
regno
add
city
mark

The field ‘zip’ is removed from the structure
named studentTO EXTRACT DATA FROM
STRUCTURE ARRAY
To get the information in the structure

>>
student
(2).add
ans =
[]

Returns empty array, since there is no data added to the field for student(2)

>>
student(2
).mark
ans =

88   80   76   90   78   81   99

Returns the marks corresponding to student(2)

```
>>
student(2).
mark(2)ans
=
   80
```

Returns the mark corresponding to student(2) with index 2

```
>>
mean(student(2)
.mark)ans =
  84.5714
```

Returns the mean of the marks corresponding to student(2) Similarly any operation can be performed with the extracted data

TO EXTRACT/SET DATA

getfield is a function that gets the current value stored in the field

```
>> city=
getfield(student(1),'city'
)city =
Chennai
```

Returns the current value of the field ‘city’ corresponding to student(1)

setfield is a function that inserts new value in to the field

```
>>
setfield(student(2),'regno',3513
105')ans =
    nam
   e:
   'shiva
   '
   regno
```

:
3513
105
add:
[]
city: []

mark: [88 80 76 90 78 81 99]

Sets the value to the field ‘regno’ corresponding to student(2)

PASSING STRUCTURES TO FUNCTIONS

An entire structure can be passed to a function, or individual fields can be passed. For example, here are two different versions of a function that calculates the profit on a software package. The profit is defined as the price minus the cost. In the first version, the entire structure variable is passed to the function, so the function must use the dot operator to refer to the *price* and *cost* fields of the input argument.

calcprof.m
function profit = calcprof(packstruct)
% Calculates the profit for a software package
% The entire structure is passed to the
functionprofit = packstruct.price –
packstruct.cost;

>> calcprof(package)

In the second version, just the *price* and *cost* fields are passed to the function using the dot operator inthe function call. These are passed to two scalar input arguments in the function header, so there is no reference to a structure variable in the function itself, and the dot operator is not needed in the function.

calcprof2.m
function profit = calcprof2(oneprice, onecost)
% Calculates the profit for a software package
% The individual fields are passed to the

functionprofit = oneprice – onecost;
>> calcprof2(package.price, package.cost)

ans = 19.9600

It is important, as always with functions, to make sure that the arguments in the function call correspondone-to-one with the input arguments in the function header. In the case of *calcprof*, a structure variable is passed to an input argument, which is a structure. For the second function *calcprof2*, two individual fields, which are double values, are passed to two double arguments.

RELATED STRUCTURE FUNCTIONS

There are several functions that can be used with structures in MATLAB. The function **isstruct** will return 1 for logical true if the variable argument is a structure variable, or 0 if not. The **isfield** function returns logical true if a fieldname (as a string) is a field in the structure argument, or logical false if not
>> isstruct(package)

ans = 1
>> isfield(package,'cost')

ans = 1

The **fieldnames** function will return the names of the fields that are contained in a structure variable.
>> pack_fields = fieldnames(package)

pack_fields =
_item_no'
_cost'
_price'
_code'
Since the names of the fields are of varying lengths, the **fieldnames** function returns a cell array with the names of the fields. Curly braces are used to refer to the elements, since pack_fields is a cell array.For example, we can refer to the length of one of the strings:

>> length(pack_fields{2})

ans = 4

Table 1.1 File I/O operations

fclose	Close one or all open files
feof	Test for end of file
ferror	File I/O error information
fgetl	Read line from file, removing newline characters
fgets	Read line from file, keeping newline characters
fileread	Read contents of file as text
fopen	Open file, or obtain information about open files
fprintf	Write data to text file
fread	Read data from binary file
frewind	Move file position indicator to beginning of open file
fscanf	Read data from text file
fseek	Move to specified position in file
ftell	Current position
fwrite	Write data to binary file

SAVING VARIABLES TO FILES & LOADING VARIABLES FROM FILES

save filename x y -ASCII

filename is the name of the file that you want to write data to.

• x, y are variables to be written to the file.

‣ If omitted, all variables are written.

• -ASCII tells Matlab to write the data in a format that you can read.

‣ If omitted, data will be written in binary format.

‣ best for large amounts of data

load filename x y

• This is the complimentary command to save.

• Reads variables x and y from file filename

‣ If variables are omitted, all variables are loaded...**FORMATTED OUTPUT IN MATLAB disp(x)** - prints the contents of variable x.
fprintf(...) - use for formatted printing

• Allows much more control over output

• Syntax: fprintf('text & formatting',variables);

• Text formatting: ‣ %a.bc

‣ a - minimum width of output buffer

‣ b - number of digits past decimal point

‣ c - formatting scheme

- ‣ f - floating point (typical format) 12.345
- ‣ e - scientific notation - 1.2345e1

‣ s - string format

```
x = [1.1 2.2 3.3 4.4];
y = 2*x;
fprintf('Hello. (%1.3f,%1.3f), (%1.1f,%1.0f)\n',...
    x(1),y(1),x(3),y(3));
```

```
Hello. (1.100,2.200), (3.3,7)
```

FILE OUTPUT IN MATLAB

• **Open the file**

‣ fid = fopen(filename,’w’);

‣ ‘w’ tells matlab that we want to WRITE to the file.

‣ see “help fopen” for more information.

• **Write to the file**

‣ fprintf(fid,format,variables);

• **Close the file**

‣ fclose(fid);

FILE INPUT IN MATLAB

Import wizard “File→Import Data”

• Allows you to import data from delimited files (spreadsheets, etc)

Importing “spreadsheet” data

• dlmread - import data from a delimited file (you choose the delimiter)

• xlsread - import data from Excel.

General file input - three steps:

• fid=fopen(filename,’r’) - open a file to allow detailed input control.

‣ ‘r’ tells matlab that we want to READ from the file.

• a=fscanf(fid,format,size);

‣ Works like file writing, but use fscanf rather than fprintf.

‣ fid - file id that you want to read from

‣ format - how you want to save the information (string, number)

‣ ‘%s’ to read a string, ‘%f’ to read a floating point number, ‘%e’ to read scientific notation.

‣ size - how many entries to read.

‣ feof(fid) - returns true if end of file, false otherwise.

• fclose(fid);

CREATING STRING VARIABLES

A string consists of any number of characters (including, possibly, none). These are examples ofstrings: ‗‘

‗x‘

‗cat‘

_Hello there‘
_123‘

A ***substring*** is a subset or part of a string. For example, _there‘ is a substring within the string _Hello there‘. ***Characters*** include letters of the alphabet, digits, punctuation marks, white space, and control characters. ***Control characters*** are characters that cannot be printed, but accomplisha task (such as a backspace or tab). ***Whitespace characters*** include the space, tab, newline (which moves the cursor down to the next line), and carriage return (which moves the cursor to the beginning of the current line). ***Leading blanks*** are blank spaces at the beginning of a string, for example, _ hello‘, and ***trailing blanks*** are blank spaces at the end of a string. There are several ways that string variables can be created. One is using assignment statements:

>> word = 'cat';

Another method is to read into a string variable. Recall that to read into a string variable using the

input

function, the second argument _s‘ must be included:

>> strvar = input('Enter a string: ', 's')

Enter a string:
xyzabc
strvar =
xyzabc

If leading or trailing blanks are typed by the user, these will be stored in the string. For example,in the following the user entered four blanks and then _xyz‘:

>> s = input('Enter a string: ','s')

Enter a string:
xy
z s
=
xy
z

Strings as Vectors

Strings are treated as ***vectors of characters***—or in other words, a vector in which every element is a single character—so many vector operations can be performed. For example, the number of characters in a string can be found using the **length** function:

>>

length(‗cat‘)

Notice that there is a difference between an ***empty string***, which has a length of zero, and a string consisting of a blank space, which has a length of one. Expressions can refer to an individual element (a character within the string), or a subset of a string or a transpose of a string:

>> mystr = 'Hi';

>> mystr(1)

ans

= H

>> mystr'

ans

= H

i

>> sent = 'Hello there';

>> length(sent)

ans

= 11

>> sent(4:8)

Notice that the blank space in the string is a valid character within the string. A matrix can be created, which consists of strings in each row. So, essentially it is created as a column vector of strings, but the end result is that this would be treated as a matrix in which every element is a character:

>> wordmat = ['Hello';'Howdy']

wordmat

>> *size(wordmat)*

ans =

2 5

This created a 2 5 matrix of characters. With a character matrix, we can refer to an individualelement, which is a character, or an individual row, which is one of the strings:

>> *wordmat(2,4)*

ans

= d

>> *wordmat(1,:)*

Ans

=

Hello

Since rows within a matrix must always be the same length, the shorter strings must be paddedwith blanks so that all strings have the same length, otherwise an error will occur.

>> *greetmat = [‘Hello’; ‘Goodbye’]*

??? Error using ==> vertcat

CAT arguments dimensions are not consistent.

>> *greetmat = [‘Hello ’; ‘Goodbye’]*

greetmat

=

Hello Goodbye

>> *size(greetmat)*

ans =

2 7

Operations on Strings

MATLAB has many built-in functions that work with strings. Some of the string manipulation functions that perform the most common operations will be described here.

Concatenation

String concatenation means to join strings together. Of course, since strings are just vectors of characters, the method of concatenating vectors works for strings, also. For example, to create one long string from two strings, it is possible to join them by putting them in square brackets:

>> first = 'Bird';

>> last = 'house';

>> [first last] ans =
Birdhouse

The function **strcat** does this also horizontally, meaning that it creates one longer string from the inputs.

>> first = 'Bird';

>> last = 'house';

>> strcat(first,last)

ans =
Birdhous
e

There is a difference between these two methods of concatenating, however, if there are leading or trailing blanks in the strings. The method of using the square brackets will concatenate the strings, including all leading and trailing blanks.

>> str1 = 'xxx ';

>> str2 = ' yyy';

>> [str1 str2]

ans =
xxx
yyy
>> *length(ans)*

ans
= 12

The **strcat** function, however, will remove trailing blanks (but not leading blanks) from strings before concatenating. Notice that in these examples, the trailing blanks from *str1* are removed, but the leading blanks from *str2* are not:

>> *strcat(str1,str2)*

ans =
xxx
yyy
>>
length(ans)
ans =
9
>> *strcat(str2,str1)*

ans =
yyyxx
x
>> *length(ans)*

ans
= 9

The function **strvcat** will concatenate vertically, meaning that it will create a column vector of strings.
>> *strvcat(first,last)*
ans =
Bird
hous

e
>> *size(ans)*
ans =
2 5

Note that **strvcat** will pad with extra blanks automatically, in this case to make both strings have a length of 5.

Creating Customized Strings

There are several built-in functions that create customized strings, including **char, blanks**, and **sprintf**. We have seen already that the **char** function can be used to convert from an ASCII code to a character, for example:

>> *char(97)*

ans
= a

The **char** function can also be used to create a matrix of characters. When using the **char** function to create a matrix, it will automatically pad the strings within the rows with blanks as necessary so that they are all the same length, just like **strvcat**.

>> *clear greetmat*

>> *greetmat = char('Hello','Goodbye')*

greetmat
= Hello
Goodbye
>> *size(greetmat)*

ans
= 2 7

The **blanks** function will create a string consisting of n blank characters which are kind of hard to see here! However, in MATLAB if the mouse is moved to highlight the result in *ans*, the blanks can be seen.

>> blanks(4)

ans =

>> length(ans)

ans

= 4

Usually this function is most useful when concatenating strings, and you want a number of blank spaces in between. For example, this will insert five blank spaces in between the words:

>> [first blanks(5) last]

ans =

Bird house

Displaying the transpose of the **blanks** function can also be used to move the cursor down. In the Command Window, it would look like this:

>> disp(blanks(4)')

This is useful in a script or function to create space in output, and is essentially equivalent to printing the newline character four times. The **sprintf** function works exactly like the **fprintf** function, but instead of printing it creates a string. Here are several examples in which the output is not suppressed so the value of the string variable is shown:

>> sent1 = sprintf('The value of pi is %.2f', pi)

sent1 =

The value of pi is 3.14

>> sent2 = sprint ('Some numbers: %5d, %2d', 33, 6)

sent2 =

Some numbers: 33, 6

>> length (sent2)

ans

= 23

In the following example, on the other hand, the output of the assignment is suppressed so the string is created including a random integer and stored in the string variable. Then, some exclamation points are concatenated to that string.

>> phrase = sprintf('A random integer is
%d', ›. randint(1,1,[5,10]));

>> strcat(phrase, '!!!')

ans =
A random integer is 7!!!

All the conversion specifiers that can be used in the **fprintf** function can also be used in the **sprintf** function.

Removing Whitespace Characters

MATLAB has functions that will remove trailing blanks from the end of a string and/or leading blanks from the beginning of a string. The **deblank** function will remove blank spaces from the end of a string. For example, if some strings are padded in a string matrix so that all are the same length, it is frequently preferred to then remove those extra blank spaces in order to actually use the string.

>> names = char('Sue', 'Cathy','Xavier')

names
= Sue
Cathy
Xavier
>> name1 = names(1,:)

name1
= Sue
>> length(name1)

ans
= 6
>> name1 = deblank(name1);

>> length(name1)

ans
= 3

Note: The **deblank** function removes only trailing blanks from a string, not leading blanks. The **strtrim** function will remove both leading and trailing blanks from a string, but not blanks in the middle of the string. In the following example, the three blanks in the beginning and four blanks in the end are removed, but not

the two blanks in the middle. Selecting the result in MATLAB with the mouse would show the blank spaces.

>> strvar = [blanks(3) 'xx' blanks(2) 'yy' blanks(4)]

strvar
= xx
yy
>> length(strvar)

ans
= 13
>> strtrim(strvar)

ans
= xx
yy
>> length(ans)

ans
= 6

Changing Case

MATLAB has two functions that convert strings to all uppercase letters, or all lowercase, called **upper** and
lower.
>> mystring = 'AbCDEfgh';

>> lower(mystring)

ans = abcdefgh

>>

Upper (ans) ans = ABCDEFGH

Comparing Strings

There are several functions that compare strings and return logical true if they are equivalent, or logical false if not. The function **strcmp** compares strings, character by character. It returns logical true if the strings are completely identical (which infers that they must be of the same length, also) or logical false if the strings are not the same length or any corresponding characters are not identical. Here are some examples of these comparisons:

>> *word1 = 'cat';*

>> *word2 = 'car';*

>> *word3 = 'cathedral';*

>> *word4 = 'CAR';*

>> *strcmp(word1,word2)*

ans

= 0

>> *strcmp(word1,word3)*

ans

= 0

>> *strcmp(word1,word1)*

ans

= 1

>> *strcmp(word2,word4)*

ans

= 0

The function **strncmp** compares only the first *n* characters in strings and ignores the rest. The first two arguments are the strings to compare, and the third argument is the number of characters to compare (the value of *n*).

>> strncmp(word1,word3,3)

ans
= 1

>> strncmp(word1,word3,4)

ans
= 0
There is also a function **strncmpi** that compares *n* characters, ignoring the case.

Finding, Replacing, and Separating Strings

There are several functions that find and replace strings, or parts of strings, within other strings and functions that separate strings into substrings. The function **findstr** receives two strings as input arguments. It finds all occurrences of the shorter string within the longer, and returns the subscripts of the beginning of the occurrences. The order of the strings does not matter with **findstr**; it will always find the shorter string within the longer, whichever that is. The shorter string can consist of one character, or any number of characters. If there is more than one occurrence of the shorter string within the longer one, **findstr** returns a vector with all indices. Note that what is returned is the index of the beginning of the shorter string.

>> findstr('abcde', 'd')

ans
= 4
>> findstr('d','abcde')

ans
= 4
>> findstr('abcde', 'bc')

ans
= 2
>> findstr('abcdeabcdedd', 'd')

ans =
4 9 11 12

The function **strfind** does essentially the same thing, except that the order of the arguments does make a difference. The general form is **strfind(string, substring)**; it finds all occurrences of the substring within the string, and returns the subscripts.

>> strfind('abcdeabcde','e')

ans =

5 10

For both **strfind** and **findstr**, if there are no occurrences, the empty vector is returned.

>> strfind('abcdeabcde','ef')

ans

= [ ]

The function **strrep** finds all occurrences of a substring within a string, and replaces them with a new substring. The order of the arguments matters. The format is: strrep(string, oldsubstring, newsubstring) The following example replaces all occurrences of the substring _e' with the substring _x':

>> strrep('abcdeabcde','e','x')

ans =

abcdxabcd

x

All strings can be any length, and the lengths of the old and new substrings do not have to be the same. In addition to the string functions that find and replace, there is a function that separates a string into two substrings. The **strtok** function breaks a string into pieces; it can be called several ways. The function receives one string as an input argument. It looks for the first ***delimiter***, which is a character or set of characters that act as a separator within the string. By default, the delimiter is any whitespace character. The function returns a ***token***, which is the beginning of the string, up to (but not including) the first delimiter. It also returns the rest of the string, which includes the delimiter. Assigning the returned values to a vector of two variables will capture both of these. The format is

[token rest] = strtok(string)

where *token* and *rest* are variable names. For example,

>> sentence1 = 'Hello there'

sentence1
= Hello
there
>> [word rest] = strtok(sentence1)
word
=
Hello
rest =
there

>> length(word)

ans
= 5
>> length(rest)

ans
= 6
Notice that the rest of the string includes the blank space delimiter. By default, the delimiter for the token is a whitespace character (meaning that the token is defined as everything up to the blank space), but alternate delimiters can be defined. The format

[token rest] = strtok(string, delimeters)

returns a token that is the beginning of the string, up to the first character contained within the delimiters string, and also the rest of the string. In the following example, the delimiter is the character _l‘.

>> [word rest] = strtok(sentence1,'l')

word
= He
rest =
llo there
Leading delimiter characters are ignored, whether it is the default whitespace or a specified delimiter. For example, the leading blanks are ignored here:

>> [firstpart lastpart] = strtok(' materials science')

firstpart
=
material
s
lastpart
= science

Evaluating a String

The function **eval** is used to evaluate a string as a function.For example, in the following, the string ‗plot(x)' is interpreted to be a call to the **plot** function, and it produces the plot shown in Figure 6.2.

>> x = [2 6 8 3];

>> eval('plot(x)')

This would be useful if the user entered the name of the type of plot to use. In this example, the string that the user enters (in this case ‗bar') is concatenated with the string ‗(x)' to create the string ‗bar(x)'; this is then evaluated as a call to the **bar** function as seen in Figure 6.3. The name of the plot type is also used in the title.

The is functions for strings

There are several **is** functions for strings, which return logical true or false. The function **isletter** returns logical true if the character is a letter of the alphabet. The function **isspace** returns logical true if the character is a whitespace character. If strings are passed to these functions, they will return logical true or false for every element, or, in other words, every character.

>> isletter('a')

ans
= 1
>> isletter('EK127')

ans =
1 1 0 0 0
>> isspace('a b')

ans

= 0 1

0

The **ischar** function will return logical true if an array is a character array, or logical false if not.

>> vec = 'EK127';

>> ischar(vec)

ans

= 1

>> vec = 3:5;

>> ischar(vec)

ans

= 0

Converting between string and number types

MATLAB has several functions that convert numbers to strings in which each character element is a separate digit, and vice versa. (Note: these are different from the functions **char, double**, etc., that convert characters to ASCII equivalents and vice versa.) To convert numbers to strings, MATLAB has the functions **int2str** for integers and **num2str** for real numbers (which also works with integers). The function **int2str** would convert, for example, the integer 4 to the string ‗4'.

>> rani = randint(1,1,50)

rani

= 38

>> s1 = int2str(rani)

s1

=

38

>> length(rani)

ans

= 1

>> length(s1)

ans = 2

The variable *rani* is a scalar that stores one number, whereas *s1* is a string that stores two characters, ‗3‘ and ‗8‘. Even though the result of the first two assignments is 38, notice that the indentation in the Command Window is different for the number and the string. The **num2str** function, which converts real numbers, can be called in several ways. If only the real number is passed to the **num2str** function, it will create a string that has four decimal places, which is the default in MATLAB for displaying real numbers. The precision can also be specified (which is the number of digits), and format strings can also be passed, as shown:

>> str2 = num2str(3.456789)

str2 =
3.456
8
>> length(str2)

ans
= 6
>> str3 = num2str(3.456789,3)

str3
=
3.46
>> str = num2str(3.456789,'%6.2f')

str
=
3.4
6
Note that in the last example, MATLAB removed the leading blanks from the string. The function
str2num
does the reverse; it takes a string in which a number is stored and converts it to the type **double**:

>> num = str2num('123.456')

num =
123.4560
If there is a string in which there are numbers separated by blanks, the **str2num** function will convert this to a vector of numbers (of the default type double). For example,
>> mystr = '66 2 111';
>> numvec = str2num(mystr)

numvec
= 66 2
111
>> *sum(numvec)*

ans
=
179

Input and Output

The previous script would be much more useful if it were more general; for example, if the value of the radius could be read from an external source rather than being assigned in the script. Also, it would be better to have the script print the output in a nice, informative way. Statements that accomplish these tasks are called ***input/output*** statements, or ***I/O*** for short. Although for simplicity examples of input and output statements will be shown here from the Command Window, these statements will make the most sense in scripts.

Input Function

Input statements read in values from the default or ***standard input device***. In most systems, the default input device is the keyboard, so the input statement reads in values that have been entered by the ***user***, or the person who is running the script. In order to let the user know what he or she is supposed to enter, the script must first ***prompt*** the user for the specified values. The simplest input function in MATLAB is called **input**. The **input** function is used in an assignment statement. To call it, a string is passed, which is the prompt that will appear on the screen, and whatever the user types will be stored in the variable named on the left of the assignment statement. To make it easier to read the prompt, put a colon and then a space after the prompt. For example,
>> *rad = input('Enter the radius: ')*

Enter the radius: 5
rad =
5
If character or string input is desired, ‗s' must be added after the prompt:
>> *letter = input('Enter a char: ','s')*

Enter a char: *g*
letter
= g
Notice that although this is a string variable, the quotes are not shown. However, they are shown

in the Workspace Window. If the user enters only spaces or tabs before pressing the Enter key, they are ignored and an ***empty string*** is stored in the variable:

>> mychar = input('Enter a character: ', 's')

Enter a
character:
mychar =
‗‘

Notice that in this case the quotes are shown, to demonstrate that there is nothing inside of the string. However, if blank spaces are entered before other characters, they are included in the string. In this example, the user pressed the space bar four times before entering -go‖:

>> mystr = input('Enter a string: ', 's')

Enter a string: go
mystr =
go
>> length(mystr)

ans
= 6

It is also possible for the user to type quotation marks around the string rather than including the second argument ‗s' in the call to the **input** function:

>> name = input('Enter your name: ');

Enter your name: *'Stormy'* However, it is better to signify that character input is desired in the **input** function itself. Normally, the results from **input** statements are suppressed with a semicolon at the end of the assignment statements, as shown here. Notice what happens if string input has not been specified, but the user enters a letter rather than a number:

>> num = input('Enter a number: ')

Enter a number: t
??? Error using ==> input
Undefined function or
variable ‗t'.

Enter a number: 3

num
= 3
MATLAB gave an ***error message*** and repeated the prompt. However, if *t* is the name of a variable, MATLAB will take its value as the input:

>> t = 11;
>> *num = input('Enter a number: ')*

Enter a number: t
num =
11
Separate **input** statements are necessary if more than one input is desired. For example

>> *x = input('Enter the x coordinate: ');*

>> *y = input('Enter the y coordinate: ');*
Output Statements: disp **and** fprintf

Output statements display strings and the results of expressions, and can allow for ***formatting***, or customizing how they are displayed. The simplest output function in MATLAB is **disp**, which is used to display the result of an expression or a string without assigning any value to the default variable *ans*. However, **disp** does not allow formatting. For example,

>> *disp('Hello')*
Hello
>> *disp(4^3)*

64

Formatted output can be printed to the screen using the **fprintf** function. For example,

>> *fprintf('The value is %d, for sure!\n',4^3)*

The value is 64, for sure!

To the **fprintf** function, first a string (called the ***format string***) is passed, which contains any text to be printed as well as formatting information for the expressions to be printed. In this example, the %d is an example of format information. The %d is sometimes called a ***placeholder***; it specifies where the value of the expression that is after the string is to be printed. The character in the placeholder is called the ***conversion character***, and itspecifies the type of value that is being printed. There are others, but what follows is a list of the simple placeholders:

%d	integers (it actually stands for decimal integer)
%f	floats
%c	single characters
%s	strings

Don‘t confuse the % in the placeholder with the symbol used to designate a comment. The character ‗\n‘ at the end of the string is a special character called the ***newline*** character; when it is printed the output moves down to the next line. A ***field width*** can also be included in the placeholder in **fprintf**, which specifies how many characters total are to be used in printing. For example, %5d would indicate a field width of 5 for printing an integer and %10â•›s would indicate a field width of 10 for a string. For floats, the number of decimal places can also be specified; for example, %6.2f means a field width of 6 (including the decimal point and the decimal places) with two decimal places. For floats, just the number of decimal places can also be specified; for example, %.3f if indicates three decimal places.

>> fprintf('The int is %3â•›d and the float is %6.2f\n',5,4.9)

The int is 5 and the float is 4.90 Note that if the field width is wider than necessary, ***leading*** blanks are printed, and if more decimal places are specified than necessary, ***trailing*** zeros

are printed. There are many other options for the format string. For

example, the value being printed can be left-justified within the field width using a minus sign. The following example shows the difference between printing the integer 3 using %5d and using %–5d. The x‘s are just used to show the spacing.

>> fprintf('The integer is xx%5dxx and xx%-5dxx\n',3,3)

The integer is xx 3xx and xx3 xx

Also, strings can be truncated by specifying decimal places:
>> *fprintf('The string is %s or*
%.4s\n', 'truncate',... 'truncate')

The string is truncate or trun. There are several special characters that can be printed in the format string in addition to the newline character. To print a slash, two slashes in a row are used, and also to print a single quote two single quotes in a row are used. Additionally, \t is the tab character.

>> *fprintf('Try this out: tab\t quote '' slash \\ \n')*

Try this out: tab quote _ slash \

Scripts with Input and Output

Putting all this together, we can implement the algorithm from the beginning of this chapter. The following script calculates and prints the area of a circle. It
first prompts the user for a radius, reads in the radius, and then calculates and prints the area of the circle based on this radius.

```
script2.m
% This script calculates the area of a circle
% It prompts the user for the radius
% Prompt the user for the radius and calculate
% the area based on that radius
radius = input(_Please enter
the radius: ‘); area = pi *
(radius^2);
% Print all variables in a sentence
format fprintf(_For a circle with a
radius of %.2f,‘,radius) fprintf(_the
area is %.2f\n‘,area)
```

Executing the script produces the following output:

>> *script2*

Please enter the radius: 3.9
For a circle with a radius of 3.90, the area is 47.78

Notice that the output from the first two assignment statements is suppressed by putting semicolons at the end. That is frequently done in scripts, so that the exact format of what is displayed by the program is controlled by the **fprintf** functions.

Introduction to File Input/Output (Load **and** Save**)**

In many cases, input to a script will come from a data file that has been created by another source. Also, it is useful to be able to store output in an external file that can be manipulated and/or printed later. In this section, we will demonstrate how to read from an external data file, and also how to write to an external data file. There are basically three different operations, or ***modes***, on files. Files can be:

- Read from
- Written to
- A*ppended* to

Writing to a file means writing to a file, from the beginning. Appending to a file is also writing, but starting at the end of the file rather than the beginning. In other words, appending to a file means adding to what was already there. There are many different file types, which use different filename extensions. For now, we will keep it simple and just work with .dat or .txt files when working with data or text files. There are several methods for reading from files and writing to files; for now we will use the **load** function to read and the **save** function to write to files.

Writing Data to a File

The **save** function can be used to write data from a matrix to a data file, or to append to a data file. The format is:

save filename matrixvariablename –ascii.

The -ascii qualifier is used when creating a text or data file. The following creates a matrix and then saves the values of the matrix variable to a data file called testfile.dat:

>> mymat = rand(2,3)

mymat =

0.4565	0.821	0.6154
0.0185	0.444	0.7919

>> save testfile.dat mymat –ascii

This creates a file called testfile.dat that stores the numbers

0.4565	0.821	0.6154
0.0185	0.444	0.7919

The **type** command can be used to display the contents of the file; notice that scientificnotation is used:

>> type testfile.dat

4.5646767e–001	8.2140716e–001	*6.1543235e–001*
1.8503643e–002	4.4470336e–001	7.9193704e–001

Note: If the file already exists, the **save** function will overwrite it; **save** always beginswriting from the beginning of a file.

Appending Data to a Data File

Once a text file exists, data can be appended to it. The format is the same as previously, with the addition of the qualifier -append. For example, the following creates a new random matrix and appends it to the file just created:

>> mymat = rand(3,3)

mymat =

0.9218 0.4057 0.4103

0.7382 0.9355 0.8936

0.1763 0.9169 0.0579

>> save testfile.dat mymat – ascii – append

This results in the file testfile.dat containing

0.4565	0.8214	0.6154
0.0185	0.4447	0.7919
0.9218	0.4057	0.4103

0.7382	0.9355	0.8936
0.1763	0.9169	0.0579

Note: Although technically any size matrix could be appended to this data file, in order tobe able to read it back into a matrix later there would have to be the same number of values on every row.

Reading from a File

Once a file has been created (as previously), it can be read into a matrix variable. If the file is a data file, the **load** function will read from the file filename.ext (e.g., the extension might be .dat) and create a matrix with the same name as the file. For example, if the data file testfile.dat had been created as shown in the previous section, this would read from it:

>> *clear*

>> *load testfile.dat*

>> *who*

Your variables are:

testfile

>> *testfile*

testfile =

0.4565	0.821	0.6154
0.0185	0.444	0.7919
0.9218	0.405	0.4103
0.7382	0.935	0.8936
0.1763	0.916	0.0579

Note: The **load** command works only if there are the same number of values in each line, so that the data can be stored in a matrix, and the **save** command only writes from a matrix to a file. If this is not the case, lower-level file I/O functions must be used

m-files

Basic function m-files have these properties:

1. The function is defined in a separate m-file.
2. Function name is the same as the name of the m-file.

3. The function can have many input parameters and many output parameters.

• input parameters take on the values given by the calling function.

• output parameters have values that are returned to the calling function.

4. Variables in the function have their own workspace (memory).

• Variables in the function are separate from variables in other functions and in the command window environment, even if those variables have the same names.

• Values are only shared via input and output parameters.

User Defined Functions

A user-defined function is a Matlab program that is created by the **user, saved as a function file,** and then can be used like a built-in function. A function in general has input arguments (or parameters) and output variables (or parameters) that can be scalars, vectors, or matrices of any size.

Syntax

```
function [ output_args ] = Untitled3( input_args )

%UNTITLED3 Summary of this function goes here
%   Detailed explanation goes here



end
```

outArgs are enclosed in [ ]

- outArgs is a comma-separated list of variable names
- [ ] is optional if there is only one parameter
- functions with no outArgs are legal

inArgs are enclosed in ( )

- inArgs is a comma-separated list of variable names
- functions with no inArgs are legal

The filename of the function should be same as the function name

OPERATORS

- **Arithmetic Operations**

  Addition, subtraction, multiplication, division, power, rounding

- **Relational Operations**

  Value comparis ons

- **Logical Operations**

  True or false (Boolean) conditions

- **Set Operations**

  Unions, intersection, set membership

- **Bit-Wise Operations**

  Set, shift, or compare specific bit fields

ARITHMETIC OPERATIONS

Table 1.2 Arithmetic Operators

Operator	Purpose	Description
+	Addition	A+B adds A and B.
+	Unary plus	+A returns A.
-	Subtraction	A-B subtracts B from A
-	Unary minus	-A negates the elements of A.
*	Matrix multiplication	C = A*B is the linear algebraic product of the matrices A and B. The number of columns of A must equal the number of rows of B.
\	Matrix left division	x = A\B is the solution to the equation Ax = B. Matrices A and B must have the same number of rows.
/	Matrix right division	x = B/A is the solution to the equation xA = B. Matrices A and B must have the same number of columns. In terms of the left division operator, B/A = (A'\B')'.
^	Matrix power	A^B is A to the power B, if B is a scalar. For other values of B, the calculation involves eigen values and eigenvectors.

'	Complex conjugate transpose	A' is the linear algebraic transpose of A. For complex matrices, this is the complex conjugate transpose.
.^	Element-wise power	A.^B is the matrix with elements A(i,j) to the B(i,j) power.
./	Right array division	A./B is the matrix with elements A(i,j)/B(i,j).
.\	Left array division	A.\B is the matrix with elements B(i,j)/A(i,j).
.'	Array transpose	A.' is the array transpose of A. For complex matrices, this does not involve conjugation.

RELATIONAL OPERATORS Conditions in **if** statements use expressions that are conceptually, or logically, either true or false. These expressions are called ***relational expressions***, or sometimes ***Boolean*** or ***logical*** expressions. These expressions can use both ***relational operators***, which relate two expressions of compatible types, and ***logical operators***, which operate on logical operands.

Table 1.3 Relational Operators

Symbol	Role
==	Equal to
~=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

All concepts should be familiar, although the operators used may be different from those used in other programming languages, or in mathematics classes. In particular, it is important to note thatthe operator for equality is two consecutive equal signs, not a single equal sign (recall that the single equal sign is the assignment operator). For numerical operands, the use of these operators is straightforward.
For example,
3 < 5 means -3 less than 5,‖

which is conceptually a true expression. However, in MATLAB, as in many programming languages, ***logical true*** is represented by the integer 1, and ***logical false*** is represented by the integer 0. So, the expression

3 < 5 actually has the value 1 in MATLAB.
Displaying the result of expressions like this in the Command Window demonstrates the values ofthe expressions.

>> *3 < 5*

ans = 1

>> *9 < 2*

ans = 0

However, in the Workspace Window, the value shown for the result of these expressions would be true or false. The type of the result is **logical**.

Mathematical operations could be performed on the resulting 1 or 0.

>> *5 < 7*

ans = 1

>> *ans + 3*

ans = 4

Comparing characters, for example _a‘ < _c‘, is also possible. Characters are compared using their ASCII equivalent values. So, _a‘ < _câ•›‘ is conceptually a true expression, because the character _a‘ comes before the character _c‘.

>> *‘a’ < ‘c’*

ans = 1

LOGICAL OPERATORS

Table 1.4 Logical Operators

Symbol	Role
&	Logical AND
|	Logical OR
&&	Logical AND (with short-circuiting)
||	Logical OR (with short-circuiting)
~	Logical NOT

All logical operators operate on logical or Boolean operands. The **not** operator is a unaryoperator; the others are binary. The **not** operator will take a Boolean expression, which is conceptually true or false, and give the opposite value. For example, (3 < 5) is conceptually false since (3 < 5) is true. The **or** operator has two Boolean expressions as operands. The result is true if either or both of the operands are true, and false only if both operands are false. The **and** operator also operates on two Boolean operands. The result of an **and** expression is true only if both operands are true; it is false if either or both are false. In addition to these logical operators, MATLAB also has a function **xor**, which is the exclusive or function. It returns logical true ifone (and only one) of the arguments is true. For example, in the following only the first argumentis true,

so the result is true:

>> xor(3 < 5, 'a' > 'c')

ans = 1

In this example, both arguments are true so the result is false:

>> xor(3 < 5, 'a' < 'c')

ans = 0

Given the logical values of true and false in variables x and y, the ***truth table*** shows how the logical operators work for all combinations. Note that the logical operators are commutative (e.g., x || y is the same as y || x).

Table 1.5 Special Characters

Symbol	Role
,	Use commas to separate row elements in an array, array subscripts, function input and output arguments, and commands entered on the same line.
:	Use the colon operator to create regularly spaced vectors, index into arrays, and define the bounds of a for loop.
;	Use semicolons to separate rows in an array creation command, or to suppress the output display of a line of code.
( )	Use parentheses to specify precedence of operations, enclose function input arguments, and index into an array
[]	Square brackets enable array construction and concatenation, creation of empty matrices, deletion of array elements, and capturing values returned by a function.
{}	Use curly braces to construct a cell array, or to access the contents of a particular cell in a cell array.
%	The percent sign is most commonly used to indicate nonexecutable text within the body of a program. This text is normally used to include comments in your code.
‘ ’	Use single quotes to create character vectors that have class char

OPERATOR PRECEDENCE RULES

1. Parentheses ()
2. Transpose (.'), power (.^), complex conjugate transpose ('), matrix power (^)
3. Power with unary minus (.^-), unary plus (.^+), or logical negation (.^~) as well as matrix powerwith unary minus (^-), unary plus (^+), or logical negation (^~).
4. Unary plus (+), unary minus (-), logical negation (~)
5. Multiplication (.*), right division (./), left division (.\), matrix multiplication (*), matrix rightdivision (/), matrix left division (\)

6. Addition (+), subtraction (-)
7. Colon operator (:)
8. Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equalto (==), not equal to (~=)
9. Element-wise AND (&)
10. Element-wise OR (|)
11. Short-circuit AND (&&)
12. Short-circuit OR (||)

REFERENCES

- Stephen J Chapman, “ Programming in MATLAB for Engineers”, Brooks, 2002
- Duane Hanselman ,Bruce LittleField, “Mastering MATLAB 7” , Pearson Education Inc, 2005
- William J.Palm, “Introduction to MATLAB 6.0 for Engineers”, Mc Graw Hill & Co, 2001
- M.Herniter, “Programming in MATLAB”, Thomson Learning, 2001

QUESTION BANK

PART A	CO
1. List the various windows available in MATLAB?	1
2. What is matrix?	1
3. List 5 built in functions in MATLAB	1
4. Develop MATLAB function for finding the determinant of a matrix?	1
5. List the various relational operators in the MATLAB?	1
6. What is the use of strcat functions	1
7. Compare floor and ceil functions in MATLAB	1
8. What is the use of getfield command in MATLAB?	1
9. What is the need for celldisp command in MATLAB?	1
10. What is the use of CHAR command in MATLAB	1

PART B

1. Design a string calculator in MATLAB that performs various string operations Compare and contrast the various input and output statements in MATLAB support your answer with suitable examples. 1
2. Develop a cell array(a) whose size is 2*2. 1

    a(1,1)= ‘ India Australia Matches’
    a(1,2)=[50 , 60, 100]
    a(2,1)=’Viratkohli’
    a(2,2)=[ ];

use preallocation and assignment statements .Also display 60. 1

3. Develop a software to assist the mentor in maintaining the address details of each student use structure array 1
4. Explain the various ways of reading and writing data into files. Illustrate with suitable examples 1
5. Explain the various data types used in MATLAB .support your

answer with suitable examples . 1

6. Develop a function foe finding factorial of a number. Also design MATLAB codes for finding binomial co-efficient. 1

7. Explain different methods of string and accessing values from matrices and vectors in MATLAB 1

SCHOOL OF ELECTRICAL AND ELECTRONICS
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – II – Software Tools for Engineering Applications – SECA4001

UNIT 2 LOOPS & CONTROL STATEMENTS

Control Flow & Decision statements- IF - IF ELSE - NESTED IF ELSE - SWITCH - TRY & CATCH - FOR -WHILE – NESTED FOR - FOR with IF statements, MATLAB program organization, Debugging methods- Setting and running with breakpoints – Examining values-Correcting and ending debugging, Example programs using the above commands. MISCELLANEOUS TOPICS: Date & Time Functions, Time Computations, File & Directory management

BRANCHES & LOOPS

BRANCHES are MATLAB statements that permit us to select and execute specific section of code called blocks,while skipping other sections of code.

- if
- switch
- try-catch

LOOPS are MATLAB construct that allow us to execute a sequence of statements more than once

- for
- while

The If Statement

The **if** statement chooses whether or not another statement, or group of statements, is executed.The general form of the **if** statement is:

```
if condition
        action
end
```

A ***condition*** is a relational expression that is conceptually, or logically, either true or false. The ***action*** is a statement, or a group of statements, that will be executed if the condition is true. When the **if** statement is executed, first the condition is evaluated. If the value of the condition is conceptually true, the action will be executed, and if not, the action will not be executed. The action can be any number of statements until the reserved word **end**; the action is naturally bracketed by the reserved words **if** and **end**. (Note: This is different from the **end** that is used as an index into a vector or matrix.)

For example, the following **if** statement checks to see whether the value of a variable is negative. If it is, the value is changed to a positive number by using the absolute value function; otherwise nothing is changed.

```
if num < 0
        num = abs(num)
end
```

If statements can be entered in the Command Window, although they generally make more sensein scripts or functions. In the Command Window, the **if** line would be entered, then the Enter key, then the action, the Enter key, and finally **end** and Enter; the results will immediately follow. For example, the previous **if** statement is shown twice here. Notice that the output from the assignment is not suppressed, so the result of theÂ€action will be shown if the action is executed. The first time the value of the variable is negative so the action is executed and the variable is modified, but in the second case the variable is positive so the action is skipped.

```
>> num = −4;
>> if num < 0
      num
      =
      abs(n
      um)
      end
      num = 4
>> num = 5;
>> if num < 0
      num = abs(num)
end
>>
```

This may be used, for example, to make sure that the square root function is not used on a negative number. The following script prompts the user for a number, and prints the square root. If the user enters a negative number, the **if** statement changes it to positive before taking the square root.

```
sqrtifexamp.m
% Prompt the user for a number and print
itssqrt num = input('Please enter a
number: ');
% If the user entered a negative
number,change it
if num < 0
    num =
     abs(n
     um);
```

```
        end
fprintf('The sqrt of %.1f is %.1f\n',num,sqrt(num))
```

Here are two examples of running this script:

>> *sqrtifexamp*

Please enter a number:
−4.2The sqrt of 4.2 is
2.0

>> *sqrtifexamp*

Please enter a
number: 1.44
The sqrt of 1.4
is 1.2

In this case, the action of the **if** statement was a single assignment statement. The action can be any number of valid statements. For example, we may wish to print a note to the user to say that the number entered was being changed.

```
sqrtifexampii.m
% Prompt the user for a number and print its sqrt
num =input('Please enter a number: ');
% If the user entered a negative number, tell
        % the user
        and change it
        if num < 0
        disp('OK, we''ll use the
        absolute value')num =
        abs(num);
end
fprintf('The sqrt of %.1f is %.1f\n',num,sqrt(num))
```

>> *sqrtifexampii*

Please enter a number: −25 OK, we'll
use theabsolute value
The sqrt of 25.0 is 5.0
Notice the use of two single quotes in the **disp** statement in order to print one single quote

The If-Else **statement**

The **if** statement chooses whether an action is executed or not. Choosing between two actions, or choosing from several actions, is accomplished using **if-else**, nested **if**, and **switch** statements. The **if-else** statement is used to choose between two statements, or sets ofstatements.

The general form is:

```
if
conditi
on
action1
else
action2
end
```

First, the condition is evaluated. If it is conceptually true, then the set of statements designated as action1 is executed, and that is it for the **if-else** statement. If instead the condition is conceptually false, the second set of statements designated as action2 is executed, and that‘s it. The first set of statements is called the action of the **if** clause; it is what will be executed if the expression is true. The second set of statements is called the action of the **else** clause; it is what will be executed if the expression is false. One of these actions, and only one, will be executed—which one depends on the value of the condition. For example, to determine and print whether ornot a random number in the range from 0 to 1 is less than 0.5, an **if-else** statement could be used:

```
if rand < 0.5
disp(‘It was less than
.5!‘)else
disp(‘It was not less than
.5!‘)end
```

One application of an **if-else** statement is to check for errors in the inputs to a script. For example, an earlier script prompted the user for a radius, and then used that to calculate the area of a circle. However, it did not check to make sure that the radius was valid (e.g., a positive number). Here is a modified script that checks the radius:

```
checkradius.m
% This script calculates the area of a circle
% It error-checks the user‘s radius
radius = input(‘Please enter the
radius: ‘);if radius <= 0
          fprintf(_Sorry; %.2f is not a valid radius\n‘,radius)

else



end
```

Examples of running this script when the user enters invalid and then valid radii are shown here:

>> *checkradius*

Please enter the radius: −4
Sorry; −4.00 is not a valid radius

>> *checkradius*

Please enter the radius: 5.5
For a circle with a radius of 5.50, the area is 95.03

The **if-else** statement in this example chooses between two actions: printing an error message, or actually using the radius to calculate the area, and then printing out the result. Notice that the action of the **if** clause is a single statement, whereas the action of the **else** clause is a group of three statements.

Nested If-Else Statements

The **if-else** statement is used to choose between two statements. In order to choose from more than two statements, the **if-else** statements can be nested, one inside of another. For example, consider implementing the following continuous

mathematical function y = f(x):
y = 1 for x < −1
y = x2 for −1 ≤ x
≤ 2y = 4 for x >
2

The value of y is based on the value of x, which could be in one of three possible ranges. Choosingwhich range could be accomplished with three separate **if** statements, as follows:

```
if x < −1
        y = 1;
end
if x > = −1 && x < = 2
        y = x^2;
end
if x > 2
        y= 4;
end
```

Since the three possibilities are mutually exclusive, the value of *y* can be determined by using three separate **if** statements. However, this is not very efficient code: all three Boolean expressions must be evaluated, regardless of the range in which *x* falls. For example, if *x* is less than –1, the first expression is true and 1 would be assigned to *y*. However, the two expressionsin the next two **if** statements are still evaluated. Instead of writing it this way, the expressions canbe ***nested*** so that the statement ends when an expression is found to be true:

f x < −1 else

y = 1;

% If we are here, x must be > = −1

% Use an if-else statement to choose

% between the two remaining ranges if x > = −1 && x < = 2

y = x^2; else

% No need to check

% If we are here, x must be > 2 y = 4;

end

```
end
```

By using a nested **if-else** to choose from among the three possibilities, not all conditions must be tested as they were in the previous example. In this case, if *x* is less than –1, the statement to assign 1 to *y* is executed, and the **if-else** statement is completed so no other conditions are tested. If, however, *x* is not less than –1, then the else clause is executed. If the else clause is executed, then we already know that *x* is greater than or equal to –1 so that part does not need to be tested. Instead, there are only two remaining possibilities: either *x* is less than or equal to 2, or it is greater than 2. An **if-else** statement is used to choose between those two possibilities. So, the action of the **else** clause was another **if-else** statement. Although it is long, this is one **if-else** statement, a nested **if-else** statement. The actions are indented to show the structure. Nesting **if- else** statements in this way can be used to choose from among three, four, five, six, or more options—the possibilities are practically endless! This is actually an example of a particular kind of nested **if-else** called a cascading **if-else** statement. In this type of nested **if-else** statement, the conditions and actions cascade in a stair-like pattern.

For example, if there are *n* choices (where *n* > 3 in this example), the following general

formwould be used: if condition1
Action1
elseif condition2action2
elseif condition3action3
% etc: there can be
many ofthese else
action n % the nth action
end

The actions of the **if**, **elseif**, and **else** clauses are naturally bracketed by the reserved words **if**, **elseif**, **else**, and **end**. For example, the previous example could be written using the **elseif** clause rather than nesting **if- else** statements:

So, there are three ways of accomplishing this task: using three separate **if** statements, using nested **if- else** statements, and using an **if** statement with **elseif** clauses, which is the simplest. This could be implemented in a function that receives a value of x and returns the corresponding value of y:

Another example demonstrates choosing from more than just a few options. The following function receives an integer quiz grade, which should be in the range from 0 to 10. The program then returns a corresponding letter grade, according to the following scheme: a 9 or 10 is an ‗A‘, an8 is a ‗B‘, a 7 is a ‗C‘, a 6 is a ‗D‘, and anything below that is an ‗F‘. Since the possibilities are mutually exclusive, we could implement the grading scheme using separate **if** statements. However, it is more efficient to have one **if- else** statement with multiple **elseif** clauses. Also, the function returns the value ‗X‘ if the quiz grade is not valid. The function does assume that the input is an integer.

```
letgrade.m
function grade = letgrade(quiz)
% This function returns the letter grade corresponding
% to the integer quiz grade argument
% First, error-check
       if quiz < 0 || quiz > 10
       grade = ‗X‘;
       % If here, it is valid so figure out the
       %   corresponding
       letter  grade elseif
       quiz  ==  9  ||  quiz
       ==   10   grade   =
       ‗A‘;
       elseif quiz == 8
               grade
       =‗B‘;elseif
       quiz == 7
               grade
       = ‗C‘;elseif
       quiz == 6
       grade = ‗D‘;
 else        end
```

grade = _F‘;

Here are three examples of calling this function:
>> quiz = 8;

>> lettergrade = letgrade(quiz)

lettergrade
= B
>> quiz = 4;

>> letgrade(quiz)

ans
= F

>> quiz = 22;

>> lg = letgrade(quiz)

lg = X

In the part of this **if** statement that chooses the appropriate letter grade to return, all the Boolean expressions are testing the value of the variable *quiz* to see if it is equal to several possible values, in sequence (first 9 or 10, then 8, then 7, etc.). This part can be replaced by a **switch** statement.

The Switch Statement

A **switch** statement can often be used in place of a nested **if-else** or an **if** statement with many **elseif** clauses. Switch statements are used when an expression is tested to see whether it is *equal to* one of several possible values. The general form of the **switch** statement is:

```
    switch switch_expressioncase caseexp1
          action1 case caseexp2
          action2 case caseexp3
          action3
 % etc: there can be
many ofthese otherwise
          action n
end
```

The **switch** statement starts with the reserved word **switch**, and ends with the reserved word **end**. The switch_expression is compared, in sequence, to the case expressions (caseexp1, caseexp2, etc.). If the value of the switch_expression matches caseexp1, for example, then action1 is executed and the **switch** statement ends. If the value matches caseexp3, then action3 is executed, and in general if the value matches caseexpi, where i can be any integer from 1 to n, then actioni is executed. If the value of the switch_expression does not match any of the case expressions, the action after the word **otherwise** is executed. For the previous example, the **switch** statement can be used as follows:

```
switchletgrade.m
function grade = switchletgrade(quiz)
% This function returns the letter grade corresponding
% to the integer quiz grade argument using switch
        % First, error-check if quiz < 0 ||
                quiz > 10grade = _X‘;

          % If here, it is valid so figure out the
else      % corresponding letter grade using a
          switch switch quiz
          case 10
                grade= 'A‘;
          case 9
                grade = _D‘;
          case 8
    grade=‘B‘;
          case 7
    grade= ‘A‘;
          case 6
                otherwise
                        grade = _F‘;
                end
end
```

Here are two examples of calling this function:

```
        >> quiz = 22;

        >> lg = switchletgrade(quiz)

lg = X
        >> quiz = 9;

        >> switchletgrade(quiz)

ans
= A
```

Note that it is assumed that the user will enter an integer value. If the user does not, either an error message will be printed or an incorrect result will be returned. Since the same action of printing _A‗ is desired for more than one case, these can be combined as follows:

```
switch quiz
    case {10,9}
        grade = _A‗;
    case 8
        grade = _B‗;
    % etc.
end
```

(The curly braces around the case expressions 10 and 9 are necessary.) In this example, we error-checked first using an **if-else** statement, and then if the grade was in the valid range, used a **switch** statement to find the corresponding letter grade.

Sometimes the **otherwise** clause is used instead for the error message. For example, ifthe user is supposed to enter only a 1, 3, or 5, the script might be organized as follows:

switcherror.m

```
% Example of otherwise for error
message choice =input(‗Enter a 1, 3, or 5: ‗);
switch choice
    case 1
        disp(_It‗‗s a one!!‗)
    case 3
        disp(_It‗‗s a three!!‗)
    case 5
        disp(_It‗‗s a five!!‗)
    otherwise
        disp(_Follow directions next time!!‗)
end
```

In this case, actions are taken if the user correctly enters one of the valid options. If the user does not, the **otherwise** clause handles printing an error message. Note the use of two single quotes within the string to print one.

>> switcherror

Enter a 1, 3, or 5: 4
Follow directions next time!!

The for **Loop**

The **for** statement, or the **for** loop, is used when it is necessary to repeat statement(s) in a scriptor function, and when it *is* known ahead of time how many times the statements will be repeated. The statements that are repeated are called the action of the loop. For example, it may be known that the action of the loop will be repeated five times. The terminology used is that we ***iterate*** through the action of the loop five times. The variable that is used to iterate through values is called a ***loop variable***, or an ***iterator variable***. For example, the variable might iterate through the integers 1 through 5 (e.g., 1, 2, 3, 4, and then 5). Although variable names in general should be mnemonic, it is common for an iterator variable to be given the name *i* (and if more than one iterator variable is needed, *i*, *j*, *k*, *l*, etc.) This is historical, and is because of the way integer variables were named in Fortran. However, in MATLAB both **i** and **j** are built-in values for -1 ,so using either as a loop variable will override that value. If that is not an issue, then it is acceptable to use *i* as a loop variable. The general form of the **for** loop is:

```
for loopvar = rangeaction
end
```

where *loopvar* is the loop variable, *range* is the range of values through which the loop variable is to iterate, and the action of the loop consists of all statements up to the **end**. The range can be specified using any vector, but normally the easiest way to specify the range of values is to use the colon operator. As an example, to print a column of numbers from 1 to 5:

```
for i = 1:5
        fprintf(‗%d\n‘,i)
end
```

This loop could be entered in the Command Window, although like **if** and **switch** statements, loops will make more sense in scripts and functions. In the Command Window, the results wouldappear after the **for** loop:

>> for i = 1:5 fprintf('%d\n',i)

e

n

d

1
2
3
4
5

What the **for** statement accomplished was to print the value of *i* and then the newline character for every value of *i*, from 1 through 5 in steps of 1. The first thing that happens is that *i* is initialized to have the value

1. Then, the action of the loop is executed, which is the **fprintf** statement that prints the value of*i* (1), and then the newline character to move the cursor down. Then, *i* is incremented to have the value of 2. Next, the action of the loop is executed, which prints 2 and the newline. Then, *i* is incremented to 3 and that is printed, then *i* is incremented to 4 and that is printed, and thenfinally *i* is incremented to 5 and that is printed. The final value of *i* is 5; this value can be used once the loop hasfinished.

Finding Sums and Products

A very common application of a **for** loop is to calculate sums and products. For example, insteadof just printing the integers 1 through 5, we could calculate the sum of the integers 1 through 5 (or, in general, 1 through *n*, where *n* is any positive integer). Basically, we want to implement or calculate the sum 1 + 2 + 3

+ … + n. In order to do this, we need to add each value to a ***running sum***. A running sum is a sum that will keep changing; we keep adding to it. First the sum has to be initialized to 0, then in this case it will be 1 (0 + 1), then 3 (0 + 1 + 2), then 6 (0 + 1 + 2 + 3), and so forth. In a function to calculate the sum, we need a loop or iterator variable *i*, as before, and also a variable to store the running sum. In this case we will use the output argument *runsum* as the running sum. Every time through the loop, the next value of *i* is added to the value of *runsum*. This function will return the end result, which is the sum of all integers from 1 to the input argument *n* stored in the output argument *runsum*.

```
sum_1_to_n.m
function runsum = sum_1_to_n(n)
% This function returns the sum of
% integers from
1 ton runsum =
0;
for i = 1:n
```

```
        runsum = runsum + i;
end
```

As an example, if 5 is passed to be the value of the input argument *n*, the function will calculate and return 1 + 2 + 3 + 4 + 5, or 15:

```
>> sum_1_to_n(5)

    ans
   = 15
```

Note that the output was suppressed when initializing the sum to 0 and when adding to it during the loop. Another very common application of a **for** loop is to find a ***running product***. For example, instead of finding the sum of the integers 1 through n, we could find the product of the integers 1 through n. Basically, we want to implement or calculate the product 1 * 2 * 3 * 4 *…
* n, which is called the ***factorial*** of n, written n!.

For Loops that Do Not Use the Iterator Variable in the Action

In all the examples that we have seen so far, the value of the loop variable has been used in someway in the action of the **for** loop: we have printed the value of *i*, or added it to a sum, or multiplied it by a running product, or used it as an index into a vector. It is not always necessary to actually use the value of the loop variable, however. Sometimes the variable is simply used to iterate, or repeat, a statement a specified number of times. For example,

```
for i = 1:3
        fprintf(‗I will not chew gum\n‘)
end
```

produces the output:

```
I will not chew
gum I will not
chew gum I will
not chew gum
```

The variable *i* is necessary to repeat the action three times, even though the value of *i* is not usedin the action of the loop.

Nested for Loops

The action of a loop can be any valid statement(s). When the action of a loop is another loop, this is called a ***nested loop***. As an example, a nested **for** loop will be demonstrated in a scriptthat will print a box of *‘s. Variables in the script will specify how many rows and columns to print. For example, if *rows* has the value 3, and *columns* has the value 5, the

output would be:

*****
*****
*****

Since lines of output are controlled by printing the newline

- character, the basicalgorithm is: For every row of output,
- Print the required number of *'s
- Move the cursor down to the next line (print the ‗\n')

printstars.m
% Prints a box of stars
% How many will be specified by 2 variables
% for the number of rows
andcolumns rows = 3;
columns = 5;
% loop over
the rows for
i=1:rows
    % for every row loop to print *'s and then one \n for
j=1:columnsfprintf(‗*')
        end fprintf(‗\n')
end

Running the script displays the output:

>> *printstars*

*****
*****
*****

The variable *rows* specifies the number of rows to print, and the variable *columns* specifies how many *'s to print in each row. There are two loop variables: *i* is the loop variable for the rows, and *j* is the loop variable for the columns. Since the number of rows and columns are known (given by the variables *rows* and *columns*), **for** loops are used. There is one **for** loop to loop over the rows, and another to print the required number of *'s. The values of the loop variables are notused within the loops, but are used simply to iterate the correct number of times. The first **for** loop specifies that the action will be repeated *rows* times. The action of this loop is to print *'s and then the newline character. Specifically, the action is to loop to print *columns* *'s across on one line. Then, the newline character is printed after all five stars to move the cursor down forthe next line. The first **for** loop is called the ***outer loop***; the second **for** loop is called the ***inner loop***. So, the outer loop is over the rows, and the inner loop is over the columns. The outer loop must be over the rows because the program is printing a certain number of rows of output. For each row, a loop is necessary to print the required number of *'s; this is the inner **for** loop. Whenthis script is executed, first the outer loop variable *i* is initialized to 1. Then, the action is executed. The action consists of the inner loop, and then printing the newline character. So, while the outer loop variable has the value 1, the inner loop variable *j* iterates through all its values. Since the value of *columns* is 5, the inner loop will print a * five times. Then, the newline character is printed and the outer loop variable *i* is incremented to 2. The action of the outer loop is then executed again, meaning the inner loop will print five *'s, and then the newline character will be printed. This continues, and in all, the action of the outer loop will be executed *rows* times. Notice the action of the outer loop consists of two statements (the **for** loop and an **fprintf** statement). The action of the inner loop, however, is only a single statement. The **fprintf** statement to print the newline character must be separate from the other **fprintf** statement that prints the *. If we simply had fprintf(‗*\n') as the action of the inner loop, this would print a long column of 15 *'s, not a box In these examples, the loop variables were used just to specify the number of times the action is to be repeated. These same loops could be used instead to produce

a multiplication table by multiplying the values of the loop variables. The following function *multtable* calculates and returns a matrix that is a multiplication table. Two arguments are passedto the function, which are the number of rows and columns for this matrix

```
multtable.m
function outmat = multtable (rows, columns)
% Creates a matrix which is a multiplication table
% Preallocate the matrix
outmat =
zeros(rows,columns); for i
= 1:rows
        for j = 1:columns
            outmat(i,j) = i * j;
        end
end
```

In the following example, the matrix has three rows and five columns:

>> multtable(3,5)

```
ans =
    1    2    3    4    5
    2    4    6    8    10
    3    6    9    12   15
```

Notice that this is a function that returns a matrix; it does not print anything. It preallocates the matrix to zeros, and then replaces each element. Since the number of rows and columns are known, **for** loops are used. The outer loop loops over the rows, and the inner loop loops over the columns. The action of the nested loop calculates i * j for all values of *i* and *j*. First, when *i* has the value 1, *j* iterates through the values 1 through 5, so first we are calculating 1 *

1, then 1 * 2, then 1 * 3, then 1 * 4, and finally 1 * 5. These are the values in the first row (first in element (1,1), then (1,2), then (1,3), then (1,4), and finally (1,5)). Then, when *i* has the value 2, the elements in the second row of the output matrix are calculated, as *j* again iterates through the values from 1 through 5. Finally, when *i* has the value 3, the values in the third row are calculated (3 * 1, 3 * 2, 3 * 3, 3 * 4, and 3 * 5). This function could be used in a script that prompts the user for the number of rows and columns, calls this function to return a multiplication table, and writes the resulting matrix to a file:

```
createmulttab.m
% Prompt the user for rows and columns and
% create a multiplication table to store in
% a file mymulttable.dat
num_rows = input('Enter the number of rows:
'); num_cols = input('Enter the number of
columns: ');multmatrix = multtable(num_rows,
num_cols); save mymulttable.dat multmatrix –
ascii
```

Here is an example of running this script, and then loading from the file into a matrix in order toverify that the file was created:

>> *createmulttab*

Enter the number of rows: 6
Enter the number of columns:
4

>> *load mymulttable.dat*

>> *mymulttable*

mymulttable =

1	2	3	4
2	4	6	8
3	6	9	12
4	8	12	16
5	10	15	20
6	12	18	24

Logical Vectors

The relational operators can also be used with vectors and matrices. For example, let‘s say that there is a vector, and we want to compare every element in the vector to 5 to determine whether it is greater than 5 or not. The result would be a vector (with the same length as the original) with logical true or false values. Assume a variable vec as shown here.

>> *vec = [5 9 3 4 6 11];*

In MATLAB, this can be accomplished automatically by simply using the relational operator >.

>> *isg = vec > 5*

isg =
0 1 0 0 1 1

Notice that this creates a vector consisting of all logical true or false values. Although this is a vector of ones and zeros, and numerical operations can be done on the vector *isg*, its type is **logical** rather than **double**.

>> *doubres = isg + 5*

ans =
5 6 5 5 6 6

>> *whos*

Name	Size	Bytes	Class
doubres	1x6 4	8	double array
isg	1x6	6	logical array
vec	1x6	48	double array

To determine how many of the elements in the vector *vec* were greater than 5, the **sum** function could be used on the resulting vector *isg*:

```
>>
sum(isg)
ans =
3
```

The ***logical vector*** *isg* can also be used to index into the vector. For example, if only the elements from the vector that are greater than 5 are desired:

>> *vec(isg)*

```
ans =
9          6     11
```

Because the values in the vector must be **logical** 1‘s and 0‘s, the following function that appears at first to accomplish the same operation using the programming method, actually does not. The function receives two input arguments: the vector, and an integer with which to compare (so it is somewhat more general). It loops through every element in the input vector, and stores in the result vector either a 1 or 0 depending on whether vec(i) > n is true or false.

```
testvecgtn.m
function outvec = testvecgtn(vec,n)
% Compare each element in vec to see whether it
% is greater than n or not
% Preallocate the vector
outvec = zeros(size(vec));
for i = 1:length(vec)
        % Each element in the output vector stores 1
         or 0if vec(i) > n
         outvec(i
          ) = 1;
          else
         outvec(i
         ) = 0;end
end
```

Calling the function appears to return the same vector as simply vec > 5, and summing the result still works to determine how many elements were greater than 5.

>> *notlog = testvecgtn(vec,5)*

```
notlog =
```

0 1 0 0 1 1

>> *sum(notlog)*

ans =

3

However, as before, it could not be used to index into a vector because the elements are **double**, not

logical:

>> *vec(notlog)*

??? Subscript indices must either be real positive integers or logicals.

While **Loops**

The **while** statement is used as the conditional loop in MATLAB; it is used to repeat an action when ahead of time it is *not* known *how many* times the action will be repeated. The general form of the **while** statement is:

```
    while
condition
   action
end
```

The action, which consists of any number of statement(s), is executed as long as the condition is true. The condition must eventually become false to avoid an ***infinite loop***. (If this happens, Ctrl-C will exit the loop.) The way it works is that first the condition is evaluated. If it is logicallytrue, the action is executed. So, to begin with it is just like an **if** statement. However, at that point the condition is evaluated again. If it is still true, the action is executed again. Then, the action is evaluated again. If it is still true, the action is executed again. Then, the action is… eventually, this has to stop! Eventually something in the action has to change something in the condition so it becomes false. As an example of a conditional loop, we will write a function that will find the first factorial that is greater than the input argument *high*. Previously, we wrote a function to calculate a particular factorial. For example, to calculate 5! we found the product 1 * 2 * 3 * 4 * 5. In that case a **for** loop was used, since it was known that the loop would be repeated five times. Now, we do not know how many times the loop will be repeated. The basic algorithm is to have two variables, one that iterates through the values 1, 2, 3, and so on, and one that stores the factorial of the iterator at each step. We start with 1, and 1 factorial, which is 1. Then, we check the factorial. If it is not greater than *high*, the iterator variable will then increment to 2, and find its factorial (2). If this is not greater than *high*, the iterator will then increment to 3, and the function will find its factorial (6). This continues until we get to the first factorial that is greater than *high*. So, the process of incrementing a variable and finding its factorial is repeated until we get to the first value greater than *high*. This is implemented using a **while** loop:

```
factgthigh.m
function facgt = factgthigh(high)
% Finds the first factorial >
high i=0;
fac=1;
```

```
while fac <= high
        i=i+1;
        fac = fac * i;
end
facgt = fac;
```

Here is an example of calling the function, passing 5000 for the value of the input argument *high.*

    >> factgthigh(5000)

ans
= 5040

The iterator variable *i* is initialized to 0, and the running product variable *fac*, which will store the factorial of each value of *i*, is initialized to 1. The first time the **while** loop is executed, the condition is conceptually true: 1 is less than or equal to 5000. So, the action of the loop is executed, which is to increment *i* to 1 and *fac* to 1 (1 * 1). After the execution of the action of the loop, the condition is evaluated again. Since it will still be true, the action is executed: *i* is incremented to 2, and *fac* will get the value 2 (1 * 2). The value 2 is still <= 5000, so the action will be executed again: *i* will be incremented to 3, and *fac* will get the value 6 (2 * 3). This continues until the first value of *fac* is found that is greater than 5000. As soon as *fac* gets to this value, the condition will be false and the **while** loop will end. At that point the factorial is assigned to the output argument, which returns the value. The reason that *i* is initialized to 0 rather than 1 is that the first time the loop action is executed, *i* becomes 1 and *fac* becomes 1 so we have 1 and 1!, which is 1. Notice that the output of all assignment statements is suppressed in the function.

Multiple Conditions in a While Loop

In the previous section, we wrote a function *myany* that imitated the built-in **any** function by returning logical true if any value in the input vector was logical true, and logical false otherwise. The function was inefficient because it looped through all the elements in the input vector, even though once one logical true value is found it is no longer necessary to examine any other elements. A **while** loop will improve on this. Instead of looping through all the elements, whatwe really want to do is to loop until either a logical true value is found, or until we‘ve gone through the entire vector. Thus, we have two parts to the condition in the **while** loop. In the following function, we initialize the output argument to logical false, and an iterator variable *i* to

The action of the loop is to examine an element from the input vector: if it is logical true, we change the output argument to be logical true. Also in the action the iterator variable is incremented. The action of the loop is continued as long as the index has not yet reached the end of the vector, and as long as the output argument is still logical false.

```
myanywhile.m
function logresult = myanywhile(vec)
% Simulates the built-in function any
% Uses a while loop so that the action halts
% as soon as any true value
isfound logresult =
logical(0);
i = 1;
        while i <= length(vec) &&
                  logresult == 0if vec(i) = 0
                  logresult =
                   logical(1);
                   end
                  i = i + 1;
end
```

The output produced by this function is the same as the *myany* function, but it is more efficient because now as soon as the output argument is set to logical true, the loop ends.

Debugging Techniques

Any error in a computer program is called a ***bug***. This term is thought to date back to the 1940s, when a problem with an early computer was found to have been caused by a moth in the computer's circuitry! The process of finding errors in a program, and correcting them, is still called ***debugging***.

Types of Errors

There are several different kinds of errors that can occur in a program, which fall into the categories of ***syntax errors, run-time errors***, and ***logical errors***. Syntax errors are mistakes in using the language. Examples of syntax errors are missing a comma or a quotation mark, or misspelling a word. MATLAB itself will flag syntax errors and give an error message. For example, the following string is missing the end quote:

>> mystr = 'how are you;

```
??? mystr = _how are you;
                |
Error: A MATLAB string constant is not terminated properly.
```

Another common mistake is to spell a variable name incorrectly, which MATLAB will also catch.

>> value = 5;

>> newvalue = valu + 3;

??? Undefined function or variable 'valu'.

Run-time, or execution-time, errors are found when a script or function is executing. With most languages, an example of a run-time error would be attempting to divide by zero. However, in MATLAB, this will generate a warning message. Another example would be attempting to refer to an element in an array that does not exist.

runtime_ex.m

```
% This script shows an execution-
timeerror vec = 3:5;
for i = 1:4
        disp(vec(i))
end
```

This script initializes a vector with three elements, but then attempts to refer to a fourth. Running it prints the three elements in the vector, and then an error message is generated when it attempts to refer to the fourth element. Notice that it gives an explanation of the error, and it gives the line number in the script in which the error occurred.

    >> *runtime_ex*

```
3
4
5
??? Attempted to access vec(4); index out of bounds
becausenumel(vec)=3.
        Error in ==>
                runtime_ex at
                6disp(vec(i))
```

Logical errors are more difficult to locate, because they do not result in any error message. A logical error is a mistake in reasoning by the programmer, but it is not a mistake in the programming language. An example of a logical error would be dividing by 2.54 instead of multiplying in order to convert inches to centimeters. The results printed or returned would be incorrect, but this might not be obvious. All programs should be robust and should wherever possible anticipate potential errors, and guard against them. For example, whenever there is input into a program, the program should error-check and make sure that the input is in the correct range of values. Also, before dividing, the denominator should be checked to make sure that it is not zero. Despite the best precautions, there are bound to be errors in programs.

Tracing

Many times, when a program has loops and/or selection statements and is not running properly, itis useful in the debugging process to know exactly which statements have been executed. For example, here is a function that attempts to display In Middle of Range if the argument passed toit is in the range from 3 to 6, and Out of Range otherwise.

```
testifelse.m
function testifelse(x)
% This function will test the debugger

if 3 < x < 6
                disp('In middle of range')
else

      disp('Out of
  range')end
```

However, it seems to print In Middle of Range for all values of x:

    >> *testifelse(4)*

In middle of range

>> *testifelse(7)*

In middle of range

>> *testifelse(–2)*

In middle of range

One way of following the flow of the function, or ***tracing*** it, is to use the **echo** function. The **echo** function, which is a toggle, will display every statement as it is executed as well as results from the code. For scripts, just **echo** can be typed, but for functions, the name of the function must be specified, for example, echo function name on/off

>> *echo testifelse on*

Editor/Debugger

MATLAB has many useful functions for debugging, and debugging can also be done through its editor, called the Editor/Debugger. Typing **help debug** at the prompt in the Command Window will show some of the debugging functions. Also, in the Help Browser, clicking the Search tab and then typing **debugging** will display basic information about the debugging processes. It can be seen in the previous example that the action of the **if** clause was executed and it printed In Middle of Range, but just from that it cannot be determined why this happened. There are several ways to set ***breakpoints*** in a file (script or function) so that the variables or expressions can be examined. These can be done from the Editor/Debugger, or commands can be typed from the Command Window. For example, the following **dbstop** command will set a breakpoint in the fifth line of this function (which is the action of the **if** clause), which allows us to type variable names and/or expressions to examine their values at that point in the execution. The function **dbcont** can be used to continue the execution, and **dbquit** can be used to quit the debug mode. Notice that the prompt becomes K>> in debug mode.

>> dbstop testifelse 5

>> testifelse(–2)

5 disp(‗In middle of range')
K>> x

x =
 –2
K>> 3 < x

ans =
 0
K>> 3 < x < 6

ans =
 1
K>> dbcont

In middle of range end
>>
By typing the expressions 3 < x and then 3 < x < 6, we can determine that the expression 3 < x will return either 0 or 1. Both 0 and 1 are less than 6, so the expression will always be true, regardless of the value of x!

Function Stubs

Another common debugging technique, which is used when there is a script main program that calls many functions, is to use ***function stubs***. A function stub is a placeholder, used so that the script will work even though that particular function hasn‘t been written yet. For example, a programmer might start with a script main program that consists of calls to three function that accomplish all the tasks.

```
mainmfile.m
% This program gets values for x and y, and
% calculates and prints
z [x, y] = getvals;
z = calcz(x,y);
printall(x,y,z)
```

The three functions have not yet been written, however, so function stubs are put in place so that the script can be executed and tested. The function stubs consist of the proper function headers, followed by a simulation of what the function will eventually do (e.g., it puts arbitrary values in for the output arguments).

```
getvals.m
function [x, y] = getvals
x = 33;
y = 11;
calcz.m

function z = calcz(x,y)
z = 2.2;

printall.m
function printall(x,y,z)
disp(‘Something‘)
```

Then, the functions can be written and debugged one at a time. It is much easier to write a working program using this method than to attempt to write everything at once—then, when errors occur, it is not always easy to determine where the problem is!

EXAMPLE PROBLEMS FOR BRANCHES

```
% TO FIND SOLUTION OF QUADRATIC EQUATION
a=input('a =');

b=input('b=');

c=input('c=');

y=b^2 -

4*a*c;if y>0

        x1=(-b+sqrt(y))/(2*a);

        x2=(-b-

        sqrt(y))/(2*a);

        disp(x1)

        disp(x2)

  disp('The roots are real and

distinct')elseif y==0

        x1=-

        b/(2*

        a);

        disp(

        x1)

  disp('The roots are real and

repeated')else

        x1=-b/(2*a);

        x2=sqrt(abs(y))/(

        2*a);

        s1=x1+1i*x2;

        s2=x1

        -

        1i*x2

        ;

        disp(s

        1)
```

```
            disp(s
            2)
    disp('The roots are complex
conjugate')end
```

OUTPUT

```
>>
quadra_solna
=1
b=4
c=6
          -2.0000 + 1.4142i
          -2.0000 - 1.4142i
The roots are complex conjugate
```

```
% Example for switch case
value= input('enter the
value');switch(value)
            case{1,3,5,7,9,11}
               disp('odd
            number')
            case{2,4,6,8,
            10}
               disp('even
            number')
            otherwise
               disp('number out of range')
end
```

OUTPUT

>> sw_cs

enter the

value3odd

number

>> sw_cs

enter the

value6even

number

>> sw_cs

enter the value20

number out of

range

```
% TO BUILD A CALCULATOR USING SWITCH CASE
a=input('enter a matrix');
b=input('enter b matrix');
option=input('enter option as string');
switch option
        case '+'
          c=a+b;
           case '-'
          c=a-b;
          case '*'
          c=a*b;
          case'inv'
          c=inv(a);otherwise
          disp('invalid operation')
end
disp(c)
```

OUTPUT

```
>> calci
enter a matrix [2 3 4;5 6 7;1 8 9]
enter b matrix[1 2 3;-1 -2 -3;4 5
6]enter option as string'+'
        3     5     7
        4     4     4
        5    13    15
```

```
>> calci

enter a matrix[1 2 3;-1 -2 -3;4 5 6]

enter b matrix[2 3 4;5 6 7;1 8 9]

enter option as string't'

     1     -1     4

     2     -2     5

     3     -3     6
```

```
% EXAMPLE FOR TRY/CATCH
a=[1 5 -6 8 9 -2];
try
        ind=input('Enter the subscript of the element to be
        displayed');disp(num2str(a(ind)));
catch
  disp(['Illegal subscript:' num2str(ind)]);
end
```

OUTPUT

```
>> ty_ct

Enter the subscript of the element to be displayed3

-6

>> ty_ct

Enter the subscript of the element to be displayed8

Illegal subscript:8
```

% PROGRAM TO ADD FIRST 10 DIGITS

```
n=0;
sum=0;
while
n<11
        sum
        =
        sum
        +n;
        n=n
        +1;
end
fprintf('The sum of first 10 numbers \t %d \n',sum)
```

OUTPUT

```
>> sum_10_digit

The sum of first 10 numbers          55
```

% PROGRAM TO FIND SQUARE OF INTEGERS LESS

```
THAN 5k=0;
      while k<5 k=k+1; ksq=k^2;
  fprintf('square of %d is %d\n',
k,ksq);end
```

OUTPUT

```
>> sq_no square of 1 is 1
square of 2 is 4
square of 3 is 9
square of 4 is 16
square of 5 is 25
```

%PROGRAM TO OBTAIN SUM OF ALL INTEGERS FROM 0 TO 20

```
sum=0; for i=0:20
        sum=sum+i;end
disp(sum)
```

OUTPUT

```
>> sum_20_no210
```

% TO FIND MEAN, VARIANCE AND STANDARD DEVIATION OF A DATA

```
data = input('Enter the
dataset');n= length(data);
mean_value =
sum(data)/n;for i=1:n
        diff= data -
        mean_value;
        var=sum(diff.
        ^2)/(n-1);
        std=sqrt(var);
end
disp(mean_va
lue)disp(var)
disp(std)
```

OUTPUT

```
>> mean_std
Enter the dataset[10 20 30 40 50]
        30

       250

       15.8114
```

FILE & DIRECTORY MANAGEMENT

Table 2.1 File Operation

Action	Function Alternative
Create a new folder	Use the mkdir function. For example, create a subfolder named newdir in a parent folder named parentFolder: mkdir('parentFolder','newdir');
Move a file or folder	Use the movefile function. For example, move the file named myfile.min the current folder to the folder, d:/work: movefile('myfile.m','d:/work');
Rename a file or folder	Use the movefile function. For example, in the current folder, renamemyfile.m to oldfile.m: movefile('myfile.m','oldfile.m');
Open a file in MATLAB	Use the open function. The file opens in MATLAB or in an external application, depending on the file extension.
Delete a file or folder	To delete a file, use the delete function. For example, delete a file named myfile.m in the current folder: delete('myfile.m'); By default, files are permanently removed. To move deleted files to a temporary folder instead, use the recycle function or set the **Deleting files** preference. To delete a folder, use the rmdir function.

Table 2.2 Low Level File I/O operations

fclose	Close one or all open files
feof	Test for end of file
ferror	File I/O error information
fgetl	Read line from file, removing newline characters
fgets	Read line from file, keeping newline characters
fileread	Read contents of file as text
fopen	Open file, or obtain information about open files
fprintf	Write data to text file
fread	Read data from binary file
frewind	Move file position indicator to beginning of open file
fscanf	Read data from text file
fseek	Move to specified position in file
ftell	Current position
fwrite	Write data to binary file

```
x = 100*rand(8,1);
fileID = fopen('nums1.txt','w');
fprintf(fileID,'%4.4f\n',x);
fclose(fileID);
```

View the contents of the file.

```
type nums1.txt

81.4724
90.5792
```

12.6987
91.3376
63.2359
9.7540
27.8498
54.6882

A = fscanf(fileID,formatSpec)

81.4724

90.5792

12.6987

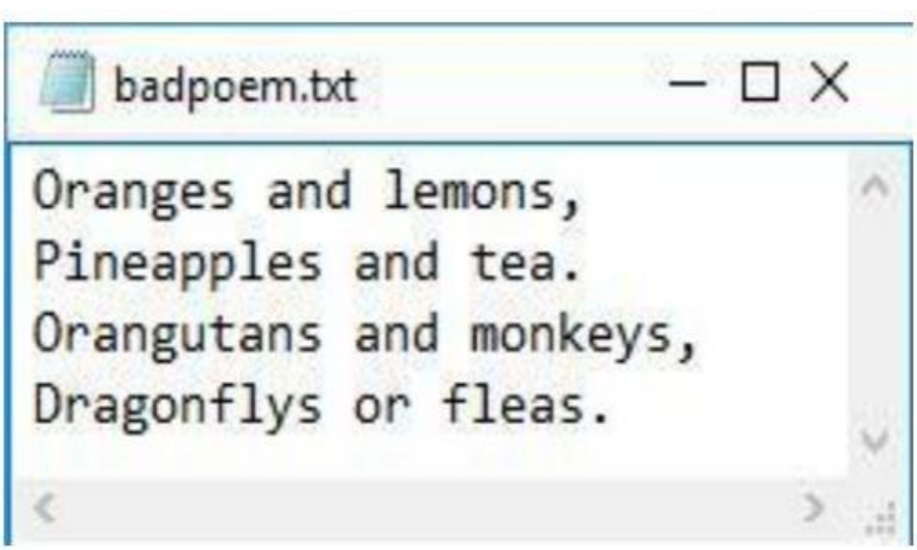
91.3376

63.2359

9.7540

27.8498

54.6882


badpoem.txt

Oranges and lemons,
Pineapples and tea.
Orangutans and monkeys,
Dragonflys or fleas.

```
fid = fopen('badpoem.txt');
 line_ex = fgetl(fid) % read line excluding
newline character
line_ex = 'Oranges and lemons,'
frewind(fid);
line_in = fgets(fid) % read line including newline character
line_in = 'Oranges and lemons,
```

```
fid = fopen('badpoem.txt');ftell(fid)

ans = 0

tline1 =
fgetl(fid) %
read the first
linetline1 =
'Oranges and
lemons,'
ftell(fid)

ans = 20

 Read the second line and
 examine the current position.
 tline2 = fgetl(fid) % read the
 second line
tline2 = 'Pineapples and tea.'fseek(fid,20,'bof');
fgetl(fid)

ans = 'Pineapples and tea.’

fid = fopen('badpoem.txt');while ~feof(fid)
tline = fgetl(fid);
disp(tline)
 End
```

Oranges and lemons, Pineapples and tea. Orangutans and monkeys, Dragonflys or fleas.

Table 2.3 File Format for Import & Export

File Content	Description	Import	Export
MATLAB	Saved MATLAB workspace	load	save

formatted data	Partial access of variables in MATLABworkspace	matfile	matfile
Text	Comma delimited numbers	csvread	csvwrite
	Delimited numbers	dlmread	dlmwrite
	Delimited numbers, or a mix of textand numbers	textscan	none
	Column-oriented delimited numbers or a mix of text andnumbers	readtable	Writetable
Spreadsheet	Worksheet or range of spreadsheet	xlsread	Xlswrite
	Column-oriented data in worksheetor range of spreadsheet	readtable	writetable

FTP FILE OPERATION

- FTP – File Transfer Protocol
- Connect to the server using the ftpfunction.
- Perform operations using the appropriate MATLAB® FTP functions, such as the cd, dir, and mget functions.
- Specify the FTP object for all operations.
- When you finish work on the server, close the connection using the closefunction

open the connection.
ftpobj = ftp('ftp.ngdc.noaa.gov')
ftpobj =
FTP Object

DMSP	Solid_Earth	
google12c4c939d7b90761.html	mgg	
INDEX.txt	coastwatch	hazards
pub		
README.txt	dmsp4alan	index.html
tmp		
STP	ftp.html	international
wdc		
Snow_Ice	geomag	ionosonde

host: ftp.ngdc.noaa.govuser:
anonymous
dir: /mode: binary

List the contents of the top-level folder on the FTP server. dir(ftpobj)

Download the file named INDEX.txt using the mget function. mget copies the file to the current MATLAB folder on your local machine. To view the contents of your copy of the file, use the type function.

mget(ftpobj,'INDEX.txt');type
INDEX.txt

DIRECTORY/FILE DESCRIPTION OF CONTENTS
--------- --
pub/ Public access area
DMSP/ Defense Meteorological Satellite Data Archivegeomag/
Geomagnetism and geomagnetics models
hazards/ Natural Hazards data, volcanoes, tsunamis, earthquakes
Change to the subfolder named pub on the FTP server.
cd(ftpobj,'pub')

ans = '/pub'

List the contents. pub is now the current folder on the FTP server. However, note that the current MATLAB folder on your local machine has not changed. When you specify an FTP object using functions such as cd and dir, the operations take place on the FTP server, not your local machine.

dir(ftpobj)
WebCD coast glac_lib krm outgoing
results rgonClose the connection to
the FTP server. close(ftpobj)

DATE AND TIME ARITHMETIC

Create a datetime scalar. By default, datetime arrays are not associated wtiha time zone.

t1 = datetime('now')t1 = datetime

26-Feb-2018 19:50:34

Find future points in time by adding a sequence of hours.

t2 = t1 + hours(1:3)

t2 = 1x3 datetime array
26-Feb-2018 20:50:34 26-Feb-2018 21:50:34   26-Feb-2018 22:50:34

Verify that the difference between each pair of datetime values in t2 is 1 hour.
dt = diff(t2)
dt = 1x2
duration
array
01:00:00
01:00:00

diff returns durations in terms of exact numbers of hours, minutes,

and seconds.Subtract a sequence of minutes from a datetime to find

past points in time.
t2 = t1 - minutes(20:10:40)
t2 = 1x3 datetime array
26-Feb-2018 19:30:34 26-Feb-2018 19:20:34   26-Feb-2018 19:10:34

Add a numeric array to a datetime array. MATLAB® treats each value in the numericarray as a number of exact, 24-hour days.
t2 = t1 + [1:3]
t2 = 1x3 datetime array
27-Feb-2018 19:50:34 28-Feb-2018 19:50:34   01-Mar-2018 19:50:34

t1 =
datetime(2014,3,8,0,0,0,'TimeZone','America/New
_York')t1 = *datetime*
08-Mar-2014 00:00:00

Find future points in time by adding a sequence of fixed-length (24-hour) days.
t2 = t1 + days(0:2)
t2 = *1x3 datetime array*
08-Mar-2014 00:00:00 09-Mar-2014 00:00:00   10-Mar-2014 01:00:00

Add a number of calendar days to t1.
t3 = t1 +
caldays(0:
2) t3 = *1x3*
datetime

array

08-Mar-2014 00:00:00 09-Mar-2014 00:00:00   10-Mar-2014 00:00:00

Add a number of calendar months to January 31, 2014.

t1 =
datetime(2014,1,31)
t1 = *datetime* 31-
Jan-2014

t2 = t1 +
calmonths(1
:4)t2 = *1x4*
datetime
array

28-Feb-2014 31-Mar-2014   30-Apr-2014   31-May-2014

dt = caldiff(t2,'days')
dt = *1x3 calendarDuration array*
31d 30d   31d

t2 = datetime(2014,1,31) + calmonths(3) +
caldays(30)t2 = *datetime*
30-May-2014

d1 = calyears(1) + calmonths(2) +
caldays(20)d1 = *calendarDuration*
1y 2mo 20d

d2 = calmonths(11) +
caldays(23)d2 =
calendarDuration
11mo 23d

d = d1 + d2d = ***calendarDuration***
2y 1mo 43d

t1 =
datetime('
today')t1
= ***datetime***
26-Feb-2018

t2 = t1 + calmonths(0:2)
+ caldays(4)t2 = ***1x3***
datetime array
02-Mar-2018 30-Mar-2018   30-Apr-2018

dt = between(t1,t2)
dt = ***1x3 calendarDuration array***
4d 1mo 4d   2mo 4d

```
>> calendar(date)
              Mar 2019
      S     M    Tu     W    Th     F     S
      0     0     0     0     0     1     2
      3     4     5     6     7     8     9
     10    11    12    13    14    15    16
     17    18    19    20    21    22    23
```

```
>> eomday(1976,2)ans =
   29
```

```
>> weekday('21-dec-1994')
```

```
>>t=nowt =
 7.3750e+05
>
d
a
t
e
s
t
r
(
t
)

a
n
s

=
 14-Mar-2019 21:52:55
```

```
>> date
```

ans =

14-Mar-2019

tic Start a stopwatch timer.
tic and TOC functions work together to measure elapsed time.tic, by itself, saves the current time that TOC uses later to measure the time elapsed between the two.

TSTART = tic saves the time to an output argument, TSTART. The numeric value of TSTART is only useful as an input argument for a subsequent call to TOC.

toc Read the stopwatch timer.
TIC and toc functions work together to measure elapsed time.toc, by itself, displays the elapsed time, in seconds, since the most recent execution of the TIC command.

T = toc; saves the elapsed time in T as a double scalar.

toc(TSTART) measures the time elapsed since the TIC command that generated TSTART.

```
n = 1000;
tic;
sum = 0;
for i=1:n tstart = tic;
sum = sum + i;
 telapsed = toc(tstart);

% min
Time = min(telapsed,minTime);end
disp(telapsed)
```

cputime CPU time in seconds.

cputime returns the CPU time in seconds that has been used by the MATLABprocess since MATLAB started.

```
>> cputime
ans =
  75.3953
>> cputime
ans =
  75.6137
```

REFERENCES

1. Stephen J Chapman, " Programming in MATLAB for Engineers", Brooks, 2002
2. Duane Hanselman ,Bruce LittleField, "Mastering MATLAB 7" , Pearson Education Inc, 2005
3. William J.Palm, "Introduction to MATLAB 6.0 for Engineers", Mc Graw Hill & Co, 2001
4. M.Herniter, "Programming in MATLAB", Thomson Learning, 2001

QUESTION BANK

	PART - A	CO
1	List out the relational operators that are used in matlab.	2
2	Differentiate conditional and unconditional looping.	2
3	Write the syntax for **for loop.**	2
4	Mention the special matlab statements that control the program execution sequence	2
5	Write a matlab program, using **while loop**, for finding the squares of integers less than 10.	2
6	What is meant by nested if structure.	2
7	Write a matlab program to find the average of any 10 numbers using for statement.	2
8	Develop a matlab code to add given two row vectors [1 2 3] & [4 5]. Illustrate using try-catch structure	2
9	Differentiate between looping and branching structures.	2
10	List out the logical operators that are used in matlab format.	2
11	List few low level file operations with example	2
12	What is the syntax to create ftp object?	2

PART - B

1	Design the matlab code for performing the basic arithmetic operations for matrix manipulation by using the switch-case structures.	2
2	Develop a matlab code for finding the mean, standard deviation and variance for a given n numbers.	2
3	Write a matlab code to sort the given numbers in descending order.	2
4	Develop a matlab coding for solving simultaneous equations consisting of three variables and there by finding the variables.	2
5	Explain in detail the various operators used in matlab with the expressions in both algebraic and matlab form. Also give the results of the expressions.	2

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – III – Software Tools for Engineering Applications – SECA4001

UNIT 3 PLOTS IN MATLAB & GUI

Basic 2D plots, Parametric & Implicit plots, subplot, LOG, LOG-LOG, SEMILOG-POLAR-COMET, exporting figures, HOLD, STEM, BAR, HIST, Interactive plotting, , axis labels, and annotations, Specifying line styles and colors, 3D plots – Mesh Surface- Contour –Plots with special graphics, View command, Plotting file data, Plotting from a function, GUI – Event based user interfaces, Matlab GUIDE, call back function, GUI controls, Example programs.

Two Dimensional Plots(2D plots):

The common 2-D plot commands includes the following mentioned below.

- plot - Basic line plots
- plotyy - Plot multiple plots using two Y axes, one on each side
- polar - Polar coordinate plot
- loglog, semilogx, semilogy - logarithmic plots
- errorbar - Plot error bars along a curve, not necessarily equal.
- bar, barh - Bar plots, vertical or horizontal
- pie - Pie charts
- hist - Histograms
- contour - Contour plots.
- comet – Animated plot
- feather - Vectors distributed along a line.
- quiver - Vectors distributed over a 2-D field, e.g. wind directon & strength.

Basic 2D plots:

"plot()" gives a linear plot. plot(x,y) plots vector y versus vector x. If x or y is a matrix, then thevector is plotted versus the rows/columns of the matrix.

plot(y) plots the columns of y versus their index.

If y is complex, plot(y) is equivalent to plot(real(y),imag(y)).Mostly the imaginary part is normallyignored.

```
x=1:0.5:10;
y=1:0.5:10;
plot(x,y);
xlabel('x
axis
name');
ylabel('y
axis
```

name');
title('Plot
name');

The below fig 3.1 shows the basic plot.

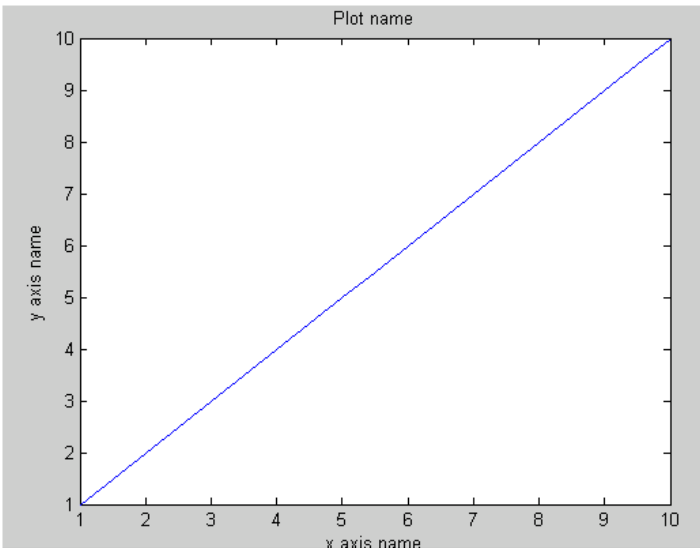


Fig 3.1 Basic plotThe below figure 3.2 gives the

various details of the plot

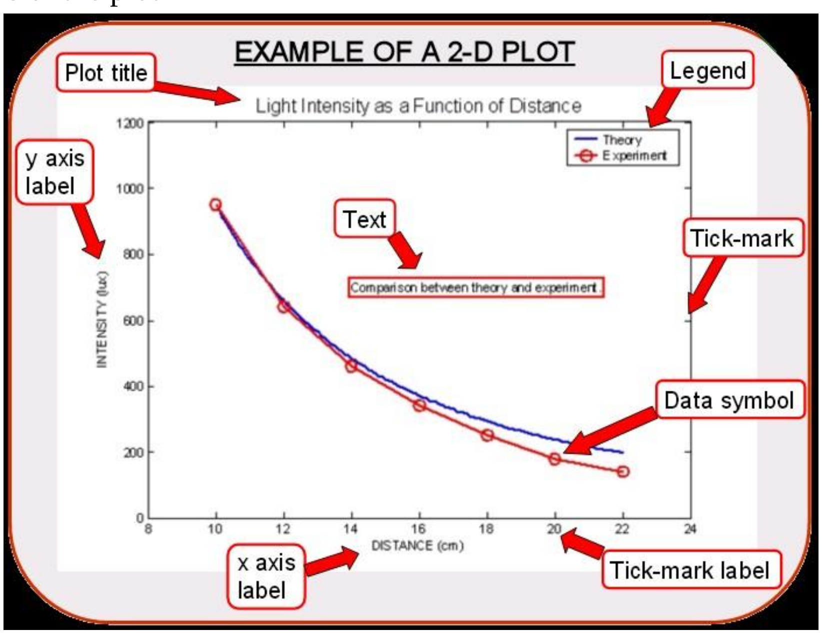


Fig 3.2 Plot with various parameters.

Line styles, Markers and Colours:

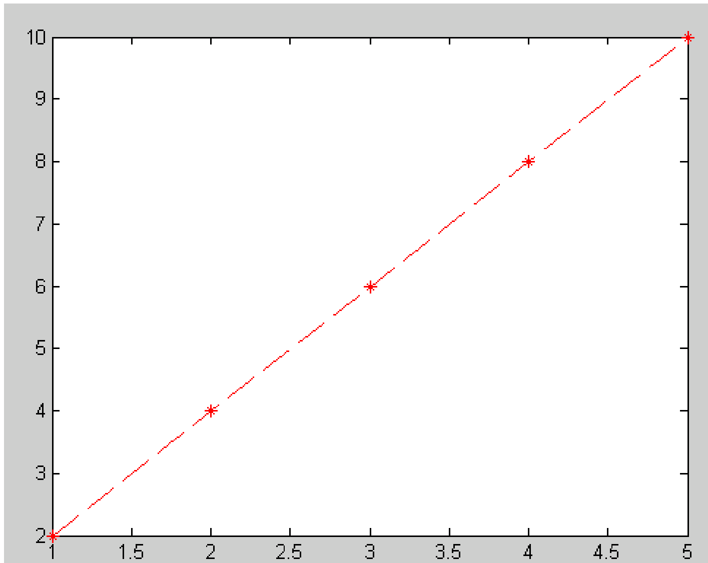
Various line types, plot symbols and colors may be obtained with plot(x,y,s) where s is acharacter string made from one element from any or all the following 3 columns given below,

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

Eg. plot(x,y,'c+:') plots a cyan dotted line with a plus at each data point whereas plot(x,y,'bd') plots bluediamond at each data point but does not draw any line.

```
x=1:5;
y=1:5;
z=x+
y;
plot(z
,'r*--
```

')



The fig 3.3 shows the plot for the above matlab program.

Fig 3.3 Plot command with various styles.

Legend:

To plot multiple dependent vectors on the same plot and to distinguish them from each other via a legend, the syntax is very similar to the axis labeling above. It is also possible to set colors for the different vectors and to change the location of the legend on the figure.

The below example matlab program and fig 3.4 shows the details of the legend

```
command.clear all;
X = [3 9 27]; % dependent vectors of
interestY = [10 8 6];
Z = [4 4 4];
t = [1 2 3]; % independent vector
hold on % allow all vectors to be plotted in same
figureplot(t, X, 'blue', t, Y, 'red', t, Z, 'green');
 title('Plot of Distance over Time')
% titleylabel('Distance (m)') %
label for y axis xlabel('Time (s)')
% label for x axis legend('Trial 1',
'Trial 2', 'Trial 3');
legend('Location','NorthWest') % move legend to upper left
```

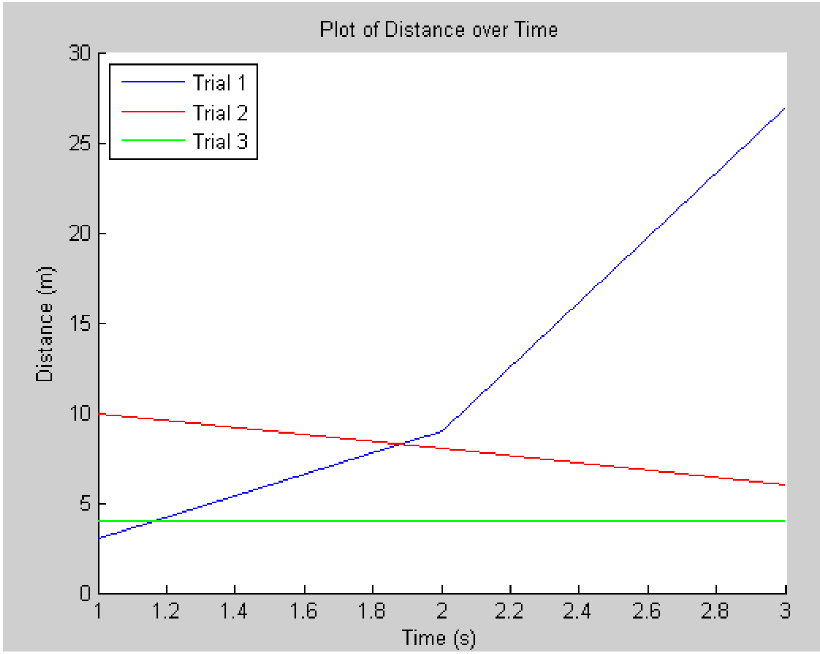


Fig 3.4 Plot command with legend function

subplot:

Another function that is very useful with any type of plot is subplot, which creates a matrix ofplots in the current Figure Window. Three arguments are passed to it in the form subplot(r,c,n); where

r and c are the dimensions of the matrix and n is the number of the particular plot within this matrix. The plots are numbered row wise starting in the upper left corner. In many cases, it is useful to createa subplot in a for loop so the loop variable can iterate through the integers 1 through n. When the subplot function is called in a loop, the first two arguments will always be the same since they give the dimensions of the matrix. The third argument will iterate through the numbers assigned to the elements of the matrix.

When the subplot function is called, it makes that element the active plot; then, any plot function can be used complete with axis labeling, titles, and such within that element.

Plot Types:

Besides **plot** and **subplot**, there are other plot types such as **histograms, stem** plots, **area** plots and **pie** charts, as well as other functions that customize graphs. The functions

bar, barh, area, and **stem** essentially display the same data as the plot function, but in different forms. The **bar**() function draws a bar chart , **barh()** draws a horizontal bar chart, **area** draws the plot as a continuous curve and fills in under the curve that is created, and **stem** draws a stem plot or discrete plot.

For example, the following script creates a figure window that uses a 2 × 2 subplot to demonstratethese four plot types using the same x and y points (see Figure 3.5).

```
x = 1:6;
y = [33 11 5 9 22 30];
subplot(2,2,1);
bar(x,y);
title('bar');
subplot(2,2,2);
barh(x,y);
title('barh');
subplot(2,2,3);
area(x,y);
title('area');
subplot(2,2,4);
stem(x,y);
title('stem');
```

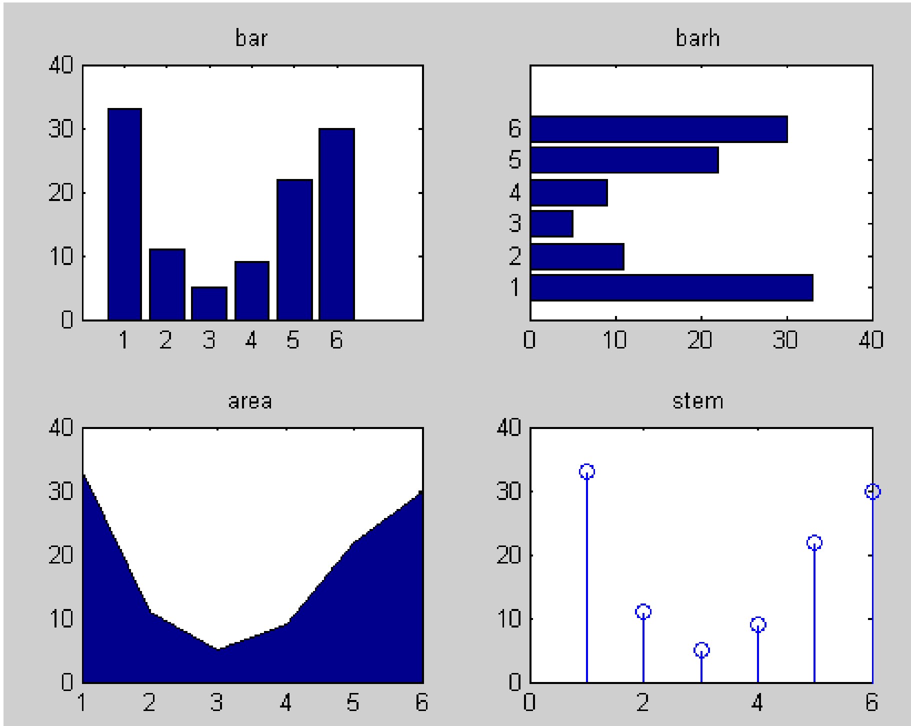


Fig 3.5 Types of plot using subplot command

LOGLOG( ):

It creates a plot using a logarithmic scale for both the x-axis and the y-axisloglog(x,y,'LineWidth',2). Fig 3.6 shows the simulated output.

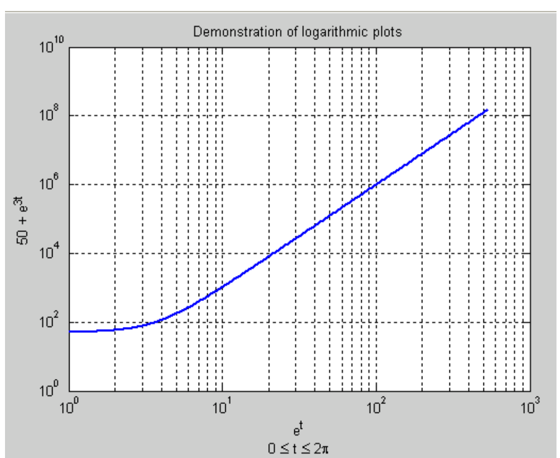


Fig 3.6 Example of loglog() command

SEMILOG():

(A) SEMILOGX():

The syntax is semilogx(x, y) and the output is shown in fig 3.7

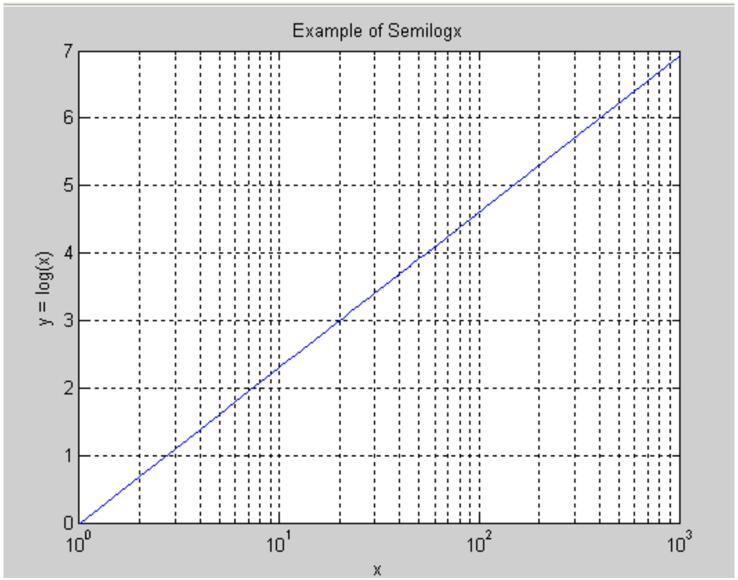


Fig 3.7 Example of semilogx() command

(B) SEMILOGY():

The syntax is semilogy(x, y) and the output is shown in fig 3.8

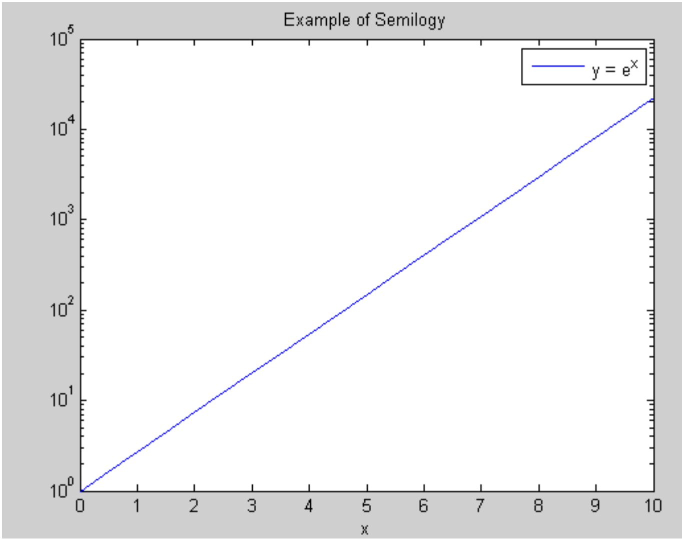


Fig 3.8 Example of semilogy() command

POLAR():

The polar function creates polar plots from angle and magnitude data. It takes the forms

polar(theta,rho), where theta corresponds to the angle (in radians) and rho corresponds to the magnitude. The variables theta and rho must be identically sized vectors. An example is given below.

t = 0 : 2*pi/100 :
2*pi;r = 1 -
sin(t);
polar(t, r)

The output figure 3.9 obtained is shown below

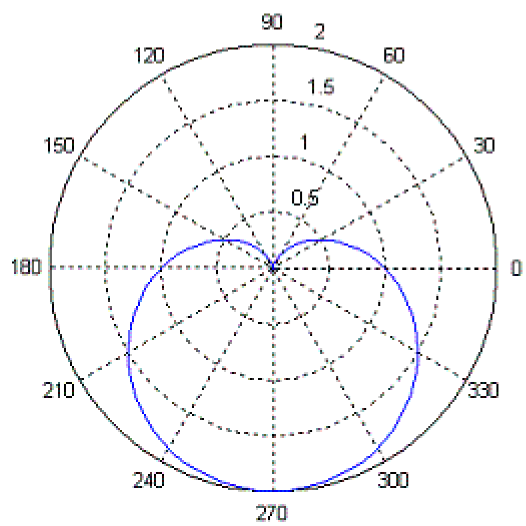


Fig 3.9 Example of polar plot

COMET():

comet(a) displays an animated comet plot of the vector a.
comet(a, b) displays an animated comet plot of vector b vs. a.
comet(a, b, p) uses a comet of length p*length(b). Default is p = 0.10.

Controlling Axes:

The scaling and appearance of plot axis can be controlled with the **axis** function. To set scalingfor the x and y axes on the current 2-D plot, use the below given command axis([xmin xmax ymin ymax])

To scale the axes on 3-D plot, use the command for x,y,z min &

maxAlso,

axis('auto') - returns the axis scaling to its default where the best axis limits are computed automatically
axis('square') - makes the current axis box square in size, otherwise a circle will look like an oval;

axis('off ') - turns off the axes
axes axis('on') - turns on axis labeling and tic marks.

axis(manual) - freezes the scaling at the current limits, so that if hold is turned on, subsequent plots willuse the same limits.

axis(tight) - sets the axis limits to the range of the data.

axis(fill) - sets the axis limits and plot box aspect ratio so that the axis fills the position rectangle. Thisoption only has an effect if plot box aspect ratio mode or data aspect ratio mode are manual.

axis(equal) - sets the aspect ratio so that equal tick mark increments on the x-,y- and z-axis are equal insize.

axis(normal) - restores the current axis box to full size and removes any restrictions on the scaling ofthe units. This undoes the effects of axis square and axis equal.

Labeling:

MATLAB also allows labelling of axes. ie x,y axes and title and is shown in fig 3.1

- **xlabel()** function allows to label the x

  axis.xlabel('string')

- **ylabel()** function allows labelling of

  y axisylabel('string')

- **title()**– function allows giving title to our plottitle(‘string’)

fplot():

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

The syntax is given below

- fplot(fun, limits) – A function fun is plotted in between the limits specified
- fplot(fun, limits, linespace) – allows plotting fun with line specification
- fplot(fun, limits, tol) -allows plotting with relative error tolerance ‘tol’.If not specifieddefault tolerance will be 2e-3 ie .2% accuracy.
- fplot(fun, limits, tol, linespace) – allows plotting with relative tolerance and linespecification

The fig 3.10 below shows for the function

fplot('x.^2' , [0,50])

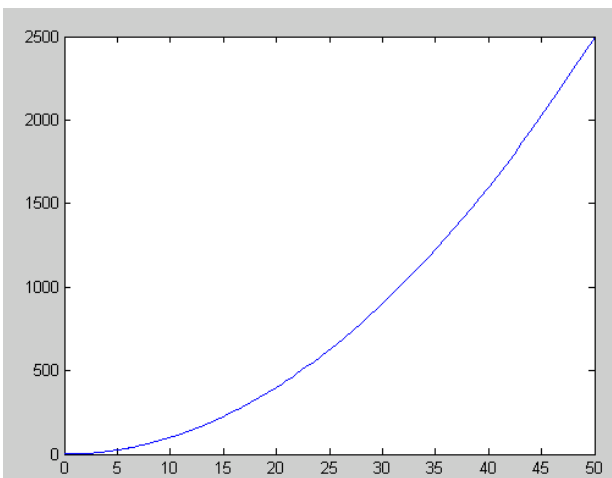


Fig 3.10 Example of fplot().

ezplot():

Easy to use function plotter

ezplot(fun) plots the expression fun(x) over the default domain $-2\pi < x < 2\pi$.

ezplot(f,[min,max]) plots $f = f(x)$ over the domain: min < x <

max.For implicitly defined functions, $f = f(x,y)$:

ezplot(f) plots $f(x,y) = 0$ over the default domain -2π < x < 2π, -2 π < y < 2 π.

ezplot(f,[xmin,xmax,ymin,ymax]) plots $f(x$,y) =0 over xmin < x < xmax and ymin < y < ymax.

ezplot(f,[min,max]) plots $f(x,y) = 0$ over min < x < max and min < y < max.

ezplot(x,y) plots the parametrically defined planar curve $x = x(t)$ and $y = y(t)$ over the default domain0< t < 2 π.

ezplot(x,y,[tmin,tmax]) plots $x = x(t)$ and $y = y(t)$ over tmin < t < tmax.

ezplot(...,figure) plots the given function over the specified domain in the figure window identified bythe handle figure.

ezpolar():

Easy to use polar coordinate plotter

ezpolar(f) plots the polar curve $rho = f(theta)$ over the default domain 0 < thetaπ< 2
ezpolar(f,[a,b]) plots f for a < $theta$ < b.

Example Program: To Plot the function 1+cos(t) over the domain

[0,2π].figure

ezpolar('1+cos(t)')

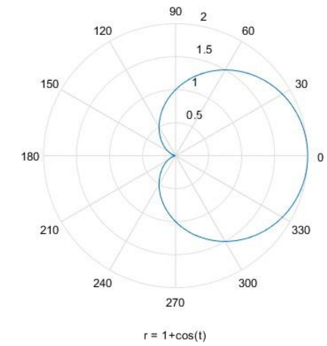


Fig 3.11 Example of ezpolar()

polyval():

Polynomial evaluation in matlab.

y = polyval(p,x) returns the value of a polynomial(p) of degree n evaluated at x. The input argument p isa vector of length N+1 whose elements are the coefficients in descending powers of the polynomial to be evaluated.

Y = P(1)*X^N + P(2)*X^(N-1) + ... + P(N)*X + P(N+1)

x can be a matrix or a vector. In either case, polyval evaluates p at each element of x.

The polynomial coefficients in p can be calculated for different purposes by functions like polyint,polyder, and polyfit, but we can specify any vector for the coefficients.

The polynomial $p(x) = 3x^2 + 2x + 1$ is evaluated at x = 5, 7, and 9

withp = [3 2 1];
polyval(p,[5 7 9])
and the output is 86 162
262

HOLD:

Retain current plot when adding new

plotsSyntax

hold on retains plots in the current axes so that new plots added to the axes do not delete existing plots.

hold off sets the hold state to off so that new plots added to the axes clear existing plots and reset allaxes properties.

hold all is the same as hold on. This syntax will be removed in a future release. Use **hold on** instead.hold toggles the hold state between on and off.

hold(ax,___) sets the hold state for the axes specified by ax instead of the current axes.

Example:

```
x =
linspace
(-pi,pi);
y1 =
sin(x);
plot(x,y
1)

hold on
y2 = cos(x);plot(x,y2) hold off
```

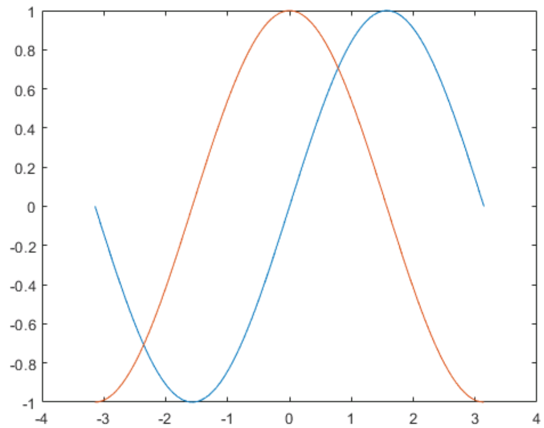


Fig 3.12 Example of hold command in plotting graphs

STEM():

It is used for discrete plot.

stem(y) plots the data sequence y as stems from the x axis terminated with circles for the data value. If yis a matrix then each column is plotted as a separate series.

stem(x,y) plots the data sequence y at the values

specified in x.The example for stem command is

shown in fig. 3.5

BAR(): Bar Graph

The bar(x,y) draws the columns of the m-by-n matrix y as m groups of n vertical bars. The vector x must not have duplicate values. Similarly bar(y) uses the default value of x=1:m. for vector inputs, bar(x,y) or bar(y) draws length(y) bars. Barh() is used for horizontal bar graphs. Fig.3.5 showsthe output of bar() and barh() command

HIST():

hist() function is used to plot the histogram.

n=hist(y) - bins the elements in vector y into 10 equally spaced containers and returns the number ofelements in each container as a row vector.

n=hist(y,x) - where x is a vector, returns the distribution of y along length (x) bins with centers specifiedby x.

n=hist(y,nbins) - where n bins is scalar and uses it as number of bins

x=rand(1000,1); % rand function generates nxn matrix ; rand(m,n) generates mxn square matrixfix(x); % used to round it off to 0
hist(x,10) % returns the distribution of x in 10 length bin

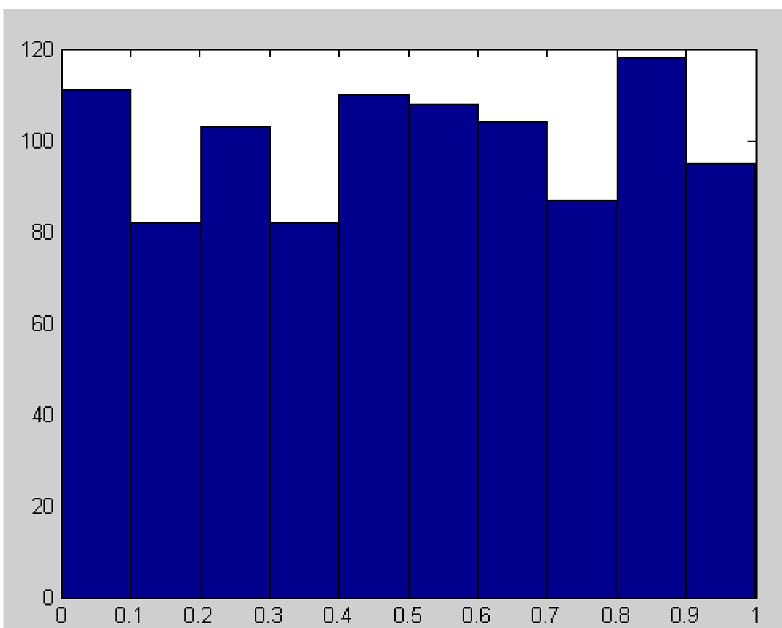


Fig 3.13 Histogram Plot

Polyfit :

Polynomial curve fitting

p = polyfit(x,y,n) finds the coefficients of a polynomial p(x) of degree n that fits the data, p(x(i)) to y(i), in a least squares sense. The result p is a row vector of length n+1 containing the polynomial coefficientsin descending powers

$$p(x) = p_1x^n + p_2x^{n-1} + \ldots + p_nx + p_{n+1}$$

[p,S] = polyfit(x,y,n) returns the polynomial coefficients p and a structure S for use with polyval to obtain error estimates or predictions. If the errors in the data y are independent normal with constantvariance, polyval produces error bounds that contain at least 50% of the predictions.

Polyfit is a Matlab function that computes a least squares polynomial for a given set of data.
Polyfit generates the coefficients of the polynomial, which can be used to model a curve to fit the data.

Polyval evaluates a polynomial for a given set of x values. So, Polyval generates a curve to fitthe data based on the coefficients found using polyfit.

3D PLOTS

- mesh(X,Y,Z) creates a mesh plot, which is a three-dimensional surface that has solid edge colors and no face colors. The function plots the values in matrix Z as heights above a grid in the *x*-*y* plane defined by X and Y. The edge colors vary according to the heights specified by Z.
- mesh(Z) creates a mesh plot and uses the column and row indices of the elements in Z as the *x*- and *y*-coordinates
- mesh(Z,C) additionally specifies the color

of the edges.EXAMPLE

1. CREATE MESH PLOT

Create three matrices of the same size. Then plot them as a mesh plot. The plot uses Z for both heightand color

[X,Y] = meshgrid(-8:.5:8); R = sqrt(X.^2 + Y.^2) + eps;Z = sin(R)./R;
mesh(X,Y,Z)

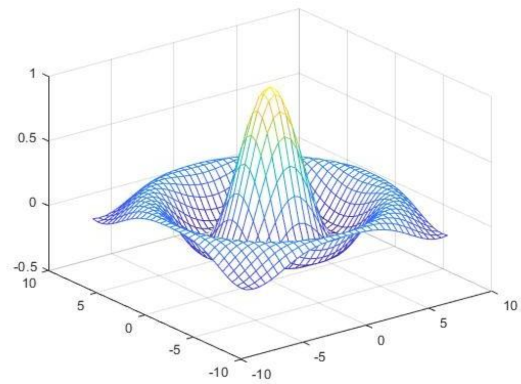


Fig 3.14 Mesh Plot

2. SPECIFY COLORMAP COLORS FOR MESH PLOT

Specify the colors for a mesh plot by including a fourth matrix input, C. The mesh plot uses Z for height and C for color. Specify the colors using a *colormap*, which uses single numbers to stand for colors on a spectrum. When you use a colormap, C is the same size as Z. Add a color bar to the graph to show how the data values in C correspond to the colors in the colormap.

```
[X,Y] = meshgrid(-8:.5:8); R = sqrt(X.^2 + Y.^2) + eps;Z = sin(R)./R;
C = X.*Y;
mesh(X,Y,Z,C)
colorbar
```

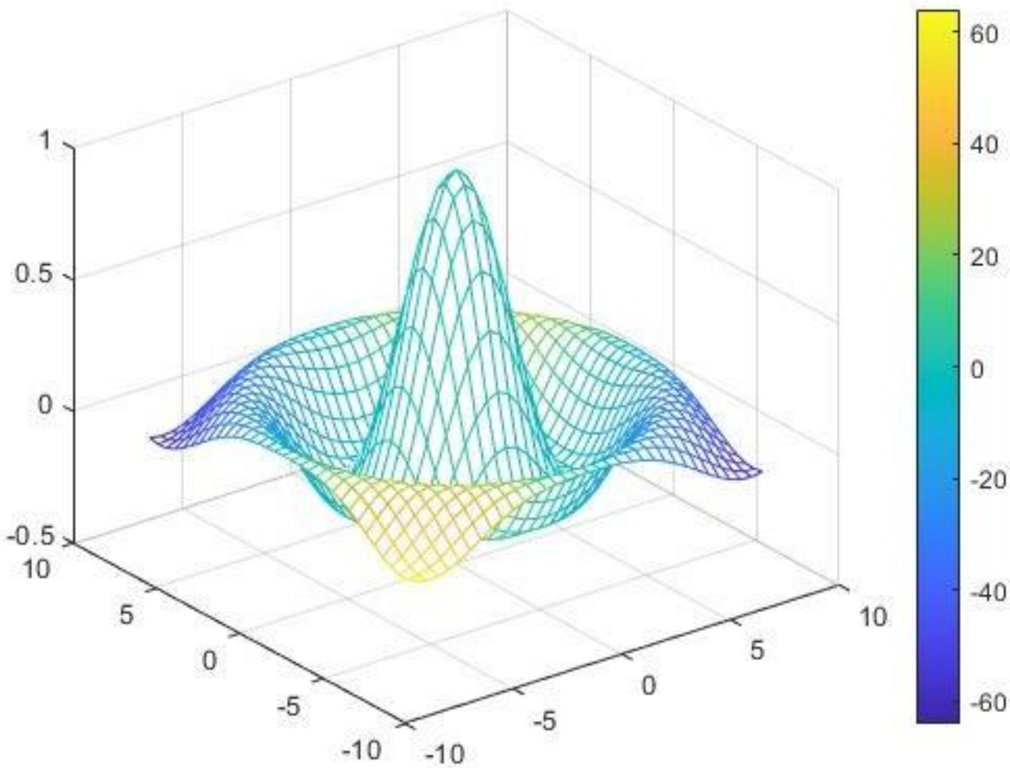


Fig 3.15 Mesh Plot with Colour bar

3. SPECIFY TRUE COLORS FOR MESH PLOT

Specify the colors for a mesh plot by including a fourth matrix input, CO. The mesh plot uses Z for height and CO for color. Specify the colors using *truecolor*, which uses triplets of numbers to stand for all possible colors. When you use truecolor, if Z is m-by-n, then CO is m-by-n-by-3. The first page of the array indicates the red component for each color, the second page indicates the green component, and the third page indicates the blue component.

```
[X,Y,Z] = peaks(25);
CO(:,:,1) = zeros(25); % red
CO(:,:,2) = ones(25).*linspace(0.5,0.6,25); % green
CO(:,:,3) =
ones(25).*linspace(0,1,25); % blue
mesh(X,Y,Z,CO)
```

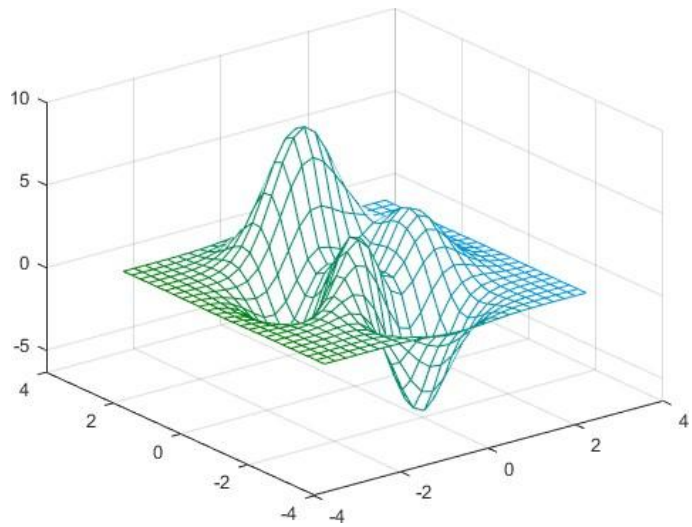


Fig 3.16 Mesh Plot

CONTOUR

contour3(Z) creates a 3-D contour plot containing the isolines of matrix Z, where Z contains height values on the *x*-*y* plane. MATLAB® automatically selects the contour lines to display. The column and row indices of Z are the *x* and *y* coordinates in the plane, respectively.

contour3(X,Y,Z) specifies the *x* and *y* coordinates for the values in Z

contour3(_,levels) specifies the contour lines to display as the last argument in any of the previous syntaxes. Specify levels as a scalar value n to display the contour lines at n automatically chosen levels (heights). To draw the contour lines at specific heights, specify levels as a vector of monotonically increasing values. To draw the contours at one height (k), specify levels as a two-element row vector [k k].

contour3(_,LineSpec) specifies the style and color of the

contour lines.

EXAMPLES

1. CONTOURS OF SPACE

Define Z as a function of X and Y. In this case, call the sphere function to create X, Y, and Z. Then plot the contours of Z.

```
[X,Y,Z] = sphere(50);
contour3(X,Y,Z);
```

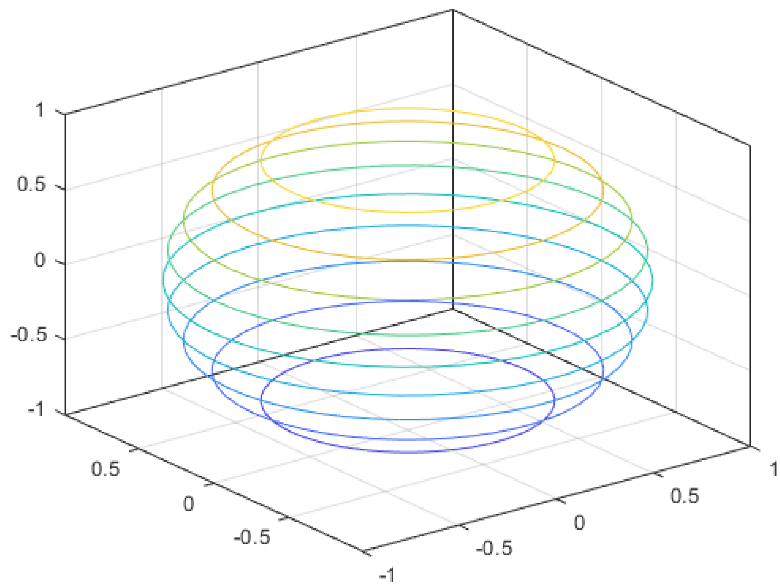


Fig 3.17 Contour plot

2. CONTOURS AT FIFTY LEVELS

Define Z as a function of two variables, X and Y. Then plot the contours of Z. In this case, letMATLAB® choose the contours and the limits for the *x*- and *y*-axes.

```
[X,Y] =
meshgrid(-
5:0.25:5);Z =
X.^2 + Y.^2;
contour3(Z)
```

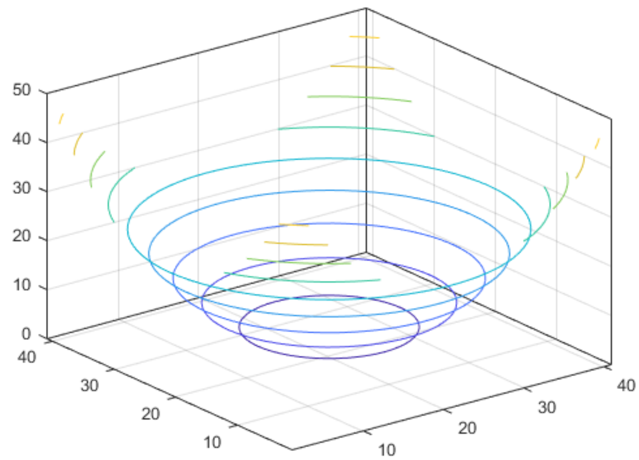


Fig 3.18 Contour plot

Now specify 50 contour levels, and display the results within the *x* and *y* limits used to calculate Z.contour3(X,Y,Z,50)

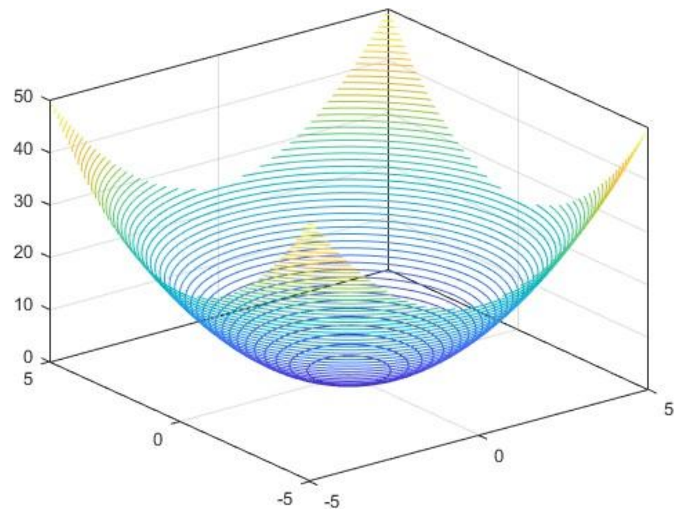


Fig 3.19 Contour plot

3. CONTOURS AT SPECIFIC LEVELS WITH LABELS

Define Z as a function of two variables, X and Y. Then plot the contours at Z = [-.2 -.1 .1 .2]. Show the contour labels by setting the ShowText property to 'on'

```
[X,Y] = meshgrid(-2:0.25:2);
Z = X.*exp(-X.^2-Y.^2);
contour3(X,Y,Z,[-.2 -.1 .1 .2],'ShowText','on')
```

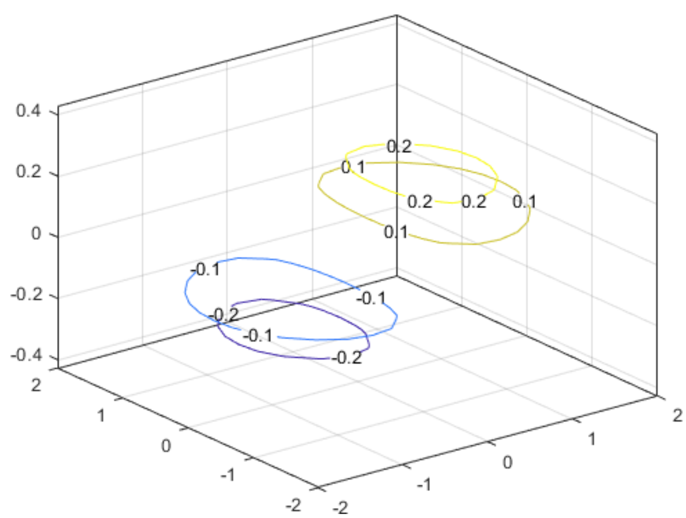


Fig 3.20 Contour plot with coordinates

POLYFIT

Polynomial curve fitting

- p = polyfit(x,y,n) returns the coefficients for a polynomial p(x) of degree n that is a best fit (in a least-squares sense) for the data in y. The coefficients in p are in descending powers, and the length of p is n+1

$p(x)=p_1x^n+p_2x^{n-1}+...+p_nx+p_{n+1}$.

- [p,S] = polyfit(x,y,n) also returns a structure S that can be used as an input to polyval to obtain error estimates.
- [p,S,mu] = polyfit(x,y,n) also returns mu, which is a two-element vector with centering and scaling values. mu(1) is mean(x), and mu(2) is std(x). Using these values, polyfit centers x atzero and scales it to have unit standard deviation,

  $\hat{x}=\frac{x-\bar{x}}{\sigma}$

  This centering and scaling transformation improves the numerical properties of both the polynomial and the fitting algorithm.

EXAMPLE

1. FIT POLYNOMIAL TO TRIGONOMETRIC

FUNCTIONFit Polynomial to Trigonometric Function
x = linspace(0,1,5);
y = 1./(1+x);

Fit a polynomial of degree 4 to the 5 points. In general, for n points, you can fit a polynomial of degreen-1 to exactly pass through the points.
p = polyfit(x,y,4);

Evaluate the original function and the polynomial fit on a finer grid of points between 0 and 2.x1 = linspace(0,2);
y1 = 1./(1+x1);
f1 = polyval(p,x1);

Plot the function values and the polynomial fit in the wider interval [0,2], with the points used to obtain the polynomial fit highlighted as circles. The polynomial fit is good in the original [0,1] interval, but quickly diverges from the fitted function outside of that interval.

figure plot(x,y,'o') hold on plot(x1,y1)
plot(x1,f1,'r--')

legend('y','y1','f1')

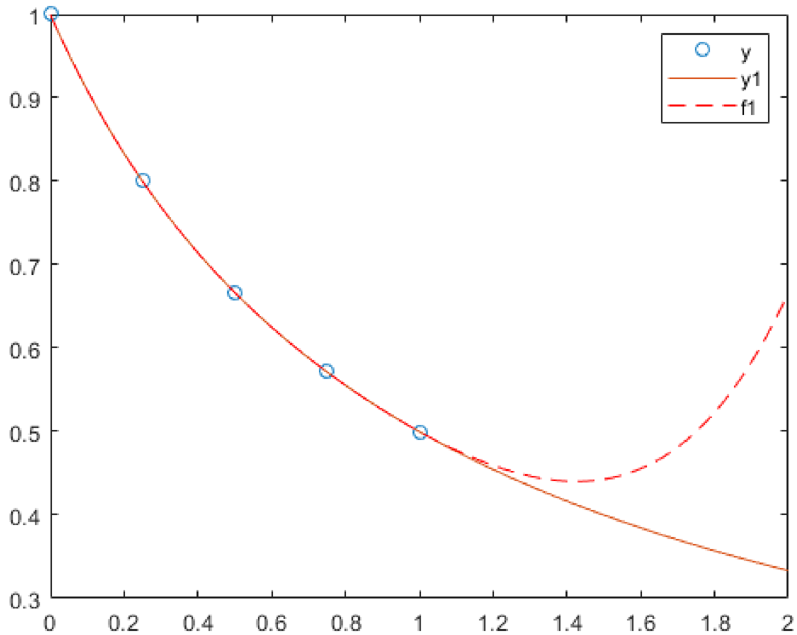


Fig 3.21 Polyfit

2. FIT POLYNOMIAL TO ERROR FUNCTION

First generate a vector of x points, equally spaced in the interval [0,2.5], and then evaluate erf(x) at thosepoints.

x = (0:0.1:2.5)';
y = erf(x);

Determine the coefficients of the approximating polynomial of degree

6.p = polyfit(x,y,6)
p = *1×7*

0.0084 -0.0983 0.4217 -0.7435 0.1471 1.1064 0.0004

To see how good the fit is, evaluate the polynomial at the data points and generate a table showing thedata, fit, and error.

f = polyval(p,x);
T = table(x,y,f,y-f,'VariableNames',{'X','Y','Fit','FitError'})T=*26×4 table*

X	Y	Fit	FitError
0	0	0.00044117	-0.00044117
0.1	0.11246	0.11185	0.00060836
0.2	0.2227	0.22231	0.00039189
0.3	0.32863	0.32872	-9.7429e-05
0.4	0.42839	0.4288	-0.00040661
0.5	0.5205	0.52093	-0.00042568
0.6	0.60386	0.60408	-0.00022824
0.7	0.6778	0.67775	4.6383e-05
0.8	0.7421	0.74183	0.00026992
0.9	0.79691	0.79654	0.00036515
1	0.8427	0.84238	0.0003164
1.1	0.88021	0.88005	0.00015948
1.2	0.91031	0.91035	-3.9919e-05
1.3	0.93401	0.93422	-0.000211
1.4	0.95229	0.95258	-0.00029933
1.5	0.96611	0.96639	-0.00028097
⋮			

In this interval, the interpolated values and the actual values agree fairly closely. Create a plot to showhow outside this interval, the extrapolated values quickly diverge from the actual data.

x1 = (0:0.1:5)';

```
y1 = erf(x1);
f1 = polyval(p,x1);figure
plot(x,y,'o') hold on
plot(x1,y1,'-')
plot(x1,f1,'r--')
axis([0 5 0 2])
hold off
```

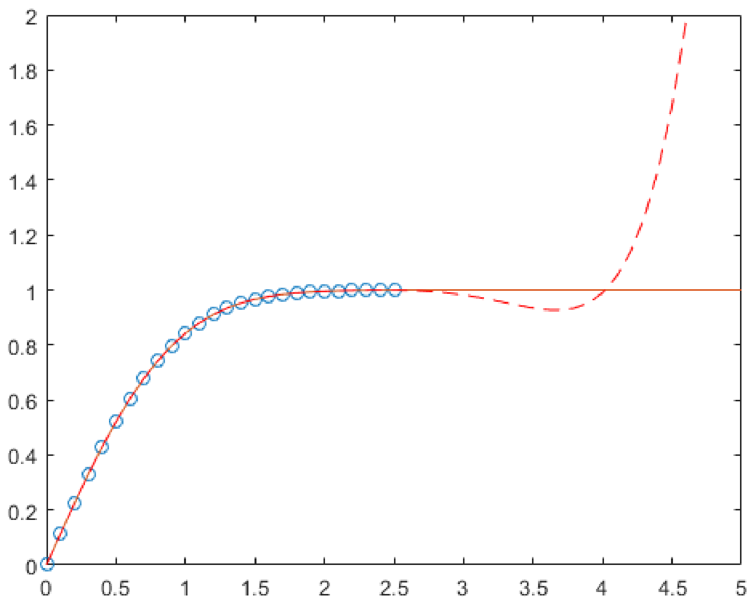


Fig 3.22 Plot for polyval

Text

- text(x,y,str)
- text(x,y,z,str)
- text(_,Name,Value)
- t = text(_)

Description

text(x,y,str) adds a text description to one or more data points in the current axes using the text specified by str. To add text to one point, specify x and y as scalars in data units. To add text to multiple points, specify x and y as vectors with equal length.

text(x,y,z,str) positions the text in 3-D coordinates.

text(__,Name,Value) specifies text object properties using one or more name-value pairs. For example, 'FontSize',14 sets the font size to 14 points. You can specify text properties with any of the input argument combinations in the previous syntaxes. If you specify the Position and String properties as name- value pairs, then you do not need to specify the x, y, z, and str inputs.

t = text(_) returns one or more text objects. Use t to modify properties of the text objects after they are created. For a list of properties and descriptions, see Text Properties. You can specify an output with any of the previous syntaxes.

```
x = 0:pi/20:2*pi;
y = sin(x);
plot(x,y)
text(pi,0,'\leftarrow sin(\pi)')
```

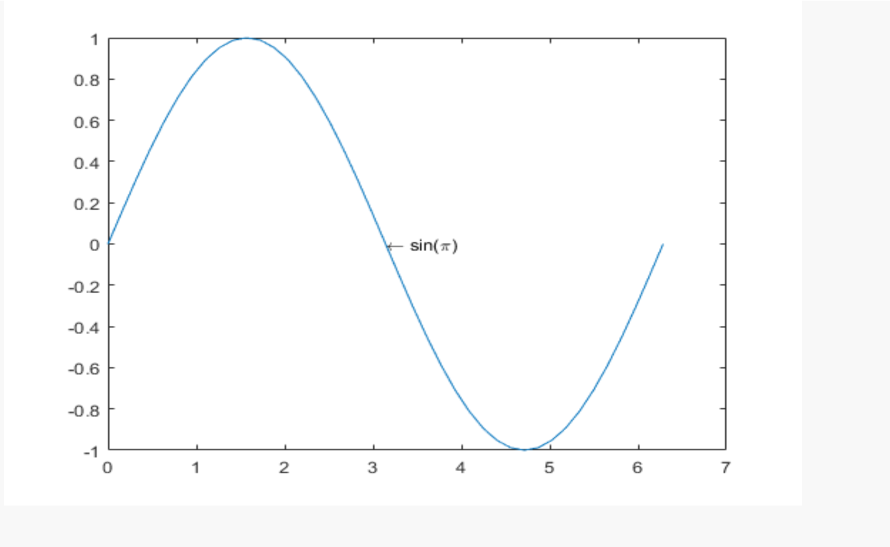


Fig 3.23 Text on plot

gtext

- Add text to figure using mouse
- gtext(str)
- gtext(str,Name ,Value)
- t =gtext( )

Description

gtext(str) inserts the text, str, at the location you select with the mouse. When you hover over the figure window, the pointer becomes a crosshair. gtext is waiting for you to select a location. Move the pointer to the location you want and either click the figure or press any key, except **Enter**.

gtext(str,Name,Value) specifies text properties using one or more name-value pair arguments. For example, 'FontSize',14 specifies a 14-point font.

t = gtext(__) returns an array of text objects created by gtext. Use t to modify properties of the text objects after they are created. For a list of properties and descriptions, see Text Properties. You can return an output argument using any of the arguments from the previous syntaxes.

plot(1:10)

gtext('MyPlot')

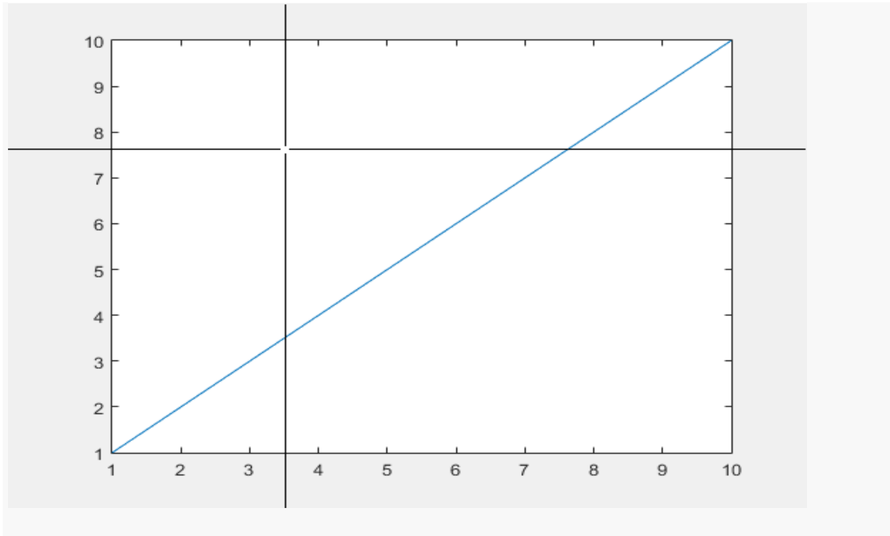


Fig 3.24 Cursor on plot for gtext

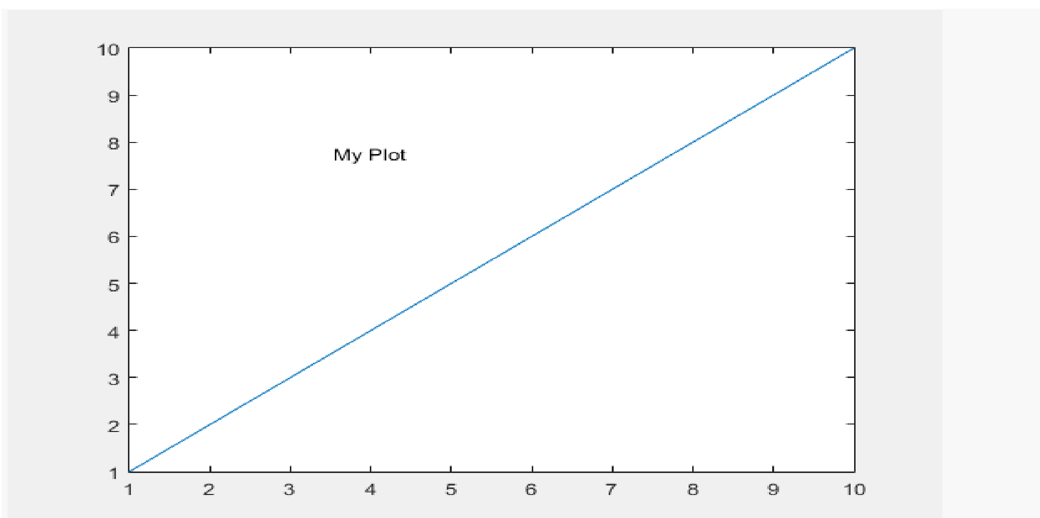


Fig 3.25 Text on the plot in the specified location using gtext

GRAPHICAL USER INTERFACE

A Graphical User Interface (GUI) is a pictorial interface to a program. A good GUI can make programs easier to use by providing them with a consistent appearance and with intuitive controls like pushbuttons, list boxes, sliders, menus etc. The GUI must be developed in an understandable and predictable manner. For example, when a mouse click occurs on a pushbutton, the GUI should initiate the action performed on the label of the button. Three principal elements to create a GUI are as follows:

Components: Each item on a MATLAB GUI (pushbuttons, labels, edit boxes etc.) is a graphical component. The types of components include graphical controls (pushbuttons, edit boxes, lists, sliders etc), static elements (frames and text strings), menus and axes. Graphical

functions uimenu and uicontextmenu. Axes, which are used to display graphical data are created by the function axes.

Figures: The components of a GUI must be arranged within a figure, which is a window on the computer screen. In the past, figures have been created automatically whenever one has plotted data. However, empty figures can be created with the function **figure** and can be used to hold any combination of components.

Calbacks: A mouse click or a key press is an event, and the MATLAB program must respond to each event if the program is to perform its function. For example, if a user clicks on a button, that event must cause the MATLAB code that implements the function of the button to be executed. The code executed in response to an event is known as a callback. There must be a callback to implement the function of each graphical component on the GUI.

controls and static controls are created by the function uicontrol, and menus are created by the

The following figure and table depicts the basic elements in GUI.

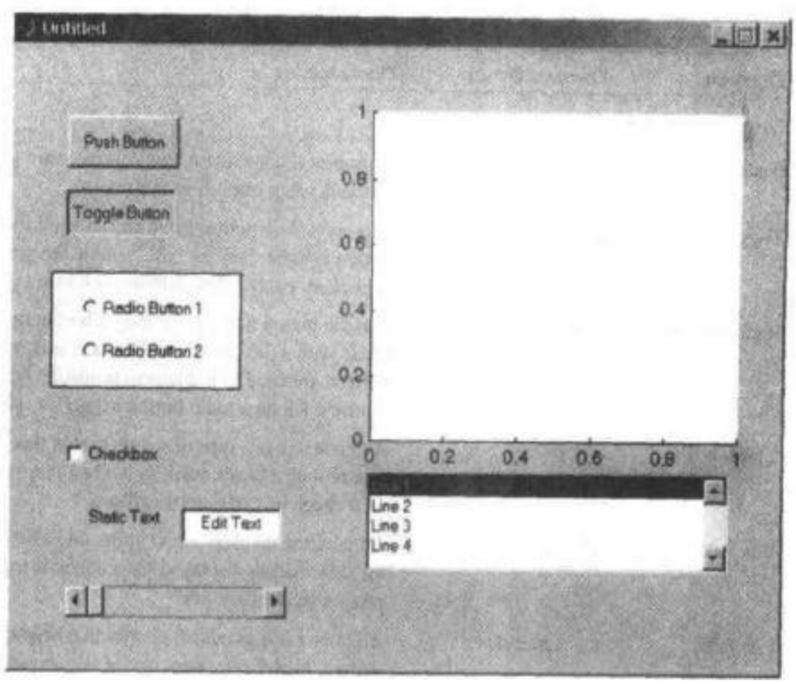


Fig 3.26 Panel Showing GUI

Element	Created By	Description
Graphical Controls		
Pushbutton	`uicontrol`	A graphical component that implements a pushbutton. It triggers a callback when clicked with a mouse.
Toggle button	`uicontrol`	A graphical component that implements a toggle button. A toggle button is either "on" or "off," and it changes state each time that it is clicked. Each mouse button click also triggers a callback.
Radio button	`uicontrol`	A radio button is a type of toggle button that appears as a small circle with a dot in the middle when it is "on." Groups of radio buttons are used to implement mutually exclusive choices. Each mouse click on a radio button triggers a callback.
Check box	`uicontrol`	A check box is a type of toggle button that appears as a small square with a check mark in it when it is "on." Each mouse click on a check box triggers a callback.
Edit box	`uicontrol`	An edit box displays a text string and allows the user to modify the information displayed. A callback is triggered when the user presses the `Enter` key.
List box	`uicontrol`	A list box is a graphical control that displays a series of text strings. A user can select one of the text strings by single- or double-clicking on it. A callback is triggered when the user selects a string.
Popup menus	`uicontrol`	A popup menu is a graphical control that displays a series of text strings in response to a mouse click. When the popup menu is not clicked on, only the currently selected string is visible.
Slider	`uicontrol`	A slider is a graphical control to adjust a value in a smooth, continuous fashion by dragging the control with a mouse. Each slider change triggers a callback.
Static Elements		
Frame	`uicontrol`	Creates a frame, which is a rectangular box within a figure. Frames are used to group sets of controls together. Frames never trigger callbacks.
Text field	`uicontrol`	Creates a label, which is a text string located at a point on the figure. Text fields never trigger callbacks.
Menus and Axes		
Menu items	`uimenu`	Creates a menu item. Menu items trigger a callback when a mouse button is released over them.
Context menus	`uicontextmenu`	Creates a context menu, which is a menu that appears over a graphical object when a user right-clicks the mouse on that object.
Axes	`axes`	Creates a new set of axes to display data on. Axes never trigger callbacks.

The basic steps involved in the creation of GUI are as follows;

- Decide what elements are required for the GUI and the function of each element.
- Use a MATLAB tool called GUIDE (GUI Development Environment) to lay out the components on a figure. The size of the figure and the alignment and spacing of components on the figure can be adjusted using built into guide.
- Use a MATLAB tool called the property inspector to give each component a name and to set the characteristics of each component, such as its color, the text it displays and so on

- Save the figure to a file. When the figure is saved, two files will be created on disk with the same name but different extents. The fig file contains the actual GUI and the M-file contains the code to load the figure and skeleton callbacks for each GUI element.
- Write a code to implement the behavior associated with each callback function

As an example, create a GUI that has a pushbutton and axes. When the pushbutton is clicked the image must be displayed on the axes.

To lay the components on the GUI, run the MATLAB function guide. When guide is executed, it creates the window as shown below.

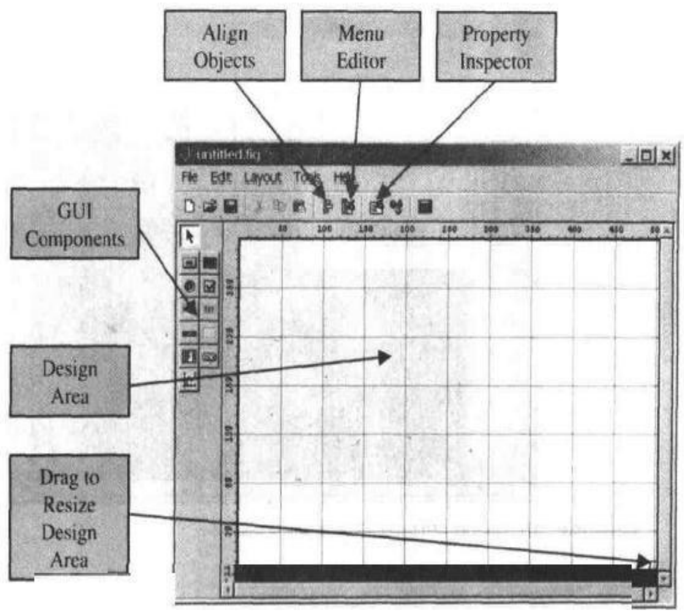


Fig 3.27 GUI Design Panel

If needed, the size of the layout area can be changed. It is done by dragging the small square on the lower right corner of the layout area until it has the desired size and shape. Click on “pushbutton” button in the list of GUI components and create the shape of the pushbutton in the layout area. Similarly place the axes above the pushbutton.

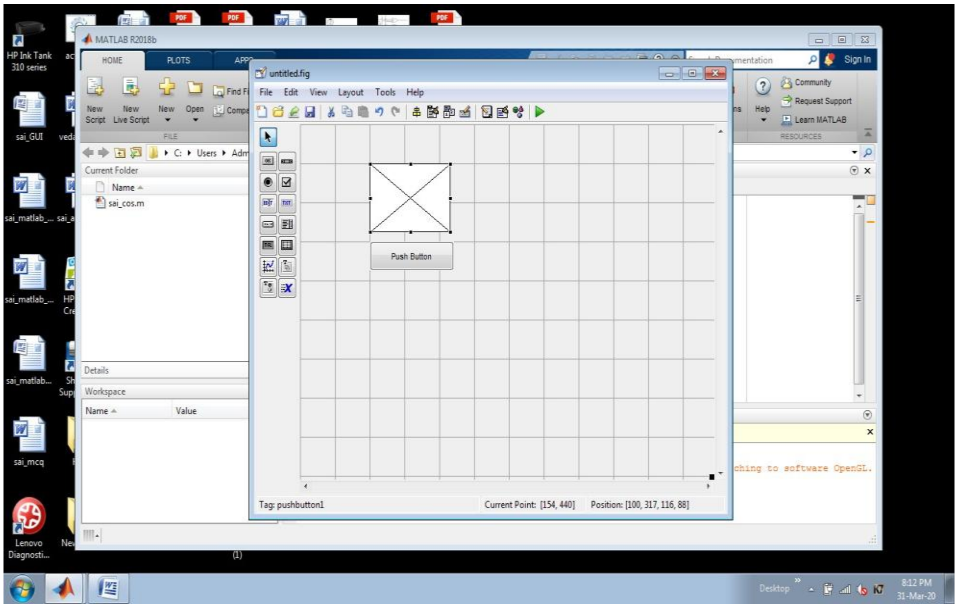


Fig 3.28 GUI Design

The next step is to go to property inspector and change the string and tag of pushbutton as input

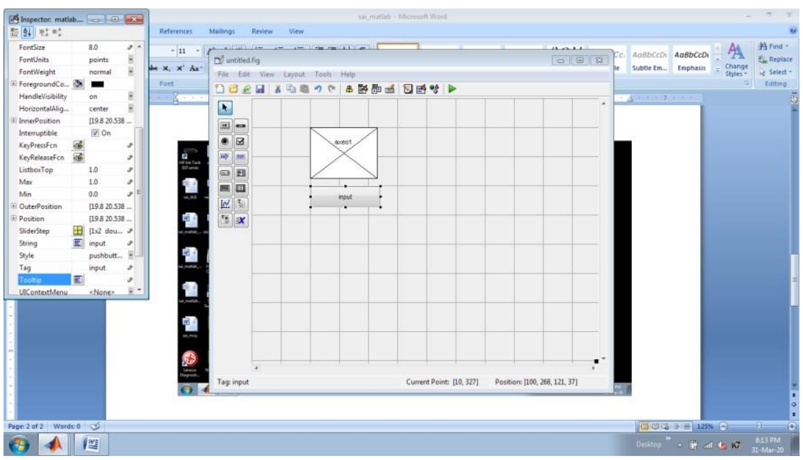


Fig 3.29 GUI Design

The next step is to save the figure. Once the figure is saved, automatically the M-file is also opened where the code has to be modified.

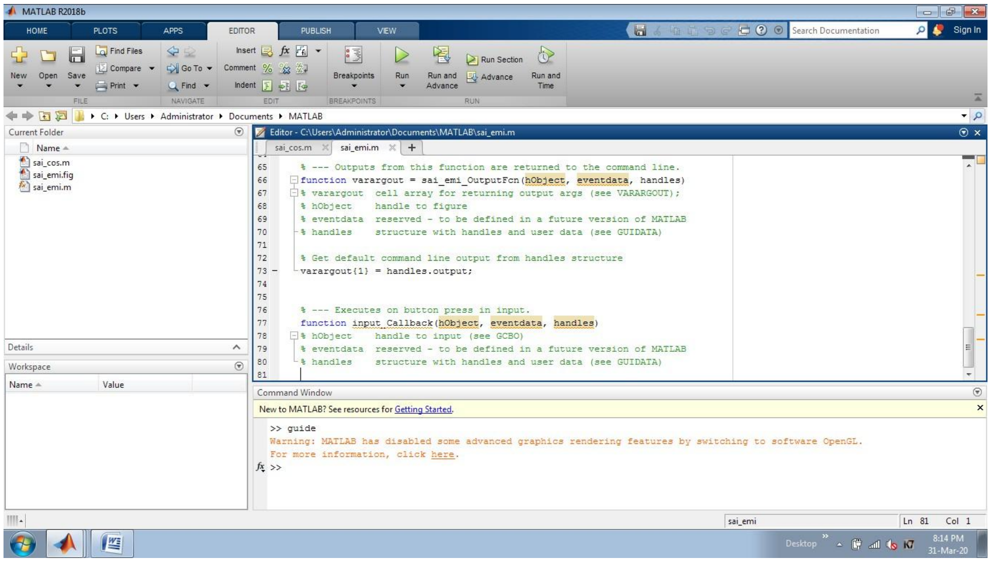


Fig 3.30 GUI call back

Then the program has to be coded for the specific task as given below and is executed. Once executed, the output figure window is opened.

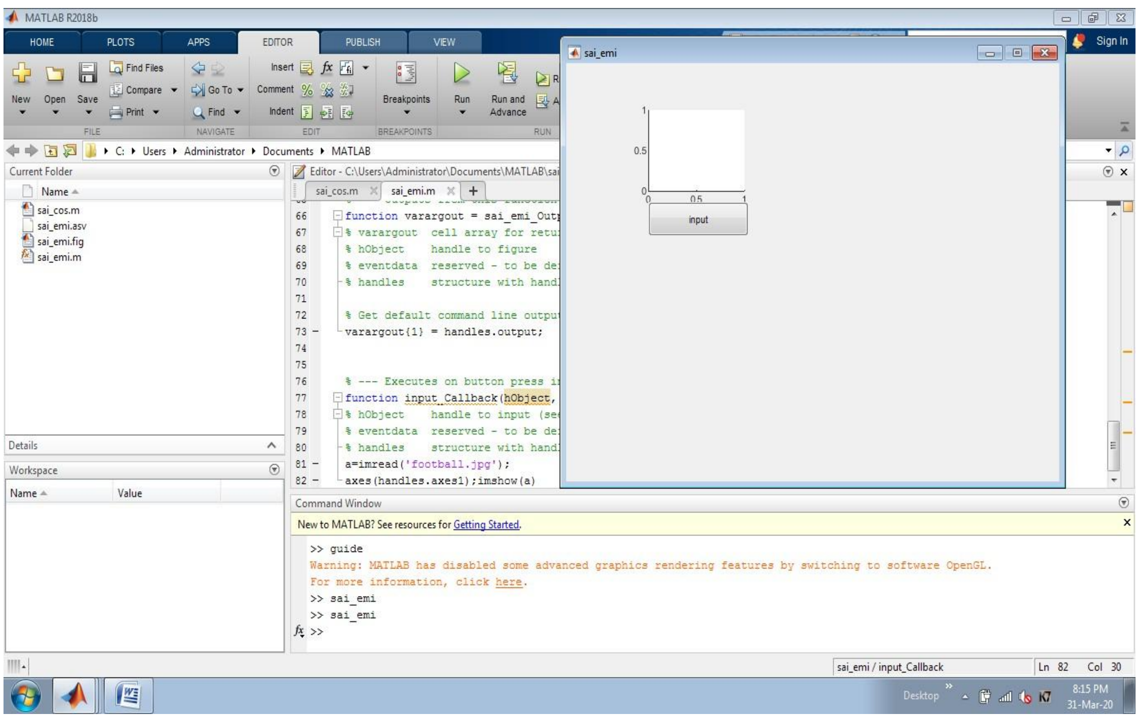


Fig 3.31 GUI at Execution

On a mouse click on pushbutton, the program is executed and the output is obtained.

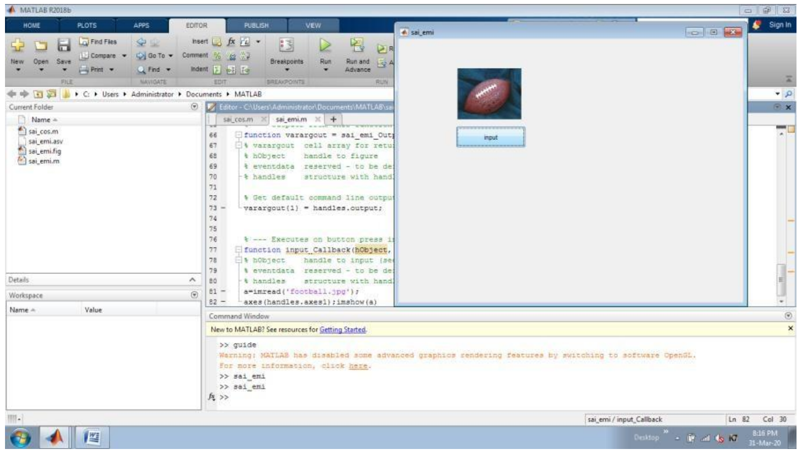


Fig 3.32 GUI Execution Result

In this way, a GUI can be created for any specific task.

REFERENCES

1. Rudra Pratap, "Getting Started with MATLAB 6.0" ,1st Edition, Oxford University Press-2004.
2. Stephen J Chapman, " Programming in MATLAB for Engineers", Brooks, 2002
3. Duane Hanselman ,Bruce LittleField, "Mastering MATLAB 7" , Pearson Education Inc, 2005
4. William J.Palm, "Introduction to MATLAB 6.0 for Engineers", Mc Graw Hill & Co, 2001
5. M.Herniter, "Programming in MATLAB", Thomson Learning, 2001

QUESTION BANK

Part A — CO

1. Write the syntax of ezplot 3
2. Develop a matlab code for stem plot 3
3. What is stair plot? 3
4. Compare bar and barh statement 3
5. What is pie plot? 3
6. Develop a matlab code for exploded slices in a pie graph 3
7. What is Histogram? 3
8. What is the difference between surf plot and mesh plot? 3
9. How to print a .tif file? 3
10. How to print a plot? 3
11. Explain GUI?

Part B

1. Explain the matlab operation to draw a two dimensional plots? 3
2. Write the coding to draw the following plots 3
    a. Stem Plot
    b. Stair plot
    c. Bar plot
    d. Pie plot
3. Explain about Histogram and also how to draw Histogram. 3
4. Explain about function handles for optimization with examples. 3
5. Explain the matlab programming to draw a three dimensional plots. 3
6. Write the program to draw the following plots 3
    a. Contour Plot
    b. mesh plot
    c. surf plot
7. Explain GUI with any one as an example. 3
8. Develop a matlab code for performing result analysis of a class for 5 Different subjects in a semester. 3

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – IV – Software Tools for Engineering Applications – SECA4001

UNIT 4 ADVANCED MATHEMATICS APPLICATIONS

Fitting Curves to Data -Polynomials, Addition, multiplication and division of polynomials, Roots and derivative of a polynomial, curve fitting, polyfit, Interpolation, Extrapolation, Least squares, basic fitting interface, Complex Numbers, Adding, Subtracting and Multiplying Complex Numbers, Integration and Differentiation, Trapezoidal Rule, Calculus in Symbolic Math Toolbox

DIFFERENTIATION AND INTEGRATION

Differentiation

The derivative of a function y = f(x) is written as f'(x) and is defined as the rate of change of thedependent variable y with respect to x. The derivative is the slope of the line tangent to the function at a given point.

MATLAB has a function **polyder**, which will find the derivative of a polynomial.

For example, for the polynomial $x^3 + 2x^2 – 4x + 3$, which would be represented by the vector [12 - 4 3], the derivative is found by:

origp = [1 2 -4 3]; diffp = polyder(origp)diffp =

3 4 -4

which shows that the derivative is the polynomial $3x^2 + 4x – 4$. The function **polyval** can then beused to find the derivative for certain values of x;

for example for x = 1, 2, and 3:

polyval(
diffp,
1:3)ans
=

3 16 35

MATLAB has a built-in function, **diff**, which returns the differences between consecutiveelements in a vector.

For example, diff([4 7 15 32])

ans =
3 8 17

For a function y = f(x) where x is a vector, the values of f'(x) can be approximated as **diff(y)** divided by **diff(x)**. For example, the previous equation can be written as an anonymous function.

```
f = @ (x) x .^ 3 + 2 .* x .^ 2 - 4 .* x + 3;x = 1:3;
y=f(x)
y =
2 11 36
diff(y)ans = 9 25
diff(x)ans = 1 1
diff(y) ./ diff(x)
ans =
9 25
```

To differentiate, syms function can be used as follows:

```
syms x f
f = x^3 + 2*x^2 - 4*x + 3
diff(f)
ans =
3*x^2-4+4*x
```

Integration

There are several functions in Symbolic Math Toolbox to perform calculus operations symbolically; for example, **diff** to differentiate and **int** to integrate.

For example, to find the indefinite integral of the function f(x)

= 3x2 – 1:syms x
int(3*x^2 - 1)
ans =x^3-x

Instead, to find the definite integral of this function from x =

2 to x = 4:int(3*x^2 - 1, 2, 4)
ans =
54

Limits can be found using the **limit** function; for example, for the difference equation describedpreviously:

```
 syms x h
f = x^3 + 2*x^2 - 4*x + 3
limit((f(x+h)-f(x))/h,h,0)
 ans =
                 3*x^2-4+4*x
```

Numerical Integration

```
q = integral(fun,xmin,xmax)
q = integral(fun,xmin,xmax,Name,Value)
```

Description

q = integral(fun,xmin,xmax) numerically integrates function fun from xmin to xmax using global adaptive quadrature and default error tolerances.

q = integral(fun,xmin,xmax,Name,Value) specifies additional options with one or more Name,Value pair arguments. For example, specify 'WayPoints' followed by a vector of real or complex numbers to indicate specific points for the integrator to use.

Create the function f(x)=e−x2(lnx)2

```
fun = @(x) exp(-x.^2).*log(x).^2;

>> q = integral(fun,0,Inf)
>> q = 1.9475
```

Numerical integration by Trapezoidal Rule

Z = **trapz(Y)** computes an approximation of the integral of Y via the trapezoidal method (with unit spacing). To compute the integral for spacing different from one, multiply Z by the spacing increment.

For vectors, trapz(Y) is the integral of Y. For matrices, trapz(Y) is a row vector with the integral over each column. For N-D arrays, trapz(Y) works across the first non-singleton dimension.

Z = trapz(X,Y) computes the integral of Y with respect to X using the trapezoidal method. X and Y must be vectors of the same length, or X must be a column vector and Y an array whose first non-singleton dimension is length(X). trapz operates along this dimension.

Z = trapz(X,Y,DIM) or trapz(Y,DIM) integrates across dimension DIM of Y. The length of X must be the same as size(Y,DIM)).

Example: If Y = [0 1 2 3 4 5] then trapz(Y,1) is [1.5 2.5 3.5] and trapz(Y,2) is [2 8];

INTERPOLATION AND EXTRAPOLATION

Since MATLAB only represents functions as arrays of values, a common problem that comes up is finding function values at points not in the arrays. Finding function values between data points in the array is called **interpolation**; finding function values beyond the endpoints of the array is called **extrapolation**.

It can be done in three different ways

- Polynomial interpolation
  - Using interp commands in MATLAB
  - Using polyfit and polyval
- FFT based Interpolation

Polynomial Interpolation

The function **interp1** performs one-dimensional interpolation, an important operation for data analysis and curve fitting. This function uses polynomial techniques, fitting the supplied data with polynomial functions between data points and evaluating the appropriate function at the desired interpolation points.

Its most general form is yi = interp1(x,y,xi,method) where y is a vector containing the values of a function, and x is a vector of the same length containing the points for which the values in y are given. xi is a vector containing the points at which to interpolate. method is an optional string specifying an interpolation method.

Methods

- **Nearest neighbor interpolation** (method = 'nearest'). This method sets the value of an interpolated point to the value of the nearest existing data point.
- **Linear interpolation** (method = 'linear'). This method fits a different linear function between each pair of existing data points, and returns the value of the relevant function at the points specified by xi. This is the default method for the interp1 function.
- **Cubic spline interpolation** (method = 'spline'). This method fits a different cubic function between each pair of existing data points, and uses the spline function to perform cubic spline interpolation at the data points.

- **Cubic interpolation** (method = 'pchip' or 'cubic'). These methods are identical. They use the pchip function to perform piecewise cubic Hermite interpolation within the vectors x and y. These methods preserve monotonicity and the shape of the data.

Example

```
x=[1 2 3 4 5]; y=[10 20 30 40 50];
x1=1.5;
y1=interp1(x,y,x1);
y1
=15
```

Extrapolation

If any element of xi is outside the interval spanned by x, the specified interpolation method is used for extrapolation. Alternatively,

yi = interp1(x,y,xi,method,extrapval) replaces extrapolated values with extrapval. NaN is often used for extrapval.

Two Dimensional Interpolation

The function interp2 performs two-dimensional interpolation, an important operation for image processing and data visualization.

Its most general form is

ZI = interp2(X,Y,Z,XI,YI,method) where Z is a rectangular array containing the values of a two-dimensional function, and X and Y are arrays of the same size containing the points for which the values in Z are given. XI and YI are matrices containing the points at which to interpolate the data. method is an optional string specifying an interpolation method. There are three different interpolation methods for two-dimensional data.

- Nearest neighbor interpolation (method = 'nearest'). This method fits a piecewise constant surface through the data values. The value of an interpolated point is the value of the nearest point.
- Bilinear interpolation (method = 'linear'). This method fits a bilinear surface through existing data points. The value of an interpolated point is a combination of the values of the four closest points. This method is piecewise bilinear, and is faster and less memory-intensive than bicubic interpolation.
- Bicubic interpolation (method = 'cubic'). This method fits a bicubic surface through existing data points. The value of an interpolated point is a combination of the values of the sixteen closest points. This method is piecewise bicubic, and produces a much smoother surface than bilinear interpolation. This can be a key advantage for applications like image processing. Use bicubic interpolation when the interpolated data and its derivative must be continuous.

All of these methods require that X and Y be monotonic, that is, either always increasing or always decreasing from point to point. One should prepare these matrices using the meshgrid function, or else be sure that the “pattern” of the points emulates the output of meshgrid. In addition, each method automatically maps the input to an equally spaced domain before interpolating. If X and Y are already equally spaced, one can speed execution time by prepending an asterisk to the method string, for example, '*cubic'.

Three dimensional and multidimensional interpolation functions are also available in MATLAB

Polyfit and polyval functions

polyfit finds the coefficients of a polynomial that fits a set of data in a least-squares sense.
p = polyfit(x,y,n) where x and y are vectors containing the x and y data to be fitted, and n is theorder of the polynomial to return.

For example, consider the x-y test data.
x = [1 2 3 4 5]; y = [5.5 43.1 128 290.7 498.4];
A third order polynomial that approximately fits the data isp = polyfit(x,y,3)

p = -0.1917 31.5821 -60.3262 35.3400

Having determined the third order polynomial, using polyval function in MATLAB, values of yfor any values of x can be determined.

Here, let x2=[1.5,2.5];y1=polyval(p,x2);
y1 = 15.2637 78.9181

In this way, polyfit and polyval functions can be used for interpolation.

polyfit(x,y,n) finds the coefficients of a polynomial p(x) of degree n that fits the y data by minimizing the sum of the squares of the deviations of the data from the model (least-squares fit).
polyval(p,x) returns the value of a polynomial of degree n that was determined by polyfit, evaluated at x.

```
t = [0 0.3 0.8 1.1 1.6 2.3];
y = [0.6 0.67 1.01 1.35 1.47 1.25];
p = polyfit(t,y,2)
```

>> p = *1×3*

-0.2942 1.0231 0.4981

MATLAB calculates the polynomial coefficients in descending powers.

The second-degree polynomial model of the data is given by the equation

y=−0.2942t2+1.0231t+0.4981.

```
t2 = 0:0.1:2.8;
y2 = polyval(p,t2);
figure
plot(t,y,'o',t2,y2)
title('Plot of Data (Points) and Model (Line)')
```

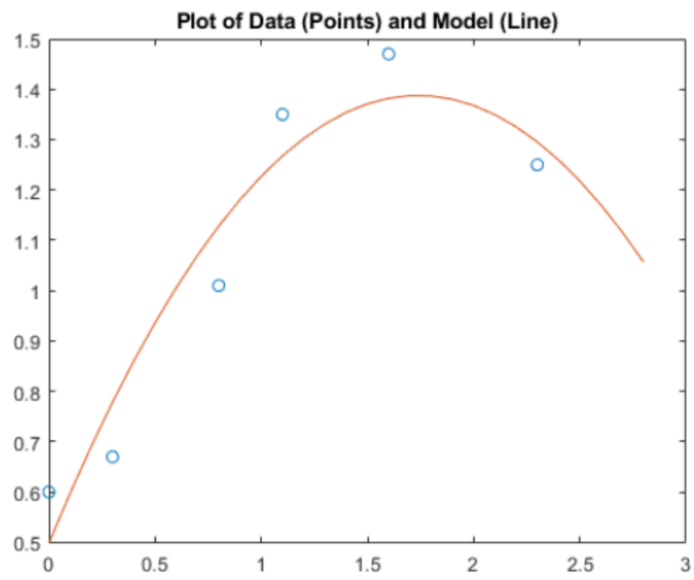


Fig.4.1 Interpolation curve

In addition, to the above mentioned interpolation techniques, **Linspace** can also be used forobtaining data points between any two values.

y = linspace(x1,x2,n) generates n points. The spacing between the points is (x2-x1)/(n-1).

- linspace is similar to the colon operator, “:”
- but gives direct control over the number of points and always includes theendpoints
- “lin” in the name “linspace” refers to generating linearly spaced values

Example

y1 = linspace(-5,5,7)

y1 = *1×7*

-5.0000 -3.3333 -1.6667 0 1.6667 3.3333 5.0000

- Here 1x7 specifies seven values stored in a single row
- Totally 7 values
- It includes both -5 and +5

y = linspace(x1,x2) returns a row vector of 100 evenly spaced points between x1 and x2.

FFT Based Interpolation

The function interpft performs one-dimensional interpolation using an FFT-based method. This method calculates the Fourier transform of a vector that contains the values of a periodic function. It then calculates the inverse Fourier transform using more points.

Its form is

y = interpft(x,n) where x is a vector containing the values of a periodic function, sampled at equally spaced points. n is the number of equally spaced points to return.

Polynomial Multiplication in MATLAB

The matlab function **conv** (convolution) can be used to perform polynomial multiplication. For example:

```
B1 = [1 1];   % 1st row of Pascal's triangle
B2 = [1 2 1]; % 2nd row of Pascal's triangle
B3 = conv(B1,B2) % 3rd row
% B3 = 1  3  3  1
B4 = conv(B1,B3) % 4th row
% B4 = 1  4  6  4  1
```

% ..The matlab conv(B1,B2) is identical to filter(B1,1,B2), except that conv returns the complete convolution of its two input vectors, while filter truncates the result to the length of the ``input signal'' B2.7.11 Thus, if B2 is zero-padded with length(B1)-1 zeros, it will return the complete convolution:

```
B1 = [1 2 3];
B2 = [4 5 6 7];
conv(B1,B2)
% ans = 4  13  28  34  32  21
filter(B1,1,B2)
% ans = 4  13  28  34
filter(B1,1,[B2,zeros(1,length(B1)-1)])
% ans = 4  13  28  34  32  21
```

Division of two polynomial

r = **polynomialReduce(p,d)** returns the Polynomial Reduction of p by d with respect to all variables in p determined by symvar. The input d can be a vector of polynomials.

```
syms x y
p = x^3 - x*y^2 + 1;
d = x + y;
[r,q] = polynomialReduce(p,d)
```

Table 4.1 Calculus in Symbolic Math Toolbox

`limit`	Limit of symbolic expression
`diff`	Differentiate symbolic expression or function
`functionalDerivative`	Functional derivative (variational derivative)
`int`	Definite and indefinite integrals
`vpaintegral`	Numerical integration using variable precision
`changeIntegrationVariable`	Integration by substitution
`integrateByParts`	Integration by parts
`release`	Evaluate integrals

Table 4.2 Vector Analysis

`curl`	Curl of vector field
`divergence`	Divergence of vector field
`gradient`	Gradient vector of scalar function
`hessian`	Hessian matrix of scalar function
`jacobian`	Jacobian matrix
`laplacian`	Laplacian of scalar function
`potential`	Potential of vector field
`vectorPotential`	Vector potential of vector field

Table 4.3 Series Expansion

`pade`	Pade approximant
`rsums`	Interactive evaluation of Riemann sums
`series`	Puiseux series
`taylor`	Taylor series

Table 4.4 Sums & Products

`cumprod`	Symbolic cumulative product
`cumsum`	Symbolic cumulative sum
`symprod`	Product of series
`symsum`	Sum of series
`vpasum`	Numerical summation using variable precision

REFERENCES

- Stephen J Chapman, “ Programming in MATLAB for Engineers”, Brooks, 2002
- Duane Hanselman ,Bruce LittleField, “Mastering MATLAB 7” , Pearson Education Inc, 2005
- William J.Palm, “Introduction to MATLAB 6.0 for Engineers”, Mc Graw Hill & Co, 2001
- M.Herniter, “Programming in MATLAB”, Thomson Learning, 2001
- www.mathworks.com

PART A

		CO
1	What is the MATLAB function for finding the 2D interpolation	4
2	How trapezoidal rule of integration is obtained with MATLAB?	4
3	Obtain differentiation of 3x^2+2x-1 with a matlab function	4
4	Distinguish between *polyfit* and *polyval*	4
5	What is the need for datafitting?	4
6	How to represent complex numbers in matlab?	4
7	Give few matlab functions for obtaining integration	4
8	How to perform cumulative integration in matlab?	4
9	Perform addition of two polynomial in matlab	4
10	What is extrapolation?	4

PART B 4

1 In a chemical reactor, the temperature and pressure measurement is made at every 2 hour for 24 hours. The measurement data is given in the table 1. Develop a MATLAB program to make a trend of the data, obtain the maximum and minimum temperature and pressure within 24 hours and obtain the pressure at 1.5 hrs when the temperature is 15.8 4

Time (hr)	0	2	4	6	8	10	12	14	16	18	20	22
Temperature (°C)	10	20	30	45	57	68	89	90	105	134	163	210
Pressure (psi)	5	6.4	8.2	15.1	34	49.5	93.2	99	106	129.5	140	150.5

2	Explain polynomial multiplication using MATLAB with a suitable example	4
3	Explain various built-in functions that can perform integration	4
4	How differentiation is performed in matlab? Explain with suitable examples.	4

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – V – Software Tools for Engineering Applications – SECA4001

UNIT V

Simulink- Introduction, Block setting, Model annotation, solver, sinks library, sources, math operations library, user defined functions and look up table in Simulink, ports and subsystems, masked sub system, program controls in Simulink: FOR, WHILE, IF, CASE, Signal routing and logical's, Exporting Simulink data to Matlab, Applications – Modelling of a simple PID controller using SIMULINK, Plotting the Frequency response of FIR & IIR filters using MATLAB.

CREATING A SIMPLE MODEL IN SIMULINK

You can use Simulink® to model a system and then simulate the dynamic behavior of that system. Simulink allows you to create block diagrams, where blocks you connect represent parts of a system, and signals represent input/output relationships between those blocks. The primary function of Simulink is to simulate behavior of system components over time. In its simplest form, this task involves keeping a clock, determining the order in which the blocks are to be simulated, and propagating the outputs, computed in the block diagram, to the next block. Consider a switch that turns on a heater. At each time step, Simulink must compute the output of the switch, propagate it to the heater, and then compute the heat output.

Often, the effect of a component's input on its output is not instantaneous. For example, turning on a heater does not result in an instant change in temperature. Rather, this action provides input to a differential equation, and the history of the temperature (a *state*) is also a factor. When the simulation of a block diagram requires solving a differential or difference equation, Simulink employs memory and numerical solvers to compute the state values for the time step.

Simulink handles data in three categories:

- Signals — Block inputs and outputs, computed during simulation
- States — Internal values, representing the dynamics of the block, computed during simulation
- Parameters — Values that affect the behavior of a block, controlled by the user

At each time step, Simulink computes new values for signals and states. By contrast, you specify parameters when you build the model and can occasionally change them while simulation is running.

Model Overview

The basic techniques you use to create a simple model in this tutorial are the same techniques that you use for more complex models. This example simulates simplified motion of a car, after a brief press of the accelerator pedal.

A Simulink block is a model element that defines a mathematical relationship between its input and output. To create this simple model, you need four Simulink blocks.

Table 5.1 Simulink blocks

Block name	Block Purpose	Model Purpose
Pulse Generator	Generate an input signal for the model	Simulate the accelerator pedal
Gain	Multiply the input signal by a factor	Simulate how pressing the acceleratoraffects the car’s acceleration
Integrator, Second-Order	Integrate input signal twice	Obtain position from acceleration
Outport	Designate a signal as an output fromthe model	Designate the position as an output fromthe model

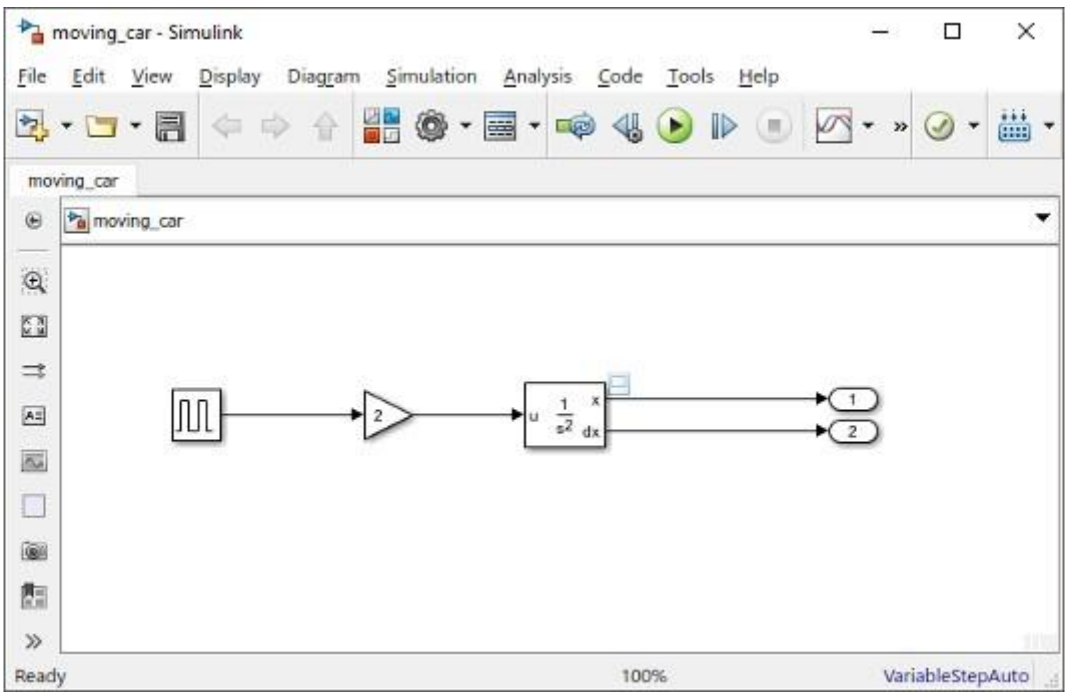


Figure 5.1 Simulink Model

Simulating this model integrates a brief pulse twice to get a ramp and then displays the result in a Scope window. The input pulse represents a press of the accelerator pedal in a car, and the output ramp represents the increasing distance from the starting point.

Open New Model

Use the Simulink Editor to build your models.

1. Start MATLAB®. From the MATLAB Toolstrip, click the **Simulink** button .

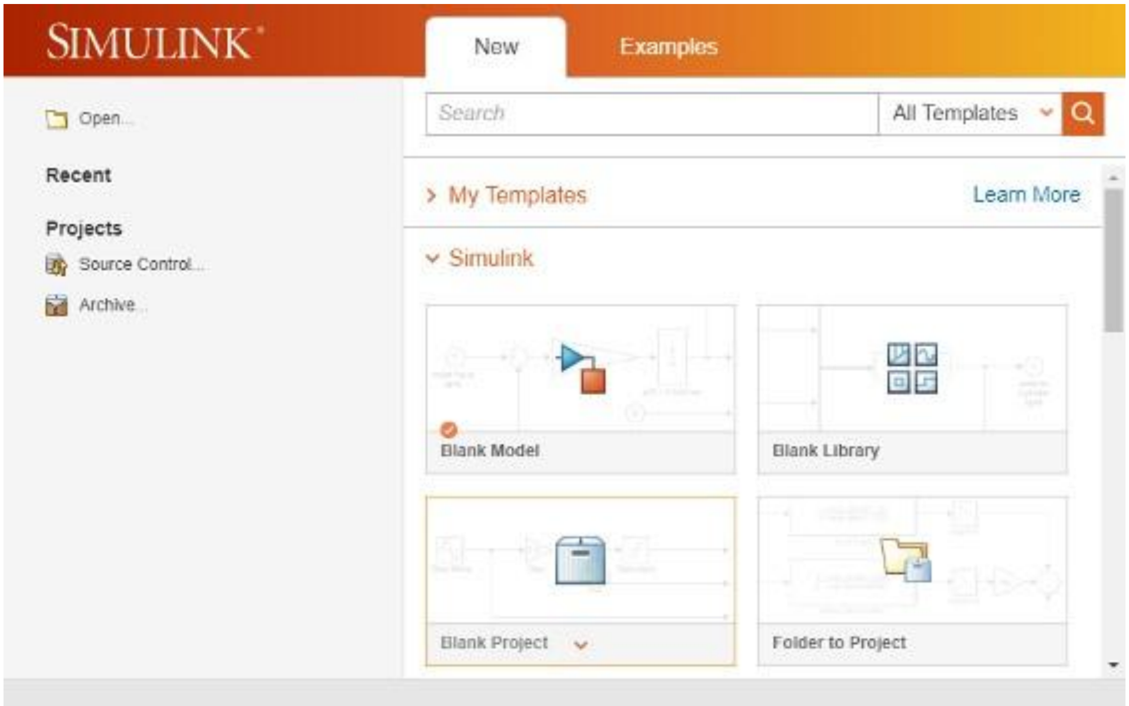


Figure 5.2 Simulink new file

2. Click the **Blank Model** template.

   The Simulink Editor opens.

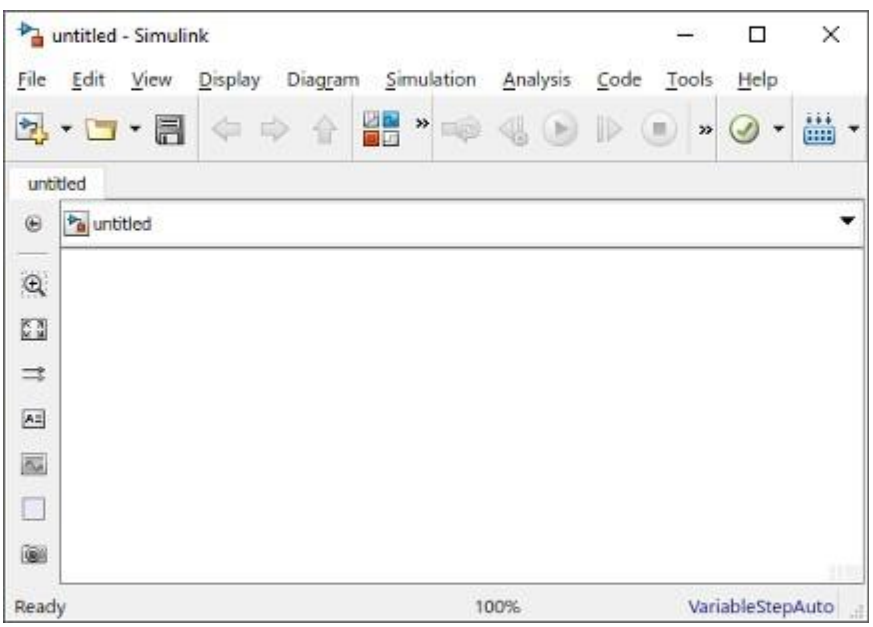


Figure 5.3 New Model file

3. From the **File** menu, select **Save as**. In the **File name** text box, enter a name for your model,For example, simple_model. Click **Save**. The model is saved with the file extension.slx.

Open Simulink Library Browser

Simulink provides a set of block libraries, organized by functionality in the Library Browser. Thefollowing libraries are common to most workflows:

- Continuous — Building blocks for systems with continuous states
- Discrete — Building blocks for systems with discrete states
- Math Operations — Blocks that implement algebraic and logical equations
- Sinks — Blocks that store and show the signals that connect to them
- Sources — Blocks that generate the signal values that drive the model

1. From the Simulink Editor toolbar, click the **Library Browser** button .

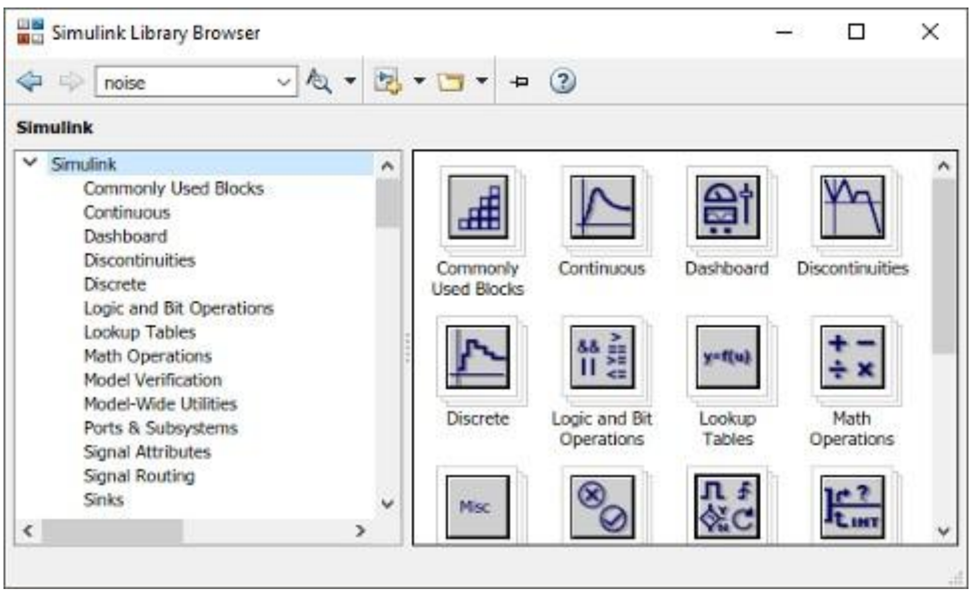


Figure 5.4 Simulink Library

2. Set the Library Browser to stay on top of the other desktop windows. On the Library Browsertoolbar, select the **Stay on top** button .

To browse through the block libraries, select a MathWorks® product and then a functional area inthe left pane. To search all of the available block libraries, enter a search term.

For example, find the Pulse Generator block. In the search box on the browser toolbar, enter pulse, and then press the Enter key. Simulink searches the libraries for blocks with pulse in their name or description, and then displays the blocks.

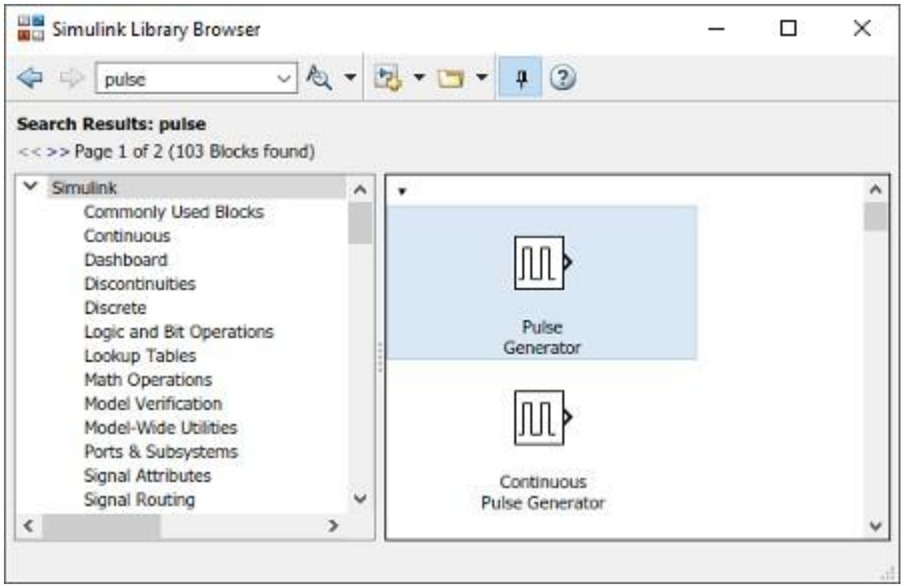


Figure 5.5 Simulink Library blocks

Get detailed information about a block. Right-click a block, and then select **Help for the PulseGenerator block**. The Help browser opens with the reference page for the block.

Blocks typically have several parameters. You can access all parameters by double-clicking theblock.

Add Blocks to a Model

To start building the model, browse the library and add the blocks.

1. From the Sources library, drag the Pulse Generator block to the Simulink Editor. A copy of the Pulse Generator block appears in your model with a text box for the value of the **Amplitude** parameter. Enter 1.

Figure 5.6 Simulink block – pulse generator

Parameter values are held throughout the simulation.

2. Add the following blocks to your model using the same approach.

Block	**Library**
Gain	Simulink/Math Operations
Integrator, Second Order	Simulink/Continuous
Outport	Simulink/Sinks

3. Add a second Outport block right-clicking and dragging the existing one.
4. Your model should now have the blocks you need.
5. Arrange the blocks as follows by clicking and dragging each block. To resize a block, clickand drag a corner.

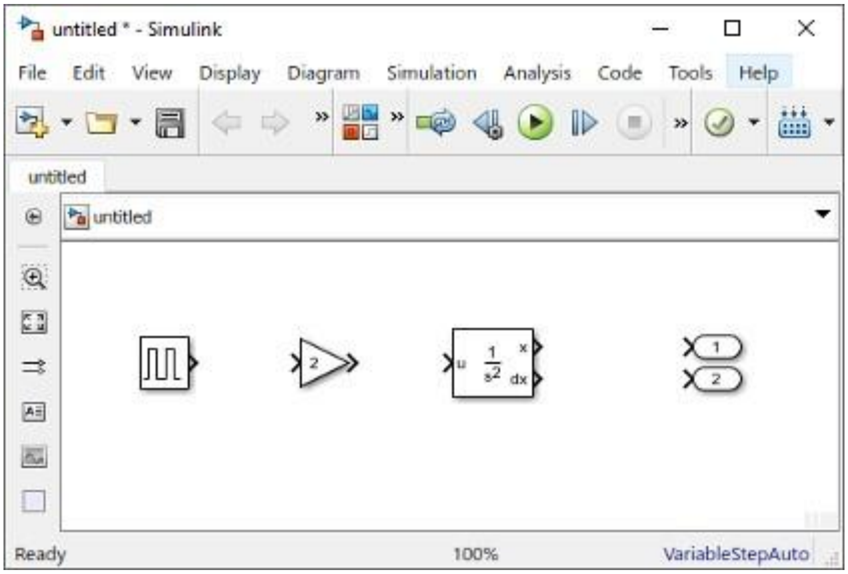


Figure 5.7 Placing simulink blocks

Connect Blocks

Connect the blocks by creating lines between output ports and input ports.

1. Click the output port on the right side of the Pulse Generator block.

   The output port, and all input ports suitable for a connection get highlighted.

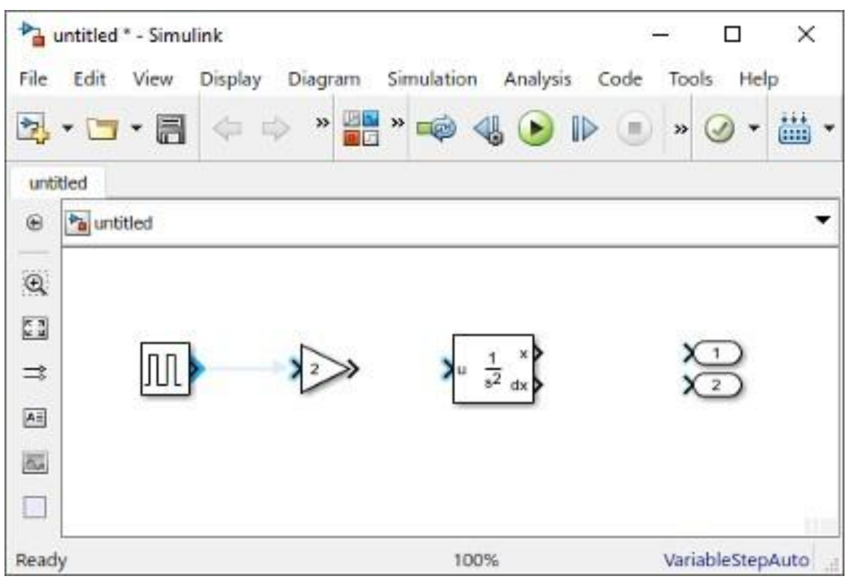


Figure 5.8 Simulink connections between blocks

2. Click the input port of the Gain block.

   Simulink connects the blocks with a line and an arrow indicating the direction of signal flow.

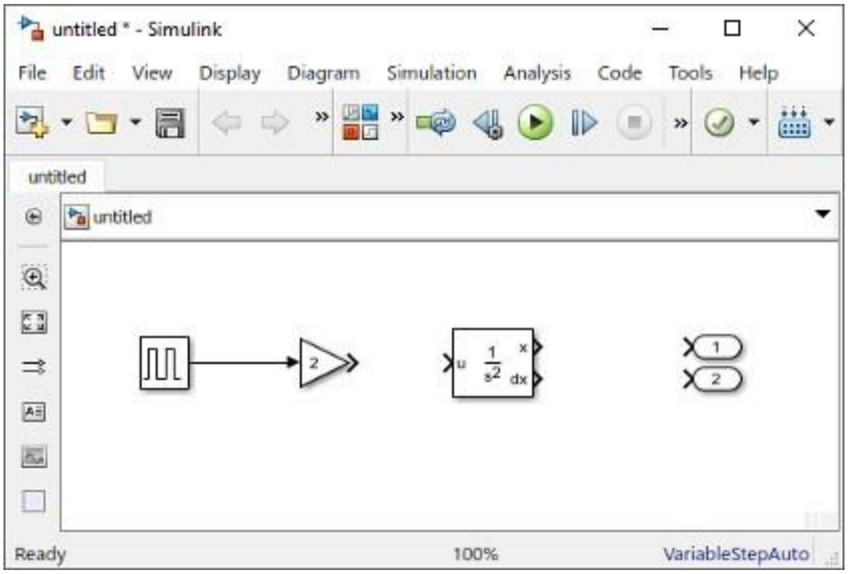


Figure 5.9 Simulink connections between blocks

3. Connect the output port of the Gain block to the input port on the Integrator, SecondOrder block.
4. Connect the two outputs of the Integrator, Second Order block to the two Outport blocks.
5. Save your model. Select **File** > **Save** and provide a name.

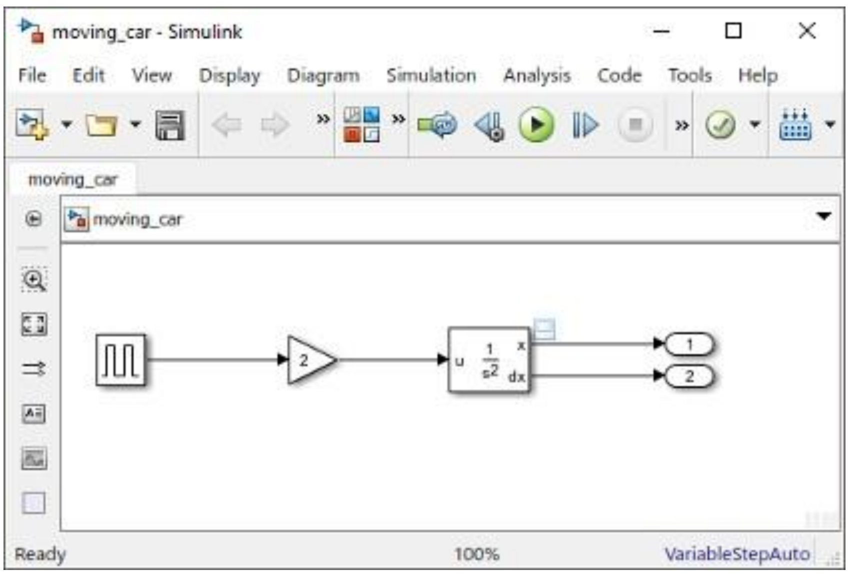


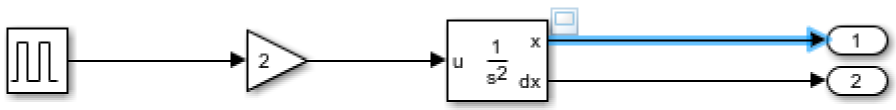
Figure 5.10 Simulink completed connections

Your model is complete.

Add Signal Viewer

To view the results, connect the first output to a Signal Viewer.

Access the context menu by right-clicking the signal. Select **Create & Connect Viewer >Simulink > Scope**. This creates a viewer icon on the signal, and opens a Viewer display.

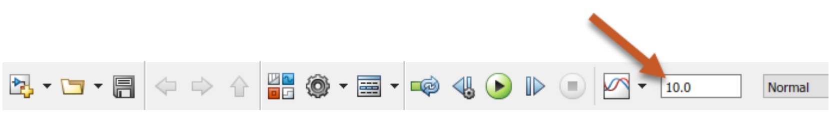


You can open the viewer at any time by double-clicking the icon.

Run Simulation

After you define the configuration parameters, you are ready to simulate your model.

1. On the model window, set the simulation stop time by changing the value at the toolbar.



   The default stop time of 10.0 is appropriate for this model. This time value has no unit. Time unit in Simulink depends on how the equations are constructed. This example simulates the simplified motion of a car for 10 seconds.

2. To run the simulation, click the **Run** simulation button .

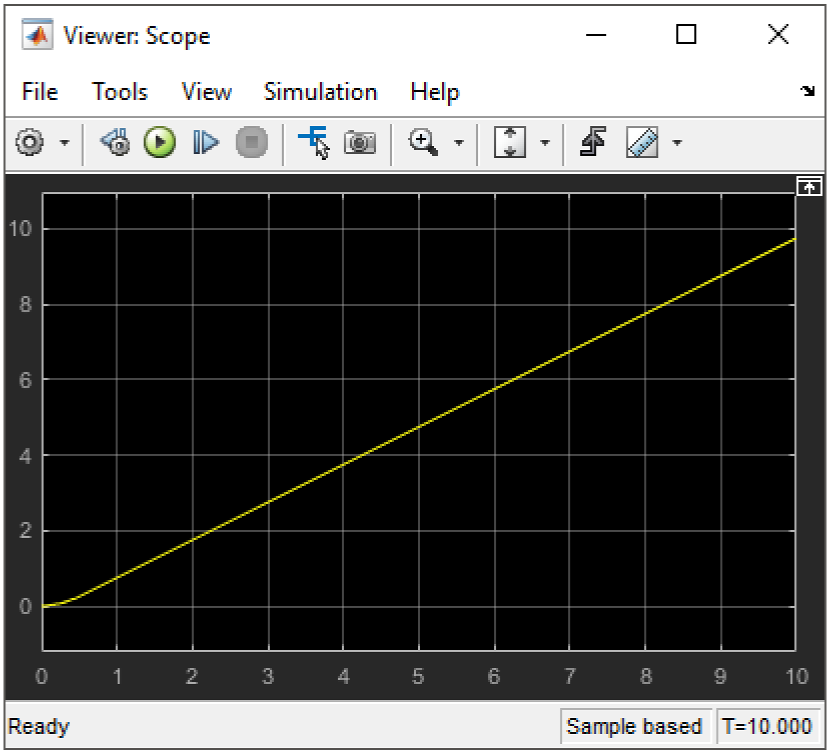
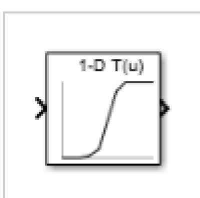


Figure 5.11 Signal in Simulink Scope

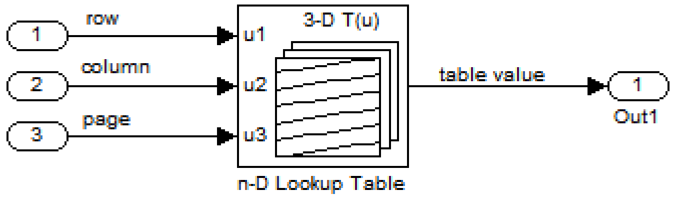
1-D Lookup Table

Library:
Simulink / Lookup Tables
HDL Coder / Lookup Tables



Description
Supported Block Operations
The 1-D, 2-D, and n-D Lookup Table blocks evaluate a sampled representation of a function in N variables
y=F(x1,x2,x3,...,xN)
where the function *F* can be empirical. The block maps inputs to an output value by looking up or interpolating a table of values you define with block parameters. The block supports flat (constant), linear (linear point-slope), Lagrange (linear Lagrange), nearest, cubic-spline, and Akima spline interpolation methods. You can apply these methods to a table of any dimension from 1 through 30.

In the following block, the first input identifies the first dimension (row) breakpoints, the second input identifies the second dimension (column) breakpoints, and so on.



See Port Location After Rotating or Flipping for a description of the port order for various block orientations.

When the **Math and Data Types** > **Use algorithms optimized for row-major array layout** configuration parameter is set, the 2-D and n-D Lookup Table block behavior changes from column-major to row-major. For these blocks, the column-major and row-major algorithms may differ in the order of the output calculations, possibly resulting in slightly different numerical values.

GENERATING SECOND ORDER SYSTEM RESPONSE IN SIMULINK

To obtain the step response of a 2^{nd} order system for both open and closed loop.

Blocks Required

- Step input
- Transfer Function
- PID Controller
- Summer block
- Scope

- Step input
- Simulink –> Sources –> StepView output
- Simulink –> Sink –> ScopeSummer Block
- Simulink –>

Math Operations–>
SummerTransfer
Function

- Simulink –>

Continuous–> Transfer
functionPID Controller

- Simulink –> Continuous–> PID controller

Block Diagram of Open Loop system and closed loop System

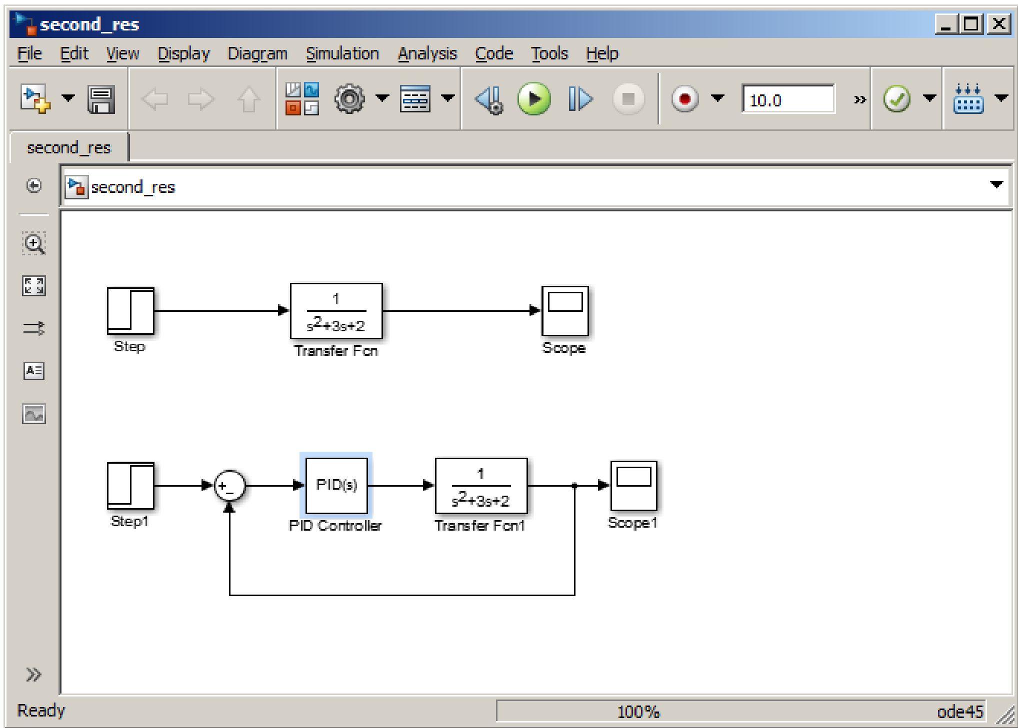


Figure 5.12 Open and closed loop PID controller for second order system

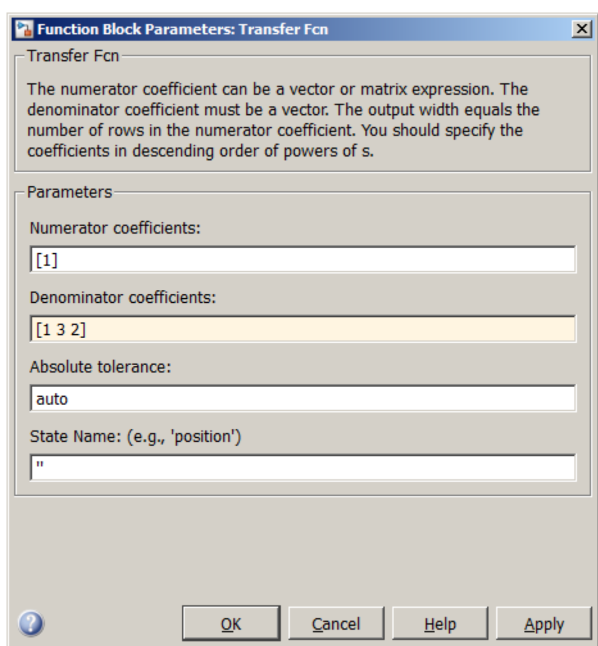


Figure 5.13 Transfer function parameters

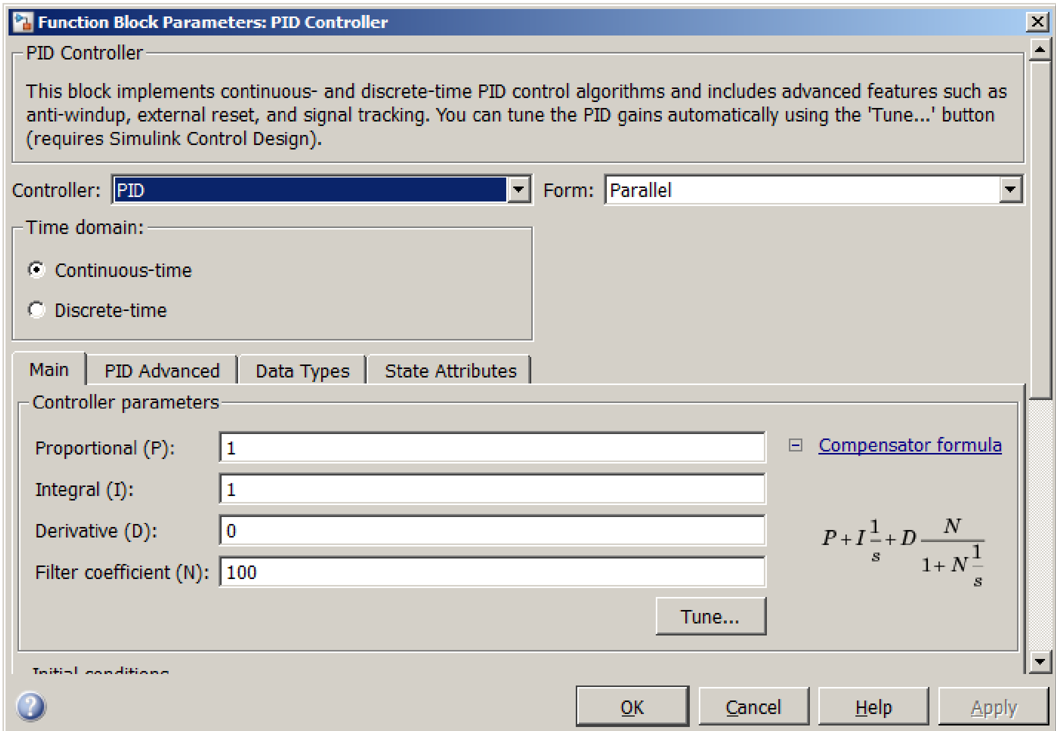


Figure 5.14 PID controller parameters

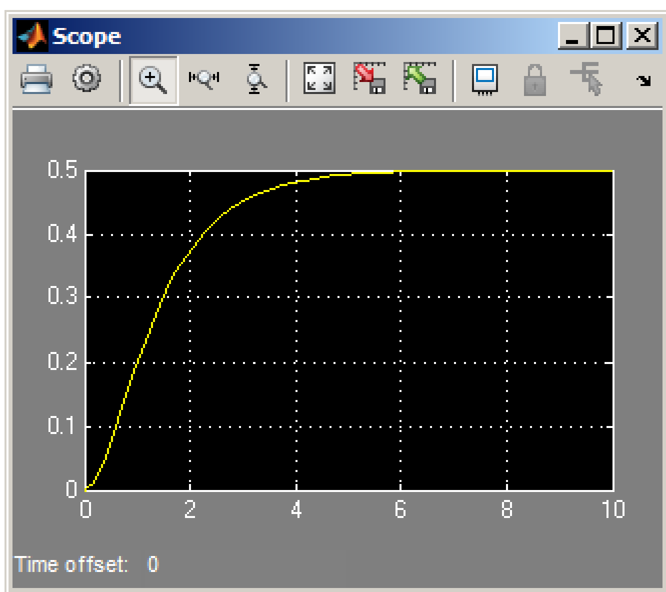


Figure 5.15 Open loop response of second order system

.

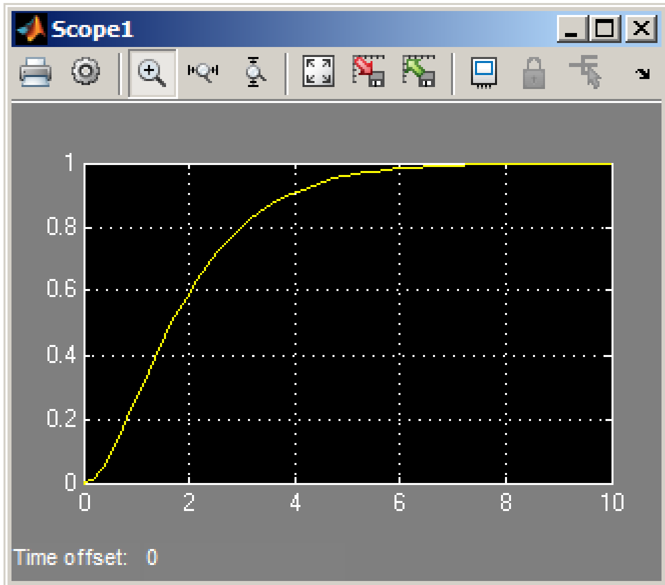


Figure 5.16 Closed loop response of second order system with PID Controller

Create a Subsystem

Subsystem Advantages

Subsystems allow you to create a hierarchical model comprising many layers. A subsystem is a set of blocks that you replace with a single Subsystem block. As your model increases in size andcomplexity, you can simplify it by grouping blocks into subsystems. Using subsystems:

- Establishes a hierarchical block diagram, where a Subsystem block is on one layer and the blocksthat make up the subsystem are on another
- Keeps functionally related blocks together
- Helps reduce the number of blocks displayed in your model window

When you make a copy of a subsystem, that copy is independent of the source subsystem. To reuse the contents of a subsystem across a model or across models, use either model referencingor a library.

Ways to Create a Subsystem

You can create a subsystem using these approaches:

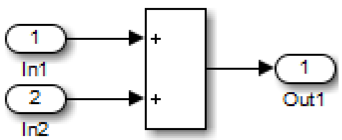
- Add a Subsystem block to your model, and then open the block and add blocks to the subsystemwindow. Create a Subsystem in a Subsystem Block.
- Select the blocks that you want in the subsystem, and from the right-click context menu,select **Create Subsystem from Selection**. Create a Subsystem from Selected Blocks.
- Copy a model to a subsystem. In the Simulink® Editor, copy and paste the model into a subsystem window, or use Simulink.BlockDiagram.copyContentsToSubsystem.
- Copy an existing Subsystem block to a model.
- Drag a box around the blocks you want in a subsystem, and select the type of subsystem youwant from the context options. Create a Subsystem Using Context Options.

Create a Subsystem in a Subsystem Block

Add a Subsystem block to the model, and then add the blocks that make up the subsystem.

1. Copy the Subsystem block from the Ports & Subsystems library into your model.
2. Open the Subsystem block by double-clicking it.
3. In the empty subsystem window, create the subsystem contents. Use Inport blocks to represent input from outside the subsystem and Outport blocks to represent external output.

   For example, this subsystem includes a Sum block and Inport and Outport blocks to representinput to and output from the subsystem.



When you close the subsystem window, the Subsystem block includes a port for each Inportand Outport block.

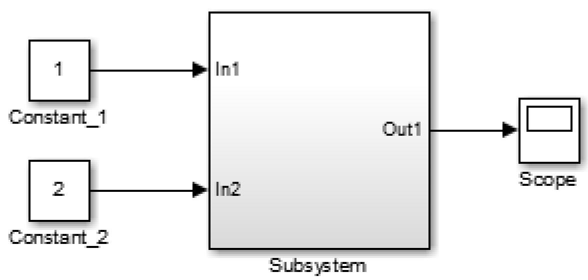


Figure 5.17 Subsystem

Create a Subsystem from Selected Blocks

1. Select the blocks that you want to include in a subsystem. To select multiple blocks in one area of the model, drag a bounding box that encloses the blocks and connecting lines that youwant to include in the subsystem.

   The figure shows a model that represents a counter. The bounding box selects the Sum andUnit Delay blocks.

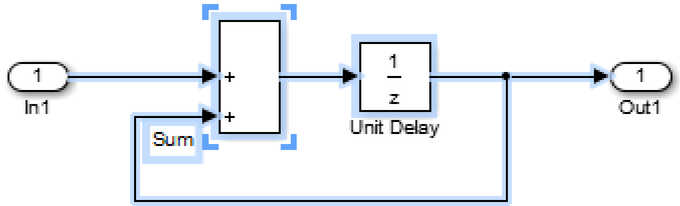
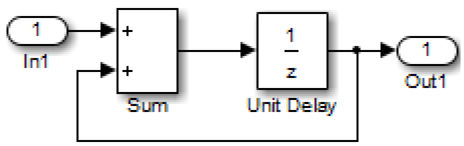


Figure 5.18 Creating subsystem

2. Select **Diagram** > **Subsystems & Model Reference** > **Create Subsystem from Selection**.A Subsystem block appears, which encloses the selected blocks.

To edit the subsystem contents, open the Subsystem block. For example:



adds Inport and Outport blocks to represent input from and output to blocks outside thesubsystem.

You can change the name of the Subsystem block and modify the block the way that you do withany other block (for example, you can mask the subsystem).

Create a Subsystem Using Context Options

1. Drag a box around the blocks you want in your subsystem.

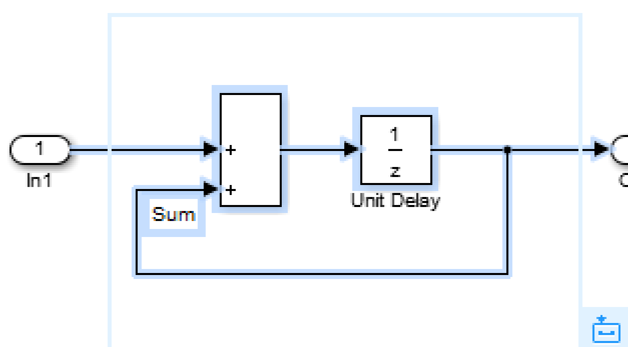
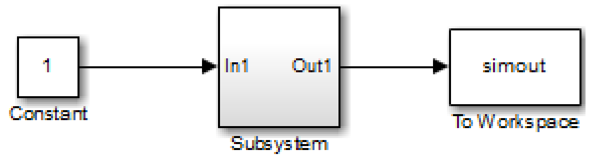


Figure 5.19 Creating subsystem

2. View the subsystems you can create with these blocks by hovering over the first contextoption that appears.



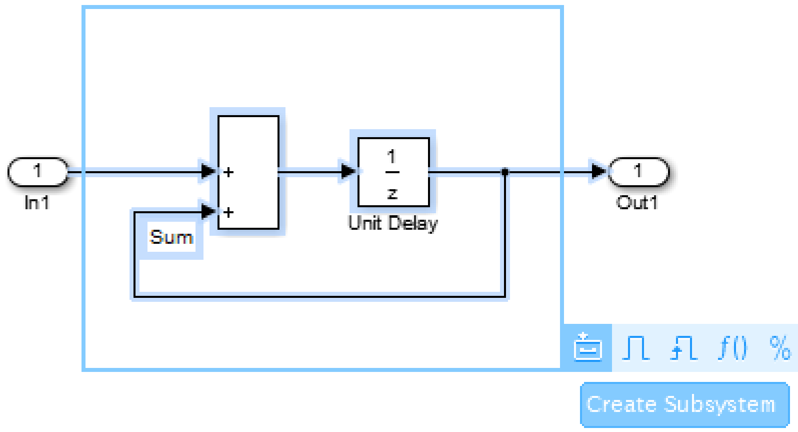


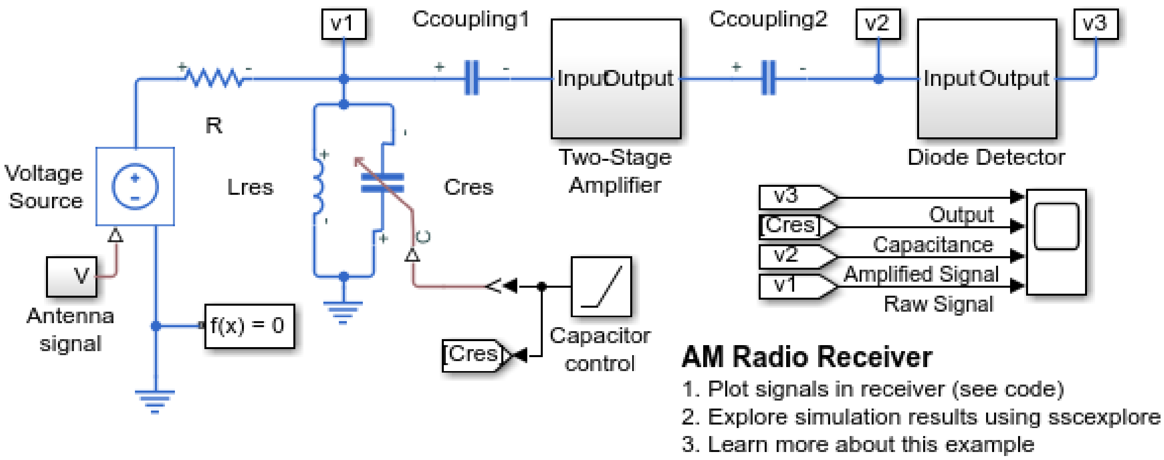
Figure 5.20 Subsystem Created

3. Select the type of subsystem you want to create from these options. A Subsystem block appears, which encloses the selected blocks.

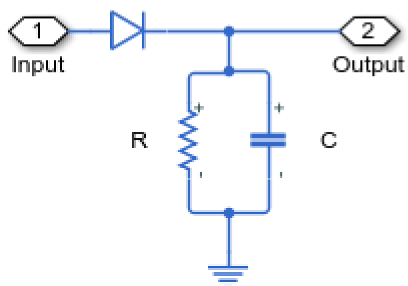
Few application programs in simulinkExample 1: AM Radio Receiver

This example shows a simplified AM radio receiver. A single tone signal at 2 kHz is transmitted with a carrier frequency of 600 kHz. The variable capacitor, Cres, in the resonant circuit is used in order to sweep through a certain frequency span. When the resonance passes through 600 kHz, the signal is picked up and amplified by a two-stage Class A RF power amplifier. The signal is finally extracted by a diode detector, where it would normally be passed on to an audio amplifier (not included here). The Scope displays the final output, the value of the resonant capacitance, and the received and amplified signals.

Model



Diode Detector Subsystem



Two-Stage Amplifier Subsystem

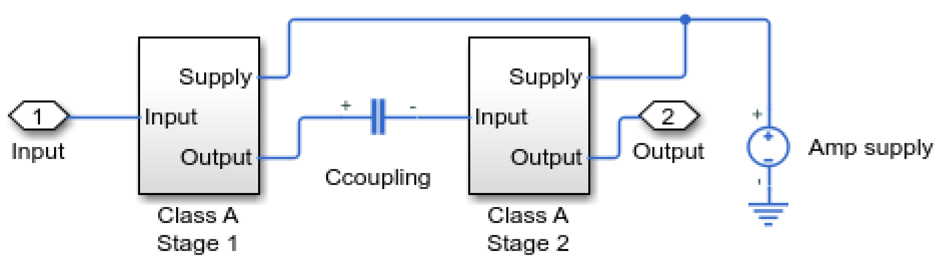


Figure 5.21 Creating AM Radio Receiver model

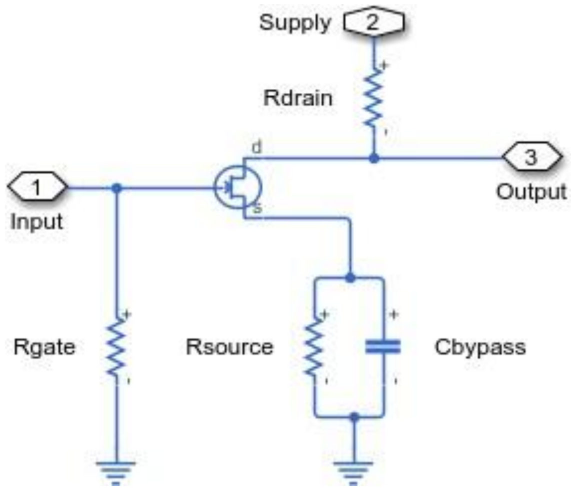
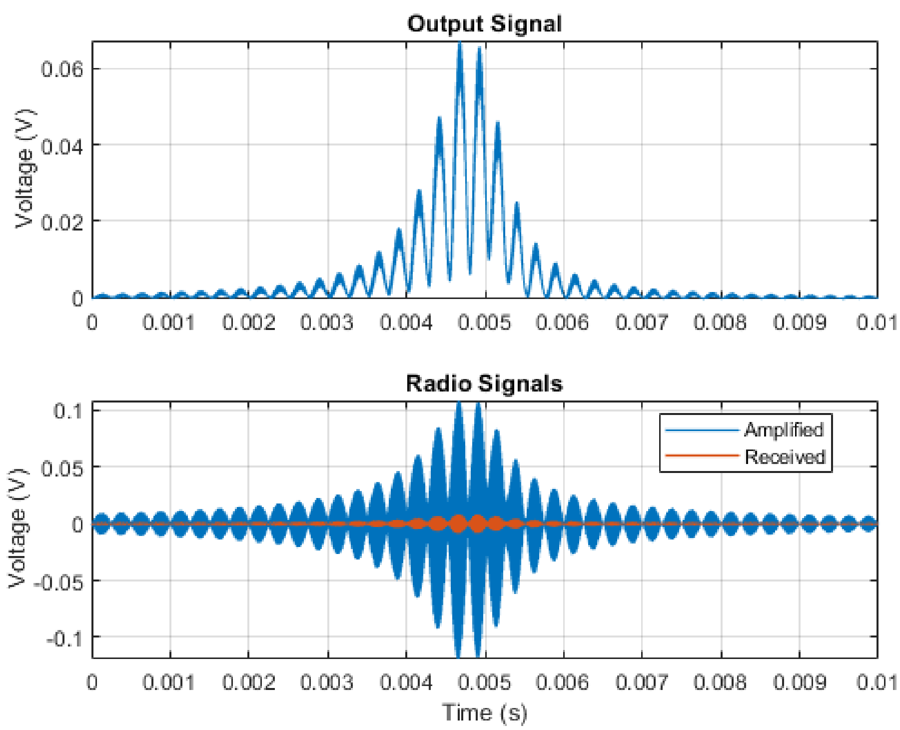


Figure 5.22 Class A Stage1 Subsystem

Simulation Results from Simscape Logging

The plots below shows received, amplified, and output signals in the radio receiver. As the resonance in the resonant circuit passes through 600 kHz, the signal is picked up and amplified by a two-stage Class A RF power amplifier.



.**Figure 5.23 Output of Radio receiver model**

Example 2: FIR and IIR Filter Design and their Frequency Responses

This example shows how to design FIR and IIR filters based on frequency response specifications using the designfilt function in the Signal Processing Toolbox® product. The example concentrates on low pass filters but most of the results apply to other response types as well. And also focuses on the design of digital filters rather than on their applications..

Low pass Filter Specifications

The ideal low pass filter is one that leaves unchanged all frequency components of a signalbelow a designated cutoff frequency ω_c, and rejects all components above ω_c. Because the impulse response required to implement the ideal low pass filter is infinitely long, it is impossible to design an ideal FIR low pass filter

Finite length approximations to the ideal impulse response lead to the presence of ripples in both the pass band ($\omega<\omega_c$) and the stop band ($\omega>\omega_c$) of the filter, as well as to a nonzero transition width between pass band and stop band.

Both the pass band/stop band ripples and the transition width are undesirable but unavoidable deviations from the response of an ideal low pass filter when approximated with a finite impulse response.

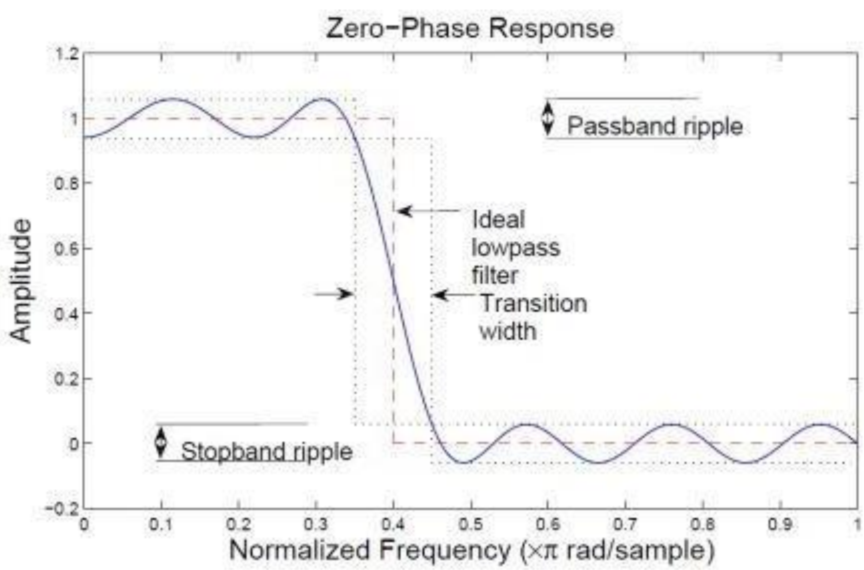


Figure 5.24 Response of Low pass Filter

Practical FIR designs typically consist of filters that have a transition width and maximum pass band and stop band ripples that do not exceed allowable values. In addition to those design specifications, one must select the filter order, or, equivalently, the length of the truncated impulse response.

FIR filters are very attractive because they are inherently stable and can be designed to have linear phase. Nonetheless, these filters can have long transient responses and might prove computationally expensive in certain applications.

Minimum-Order FIR Filter Design

Minimum-order designs are obtained by specifying pass band and stop band frequencies as well as a pass band ripple and a stop band attenuation. The design algorithm then chooses the minimum filter length that complies with the specifications.

Design a minimum-order low pass FIR filter with a pass band frequency of 0.37*pi rad/sample, a stop band frequency of 0.43*pi rad/sample (hence the transition width equals 0.06*pi rad/sample), a pass band ripple of 1 dB and a stop band attenuation of 30 dB.

```
Fpass = 0.37;
Fstop = 0.43;
Ap = 1;
Ast = 30;


d = designfilt('lowpassfir','PassbandFrequency',Fpass,...
  'StopbandFrequency',Fstop,'PassbandRipple',Ap,'StopbandAttenuation',Ast);


hfvt = fvtool(d);
```

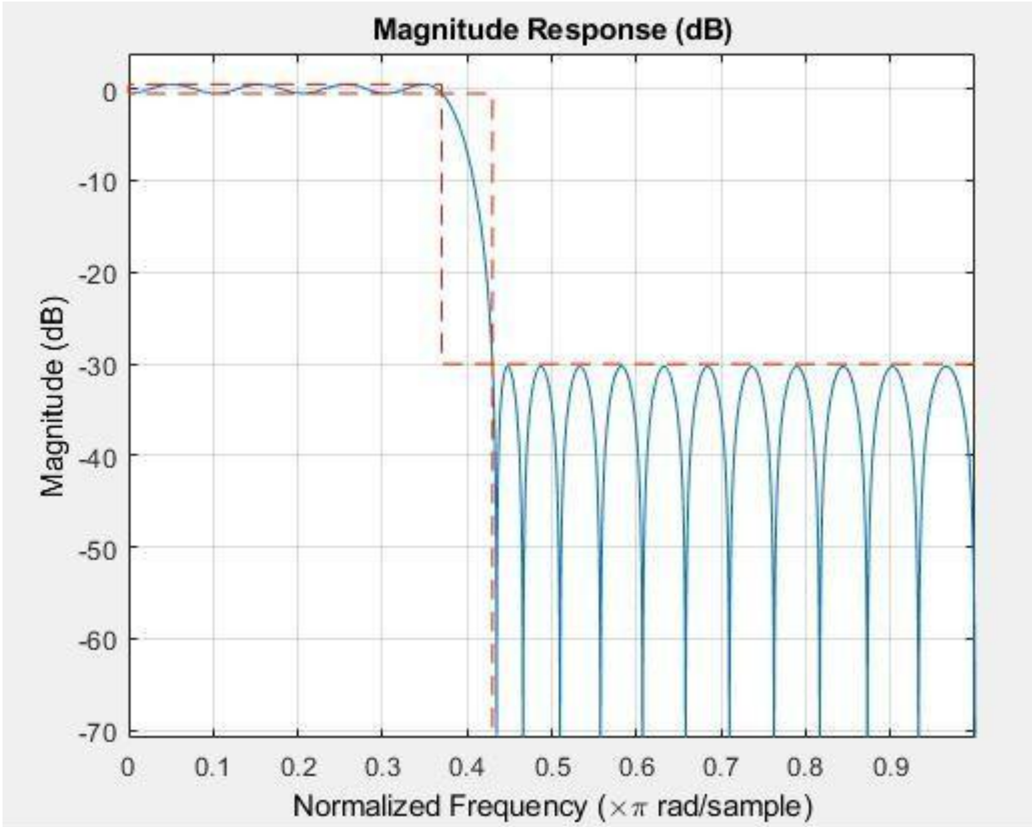


Figure 5.25 Response of FIR Low pass Filter

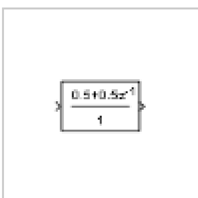
Discrete FIR FIlter

Library:

Simulink / Discrete

HDL Coder / Discrete

HDL Coder / HDL Floating Point Operations



Description

The Discrete FIR Filter block independently filters each channel of the input signal with the specified digital FIR filter. The block can implement static filters with fixed coefficients, and time-varying filters with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. The **Input processing** parameter allows you to specify whether the block treats each element of the input as an independent channel (sample-based processing), or each column of the input as an independent channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox™ license.

IIR Filter Design

One of the drawbacks of FIR filters is that they require a large filter order to meet some design specifications. If the ripples are kept constant, the filter order grows inversely proportional to the transition width. By using feedback, it is possible to meet a set of design specifications with a farsmaller filter order. This is the idea behind IIR filter design. The term "infinite impulse response"(IIR) stems from the fact that, when an impulse is applied to the filter, the output never decays tozero.

IIR filters are useful when computational resources are at a premium. However, stable, causal IIR filters cannot have perfectly linear phase. Avoid IIR designs in cases where phase linearity is a requirement.

Another important reason for using IIR filters is their small group delay relative to FIR filters, which results in a shorter transient response.

Butterworth Filters

Butterworth filters are maximally flat IIR filters. The flatness in the pass band and stop band causes the transition band to be very wide. Large orders are required to obtain filters with narrow transition widths.

Design a minimum-order Butterworth filter with pass band frequency 100 Hz, stop band frequency 300 Hz, maximum pass band ripple 1 dB, and 60 dB stop band attenuation. The sample rate is 2 kHz.

```
Fp = 100;
```

```
Fst = 300;
Ap = 1;
Ast = 60;
Fs = 2e3;

dbutter = designfilt('lowpassiir','PassbandFrequency',Fp,...
  'StopbandFrequency',Fst,'PassbandRipple',Ap,...
  'StopbandAttenuation',Ast,'SampleRate',Fs,'DesignMethod','butter');

hfvt = fvtool(dbutter,dcheby1,dcheby2,dellip);
axis([0 1e3 -80 2]);
legend(hfvt,'Butterworth', 'Chebyshev Type I',...
  'Chebyshev Type II','Elliptic')
```

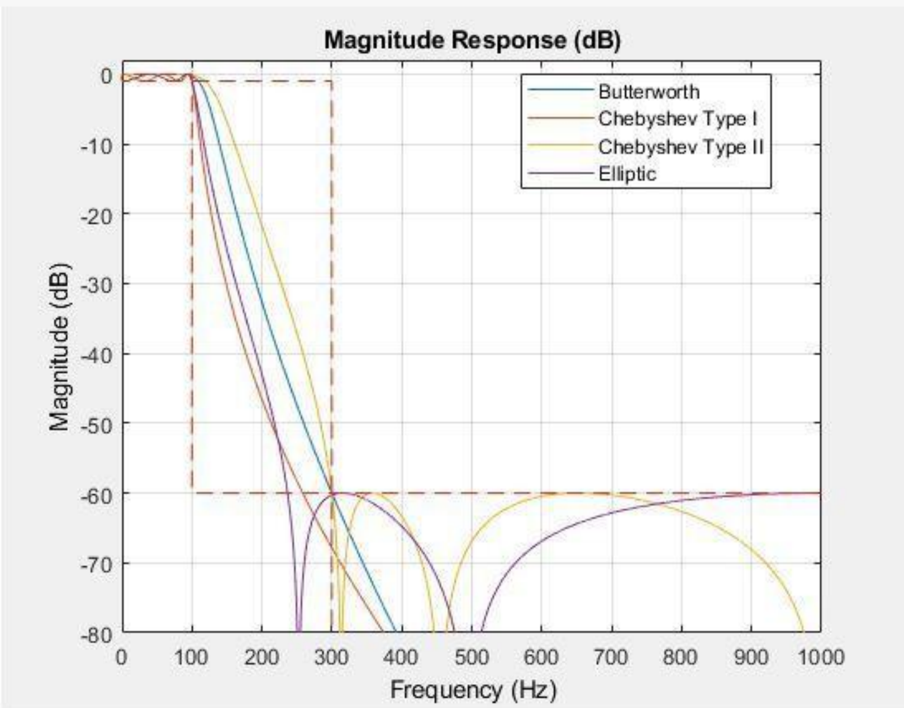


Figure 5.26 Response of IIR Low pass Filter

Program controls in Simulink

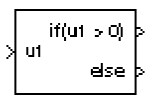
If

Implement a C-like if-else control flow statement in Simulink

Library

Ports & Subsystems

Description



The If block, along with If Action subsystems containing Action Port blocks, implements standard C-like if-else logic.

The following shows a completed if-else control flow statement.

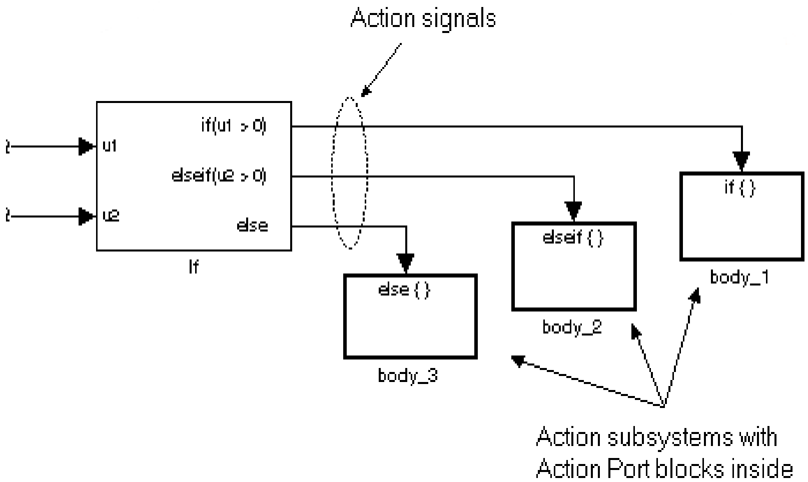


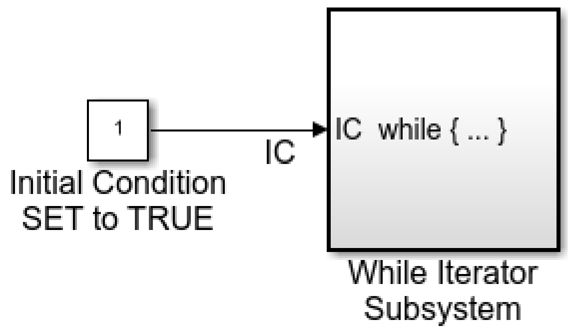
Figure 5.27 Implementation of If-Else using Simulink

In this example, the inputs to the If block determine the values of conditions represented as output ports. Each output port is attached to an If Action subsystem. The conditions are evaluated top down starting with the if condition. If a condition is true, its If Action subsystem is executed and the If block does not evaluate any remaining conditions.

Modeling Pattern for While Loop: While Iterator Subsystem block

One method for creating a while loop is to use a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library.

1. Open example model ex_while_loop_SL.



The model contains a While Iterator Subsystem block that repeats execution of the contents of the subsystem during a simulation time step.

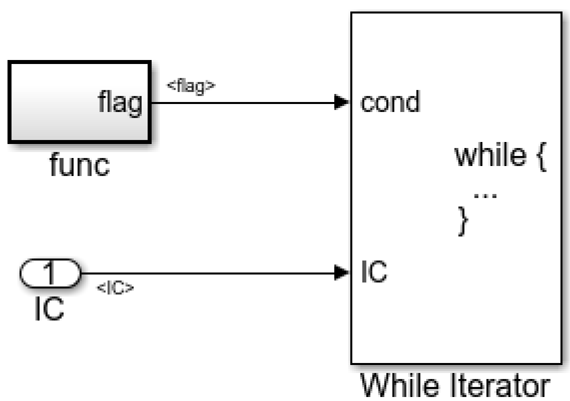


Figure 5.28 Implementation of While loop using Simulink

Observe the following settings in the model:

The Constant block provides an initial condition to the While Iterator Subsystem. For the Constant block, the **Constant value** is 1 and the **Output data type** is boolean. The initial condition can be dependent on the input to the block.

In the While Iterator Subsystem, the func subsystem block has an output flag of 0 or 1 depending on the result of the algorithm in func( ). func() is the **Function name** in func subsystem.

In the While Iterator Subsystem, for the While Iterator block, the **Maximum number of iterations** is 100.

For the While Iterator block, the **While loop type** is while.

Modeling Pattern for For Loop: For-Iterator Subsystem block

One method for creating a for loop is to use a For Iterator Subsystem block from the **Simulink > Ports and Subsystems** library.

1. Open example model ex_for_loop_SL.

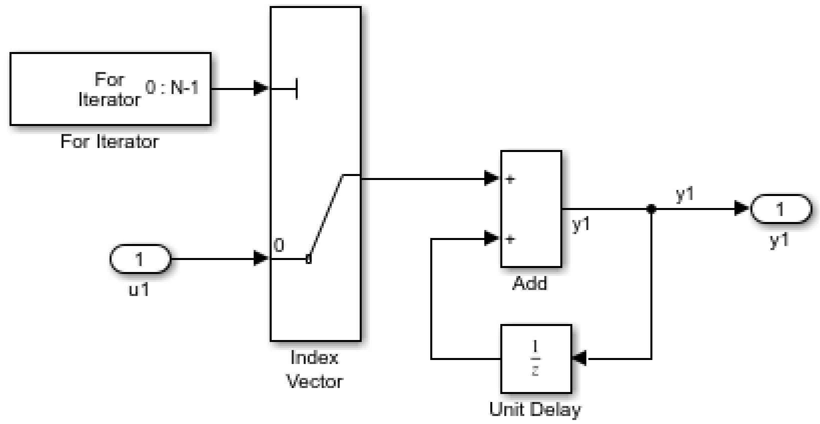


Figure 5.29 Implementation of for loop using Simulink

The model contains a For Iterator Subsystem block that repeats execution of the contents of the subsystem during a simulation time step.

Observe the following settings in the model:

Open the For Iterator block. In the Block Parameters dialog box, the **Index-mode** parameter is Zero-based and the **Iteration limit** parameter is 10.

Open the Unit Delay block. In the Block Parameters dialog box, the **Initial Conditions** parameter is 0. This parameter initializes the state to zero.

Table 5.1 Signal Routing

Bus Assignment	Replace specified bus elements
Bus Creator	Create bus from input elements
Bus Selector	Select elements from incoming bus
Data Store Memory	Define data store
Data Store Read	Read data from data store
Data Store Write	Write data to data store
Demux	Extract and output elements of virtual vector signal

Environment Controller	(Removed) Create branches of block diagram that apply only to simulation or only to code generation
From	Accept input from Goto block
Goto	Pass block input to From blocks
Goto Tag Visibility	Define scope of Goto block tag
Index Vector	Switch output between different inputs based on value of first input
Manual Switch	Switch between two inputs
Manual Variant Sink	Switch between multiple variant choices at output
Manual Variant Source	Switch between multiple variant choices at input
Merge	Combine multiple signals into single signal
Multiport Switch	Select output signal based on control signal
Mux	Combine input signals of same data type and complexity into virtual vector
Parameter Writer	Write to a model instance parameter
Selector	Select input elements from vector, matrix, or multidimensional signal
State Reader	Read a block state
State Writer	Write to a block state
Switch	Combine multiple signals into single signal
Variant Sink	Route amongst multiple outputs using Variants
Variant Source	Route among multiple inputs using Variants
Vector Concatenate, Matrix Concatenate	Concatenate input signals of same data type to create contiguous output signal

Export Simulink data to MATLAB

You can use a **To Workspace** (Simulink) block, from the DSP System Toolbox™/Sinks library to send data to the MATLAB workspace as a vector. For example, you can send the error rate data from the Hamming code model, described in the section Reducing the Error Rate Using a Hamming Code. To insert a To Workspace (Simulink) block into the model, follow these steps:

To open the model, at the MATLAB prompt, enter doc_hamming.

To add a To Workspace (Simulink) block, begin typing the name 'to workspace' in the model window and select the To Workspace block from the DSP System Toolbox/Sinks library. Connect it as shown.

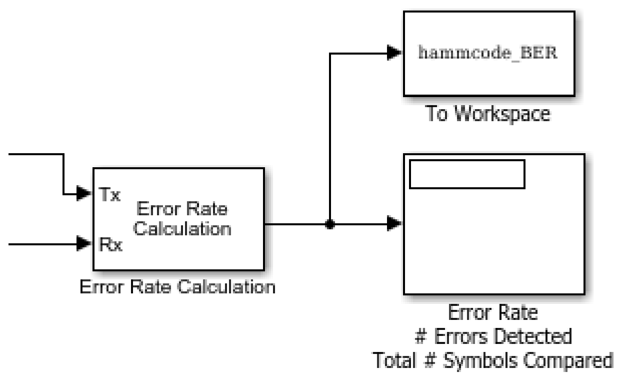


Figure 5.30 model for exporting data to workspace

Configure the To Workspace Block

To configure the To Workspace (Simulink) block, follow these steps:

- Double-click the block to display its dialog box.
- Type hammcode_BER in the **Variable name** field.
- Type 1 in the **Limit data points to last** field. This limits the output vector to the values at the final time step of the simulation.
- Click **OK**.

REFERENCES

- Stephen J Chapman, " Programming in MATLAB for Engineers", Brooks, 2002
- Duane Hanselman ,Bruce LittleField, "Mastering MATLAB 7" , Pearson Education Inc, 2005
- William J.Palm, "Introduction to MATLAB 6.0 for Engineers", Mc Graw Hill & Co, 2001
- M.Herniter, "Programming in MATLAB", Thomson Learning, 2001
- www.mathworks.com

PART A	CO
1. What is Simulink?	5
2. Mention the advantages of Simulink.	5
3. List the applications of Simulink.	5
4. How to create and run Simulink?	5
5. Name the command used for opening Simulink window in the command window	
6. What is a subsystem?	5
7. Infer the significance of creating a subsystem.	5
8. Give the design procedure of any simple application in Simulink	5
9. What is the use of to workspace module?	5
10. List the standard test signals available in Simulink	5

PART B

1. Create a Simulink application to generate PID Controller response	5
2. How to create a subsystem? Explain with suitable example.	5
3. How to formulate if , for and while loop using Simulink	5
4. Design a 2nd order system in Simulink and plot its open and close loop response.	
5. Develop FIR filter and obtain the frequency response using MATLAB	
6. Develop IIR filter and obtain frequency response using MATLAB	