Delhi Technological University

Department Of Electrical Engineering

NEOM PRACTICAL FILE

SUBMITTED TO :

Dr. Anup Kumar Mandpura
Assistant Professor
Department of Electrical Engineering

SUBMITTED BY :

Prashant
23/EE/181
EE - 3

Experiment : 1

Objective : To find out the root of the given by using the bisection method

Bisection Method Definition

The bisection method is used to find the roots of polynomial equation. It separates the interval and subdivides the interval in which the root of the equation lies. The principle behind this method is the intermediate theorem for continuous functions. It works by narrowing the gap between the positive and negative intervals until it closes in on the correct answer. This method narrows the gap by taking the average of the positive and negative intervals. It is a simple method.

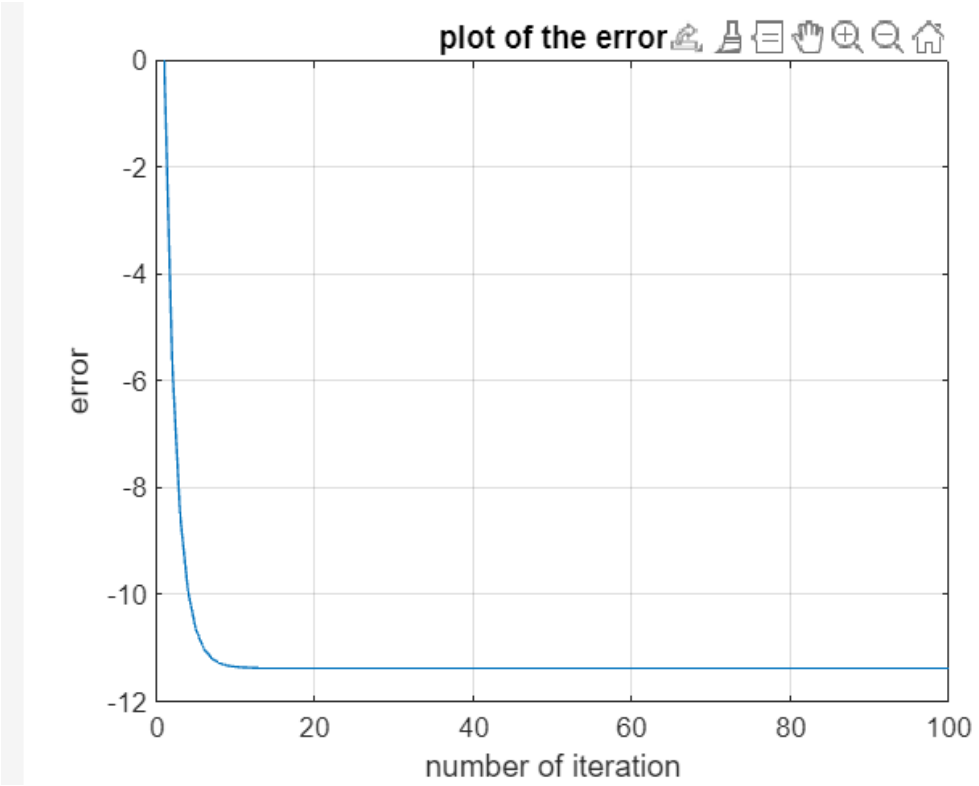
Software required : MATLAB

Program code :

```
f=@(x)x^3-6*x^2-11*x-6;
a=2.5;
b=1.5;
for i=1:100
    c=(a+b)/2;
    if f(c)>0
        b=c;
    else
        a=c;
    end
end

a=1;  b=2;  p=c;
for i=1:100
    c=(a+b)/2;
    error(i)=f(c)-f(p);
    if f(c)>0
        b=c;
    else
        a=c;
    end
end
fprintf('root of the given equation is %f',p);
plot(error);
title('plot of the error')
xlabel('number of iteration')
ylabel('error')
grid on
```

Program output :



Command Window

```
>> bisection2
root of the given equation is 2.000000
```

Experiment : 2

Objective : To find out the root of the given by using the Regula-Fassi method.

Regula Falsi Method

Regula Falsi Method, also known as the False Position Method, is a numerical technique used to find the roots of a non-linear equation of the form f(x)=. This method is based on the concept of bracketing, where two initial guesses, x_0 and x_1, are chosen such that the function values at these points have opposite signs, indicating that a root lies between them.

c = a − f (a). (b-a) / [f (b) − f (a)] .

Software required : MATLAB

Program code :

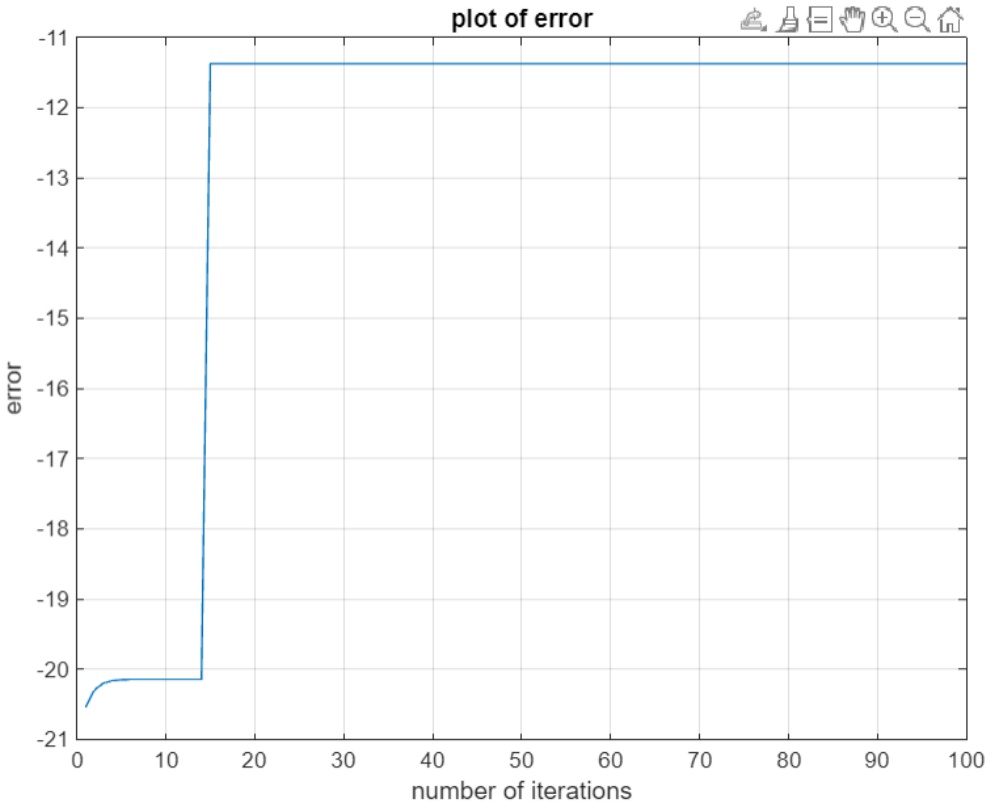
```
% REGULA-FASSI METHOD
f=@(x)x^3-2*x-5;
a=2;
b=3;
for i=1:14
    c = (a*f(b)-b*f(a))/f(b)-f(a);
    if f(c)>0
        b=c;
    else
        a=c;
    end

end
a=2;
b=3;
p=c;
for i=1:14
    c=(a*f(b)-b*f(a))/(f(b)-f(a));
    error(i)=f(c)-f(p);
    if f(c)>0
        b=c;
    else
        a=c;
    end

end
fprintf('root of the given equation is %f',c)
plot(error)
title('plot of error')
```

```
xlabel('number of iterations')
ylabel('error')
grid on;
```

Program output :



Command Window

```
>> regulafassi
root of the given equation is 2.094551
>>
```

Experiment : 3

Objective : To find out the root of the given by using the Newton – Raphson method.

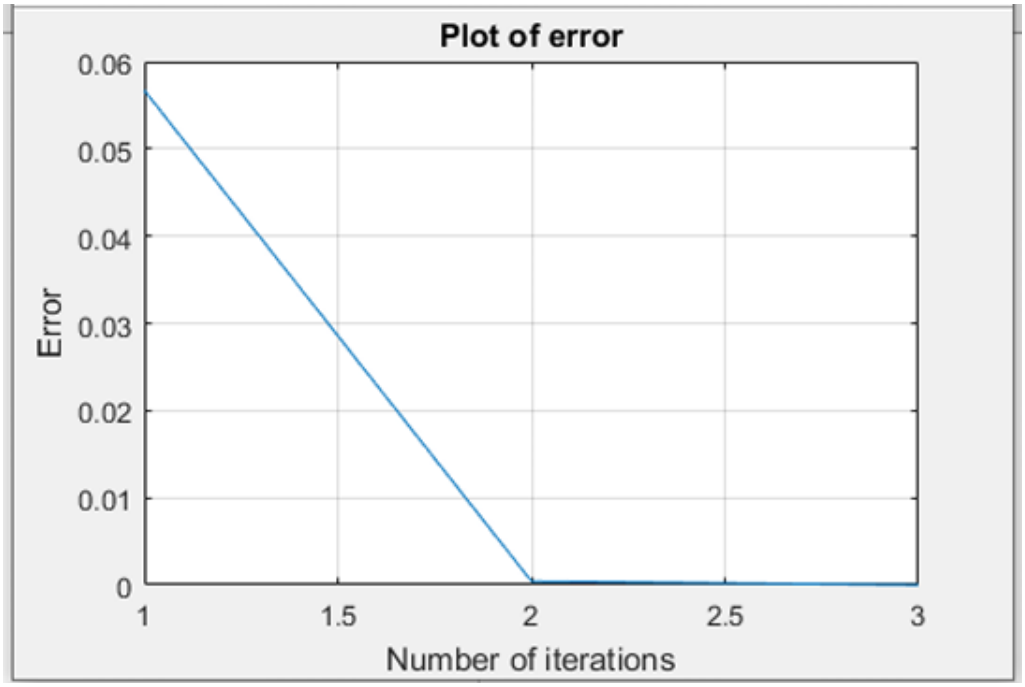
The **Newton Raphson Method** is referred to as one of the most commonly used techniques for finding the roots of given equations. It can be efficiently generalised to find solutions to a system of equations. Moreover, we can show that when we approach the root, the method is quadratically convergent. In this article, you will learn how to use the Newton Raphson method to find the roots or solutions of a given equation, and the geometric interpretation of this method.
$x_1 = x_0 - f(x_0)/f'(x_0)$.

Software required : MATLAB

Program code :

```
syms x
f = @(x) x-2+log(x);
x0=1.5;
F = diff(f,x);
epsilon=0.0001;
for i =1:100
        f0=vpa(subs(f,x,x0));
        f0_der=vpa(subs(F,x,x0));
        y=x0-f0/f0_der;
        err(i)=abs(y-x0);
    if err(i)<epsilon
    break
    end
        x0=y;
end
fprintf('The Root is : %f \n',y);
fprintf('No. of Iterations : %d\n',i);
plot(err)
title('Plot of error')
xlabel('Number of iterations')
ylabel('Error')
grid on;
```

Program output :



Command Window

```
>> NEWTON_RAPSON
The Root is : 1.557146
No. of Iterations : 3
>>
```

Experiment : 4

Objective : To find out the root of the given by using the Rung-Kutta method.

Theory-

The Order 2 Runge-Kutta method, often referred to as the **midpoint method**, is a numerical technique used to approximate the solutions of ordinary differential equations (ODEs). It is part of the broader family of Runge-Kutta methods, which provide iterative procedures to approximate solutions of initial value problems. The Order 2 method is considered a second-order method because the local error per step is proportional to the square of the step size, h2, while the global error is proportional to h. This means the method is more accurate than basic techniques like Euler's method, which is only first-order. The central idea behind the Order 2 Runge-Kutta method is to take an initial estimate of the slope at the beginning of the interval, use this to predict the slope at the midpoint of the interval, and then use this midpoint slope to compute the next point in the solution. The general formula involves two stages. First, an intermediate value, k1, is computed using the slope at the current point. Then, a second slope estimate, k2, is calculated using the slope at the midpoint of the interval, adjusted by the first slope. The solution is then updated by advancing from the current point to the next using this midpoint slope. Mathematically, this can be expressed as:

k1=f(tn,yn)

k2=f(tn+h/2,yn+h/2k1)

Software required : MATLAB

Program code :

Problem- Find approximate value of y when x-0.02.Given- =x2+y and y(0)=1

```
f=@(x,y) x^2+y;
x=x0;
y=y0;

h=0.01;
x1=0.02;
y0=1;
x0=0;
k1=0;
k2=0;

while x<x1
    k1=h*f(x,y);
    k2=h*f(x+h/2,y+k1/2);
    y=y+(k1+k2)/2;
    x=x+h;
end

fprintf('Approximate value of y when x-0.02 is %f',y);
```

Program output :

Command Window

```
>> Rk_order2
Approximate value of y when x-0.02 is 1.020152
>>
```

Experiment : 5

Objective : To find out the root of the given by using the Rung-Kutta method of 4th order.

Theory-

The most commonly used Runge Kutta method to find the solution of differential equations is the RK4 method, i.e., the fourth-order Runge-Kutta method. The Runge-Kutta method provides the approximate value of y for a given point x. Only the first order ODEs can be solved using the Runge Kutta RK4 method.

Runge-Kutta Fourth Order Method Formula

The formula for the fourth-order Runge-Kutta method is given by:

$y_1 = y_0 + (⅙) (k_1 + 2k_2 + 2k_3 + k_4)$

Here,

$k_1 = hf(x_0, y_0)$

$k_2 = hf[x_0 + (½)h, y_0 + (½)k_1]$

$k_3 = hf[x_0 + (½)h, y_0 + (½)k_2]$

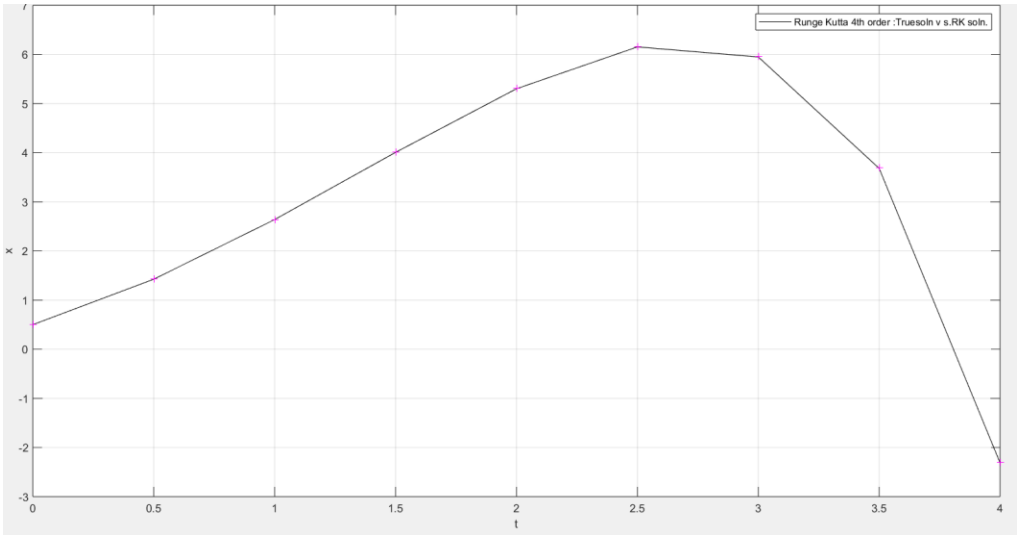
$k_4 = hf(x_0 + h, y_0 + k_3)$

Software required : MATLAB

Program code :

```
f=@(t,y)y-t^2+1;
h=0.5;
t(1)=0;
x(1)=0.5;
y_true=zeros(1,8);
for i=1:8
    k1=h*f(t(i),x(i));
    k2=h*f(t(i)+h/2,x(i)+k1/2);
    k3=h*f(t(i)+h/2,x(i)+k2/2);
    k4=h*f(t(i)+h,x(i)+k3);
    x(i+1)=x(i)+(k1+2*k2+2*k3+k4)/6;
    t(i+1)=t(i)+h;
    y_true=t.^2+2*t+1.0-0.5*exp(t);
end
plot(t,x,'k-',t,y_true,'m+')
xlabel('t')
ylabel('x')
legend('Runge Kutta 4th order :Truesoln v s.RK soln.')
```

Program output :



Name	Value
f	@(t,y)y-t^2+1
h	0.5000
i	8
k1	-3.7850
k2	-5.6375
k3	-6.1006
k4	-8.7103
t	[0,0.5000,1,1.500...
x	[0.5000,1.4251,2....
y_true	[0.5000,1.4256,2....

Experiment : 6

Objective : To find out the root of a given polynomial by using fixed point iteration method.

Theory-

Fixed Point Iteration Method

Suppose we have an equation $f(x) = 0$, for which we have to find the solution. The equation can be expressed as $x = g(x)$. Choose $g(x)$ such that $|g'(x)| < 1$ at $x = x_o$ where x_o is some initial guess called fixed point iterative scheme. Then the iterative method is applied by successive approximations given by $x_n = g(x_{n-1})$, that is, $x_1 = g(x_o)$, $x_2 = g(x_1)$ and so on.

Important Facts

Some interesting facts about the fixed point iteration method are

•The form of $x = g(x)$ can be chosen in many ways. But we choose $g(x)$ for which $|g'(x)|<1$ at $x = x_o$.

•By the fixed-point iteration method, we get a sequence of x_n, which converges to the root of the given equation.

•Lower the value of $g'(x)$, fewer the iterations are required to get the approximate solution.

•The rate of convergence is more if the value of $g'(x)$ is smaller.

•The method is useful for finding the real roots of the equation, which is the form of an infinite series.

•The type of convergence seen is linear.

Software required : MATLAB

Program code :

```
g=@(x)((2*x+5)/2)^(1/2);
err=1;
x=2;
y=2;
while err>0.001
    x=g(x);
    err=abs(x-y);
    y=x;
end
x
y
```

Program output :

```
x =

    2.1582


y =

    2.1582
```