Ahsanullah University of Science and Technology

Department of Electrical and Electronic Engineering

LABORATORY MANUAL

FOR

ELECTRICAL AND ELECTRONIC SESSIONAL COURSE

Student Name :

Student ID :

Course no : EEE 3110

Course Title : Numerical Technique Laboratory

For the students of

Department of Electrical and Electronic Engineering

3rd Year, 1st Semester

Experiment 1: Introduction to MATLAB
(Part 1)

MATLAB works with three types of windows on your computer screen. These are the Command window, the Figure window and the Editor window. The Figure window only pops up whenever you plot something. The Editor window is used for writing and editing MATLAB programs (called M-files) and can be invoked in Windows from the pull-down menu after selecting File | New | M-file.

Create a folder of your group name in C drive.
Open MATLAB 6.5. In "current directory" select your folder.

From the "file" menu start a new m-file. Always write your program in m-file & save it. The file/function/variable name must start with a letter and cannot contain space. The name can be a mix of letters, digits, and underscores. (*e.g.*, vector_A, but not vector-A (since "-" is a reserved char). must *not* be longer than 31 characters.

You can also write any command or program in "command window".
Function "clear all" removes all variables, globals, functions and MEX links. Write clear all at the beginning of your m-file.
Write "clc" in command window to clear the command window, "clg" to clear graphics window.
MATLAB is case sensitive. *e.g.*, NAME, Name, name are 3 distinct variables.
Write "help function_name" in command window after the ">>" sign to see the description of the function. For example, type "help sin" to know about sin functions in MATLAB. You can also use help in the menubar.
Explore MATLAB's lookfor and help capability by trying the following:

```
>> lookfor keywords
>> help
>> help plot
>> help ops
>> help arith
```

Special Characters:

There are a number of special **reserved** characters used in *MATLAB* for various purposes. Some of these are used as arithmetic operators, namely, +, -, *, / and \. While others perform a multitude of purposes:

- % -- anything after % (and until end of line) is treated as comments, *e.g.*,

```
>> x = 1:2:9; % x = [1 3 5 7 9];
```

- ; -- delimits statements; suppresses screen output, *e.g.*,

```
>> x = 1:2:9; y = 2:10; % two statements on the same line
```

- ... -- statement continuation, *e.g.*,

```
>> x = [ 1 3 5 ...
7 9]; % x = [1 3 5 7 9] splitted into 2 lines
```

- : -- range delimiter, *e.g.*,

```
>> x = [1:2:9]; % x=[1,3,5,7,9]
```

- ' -- matrix transposition, *e.g.*,

```
>> x = [1:2:9]'; % x changed from row vector to column vector
```

If the vector/matrix is complex, " ' " results in complex conjugation ***and*** *matrix transposition.*

- , -- command delimiter, *e.g.*,

```
>> x = [1:2:9], y = [1:9] % two statements on the same line
```

- . -- precede an arithmetic operator to perform an elemental operation, instead of matrix operation, *e.g.*,

```
>> x = 3:3:9

x =

    3     6     9

>> y = 3*ones(3,1)'

y =

    3     3     3
```

>> z =x./y

z =

1 2 3

- * -- "wild card", *e.g.*,

>> clear A* % clears all variables that start with A.

Note that many of these characters have multiple functionalities (function overloading) depending on the context, *e.g.*, "*" is used for scalar multiply, matrix multiply and "wild card" as seen above.

Arithmetic Operations & Built in functions:

#Example1

Find $y = \exp(5^2 / 3pi)$.

Solution: In m-file write the following command

```
clear all;
a=5^2;
b=3*pi;
y=exp(a/b);
disp(y)
```

#Save the file and "run" the program from "Debug" menu. Type "y" in command window and press "enter".
# Remove ";" from all the lines and run the program.
# Write each line in command window and press "enter" after each line.
# Variable names are assigned to expressions by using equal sign. For example, a=5^2; here "a" is the variable that store the value of 5^2 or 25.
# See the list of built in functions from "help" menu. Some built in functions are
abs() cos() sin() exp() log() real() sqrt() floor() ceil()

Matrices:

Write A= [1 2 3; 4 5 6; 7 6 3] in command window and press "enter". It is a 3×3 matrix.
Write A(1,3) in command window to view the 3rd element in 1st row. The first parameter within bracket denotes row and the second parameter denotes column.
Z=zeros(2,3) creates a 2×3 matrix of zeros. Similarly ones(), eye() create special types of matrices.
Write A=0:0.3:3 in command window. 0 is the starting value, 3 is the end value and 0.3 is the step value.
Write "help size", "help prod" and "help length" in command window.

Matrix Operations:

All arithmetic operations can be performed on matrices.
Operations can be performed on each element or on whole matrix. For example,

```
>> x = 3:3:9

>> y = 3*ones(3,1)'

>> z =x./y
```

Some operations are performed on square matrices only.

+ - * / ^ (algebraic/matrix definitions)
.+ .- .* ./ .^ (element by element operation)

Additionally,

" ' " performs matrix transposition; when applied to a complex matrix, it includes elemental conjugations followed by a matrix transposition
\ and .\ perform matrix and elemental left division

Report:

#Exercise 1.

Find the value of $y = \ln(\sinh(\exp(54^3 / 6 * pi)))$

 # Exercise 2:

Find the size, and length of following matrices
A=[1 2 3; 4 5 6;7 6 54; 65 23 45]
B=7:1:13.5
# Write A(1:2,2:3) in command window. Write A([1 2],[2 3]). These are different ways to select a submatrix of a matrix.
#A(1,1)=sin(5); assign a new value to an element of A.

#Exercise 3.

A=[2 3; 4 5]; B=[3 4; 6 7];
Find A+B, A*B, A.*B,A/B,A\B, A.^2,A./B

 # Exercise 4 .

Define the matrices

```
A=[17 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
B=[ 2 3 4 5 ; 6 7 8 9 ; 10 11 12 13 ; 14 15 16 17 ]
C=[ 1 2 3 ; 4 5 6 ; 7 8 9 ]
y=[ 4 3 2 1 ]'
```

Note the transpose `'` on the `y`-vector which makes `y` a column vector.

a) Compute *AB* and *BA*. Is matrix multiplication commutative?

b) Compute *AC*. Why do you get an error message?

c) Solve the following system of equations:

$$17x_1+2x_2+3x_3+4x_4 = 4$$
$$5x_1+6x_2+7x_3+8x_4 = 3$$
$$9x_1+10x_2+11x_3+12x_4 = 2$$
$$13x_1+14x_2+15x_3+16x_4 = 1$$

Experiment 2: Applications of MATLAB

Graphics:

MATLAB can produce 2 and 3 dimensional plots.

MATLAB is an interactive environment in which you can program as well as visualize your computations. It includes a set of high-level graphical functions for:

- Line plots(plot, plot3, polar)
- Bar graphs(bar, barh, bar3, bar3h, hist, rose, pie, pie3)
- Surface plots(surf, surfc)
- Mesh plots(mesh, meshc, meshgrid)
- Contour plots(contour, contourc, contourf)
- Animation(moviein, movie)

#Example 2.

```
x=0:0.1:pi;
   y=cos(x);
plot(y);
  plot(x,cos(x),'r');
     plot(x,y,x,y.^2);
```

#Example 3.

```
x=0:0.1:pi;
  y=cos(x);
   plot(y);
     hold on
       plot(x,cos(x),'r');
```

#Example 4.

```
x=linspace(0,7);
 y=exp(x);
  subplot(2,1,1), plot(x,y);
    subplot(2,1,2), semilogy(x,y);
```

 # Example 5.

```
x = magic(3);
   bar(x);
      grid
```

Loops & Conditionals:

#MATLAB has the following flow control constructs:
• if statements
• switch statements
• for loops
• while loops
• break statements

The if, for, switch and while statements need to terminate with an end statement.
Example:

#Example 6. IF:

```
    x=-3;
if x>0
  a=10;
   elseif x<0
      a=11;
   elseif x= = 0
    a=12;
     else
       a=14;
end
```

What is the value of 'a' after execution of the above code?

#Example 7. WHILE:

```
   x=-10;
while x<0
     x=x+1;
end
```

What is the value of x after execution of the above loop?

#Example 8. FOR loop:

```
   x=0;
for i=1:10
     x=x+1;
end
```

What is the value of x after execution of the above loop?

Defining matrices via the vector technique

Using the for loop in MATLAB is relatively expensive. It is much more efficient to perform the same task using the vector method. For example, the following task

```
for j=1:n
   for i=1:m
     A(i,j) = B(i,j) + C(i,j);
   end
end
```

can be more compactly and efficiently represented (and computed) by the vector method as follows:

```
A(1:m,1:n) = B(1:m,1:n) + C(1:m,1:n);
```

If the matrices are all of the same size (as is the case here), then the above can be more succinctly written as

```
A = B + C;
```

For sufficiently large matrix operations, this latter method is vastly superior in performance.

#Example 9. BREAK:

The break statement lets you exit early from a for or a while loop:

```
     x=-10;
while x<0
       x=x+2;
    if x = = -2
       break;
    end
end
```

What is the value of x after execution of the above loop?

Relational Operators

Symbol	Meaning
<=	Lessthanequal
<	Less than
>=	Greater than equal
>	Greater than
==	Equal
~ =	Not equal

Logical Operators

Symbol	Meaning
&	AND
|	OR
~	NOT

Defining functions:

In MATLAB there is scope for user-defined functions.

Suggestion: Since MATLAB distinguishes one function from the next by their file names, name files the same as function names to avoid confusion. Use only lowercase letter to be consistent with MATLAB's convention.

To define a function, start a new M-file
The first line of M-file should be

```
function  variable_name=function_name(parameters);
```

variable_name is the name of variable whose value will be returned.
function_name is user defined according to the rules stated previously.

#Example 10:

```
function y=cal_pow(x);

y=1+x^2;

end
```

Save this function as cal_pow.
Start another new M-file .This will be our main file.
Write the following commands and run the file:

```
clear all;
   x=0:1:3;
     t=length(x);

for i=1:t
    val(i)=cal_pow(x(i));
end

plot(x,val);
```

Report:

#**Exercise1**.
Plot the following functions in the same window y1=sin x, y2=sin 2x, y3=sin 3x, y4=sin 4x where x varies from 0 to pi.

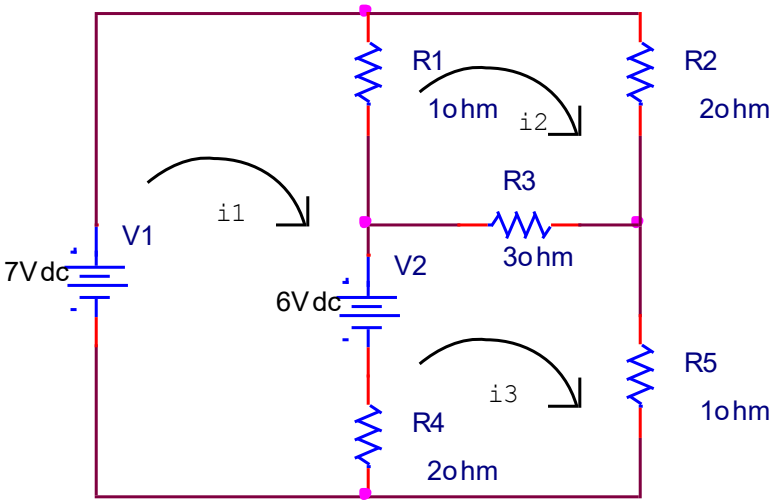
Exercise 2.
Write a program to compute the variance of an array **x**. The variance σ is defined to be:

$$\sigma = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

where $\bar{x}$ is the average of the array **x**.

For **x**, use all the integers from 1 to 1000.

 # Exercise 3 .
Solve the following circuit to find i1, i2, and i3.



Ans: i1= 3 amp, i2= 2 amp, i3= 3 amp.

Experiment 3: Curve Fitting

Introduction:

Data is often given for discrete values along a continuum. However we may require estimates at points between the discrete values. Then we have to fit curves to such data to obtain intermediate estimates. In addition, we may require a simplified version of a complicated function. One way to do this is to compute values of the function at a number of discrete values along the range of interest. Then a simpler function may be derived to fit these values. Both of these applications are known as **curve fitting**.

There are two general approaches of curve fitting that are distinguished from each other on the basis of the amount of error associated with the data. First, where the data exhibits a significant degree of error, the strategy is to derive a single curve that represents the general trend of the data. Because any individual data may be incorrect, we make no effort to intersect every point. Rather, the curve is designed to follow the pattern of the points taken as a group. One approach of this nature is called ***least squares regression***.

Second, where the data is known to be very precise, the basic approach is to fit a curve that passes directly through each of the points. The estimation of values between well known discrete points from the fitted exact curve is called ***interpolation***.

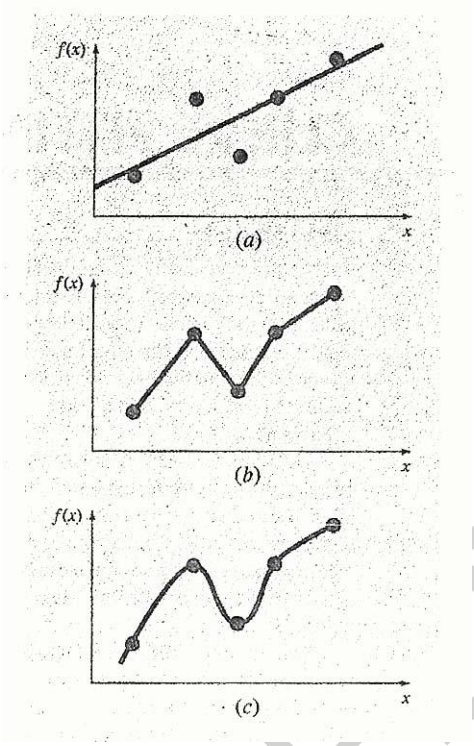


Figure 1: (a) Least squares linear regression (b) linear interpolation (c) curvilinear interpolation

Least squares Regression:

Where substantial error is associated with data, polynomial interpolation is inappropriate and may yield unsatisfactory results when used to predict intermediate values. A more appropriate strategy for such cases is to derive an approximating function that fits the shape or general trend of the data without necessarily matching the individual points. Now some criterion mush be devised to establish a basis for the fit. One way to do this is to derive a curve that minimizes the discrepancy between the data points and the curve. A technique for accomplishing this objective is called least squares regression, where the goal is to minimize the sum of the square errors between the data points and the curve. Now depending on whether we want to fit a straight line or other higher order polynomial, regression may be linear or polynomial. They are described below.

Linear regression:

The simplest example of least squares regression is fitting a straight line to a set of paired observations: (x1, y1), (x2, y2), , , ,(xn, yn). The mathematical expression for straight line is

$y_m=a_0+a_1x$

Where a_0 and a_1 are coefficients representing the intercept and slope and y_m is the model value. If y_0 is the observed value and e is error or residual between the model and observation then

$e=y_0-y_m=y_0 - a_0 - a_1x$

Now we need some criteria such that the error e is minimum and also we can arrive at a unique solution (for this case a unique straight line). One such strategy is to minimize the sum of the square errors. So sum of square errors

$$S_r = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_{i,observed} - y_{i,model})^2 = \sum_{i=1}^{n} (y_i - a_0 - a_1 x_i)^2 \quad \text{.....1}$$

To determine the values of a_0 and a_1, equation (1) is differentiated with respect to each coefficient.

$$\frac{\partial S_r}{\partial a_0} = -2\sum (y_i - a_0 - a_1 x_i)$$

$$\frac{\partial S_r}{\partial a_1} = -2\sum (y_i - a_0 - a_1 x_i) x_i$$

Setting these derivatives equal to zero will result in a minimum Sr. If this is done, the equations can be expressed as

$$0 = \sum y_i - \sum a_0 - \sum a_1 x_i$$

$$0 = \sum y_i x_i - \sum a_0 x_i - \sum a_1 x_i^2$$

Now realizing that $\sum a_0 = n a_0$ **,** we can express the above equations as a set of two simultaneous linear equations with two unknowns a_0 and a_1.

$$n a_0 + (\sum x_i) a_1 = \sum y_i$$

$$(\sum x_i) a_0 + (\sum x_i^2) a_1 = \sum x_i y_i$$

from where

$$a_1 = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}$$

$$a_0 = \bar{y} - a_1 \bar{x}$$

Where $\bar{y}$ and $\bar{x}$ are the means of y and x respectively

Example 1:
Fit a straight line to the x and y values of table 1

Table 1:

x	y
1	0.5
2	2.5
3	2.0
4	4.0
5	3.5
6	6.0
7	5.5

Ans: a_0=0.071142857, a_1=0.83928

```
%program for fitting a straight line
%entering no. of observed points
n=input('How many points ');

%taking input
for i=1:n
   x(i)=input('');
   y(i)=input('');
end

%calculating coefficients
sumx=0;
sumy=0;
sumxy=0;
sumxsq=0;
for i=1:n
sumx=sumx+x(i);
sumy=sumy+y(i);
sumxy=sumxy+x(i)*y(i);
sumxsq=sumxsq+x(i)^2;
end

format long ;

%calculating a1 and a0
a1=(n*sumxy-sumx*sumy)/(n*sumxsq-sumx^2)
a0=sumy/n-a1*sumx/n

%plotting observed data
```

```
plot(x,y,'o')
hold on;

%plotting fitted data
ym=a0+a1.*x;
plot(x,ym);
```

Polynomial Regression:

In some cases, we have some engineering data that cannot be properly represented by a straight line. We can fit a polynomial to these data using polynomial regression.

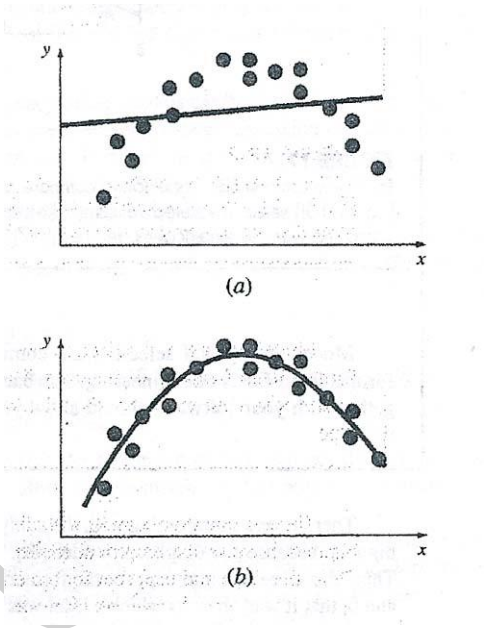


Figure 2: (a) Data that is ill-suited for linear least squares regression (b) indication that a parabola is preferable

The least squares procedure can be readily extended to fit the data to a higher order polynomial. For example, we want to fit a second order polynomial

$y_m=a_0 + a_1x+ a_2x^2$

For this case the sum of the squares of residuals is

$$S_r = \sum_{i=1}^{n}(y_i - a_0 - a_1x_i - a_2x_i^2)^2 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 2$$

Taking derivative of equation (2) with respect to unknown coefficients a_0, a_1 and a_2

$$\frac{\partial S_r}{\partial a_0} = -2\sum (y_i - a_0 - a_1 x_i - a_2 x_i^2)$$
$$\frac{\partial S_r}{\partial a_1} = -2\sum x_i (y_i - a_0 - a_1 x_i - a_2 x_i^2)$$
$$\frac{\partial S_r}{\partial a_2} = -2\sum x_i^2 (y_i - a_0 - a_1 x_i - a_2 x_i^2)$$

These equations can be set equal to zero and rearranged to develop the following set of normal equations:

$$na_0 + (\sum x_i)a_1 + (\sum x_i^2)a_2 = \sum y_i$$
$$(\sum x_i)a_0 + (\sum x_i^2)a_1 + (\sum x_i^3)a_2 = \sum x_i y_i$$
$$(\sum x_i^2)a_0 + (\sum x_i^3)a_1 + (\sum x_i^4)a_2 = \sum x_i^2 y_i$$

Now a_0, a_1 and a_2 can be calculated using matrix inversion.

Linearization of Nonlinear Relationships:

Linear regression is a powerful technique for fitting a best line to data. However it is dependent on the fact that the relationship between the dependent and independent variables should be linear. This is always not the case. In those cases, we use polynomial regression. In some cases, transformation can be used to express the data in a form that is compatible with linear regression.

One example is the exponential model

$$y = a_1 e^{b_1 x} \quad \text{……..3}$$

Where a_1 and b_1 are constants.

Another example of a nonlinear model is the simple power equation

$$y = a_2 x^{b_2} \quad \text{……..4}$$

Where a_2 and b_2 are constants.

Nonlinear regression techniques are available to fit these equations to experimental data directly. However, a simpler alternative is to use mathematical manipulations to transform the equations into linear forms. Then simple linear regression can be used to fit the equations to data.

For example equation (3) can be linearized by taking its normal logarithm to yield

$$\ln y = \ln a_1 + b_1 x$$

Thus a plot of lny vs x will yield a straight line with a slope of b_1 and an intercept of $\ln a_1$

Equation (4) can be linearized by taking its base10 logarithm to give

$$\log y = b_2 \log x + \log a_2$$

Thus a plot of logy vs logx will yield a straight line with a slope of b_2 and an intercept of $\log a_2$

Report:

 # Exercise 1:
Fit a second order polynomial to the data given in table 2

Table 2

x	y
0	2.1
1	7.7
2	13.6
3	27.2
4	40.9
5	61.1

Ans: a_0=2.47857, a_1=2.35929, a_2=1.86071

 # Exercise 2:

Fit the equation $y = a_2 x^{b_2}$ to the data given in Table 3

Table 3

x	y
1	0.5
2	1.7
3	3.4
4	5.7
5	8.4

Ans: a_2=0.5, b_2=1.75

Hints: find logx and logy for all points. Using these converted points, using linear regression find slope b_2 and intercept loga_2. Then find a_2 and b_2.

Experiment 04: Solution of Simultaneous Linear Algebraic Equations

Objective

Systems of linear algebraic equations occur often in diverse fields of science and engineering and is an important area of study. In this experiment we will be concerned with the different techniques of finding solution of a set of n linear algebraic equations in n unknowns.

Concept of linear equations and their solution

A set of linear algebraic equations looks like this:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots a_{2N}x_N &= b_2 \\ \ldots \quad \ldots \quad \ldots \quad \ldots & \\ a_{M1}x_1 + a_{M2}x_2 + \ldots a_{MN}x_N &= b_M \end{aligned} \qquad (1)$$

Here the N unknowns x_j , j = 1, 2, . . .,N are related by M equations. The coefficients a_{ij} with i = 1, 2, . . .,M and j = 1, 2, . . .,N are known numbers, as are the right-hand side quantities b_i, i = 1, 2, . . .,M.

Existence of solution

If N = M then there are as many equations as unknowns, and there is a good chance of solving for a unique solution set of x_j's. Analytically, there can fail to be a unique solution if one or more of the M equations is a linear combination of the others (This condition is called row degeneracy), or if all equations contain certain variables only in exactly the same linear combination(This is called column degeneracy). (For square matrices, a row degeneracy implies a column degeneracy, and vice versa.) A set of equations that is degenerate is called singular.

Numerically, at least two additional things can go wrong:

• While not exact linear combinations of each other, some of the equations may be so close to linearly dependent that round off errors in the machine renders them linearly dependent at some stage in the solution process. In this case your numerical procedure will fail, and it can tell you that it has failed.

• Accumulated round off errors in the solution process can swamp the true solution. This problem particularly emerges if N is too large. The numerical procedure does not fail algorithmically. However, it returns a set of x's that are wrong, as can be discovered by direct substitution back into the original equations. The closer a set of equations is to being singular, the more likely this is to happen.

Matrices

Equation (1) can be written in matrix form as

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \qquad (2)$$

Here the raised dot denotes matrix multiplication, **A** is the matrix of coefficients, **x** is the column vector of unknowns and **b** is the right-hand side written as a column vector,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1N} \\ a_{21} & a_{22} & \ldots & a_{2N} \\ \ldots & \ldots & \ldots & \ldots \\ a_{M1} & a_{M2} & \ldots & a_{MN} \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ .. \\ x_N \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ .. \\ b_M \end{bmatrix}$$

Finding Solution

There are so many ways to solve this set of equations. Below are some important methods.

(1) Using the backslash and pseudo-inverse operator

In MATLAB, the easiest way to determine whether Ax = b has a solution, and to find such a solution when it does, is to use the backslash operator. Exactly what $\mathbf{A} \backslash \mathbf{b}$ returns is a bit complicated to describe, but if there is a solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, then $\mathbf{A} \backslash \mathbf{b}$ returns one. Warnings: (1) $\mathbf{A} \backslash \mathbf{b}$ returns a result in many cases when there is no solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. (2) $\mathbf{A} \backslash \mathbf{b}$ sometimes causes a warning to be issued, even when it returns a solution. This means that you can't just use the backslash operator: you have to check that what it returns is a solution. (In any case, it's just good common sense to check numerical computations as you do them.) In MATLAB this can be done as follows:

Using backslash operator:
x = A\b;

You can also use the pseudo-inverse operator:
x=pinv(A)*b; % it is also guaranteed to solve Ax = b, if Ax = b has a solution.

As with the backslash operator, you have to check the result.

(2) Using Gauss-Jordan Elimination and Pivoting

To illustrate the method let us consider three equations with three unknowns:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = a_{14} \qquad (A)$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = a_{24} \qquad (B)$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = a_{34} \qquad (C)$$

Here the quantities b_i, i = 1, 2, . . .,M's are replaced by a_{iN+1}, where i=1,2, ….M for simplicity of understanding the algorithm.

The First Step is to eliminate the first term from Equations (B) and (C). (Dividing (A) by a_{11} and multiplying by a_{21} and subtracting from (B) eliminates x_1 from (B) as shown below)

$$(a_{21}-\frac{a_{11}}{a_{11}}a_{21})x_1+(a_{22}-\frac{a_{12}}{a_{11}}a_{21})x_2+(a_{23}-\frac{a_{13}}{a_{11}}a_{21})x_3=(a_{24}-\frac{a_{14}}{a_{11}}a_{21})$$

Let, $\frac{a_{21}}{a_{11}}=k_2$, then

$$(a_{21}-k_2a_{11})x_1+(a_{22}-k_2a_{12})x_2+(a_{23}-k_2a_{13})x_3=(a_{24}-k_2a_{14})$$

Similarly multiplying equation (A) by $\frac{a_{31}}{a_{11}}=k_3$ and subtracting from (C), we get

$$(a_{31}-k_3a_{11})x_1+(a_{32}-k_3a_{12})x_2+(a_{33}-k_3a_{13})x_3=(a_{34}-k_3a_{14})$$

Observe that $(a_{21}-k_2a_{11})$ and $(a_{31}-k_3a_{11})$ are both zero.

In the steps above it is assumed that a_{11} is not zero. This case will be considered later in this experiment.

The above elimination procedure is called triangularization.

Algorithm for triangularizing n equations in n unknowns:

```
1       for i = 1 to n and j = 1 to (n+1) in steps of 1 do read a_ij endfor
2       for k = 1 to (n-1) in steps of 1 do
3           for i = (k+1) to n in steps of 1 do
4           u ← a_ik / a_kk
5           for j = k to (n+1) in steps of 1 do
6               a_ij ← a_ij - u a_kj endfor
            endfor
        endfor
```

The reduced equations are:

$$a_{11}x_1+a_{12}x_2+a_{13}x_3=a_{14}$$
$$a_{22}x_2+a_{23}x_3=a_{24}$$
$$a_{32}x_2+a_{33}x_3=a_{34}$$

The next step is to eliminate a_{32} from the third equation. This is done by multiplying second equation by $u=a_{32}/a_{22}$ and subtracting the resulting equation from the third. So, same algorithm can be used.

Finally the equations will take the form:

$$a_{11}x_1+a_{12}x_2+a_{13}x_3=a_{14}$$
$$a_{22}x_2+a_{23}x_3=a_{24}$$
$$a_{33}x_3=a_{34}$$

The above set of equations are said to be in triangular (Upper) form.

From the above upper triangular form of equations, the values of unknowns can be obtained by back substitution as follows:

$$x_3 = a_{34} / a_{33}$$

$$x_2 = (a_{24} - a_{23}x_3) / a_{22}$$

$$x_2 = (a_{14} - a_{12}x_2 - a_{13}x_3) / a_{11}$$

Algorithmically, the back substitution for n unknowns is shown below:

1 $x_n \leftarrow a_{n(n+1)} / a_{nn}$

2 *for* $i = (n-1)$ to 1 in step of -1 *do*

3 $sum \leftarrow 0$

4 *for* $j = (i+1)$ to n in steps of 1 *do*

5 $sum \leftarrow sum + a_{ij}x_j$ *endfor*

6 $x_i \leftarrow (a_{i(n+1)} - sum) / a_{ii}$

endfor

Pivoting

In the triangularization algorithm we have used,

$$u \leftarrow a_{ik} / a_{kk}$$

Here it is assumed that a_{kk} is not zero. If it happens to be zero or nearly zero, the algorithm will lead to no results or meaningless results. If any of the a_{kk} is small it would be necessary to reorder the equations. It is noted that the value of a_{kk} would be modified during the elimination process and there is no way of predicting their values at the start of the procedure.

The elements a_{kk} are called pivot elements. In the elimination procedure the pivot should not be zero or a small number. In fact for maximum precision the pivot element should be the largest in absolute value of all the elements below it in its column, i.e. a_{kk} should be picked up as the maximum of all a_{mk} where, $m \geq k$

So, during the Gauss elimination, a_{mk} elements should be searched and the equation with the maximum value of a_{mk} should be interchanged with the current position. For example if during elimination we have the following situation:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 4 \\ 0.3x_2 + 4x_3 &= 5 \\ -8x_2 + 3x_3 &= 6 \end{aligned}$$

As $|-8| > 0.3$, 2nd and 3rd equations should be interchanged to yield:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 4 \\ -8x_2 + 3x_3 &= 6 \\ 0.3x_2 + 4x_3 &= 5 \end{aligned}$$

It should be noted that interchange of equations does not affect the solution.

The algorithm for picking the largest element as the pivot and interchanging the equations is called pivotal condensation.

Algorithm for pivotal condensation

```
1       max ← |a_kk|
2       p ← k
3       for m = (k+1) to n in steps of 1 do
4           if (|a_mk| > max) then
5           max ← |a_mk|
6           p ← m
7           endif
        endfor
8       if (p ~= k)
9           for q = k to (n+1) in steps of 1 do
10              temp ← a_kq
```

11 $a_{kq} \leftarrow a_{pq}$

12 $a_{pq} \leftarrow temp$

endfor

endif

(3) Using Gauss-Seidel Iterative Method

There are several iterative methods for the solution of linear systems. One of the efficient iterative methods is the Gauss-Seidel method.

Let us consider the system of equations:

$$\begin{aligned} 4x_1 - x_2 + x_3 &= 7 \\ 4x_1 - 8x_2 + x_3 &= -21 \\ -2x_1 + x_2 + 5x_3 &= 15 \end{aligned}$$

The Gauss-Seidel iterative process is suggested by the following equations:

$$x_1^{k+1} = \frac{7 + x_2^k - x_3^k}{4}$$

$$x_2^{k+1} = \frac{21 + 4x_1^{k+1} + x_3^k}{8}$$

$$x_3^{k+1} = \frac{15 + 2x_1^{k+1} - x_2^{k+1}}{5}$$

The very first iteration, that is $x_2^0, x_3^0, ...x_n^0$ (for n equations) are set equal to zero and x_1^1 is calculated. The main point of Gauss-Seidel iterative process to observe is that always the latest approximations for the values of variables are used in an iteration step.

It is to be noted that in some cases the iteration diverges rather than it converges. Both the divergence and convergence can occur even with the same set of equations but with the change in the order. The sufficient condition for the Gauss-Seidel iteration to converge is stated below.

The Gauss-Seidel iteration for the solution will converge (if there is any solution) if the matrix **A** (as defined previously) is strictly diagonally dominant matrix.

A matrix **A** of dimension $N \times N$ is said to be strictly diagonally dominant provided that

$$|a_{kk}| > \sum_{\substack{j=1 \\ j \neq k}}^{N} |a_{kj}| \text{ for } k = 1, 2, \ldots N$$

Report:

 # Exercise 1.

Given the simultaneous equations shown below (i) triangularize them (ii) use back substitution to solve for x_1, x_2, x_3.

$$\begin{aligned} 2x_1 + 3x_2 + 5x_3 &= 23 \\ 3x_1 + 4x_2 + x_3 &= 14 \\ 6x_1 + 7x_2 + 2x_3 &= 26 \end{aligned}$$

For generalization, you will have to write a program for triangularizing n equations in n unknowns with back substitution.

#Exercise 2.

Modify the MATLAB program written in exercise 1 to include pivotal condensation.

#Exercise 3.

Try to solve the following systems of equations (i) Gauss-Jordan elimination (ii) Gauss-Jordan elimination with pivoting

(A)
$$\begin{aligned} 2x_1 + 4x_2 - 6x_3 &= -4 \\ x_1 + 5x_2 + 3x_3 &= 10 \\ x_1 + 3x_2 + 2x_3 &= 5 \end{aligned}$$

(B)
$$\begin{aligned} x_1 + x_2 + 6x_3 &= 7 \\ -x_1 + 2x_2 + 9x_3 &= 2 \\ x_1 - 2x_2 + 3x_3 &= 10 \end{aligned}$$

(C)
$$\begin{aligned} 4x_1 + 8x_2 + 4x_3 &= 8 \\ x_1 + 5x_2 + 4x_3 - 3x_4 &= -4 \\ x_1 + 4x_2 + 7x_3 + 2x_4 &= 10 \\ x_1 + 3x_2 - 2x_4 &= -4 \end{aligned}$$

#Exercise 4.

Solve the following equations using Gauss-Seidel iteration process:

(A)
$$\begin{aligned} 8x_1 - 3x_2 &= 10 \\ -x_1 + 4x_2 &= 6 \end{aligned}$$

(B)
$$\begin{aligned} 4x - y &= 15 \\ x + 5y &= 9 \end{aligned}$$

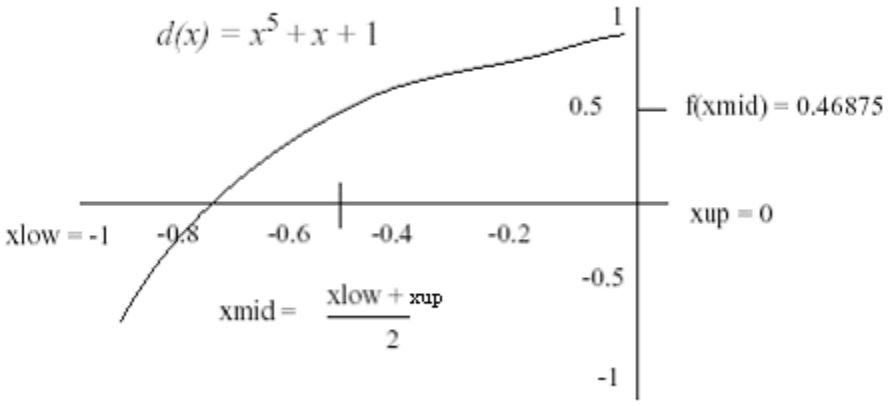
(C)
$$\begin{aligned} 5x_1 - x_2 + x_3 &= 10 \\ 2x_1 + 8x_2 - x_3 &= 11 \\ -x_1 + x_2 + 4x_3 &= 3 \end{aligned}$$

(D)
$$\begin{aligned} 2x + 8y - z &= 11 \\ 5x - y + z &= 10 \\ -x + y + 4z &= 3 \end{aligned}$$

Experiment 5: Solutions to Non-Linear Equations
(Bisection Method & False-Position Method)

Bisection method:

The Bisection method is one of the simplest procedures for finding root of a function in a given interval.
The procedure is straightforward. The approximate location of the root is first determined by finding two values that bracket the root (a root is bracketed or enclosed if the function changes sign at the endpoints). Based on these a third value is calculated which is closer to the root than the original two value. A check is made to see if the new value is a root. Otherwise a new pair of bracket is generated from the three values, and the procedure is repeated.



Consider a function $d(x)$ and let there be two values of x, x_{low} and x_{up} ($x_{up} > x_{low}$), bracketing a root of $d(x)$.

Steps:

1. The first step is to use the brackets x_{low} and x_{up} to generate a third value that is closer to the root. This new point is calculated as the mid-point between x_{low} and, namely $x_{mid} = \frac{x_{low} + x_{up}}{2}$. The method therefore gets its name from this bisecting of two values. It is also known as interval halving method.

2. Test whether x_{mid} is a root of $d(x)$ by evaluating the function at x_{mid}.
3. If x_{mid} is not a root,
   a. If $d(x_{low})$ and $d(x_{mid})$ have opposite signs i.e. $d(x_{low}).d(x_{mid})<0$, root is in left half of interval.
   b. If $d(x_{low})$ and $d(x_{mid})$ have same signs i.e. $d(x_{low}).d(x_{mid})>0$, root is in right half of interval.
4. Continue subdividing until interval width has been reduced to a size ≤ ε where ε = selected *x* tolerance.

Algorithm: Bisection Method

```
Input xLower, xUpper, xTol
yLower = f(xLower) (* invokes fcn definition *)
xMid = (xLower + xUpper)/2.0
yMid = f(xMid)
iters = 0 (* count number of iterations *)
While ( (xUpper - xLower)/2.0 > xTol )
iters = iters + 1
if( yLower * yMid > 0.0) Then xLower = xMid
Else xUpper = xMid
Endofif
xMid = (xLower + xUpper)/2.0
yMid = f(xMid)
Endofwhile
Return xMid, yMid, iters (* xMid = approx to root *)
```

Note: For a given *x* tolerance (epsilon), we can calculate the number of iterations directly. The number of divisions of the original interval is the smallest value of *n* that satisfies: $\frac{x_{up}-x_{low}}{2^n} \langle \varepsilon \text{ or } 2^n \rangle \frac{x_{up}-x_{low}}{\varepsilon}$

Thus $n \rangle \log 2\left(\frac{x_{up}-x_{low}}{\varepsilon}\right)$.

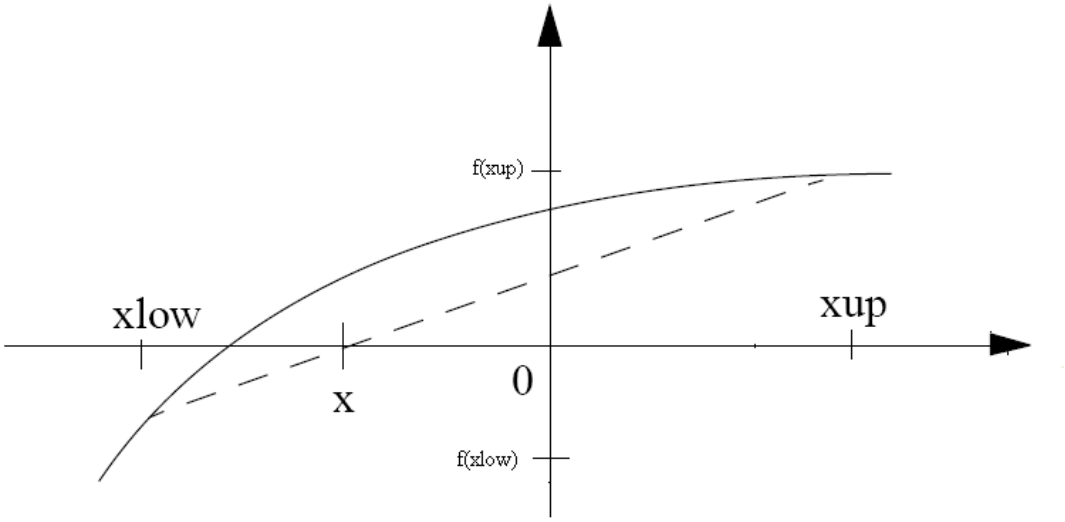
In our previous example, x_{low} = -1, x_{up} =0 and ε = selected *x* tolerance = 10^{-4}.

So we have $n = 14$.

False-Position Method (Regula Falsi)

A shortcoming of the bisection method is that, in dividing the interval from x_{low} to x_{up} into equal halves, no account is taken of the magnitude of $f(x_{low})$ and $f(x_{up})$. For example, if $f(x_{low})$ is much closer to zero than $f(x_{up})$, it is likely that the root is closer to x_{low} than to x_{up}. An alternative method that exploits this graphical insight is to join $f(x_{low})$ and $f(x_{up})$ by a straight line. The intersection of this line with the x axis

represents an improved estimate of the root. The fact that the replacement of the curve by a straight line gives the false position of the root is the origin of the name, method of false position, or in Latin, Regula Falsi. It is also called the Linear Interpolation Method.



Using similar triangles, the intersection of the straight line with the x axis can be estimated as

$$\frac{f(x_{low})}{x - x_{low}} = \frac{f(x_{up})}{x - x_{up}}$$

That is $x = x_{up} - \dfrac{f(x_{up})(x_{low} - x_{up})}{f(x_{low}) - f(x_{up})}$

This is the False Position formulae. The value of x then replaces whichever of the two initial guesses, x_{low} or x_{up}, yields a function value with the same sign as $f(x)$. In this way, the values of x_{low} and x_{up} always bracket the true root. The process is repeated until the root is estimated adequately.

Report:

#Exercise 1.

Find the real root of the equation $d(x) = x^5 + x + 1$ using Bisection Method. x_{low} = -1, x_{up} =0 and ε = selected x tolerance $=10^{-4}$.

#Exercise 2.

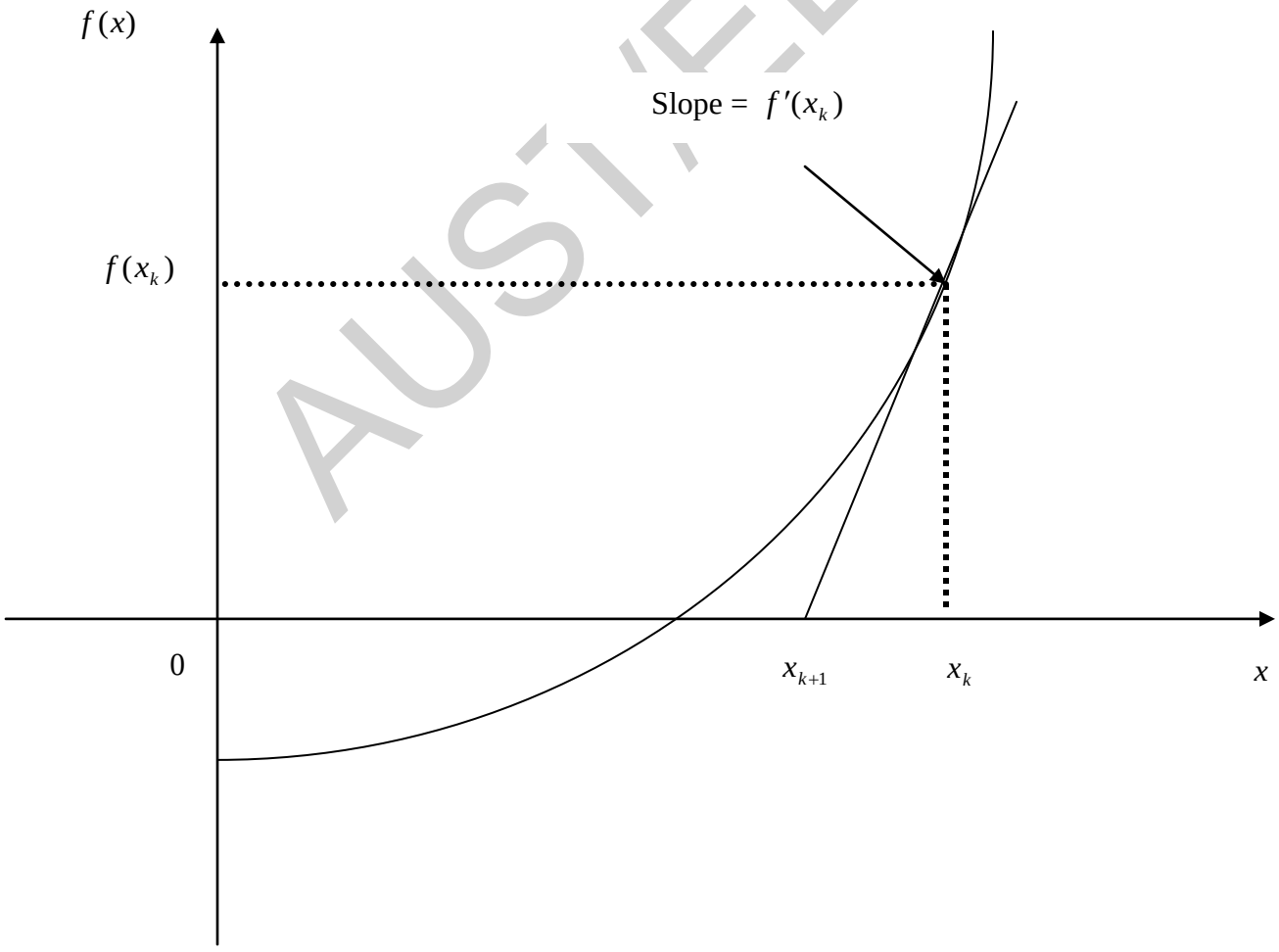
Find the root of the equation $d(x) = x^5 + x + 1$ using False Position Method. x_{low} = -1, x_{up} =0 and ε = selected x tolerance $=10^{-4}$. (Develop the algorithm by yourself. It is very similar to Bisection Method).

Experiment 6: Solutions to Non-Linear Equations
(Newton Raphson Method & Secant Method)

Newton Raphson Method:

If $f(x)$, $f'(x)$ and $f''(x)$ are continuous near a root x, then this extra information regarding the nature of $f(x)$ can be used to develop algorithms that will produce sequences $\{x_k\}$ that converge faster to x than either the bisection or false position method. The Newton-Raphson (or simply Newton's) method is one of the most useful and best-known algorithms that relies on the continuity of $f'(x)$ and $f''(x)$.

The attempt is to locate root by repeatedly approximating $f(x)$ with a linear function at each step. If the initial guess at the root is x_k, a tangent can be extended from the point $[x_k, f(x_k)]$. The point where this tangent crosses the x axis usually represents an improved estimate of the root.



The Newton-Raphson method can be derived on the basis of this geometrical interpretation. As in the figure, the first derivative at x is equivalent to the slope:

$$f'(x_k) = \frac{f(x_k) - 0}{x_k - x_{k+1}}$$

which can be rearranged to yield

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

which is called the Newton Raphson formulae.

So the Newton-Raphson Algorithm actually consists of the following steps:

1. Start with an initial guess x_0 and an x-tolerance ε.

2. Calculate $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ $k = 0,1,2,\ldots$

Algorithm - Newton's Method

```
Input x0, xTol
iters = 1
dx = -f(x0)/fDeriv(x0) (* fcns f and fDeriv *)
root = x0 + dx
While (Abs(dx) > xTol)
dx = -f(root)/fDeriv(root)
root = root + dx
iters = iters + 1
End of while
Return root, iters
```

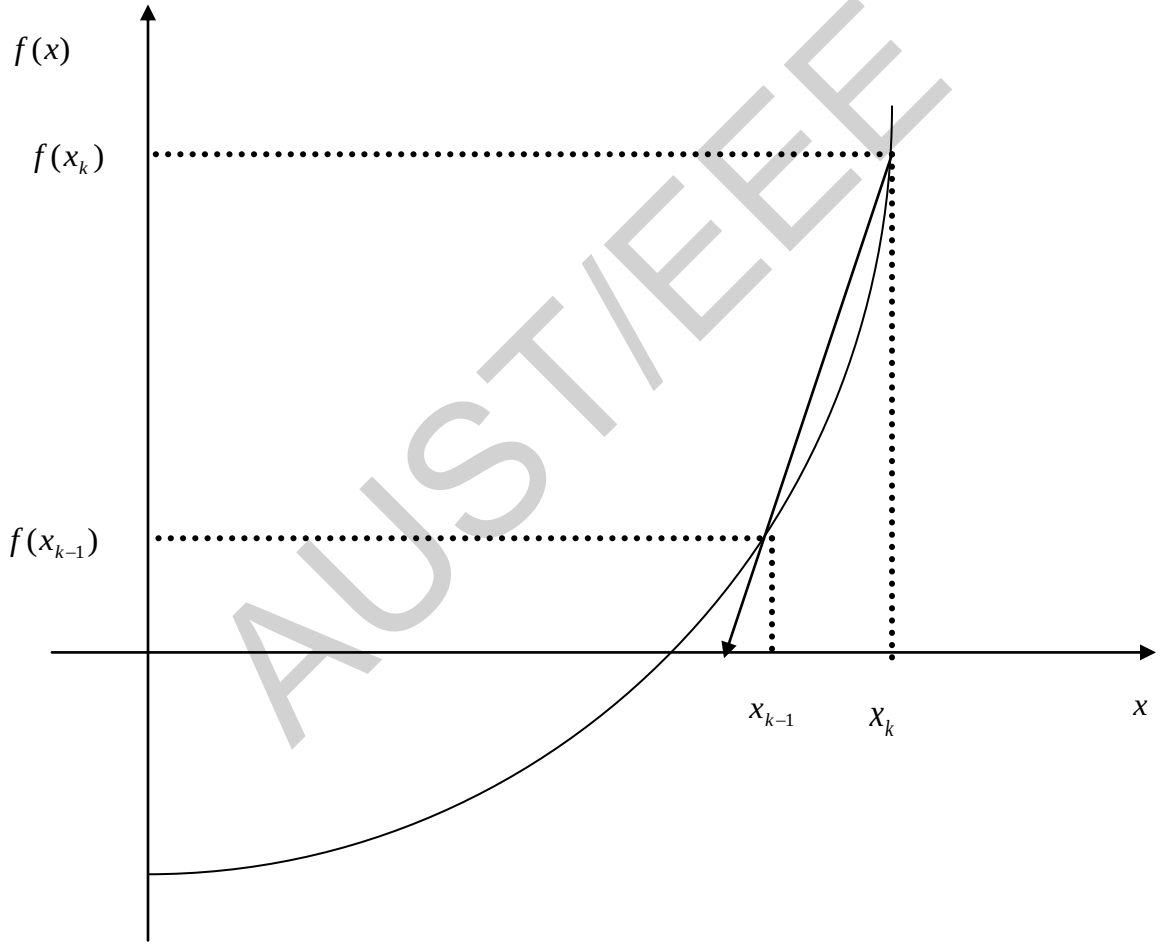
The Secant Method:

The Newton-Raphson algorithm requires two functions evaluations per iteration, $f(x_k)$ and $f'(x_k)$. Historically, the calculation of a derivative could involve considerable effort. Moreover, many functions have non-elementary forms (integrals, sums etc.), and it is desirable to have a method for finding a root that does not depend on the computation of a derivative. The secant method does not need a formula for the derivative and it can be coded so that only one new function evaluation is required per iteration.

The formula for the secant method is the same one that was used in the Regula Falsi method, except that the logical decisions regarding how to define each succeeding term are different.

In the Secant method, the derivative can be approximated by a backward finite divided difference, as in the figure,

$$f'(x_k) \cong \frac{f(x_{k-1}) - f(x_k)}{x_{k-1} - x_k}$$



Using Newton-Raphson method,

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

Substituting $f'(x_k)$,

$$x_{k+1} = x_k - \frac{f(x_k)(x_{k-1} - x_k)}{f(x_{k-1}) - f(x_k)}$$

Notice that the approach requires initial estimates of x.

Algorithm - Secant Method

Input xk, xkMinus1, xTol, maxiters
iters = 1
yk =f(xk) (* invokes function f *)
ykMinus1 = f(xkMinus1)
root = (xkMinus1*yk - xk*ykMinus1)/(yk - ykMinus1)
ykPlus1 = f(root)
While((Abs(root - xk) > xTol) and (iters < maxiters))
xkMinus1 = xk
ykMinus1 = yk
xk = root
yk = ykPlus1
root = (xkMinus1*yk - xk*ykMinus1)/(yk - yk Minus1)
ykPlus1 = f(root)
iters = iters + 1
Endofwhile
Return root, ykPlus1, iters

Report:

Exercise 3.

Use the Newton Raphson method to estimate the root of $f(x) = e^{-x} - 1$, employing an initial guess of $x_0 = 0$. The tolerance is = 10^{-8}.

Exercise 4.

Find the root of the equation $f(x) = 3x + \sin(x) - e^x$, starting values are 0 and 1. The tolerance limit is 0.0000001.

Experiment 7: Interpolation

Introduction:

Forming a polynomial:

A polynomial, p(x) of degree n in MATLAB is stored as a row vector, **p**, of length n+1. The components represent the coefficients of the polynomial and are given in the descending order of the power of x, that is

p = [a_n a_{n-1} …….. a_1 a_0]

is interpreted as

$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots\ldots\ldots + a_1 x + a_0$

In MATLAB the following commands are used to evaluate a polynomial:
polyval, **poly**, **roots**, **conv** etc.

Interpolation:

In the mathematical subfield of numerical analysis, **interpolation** is a method of constructing new data points from a discrete set of known data points.

In engineering and science one often has a number of data points, as obtained by sampling or some experiment, and tries to construct a function which closely fits those data points. This is called curve fitting. Interpolation is a specific case of curve fitting, in which the function must go exactly through the data points.

Definition:

Given a sequence of *n distinct* numbers x_k called **nodes** and for each x_k a second number y_k, we are looking for a function *f* so that

$$f(x_k) = y_k \ , \ k = 1, \ldots, n$$

A pair x_k,y_k is called a **data point** and *f* is called the **interpolant** for the data points.

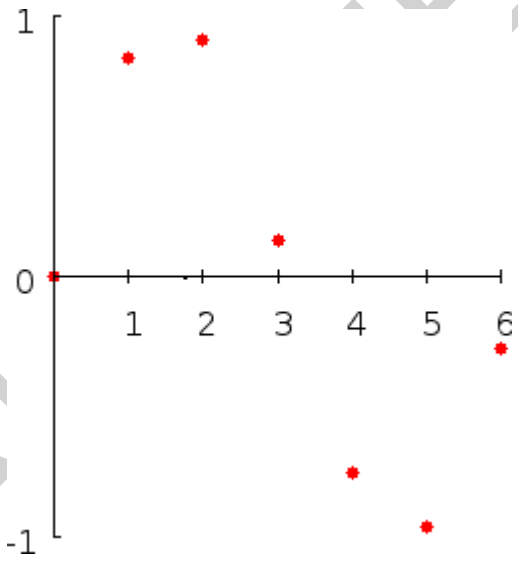
For example, suppose we have a table like this, which gives some values of an unknown function *f*. The data are given in the table:

Table 1

x	*f(x)*
0	0
1	0.8415
2	0.9093
3	0.1411
4	-0.7568
5	-0.9589
6	-0.2794

The plot can be shown as:

.

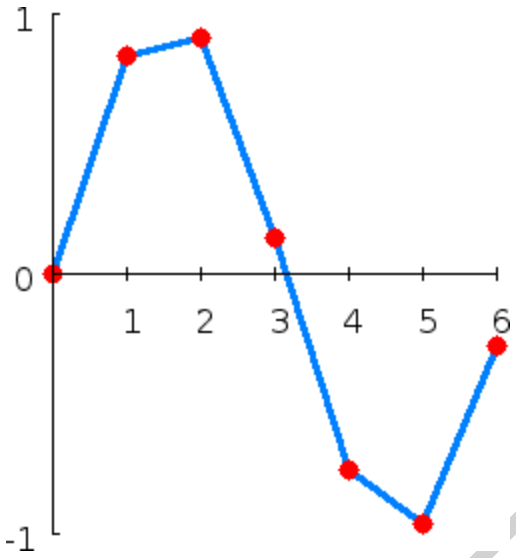


What value does the function have at, say, $x = 2.5$? Interpolation answers questions like this.

Types of interpolation:

A. Linear interpolation

One of the simplest methods is linear interpolation. Consider the above example of determining $f(2.5)$. We join the data points by linear interpolation and get the following plot:



Noe we can get *f*(2.5). Since 2.5 is midway between 2 and 3, it is reasonable to take *f*(2.5) midway between *f*(2) = 0.9093 and *f*(3) = 0.1411, which yields 0.5252.

Generally, linear interpolation takes two data points, say (x_a,y_a) and (x_b,y_b), and the interpolant is given by

$$f(x) = \frac{x - x_b}{x_a - x_b} y_a - \frac{x - x_a}{x_a - x_b} y_b$$

This formula can be interpreted as a weighted mean.

Linear interpolation is quick and easy, but it is not very precise.

B. Polynomial interpolation

Polynomial interpolation is a generalization of linear interpolation. Note that the linear interpolant is a linear function. We now replace this interpolant by a polynomial of higher degree.

Consider again the problem given above. The following sixth degree polynomial goes through all the seven points:

$$f(x) = -0.0001521x^6 - 0.003130x^5 + 0.07321x^4 - 0.3577x^3 + 0.2255x^2 + 0.9038x$$

Substituting $x = 2.5$, we find that $f(2.5) = 0.5965$.

Generally, if we have n data points, there is exactly one polynomial of degree $n-1$ going through all the data points. The interpolation error is proportional to the distance between the data points to the power n.

However, polynomial interpolation also has some disadvantages. Calculating the interpolating polynomial is relatively very computationally expensive Furthermore, polynomial interpolation may not be so exact after all, especially at the end points.

a. Lagrange Polynomial:

The Lagrange interpolating polynomial is the polynomial $P(x)$ of degree $(n-1)$ that passes through the n points $(x_1, y_1 = f(x_1))$, $(x_2, y_2 = f(x_2))$, ..., $(x_n, y_n = f(x_n))$, and is given by

$$P(x) = \sum_{j=1}^{n} P_j(x),$$

where

$$P_j(x) = y_j \prod_{\substack{k=1 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}.$$

Written explicitly,

$$P(x) = \frac{(x-x_2)(x-x_3)\cdots(x-x_n)}{(x_1-x_2)(x_1-x_3)\cdots(x_1-x_n)} y_1 + \frac{(x-x_1)(x-x_3)\cdots(x-x_n)}{(x_2-x_1)(x_2-x_3)\cdots(x_2-x_n)} y_2 + \cdots + \frac{(x-x_1)(x-x_2)\cdots(x-x_{n-1})}{(x_n-x_1)(x_n-x_2)\cdots(x_n-x_{n-1})} y_n.$$

When constructing interpolating polynomials, there is a tradeoff between having a better fit and having a smooth well-behaved fitting function. The more data points that are used in the interpolation, the higher the degree of the resulting polynomial, and therefore the greater oscillation it will exhibit between the data points. Therefore, a high-degree interpolation may be a poor predictor of the function between points, although the accuracy at the data points will be "perfect."

For $n = 3$ points,

$$P(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} y_1 + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} y_2 + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} y_3$$

Note that the function $P(x)$ passes through the points (x_i, y_i), as can be seen for the case $n = 3$,

$$P(x_1) = \frac{(x_1-x_2)(x_1-x_3)}{(x_1-x_2)(x_1-x_3)} y_1 + \frac{(x_1-x_1)(x_1-x_3)}{(x_2-x_1)(x_2-x_3)} y_2 + \frac{(x_1-x_1)(x_1-x_2)}{(x_3-x_1)(x_3-x_2)} y_3 = y_1$$

$$P(x_2) = \frac{(x_2-x_2)(x_2-x_3)}{(x_1-x_2)(x_1-x_3)} y_1 + \frac{(x_2-x_1)(x_2-x_3)}{(x_2-x_1)(x_2-x_3)} y_2 + \frac{(x_2-x_1)(x_2-x_2)}{(x_3-x_1)(x_3-x_2)} y_3 = y_2$$

$$P(x_3) = \frac{(x_3-x_2)(x_3-x_3)}{(x_1-x_2)(x_1-x_3)} y_1 + \frac{(x_3-x_1)(x_3-x_3)}{(x_2-x_1)(x_2-x_3)} y_2 + \frac{(x_3-x_1)(x_3-x_2)}{(x_3-x_1)(x_3-x_2)} y_3 = y_3.$$

Algorithm for the Lagrange Polynomial: To construct the Lagrange polynomial

$$P(x) = \sum_{k=0}^{n} y_k L_{n,k}(x)$$

of degree n, based on the n+1 points (x_k, y_k) for $k = 0, 1, \cdots, n$. The Lagrange coefficient polynomials for degree n are:

$$L_{n,k}(x) = \frac{(x - x_0) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_0) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}$$

for $k = 0, 1, \cdots, n$.

So, for a given *x* and a set of *(N+1)* data pairs, (x_i, f_i), *i= 0, 1, ... N:*

Set SUM=0

DO FOR *i*=0 to *N*

Set *P*=1

DO FOR *j*=0 to *N*

IF *j~=i*

Set *P=P*(x-x(j))/(x(i)-x(j))*

End DO(*j*)

Report:

#Exercise 1:

Construct a polynomial such that C(x)= A(x)*B(x)
Where A(x)= $3x^2$+2x-4 and B(x)= $2x^3$-2
Also find the roots of A(x), B(x) and C(x).

#Exercise 2.

Plot the curve corresponding to table1 using linear interpolation.

#Exercise 3.

$y = \sin(x); x = 0:10; x(i) = 0:0.25:10;$ Construct the interpolant y and plot.

#Exercise 4.

Write a MATLAB program implementing Lagrange Polynomial.

#Exercise 5.

Construct a Lagrange interpolating polynomials for the data points given in table 1.

Experiment 8: Numerical Differentiation

Introduction:

We are familiar with the analytical method of finding the derivative of a function when the functional relation between the dependent variable y and the independent variable x is known. However, in practice, most often functions are defined only by tabulated data, or the values of y for specified values of x can be found experimentally. Also in some cases, it is not possible to find the derivative of a function by analytical method. In such cases, the analytical process of differentiation breaks down and some numerical process have to be invented. The process of calculating the derivatives of a function by means of a set of given values of that function is called numerical differentiation. This process consists in replacing a complicated or an unknown function by an interpolation polynomial and then differentiating this polynomial as many times as desired.

Forward Difference Formula:

All numerical differentiation are done by expansion of Taylor series

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)h^2}{2} + \frac{f'''(x)h^3}{6} + \text{……..(1)}$$

From (1)

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h) \text{……..(2)}$$

Where, $O(h)$ is the truncation error, which consists of terms containing h and higher order terms of h.

Central Difference Formula (of order O (h^2)):

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)h^2}{2} + \frac{f'''(c_1)h^3}{6} \text{.....} \text{….(3)}$$

$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)h^2}{2} - \frac{f'''(c_2)h^3}{6} \ldots\ldots\ldots (4)$$

Using (3) and (4)

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \ldots\ldots\ldots (5)$$

Where, $O(h^2)$ is the truncation error, which consists of terms containing h^2 and higher order terms of h.

Central Difference Formula (of order O (h^4)):

Using Taylor series expansion it can be shown that

$$f'(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} + O(h^4) \ldots\ldots\ldots (6)$$

Here the truncation error reduces to h^4

Richardson's Extrapolation:

We have seen that

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

Which can be written as

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} + Ch^2$$

$$\text{Or, } f'(x) \approx D_0(h) + Ch^2 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (7)$$

If step size is converted to 2h

$$f'(x) \approx D_0(2h) + 4Ch^2 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (8)$$

Using (7) and (8)

$$f'(x) \approx \frac{4D_0(h) - D_0(2h)}{3} = \frac{-f_2 + 8f_1 - 8f_{-1} + f_{-2}}{12h} \ldots\ldots\ldots\ldots\ldots\ldots (9)$$

Equation (9) is same as equation (6)

The method of obtaining a formula for $f'(x)$ of higher order from a formula of lower order is called extrapolation. The general formula for Richardson's extrapolation is

$$f'(x) = D_k(h) + O(h^{2k+2}) = \frac{4^k D_{k-1}(h) - D_{k-1}(2h)}{4^k - 1} + O(h^{2k+2}) \ldots\ldots(10)$$

Algorithm for Richardson Approximation:

%Input-f(x) is the input function
% -delta is the tolerance for error
% -toler is the tolerance for relative error
%Output-D is the matrix of approximate derivatives
% -err is the error bound
% -relerr is the relative error bound
% -n is the coordinate for best approximation

Define
err=1
relerr=1
h=1
j=1

Compute D(1,1)=(f(x+h)-f(x-h))/(2h)

While relerr > toler & err > delta &j <12

Compute
h=h/2;
D(j+1,1)=(f(x+h)-f(x-h))/(2h)

DO For k=1:j
Compute D(j+1,k+1)=D(j+1,k)+ (D(j+1,k)-D(j,k))/((4^k)-1)
END DO(k)

Compute
err=|D(j+1,j+1)-D(j,j)|
relerr==2err/(|D(j+1,j+1)|+|D(j,j)|+eps)
j=j+1
END While

Report:

#Exercise 1:

Given f(x) =e^x, find $f'(1)$ using h=10^{-1}, 10^{-2},,,, upto 10^{-10}. Find out the error in each case by comparing the calculated value with exact value (Use forward difference formula).

#Exercise 2:

Given f(x) =e^x, find $f'(1)$ using h=10^{-1}, 10^{-2},,,, up to 10^{-10}. Use equation (5). Find out the error in each case by comparing the calculated value with exact value.

#Exercise 3:

Given f(x) =sin (cos (1/x)) evaluate $f'(1/\sqrt{2})$. Start with h =1 and reduce h to 1/10 of previous step in each step. If D_{n+1} is the result in (n+1) th step and D_n is the result in nth step then continue iteration until $|D_{n+1}-D_n|>=|D_n-D_{n-1}|$ or $|D_n-D_{n-1}|$ <tolerance. Use equation (6) for finding D.

#Exercise 4:

Given f(x) =sin (x^3-7x^2+6x+8) evaluate $f'(\frac{1-\sqrt{5}}{2})$. Use Richardson's extrapolation.

Approximation should be accurate up to 13 decimal places.

Experiment 09: Numerical Integration

Introduction

There are two cases in which engineers and scientists may require the help of numerical integration technique. (1) Where experimental data is obtained whose integral may be required and (2) where a closed form formula for integrating a function using calculus is difficult or so complicated as to be almost useless. For example the integral

$$\Phi(t) = \int_0^x \frac{t^3}{e^t - 1} dt.$$

Since there is no analytic expression for $\Phi(x)$, numerical integration technique must be used to obtain approximate values of $\Phi(x)$.

Formulae for numerical integration called quadrature are based on fitting a polynomial through a specified set of points (experimental data or function values of the complicated function) and integrating (finding the area under the fitted polynomial) this approximating function. Any one of the interpolation polynomials studied earlier may be used.

Some of the Techniques for Numerical Integration

Trapezoidal Rule

Assume that the values of a function $f(x)$ are given at x_1, x_1+h, x_1+2h ……x_1+nh and it is required to find the integral of $f(x)$ between x_1 and x_1+nh. The simplest technique to use would be to fit **straight lines** through f(x_1), f(x_1+h) ……and to determine the **area** under this approximating function as shown in Fig 7.1.

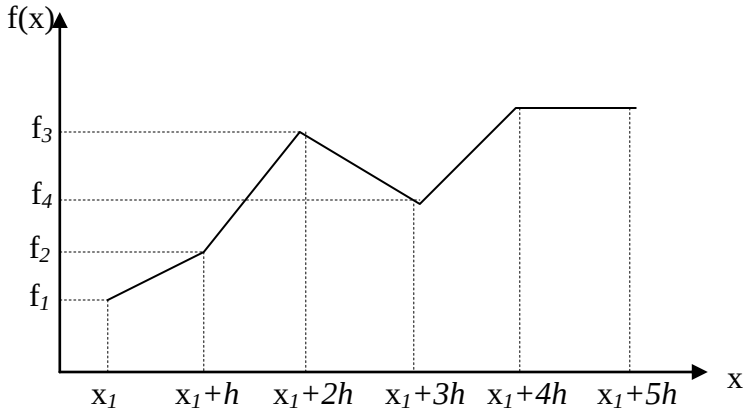


Fig. 7.1 Illustrating trapezoidal rule

For the first two points we can write:

$$\int_{x_1}^{x_1+h} f(x)dx = \frac{h}{2}(f_1 + f_2)$$

This is called first-degree Newton-Cotes formula.

From the above figure it is evident that the result of integration between x_1 and x_1+nh is nothing but the sum of areas of some trapezoids. In equation form this can be written as:

$$\int_{x_1}^{x_1+nh} f(x)dx = \sum_{i=1}^{n} \frac{(f_i + f_{i+1})}{2} h$$

The above integration formula is known as Composite Trapezoidal rule.

The composite trapezoidal rule can explicitly be written as:

$$\int_{x_1}^{x_1+nh} f(x)dx = \frac{h}{2}(f_1 + 2f_2 + 2f_3 + ...2f_n + f_{n+1})$$

Simpson's 1/3 Rule

This is based on approximating the function f(x) by fitting **quadratics** through sets of **three** points. For only three points it can be written as:

$$\int_{x_1}^{x_1+2h} f(x)dx = \frac{h}{3}(f_1 + 4f_2 + f_3)$$

This is called second-degree Newton-Cotes formula.

It is evident that the result of integration between x_1 and x_1+nh can be written as

$$\int_{x_1}^{x_1+nh} f(x)dx = \sum_{i=1,3,5,...,n-1} \frac{h}{3}(f_i + 4f_{i+1} + f_{i+2})$$

$$= \frac{h}{3}(f_1 + 4f_2 + 2f_3 + 4f_4 + 2f_5 + 4f_6 + \ ... \ 4f_n + f_{n+1})$$

In using the above formula it is implied that f is known at an **odd number of points** (**n+1 is odd,** where n is the no. of subintervals).

Simpson's 3/8 Rule

This is based on approximating the function f(x) by fitting **cubic** interpolating polynomial through sets of **four** points. For only four points it can be written as:

$$\int_{x_1}^{x_1+3h} f(x)dx = \frac{3h}{8}(f_1 + 3f_2 + 3f_3 + f_4)$$

This is called third-degree Newton-Cotes formula.

It is evident that the result of integration between x_1 and x_1+nh can be written as

$$\int_{x_1}^{x_1+nh} f(x)dx = \sum_{i=1,4,7,...,n-2} \frac{h}{3}(f_i + 3f_{i+1} + 3f_{i+2} + f_{i+3})$$

$$= \frac{3h}{8}(f_1 + 3f_2 + 3f_3 + 2f_4 + 3f_5 + 3f_6 + 2f_7 + ... \ + 2f_{n-2} + 3f_{n-1} + 3f_n + f_{n+1})$$

In using the above formula it is implied that f is known at (n+1) points where **n is divisible by 3**.

An algorithm for integrating a **tabulated function** using composite trapezoidal rule:

Remarks: f_1, f_2,……, f_{n+1} are the tabulated values at x_1, x_{1+h},……x_{1+nh} (n+1 points)

1 Read h

2 *for* $i = 1$ to $n+1$ Read f_i *endfor*

3 $sum \leftarrow (f_1 + f_{n+1})/2$

4 *for* $j = 2$ to n *do*

5 $sum \leftarrow sum + f_j$

endfor

6 $\text{int}egral \leftarrow h \,.\, sum$

7 *write* $\text{int}egral$

stop

An algorithm for integrating a known function using composite trapezoidal rule:

If f(x) is given as a closed form function such as $f(x) = e^{-x}\cos x$ and we are asked to integrate it from x_1 to x_2, we should decide first what h should be. Depending on the value of h we will have to evaluate the value of f(x) inside the program for $x=x_1+nh$ where n=0,1, 2,….n and $n = (x_2 - x_1)/h$.

1 $h = (x_2 - x_1)/n$

2 $x \leftarrow x_1$

3 $sum \leftarrow f(x)$

4 *for* $i = 2$ to n *do*

5 $x \leftarrow x + h$

6 $sum \leftarrow sum + 2f(x)$

endfor

7 $x \leftarrow x_2$

8 $sum \leftarrow sum + f(x)$

9 $\text{int}egral \leftarrow \frac{h}{2}\,.\, sum$

10 *write* $\text{int}egral$

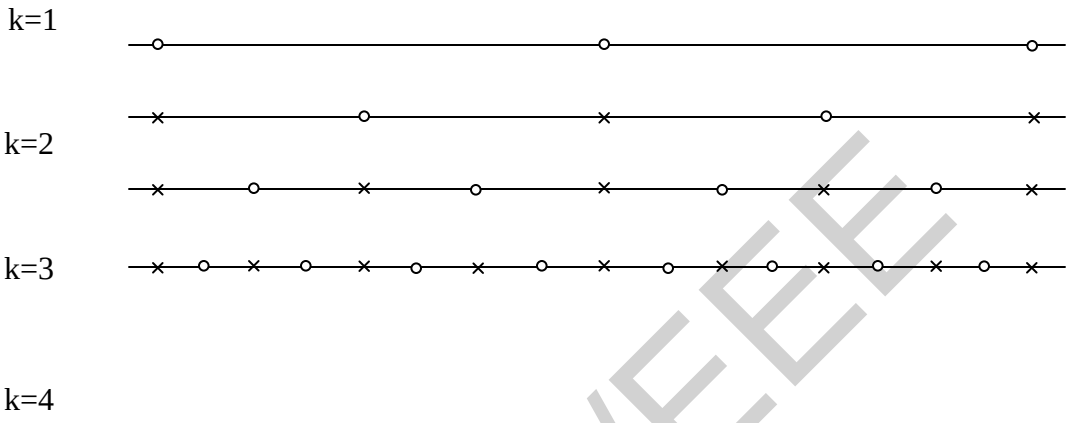
stop

Adaptive Integration

When f(x) is a known function we can choose the value for h arbitrarily. The problem is that we do not know a priori what value to choose for h to attain a desired accuracy (for example, for an arbitrary h sharp picks of the function might be missed). To overcome this problem, we can start with two subintervals, $h = h_1 = (x_2 - x_1)/2$ and apply either trapezoidal or Simpson's 1/3 rule. Then we let $h_2 = h_1/2$ and apply the formula again,

now with four subintervals and the results are compared. If the new value is sufficiently close, the process is terminated. If the 2nd result is not close enough to the first, h is halved again and the procedure is repeated. This is continued until the last result is close enough to its predecessor. This form of numerical integration is termed as adaptive integration.

The no. of computations can be reduced because when h is halved, all of the old points at which the function was evaluated appear in the new computation and thus repeating evaluation can be avoided. This is illustrated below.



An algorithm for adaptive integration of a known function using trapezoidal rule:

1 Read x_1, x_2, e Remark: The allowed error in integral is e

2 $h \leftarrow x_2 - x_1$

3 $S_1 \leftarrow (f(x_1) + f(x_2))/2$

4 $I_1 \leftarrow h \,.\, S_1$

5 $i \leftarrow 1$

Repeat

6 $x \leftarrow x_1 + h/2$

7 *for* $j = 1$ to i *do*

8 $S_1 \leftarrow S_1 + f(x)$

9 $x \leftarrow x + h$

endfor

10 $i \leftarrow 2i$

11 $h \leftarrow h/2$

12 $I_0 \leftarrow I_1$

13 $I_1 \leftarrow h \,.\, S_1$

14 *until* $|I_1 - I_0| \leq e \,.\, |I_1|$

15 *write* $I_1,\ h,\ i$

Stop

Report:

 # Exercise1.

Integrate the function tabulated in Table 7.1 over the interval from x=1.6 to x=3.8 using composite trapezoidal rule with (a) h=0.2, (b) h=0.4 and (c) h=0.6

Table 7.1

X	f(x)	X	f(x)
1.6	4.953	2.8	16.445
1.8	6.050	3.0	20.086
2.0	7.389	3.2	24.533
2.2	9.025	3.4	29.964
2.4	11.023	3.6	36.598
2.6	13.468	3.8	44.701

The data in Table 7.1 are for $f(x) = e^x$. Find the true value of the integral and compare this with those found in (a), (b) and (c).

 # Exercise 2.

(a) Integrate the function tabulated in Table 7.1 over the interval from x=1.6 to x=3.6 using Simpson's composite 1/3 rule.

(b) Integrate the function tabulated in Table 7.1 over the interval from x=1.6 to x=3.4 using Simpson's composite 3/8 rule.

 # Exercise 3.

(a) Find (approximately) each integral given below using the composite trapezoidal rule with $n = 12$.

(i) $\int_{-1}^{1} (1+x^2)^{-1} dx$ (ii) $\int_{0}^{4} x^2 e^{-x} dx$

(b) Find (approximately) each integral given above using the Simpson's composite 1/3 and 3/8 rules with $n = 12$.

 # Exercise 4.

Evaluate the integral of xe^{-2x^2} between x=0 and x=2 using a tolerance value sufficiently small as to get an answer within 0.1% of the true answer, 0.249916 (Use adaptive integration for both Ex 4 & 5).

 # Exercise 5.

Evaluate the integral of $\sin^2(16x)$ between $x = 0$ and $x = \pi/2$. Why the result is erroneous? How can this be solved? (The correct result is $\pi/4$)

Experiment 10: Solutions to Linear Differential Equations

In mathematics, a **differential equation** is an equation in which the derivatives of a function appear as variables. Differential equations have many applications in physics and chemistry, and are widespread in mathematical models explaining biological, social, and economic phenomena.

Differential equations are divided into two types:

a. An **Ordinary Differential Equation (ODE)** only contains function of one variable, and derivatives in that variable.
b. A **Partial differential Equation (PDE)** contains multivariate functions and their partial derivatives.

The **order** of a Differential equation is that of the highest derivative that it contains. For instance, a **first-order Differential equation** contains only first derivatives.

A **linear differential equation** of order n is a differential equation written in the following form:

$$a_n(x)\frac{d^n y}{dx^n} + a_{n-1}(x)\frac{d^{n-1} y}{dx^{n-1}} + \ldots\ldots\ldots + a_1(x)\frac{dy}{dx} + a_0(x)y = f(x)$$

where $a_n(x) \neq 0$.

Initial value problem:

A problem in which we are looking for the unknown function of a differential equation where the values of the unknown function and its derivatives at some point are known is called an **initial value problem** (in short IVP).

If no initial conditions are given, we call the description of all solutions to the differential equation the **general solution**.

Methods of solving the Ordinary Differential Equations:

1. **Euler's Method:**

Let $y'(x) = f(x, y(x))$

$y(x_0) = y_0$

Here $f(x, y)$ is a given function, y_0 is a given initial value for y at $x = x_0$.
The unknown in the problem is the function $y(x)$.

Our goal is to determine (approximately) the unknown function $y(x)$ for $x > x_0$. We are told explicitly the value of $y(x_0)$, namely y_0. Using the given differential equation, we can also determine exactly the instantaneous rate of change of y at x_0. The basic idea is simple. This differential equation tells us how rapidly the variable y is changing and the initial condition tells us where y starts
$y'(x_0) = f(x_{0,} y(x_0)) = f(x_0, y_0).$

If the rate of change of $y(x)$ were to remain $f(x_0, y_0)$ for all time, then $y(x)$ would be exactly $y_0 + f(x_0, y_0)(x - x_0)$. The rate of change of $y(x)$ does not remain $f(x_0, y_0)$ for all time, but it is reasonable to expect that it remains close to $f(x_0, y_0)$ for x close to x_0. If this is the case, then the value of $y(x)$ will remain close to $y_0 + f(x_0, y_0)(x - x_0)$ for x close to x_0. So pick a small number h and define
$x_1 = x_0 + h$
$y_1 = y_0 + f(x_0, y_0)(x_1 - x_0) = y_0 + f(x_0, y_0)h$
By the above argument
$y(x_1) \approx y_1$

Now we start over. We now know the approximate value of y at x_1. If $y(x_1)$ were exactly y_1, then the instantaneous rate of change of y at x_1 would be exactly $f(x_1, y_1)$. If this rate of change were to persist for all future value of x, $y(x)$ would be exactly $y_1 + f(x_1, y_1)(x - x_1)$.

As $y(x_1)$ is only approximately y_1 and as the rate of change of $y(x)$ varies with x, the rate of change of $y(x)$ is only approximately $f(x_1, y_1)$ and only for x near x_1. So we approximate $y(x)$ by $y_1 + f(x_1, y_1)(x - x_1)$ for x bigger than, but close to, x_1. Defining
$x_2 = x_1 + h = t_0 + 2h$
$y_2 = y_1 + f(x_1, y_1)(x_2 - x_1) = y_1 + f(x_1, y_1)h$

We have $y(x_2) \approx y_2$

We just repeat this argument. Define, for $n = 0,1,2,3\ldots\ldots$

$x_n = x_0 + nh$

Suppose that, for some value of n, we have already computed an approximate value y_n for $y(x_n)$. Then the rate of change of $y(x)$ for x close to x_n is

$f(x, y(x)) \approx f(x_n, y(x_n)) \approx y(x_n, y_n)$

and, again for x near x_n,

$y(x) = y_n + f(x_n, y_n)(x - x_n)$.

Hence

$y(x_{n+1}) \approx y_{n+1} = y_n + f(x_n, y_n)h$

This algorithm is called **Euler's Method**. The parameter h is called the **step size**.

2. The Improved Euler's Method

Euler's method is one algorithm that generates approximate solutions to the initial value problem

$$y'(x) = f(x, y(x))$$

$$y(x_0) = y_0$$

In applications, $f(x, y)$ is a given function and x_0 and y_0 are given numbers. The function $y(x)$ is unknown. Denote by $\varphi(x)$ the exact solution for this initial value problem. In other words $\varphi(x)$ is the function that obeys the following relation exactly.

$\varphi'(x) = f(x, \varphi(x))$

$\varphi(x_0) = y_0$

Fix a step size h and define $x_n = x_0 + nh$. We now derive another algorithm that generates approximate values for φ at the sequence of equally spaced values $x_0, x_1, x_2, \ldots\ldots$ We shall denote the approximate values y_n with $y_n = \varphi(x_n)$

By the fundamental theorem of calculus and the differential equation, the exact solution obeys

$$\varphi(x_{n+1}) = \varphi(x_n) + \int_{x_n}^{x_{n+1}} \varphi'(x)dx = \varphi(x_n) + \int_{x_n}^{x_{n+1}} f(x, \varphi(x))dx$$

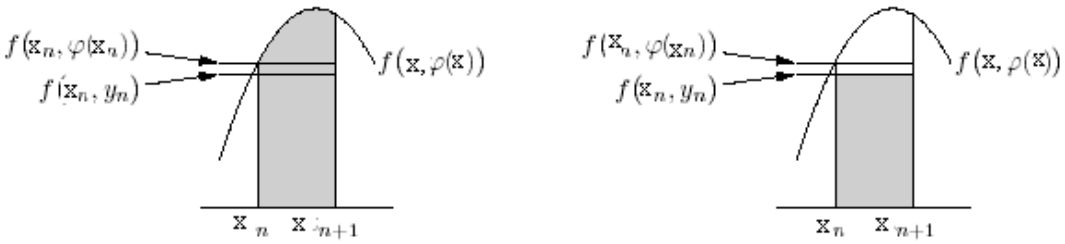
Fix any n and suppose that we have already found $y_0, y_1, y_2, \ldots\ldots, y_n$. Our algorithm for computing y_{n+1} will be of the form

$$y_{n+1} = y_n + \text{approximate value for } \int_{x_n}^{x_{n+1}} f(x, \varphi(x))dx$$

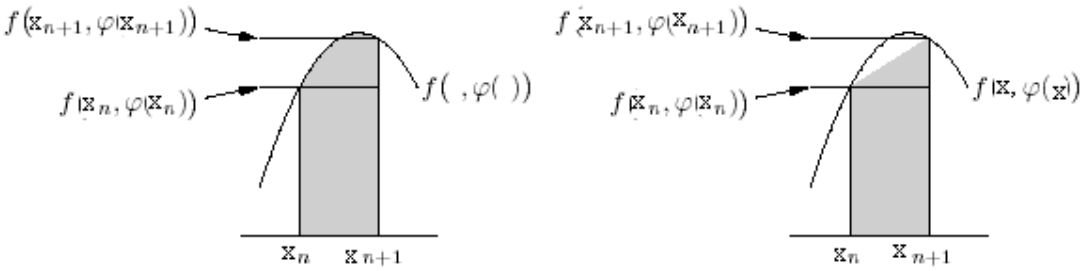
In fact Euler's method is of precisely this form. In Euler's method, we approximate $f(x,\varphi(x))$ for $x_n \le x \le x_{n+1}$ by the constant $f(x_n, y_n)$. Thus Euler's approximate value for

$$\int_{x_n}^{x_{n+1}} f(x,\varphi(x))dx = \int_{x_n}^{x_{n+1}} f(x_n, y_n)dx = f(x_n, y_n)h$$

The area of the complicated region $0 \le y \le f(x,\varphi(x))$; $x_n \le x \le x_{n+1}$ (represented by the shaded region under the parabola in the left half of the figure below) is approximated by the area of the rectangle $0 \le y \le f(x_n, y_n)$; $x_n \le x \le x_{n+1}$ (the shaded rectangle in the right half of the figure below).



Our second algorithm, the improved Euler's method, gets a better approximation by attempting to approximate by the trapezoid on the right below rather than the rectangle on the right above. The exact area of this trapezoid is the length h of the base multiplied by



the average, $\frac{1}{2}\left[f(x_n,\varphi(x_n)) + f(x_{n+1},\varphi(x_{n+1})) \right]$, of the heights of the two sides. Of course we do not know $\varphi(x_n)$ or $\varphi(x_{n+1})$ exactly. Recall that we have already found $y_0, y_1, y_2, \ldots\ldots, y_n$ and are in the process of finding y_{n+1}. So we already have an approximation for $\varphi(x_n)$, namely y_n, but not for $\varphi(x_{n+1})$. Improved Euler uses

$$\varphi(x_{n+1}) \approx \varphi(x_n) + \varphi'(x_n)h \approx y_n + f(x_n, y_n)h$$

in approximating $\frac{1}{2}\left[f(x_n, \varphi(x_n)) + f(x_{n+1}, \varphi(x_{n+1})) \right]$. Altogether Improved Euler's approximate value for $\int_{x_n}^{x_{n+1}} f(x, \varphi(x))dx = \frac{1}{2}\left[f(x_n, y_n) + f(x_{n+1}, y_n + f(x_n, y_n)h) \right]h$

so that the improved Euler's method algorithm is

$$y(x_{n+1}) \approx y_{n+1} = y_n + \frac{1}{2}\left[f(x_n, y_n) + f(x_{n+1}, y_n + f(x_n, y_n)h) \right]h$$

The general step is

$p_{n+1} = y_n + hf(x_n, y_n)$, $\quad x_{n+1} = x_n + h$,

$$y_{n+1} = y_n + \frac{h}{2}(f(x_n, y_n) + f(x_{n+1}, p_{n+1}))$$

Report:

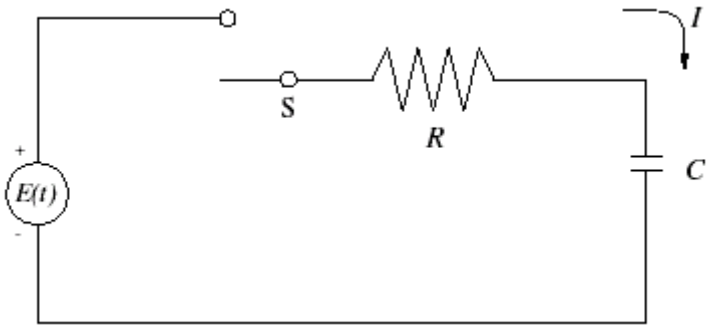
#Exercise 1.

Use Euler's method to solve the IVP $y' = \frac{x-y}{2}$ on $[0,3]$ with $y(0) = 1$. Compare solutions for $h = 1, \frac{1}{2}, \frac{1}{4}$ and $\frac{1}{8}$.

The exact solution is $y = 3e^{-x/2} - 2 + x$.

#Exercise 2.

Consider the following circuit:



In this circuit, $R = 20K\Omega$, $C = 10\mu F$, $E = 117V$ and Q(0)=0. Find Q for $t = 0$ to $t = 3$sec.

#Exercise 3.

Solve exercise 1 and 2 using Improved Euler's method.