LAB MANUAL

Simulation and Modeling LAB

B.Tech– VI Semester

KCT College OF ENGG AND TECH.

VILLAGE FATEHGARH

DISTT.SANGRUR

INDEX

Experiment 1. Introduction to MATLAB

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:

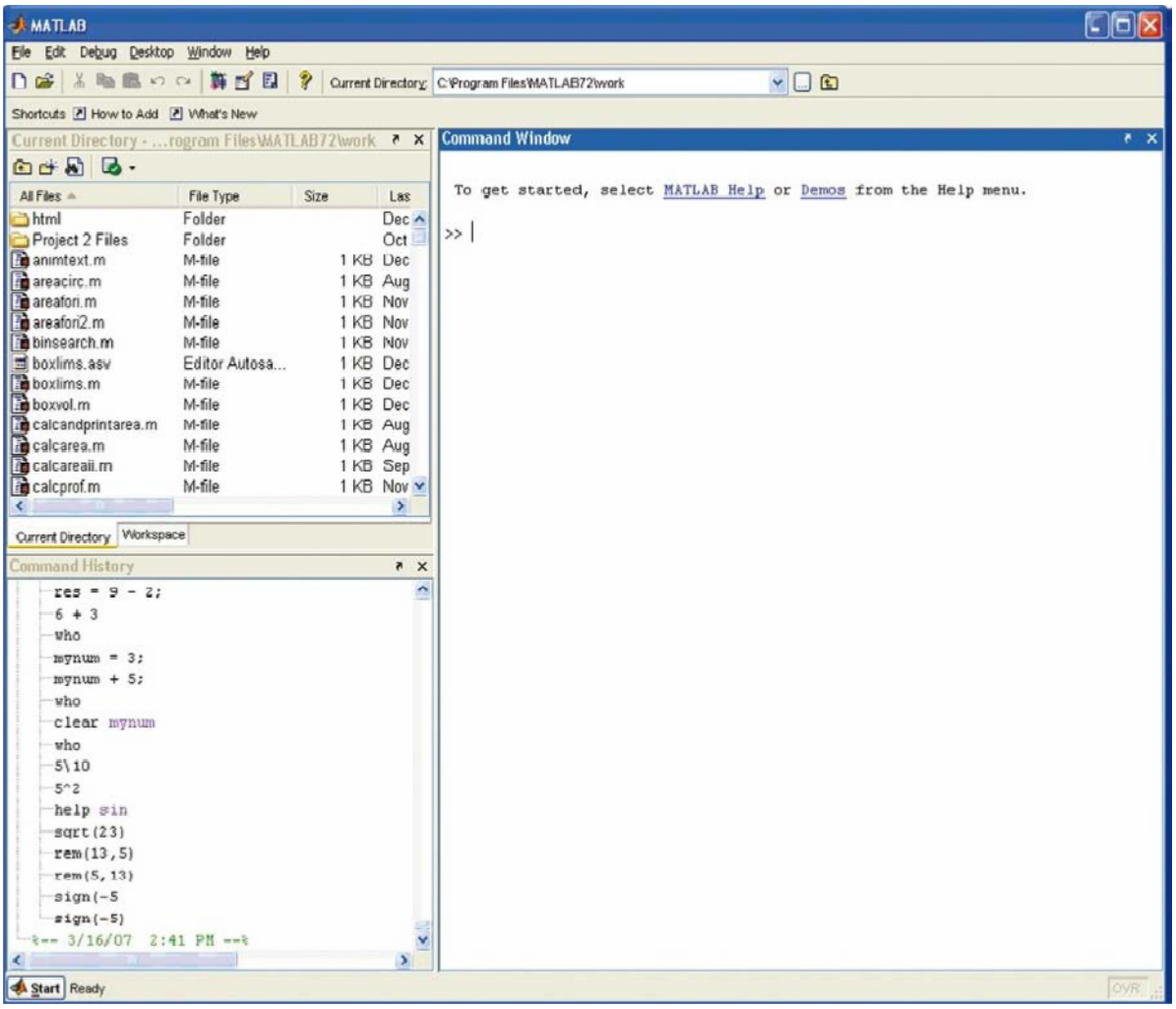
>>

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

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 that can 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 is to 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 way it 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.

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 identifier names, such as filenames, in future chapters. The rules for identifier names are:

1. 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.
2. There is a limit to the length of the name; the built-in function **namelengthmax** tells how many characters this is.
3. MATLAB is case-sensitive. That means that there is a difference between upper- and lowercase letters. So, variables called *mynum*, *MYNUM*, and *Mynum* are all different.
4. There are certain words called ***reserved words*** that cannot be used as variable names.
5. 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.

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
```

Operators:
There are in general two kinds of operators: ***unary*** operators, which operate on a single value or ***operand***; and ***binary*** operators, which operate on two values or operands. The symbol "–", for example, is both the unary operator for negation and the binary operator for subtraction.
Here are some of the common operators that can be used with numeric expressions:
+ addition
– negation, subtraction
* multiplication
/ division (divided by e.g. 10/5 is 2)
\ division (divided into e.g. 5\10 is 2)
^ exponentiation (e.g., 5^2 is 25)

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….
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.
1. 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. 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
```

2. 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.
3.The type **logical** is used to store true/false values.

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. Here are some diagrams showing, from left to right, a scalar, a column vector, a row vector, and a matrix:

5

3
7
4

5	88	3	11

9	6	3
5	7	2
4	33	8

The scalar is 1 × 1, the column vector is 3 × 1 (3 rows by 1 column), the row vector is 1 × 4 (1 row by 4 columns), and the matrix is 3 × 3. 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." For this 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.

MAT LAB Scripts:

A script is a sequence of MATLAB instructions that is stored in a file and saved. The contents of a script can be displayed in the Command Window using the **type** command. The script can be executed, or run, by simply entering the name of the file (without the .m extension).

To create a script, click File, then New, then M-file. A new window will appear called the Editor. To create a new script, simply type the sequence of statements (notice that line numbers will appear on the left). When finished, save the file using File and then Save. Make

sure that the extension .m is on the filename (this should be the default). The rules for filenames are the same as for variables (they must start with a letter, after that there can be letters, digits, or the underscore, etc.). By default, scripts will be saved in the Work Directory. If you want to save the file in a different directory, the Current Directory can be changed.
For example, we will now create a script called script1.m that calculates the area of a circle. It assigns a value for the radius, and then calculates the area based on that radius.

script1.m

```
radius = 5
area = pi * (radius^2)
```

In the Command Window, the contents of the script can be displayed, and the script can be executed. The **type** command shows the contents of the file named script1.m (notice that the .m is not included):

```
>> type script1
radius = 5
area = pi * (radius^2)
```

There are two ways to view a script once it has been written: either open the Editor Window to view it, or use the **type** command as shown here to display it in the Command Window.
To actually run or execute the script, the name of the file is entered at the prompt (again, without the .m). When executed, the results of the two assignment statements are displayed, since the output was not suppressed for either statement.

```
>> script1
radius =
5
area =
78.5398
```

Once the script has been executed, you may find that you want to make changes to it (especially if there are errors!). To edit an existing file, there are several methods to open it. The easiest are:

- Click File, then Open, then click the name of the file.
- Click the Current Directory tab (if it is not already shown), then doubleclick the name of the file.

Documentation:

It is very important that all scripts be ***documented*** well, so that people can understand what the script does and how it accomplishes that. One way of documenting a script is to put ***comments*** in it. In MATLAB, a comment is anything from a % to the end of that particular line. Comments are completely ignored when the script is executed.
script1b.m

```
% This program calculates the area of a circle
% First the radius is assigned
radius = 5
% The area is calculated based on the radius
area = pi * (radius^2)
```

Experiment2: Various types of data types and Input output functions in MATLAB.

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
```

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 it specifies 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.

Experiment3: Programming of Selection statements in MATLAB.

Relational Expressions:
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.
1.The relational operators in MATLAB are:
> greater than
< less than
>= greater than or equals
<= less than or equals
== equality
= inequality

2.The logical operators are:

|| or for scalars
&& and for scalars
~ not

1. 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
```

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
```

Eg 2:

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

In the above example, 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.

2. 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 of statements.
The general form is:

```
if condition
action1
else
action2
end
```

1.For example, to determine and print whether or not 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:
Eg 2: 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
area = calcarea(radius);
fprintf('For a circle with a radius of %.2f,',radius)
fprintf('the area is %.2f\n',area)
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

3. 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 ≤ 2
y = 4 for x > 2
The value of y is based on the value of x, which could be in one of three possible ranges. Choosing which 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 expressions in the next two **if** statements are still evaluated. Instead of writing it this way, the expressions can be ***nested*** so that the statement ends when an expression is found to be true:

```
if x < −1
y = 1;
else
% 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
```

4. 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 switchexpression
case caseexp1
action1

case caseexp2

 action2

case caseexp3
action3
----

otherwise
 actionn
end
```

For 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 > 10
grade = ‘X’;
else
% If here, it is valid so figure out the
% corresponding letter grade using a switch
switch quiz
‡case 10
grade = ‘A’;
case 9
grade = ‘A’;
case 8
grade = ‘B’;
case 7
grade = ‘C’;
case 6
grade = ‘D’;
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
```

Experiment4: Programming of looping statements in MATLAB.

Looping statements that allow other statement(s) to be repeated. The statements that do this are called looping statements, or loops.

There are two basic kinds of loops in programming: counted loops, and conditional loops. A counted loop is one that repeats statements a specified number of times (e.g., ahead of time it is known how many times the statements are to be repeated). In a counted loop, for example, you might say "repeat these statements 10 times." A conditional loop also repeats statements, but ahead of time it is not known how many times the statements will need to be repeated. With a conditional loop, for example, you might say "repeat these statements until this condition becomes false." The statement(s) that are repeated in any loop are called the action of the loop.

There are two different loop statements in MATLAB: the for statement and the while statement.

1.The for **Loop:**

The **for** statement, or the **for** loop, is used when it is necessary to repeat statement(s) in a script or 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 general form of the **for** loop is:

```
for loopvar = range
action
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
```

2.For eg: Finding sum and product

sum_1_to_n.m

```
function runsum = sum_1_to_n(n)
% This function returns the sum of
% integers from 1 to n
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
```

2.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 script that 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:

```
*****
*****
*****
```

printstars.m

```
% Prints a box of stars
% How many will be specified by 2 variables
% for the number of rows and columns
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:columns
fprintf('*')
end
fprintf('\n')
end
```

Running the script displays the output:

```
>> printstars
*****
*****
*****
```

2.For eg: To print a triangle stars

printtristars.m

```
% Prints a triangle of stars
% How many will be specified by a variable
% for the number of rows
rows = 3;
for i=1:rows
% inner loop just iterates to the value of i
for j=1:i
fprintf('*')
end
fprintf('\n')
end
```

>> printtristars
*
**
***

3.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

For eg: 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
```

Experiment 5: Programming of Functions and plot functions in MATLAB.

User-Defined Functions that Return a Single Value: These are functions that the programmer defines, and then uses, in either the Command Window or in a script. Functions can return different types of results. For now, we will concentrate on the kind of function that calculates and returns a single result, much like builtin functions such as **sin** and **abs**. The **length** function is an example of a built-in function that calculates a single
value; it returns the length of a vector. As an example,

length(vec) is an expression; it represents the number of elements in the vector *vec*. This expression could be used in the Command Window or in a script. Typically,
the value returned from this expression might be assigned to a variable:

```
>> vec = 1:3:10;
>> lv = length(vec)
lv =
4
```

Alternatively, the length of the vector could be printed

```
>> fprintf('The length of the vector is %d\n', length(vec))
The length of the vector is 4
```

Function Definitions:
There are different ways to organize scripts and functions, but for now every function that we write will be stored in a separate M-file, which is why they are commonly called M-file functions.
A function in MATLAB that returns a single result consists of

- The function header (the first line); this has

– the reserved word function
– since the function returns a result, the name of the output argument followed by the assignment operator =
– the name of the function (Important: This should be the same as the name of the M-file in which this function is stored in order to avoid confusion)
– the input arguments in parentheses; these correspond to the arguments that are passed to the function in the function call

- A comment that describes what the function does (this is printed if help is used)
- The body of the function, which includes all statements and eventually must assign a value to the output argument

The general form of a function definition for a function that calculates and
returns one value looks like this:
functionname.m

```
function outputargument = functionname(input arguments)
% Comment describing the function
Statements here; these must include assigning a value to
the output argument
```

For example, the following is a function called calcarea, which calculates and returns the area of a circle; it is stored in a file called calcarea.m.
For eg: calcarea.m

```
function area = calcarea(rad)
% This function calculates the area of a circle
area = pi * rad * rad;
```

The function can be displayed in the Command Window using the type command.

```
>> type calcarea
function area = calcarea(rad)
% This function calculates the area of a circle
area = pi * rad * rad;
```

Calling a Function:
Here is an example of a call to this function in which the value returned is stored in the default variable ans:

```
>> calcarea(4)
ans =
50.2655
```

Technically, calling the function is done with the name of the file in which the function resides. In this example, the function name is *calcarea* and the name of the file is calcarea.m. The result returned from this function can also be stored in a variable in an assignment statement; the name could be the same as the name of the output argument in the function itself but that is not necessary; for example, either of these assignments would be fine:

```
>> area = calcarea(5)
area =
78.5398
>> myarea = calcarea(6)
myarea =
113.0973
```

The value returned from the calcarea function could also be printed using either **disp** or **fprintf**:

```
>> disp(calcarea(4))
50.2655
>> fprintf('The area is %.1f\n', calcarea(4))
The area is 50.3
```

Notice that the printing is not done in the function itself; rather, the function returns the area and then a print statement can print or display it. Using **help** with the function displays the contiguous block of comments under the function header:

>> help calcarea
This function calculates the area of a circle

Calling a User-Defined Function from a Script:
Now, we'll modify our script that prompts the user for the radius and calculates the area of a circle, to call our function *calcarea* to calculate the area of the circle rather than doing this in the script.

For eg: script3.m

```
% This script calculates the area of a circle
% It prompts the user for the radius
radius = input('Please enter the radius:');
% It then calls our function to calculate the
% area and then prints the result
area = calcarea(radius);
fprintf('For a circle with a radius of %.2f,',radius)
fprintf('the area is %.2f\n',area)
```

So, the program consists of the script *script3* and the function *calcarea*.
Running this will produce the following:
>> script3
Please enter the radius: 5
For a circle with a radius of 5.00, the area is 78.54

The Plot Function:

For now, we'll start with a very simple graph of one point using the plot function. The following script, plotonepoint, plots one point. To do this, first values are given for the x and y coordinates of the point in separate variables. The point is then plotted using a red*. The plot is then customized by specifying the minimum and maximum values on first the x- and then y-axis. Labels are then put on the x-axis, the y-axis, and the graph itself using the function xlabel, ylabel, and title. All this can be done from the Command Window, but it is much easier to use a script. The following shows the contents of the script plotonepoint that accomplishes this. The x-coordinate represents the time of day (e.g., 11am) and the y-coordinate represents the temperature in degrees Fahrenheit at that time:
For eg: plotonepoint.m

```
% This is a really simple plot of just one point!
% Create coordinate variables and plot a red '*'
x = 11;
y = 48;
plot(x,y,'r*')
% Change the axes and label them
axis([9 12 35 55])
xlabel('Time')
ylabel('Temperature')
```

```
% Put a title on the plot
title('Time and Temp')
```

In the call to the **axis** function, one vector is passed. The first two values are the minimum and maximum for the x-axis, and the last two are the minimum and maximum for the y-axis. Executing this script brings up a Figure Window with the plot (see Figure 2.1). To be more general, the script could prompt the user for the time and temperature, rather than just assigning values. Then, the axis function could be used based on whatever the values of x and y are, for example, axis([x–2 x+2 y–10 y+10]) In order to plot more than one point, x and y vectors are created to store the values of the (x,y) points. For example, to plot the points
(1,1)
(2,5)
(3,3)
(4,9)
(5,11)
(6,8)
first an *x* vector is created that has the x values (since they range from 1 to 6 in steps of 1, the colon operator can be used) and then a *y* vector is created with the y values. This will create (in the Command Window) x and y vectors and then plot them

```
>> x = 1:6;
>> y = [1 5 3 9 11 8];
>> plot(x,y)
```

Notice that the points are plotted with straight lines drawn in between. Also, the axes are set up according to the data; for example, the x values range from 1 to 6 and the y values from 1 to 11, so that is how the axes are set up. Also, notice that in this case the x values are the indices of the y vector (the y vector has six values in it, so the indices iterate from 1 to 6). When this is the case, it is not necessary to create the x vector.

For example, >> plot(y) will plot exactly the same figure without using an *x* vector.

Simple Related Plot Functions:
Other functions that are useful in customizing plots are **clf**, **figure**, **hold**, **legend**, and **grid**.
1. use **help** to find out more about them
2. **clf** clears the Figure Window by removing everything from it.
3. **figure** creates a new, empty Figure Window when called without any arguments. Calling it as **figure(n)** where n is an integer is a way of creating and maintaining multiple Figure Windows, and of referring to each individually.
4.**hold** is a toggle that freezes the current graph in the Figure Window, so that new plots will be superimposed on the current one. Just **hold** by itself is a **toggle**, so calling this function once turns the hold on, and then the next time turns it off. Alternatively, the commands **hold on** and **hold off** can be used.
5.**legend** displays strings passed to it in a legend box in the Figure Window, in order of the plots in the Figure Window.
6.**grid** displays grid lines on a graph. Called by itself, it is a toggle that turns the grid lines on and off. Alternatively, the commands **grid on** and **grid off** can be used.

Experiment 6: Arrays in MATLAB.

Cell Array: A ***cell array*** is a kind of data structure that stores values of different types. Cell arrays can be vectors or matrices; the different values are stored in the elements of the array. One very common use of a cell array is to store strings of different lengths.

Creating Cell Arrays

There are several ways to create cell arrays. For example, we will create a cell array in which one element will store an integer, one element will store a character, one element will store a vector, and one element will store a string. Just as with the arrays we have seen so far, this could be a 1 4 row vector, a 4 1 column vector, or a 2 2 matrix. The syntax for creating vectors and matrices is the same as before. Values within rows are separated by spaces or commas, and rows are separated by semicolons. However, for cell arrays, curly braces are used rather than square brackets.

For example, the following creates a row vector cell array with the four different types of values:

```
>> cellrowvec = {23, ‘a’, 1:2:9, ‘hello’}
cellrowvec =
[23] ‘a’ [1x5 double] ‘hello’
```

To create a column vector cell array, the values are instead separated by semicolons:

```
>> cellcolvec = {23; ‘a’; 1:2:9; ‘hello’}
cellcolvec =
[ 23]
‘a’
[1x5 double]
‘hello’
```

This method creates a 2 2 cell array matrix:

```
>> cellmat = {23 ‘a’; 1:2:9 ‘hello’}
cellmat =
[ 23] ‘a’
[1x5 double] ‘hello’
```

Another method of creating a cell array is simply to assign values to specific array elements and build it up element by element. However, as explained before, extending an array element by element is a very inefficient and time-consuming method. It is much more efficient, if the size is known ahead of time, to preallocate the array. For cell arrays, this is done with the **cell** function. For example, to preallocate a variable *mycellmat* to be a 2 2 cell array, the **cell** function would be called as follows:

```
>> mycellmat = cell(2,2)
mycellmat =
[] []
[] []
```

Note that this is a function call so the arguments to the function are in parentheses. This creates a matrix in which all the elements are empty vectors. Then, each element can be replaced by the desired value. How to refer to each element in order to accomplish this will be explained next.

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

```
>> celldisp(cellrowvec)
cellrowvec{1} =
23
cellrowvec{2} =
a
cellrowvec{3} =
1 3 5 7 9
cellrowvec{4} =
hello
```

Experiment 7: Introduction regarding usage of any Network Simulator.

Network Simulator version 2 (NS-2) is a free and open source discrete event network simulator developed at UC Berkeley You can add your own protocol, contribute to the code and, from time to time, you need to troubleshoot some of the bugs NS is a discrete event simulator where the advance of time depends on the timing of events which are maintained by a scheduler. NS-2 works under Linux, Mac, and Windows. Current release is ns-2.31. Release under work: ns-3 where Inria takes part of the development process.

NS-2 has a large and rich library of network and protocol objects. It covers a large part of applications (Web, FTP, CBR,. . .), protocols (transport and routing protocols), network types (Satellite links, wired and wireless LAN), network elements (mobile nodes, wireless channel models, link and queue models,. . .) and traffic models (exponential, uniform, . . .). NS also allows to add and test new protocols and applications and/or to modify existing ones. NS-2 is based on an object oriented simulator written in C++ and a OTcl interpreter (an object oriented extension of Tool Command Language TCL).
These different objects are written in C++ code in order to achieve efficiency in the simulation and faster execution times. (e.g.

Steps of a NS simulation:
Define the scenario to simulate:
1. Create the simulator object
2. { Turn on tracing }
3. Setup the network nodes {and links }
4. Setup the routing mechanism
5. Create transport connections
6. Setup user applications
7. Schedule data transmission
8. Stop the simulation
Execute the OTcl script in a Linux shell: > ns example.tcl
Extract the results from the trace files: awk, xgraph, nam,
matlab, etc .

Experiment 8: Practical Implementation of Queuing Models using C/C++.

SIMULATION OF A SINGLE SERVER QUEUE:

we will construct a simulation model for single queue and a single server, say a machine shop. In a real life dynamic system, time flow is an essential element. Whether it is a simulation model of queuing system, manufacturing system of inventory control, many parameters in these are function of time. Thus time flow mechanism is an essential part in a simulation model. There of two basic ways of incrementing time in a simulation model as.

(*a*) **Fixed Time Increment:** In fixed time increment model, also called **Time Oriented Simulation**, events are recorded after a fixed interval of time, which is constant during the simulation period. After the end of each interval, it is noted how many customers have arrived in a queue, and how many have left the server after being served. Attempt in this system is to keep time interval as small as possible, so that minor details of model are monitored. Possible in one time interval, only one customer arrives and only one leaves. Fixed time increment simulation is generally preferred for continuous simulation. Numerical methods, where time is taken as independent variable are one such example.

(*b*) **Next Event Increment Simulation:** This method is also called **Event Oriented Simulation**. In this system, time is incremented when an event occurs. For example in queuing, when a customer arrives, clock is incremented by his arrival time. In such case time period for simulation may be stochastic.

Program : Single Queue Simulation

```
// Single queue simulation
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>//contains rand() function
#include <math.h>
#include <conio.h>
#include<iomanip.h>
void main(void)
{ /* Single server queue:
arrival and service times are normally distributed.
mean and standard deviation of arrivals are 10 and 1.5 minutes.
mean and standard deviation of service times are 9.5 and 1.0 */
int i,j,run = 10;
double x,x1,x2, st, awt, pcu, wt=0, iat=0,it;
double mean=10., sd=1.5, mue=9.5, sigma=1.0;
double sb=0.,se=0.,cit=0.,cat=0.,cwt=0.;
ofstream outfile("output.txt",ios::out);
outfile<<"\n i r ' IAT CAT SB r' ST SE WT IT\n";
for (j = 1; j <= run; ++ j)
{
//Generate inter arrival time
double sum=0;
```

```
for (i=1; i < = 12; ++i)
{
x = rand()/32768.0;
sum=sum+x;
}
x1=mean+sd*(sum–6.);
iat= x1;
//cout<<“iat=”<<iat;
cat=cat+iat;
//cout<<“cat=”<<cat;
if(cat<=se)
{
sb=se;
wt=se–cat;
cwt=cwt+wt;
// cout<<“cwt=”<<cwt;
}
Else
{
sb=cat;
it=sb-se;
cit=cit+it;
}
//generate service time
sum=0.;
for(i=1; i<=12;++i)
{x=rand()/32768.;
sum=sum+x;
x2=mue+sigma*(sum-6.);
st=x2;
se=sb+st;
}
outfile<<j<<‘\t’<<setprecision(4)<<x1<<‘\t’<<setprecision(4)<<iat<<‘\t’<<setprecision
(4)<<cat<<‘\t’<< setprecision (4)<<sb<<‘\t’<< setprecision (4)<<x2<<‘\t’<<
setprecision (4)<<st<<‘\t’<< setprecision(4)<<se<<‘\t’<<setprecision(4)
<<wt<<‘\t’<<setprecision (4)<<it<<“\n”;
}
awt=cwt/run;
pcu=(cat–cit)*100./cat;
outfile<<“Average waiting time\n”;
outfile<<awt;
outfile<<“Percentage capacity utilization\n”<<pcu;
}
```

Output of this program is given below. Ten columns are respectively number of arrival, *I*, random number *r′*, inter arrival time IAT, cumulative arrival time CAT, time at which service begins

SB, random number for service time *r′*, service time ST, time for service ending SE, waiting time WT, and idle time IT.

Output of the program:

1	2	3	4	5	6	7	8	9	10
i	*r′*	*IAT*	*CAT*	*SB*	*r′*	*ST*	*SE*	*WT*	*IT*
1	10.72	10.72	10.72	10.72	7.37	7.37	18.09	0	10.72
2	8.493	8.493	19.21	19.21	10.77	10.77	29.98	0	1.123
3	11.68	11.68	30.9	30.9	9.021	9.021	39.92	0	0.913
4	10.74	10.74	41.63	41.63	9.469	9.469	51.1	0	1.714
5	9.095	9.095	50.73	51.1	9.905	9.905	61.01	0.374	1.714
6	9.972	9.972	60.7	61.01	9.988	9.988	70.99	0.306	1.714
7	11.35	11.35	72.05	72.05	9.161	9.161	81.21	0.306	1.057
8	10.74	10.74	82.79	82.79	7.081	7.081	89.87	0.306	1.58
9	9.19	9.19	91.98	91.98	10.53	10.53	102.5	0.306	2.11
10	8.542	8.542	100.5	102.5	10.85	10.85	113.4	1.989	2.11

Simulation of Single Queue Multiple Servers:

Simulation of multiple server queue is very important seeing its application in day to day life. Let us consider a case of bank where there are two service counters. Customers arrive in the bank according to some probability distribution for arrival times. When a customer enters the bank, he checks whether a counter is free or not. If a counter is free, he will go to that counter, else he will stand in queue of one of the counter, preferably in a smaller queue increasing the queue length by one. Customer is attended at the service counter as per first come first served rule. The service time from each service counter can be viewed as independent sample from some
specified distribution. It is not necessary that inter arrival time or service time be exponential. Queuing system although simple to describe, is difficult to study analytically. In such cases simulation is the only alternative.

Program : Single Queue Two Servers Simulation

```
// Single queue two servers simulation
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>//contains rand() function
#include <math.h>
#include <conio.h>
```

```
#include<iomanip.h>
void main(void)
{ /* M/D/2/3 queuing system.*/
int k,q=0,qmax=3,count=0,counter;
double r, iat,clock=0., nat=0., wt2=0., wt1=0.,it1=0.,it2=0.,cit1=0.,
cit2=0.;
double mean=3.,lambda1=5.,lambda2=4.,se1=0.,se2=0.,run=150;
ofstream outfile("output.txt",ios::out);
outfile<<"\n CLOCK IAT NAT SE1 SE2 QUE COUNT
CIT1CIT2 \n";
// Generate first arrival
while (clock<=run){
//Check the state of arriv7al and update que
r = rand()/32768.0;
iat=(-mean)*log(1–r);
nat=nat+iat;
se1=lambda1;//Service time taken by first server
counter=1;//First customer has come counter=1

outfile.precision(4);
outfile<<clock<<'\t'<<iat<<'\t'<<nat<<'\t'<<se1<<'\t'<<se2<<'\t'<<q<<'\t'
<<count<<'\t'<<cit1<<cit2<<endl;
//it1 and it2 are idle times for two servers.
while(clock<=run)
{
if(nat<=se1 && nat<=se2){
clock=nat; q=q+1;
r = rand()/32768.0;
iat=(–mean)*log(1–r);
nat=nat+iat; counter=counter+1;
}
else if(se1<=nat && se1<=se2) clock=se1;
else clock=se2;
if (q>qmax){ count=count+1;
q=q–1;
}
if (q>=1 && se1<=clock) {
it1=clock–se1;
cit1=cit1+it1;
se1=clock+lambda1;
q=q–1;
}
if(q>=1 &&se2<=clock)
{
it2=clock–se2;
cit2=cit2+it2;
```

```
se2=clock+lambda2;
q=q–1;
}
if(q==0 && se1<=clock)
{
clock=nat;
it1=clock–se1;
cit1=cit1+it1;
se1=nat+lambda1;
se1=nat+lambda1;
r = rand()/32768.0;
iat=(–mean)*log(1–r);
nat=nat+iat;
counter=counter+1;
}
if(q==0 && se2<=clock)
{
clock=nat;
it2=clock–se2;
cit2=cit2+it2;
se2=nat+lambda2;
r = rand()/32768.0;
iat=(–mean)*log(1–r);
nat=nat+iat;
counter=counter+1;
}
outfile<<clock<<'\t'<<iat<<'\t'<<nat<<'\t'<<se1<<'\t'<<se2<<'
\ t ' < < q < < ' \ t ' < < c o u n t < < ' \ t ' < < c i t 1 < < ' \ t ' < < c i t 2 < < e n d l ;
}
outfile.precision(4);
outfile<<"clock="<<clock<<"cit1="<<cit1<<"cit2="<<cit2<<"counter="<<counter<<endl;
outfile<<"Queuing system M/D/2/3"<<endl;
outfile<<"Mean of the exponential distribution="<<mean<<endl;
outfile<<"service time of two servers="<<lambda1<<'\t'<<lambda 2<<endl;
outfile<<"Simulation run time="<<clock<<endl;
outfile<<"Number of customers arrived"<<counter<<endl;
outfile<<"Number of customers returned without service"<<count<<endl;
outfile<<"idle time of serverI\n"<<cit1<<endl;
outfile<<"idle time of server II\n"<<cit2<<endl;
outfile<<"Percentage idle time of serverI\n"<<cit1*100/clock<<endl;
outfile<<"Percentage idle time of serverII\n"<<cit2*100/clock<<endl;
}
cout<<"any number"<<endl;
cin>>k;
}
```