



SRIVENKATESHWARAA
COLLEGE OF ENGINEERING AND TECHNOLOGY
Ariyur, Puducherry-605 102.

DEPARTMENT OF MECHANICAL ENGINEERING

LAB MANUAL

SUBJECT / CODE: COMPUTATIONAL METHODS LAB/ MEP53

Prepared by

RAVINDRAN.S
AP/MECH

S.NO	NAME OF THE EXPERIMENT	REMARK
1	Finding roots of the given equation with single variable using Newton Raphson method	
2	Solution of system of linear equation using Gauss Elimination methods	
3	Matrix inversion by Gauss Jordan Method	
4	Eigen value of matrix by power method	
5	Solution of system of nonlinear equation using successive substitution	
6	Numerical single and double integration using trapezoidal and Simpson's one third rule	
7	Newton forward and backward difference interpolation	
8	Fourth order Runge-kutta method for solving first order ordinary differential equations	
9	Finite difference methods for solving second order differential equation	
10	Golden section method to find minimum of a single variable objective function	

NEWTON RAPHSON METHOD

AIM:

To write the source code to find the roots of the given nonlinear equation with single variable using Newton Raphson method.

DESCRIPTION:

Let x_0 be a good estimate of r and let $r = x_0 + h$. Since the true root is r , and $h = r - x_0$, the number h measures how far the estimate x_0 is from the truth.

Since h is 'small,' we can use the linear (tangent line) approximation to conclude that

$$0 = f(r) = f(x_0 + h) \approx f(x_0) + hf'(x_0),$$

and therefore, unless $f'(x_0)$ is close to 0,

$$h \approx -\frac{f(x_0)}{f'(x_0)}.$$

It follows that

$$r = x_0 + h \approx x_0 - \frac{f(x_0)}{f'(x_0)}.$$

estimate x_1 of r is therefore given by

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

The next estimate x_2 is obtained from x_1 in exactly the same way as x_1 was obtained from x_0 :

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}.$$

Continue in this way. If x_n is the current estimate, then the next estimate x_{n+1} is given by

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

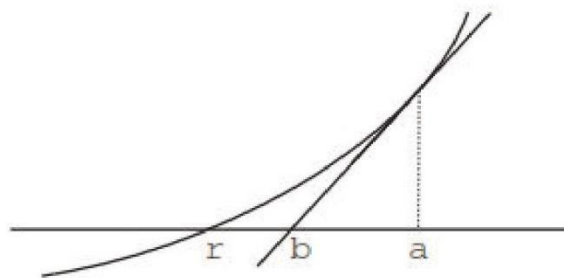
A Geometric Interpretation of the Newton-Raphson Iteration

In the picture below, the curve $y = f(x)$ meets the x -axis at r . Let a be the current estimate of r . The tangent line to $y = f(x)$ at the point $(a, f(a))$ has equation

$$y = f(a) + (x - a)f'(a).$$

Let b be the x -intercept of the tangent line. Then

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



Compare with Equation 1: b is just the ‘next’ Newton-Raphson estimate of r . The new estimate b is obtained by drawing the tangent line at $x = a$, and then sliding to the x -axis along this tangent line. Now draw the tangent line at $(b, f(b))$ and ride the new tangent line to the x -axis to get a new estimate c . Repeat.

ALGORITHM :

Step1: Read x , the initial root

Step2: Count=0

Step3: If $(f'(x)=0)$ then

 Write ‘the initial root is incorrect’

Endif

Step4: $y = x - f(x)/f'(x)$

Step5: If $(|f(y)| < 0.00001)$ then

Go to step9

Endif

Step6: Count = count + 1

Step7: If count > 500 then

Write 'an error has occurred'

Endif

Step8: $x = y$

Step9: Goto step3

Step10: write y

Step11: Stop

SOURCE CODE:

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
int power,i,cnt,flag;
int coef[10];
float x1,x2,t;
float fx1,fdx1;
void main()
{
    clrscr();
    printf("\n\n\t\t\t PROGRAM FOR NEWTON RAPHSON GENERAL");
    printf("\n\n\t\t\t ENTER THE TOTAL NO. OF POWER:::: ");
    scanf("%d",&power);
    for(i=0;i<=power;i++)
    {
        printf("\n\t x^%d:",i);
        scanf("%d",&coef[i]);
    }
    printf("\n");
    printf("\n\t THE POLYNOMIAL IS ::: ");
    for(i=power;i>=0;i--)//printing coeff.
    {
        printf(" %dx^%d",coef[i],i);
    }
    printf("\n\tINTIAL X1---->");
    scanf("%f",&x1);
    printf("\n ITERATION  X1  FX1  F'X1 ");
```

```

do
{
    cnt++;
    fx1=fdx1=0;
    for(i=power;i>=1;i--)
    {
        fx1+=coef[i] * (pow(x1,i)) ;
    }
    fx1+=coef[0];
    for(i=user_power;i>=0;i--)
    {
        fdx1+=coef[i]* (i*pow(x1,(i-1)));
    }
    t=x2;
    x2=(x1-(fx1/fdx1));
    x1=x2;
    printf("\n %d      %.3f %.3f %.3f ",cnt,x2,fx1,fdx1);
}while((fabs(t - x1))>=0.0001);
printf("\n\t THE ROOT OF EQUATION IS %f",x2);
getch();
}

```

OUTPUT:

```
PROGRAM FOR NEWTON RAPHSON GENERAL

ENTER THE TOTAL NO. OF POWER:::: 3

x^0::-1

x^1::1

x^2::0

x^3::1

THE POLYNOMIAL IS ::: 1x^3 0x^2 1x^1 -1x^0
INITIAL X1---->0.5

*****
ITERATION   X1    FX1    F' X1
*****
1           0.714  -0.375  1.750
2           0.683   0.079  2.531
3           0.682   0.002  2.400
4           0.682   0.000  2.397
THE ROOT OF EQUATION IS 0.682328
```

CONCLUSION:

Thus the source code to find the roots of the given nonlinear equation with single variable using Newton Raphson method is verified and executed.

GAUSS ELIMINATION METHODS

AIM:

To write the source code to find the Solution of system of linear equation using Gauss Elimination.

DESCRIPTION:

GAUSS ELIMINATION METHOD:

Let A be an $n \times k$ matrix and b is an $n \times 1$ vector. We wish to solve the equation

$$Ax = b$$

where $x \in \mathbb{R}^k$. One can write it out as follows

$$\begin{array}{cccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1k}x_k & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2k}x_k & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \dots & + & a_{nk}x_k & = & b_n \end{array}$$

With

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nk} \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

which is to be solved for x_1, \dots, x_n . Notice that the number of equations, n , is equal to the number of unknowns, so it is reasonable to expect that there should usually be a unique solution. Such systems arise in many different applications, and it is important to have an efficient method of solution, especially when n is large.

where A is an $n \times n$ matrix with entries $\{a_{ij}\}$, $x = (x_1, \dots, x_n)^T$ and $b = (b_1, \dots, b_n)^T$

If A is invertible, we can formally multiply through by A to obtain

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}.$$

ALGORITHM

Step 1 : Read number of variables in given system of equations ,n

Step 2 : For i=1 to n

Step 3 : For j=1 , n+1

Read a(i,j)

Step 4 : For k=1 to n-1

mx=|a(k,k)|

p=k

Step 5 : For m=k+1 to n

Step 6 : If (|a(m,k)|>mx) then

mx=|a(m,k)|

p=m

endif

step 7 : If (mx<.00001) then

write 'ill-conditioned equations'

stop

endif

step 8 : For q=k,n+1

temp=a(k,q)

a(k,q)=a(p,q)

a(p,q)=temp

step 9 : For i=k+1 to n

u=a(i,k)/a(k,k)

step 10 : For j=k to n+1

step 11 : a(i,j)=a(i,j)-u*a(k,j)

step 12 : If $(|a(n,n)|=0)$ then
 write 'ill-conditioned equations'
 stop
endif

step 13 : $x(n)=a(n,n+1)/a(n,n)$

step 14 : For $i=n-1$ to 1 in steps of -1
 sum=0

step 15 : For $j=i+1$ to n in steps of 1

step 16 : sum=sum+a(i,j)*x(j)

step 17 : $x(i)= (a(i,n+1) - \text{sum}) / a(i,i)$

step 18 : For $i=1$ to n
 write x(i)

Step 19: Stop

SOURCE CODE:

Gauss Elimination method:

```
# include <stdio.h>
# include <conio.h>
int main()
{
int i, j, k, n ;
float a[20][20], x[20] ;
double s, p ;
printf("Enter the number of equations : ") ;
scanf("%d", &n) ;
printf("\nEnter the co-efficients of the equations :\n\n") ;
for(i = 0 ; i < n ; i++)
{
for(j = 0 ; j < n ; j++)
{
printf("a[%d][%d] = ", i + 1, j + 1) ;
scanf("%f", &a[i][j]) ;
}
printf("b[%d] = ", i + 1) ;
scanf("%f", &a[i][n]) ;
}
for(k = 0 ; k < n - 1 ; k++)
{
for(i = k + 1 ; i < n ; i++)
{
p = a[i][k] / a[k][k] ;
for(j = k ; j < n + 1 ; j++)
a[i][j] = a[i][j] - p * a[k][j] ;
}
}
x[n-1] = a[n-1][n] / a[n-1][n-1] ;
```

```

for(i = n - 2 ; i >= 0 ; i--)
{
s = 0 ;
for(j = i + 1 ; j < n ; j++)
{
s += (a[i][j] * x[j]) ;
x[i] = (a[i][n] - s) / a[i][i] ;
}
}
printf("\nThe result is :\n") ;
for(i = 0 ; i < n ; i++)
printf("\nx[%d] = %.2f", i + 1, x[i]) ;
getch() ;
}

```

OUTPUT:

```

Enter the number of equations : 2
Enter the co-efficients of the equations :
a[1][1] = 1
a[1][2] = 5
b[1] = 7
a[2][1] = -2
a[2][2] = -7
b[2] = -5
The result is :
x[1] = -8.00
x[2] = 3.00_

```

CONCLUSION:

Thus the source code to find the Solution linear equation using Gauss Elimination is verified and executed.

MATRIX INVERSION BY GAUSS JORDAN METHOD

AIM:

To write the source code to find the Matrix inversion by Gauss Jordan Method.

DESCRIPTION:

For inverting a matrix, Gauss-Jordan elimination is about as efficient as any other method. For solving sets of linear equations, Gauss-Jordan elimination produces both the solution of the equations for one or more right-hand side vectors b , and also the matrix inverse A^{-1}

$$(A|I) = \left(\begin{array}{cccc|cccc} a_{1,1} & a_{1,2} & \cdots & a_{1,3} & 1 & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,3} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & 0 & 0 & \cdots & 1 \end{array} \right)$$

Inverse of the above matrix:

$$(I|C) = \left(\begin{array}{cccc|cccc} 1 & 0 & \cdots & 0 & c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ 0 & 1 & \cdots & 0 & c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & c_{n,1} & c_{n,2} & \cdots & c_{n,n} \end{array} \right)$$

Following Steps are performed to obtain inverse of a matrix using jordan method:

- Let the coefficient matrix be 'A' whose inverse is to be obtained for this matrix form an augmented matrix
- Perform row manipulations repeatedly on $[A|I]$ such that it is converted to $[I|A^{-1}]$
- The matrix A^{-1} in $[I|A^{-1}]$ is the inverse of A

Matrix inversion:

- Step 1: : Read number of variables in given system of equations ,n
- Step 2: For i=1,n
- Step 3: For j=1,n
- Step 4: Input a(l,j)
- Step 5: Next j
- Step 6: Next i
- Step 7: For i=1,n
- Step 8: Input x9i)
- Step 9: Next i
- Step 10: Input itr,err
- Step 11: Lrgl=0,0
- Step 12: For k=1,itr
- Step 13: For i=1,n
- Step 14: Y(i)=0
- Step 15: For j=1,n
- Step 16: $y(i) = y(i) + a(i,j) * x(j)$
- Step 17: Next j
- Step 18: Next i
- Step 19: Lrg=y(1)
- Step 20: For i=2,n
- Step 21: If(y(i)>lrg)
- Step 22: Lrg=y(i)
- Step 23: Else
- Step 24: Next l
- Step 25: For i=1,n
- Step 26: $X(i) = y(i) / lrg$
- Step 27: Next l
- Step 28: Print x,lrg
- Step 29: For l-1,n
- Step 30: Print x(i)

Step 31: Next i
Step 32: If $(\text{abs}(\text{lrg}-\text{lrg}) < \text{err})$
Step 33: Stop
Step 34: Else
Step 35: $\text{Lrg1}=\text{lrg}$
Step 36: Next k
Step 37: Print "iteration not sufficient"
Step 38: Stop

SOURCE CODE:

```
#include<stdio.h>

int main(void)
{
    float
    a[10][10],b[10][10],tem=0,temp=0,temp1=0,temp2=0,temp4=0,temp5=0;

    int n=0,m=0,i=0,j=0,p=0,q=0;

    printf("Enter size of 2d array(Square matrix) : ");

    scanf("%d",&n);

    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("Enter element no. %d %d :",i,j);

            scanf("%f",&a[i][j]);

            if(i==j)
                b[i][j]=1;

            else
                b[i][j]=0;

        }
    }

    for(i=0;i<n;i++)
    {
        temp=a[i][i];

        if(temp<0)
```

```

temp=temp*(-1);

p=i;
for(j=i+1;j<n;j++)
{
    if(a[j][i]<0)
        tem=a[j][i]*(-1);
    else
        tem=a[j][i];
    if(temp<0)
        temp=temp*(-1);
    if(tem>temp)
    {
        p=j;
        temp=a[j][i];
    }
}

for(j=0;j<n;j++)
{
    temp1=a[i][j];
    a[i][j]=a[p][j];
    a[p][j]=temp1;
    temp2=b[i][j];
    b[i][j]=b[p][j];
    b[p][j]=temp2;
}

```

```

temp4=a[i][i];
for(j=0;j<n;j++)
{
    a[i][j]=(float)a[i][j]/temp4;
    b[i][j]=(float)b[i][j]/temp4;
}

for(q=0;q<n;q++)
{
    if(q==i)
        continue;

    temp5=a[q][i];
    for(j=0;j<n;j++)
    {
        a[q][j]=a[q][j]-(temp5*a[i][j]);
        b[q][j]=b[q][j]-(temp5*b[i][j]);
    }
}

for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%.3f  ",b[i][j]);
    }

    printf("\n");
}

```

```
    }  
    return 0;  
}
```

OUTPUT:

```
Enter size of 2d array(Square matrix) : 2  
Enter element no. 0 0 :1  
Enter element no. 0 1 :3  
Enter element no. 1 0 :2  
Enter element no. 1 1 :7  
7.000  -3.000  
-2.000  1.000
```

CONCLUSION:

Thus the source code to find the Solution linear equation using Gauss Elimination is executed and verified.

EIGEN VALUE OF MATRIX USING POWER METHOD

AIM:

To write the source code to find the Eigen value of matrix using power method.

POWER METHOD FOR APPROXIMATING EIGENVALUES:

Eigenvalues of an matrix A are obtained by solving its characteristic equation For large values of n, polynomial equations like this one are difficult and time-consuming to solve. Moreover, numerical techniques for approximating roots of polynomial equations of high degree are sensitive to rounding errors. In this section you will look at an alternative method for approximating eigenvalues. As presented here, the method can be used only to find the eigenvalue of A that is largest in absolute value—this eigenvalue is called the dominant eigenvalue of A.

$$\lambda^n + c_{n-1}\lambda^{n-1} + c_{n-2}\lambda^{n-2} + \dots + c_0 = 0.$$

The power iteration algorithm starts with a vector b_0 , which may be an approximation to the dominant eigenvector or a random vector. The method is described by the iteration

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

So, at every iteration, the vector b_k is multiplied by the matrix A and normalized.

Under the assumptions:

- A has an eigenvalue that is strictly greater in magnitude than its other eigenvalues
- The starting vector b_0 has a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue.

then:

- A subsequence of (b_k) converges to an eigenvector associated with the dominant eigenvalue

Note that the sequence (b_k) does not necessarily converge. It can be shown that:

$b_k = e^{i\phi_k}v_1 + r_k$ where: v_1 is an eigenvector associated with the dominant eigenvalue, and $\|r_k\| \rightarrow 0$. The presence of the term $e^{i\phi_k}$ implies that (b_k) does not

converge unless $e^{i\phi_k} = \mathbf{1}$ Under the two assumptions listed above, the

sequence (μ_k) defined by: $\mu_k = \frac{b_k^* A b_k}{b_k^* b_k}$ converges to the dominant eigenvalue.

ALGORITHM:

1. Start the program.
2. Read n
3. For i=0,1,n-1
4. For j=0,1,n-1
5. Read aij
6. Next j
7. Xi ← 1
8. Next i
9. K ← 0
10. For i=0,1,n-1
11. yi ← 1
12. For j=0,1,n-1
13. Yi ← yi + aijxj
14. Next j
15. Zi ← |yi|
16. Next i
17. Z ← z0
18. J ← 0
19. For i=0,1,n-1
20. If zj ≥ zn
21. Z ← zj
22. J ← i
23. Else next i
24. If z = yj
25. d ← Z
26. Else d ← -z
27. For i=0,1,n-1
28. X ← y/d

29. Next i

30. $K \leftarrow K+1$

31. If $K \geq 50$

32. Write d

33. Else go to 10

34. End

SOURCE CODE:

```
#include<stdio.h>

#include<conio.h>

#include<math.h>

void main()

{

    int i,j,k,n;

    float A[40][40],x[40],y[40],z[40],l,d,f,e,Z;

    printf("\nEnter the order of matrix:");

    scanf("%d",&n);

    printf("\nEnter matrix elements row-wise\n");

    for(i=0;i<n;i++)

    {

        for(j=0;j<n;j++)

        {

            printf("A[%d][%d]=",i,j);

            scanf("%f",&A[i][j]);

            x[i]=l;

        }

    }

    k=0;

    line:

    for(i=0;i<n;i++)

    {

        y[i]=0;
```



```
for(j=0;j<n;j++)
y[i]=y[i]+A[i][j]*x[j];
z[i]=fabs(y[i]);
}
Z=z[0];
j=0;
for(i=1;i<n;i++)
{
if(z[i]>=Z)
{
Z=z[i];
j=i;
}
}
if(Z==y[j])
d=Z;
else
d=-Z;
for(i=0;i<n;i++)
x[i]=y[i]/d;
k=k+1;
if(k>=50)
printf("The numerically Largest Eigen value is %f\n",d);
else
goto line;
getch();
```

OUTPUT:

```
Enter the order of matrix:2
Enter matrix elements row-wise
A[1][1]=2
A[1][2]=-12
A[2][1]=1
A[2][2]=-5

Enter the column vector
X[1]=1
X[2]=1

The required eigen value is 2.014085

The required eigen vector is :
-1.000000    -0.333916
```

CONCLUSION:

Thus the source code to find the Eigen value of matrix using power method is executed and verified.

SUCCESSIVE SUBSTRUCTIVE METHOD

AIM:

To write a source code to find the Solution of system of nonlinear equation using successive subtractive method.

DESCRIPTION:

The simplest one-point iterative root-finding technique can be developed by rearranging the function $f(x)$ so that x is on the left-hand side of the equation

$$x = g(x)$$

The function $g(x)$ is a formula to predict the root. In fact, the root is the intersection of the line $y = x$ with the curve $y = g(x)$. Starting with an initial value of x , as shown in Fig. 1.2a, we obtain the value of x_2 :

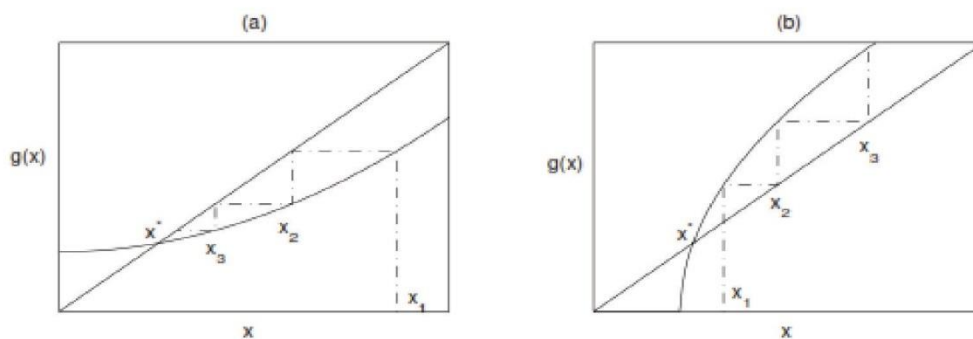
$$x_2 = g(x_1)$$

which is closer to the root than x_1 and may be used as an initial value for the next iteration. Therefore, general iterative formula for this method is

$$x_{n+1} = g(x_n)$$

which is known as the method of successive substitution or the method of $x = g(x)$.

A sufficient condition for convergence of Eq. (1.29) to the root x is that $|g'(x)| < 1$ for all x in the search interval. Fig. 1.2b shows the case when this condition is not valid and the method diverges. This analytical test is often difficult in practice. In a computer program it is easier to determine whether $x_3 - x_2 < x_2 - x_1$ and, therefore, the successive x values converge. The advantage of this method is that it can be started with only a single point, without the need for calculating the derivative of the function. n



(a) Convergence. (b) Divergence.

ALGORITHM

Step 1. Decide initial values of a & b and stopping criterion, E.

Step 2. Compute $f_1 = f(a)$ & $f_2 = f(b)$

Step 3. if $f_1 * f_2 > 0$, a & b do not have any root and go to step 7; otherwise continue.

Step 4. Compute $*x = (a+b) / 2$ and compute $f_0 = f(*x)$

Step 5. If $f_1 * f_0 < 0$ then

Set $b = *x$

Else

Set $a = *x$

Set $f_1 = f_0$

Step 6. If absolute value of $(b-a) / b$ is less than error E, then

root $= (a+b) / 2$

write the

value of root

go to step 7

else

go to step 4

Step 7. stop

SOURCE CODE:

```
#include<conio.h>

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

int user_power,i=0,cnt=0,flag=0;

int coef[10]={0};

float x1=0,t=0;

float x2=0;

void main()

{

    clrscr();

    printf("\n\n\t\t PROGRAM FOR SUCESSIVE SUBSTITUTION");

    printf("\n\n\t\t INTIAL X1---->");

    scanf("%f",&x2);

    /*****

    *****/

    printf("\n *****");

    printf("\n ITERATION  X1  FX1  ");

    printf("\n *****");

    do

    {

        cnt++;

        x1=x2;

        x2=(2-(log10(x1)));
```

```

printf("\n %d      %.3f %.3f ",cnt,x1,x2);

}while((fabs(x2 - x1))>=0.0001);

printf("\n\t THE ROOT OF EQUATION IS %f",x1);

getch();

}

```

OUTPUT:

```

                                PROGRAM FOR SUCESSIVE SUBSTITUTION
                    INITIAL X1---->0.5

*****
ITERATION      X1      FX1
*****
1              0.500  0.495
2              0.495  0.492
3              0.492  0.491
4              0.491  0.490
5              0.490  0.490
6              0.490  0.490
7              0.490  0.489
                THE ROOT OF EQUATION IS 0.489549_

```

CONCLUSION:

Thus the source code to find Solution of system of nonlinear equation using successive subtractive is executed and verified.

TRAPEZOIDAL AND SIMPSON'S ONE THIRD RULE:

AIM:

To write the source code to find numerical single and double integration using trapezoidal and Simpson's one third rule.

DESCRIPTION:

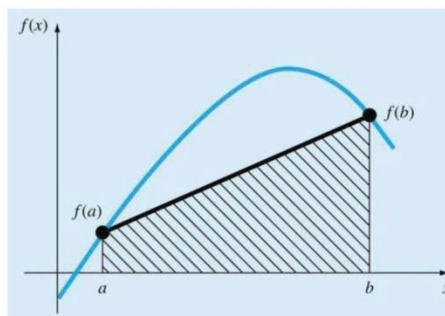
In mathematics, the trapezoidal rule is a way to approximately calculate the definite integral

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

The trapezoidal rule works by approximating the region under the graph of the function $f(x)$ as a trapezoid and calculating its area. It follows that

$$\int_a^b f(x) dx \approx (b - a) \frac{f(a) + f(b)}{2}.$$

To calculate this integral more accurately, one first splits the interval of integration $[a,b]$ into n smaller subintervals, and then applies the trapezoidal rule on each of them.



$$x_k = a + k \frac{b - a}{n}, \text{ for } k = 0, 1, \dots, n$$

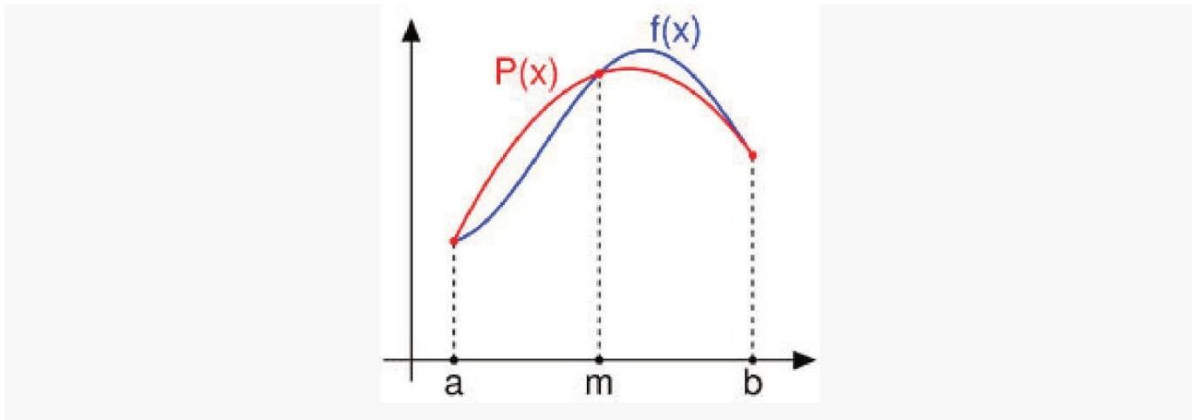
SIMPSON'S ONE THIRD RULE:

Simpson's 1/3 Rule Numerical Integration is used to estimate the value of a definite integral. It works by creating an even number of intervals and fitting a parabola in each pair of intervals. Simpson's rule provides the exact result for a quadratic function or parabola.

In numerical analysis, **Simpson's rule** is a method for numerical integration, the numerical approximation of definite integrals. Specifically, it is the following approximation:

$$\int_a^b f(x) dx \approx \frac{b-a}{6} [f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)] .$$

Simpson's rule also corresponds to the three-point Newton-Cotes quadrature rule.



ALGORITHM

- Step 1 : Read a,b the limits of integration
- Step 2 : Read n number of subintervals(number should be even)
- Step 3 : $h=(b-a)/n$
- Step 4 : $x=a$
- Step 5 : $y=f(x)$
- Step 6 : $sum=y$
- Step 7 : For $i=2$ to n $x=x+h$ $y=f(x)$
- Step 8 : If $\text{mod}(i,2)=0$ then $sum=sum+4*y$
- Step 9 : Else $sum=sum+2*y$
- Endif
- Step 10: $x=x+h$
- Step 11: $y=f(x)$
- Step 12: $sum=sum+y$

Step 13: $\text{sum} = h * \text{sum} / 3$

Step 14: write sum

Step 15: Stop

ALGORITHM:

Step 1 : Read a , b the limits of integration

Step 2 : If $b < a$ then $c = a$

$a = b$ $b = c$

Step 3: Read n , number of subintervals

Step 4: $h = b - a / n$

Step 5: $x = a$ $y = f(x)$ $\text{sum} = y$

Step 6: If $\text{count} < n$ then $x = x + h$

$y = f(x)$ $\text{sum} = \text{sum} + y$ $\text{count} = \text{count} + 1$ goto step 6

Step 7: else

Step 8: $x = x + h$ $y = f(x)$ $\text{sum} = \text{sum} + y$

Step 9: endif

Step 10: $\text{sum} = h * \text{sum} / 2$

Step 11: write sum

Step 12: Stop

SOURCE CODE:

Trapezoidal rule

```
#include<stdio.h>

#include<conio.h>

#include<math.h>

float fn(float x)

{

sqrt(x);

return sqrt(x);

}

main()

{

int i,n;

float a,b,s=0,y=0,h;

printf("Enter the no of interval =");

scanf("%d",&n);

printf("Enter the lower limit=");

scanf("%f",&a);

printf("Enter the upper limit=");

scanf("%f",&b);

h=(b-a)/n;

for(i=1;i<=n-1;i++)

{

s=s+fn(a+i*h);

}

}
```

```
y=(fn(a)+fn(b)+2*s)*h/2;
printf("the value of y is=%f",y);
getch(); }
```

Simpsons 1/3th rule:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
float f(float x);
float f(float x)
{
    return(x);
}
int main()
{
    int n,i;
    float s1=0,s2=0,sum,a,b,h;
    printf("Enter the value of upper limit = ");
    scanf("%f",&b);
    printf("Enter the value of lower limit = ");
    scanf("%f",&a);
    printf("Enter the number of intervals = ");
    scanf("%d",&n);
    h=(b-a)/n;
    if(n%2==0)
    {
```

```
for(i=1;i<=n-1;i++)
{
    if(i%2==0)
    {
        s1=s1+f(a+i*h);
    }
    else
    {
        s2=s2+f(a+i*h);
    }
}
sum=h/3*(f(a)+f(b)+4*s2+2*s1);
printf("the value is = %f",sum);
}
else
{
    printf("the rule is not appliciable");
}getch();
}
```

OUTPUT:

```
Trapezoidal Rule
Enter the no of interval =2
Enter the lower limit=1
Enter the upper limit=5
the value of y is=12.837789
```

```
Simpson 1/3 rule
Enter the value of upper limit = 5
Enter the value of lower limit = 1
Enter the number of intervals = 2
the value is = 12.774896
```

CONCLUSION:

Thus the source code to find code to find numerical single and double integration using trapezoidal and Simpson's one third rule is executed and verified.

RUNGE–KUTTA METHODS

AIM:

To write the source code to solve the first order ordinary differential equations by Fourth order Runge-kutta method .

DESCRIPTION:

RUNGE–KUTTA METHODS:

Let an initial value problem be specified as follows.

$$\dot{y} = f(t, y), \quad y(t_0) = y_0.$$

Here, y is an unknown function (scalar or vector) of time t which we would like to approximate; we are told that \dot{y} , the rate at which y changes, is a function of t and of y itself. At the initial time t_0 the corresponding y -value is y_0 . The function f and the data t_0, y_0 are given.

Now pick a step-size $h > 0$ and define

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4) \\ t_{n+1} &= t_n + h && \text{for } n = 0, 1, 2, 3, \dots, \text{ using} \\ k_1 &= f(t_n, y_n), \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right), \\ k_4 &= f(t_n + h, y_n + hk_3). \end{aligned}$$

Here y_{n+1} is the RK4 approximation of $y(t_{n+1})$, and the next value (y_{n+1}) is determined by the present value (y_n) plus the weighted average of four increments, where each increment is the product of the size of the interval, h , and an estimated slope specified by function f on the right-hand side of the differential equation.

- k_1 is the increment based on the slope at the beginning of the interval, using y , (Euler's method) ;
- k_2 is the increment based on the slope at the midpoint of the interval, using $y + \frac{h}{2}k_1$;

- k_3 is again the increment based on the slope at the midpoint, but now using $y + \frac{h}{2}k_2$;
- k_4 is the increment based on the slope at the end of the interval, using $y + hk_3$.

ALGORITHM

Step 1: Read x_1, y_1 initial values.

Step 2: Read a , value at which function value is to be found.

Step 3: Read n , the number of steps.

Step 4: $\text{count}=0$

Step 5: $h=(a-x_1)/n$

Step 6: write x_1, y_1

Step 7: $s_1=f(x_1, y_1)$

Step 8: $s_2=f(x_1+h/2, y_1+s_1*h/2)$

Step 9: $s_3=f(x_1+h/2, y_1+s_2*h/2)$

Step 10: $s_4=f(x_1+h, y_1+s_3*h)$

Step 11: $y_2=y_1+(s_1+2*s_2+2*s_3+s_4)*h/6$

Step 12: $x_2=x_1+h$

Step 13: write x_2, y_2

Step 14: $\text{count}=\text{count}+1$

Step 15: If $\text{count}<n$. then

$x_1=x_2$

$y_1=y_2$

go to step step 7

endif

Step 16: write x_2, y_2

Step 17: Stop

SOURCE CODE:

```
#include<stdio.h>

#include<conio.h>

#include<math.h>

float f(float x,float y)

{

return x+y*y;

}

main ()

{

float x0,y0,h,xn,x,y,k1,k2,k3,k4,k;

clrscr ();

printf("Enter the value of x0,y0,h,xn \n");

scanf("%f %f %f %f",&x0,&y0,&h,&xn);

x=x0;

y=y0;

while(1)

{

if (x==xn);

break;

k1=h*f(x,y);

k2=h*f(x+h/2,y+k1/2);
```

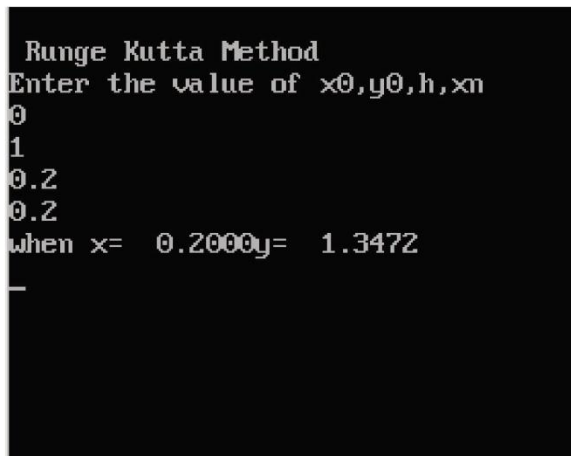


```

k3=h*f(x+h/2,y+k2/2);
k4=h*f(x+h,y+k3);
k=(k1+(k2+k3)*2+k4)/6;
x=x+h;
y+=k;
printf("when x=%8.4f""y=%8.4f \n",x,y);
}
}

```

OUTPUT:



```

Runge Kutta Method
Enter the value of x0,y0,h,xn
0
1
0.2
0.2
when x= 0.2000y= 1.3472
—

```

CONCLUSION:

Thus the source code to find the to solve the first order ordinary differential equations by Fourth order Runge-kutta method is executed and verified.

NEWTON FORWARD AND BACKWARD DIFFERENCE INTERPOLATION

AIM:

To write the source code for Newton forward and backward difference interpolation.

DESCRIPTION:

Newton Backward Difference Interpolation polynomial:

If the data size is big then the divided difference table will be too long. Suppose the desired intermediate value (\tilde{x}) at which one needs to estimate the function $(i.e. f(\tilde{x}))$ falls towards the end or say in the second half of the data set then it may be better to start the estimation process from the last data set point. For this we need to use backward-differences and backward difference table.

Let us first define backward differences and generate backward difference table, say for the data

set $(x_i, f_i), i = 0, 1, 2, 3, 4.$

First order backward difference ∇f_i is defined as:

$$\nabla f_i = f_i - f_{i-1}$$

Second order backward difference $\nabla^2 f_i$ is defined as:

$$\nabla^2 f_i = \nabla f_i - \nabla f_{i-1}$$

In general, the k^{th} order backward difference is defined as

$$\nabla^k f_i = \nabla^{k-1} f_i - \nabla^{k-1} f_{i-1}$$

Newton Forward Difference Approach:

Very often it so happens in practice that the given data set $(x_i, y_i), i = 0, 1, \dots, n$ correspond to a sequence $\{x_i\}$ of equally spaced points. Here we can assume that

$$x_i = x_0 + ih, \quad i = 0, 1, 2, \dots, n$$

where x_0 is the starting point (sometimes, for convenience, the middle data point is taken as x_0 and in such a case the integer i is allowed to take both negative and positive values.) and h is the step size. Further it is enough to calculate simple differences rather than the divided differences as in the non-uniformly placed data set case. These simple differences can be forward differences (Δf_i) or backward differences (∇f_i) . We will first look at forward differences and the interpolation polynomial based on forward differences.

The first order forward difference Δf_i is defined as

$$\Delta f_i = f_{i+1} - f_i$$

The second order forward difference $\Delta^2 f_i$ is defined as

$$\Delta^2 f_i = \Delta f_{i+1} - \Delta f_i$$

The k^{th} order forward difference $\Delta^k f_i$ is defined as

$$\Delta^k f_i = \Delta^{k-1} f_{i+1} - \Delta^{k-1} f_i$$

NEWTON FORWARD ALGORITHM:

Step 1. Read number of total data point and values of those data points

X and y =f(x)

Step 2. Read value of x=xn for which y is to be interpolated.

Step 3. Calculate forward differences

$$\Delta y_0 = y_1 - y_0$$

$$\Delta^2 y_0 = \Delta y_1 - \Delta y_0$$

$$\Delta^3 y_0 = \Delta^2 y_1 - \Delta^2 y_0 \text{ and so on}$$

Step 4. Calculate

$$H = x_1 - x_0$$

$$\text{And } r = \frac{x_r - x_0}{h}$$

Step 5. Calculate

$$Y_r = y_0 + r \Delta y_0 + \frac{r(r-1)}{2!} \Delta^2 y_0 + \frac{r(r-1)(r-2)}{3!} \Delta^3 y_0 + \dots$$

Step 6. The interpolated value of y at x=xr is equal to y0

Step 7. Display Xr and Yr

Step 8. Stop the program

NETWON BACKWARD:

Step 1. Read number of total data point and values of those data points

x and y =f(x)

Step 2. Read value of x=xn for which y is to be interpolated

Step 3. Calculate forward differences

$$\Delta y_n = y_n - y_{n-1}$$

$$\Delta^2 y_n = \Delta y_n - \Delta y_{n-1}$$

$$\Delta^3 y_n = \Delta^2 y_n - \Delta^2 y_{n-1} \text{ and so on}$$

Step 4. Calculate

$$H = x_1 - x_0$$

$$\text{And } r = \frac{x_r - x_0}{h}$$

Step 5. Calculate

$$Y_r = y_n + r \Delta y_n + \frac{r(r-1)}{2!} \Delta^2 y_n + \frac{r(r-1)(r-2)}{3!} \Delta^3 y_n + \dots$$

Step 6. The interpolated value of y at x=xr is equal to y0

Step 7. Display Xr and Yr

Step 8. Stop the program

SOURCE CODE:

Newton Forward:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
main()
{
    float x[20],y[20],f,s,h,d,p;
    int j,i,n;
    printf("enter the value of n :");
    scanf("%d",&n);
    printf("enter the elements of x:");
    for(i=1;i<=n;i++)
    {
        scanf("%f",&x[i]);
    }
    printf("enter the elements of y:");
    for(i=1;i<=n;i++)
    {
        scanf("%f",&y[i]);
    }
    h=x[2]-x[1];
    printf("Enter the value of f:");
    scanf("%f",&f);
```

```

s=(f-x[1])/h;
p=1;
d=y[1];
for(i=1;i<=(n-1);i++)
{
    for(j=1;j<=(n-i);j++)
    {
        y[j]=y[j+1]-y[j];

    }
    p=p*(s-i+1)/i;
    d=d+p*y[1];
}
printf("For the value of x=%6.5f The value is %6.5f",f,d);
getch();
}

```

Netwon backward:

```

#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
main()
{
    float x[20],y[20],f,s,d,h,p;

```

```

int j,i,k,n;

printf("enter the value of the elements :");

scanf("%d",&n);

printf("enter the value of x: n");

for(i=1;i<=n;i++)

{

    scanf("%f",&x[i]);

}

printf("enter the value of y: nn");

for(i=1;i<=n;i++)

{

    scanf("%f",&y[i]);

}

h=x[2]-x[1];

printf("enter the searching point f:");

scanf("%f",&f);

s=(f-x[n])/h;

d=y[n];

p=1;

for(i=n,k=1;i>=1,k<n;i--,k++)

{

    for(j=n;j>=1;j--)

    {

        y[j]=y[j]-y[j-1];

```



```
        }  
        p=p*(s+k-1)/k;  
        d=d+p*y[n];  
    }  
    printf("for f=%f ,ans is=%f",f,d);  
    getch();  
}
```

OUTPUT:

```
Newton Forward
  enter the value of the elements :5
enter the value of x: m0
1
2
3
4
enter the value of y: m1
7
23
55
109
enter the searching point f:0.5
for f=0.500000 ,ans is=3.125000_
```

```
Newton Backward
enter the value of n :5
enter the elements of x:0
1
2
3
4
enter the elements of y:1
7
23
55
109
Enter the value of f:0.5
For the value of x=0.50000 The value is 3.12500_
```

CONCLUSION:

Thus the source code for Newton forward and backward difference interpolation is executed and verified.

FINITE DIFFERENCE METHODS

AIM:

To write the source code to solve the second order differential equation using Finite difference methods.

DESCRIPTION:**THE FINITE DIFFERENCE SOLUTION :**

Let us divide the interval $[0, 1]$ into $n + 1$ equal parts with the points

$$0 = x_0 < x_1 < \dots < x_n < x_{n+1} = 1,$$

where $x_i = ih$ ($i = 0, \dots, n + 1$) with $h = 1/(n + 1)$. Moreover, let us introduce the notation u_i for the approximation of $\hat{u}_i = u(x_i)$ ($i = 0, \dots, n + 1$). Naturally, these values depend on h , albeit this is not indicated in the notation, furthermore $u_0 = \mu_1$ and $u_{n+1} = \mu_2$. We have n unknown values to compute: u_1, \dots, u_n . If we replace u'' by the centered difference approximations at the points x_i ($i = 1, \dots, n$), then we obtain the system of linear algebraic equations

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} + f(x_i) = 0, \quad (i = 1, \dots, n),$$

which, multiplying each equation by -1 , can be written in matrix form

$$\mathbf{A}_h \mathbf{u}_h = \mathbf{f}_h,$$

where $\mathbf{u}_h = [u_1, \dots, u_n]^\top \in \mathbb{R}^n$,

$$\mathbf{f}_h = [f(x_1) + \mu_1/h^2, f(x_2), \dots, f(x_{n-1}), f(x_n) + \mu_2/h^2]^\top$$

and \mathbf{A}_h is the tridiagonal matrix

$$\mathbf{A}_h = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & \dots & & \\ -1 & 2 & -1 & 0 & \dots & \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & \dots & 0 & -1 & 2 & -1 \\ & & & \dots & 0 & -1 & 2 \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

ALGORITHM:

Step 1:Read initial values of X_0 and y_0

Step 2;Read value of x at which y is to be calculated

Step 3:Read the step size h

Step 4:Calculate

Step 5: $Y_{n+1}=y_n+hf(x_n+y_n)$

Step 6:Increase x_n by one step

Step 7:Repeat step 4 and 5 till x_n becomes greater than x

Step 8:Display x_n,y_n on the screen in tabular format

Step 9:stop

SOURCE CODE:

```
#include <stdio.h>

main()
{
int i,j,k,m,n,x,y;
float a[20][20],l,r,t,b;
FILE *fp;
clrscr();
fp=fopen("c:\\laplace.dat","w"); //output will be stored in this file
printf("\t_____ \n");
printf("\tProgram to solve Laplace's equation by finite difference method\n");
printf("\t_____ \n");
printf("\tEnter boundary conditions\n");
```

```
printf("\tValue on left side: ");
scanf("%f",&l);

printf("\tValue on right side: ");
scanf("%f",&r);

printf("\tValue on top side: ");
scanf("%f",&t);

printf("\tValue on bottom side: ");
scanf("%f",&b);

printf("\tEnter length in x direction: ");
scanf("%d",&x);

printf("\tEnter number of steps in x direction: ");
scanf("%d",&m);

printf("\tEnter length in y direction: ");
scanf("%d",&y);

printf("\tEnter number of steps in y direction: ");
scanf("%d",&n);

m++;

n++; //number of mesh points is one more than number of steps

    for(i=1;i<=m;i++) //assigning boundary values begins
    {
        a[i][1]=b;
        a[i][n]=t;
    }

    for(i=1;i<=n;i++)
```

```

    {
        a[1][i]=l;
        a[m][i]=r;
    }          //assigning boundary values ends

for(i=2;i<m;i++)
for(j=2;j<n;j++)
a[i][j]=t; //initialization of interior grid points
for(k=0;k<100;k++)
{
    for(i=2;i<m;i++)
    {
        for(j=2;j<n;j++)
        {
            a[i][j]=(a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1])/4;
        }
    }
}          //calculation by Gauss-Seidel Method

for(i=1;i<=m;i++)
{
    for(j=1;j<=n;j++)
        fprintf(fp,"%0.2f\t",a[i][j]);
    fprintf(fp,"\n");
}

fclose(fp);

```

```
printf("\nData stored\nPress any key to exit...");  
  
    getch(); }
```

Output:

```
Program to solve Laplace's equation by finite difference method  
-----  
Enter boundary conditions  
Value on left side: 2  
Value on right side: 2  
Value on top side: 5  
Value on bottom side: 5  
Enter length in x direction: 10  
Enter number of steps in x direction: 10  
Enter length in y direction: 10  
Enter number of steps in y direction: 10  
  
Data stored  
Press any key to exit...
```

2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
5.00	3.50	2.91	2.65	2.53	2.50	2.53	2.65	2.91	3.50	5.00
5.00	4.09	3.50	3.16	2.98	2.93	2.98	3.16	3.50	4.09	5.00
5.00	4.35	3.84	3.50	3.31	3.24	3.31	3.50	3.84	4.35	5.00
5.00	4.47	4.02	3.69	3.50	3.44	3.50	3.69	4.02	4.47	5.00
5.00	4.50	4.07	3.76	3.56	3.50	3.56	3.76	4.07	4.50	5.00
5.00	4.47	4.02	3.69	3.50	3.44	3.50	3.69	4.02	4.47	5.00
5.00	4.35	3.84	3.50	3.31	3.24	3.31	3.50	3.84	4.35	5.00
5.00	4.09	3.50	3.16	2.98	2.93	2.98	3.16	3.50	4.09	5.00
5.00	3.50	2.91	2.65	2.53	2.50	2.53	2.65	2.91	3.50	5.00

2.00 2.00 2.00 2.00 2.00 2.00 2.00 2.00 2.00 2.00 2.00

CONCLUSION:

Thus the source code to solve the second order differential equation using Finite difference methods executed and verified.

GOLDEN SECTION METHOD

AIM:

To write the C program to find minimum of a single variable objective function by Golden section method.

DESCRIPTION:**GOLDEN SECTION METHOD:**

Consider a function f over the interval $[a; b]$. We assume that $f(x)$ is continuous over $[a; b]$; and that $f(x)$ is unimodal over $[a; b]$, i.e.: $f(x)$ has only one minimum in $[a; b]$.

The conditions above remind us to the bisection method and we will apply a similar idea: narrow the interval that contains the minimum comparing function values. In designing the method we seek to satisfy two goals. We want an optimal reduction factor for the search interval and minimal number of function calls. With these goals in mind we are going to determine the location to evaluate the function.

One choice inspired by the bisection method would be to compute the midpoint $m = (a + b)/2$ and to evaluate at x_1 and x_2 defined by $x_1 = m - \delta/2$ and $x_2 = m + \delta/2$, for some small value δ for which $f(x_1) \neq f(x_2)$. If $f(x_1) < f(x_2)$, then we are left with $[a; x_2]$, otherwise $[x_1; b]$ is our new search interval. While this halves the search interval in each step, we must take two new function evaluation in each step. This is not optimal.

ALGORITHM:

Step 1: Start

Step 2: Get an interval $[x_l, x_u]$ in which the maxima lies

Step 3: Calculate

$$d = \frac{\sqrt{5}-1}{2} (x_u - x_l)$$

Step 4: Calculate x_1 and x_2 such that

$$\begin{aligned}x_1 &= x_l + d \\x_2 &= x_u - d\end{aligned}$$

Step 5. Evaluate $f(x)$ at x_1 and x_2

$\text{If } f(x_2) > f(x_1), \quad x_u \leftarrow x_1 \text{ else } x_l \leftarrow x_2$

Step 6:if the required accuracy is not achieved the go to step 3

.X1 is the maxima

.Display the maxima and stop .

Step 7:Stop

SOURCE CODE:

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

```

#include<conio.h>

#include<stdlib.h>

#include<math.h>

void main()

{

double f(double x);

char ch;

double xu,xl,x1,x2,d,fx2,fx1;

clrscr ();

printf("\n\t Golden section search method \n");

printf("\n\n Enter xl=");

scanf("%lf",&xl);

printf("\n\n Enter xu=");

scanf("%lf",&xu);

printf("\n Press any key to see step by step display of results.....\n""press <q< to stop \n\n\t x\t\t\t
f(x)\n");

while(ch !='q')

{

d=(sqrt(5)-1)*(xu-xl)/2;

x1=xl+d;

x2=xu-d;

fx2=f(x2);

fx1=f(x1);

```

```
if(fx2>fx1)
xu=x1;
else
x1=x2;
printf("\n\t%f\t%f",x1,fx1);
ch=getch();
}
}
double f(double x)
{
double fx;
fx=2*sin(x)-(x*x/10);
return(fx);
}
```

OUTPUT:

```
Golden section search method

Enter x1=0

Enter xu=4

Press any key to see step by step display of results.....
press <q< to stop

      x                f(x)
2.472136          0.629974_
```

CONCLUSION:

Thus the source code to find minimum of a single variable objective function by Golden section method is executed and verified.