Laboratory Manual

14032401-4

Numerical Methods for Computing

Student Name: _____

Student ID: _____

Section: _____ Group: _____

Session (Spring / Summer / Fall) _____

Department of Computer Engineering
College of Computer and Information Systems
Umm Al-Qura University, Makkah, Saudi Arabia

This page is intentionally left blank

TABLE OF CONTENTS

This page is intentionally left blank

Experiment No. 1
Introduction to MATLAB

Objective:

The purpose of this tutorial is to acquaint the students with basics of MATLAB. The topics discussed in this lab includes Command Window, numbers and arithmetic operations, Relations and Logical Operations, some built-in functions, some Commands, The m – files, Vectors and Matrices loops and Conditional Statements.

Introduction:

The name MATLAB stands for MATrix LABoratory. MATLAB was written originally to provide easy access to matrix software developed by the LINPACK (linear system package) and EISPACK (Eigen system package) projects.

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming environment. Furthermore, MATLAB is a modern programming language environment: it has sophisticated data structures, contains built-in editing and debugging tools, and supports object-oriented programming. These factors make MATLAB an excellent tool for teaching and research.

MATLAB has many advantages compared to conventional computer languages (e.g., C, FORTRAN) for solving technical problems. MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. The software package has been commercially available since 1984 and is now considered as a standard tool at most universities and industries worldwide.

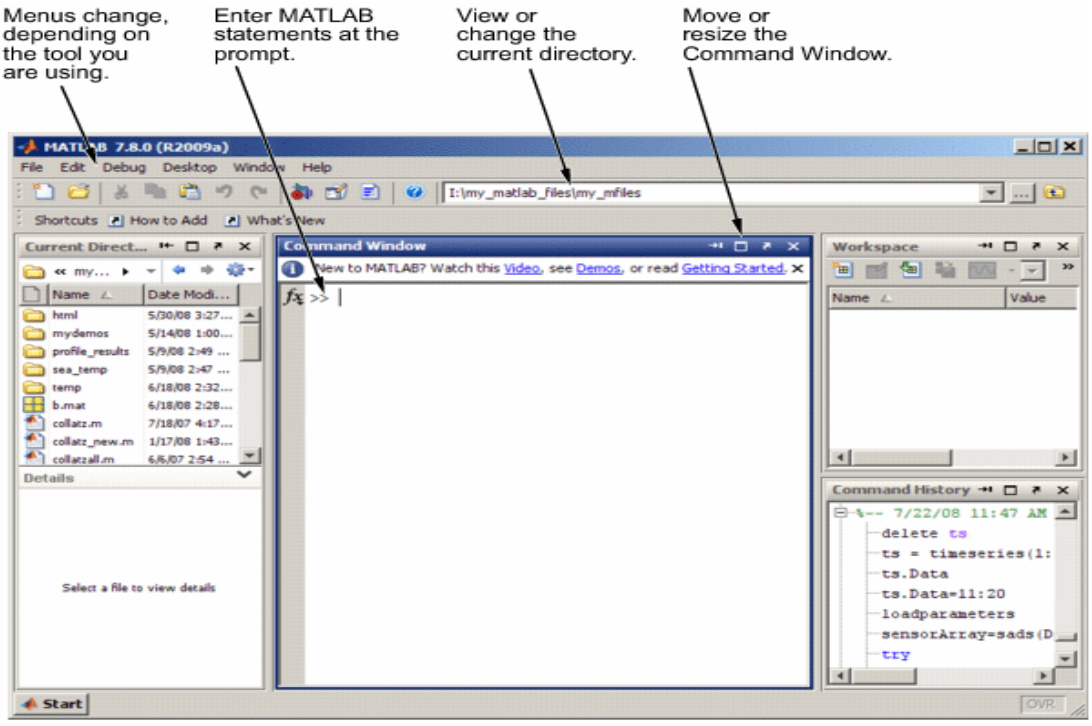


Figure 1. MATLAB Interface

When you launch MATLAB you see five basic windows:

- Command window
- Command history
- Current directory
- Workspace
- Help browser

The Command Window:

When you first open the MATLAB program a window appears which is called the Command Window.

- ✓ Used to type instructions of various types such as commands, functions and statements etc.
- ✓ MATLAB displays the prompt >>
  - to indicate that it is ready to receive instructions
  - type a command and next press the *Enter*

```
>> 4*5
ans =
    20
>> 20*5
ans =
    100
>> x = 2 + 2
x =
     4
>> y = 2+2* x
y =
    10
```

- ✓ *ans* stands for answer. *ans* now contains 100 (not 20).
- ✓ The result of each computation is saved in a variable whose name is chosen by the user.
- ✓ If any of the variables will be needed during your current MATLAB session, then you can obtain its value by typing it name and pressing the *Enter*.

To close MATLAB type ***exit*** in the Command Window and next press *Enter*.

Comments %

- ✓ The *percentage sign %* is used to indicate that line is a comment.
- ✓ It is best to include comments not only at the top of a program, but also with each section.
- ✓ For a script program it is often helpful to include the name of the program at the beginning.
- ✓ All comments are ignored by MATLAB.
- ✓ They are added to improve readability of the code.

Help Browser:

- ✓ MATLAB has thousands of built in functions and various capabilities. Therefore you should always search around for what has been done before writing a program.
- ✓ Searching in the Help window is usually the best way
- ✓ If you know the name of a MATLAB function but are not sure what it does, type in the command windown command

```
>> help nameoffunction
```

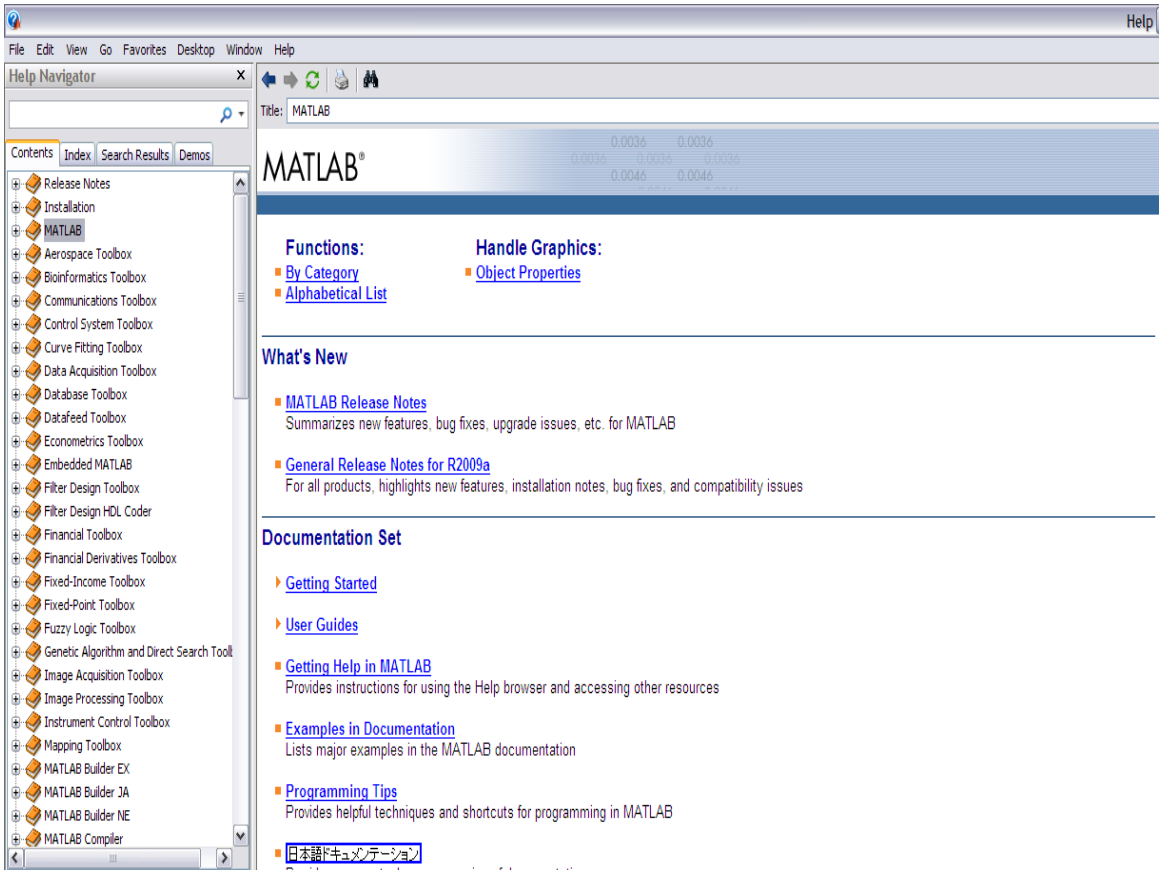


Figure 2. MATLAB Help

Numbers and Arithmetic Operations

There are three kinds of numbers used in MATLAB:

- ✓ integers
- ✓ real numbers
- ✓ complex numbers

We are concerned with integers and real numbers.

Examples:
Integer numbers:

```
xi = 10
xi =
        10
```

Real numbers:

```
xr = 10.01
xr =
        10.0100
```

List of basic arithmetic operations in MATLAB includes 6 operations

Operation	Symbol
addition	+
subtraction	-
Multiplication	*
Division	/ or \
exponentiation	^

MATLAB has two division operators: the right division(/) and the left division(\). you have to note that they do not produce the same results.

Example:

```
x = 6/2

x =
        3
```

Whereas,

```
x = 6\2

x =
        0.3333
```

- **Relations and Logical Operations**

Comparisons in MATLAB are performed with the aid of the following operators:

< less than, > greater than, >= greater than or equal to, <= less than or equal to, == equal, ~= not equal, & and, | or, ~ not

Note, == is a test of equality, 1 if yes, 0 if no. = is used in assignment.

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater
>=	Greater or equal to
==	Equal to
~=	Not equal to

There are three logical operators available in MATLAB:

Logical operator	Description
|	or
&	And
~	Not

- **Variables**

MATLAB has built-in variables and here below a list of some variables:

Variable	Meaning
pi	Π
exp(x)	e^x
eps	Epsilon
ans	Keeps track of the last output which was not assigned to another variable.
i, j	Imaginary unit
Inf	Stands for (+∞)
-Inf	Stands for (-∞)
NaN	Stands for not a number

- **Brief Description of some Commands and Functions Needed in our Course**

No.	Function	Purpose of use
1.	***Clear***	Clear variables and functions from memory.
2.	***conv(****p1,p2****)***	Multiply two polynomials *p1,p2*
3.	***det(****A****)***	Calculates the determinant of square matrix (*A*)
4.	***disp('…')***	Displays a matrix or a vector
5.	***eye(****n****)***	To generate an Identity matrix
6.	***feval(***'*fun*', *in1, in2,..***)**	Execute the function *fun* specified by string with the inputs passed *in1,in2, …* (if any)
7.	***format*** *FormatToUse*	To determine the format of the numbers displayed
8.	**help** *fun*	To learn more about the function named (*fun*) in detail.
9.	**inline(*'fun(in1,in2,..)' 'in1', 'in2', …*)**	To define a function that will be used during the current MATLAB session only.
10.	***input('prompt')***	Displays a prompt to the user and get input(s) from the user.
11.	***inv(****A****)***	Computes the inverse of the matrix (*A)* if (*A*) is invertible
12.	***Length(****V****)***	Returns the length of a vector *V*
13.	**lookfor** *fun*	To learn more about the function named (*fun*). This command is used when you do not remember the exact name of the function.
14.	***Ones(****n, m****)***	To generate Ones matrix (all entries are ones)
15.	***Plot(****x, y,…****)*** ***Title('…')*** ***xlabel('…')*** ***ylabel('…')*** ***grid***	Linear plot (2D plot). This function takes a variable number of input arguments. ***Title*** - Graph title. ***xlabel*** - X-axis label. ***ylabel*** - Y-axis label. ***grid*** - Grid lines.
16.	***Poly(V)***	Construct polynomial with the specified roots in a vectorV=*r1,r2,...*
17.	***Rand(****n, m****)***	To generate a n x m matrix with Uniformly distributed random numbers
18.	***Roots(****p****)***	Find the roots of the polynomial (*p)*.

19.	***Size(A)***	Returns the size of the matrix *A*
23.	***Who***	List current variables
24.	***Whos***	List current variables, long form.
25.	***zeros(n, m)***	To generate a Zeros matrix (all entries are zeros)

- **Brief Description of some Built – in Functions:**

No.	Function	Purpose of use
1.	***abs(x)***	Returns the absolute value of (*x)*
2.	***Ceil(x)***	Returns the ceiling of (*x)*
3.	***cos(x)***	Returns the cosine of (*x)*
4.	***floor(x)***	Returns the floor of (*x)*
5.	***log(x)***	Returns the natural logarithm of (*x)*
6.	***log10(x)***	Returns the common logarithm of (*x)*
7.	***Max(V)***	Return s the maximum component of a vector *V*
8.	***mod(x,r)***	Returns x%r
9.	***sin(x)***	Returns the sine of (*x)*
10.	***sqrt(x)***	Returns the square root of (*x)*
11.	***syms x***	syms is a shortcut to the function sym, it makes it easier for the user to create symbolic variables. Symbolic variables aren't constants like regular variables, you don't assign any value to them, you can use them to solve expressions using functions

Entering Data:
Data can be also be entered by typing directly into the Command Window, through the editor in an m-file, or through the Workspace Window by entering values in a spreadsheet format. To store a given data set in the Workspace, the data must be preceded by a variable name.

The variable name can be any combination of letters and numbers, but it must begin with a letter and they are case sensitive. Variable names are case sensitive. Also, there is a library of reserved words that cannot be used as variable names. These will appear as blue when typing them in .m files.

Variable types:
For example,

```
A=3; % simple variable
A= [1 2 3 4]; % row vector
A = [1 2 3; 4 5 6] % 2x3 matrix
```

After a variable is entered, it will be displayed in the workspace. If the variable is double clicked on in the workspace, the value of the variable will be displayed in a spreadsheet in a new window. The value of the variable can also be printed in the command window by typing the name of the variable and hitting enter.

Note: A semicolon placed at the end of a statement will suppress output to the window. It's sometimes useful to take a look at your data, but if you have a lot of data printing out to the screen while your program is running, you may slow things down.

Experiment No. 2
Introduction to Functions and Plots

Objective:

The purpose of this tutorial is to acquaint the students with basics of MATLAB functions and two-dimensional plotting. The topics discussed in this lab includes The m – files, inline functions, Vectors and Matrices.

Introduction:

From previous lab, MATLAB can be run interactively from the command line, you can write a MATLAB program by composing a text file that contains the commands you want MATLAB to perform in the order in which they appear in the file. The standard file suffix for a text file containing a MATLAB program is .m. In the MATLAB command window, selecting the pull-down menu File - > New -> M-file opens the integrated MATLAB text editor for writing a m-file. This utility is very similar to word processors, so the use of writing and saving m-files is not explained in detail here. The file is saved in the work directory of Matlab by default but can be saved in an alternate directory. You can move to another directory by clicking the [...] button and browsing to it.

The m – files:

Files that contain a computer code are called the m-files. There are two kinds of m-files:

- ✓ ***Script files***: Files that do not take the input arguments or return the output arguments.
- ✓ ***Function files***: Files that may take input arguments or return output arguments.

To create an m-file click on ***File -> New -> M-File***.

How to create function file?

function [*output1, output2,…*] = fun(*input1, input2,…*)

statement(s)

end

You have to save the function as an m – file.

How to call a function?

- ✓ In the command window you just have to type the function name and pass the inputs to the function (if any).
- ✓ When calling a function file within a script file you have to use ***feval*** command.

Example 1:

- Create an m – file and write the following code:

```
function x = square(a)
    x = a ^ 2;
end
```

- Save the m – file with the name *square.m*
- In the command window type the following line:

```
X = square(4)
```

- In the command window type the following line:

```
X = feval('square',4)
```

- Modify the m – file function as follow:

```
function [x,y] = square(a,b)
    x = a ^ 2;
    y = b ^ 2;
end
```

- Save the m – file.
- In the command window type the following line:

```
[X,Y] = square(4,5)
```

- In the command window type the following line:

```
[X,Y] = feval('square',4,5)
```

Note:
Using the semicolon at the end of any line of your code will cause the result not to be displayed on the screen.

Example 2:

```
% A comment line starts with a percent-sign.
```

Open a new m-file, go to File->Open…->Script
Type the following in the m-file:

```
% tutorial.m plots an exponential function in red circles connected by lines 'ro-'
x = linspace(0,5);
y = exp(x);
    figure(1)
    plot(x,y,'-ro');
        title('Exponential function')
        legend('e^x')
        xlabel('Time')
        ylabel('Population')
```

Save the file as "tutorial.m", make note what directory the file was saved in.

Running m-files You can run Matlab scripts in several ways. We will run the tutorial.m script from the command window and from the script itself.

To run the script from the command window:
Go to Window -> Command
At the command window prompt, type:

>> run tutorial

The name of the function must match the name of the .m file containing the function. (If not, then Matlab ignores the internal name, which just confuses everything.) As a good programming practice, the function name and the .m filename should be the same. Why? Suppose you want to call that function from another .m file or the Matlab command line, it is most logical to call it with the function name. But if you use another filename as the function name, Matlab will not be able to find the function.

Plotting and 2D Graphics

Given a set of data, it can be useful to display it graphically. This is where the plot function is useful. There are many different plot functions in Matlab that are meant for varying types of data. An overview of the different types of plots can be found at:

http://www.mathworks.com/help/techdoc/creating_plots/f9-53405.html

The basic function used to create 2-D graphs is the plot function which has a variable number of input arguments.The following is the general syntax for plotting:

plot(independent_var, dependent_var, 'formatting for data')

In general, it is possible to plot multiple sets of data on a given plot by either entering an additional set of independent and dependent variables following the first set in the parentheses as follows:

At the command window, type the following:

```
>> x = 0:.2:20;
>> y1 = sin(x)./sqrt(x+1);
>> y2 = sin(x/2)./sqrt(x+1);
>> plot(x, y1, x, y2)
```

Multiple data sets can also be plotted in the same figure by typing the command hold on.

```
>> plot(x,y1);
>> hold on
>> plot(x,y2);
```

If you have several plots active at a given point in time, the figure may need to be stipulated before using the plot function with the following syntax:

figure(figure_number)

The third potential entry in the plot function is the option to format the way in which the data is plotted. This is normally just a combination of a letter and a symbol. The letter will designate the color of the line or symbol used in plotting the data. The symbol will determine if a line is plotted or if each individual point is plotted with a symbol. The syntax for color and symbol is shown on the attached cheat sheet or can be found in the plot help listing.

Additional syntax relating to plotting is listed below:

Displays a label on the x-axis:
xlabel('Time in seconds')

Displays a label on the y-axis:
ylabel('Distance in meters')

Puts a title at the top of the figure:
title('Distance vs time')

Example:

To plot *f(x) = cos(x), and g(x) = sin(x),* xϵ[0,2π]:

```
>>X = 0:0.1:2*pi;
>>Y = cos(X);
>>Z = sin(X);
>>plot(X,Y,X,Z,'O')
```

Practice Problem: Generate a time scale from 0 to 100. Then produce an array of corresponding values for the function y=1-exp(-t./100). Plot the data with labels on the axes and a title.

Experiment No. 3
Introduction to Vectors, Matrices and Conditions

Objective:

The purpose of this tutorial is to acquaint the students with basics of MATLAB based matrix operations. The topics discussed in this lab includes Vectors, Matrices and Conditions.

Introduction:

Matlab was designed to deal mainly with numerical data in matrix or vector form. Due to this design criteria, the some of the standard mathematical operations function as matrix operations and not as you might expect.

Vectors and Matrices

In this section will see how to create *vectors* and *matrices* and the operations we can perform on them. Before we can examine how to operate on vectors or matrices, we must know how to define them in Matlab. A vector is either a column or a row of numbers, it is defined as follows:

The following command creates a vector:

```
x = [1  2  3]

x =
     1   2   3
```

There is another way to define a vector with condition that there is a common step between the successive elements. The general syntax of general operation is:

X = *start-element*:*step*:*final-element*

Example:

```
X = 0:0.5:2

X =
     0    0.5000  1.0000  1.5000  2.0000
```

A matrix:

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

The above example creates a three by three matrix. The only key to defining a matrix is to ensure that each column or row has the same number of elements.

A row vector:

```
row = [1 2 3 4];
```

A column vector:

```
column = [1;2;3;4];
```

To define a 2D matrices you may use one of the methods illustrated below:
To define a 2 x 3 matrix, say the name of the matrix is (A).

Method 1 – Using semicolon(;) at the end of each row:

```
A = [1 2 3;4 5 6]

A =
     1   2   3
     4   5   6
```

Method 2 – Using carriage return(Enter key) at the end of each row:

```
A = [1  2  3
     4  5  6]

A =
     1   2   3
     4   5   6
```

Note: You have to enclose the matrix by the *square bracket* [].

There are *Elementary Matrices* that can be generated using built – in functions, here below some examples:

```
Z = zeros(2,3)   % creates a 2x3 matrix of zeros
X = ones(2,3)    % creates a 2x3 matrix of ones
```

The components of matrices can be manipulated in several ways, here are below some examples:

```
>>A = [1 2 3;4 5 6]

A      =
     1   2   3
     4   5   6
>>A(2,3)
ans =
     6
```

Note: The *colon operator:* stands for all columns or all rows.

```
>>A(1:2,2:3)              % This selects a sub-matrix of A
ans =
     2   3
     5   6

>>A([1 2],[1 3])  % This selects a sub-matrix of A
ans =
     1   3
     4   6
>>A(2,3) = 9;              % This assigns a new value to the
                          % entry A23
```

Short-cuts to creating vectors and matrices:
Z = (1:5); %means Z = [1 2 3 4 5]
Z = (1:3:10); %means Z = [1 4 7 10]
Z = linspace(a,b,n); %means create a vector with evenly spaced points between a and b
Z = logspace(a,b,n); %means create a vector with logarithmically-spaced points between 10^a and 10^b
zeros(a, b); %means create an axb matrix of all zeros
ones(a, b); %same thing but a matrix of all ones

Operations on matrices:

Operation	Symbol
Addition	+
subtraction	-
multiplication	*
Transpose	'
Power	^
Elementwise multiplication	.*
Elementwise Division	./
Elementwise power	.^

Notes:

- For the addition and subtraction, the matrices ***must have the same size.*** Those two operation operates elementwise.
- If you want to multiply 2 matrices **A** and **B (A * B)**, and **A** has size of **n** x **m** then **B** must be with the size **m** x **p.**
- The transpose takes each row of a matrix and make it as a column.
- The power operation can be applied on square matrices only.
- The elementwise multiplication, division, and power operates on each element not on the matrix as one unit.

Matrix Manipulation:
Being able to easily manipulate the values, columns, and rows of a matrix will make your life much easier. The following are examples of the syntax used with matrices:

Enter the following matrices into your command window:

```
A=[1 2 3];
B=[4 5 6];
```

Now, try the following matrix manipulations:

```
C = [A B] % concatenate rows into a single row
D = [A;B] % concatenate A and B into two rows
E = D(1,2) % get element in row 1, column 2
F = D(2,2:3) % get elements in row 2, columns 2 and 3
G = D(:,3) % get column 3
H = D(1,:) % get row 1
I = H' % transpose
```

Examples:

- Create an m – file, name it *matrices*.
- Write the following code in the *matrices.m file.*

```
% This program perform some operations on matrices
x = input('Enter 2x2 matrix: ');
y = input('Enter 2x2 matrix: ');
disp('x + y = ');
x + y
disp('x - y = ');
x - y
disp('x * y = ');
x * y
disp('x .* y = ');
x .* y
disp('x ^ 2 = ');
x ^ 2
disp('x .^ 2 = ');
x .^ 2
disp('transpose of x = ');
x'
```

- Run the above program.

Matrix addition and subtraction:
These commands function normally and do not require any additional syntax to have the operation preformed on variables in your workspace. It is important to note, addition or subtraction is performed between elements in the same position in the array or matrix. Therefore, you will need to ensure the matrices you are operating on have the same dimensions.

Practice Problem: Enter the following variables into your command window:
A=[1 2 3];
B=[1;2;3];
Can you subtract A from B?
Can you add A to A?

Matrix multiplication and division:
'*' and '/' are for matrix multiplication and division, and the dimensions of the matrices must be compatible. '.*' and './' do element-by-element multiplication and division.
So, [1 2 3].*[4 5 6] gives [4 10 18]
A^2 means the cross-product of A with itself, and A must be a square matrix.
A.^2 means square each element of A.

So, [1 2 3].^3 gives [1 8 27] but [1 2 3]^3 gives an error. Forgetting the dot is one of the most common bugs!

Practice Problem: Enter the following data into the Matlab command window.
A=[0 2; 1 4];
B=[1 3; 2 6];
What is A*B? What is A.*B?
What does A.^-1 produce verses A^-1?

Loops and Conditional Statements:

- **For loop:**

  A *for loop* is used to execute a block of statements several times. The general syntax of a for loop is:

```
for i=1:n
  <program>
end
```

  The *break* command may be used to exit a loop.

If and elseif Statement:

If statement is used to test a certain condition(s). The general syntax of the if statement is:

```
if (condition is true)
        statement(s)
end
```

OR

```
if (condition1 is true)
        statement(s)
elseif (condition2 is true)
        statement(s)
else
        statement(s)
end
```

Example: The FOR loop performs a series of operations a known number of times. The example code below illustrate the syntax:

```
X = [2 3 1 4];
even = 0;
For k=1:length(X)
  if (x(k)%2 == 0)
          even = even + 1;
  end
end
```

Assignment vs. Equals is important in loops. An assignment is a = b and means assign value b to variable a. The equals operator is a == b results in 1 or yes if they are equal, 0 or no if they are not.

The final type of loop is a WHILE loop. It requires that a series of operations occur while the condition applied to a specified variable is correct.
The example syntax is:

```
i=10;
while i>5
i=i-1
end
```

Practice Problem: Given the matrix: M=[1 1; 1 3;]; Use loops to determine how many of the entries are greater than 1 and the location of all qualifying entries (the corresponding row and column).

Experiment No. 4
Taylor Series

Objective:

Truncation errors are those that result from using an approximation in place of an exact mathematical procedure. In order to gain insight into the properties of such errors, we now turn to a mathematical formulation that is used widely in numerical methods to express functions in an approximate fashion—the Taylor series.

Introduction:

Theorem 1. (Taylor polynomial with integral remainder) Suppose a function $f(x)$ and its first $n + 1$ derivatives are continuous in a closed interval $[c, d]$ containing the point $x = a$. Then for any value x on this interval

$$f(x) = f(a) + f'(a)(x-a) + \ldots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \frac{1}{n!}\int_a^x f^{(n+1)}(y)(x-y)^n dy.$$

It's clear from the fact that $n!$ grows rapidly as n increases that for sufficiently differentiable functions $f(x)$. Taylor polynomials become more accurate as n increases.

Methodology:

We can use Matlab to create a function or script to calculate the series for a given number of terms n.

Example 1. Write a script file that does the following:

- Prompts the user for an angle (in radians) between 0 and 2pi.
- Prompts the user for the number of terms to use in the Taylor series
- Uses a for loop to calculate the estimate for the sine of the angle by summing the number of terms in the Taylor series specified by the user. (Note: don't count the missing terms – the even powers of x – in your count of terms).
- Uses fprintf statement(s) to display the estimate of the sine, the actual sine, and the absolute value of the difference between actual and estimate. All these values should be displayed using 6 places behind the decimal point.

```
%%Matlab Code
clear
prompt = 'Enter angle value between 0 to 2pi? ';
x = input(prompt)
```

```
prompt = 'Enter the no of terms? ';


N = input(prompt)
ty = 0;
        for i = 1:N
        ty = ty +((-1)^(i))*((x^(2*i+1))/(factorial(2*i+1)));%your calculations
        end
fprintf(['actual value    ' num2str(ty)]) %display the results
```

Example 2. Find Taylor series expansion of $e\hat{}x$ about x=1 using Matlab.

```
>> syms x
>> A = taylor(exp(x), x,1)
```

Matlab Output

```
A = exp(1) + exp(1)*(x - 1) + (exp(1)*(x - 1)^2)/2 + (exp(1)*(x - 1)^3)/6 +
(exp(1)*(x - 1)^4)/24 + (exp(1)*(x - 1)^5)/120
```

Evaluate the Taylor series equation for a required value of x

```
>> B = subs(A,1.5)
```

Matlab Output

```
B = 1281/1280
```

Or, to get the answer in decimal, use double

```
>> B = double(subs(A,1.5))
```

Matlab Output

```
B = 4.4820
```

Example 3. Taylor series expansion of e^x about x=0 using Matlab

```
>>syms x
>>taylor(exp(x))
```

Matlab Output

```
x^5/120 + x^4/24 + x^3/6 + x^2/2 + x + 1
```

Use the option “order” to specify the order:

```
>>taylor(exp(x),'Order',8)          {highest power 7}
```

Matlab Output

```
x^7/5040 + x^6/720 + x^5/120 + x^4/24 + x^3/6 + x^2/2 + x + 1
```

Practice Problem: Find Taylor series using MATLAB for the following functions:

1. sin(x) up to 6^{th} order
2. cos(x) up to 8^{th} order
3. 1/1-x up to 10^{th} order

fplot is used to plot a function between specified limits.

```
>>syms x
>>A1 = taylor(sin(x))
>>A2 = taylor(sin(x),'Order',8)
>>A3 = taylor(sin(x),'Order',10)
>>fplot([A1 A2 A3])
```

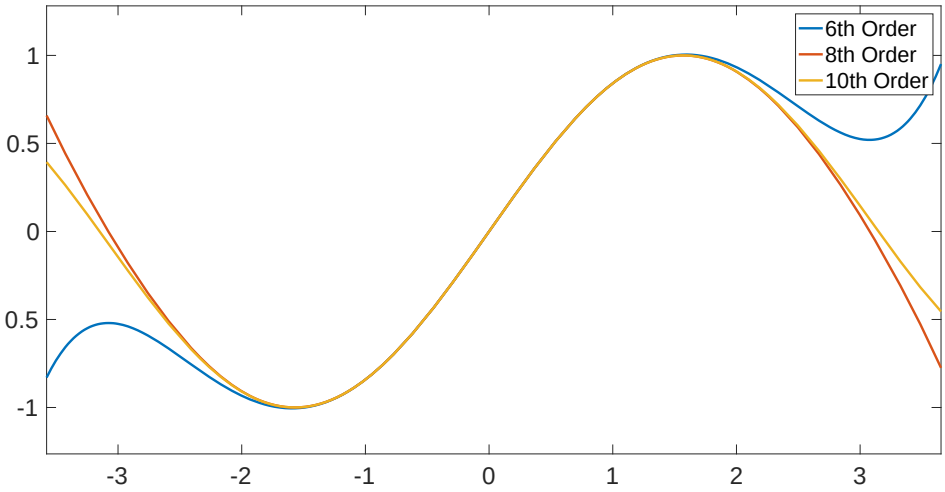


Figure 3. Plot of three Taylor series

Example 4. For the given function $f(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.25x + 1.2$, Predict the function value at $x_{i+1} = 1$ using the function value at $x_i = 0$. Use zero to 2^{nd} order approximations.

```
>>clear
>>syms x
>>f = -0.1*x^4-0.15*x^3-0.5*x^2-0.25*x+1.2;
>>for i = 1:10
>>fapprox = taylor(f,'order',i);
>>fvalue = subs(fapprox,1);
>>disp([i fvalue])
>>end
```

Experiment No. 5
Bisection Method and Locating Roots

Objective:

The first class of numerical methods is known as bracketing methods. They work by choosing two values of x, in the above case, that bracket the root. Then different methods are used to zero in on the actual root. Usually a tolerance is specified, so that the exact root is not found. The value that is closer to the root than the tolerance is kept. To gain insight into the working of these methods, we will learn how to implement bisection method using MATLAB.

Introduction:

Root finding is a skill that is particularly well suited for computer programming. Unless the roots of an equation are easy to find, iterative methods that can evaluate a function hundred, thousands, or millions of times will be required. Another was to say, "root finding" is to say "what value of x for a function will give an answer of zero". Also, root finding can be thought of as finding where two functions intersect.

Roots Command: It is used to find roots for a polynomial function

$f(x) = x^3 + 2x^2 + 10x - 20$

This is the problem Leonardo Fibonacci solved in 1225, x~1.36880810785

```
>> a=[1 2 10 -20];
a =
    1     2    10   -20
>> format long
>> roots(a)
ans = -1.684404053910685 + 3.431313501976 91i
          -1.684404053910685 - 3.431313501976 91i
          -1.368808107821373
```

Bisection Method:

Bracketing methods are based on two initial guesses that "bracket" the root—that is, are on either side of the root. The bisection method is a variation of the incremental search method in which the interval is always divided in half. If a function changes sign over an interval, the function value at the midpoint is evaluated. The location of the root is then determined as lying within the subinterval where the sign change occurs. The subinterval then becomes the interval for the next iteration. The process is repeated until the root is known to the required precision.

Algorithm:

Loop

1. Compute the mid point c=(a+b)/2
2. Evaluate f(c)
   - i. If f(a)f(c) < 0 then new interval [a, c]
   - ii. If f(a)f(c) > 0 then new interval [c, b]
   - iii. If f(a)f(c) = 0 then Stop and the root is c

End loop

Matlab Code:

```
clear
a = 5; b = 10; n = 50; es = 1e-7;
if f(a)*f(b)>0
    disp('Wrong choice of interval')
else
    t = 6.4051;     % True value of the root
    for i = 1:n
        c = (a + b)/2;
        ea = abs((b - a)/(b + a)) * 100;
        et = abs((t - c)/t) * 100;
        disp([i a b c et ea])
        if f(a)*f(c) < 0 && ea > es
            b = c;
        elseif f(a)*f(c) > 0 && ea > es
            a = c;
        else
            break
        end
    end
end
function y = f(x)
%   Problem 5.1(c) [Chapra 7th Edition]
    y = -0.5 * x.^2 + 2.5 * x + 4.5;
end
```

Figure 4. Matlab implementation of Bisection Method

Experiment No. 6
Open Methods – Newton Raphson and Secant

Objective:

These methods are related and often confused. We shall derive them in tandem, since they only differ in the last stage. The central premise for both methods is that the function is locally linear and the next iteration for the required value can be attained via linear extrapolation (or interpolation).

Introduction:

We will start using a Taylor series to derive the Newton–Raphson technique. We assume the current guess is x and this is incorrect by an amount h, so that x + h is the required value. It now remains for us to determine h or at least find an approximation to it. The Taylor expansion for the function f(x) at the point x + h is given by

$$f(x + h) = f(x) + hf'(x) + O(h^2).$$

This can be interpreted as: the value of the function at x + h is equal to the value of the function at x plus the gradient times the distance between the points. This can be considered to include further terms; now we are fitting a straight line. In this expression we have used the term $O(h^2)$: loosely this means something the same size as h^2 and the prime means differentiated with respect to the argument of the function. We now note that x + h is supposedly the actual root so f(x+h) = 0, and discarding the higher-order terms we find that

$$h \approx - f(x)/f'(x).$$

This presumes we are close to the actual root, and consequently we can discard the terms proportional to h^2 since these should be smaller than those proportional to h. This allows us to construct the iterative scheme

$$x_{n+1} = x_n - f(x_n)/f'(x_n) \text{ , } n = 0, 1, 2, \cdots .$$

This method can also be derived using geometric arguments. In these derivations the function is taken to be approximated by a straight line to determine the next point.

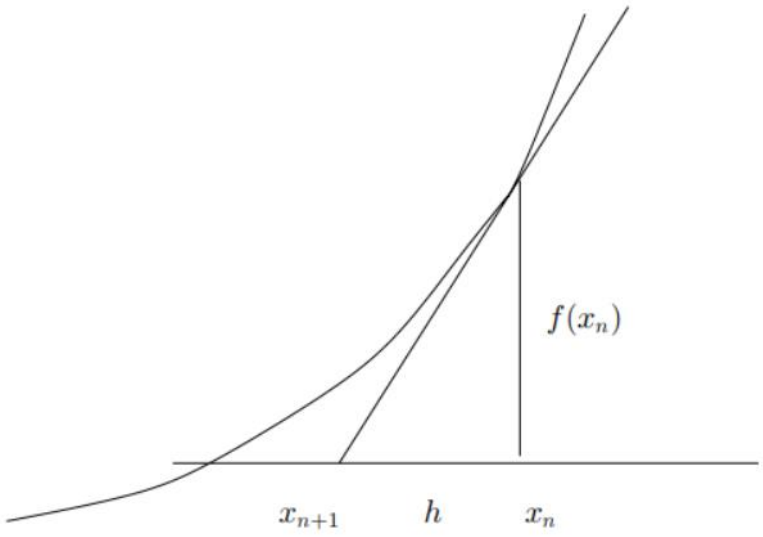


Figure 5. Newton-Raphson

The value of h is determined by using the fact that the ratio of the two sides h and $f(x_n)$ must be equal to $f'(x_n)$.
Now we shall presume that we have two routines func.m and func prime.m which give us the function and its derivative. For ease let us consider the function

$$f(x) = x - 2\sin x^2,$$
$$f'(x) = 1 - 4x\cos x^2 \text{ (using the chain rule).}$$

The Matlab code for the two functions is:

```
% func.m
function [f] = func(x)
f = x-2*sin(x.^2);

% func_prime.m
function [value] = func_prime(x)
value=1- 4*x.*cos(x.^2);
```

Notice that we have used the dot operators, even though this routine is only ever likely to be called in this context using a scalar. This permits the routine to be used from other codes in a portable fashion. This can be coded simply using:

```
x = 1;
for j = 1:10
x=x- func(x)/func_prime(x);
end
```

where we have set the initial guess to be x = 1 and supposed that the method will converge in ten iterations. We now give a more robust code to perform the iterations

```
%Newton_Raphson.m
x = input('Starting guess :');
tolerance = 1e-8;
iterations = 0;
    while (iterationstolerance)
        x = x-func(x)/func_prime(x);
        iterations = iterations + 1;
    end

    if iterations==30
        disp('No root found')
    else
        disp([' Root = ' num2str(x,10) ' found in ' ... int2str(iterations) ' iterations.'])
    end
```

Hopefully you can see the difference between the two codes and see that ultimately the second version is more useful. By entering the values 0.1, 0.6 and 1.5 we can obtain the three roots we are concerned with. Notice we have increased the number of digits printed in the answer to 10 using the form of the MATLAB command num2str which accepts two arguments.

```
%Secant.m
x0=input('Starting guess point 1 :');
x1=input('Starting guess point 2 :');
tolerance=1e-8;
iterations=0;
        while(iterations<30)& (abs(func(x1))>tolerance)
                iterations = iterations + 1 ;
                f0 = func(x0);
                f1 = func(x1);
                x2 = x0-f0*(x1-x0)/(f1-f0);
                x0 = x1;
                x1 = x2;
        end

if iterations==30
        disp('No root found')
else
        disp([' Root = ' num2str(x1,10) ' found in ' ... num2str(iterations) '
iterations.'])
```

This method works far better if the two initial points are on opposite sides of the root. In the above method we have merely chosen to proceed to use x_1 and the newly attained point x_2: however, we could equally have chosen x_0 and x_2. In order to determine which, we should use we require that the function changes sign between the two ends of the interval. This is done by changing the lines where the next interval is chosen.

Practice Problem: Using the Newton–Raphson routine determine the zero of the function

- $f(x)=e^{x} - e^{-2x} + 1$.

Instructor Solution:

```
function [value] = func(x)
value = exp(x)-exp(-2*x)+1;
```

```
function [value] = func_prime(x)
value = exp(x)+2*exp(-2*x);
```

This yields the result

```
>> Newton_Raphson
Starting guess :-2
 Root = -0.2811995743 found in 8 iterations.
```

Experiment No. 7a
Elimination Methods

Objective:

In this lab, students will get acquainted with Gauss elimination, Reduced row echelon form, Gauss-Jordon elimination and LU decomposition methods.

Introduction:

Gaussian Elimination:

The Gauss Elimination method is a method for solving a matrix equation Ax=b for x. The process is:

1. Start by augmenting the matrix A with the column vector b.
2. Perform elementary row operations to transform this augmented matrix to upper triangular form.
3. Use back-substitution to solve for the unknowns in x.

For example, for a 2 x 2 system, the first step is to augment the matrix [A b]:

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \end{bmatrix}$$

Then, EROs are applied to get the augmented matrix into an upper triangular form (which, for a 2 x 2 matrix means finding an ERO that would change a_{21} to 0):

$$\begin{bmatrix} a'_{11} & a'_{12} & b'_1 \\ 0 & a'_{22} & b'_2 \end{bmatrix}$$

Here, the primes indicate that the values may have been changed. Putting this back into the equation form, we have

$$\begin{bmatrix} a'_{11} & a'_{12} \\ 0 & a'_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \end{bmatrix}$$

So, we now use the last line which says

$$a'_{22}\, x_2 = b'_2$$

to solve first for

$$x_2 = b'_2 / a'_{22}$$

and then go back to the first line which says

$$a'_{11}\, x_1 + a'_{12}\, x_2 = b'_1$$

to solve for

$$x_1 = (b'1 - a'_{12}\, x2) / a'_{11}$$

Gauss-Jordan Elimination Method:

In Gauss-Jordan elimination method, we first transfer the Linear Equation System (LES) to a matrix system. Then, after applying necessary elementary row operations, we try to achieve a matrix which consists of an Identity matrix in the first part and the solution column in the second part. Therefore, we may find the solution directly by this solution column. For example,

$$\begin{aligned} u - v + w &= 5 \\ -u + v + 2w &= 7 \\ 2u + v - w &= 4 \end{aligned} \Rightarrow A = \left[\begin{array}{ccc:c} 1 & -1 & 1 & 5 \\ -1 & 1 & 2 & 7 \\ 2 & 1 & -1 & 4 \end{array}\right] \xrightarrow{\text{After all the elementary row operations}} A = \left[\begin{array}{ccc:c} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 4 \end{array}\right]$$

Forward Elimination and Backward Substitution:

```
%Forward Elimination to build an upper triangular matrix
        for k=1:n-1
                for i=k+1:n
                        factor = a(i,k)/a(k,k);          %normalzing factor (Step A)
                                for j=k+1:n                      %move accross the columns loop
                                        a(i,j) = a(i,j) - factor*a(k,j);    %(Step B)
                                end
                        b(i)=b(i)-factor*b(k);           %(Step C)
                end
        end

%Backward Substitution
x(n)=b(n)/a(n,n);       %solve for the last x value
        for i=n-1:-1:1
                sum = 0;
                        for j=i+1:n
                                sum = sum + a(i,j)*x(j);
                        end
                x(i)=(b(i)-sum)/a(i,i);
        end
```

Gaussian Elimination:

```
a = [3 4 -2 2 2
     4 9 -3 5 8
    -2 -3 7 6 10
     1 4 6 7 2];

%Gauss elimination method [m,n)=size(a);
[m,n]=size(a);
            for j=1:m-1
                    for z=j+1:m
                       if a(j,j)==0
                          t=a(j,:);a(j,:)=a(z,:);
                          a(z,:)=t;
                       end
                    end
                    for i=j+1:m
                       a(i,:)=a(i,:)-a(j,:)*(a(i,j)/a(j,j));
                    end
            end
x=zeros(1,m);
            for s=m:-1:1
             c=0;
                      for k=2:m
```

Matlab Built-in Operator

- Forward Slash (\ )
- Usage:
  - x = A\b

Description: Solve linear algebraic system, Ax = b, using Gauss elimination.

- Inputs:
  - A = m by n coefficient matrix
  - b = m by 1 right-hand side vector
- Outputs:
  - x = n by 1 solution vector
  - $\begin{bmatrix} 2 & -4 & 1 \\ 6 & 2 & -1 \\ -2 & 6 & -2 \end{bmatrix} \quad x = \quad \begin{bmatrix} 4 \\ 10 \\ -6 \end{bmatrix}$

```
            c=c+a(s,k)*x(k);
        end
        x(s)=(a(s,n)-c)/a(s,s);
    end

disp('Gauss elimination method:');
a
x'
```

Gauss Jordan Elimination:

```
% Gauss-Jordan Elimination Generalized
clear;
clc;
A = [1 -1 1 5;
     -1 1 2 7;
      2 1 -1 4];
[m,n] = size(A);
% All row operations
% Downwards
    for i=1:m
    A(i,:) = A(i,:)/A(i,i);

        for j=(i+1):m
            A(j,:) = A(j,:)+(-1)*A(j,i)*A(i,:);
        end
    end
%Upwards
    for i=m:-1:1
        for j=(i-1):-1:1
            A(j,:) = A(j,:) + (-1)*A(j,i)*A(i,:);
        end
    end
%Finding the values
x = []; % Solution vector
    for i=1:m
        x(i) = A(i,n);
    end
```

Practice Problem: Solve the linear system using Gauss Elimination method.

$$x_1 - x_2 + 2x_3 - x_4 = -8$$
$$2x_1 - 2x_2 + 3x_3 - 3x_4 = -20$$
$$x_1 + x_2 + x_3 = -2$$
$$x_1 - x_2 + 4x_3 + 3x_4 = 4$$

Experiment No. 7b
Elimination Methods

Reduced Row Echelon Form:
Suppose now you want to solve a system of matrices by getting the augmented matrix in reduced row-echelon form, the rref command does this in MATLAB.

For example, if we put in the augmented matrix (corresponding to a system of linear equations) for the problem:

```
>> A=[1 -1 -3 7 ; -3 1 4 -16 ; 4 -3 -5 12]
```

and then reduce it

```
>> rref(A)
```

we see that MATLAB saves us time compared to all the work we needed to do on the previous page. Here is another, this one has free variables at the end. We will see that after rref because there are non-pivot columns corresponding to variables.

```
>> A=[1 3 4 2; 1 -3 0 1]
```

and then reduce it

```
>> rref(A)
```

we get the reduced form. It's in decimals but clearly it corresponds to the augmented matrix

$$\begin{bmatrix} 1 & 0 & 2 & | & \frac{3}{2} \\ 0 & 1 & \frac{2}{3} & | & \frac{1}{6} \end{bmatrix}$$

Note that the third column is not a pivot column so x_3 is free. Hence, we have the solution.

$$x_1 = \frac{3}{2} - 2x_3$$
$$x_2 = \frac{1}{6} - \frac{2}{3}x_3$$
$$x_3 = \text{free}$$

Lastly, note that we can deal with matrices with unknown constants in them but we have to explicitly tell MATLAB that they are unknown constants. For example, suppose we wish to enter the matrix

$$\begin{bmatrix} -2 & 3 & | & h \\ 5 & -1 & | & k \end{bmatrix}$$

First we must tell MATLAB that h and k are to be dealt with symbolically. We do this with the syms command

```
>> syms h
>> syms k
```

And now we can do

```
>> A=[-2 3 h ; 5 -1 k]
>> rref(A)
```

Practice Problem: Consider the system of equations

$$x_1 + 2x_2 - 3x_3 = -3$$
$$-4x_1 - 5x_2 + 2x_3 = -2$$
$$2x_1 + 3x_2 - x_3 = 2$$

(a) Enter it into MATLAB as an (augmented) matrix named A.
(b) Continue to reduced row-echelon form.
(c) Give the solution of the system.

LU Decomposition:

Suppose we have the system of equations

$$AX = B.$$

The motivation for an LU decomposition is based on the observation that systems of equations involving triangular coefficient matrices are easier to deal with. Indeed, the whole point of Gaussian Elimination is to replace the coefficient matrix with one that is triangular. The LU decomposition is another approach designed to exploit triangular systems. We suppose that we can write

$$A = LU$$

where L is a lower triangular matrix and U is an upper triangular matrix. Our aim is to find L and U and once we have done so we have found an LU decomposition of A.

The name of the built-in function for a Lower-Upper decomposition is '**lu**'. To get the LU factorization of a **square matrix *A***, type the command

```
>>[L, U] = lu(A)
```

Matlab returns a lower triangular matrix ***L*** and an upper triangular matrix ***U*** such that ***L**U** = ***A***.

Suppose that we need to perform a factorization on

```
A= [ 1  2 -3
    -3 -4 13
     2 1 -5]
```

We can verify that if matrix

```
L = [ 1   0     0
     -3   1     0
      2  -1.5  1]
```

and matrix

```
U = [1   2  -3
     0   2   4
     0   0   7]
```

Then, factors of ***A*** are ***L*** and ***U***, that is: ***A*** = ***L**U***

The decomposition of the matrix ***A*** is an illustration of an important and well known theorem. If ***A*** is a **nonsingular matrix** that can be transformed into an upper diagonal form ***U*** by the application or row addition operations, then there exists a lower triangular matrix ***L*** such that ***A*** = ***LU***.

Row addition operations can be represented by a product of **elementary** matrices. If *n* such operations are required, the matrix ***U*** is related to the matrix ***A*** in the following way:

$$\boldsymbol{U} = \boldsymbol{E}_n \boldsymbol{E}_{n-1} \ldots \boldsymbol{E}_2 \boldsymbol{E}_1 \boldsymbol{A}$$

The lower triangular matrix ***L*** is found from

$$\boldsymbol{L} = \boldsymbol{E}_1^{-1} \boldsymbol{E}_2^{-1} \ldots \boldsymbol{E}_n^{-1}$$

L will have ones on the diagonal. The **off-diagonal elements** are zeros above the diagonal, while the elements below the diagonal are the multipliers required to perform **Gaussian elimination** on the matrix ***A***. The element l_{ij} is equal to the multiplier used to eliminate the (i, j) position.

> [L,U]=lu(A) stores an upper triangular matrix in U and a "psychologically lower triangular matrix" (i.e. a product of lower triangular and permutation matrices) in L, so that A = L*U. A can be rectangular.
>
> [L,U,P]=lu(A) returns unit lower triangular matrix L, upper triangular matrix U, and permutation matrix P so that P*A = L*U.
>
> [L,U]=lu(sparse(A),0) no pivoting

Example:
In Matlab, let's find the **LU decomposition** of the matrix

```
>>A = [-2 1 -3; 6 -1 8; 8 3 -7]
```

Write this instruction in the command window or within a script:

```
>>[L, U] = lu(A)
```

And the Matlab answer is:

```
L =
  -0.2500   -0.5385    1.0000
   0.7500    1.0000         0
   1.0000         0         0

U =
   8.0000    3.0000   -7.0000
        0   -3.2500   13.2500
        0         0    2.3846
```

We can test the answer, by typing

```
>>L*U
```

And, finally, the Matlab answer is:

```
ans =
  -2.0000    1.0000   -3.0000
   6.0000   -1.0000    8.0000
   8.0000    3.0000   -7.0000
```

Showing that ***A*** = ***L*********U***, indeed.

Experiment No. 8
Least Squares Regression and Newton Interpolation

Objective:

In this lab, students will get acquainted with least squares linear regression and Newton interpolation techniques.

Introduction:

First, we will come up with a Matlab function to compute slope and intercept of the best fit straight line along with correlation coefficient and standard error of the estimate. For given inputs vectors x and y, the output slope and intercept along with the best fit plot are provided.

```
>> x = [10 20 30 40 50 60 70 80];
>> y = [25 70 380 550 610 1220 830 1450];
>> linregr(x,y)
r2 =
0.8805
ans =
19.4702 -234.2857
```

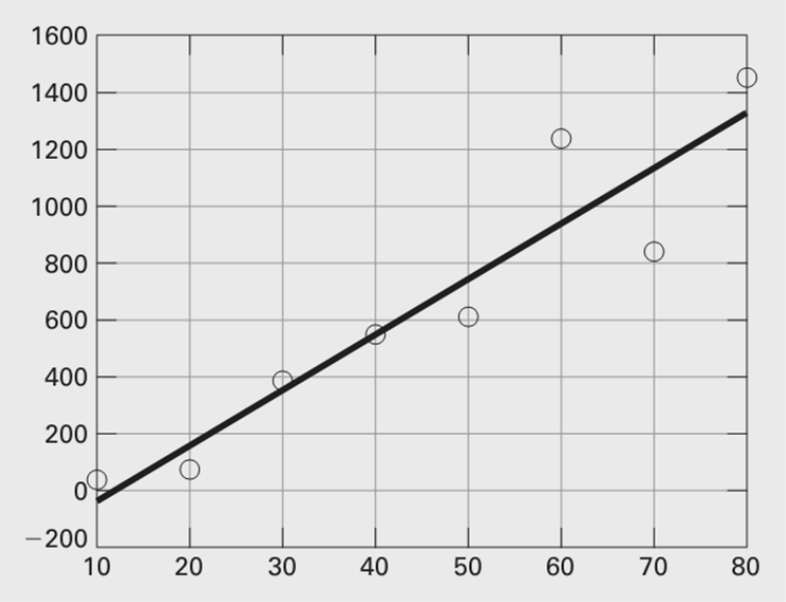


Figure 6. Linear regression

```
SUB Regress(x, y, n, a1, a0, syx, r2)

  sumx = 0: sumxy = 0: st = 0
  sumy = 0: sumx2 = 0: sr = 0
  DOFOR i = 1, n
    sumx = sumx + x_i
    sumy = sumy + y_i
    sumxy = sumxy + x_i*y_i
    sumx2 = sumx2 + x_i*x_i
  END DO
  xm = sumx/n
  ym = sumy/n
  a1 = (n*sumxy - sumx*sumy)/(n*sumx2 - sumx*sumx)
  a0 = ym - a1*xm
  DOFOR i = 1, n
    st = st + (y_i - ym)^2
    sr = sr + (y_i - a1*x_i - a0)^2
  END DO
  syx = (sr/(n - 2))^0.5
  r2 = (st - sr)/st

END Regress
```

Figure 7. Regression Psuedo Code

```
function [a, r2] = linregr(x,y)
% linregr: linear regression curve fitting
%    [a, r2] = linregr(x,y):Least squares fit of straight
%              line to data by solving the normal equations

% input:
%   x = independent variable
%   y = dependent variable
% output:
%   a = vector of slope, a(1), and intercept, a(2)
%   r2 = coefficient of determination

n = length(x);
if length(y)~=n, error('x and y must be same length'); end
x = x(:); y = y(:);     % convert to column vectors
sx = sum(x); sy = sum(y);
sx2 = sum(x.*x); sxy = sum(x.*y); sy2 = sum(y.*y);
a(1) = (n*sxy-sx*sy)/(n*sx2-sx^2);
a(2) = sy/n-a(1)*sx/n;
r2 = ((n*sxy-sx*sy)/sqrt(n*sx2-sx^2)/sqrt(n*sy2-sy^2))^2;
% create plot of data and best fit line
xp = linspace(min(x),max(x),2);
yp = a(1)*xp+a(2);
plot(x,y,'o',xp,yp)
grid on
```

Figure 8. Linear Regression Matlab Code

Polyfit and Polyval:
MATLAB has a built-in function polyfit that fits a least-squares *n*th-order polynomial to data. It can be applied as in

```
>> p = polyfit(x, y, n)
```

where *x* and *y* are the vectors of the independent and the dependent variables, respectively, and *n* = the order of the polynomial. The function returns a vector *p* containing the polynomial's coefficients. We should note that it represents the polynomial using decreasing powers of *x* as in the following representation:

$$f(x) = p_1 x^n + p_2 x^{n-1} + \cdots + p_n x + p_{n+1}$$

Because a straight line is a first-order polynomial, polyfit(x,y,1) will return the slope and the intercept of the best-fit straight line.

```
>> x = [10 20 30 40 50 60 70 80];
>> y = [25 70 380 550 610 1220 830 1450];
>> a = polyfit(x,y,1)
a =
19.4702 -234.2857
```

Thus, the slope is 19.4702 and the intercept is -234.2857.

Another function, polyval, can then be used to compute a value using the coefficients. It has the general format:

```
>> y = polyval(p, x)
```

where *p* = the polynomial coefficients, and *y* = the best-fit value at *x*. For example,

```
>> y = polyval(a,45)

y =
641.8750
```

Polynomial Interpolation:
For the case where the number of data points equals the number of coefficients, polyfit performs interpolation. That is, it returns the coefficients of the polynomial that pass directly through the data points. For example, it can be used to determine the coefficients of the parabola that passes through the last three density values:

```
>> format long
>> T = [300 400 500];
>> density = [0.616 0.525 0.457];
>> p = polyfit(T,density,2)
p =
0.00000115000000 -0.00171500000000 1.02700000000000
```

We can then use the polyval function to perform an interpolation as in

```
>> d = polyval(p,350)
d =
0.56762500000000
```

Newton Interpolation:
It is straightforward to develop an M-file to implement Newton interpolation. As the first step is to compute the finite divided differences and store them in an array.

The differences are then used in conjunction with Newton interpolation equation to perform the interpolation. An example of a session using the function would be:

```
>> format long
>> x = [1 4 6 5]';

>> y = log(x);
>> Newtint(x,y,2)

ans =
0.62876857890841
```

The algorithm for Newtint() function used in the above example is given as:

```
SUBROUTINE NewtInt (x, y, n, xi, yint, ea)
  LOCAL fdd(n,n)
  DOFOR i = 0, n
    fdd(i,0) = y(i)
  END DO
  DOFOR j = 1, n
    DOFOR i = 0, n - j
      fdd(i,j) = (fdd(i+1,j-1) - fdd(i,j-1))/(x(i+j) - x(i))
    END DO
  END DO
  xterm = 1
  yint(0) = fdd(0,0)
  DOFOR order = 1, n
    xterm = xterm * (xi - x(order-1))
    yint2 = yint(order-1) + fdd(0,order) * xterm
    ea(order-1) = yint2 - yint(order-1)
    yint(order) = yint2
  END order
END NewtInt
```

Figure 9. Newton Interpolation Psuedocode

```
function yint = Newtint(x,y,xx)
% Newtint: Newton interpolating polynomial
% yint = Newtint(x,y,xx): Uses an (n - 1)-order Newton
%   interpolating polynomial based on n data points (x, y)
%   to determine a value of the dependent variable (yint)
%   at a given value of the independent variable, xx.
% input:
%   x = independent variable
%   y = dependent variable
%   xx = value of independent variable at which
%        interpolation is calculated
% output:
%   yint = interpolated value of dependent variable

% compute the finite divided differences in the form of a
% difference table
n = length(x);
if length(y)~=n, error('x and y must be same length'); end
b = zeros(n,n);
% assign dependent variables to the first column of b.
b(:,1) = y(:);  % the (:) ensures that y is a column vector.
for j = 2:n
  for i = 1:n-j+1
    b(i,j) = (b(i+1,j-1)-b(i,j-1))/(x(i+j-1)-x(i));
  end
end
% use the finite divided differences to interpolate
xt = 1;
yint = b(1,1);
for j = 1:n-1
  xt = xt*(xx-x(j));
  yint = yint+b(1,j+1)*xt;
end
```

Figure 10. An m-file to implement Newton Interpolation

Experiment No. 9
Integration and Differentiation

Objective:

The primary objective of this laboratory session is to introduce the students with numerical integration and differentiation and their implementation.

Introduction:

Mathematically, integration is represented by

$$I = \int_a^b f(x)dx \qquad (9.1)$$

which stands for the integral of the function f(x) with respect to be independent variable *x*, evaluated between the limits *x* = a to *x* = b.

For function lying above the *x* axis, the integral expressed by Eq(9.1) corresponds to the area under the curve of *f*(*x*) between *x* = *a* and *b*.

In this lab, we will only use two common approaches to solve the numerical integration problem trapezoidal rule and simpson's 1/3.

MATLAB functions: quad & trapz

MATLAB has two functions for implementing integral problem:

- ***quad***. This function uses adaptive Simpson quadrature.
  The general call syntax for quad is as follows:

  integral = quad ('your_function', a, b, tol, trace, p1, p2,..)

- ***trapz***. This function uses trapezoidal method

The general call syntax for *trapz* is as follows:

integral = trapz(x,y)

where the two vectors *x* and *y*, hold the independent and dependent variables, respectively.

Example 9.1
Integrate the function

$$f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

from *a* = 0 to *b* = 0.8.

Solution
First, use the MATLAB function quad to integrate the function. To use quad, we must first develop an M-file to hold the function. Using a text editor, we can create the following file

```
function y=fx(x)
y=0.2+25*x-200*x.^2+675*x.^3-900*x.^4+400*x.^5;
```

It can then be stored on the MATLAB directory as **fx.m**. To see how it work, type

```
>> integral=quad ('fx', 0, 0.8)
```

where the second two entries are the integration limits. The result is

```
integral=
1.6405
```

Example 9.2
Repeat the Example 9.1.

Solution
First, generate the values of independent variable,

```
>> x=0:0.1:0.8
```

Then, generate a vector y containing the corresponding dependent variable value;

```
>> y=0.2+25*x-200*x.^2+675*x.^3-900*x.^4+400*x.^5;
```

Now, integrate these values by *trapz* function,

```
>> integral=trapz(x,y)

integral=
    1.6008
```

Trapezoidal Rule:

The trapezoidal rule is a method for finding an approximate value for a definite integral. Suppose we have the definite integral

$$\int_a^b f(x)dx$$

First the area under the curve $y = f(x)$ is divided into n strips, each of equal width $h = \frac{b-a}{n}$.

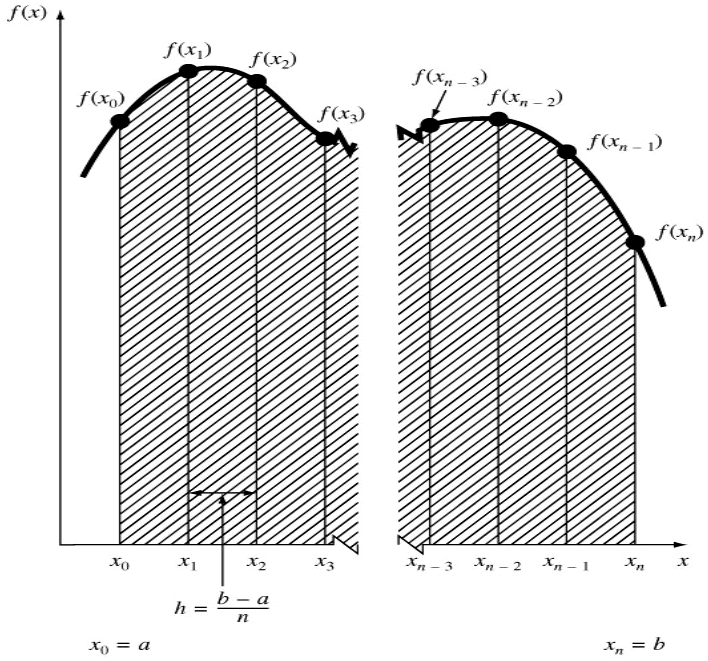


Figure 11. *The general format and nomenclature for multiple-application integrals*

Denoting the areas of the trapezoidal as I_1, I_2, ….,I_n, we have (Fig.6.1)

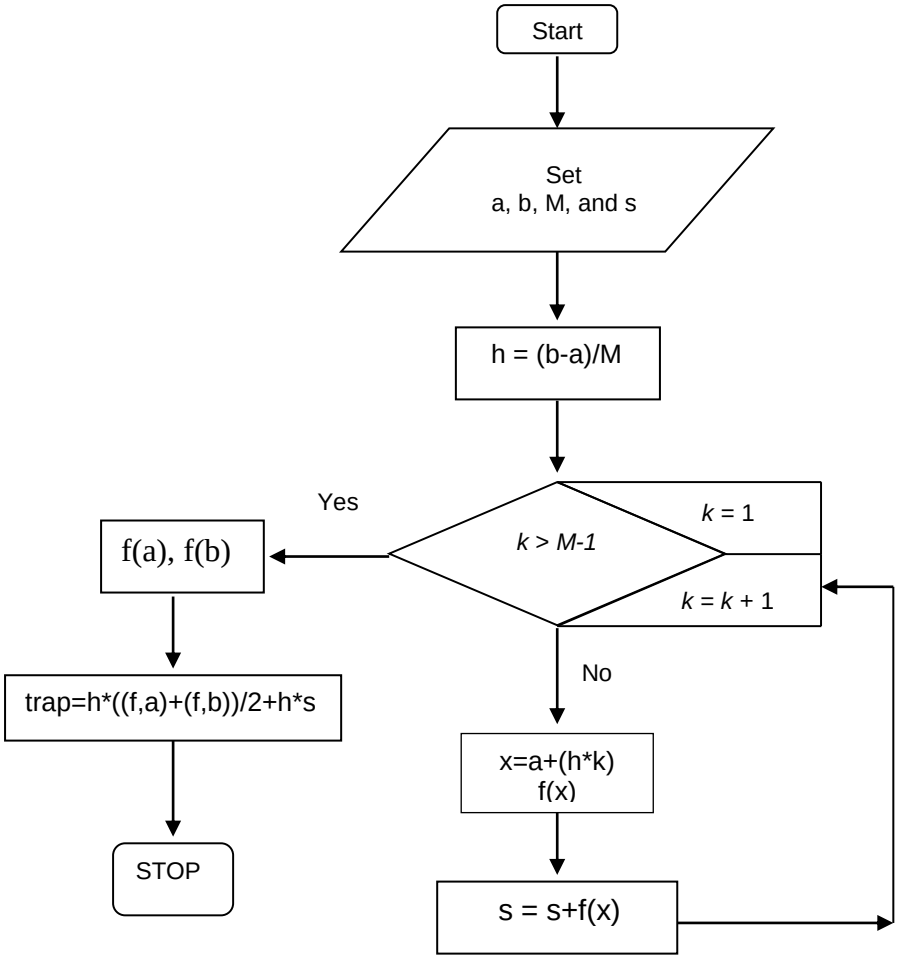
$$I_1 = \left(\frac{f(x_0) + f(x_1)}{2} \right) h, I_2 = \left(\frac{f(x_1) + f(x_2)}{2} \right) h, \ldots$$

$$I_i = \left(\frac{f(x_{i-1}) + f(x_i)}{2} \right) h, \ldots, \text{ and } I_n = \left(\frac{f(x_{n-1}) + f(x_n)}{2} \right) h.$$

The integral can be evaluated as

$$I = \int_a^b f(x)dx \approx \sum_{i=1}^{n} I_i = \frac{h}{2}(f(x_0) + 2f(x_1) + 2f(x_2) + \ldots + 2f(x_{n-1}) + f(x_n)).$$

Flowchart of Trapezoidal Rule



Example 9.3

Determine the value of the integral $I = \int_1^4 \frac{x}{\sqrt{x+1}} dx$ using trapezoidal rule with different step lengths.

```
%Input - f=inline('x/(x+1)^0.5') is the integrand
%      - a and b are lower and upper limits of integration
%      - M is the number of subintervals
%Output- s is the trapezoidal rule sum

f=input('Enter the function:');
a=input('Enter the lower bound(a):');
b=input('Enter the upper bound(b):');
M=input('Enter the number of subintervals(M):');

h=(b-a)/M;
s=0;
for k=1:(M-1)
    x=a+(h*k)
```

```
    s=s+feval(f,x)
end

s=h*(feval(f,a)+feval(f,b))/2+h*s
```

Simpson's Rule:

Simpson's 1/3 Rule is another technique used for numerical integration. When we used the single interval trapezoidal rule to estimate the integral of f(x) over the range of a to b, we drew a straight line from the point (a,f(a)) to (b,f(b)). A more accurate approach might to increase the level of the polynomial approximating the curve from linear (a polynomial of order one, as used in the trapezoidal rule) to quadratic (a polynomial of order two, as used in Simpson's 1/3 rule).

The application of Simpson's 1/3 rule to a curve is shown graphically below.

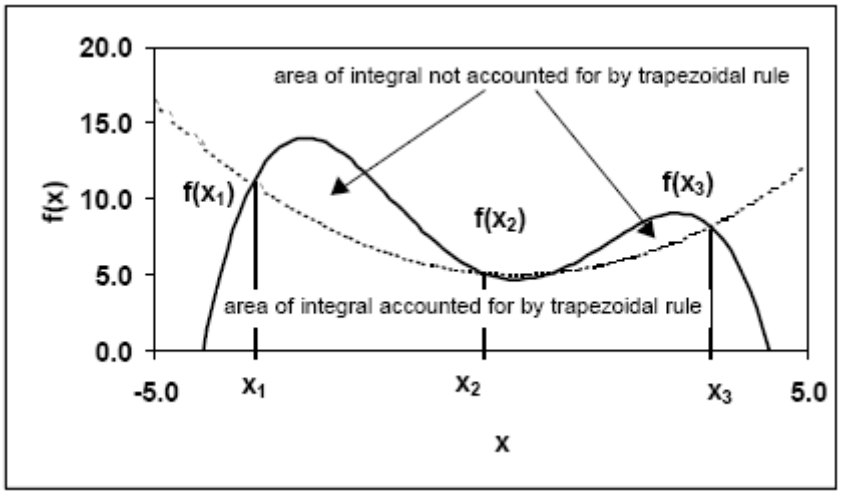


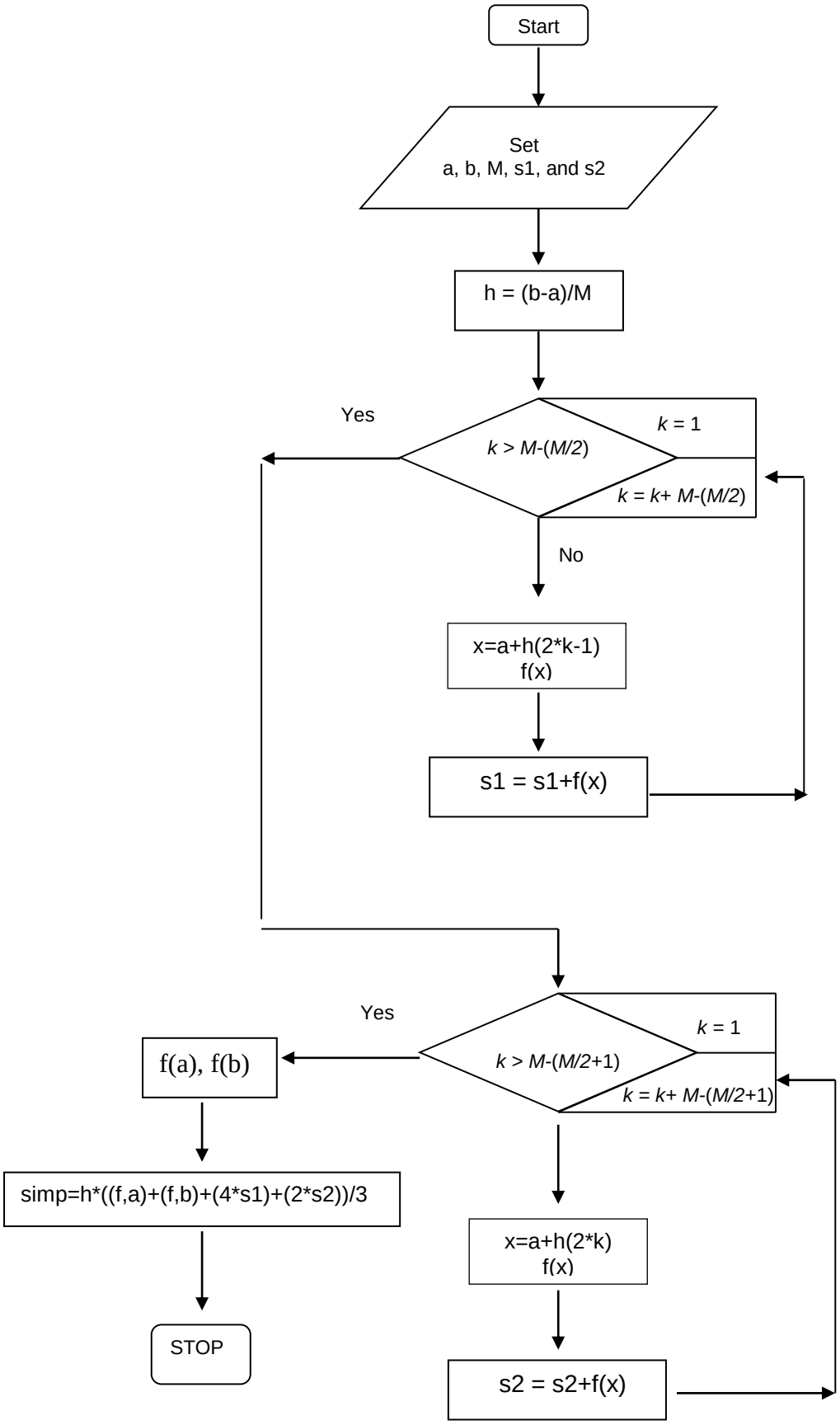
Figure 11. Simpson 1/3 Rule

The Simpson's 1/3 rule is

$$I \approx \frac{h}{3}\left[f_0 + 4 \sum_{i=1,3,5,\ldots}^{n-1} f_i + 2 \sum_{i=2,4,6}^{n-2} f_i + f_n \right] \qquad (9.3)$$

The term "1/3" in Simpson's one-third rule refers to the presence of the factor "1/3" in Eq.(9.3). For a multistage application of Simpson's one-third rule, we need to divide the range $a \le x \le b$ into n segments of equal width $h = \frac{b-a}{n}$. The number of segments must be an even number so that Eq.(9.3) can be applied of two segments.

Flowchart of Simpson's 1/3 Rule



Example 9.4

Determine the value of the integral $I = \int_1^4 \frac{x}{\sqrt{x+1}} dx$ using Simpson's 1/3 rule with step sizes = 1.

```
%Input - f=inline('x/(x+1)^0.5') is the integrand
%      - a and b are lower and upper limits of integration
%      - M is the number of subintervals
%Output- s is the trapezoidal rule sum

f=input('Enter the function:');
a=input('Enter the lower bound(a):');
b=input('Enter the upper bound(b):');
M=input('Enter the number of subintervals(M):');

h=(b-a)/M
s1=0
s2=0
for k=1:(M-(M/2))
   x=a+h*(2*k-1)
   s1=s1+feval(f,x)
end

for k=1:(M-((M/2)+1))
    x=a+h*2*k
    s2=s2+feval(f,x)
end

s=h*(feval(f,a)+feval(f,b)+(4*s1)+(2*s2))/3
```

Numerical Differentiation:

The MATLAB `diff` Function

- To make computing the numerical derivative a bit easier, MATLAB has the function `diff(x)` which computes the differences between adjacent values of the vector `x`
- The vector (matrix) that `diff(x)` returns contains one less element (row) than `x`
- If samples of $f(x)$ are contained in vector `y` and the corresponding x values in vector `x`, the derivative can be estimated using

```
deriv_y = diff(y)./diff(x);
```

- The corresponding x values are obtained from the original `x` vector by trimming either the first or last value
  - Trimming the last value results in a forward difference estimate
  - Trimming the first value results in a backward difference estimate

Example: Find the derivative of a function containing polynomial terms and a trig function

$$f(x) = 5\cos(10x) + x^3 - 2x^2 - 6x + 10$$

```
» x = 0:.01:4;
» y = 5*cos(10*x) + x.^3 - 2*x.^2 - 6*x +10;
» plot(x,y)
» deriv_y = diff(y)./diff(x);
» xd = x(2:length(x)-1);
» figure
» xd = x(2:length(x)); % Backward difference x values
» plot(xd,deriv_y)
```

EXAMPLE 23.5 Using diff and gradient for Differentiation

Problem Statement. Explore how the MATLAB's diff and gradient functions can be employed to differentiate the function $f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$ from $x = 0$ to 0.8. Compare your results with the exact solution: $f'(x) = 25 - 400x^2 + 2025x^2 - 3600x^3 + 2000x^4$.

Solution. We can first express $f(x)$ as an anonymous function

```
>> f=@(x) 0.2+25*x-200*x.^2+675*x.^3-900*x.^4+400*x.^5;
```

We then generate a series of equally-spaced values of the independent and dependent variables,

```
>> x=0:0.1:0.8;
>> y=f(x);
```

The diff function is to determine the differences between adjacent elements of each vector. For example,

```
>> format short g
>> diff(x)

 0.1000  0.1000  0.1000  0.1000  0.1000  0.1000  0.1000  0.1000
```

As expected, the result represents the differences between each pair of elements of x. To compute divided-difference approximations of the derivative, we merely perform a vector division of the y differences by the x differences by entering

```
>> d=diff(y)./diff(x)

10.89  -0.01  3.19  8.49  8.69  1.39  -11.01  -21.31
```

Note that because we are using equally-spaced values, after generating the x values, we could have simply performed the above computation concisely as

```
>> d=diff(f(x))/0.1;
```

The vector d now contains derivative estimates corresponding to the midpoint between adjacent elements. Therefore, in order to develop a plot of our results, we must first generate a vector holding the x values for the midpoint of each interval

```
>> n=length(x);
>> xm=(x(1:n-1)+x(2:n))./2;
```

We can compute values for the analytical derivative at a finer level of resolution to include on the plot for comparison.

```
>> xa=0:.01:.8;
>> ya=25-400*xa+3*675*xa.^2-4*900*xa.^3+5*400*xa.^4;
```

A plot of the numerical and analytical estimates is then generated with

```
subplot(1,2,1), plot(xm,d,'o',xa,ya)
xlabel('x'),ylabel('y')
legend('numerical','analytical'),title('(a) diff')
```

As displayed in Fig. 23.7*a*, the results of the numerical approximation compare favorably with the exact, analytical solution for this case.

We can also use the `gradient` function to determine the derivatives as

```
>> dy=gradient(y,0.1)
dy = 10.89 5.44 1.59 5.84 8.59 5.04 -4.81 -16.16 -21.31
```

As was done for the diff function, we can also display both the numerical and analytical estimates on a plot,

```
>> subplot(1,2,2), plot(x,dy,'o',xa,ya)
>> xlabel('x')
>> legend('numerical','analytical'),title('(b) gradient')
```

The results (Fig. 23.7*b*) are not as accurate as those obtained with the `diff` function (Fig. 23.7*a*). This is due to the fact that `gradient` employs intervals that are two times (0.2) as wide as for those used for `diff` (0.1).

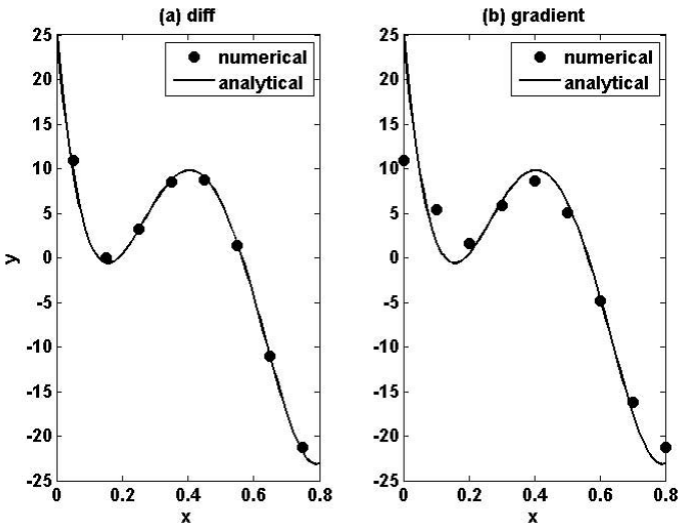


FIGURE 23.7
Comparison of the exact derivative (line) with numerical estimates (circles) computed with MATLAB's (*a*) `diff`, and (*b*) gradient functions.

Anonymous Functions

You can create handles to anonymous functions. An anonymous function is a one-line expression-based MATLAB function that does not require a program file. Construct a handle to an anonymous function by defining the body of the function, anonymous_function, and a comma-separated list of input arguments to the anonymous function, arglist. The syntax is:

h = @(arglist)anonymous_function

For example, create a handle, sqr, to an anonymous function that computes the square of a number, and call the anonymous function using its handle.

```
>> sqr = @(n) n.^2;
>> x = sqr(3)
    x =
         9
```