Page No.
Date 15 01 20

Experiment No. 1

AIM

Solution of algebraic & transcendental equation using bisection method.

THEORY

An equation of the Form $F(x)=0$ where Function is purely a polynomial is known as Algebraic Equation

eg. $x^3+2x+9=0$

$x^3-x-1=0$

$x^5-x^4=x^2-5x$

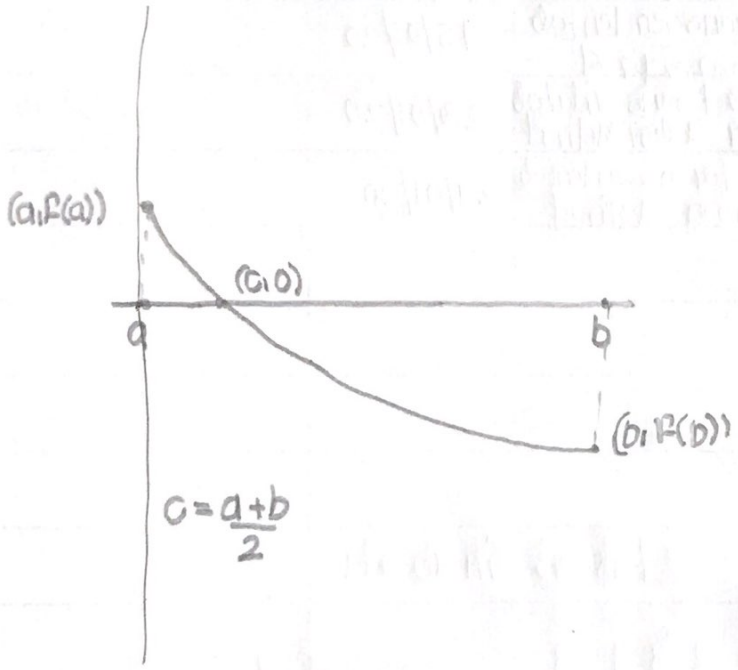
An equation of the Form $F(x)=0$ where Function consists of sinusoidal, logarithmic or exponential terms is known as Transcendental Equation

Bisection Method For Finding root

It is used to find an approximate root of a particular Function $F(x)=0$

"If a Function $F(x)$ is continous on an interval $[a,b]$ & $F(a)F(b)<0$, then a value $c \in [a,b]$ exist For which $F(c)=0$"

Let us consider a & b such that $F(a)>0$ & $F(b)<0$. Then next approx root is given by $(a+b)/2 = c$. If $F(c)>0$ then $a=c$ otherwise $b=c$ ie. $F(c)<0$.


(a, f(a))
(c, 0)
a
b
(b, f(b))
c = a+b / 2

Page No. _____
Date

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
#include <math.h>

Float function (Float);
void main()
{
  clrscr();
  Float a,b,c,Fa,Fb,Fc;
  a=0;
  b=1;

  for(int i=0; i<8 ;i++)
  {
   Fa = function(a);
   Fb= function(b);

  iF(Fa*Fb<0)
   {
    c=(a+b)/2;
    Fc= function(c);
    iF(Fc<0)
    {
      b=c;
  }
```

```
else if (fc>0)
{
  a=c;
}

printf("After Iteration %d approximate value of x0 is %f \n",
        i+1,c);

   }
}
getch();
}

Float Function(Float x0)
{
  return pow(x0,3) -5*x0+1;
}
```

RESULT

We found the solution of given equation using bisection method.

Page No.
Date 22 01 20

Experiment No. 2

AIM

Solution of Algebraic & transcendental equation using Regula Falsi Method.

THEORY

Regula Falsi Method or method of False position is a numerical method for solving an equation in one unknown. It is based on the theorem -

"If a function $f(x)$ is continous in $[a,b]$ & $f(a)\,f(b)<0$ (ie. $f(x)$ has opposite signs at a&b) then a value $c \in (a,b)$ exists such that $f(c)=0$.

For a given cts function we assume that $f(a)$ & $f(b)$ have opposite signs. We find the pt $(c,0)$ where secant (joining points $(a,f(a))$ & $(b,f(b))$ crosses x axis

Using points $(a,f(a))$ & $(b,f(b))$

$$m = \frac{f(b)-f(a)}{b-a}$$

Using points $(c,0)$ & $(b,f(b))$

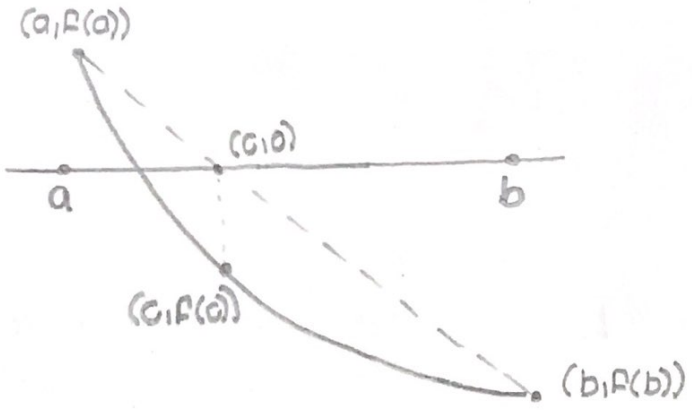
$$m = \frac{0-f(b)}{c-b}$$

Equating both & solving we get

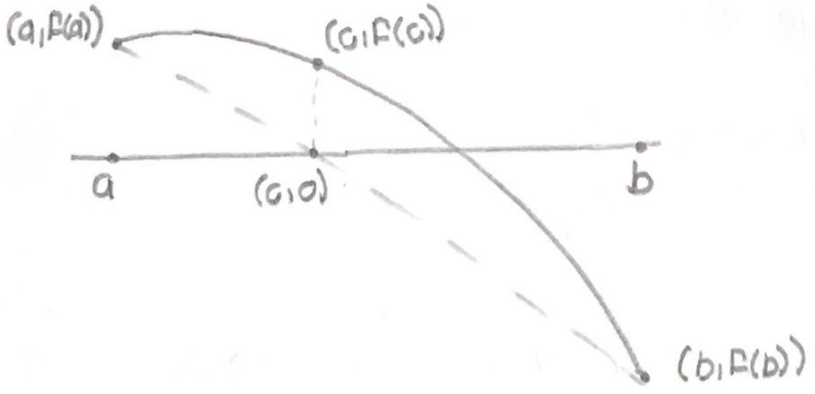
$$c = b - \frac{f(b)\,[b-a]}{f(b)-f(a)}$$

$$= \frac{af(b) - bf(a)}{f(b)-f(a)}$$

- If $f(a)\,f(c)<0$, zero lies in $[a,c]$
- If $f(c)\,f(b)<0$ zero lies in $[c,b]$
- If $f(c)=0$, then the zero is c



IF $f(a)\,f(c) < 0$, then squeeze from the right



IF $f(b)\,f(c) < 0$ then squeeze from left.

Page No. _____
Date

PROGRAM CODE

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

double function(double);
void main()
{
  clrscr();
  int i=0;
  double a,b,c,fa,fb,fc;
  a=-5.000;
  b=+5.000;

do
{
  i=i+1;
  fa=function(a);
  fb=function(b);

if(fa*fb<0)
{
  c=(a*fb-b*fa)/(fb-fa);
  fc=function(c);

  if(fc<0)
```

```
        a=c;
    }

    else
    { b=c;
    }
}

} while((fabs(Function(b)))>0.001));
    printf("After %d iterations approximate value of
            x is %lf with an error of %lf \n",i,c,fc);
    getch();
}

double function(double x)
{
    return pow(x,3) + pow(x,2) + x+7;
}
```

After 28 iterations value of x is -2.104780 with an error of 0.000936.

Page No.
Date 29 01 20

Experiment No. 3

AIM

Solution of Algebraic & transcendental Equation using Newton Raphson Method.

THEORY

Newton Raphson Method, also called Newton's Method is fastest & simplest approach of all methods to find the realroot of nonlinear function.
It is an open bracket approach, requiring only one initial guess. This method is quite often used to improve the results obtained from other iterative approaches.

The Iterative Formula for Newton Raphson Method is.

$$x_{n+1} = x_n - \frac{F(x_n)}{F'(x_n)}$$

Some of its limitations are :

1) Finding $F'(x)$ can be difficult in cases where $F(x)$ is complicated.
2) When $F'(x_n)$ tends to zero, Newton Raphson gives no solution.
3) Infinite oscillation resulting in slow convergence near local maxima or minima.

Page No.
Date

Program Code:

```
#include<stdio.h>
#include<conio.h>
#include <math.h>

double Function (double);
double derivative (double);

double Function(double x)
 {
   return pow(x,3) + pow(x,2) + x+7;
 }

double derivative (double x)
 {
   return  3*pow(x,2) + 2*x +1;
 }

void main()
{
   clrscr();
   double x=-2.00,Fx,dx,temp,tempx;
   int i=0;

 do
 { i=i+1;
   Fx= Function(x);
```

```
dx = derivative(x);
temp = x - (Fx/dx);
tempx = x;
x = temp;
printf("After iteration %d value of x is %.1f with
        Fx %.1f \n", i, x, Fx);

} while(Fabs((x-tempx)/x) > 0.005);

getch();
}
```

Output:

After iteration 1 value of x is -2.11111 with Fx 1.00000
After iteration 2 value of x is -2.10489 with Fx -0.063100

Topic _____ Date _____

Experiment No. 4

Aim:

To perform Algebra of Matrices : Addition, Multiplication & Transpose.

Software Used:

Turbo C

Theory:

1. **Addition:** Two matrices can be added together only & only if they are of same order.

Let A & B be two matrices of order $m \times n$

$\therefore \quad C_{m \times n} = A_{m \times n} + B_{m \times n}$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & & \vdots \\ b_{m2} & b_{m2} & & b_{mn} \end{bmatrix}$$

2. **Multiplication:**

Two matrices A & B of orders $m \times n$ & $n \times m$ are multiplied then the product matrix would be of order $m \times m$

Topic _____ Date _____

$$C_{m\times m} = A_{m\times n} \times B_{n\times m}$$

$$C_{m\times m} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{nm} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix}$$

where $C_{m\times m} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ \vdots & \vdots & & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mm} \end{bmatrix}$

3. Transpose : The Transpose of a matrix A can be obtained by interchanging its rows & columns.

$$A^T \text{ or } A' = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & & \vdots \\ a_{12} & a_{2n} & \cdots & a_{nm} \end{bmatrix}$$

On taking transpose of a matrix of order $m\times n$ we get a matrix of order $n\times m$.

Topic _____ Date _____

Program Code:

```
#include<stdio.h>
#include <conio.h>

Void main()
  {
     int  a[30][30], b[30][30], c[30][30], i,j,K,m;
     printf ("Enter dimensions of mxm matrix: ");
     scanf("%d", &m);
     printf("Enter the elements of 1st Matrix :- \n");

     For (i=0; i<m; i++)
       {
         For(j=0; j<m; j++)
          {
            scanF("%d", & a[i][j]);
          }
         printF("\n");
       }

     printf("Enter the elements of 2nd Matrix :- \n");
     For (i=0; i<m; i++)
      {
        For(j=0; j<m; j++)
        {
           scanf("%d", &b[i][j]);
```

Topic _____ Date _____

```
    }
  }
  printf("\n Addition: \n");
  for(i=0; i<m; i++)
    {
     for(j=0; j<m; j++)
      {
        c[i][j] = a[i][j] + b[i][j];
      }
        printf("%d \t", c[i][j]);
    }
    printf("\n");
  }

  printf("\n Multiplication: \n);
  for(i=0; i<m; i++)
   {
     printf("\n");
     for(j=0; j<m; j++)
     {
        c[i][j]=0
        for(k=0; k<m; k++)
        {
          c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
        printf("%d \t", c[i][j]);
     }
  }
```

Topic _____ Date _____

```
printf("\n Transpose of Matrix A: \n");
  for(i=0; i<m; i++)
   {
    for(j=0 ; j<m; j++)
    {
      c[i][j] = a[j][i];
      printf("%d \t", c[i][j]);
    }
    printf("\n");
  }
  printf("\n Transpose of Matrix B: \n");
   for(i=0; i<m ; i++)
    {
     for(j=0; j<m;j++)
      {
        c[i][j] = b[j][i] ;
        printf("%d \t", c[i][j]);
       }
      printf("\n");
    }
    getch();
  }
```

Result:

We performed the algebraic operations on matrices.

Output:

Enter dimensions of m×m matrix: 3

Enter the elements of 1st Matrix :-

1 2 3

4 5 6

7 8 9

Enter the elements of 2nd Matrix :-

4 6 0

3 2 1

0 5 7

Addition:

5 8 3

7 7 7

7 13 16

Multiplication:

10 25 23

31 64 47

52 103 71

Transpose of Matrix A:

1 4 7

2 5 8

3 6 9

Transpose of Matrix B:

4	3	0
6	2	5
0	1	7

Page No.
Date

Experiment No. 5

Aim:
To integrate numerically using trapezoidal rule. For $n=8$, find the value of $\int_1^5 \sqrt{1+x^2}\,dx$

Software Used:
Turbo C

Theory:
In numerical analysis, the trapezoidal Rule is a technique for approximating the definite integral

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

It works by approximating the region under the graph of the function $f(x)$ as a trapezoid & calculating the area. It follows that -

$$\int_a^b f(x)\,dx = b-a\left[\frac{f(a)+f(b)}{2}\right]$$

For more accurate results, the domain of graph is divided into n segments of equal size $\boxed{h=\frac{b-a}{n}}$

Approximate Value of Integral can be given by

$$\boxed{\int_a^b f(x)\,dx = \frac{h}{2}\left[f(a)+f(b)+\sum_{i=1}^{n-1} f(a+ih)\right]}$$

Page No. _____
Date

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

Float Function (Float);

void main()
 {
   clrscr();
   Float h=0.5, a=1.0, b=5.0;
   Float sum=0.0;
   Float Fa = Function(a);
   Float Fb = Function(b);

   For(Float i=1.5; i<5.0; i=i+0.5)
   {
    sum += Function(i);
   }

  Float integral = h* ((Fa+Fb)/2+sum);
printf("%f", integral);
getch();

}
```

Page No.
Date

```
Float Function (Float x)
{
  return sqrt(pow(x,2)+1);
}
```

Result

We integrated the given function numerically by using Trapezoidal Rule.

Output:
12.761626

Experiment No. 6

Aim: To Integrate Numerically Using Simpson's One-Third Rule

Software Used: Turbo C7

Theory: Simpson's 1/3rd rule is an extension of the trapezoidal rule in which the integrand is approximated by a second-order polynomial. Simpson rule can be derived from the various way using Newton's divided difference polynomial, Lagrange polynomial, and the method of coefficients. Simpson's 1/3 rule is defined by:

$\int_a^b f(x)\,dx = h/3[(y_0+y_n) + 4(y_1+y_3+y_5+....+y_{n-1})+2(y_2+y_4+y_6+.....+y_{n-2})]$

This rule is known as **Simpson's One-third rule**.

Program Code: Function Sqrt($1+x^3$)

```
void main()
{
 clrscr();
 float a,b,n;
 float h,sum=0,x,val;

 printf("Function: Sqrt(1+x³)\n");
 printf("Enter Lower Limit:");
 scanf("%f",&a);
 printf("Enter Upper Limit:");
 scanf("%f",&b);
 printf("Enter Number of Divisions:");
 scanf("%f",&n);

 h=(b-a)/n;
 sum+=func(a)+func(b);
 for(int i=1;i<n;i++)
 {
  x=a+i*h;
  val=func(x);
  if(i%2==0)
  {
   sum+=2*val;
  }
  else
  {
   sum+=4*val;
  }
 }
 val=(sum*h)/3;
 printf("Value Obtained after Integration is %f",val);
 getch();
}

float func(float x)
{
 return sqrt(1+x*x*x);
```

```
}
```

Output:

```
Function: sqrt(1+x^3)
Enter Lower Limit:1
Enter Upper Limit:4
Enter Number of Divisions:6
Value Obtained after Integration is 12.871811
```

Program Code: Function Logx

```
void main()
{
 clrscr();
 float a,b,n;
 float h,sum=0,x,val;

 printf("Function: logx\n");
 printf("Enter Lower Limit:");
 scanf("%f",&a);
 printf("Enter Upper Limit:");
 scanf("%f",&b);
 printf("Enter Number of Divisions:");
 scanf("%f",&n);

 h=(b-a)/n;
 sum+=func(a)+func(b);
 for(int i=1;i<n;i++)
 {
  x=a+i*h;
  val=func(x);
  if(i%2==0)
  {
   sum+=2*val;
  }
  else
  {
   sum+=4*val;
  }
 }
 val=(sum*h)/3;
 printf("Value Obtained after Integration is %f",val);
 getch();
}

float func(float x)
{
 return log10(x);
}
```

Output:

```
Function: logx
Enter Lower Limit:1
Enter Upper Limit:5
Enter Number of Divisions:8
Value Obtained after Integration is 1.757440_
```

Experiment No. 7

Aim:

To Integrate Numerically Using Simpson's Three-Eighth Rule

Software Used:

Turbo C7

Theory:

Another method of numerical integration method called "Simpson's 3/8 rule". It is completely based on the cubic interpolation rather than the quadratic interpolation. Simpson's 3/8 or three-eight rule is given by:

$\int_a^b f(x)\,dx = 3h/8[(y_0+y_n)+3(y_1+y_2+y_4+y_5+....+y_{n-1})+2(y_3+y_6+y_9+.....+y_{n-3})]$

This rule quite more accurate than the standard method, as it uses one more functional value. For 3/8 rule, the composite Simpson's 3/8 rule also exists which is similar to the generalized form. The 3/8 rule is known as Simpson's second rule of integration.

Program Code:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

  float func(float);

  void main()
  {
   clrscr();
   float a,b,n;
   float h,sum=0,x,val;

   printf("Function: sqrt(1+x^3)\n");
   printf("Enter Lower Limit:");
   scanf("%f",&a);
   printf("Enter Upper Limit:");
   scanf("%f",&b);
   printf("Enter Number of Divisions:");
   scanf("%f",&n);

   h=(b-a)/n;
   sum+=func(a)+func(b);
   for(int i=1;i<n;i++)
   {
    x=a+i*h;
```

```
  val=func(x);
  if(i%3==0)
  {
   sum+=2*val;
  }
  else
  {
   sum+=3*val;
  }
 }
 val=(sum*h*3)/8;
 printf("Value Obtained after Integration is %f",val);
 getch();
}

float func(float x)
{
 return 1/(1+x);
}
```

Output:

```
Function: 1/(1+x)
Enter Lower Limit:0
Enter Upper Limit:3
Enter Number of Divisions:6
Value Obtained after Integration is 1.388839
```

Experiment No.8

Aim:

Solution of Initial Value Problem using Euler's Method.

Software Used: Turbo C7

Theory:

In order to use Euler's Method to generate a numerical solution to an initial value problem of the form:

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

we decide upon what interval, starting at the initial condition, we desire to find the solution. We chop this interval into small subdivisions of length *h*. Then, using the initial condition as our starting point, we generate the rest of the solution by using the iterative formulas:

$$x_{n+1} = x_n + h$$

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

Program Code:

```
#include<stdio.h>
#include<conio.h>

  float derivative(float ,float);
  void main()
  {
   clrscr();
   int n;
   float xc,xn,yc,yn,h;
   printf("Enter initial value of x:");
   scanf("%f",&xc);
   printf("Enter y at inital value of x:");
   scanf("%f",&yc);
   printf("Enter final value of x:");
   scanf("%f",&xn);
   printf("Enter Interval Value:");
   scanf("%f",&h);

   for(float x=xc+h;x<=xn;x=x+h)
   {
    yc=yc+h*derivative(x-h,yc);
   }
```

```
 printf("%f",yc);


 getch();
}

float derivative(float x, float y)
{
 return x+y+x*y;
}
```

Output:

Experiment No.9

Aim:
Solution of initial value problem using fourth order Runge Kutta method.

Software Used:
Turbo C7.

Theory:
In numerical analysis, the Runge kutta methods are a family of implicit and explicit iterative methods , which include the well known routine called Euler method.

$$y(x+h) = y(x) + \frac{1}{6}(F_1 + 2F_2 + 2F_3 + F_4)$$

where

$$F_1 = hf(x, y)$$

$$F_2 = hf\left(x + \frac{h}{2}, y + \frac{F_1}{2}\right)$$

$$F_3 = hf\left(x + \frac{h}{2}, y + \frac{F_2}{2}\right)$$

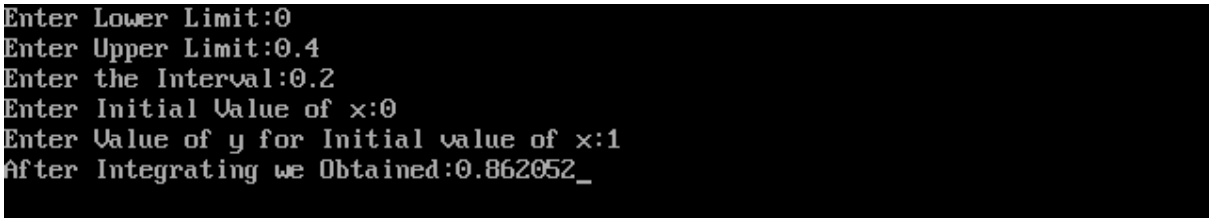
$$F_4 = hf(x + h, y + F_3)$$

Program Code:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

float f(float x,float y)
{
  float r;
  r=-2*x*y*y;
  return r;

}
void main()
{
  clrscr();
  float a,b,h,x,y;
  printf("Enter Lower Limit:");
  scanf("%f",&a);
```

```
printf("Enter Upper Limit:");
scanf("%f",&b);
printf("Enter the Interval");
scanf("%f",&h);
printf("Enter Initial Value of x");
scanf("%f",&x);
printf("Enter Value of y for Initial value of x:");
scanf("%f",&y);

 for(int i=1;i<=2;i++){

  float k1=h*f(x,y);
  float k2=h*f(x+(h/2),y+(k1/2));
  float k3=h*f(x+(h/2),y+(k2/2));
  float k4=h*f(x+h,y+k3);
  float k=(k1+(2*k2)+(2*k3)+k4)/6;
  y=y+k;
  x=x+h;
  }
printf("After Integrating we Obtained:%f",y);
getch();
}
```

Output:



Experiment No. 10

Aim:

Calculation of Eigen values and eigen vectors of matrices using power method.

Software Used:

Turbo C7.

Theory:

Eigenvalues and Eigenvectors: In linear algebra, an eigen vector or characteristic vector of a linear transformation is a nonzero vector that changes at most by a scalar factor when that linear transformation is applied to it. The corresponding eigenvalue is the factor by which the eigenvector is scaled.
In essence, an eigenvector v of a linear transformation T is a nonzero vector that, when T is applied to it, does not change direction. Applying T to the eigenvector only scales the eigenvector by the scalar value λ, called an eigenvalue. This condition can be written as the equation

$$\boldsymbol{Tv = \lambda v}$$

referred to as the eigenvalue equation or eigen equation. In general, λ may be any scalar. For example, **λ may be negative**, in which case the eigenvector reverses direction as part of the scaling, or it may be **zero or complex**.

Power Method: The power method is very good at approximating the extremal eigenvalues of the matrix, that is, the eigenvalues having largest and smallest module, denoted by λ_1 and λ_n respectively, as well as their associated eigenvectors. Solving such a problem is of great interest in several real-life applications (geosysmic, machine and structural vibrations, electric network analysis, quantum mechanics, ...) where the computation of λ_n (and its associated eigenvector x_n) arises in the determination of the proper frequency (and the corresponding fundamental mode) of a given physical system. Having approximations of λ_1 and λ_n can also be useful in the analysis of numerical methods.

Approximation of Eigenvalues: Let $A \in C_{nxn}$ be a diagonalizable matrix and let $X \in C_{nxn}$ be the matrix of its right eigenvectors xi, for i = 1, . . . , n. Let us also suppose that the eigenvalues of A are ordered as $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \ldots \geq |\lambda_n|$, where λ_1 has algebraic multiplicity equal to 1. Under these assumptions, λ_1 is called the dominant eigenvalue of matrix A. Given an arbitrary initial vector $q(0) \in C_n$ of unit Euclidean norm, consider for k = 1, 2, . . . the following iteration based on the computation of powers of matrices, commonly known as the power method:

$$Av = \lambda v$$

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|}$$

$$b_{k+1} = \frac{(A - \mu I)^{-1} b_k}{\|(A - \mu I)^{-1} b_k\|}$$

Program Code:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

void main()
{
    int i,j,n;
    float a[10][10],x[10],y[10],e[10],ymax,emax;
    printf("Enter the order of matrix \n");
    scanf("%d",&n);
    printf("Enter the Elements of the Given Matrix:-
\n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
            scanf("%f",&a[i][j]);
    }
    printf("Enter the column vector \n");
    for(i=0;i<n;i++)
    scanf("%f",&x[i]);
    do
    {
        for(i=0;i<n;i++)
        {
            y[i]=0;
```

```
            for(j=0;j<n;j++)
                y[i]=y[i]+a[i][j]*x[j];
        }
        ymax=fabs(y[0]);
        for(i=0;i<n;i++)
        {
            if((fabs(y[i]))>ymax)
            ymax=fabs(y[i]);
        }
        for(i=0;i<n;i++)
            y[i]=y[i]/ymax;
        for(i=0;i<n;i++)
            e[i]=fabs((fabs(y[i]))-(fabs(x[i])));
        emax=e[0];
        for(i=0;i<n;i++)
        {
            if(e[i]>emax)
            emax=e[i];
        }
        for(i=0;i<n;i++)
            x[i]=y[i];
    }while(emax>0.0001);
    printf("The required eigen value is: %f \n",ymax);
    printf("The required eigen vector is: \n");
    for(i=0;i<n;i++)
        printf("%f\t",y[i]);
    getch();
}
```

Output:

```
Enter Order of Matrix
3
Enter Elements of Given Matrix:-
-15 4 3
10 -12 6
20 -4 2
Enter the Column vector
1
1
1
The Required Eigen Value is: 19.999390
The Required Eigen Vector is:
1.000000        -0.499924       -1.000000
```