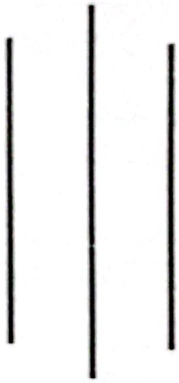
Tribhuvan University

Institute of Science & Technology

Mechi Multiple Campus

Bhadrapur, Jhapa

Lab Report (Numerical Method)



Submitted by:

Kiran Gajmer

Roll No: 27

Submitted To:

.....

Narayan Dhamala (M. Sc. CSIT)

Submission Date:

01/03

Table of Contents

2081/10/03

Table of Contents.

1. Write a program to implement Bisection Method.

Theory:

Bisection method is one of the simplest and most reliable type of iterative method for the solution of non linear equation: It is also known as binary chopping method or Half Interval Method It is bracketing method to solve the non-linear equation which means bisection method work within an interval consisting two values let say and 'b' such that $f(a) * f(b) \leq 0$, where $f(x)$ is non-linear equation that we have to solve. such that there is at leat one root inside the interval a and b.

In this method we compute next point which is mid point of a and b i.e c (let say)

$$c = \frac{a+b}{2}$$

Here, Three conditions exists

(i) $f(c) = 0$, we have root at 'c'.

(ii) $f(c).f(a) < 0$ there is root between a and c.

(iii) $f(c).f(b) < 0$ there is root between 'c' and 'b'.

The next interval is dicided as aoticing the sign of above conditions and methods is contin-ued iteratively.

In figure,

The next interval is c, b where $f(c) * f(b) =$ -ve values so there exist a root.

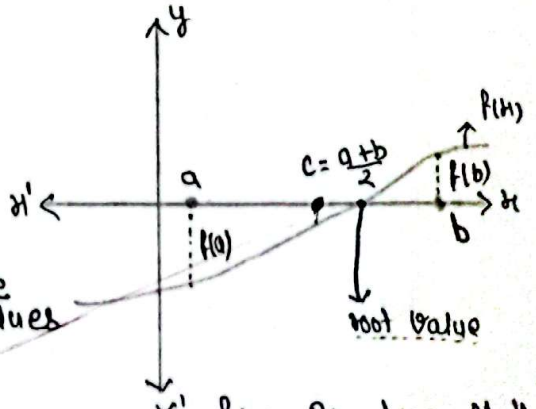


Fig:- Bisection Method.

'1'

Algorithm for Bisection Method.

1. Start.
2. Define the function fx i.e f(x) and error tolerance.
3. Input the initial guesses value a and b.
4. Compute fa = f(a) and fb = f(b).
5. Compare fa and fb as.
   - if $fa * fb \geq 0$ then there is no root exist in the given interval so go to step-3 for new initial guess values.
   - else continue.
6. Compute $c = (a+b)/2$ and fc = f(c).
7. Comape fc and product of fc and fa & fc and fb as:
   - if fc = 0, print the root as 'c'. & goto step-9
   - else if $(fa * fc < 0)$, then root is lie in 'a' and 'c' set b = c.
   - else set a = c
8. Test for accuracy.
   Compare tolerance as:
   - If abs $((b-a)/b) \leq E$ the print 'c' is the root of equation goto step-9
   - else goto step 6.
9. Stop!

Program for Bisection in C-programming language.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define F(x) (x*x*x - 4*x - 10)
#define E 0.0001

void main()
{
   float a, b, c, fa, fb, fc;
   Label_1:
   printf("Enter two initial guess values\n");
   scanf("%f%f", &a, &b);

   fa = F(a);
   fb = F(b);

   if(fa*fb >= 0)
   {
      printf("The root does not converge\n Enter coorect
          initial guess values.\n");
      goto Label_1;
   }
   else
   {
   Label_3;
   c = (a+b)/2;
   fc = F(c);
```

```
if(fc==0)
  {
   printf("The root of the equation is %f", c);
   goto Label_2;
  }
else if(fc*fa<0)
   {
   b=c;
    }
 else
    {
    a=c;
    }
 if(fabs((b-a)/b)<=E)
    {
     printf("The root of non-linear eqn. is %f", c);
    }
 else
    {
     goto label_3;
    }
  }
Label_2:
return;
}
```

Output

```
Enter the two initial guess values.
2
9
The root is 5.741394.
```

Conclusion

From the Lab, bisection method was successfully implemented to find the root of given non-linear equation. It gives accurate (root) result of $x^2-4x-10=0$ non-linear equation as 5.74139.

2. WAP to Implement false Positron Method.

Theory:

Regula falsi or false positron Method is a numerical method used to find the root of a real value function $f(x)=0$. It is based on intermediate value Theorem and uses linear inter-polation between two points on the function. It requires two Initial Guess values 'a' and 'b'.

The iterative formula for the Regula falsi method is computed as $x = \frac{af(b) - bf(a)}{f(b) - f(a)}$

and given as:- $$c = a - f(a) \times \frac{b-a}{f(b)-f(a)}$$

Algorithm:

1. Start.
2. Define a function for i.e. $F(x)$ and error tolerance.
3. Input the initial guess values a and b.
4. Compute $fa = f(a)$ and $fb = f(b)$.
5. Compare fa and fb with the below condition:
   if $(fa * fb >= 0)$ then root do not lies in the given interval so goto step-3 for new initial guess values.
6. Compute $c = a - f(a) \times (b-a)/(fb-fa)$ and $fc = f(c)$.
7. Check the following conditions
   if $fc = 0$, then print the value of 'c' as a root.
   else if $((fc * fa) < 0)$ then set $b = c$ and goto step-9.
   else set $a = c$
8. Test for accuracy.
   if $fabs((b-a)/b) <= E$ then, print the root as 'c'. and goto step-9
   else goto step-6.
9. Stop!

Program (Source code) for false Position Method.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define f(x) (x*x*x-4*x-9)
#define E 0.0001

void main()
  {
    float a, b, c, fa, fb, fc;
    Label_1:
    printf("Enter two initial guess values\n");
    scanf("%f %f", &a, &b);
    fa=f(a);
    fb=f(b);
    if (fa*fb >=0)
       {
           printf("The root does not lie converge In so
                   enter the correct initial guess.");
           goto Label_1;
       }
     else
       {
           Label_3:
           c= a-fa*(b-a)/(fb-fa);
           fc=f(c);

           if (fc==0)
              {
                 printf("The root is %f.", c);
                 goto label_2;
              }
```

```
        else if (fc * fa < 0)
            {
                b = c;
            }
        else
            {
             a = c;
            }
 if( fabs ( (b-a)/b) <= E)
        {
          printf ("The root is %f", c);
        }
   else
        {
        goto Label_3;
        }
   }
Label_2:
return;
  }
```

Output:

```
Enter two initial guess values.
2
5
The root is 2.706560.
```

Conclusion:

In this lab, false position method was successfully implemented to find the root of non-linear equation $x^3 - 4x - 9 = 0$ and the real root was successfully computed as 2.706560.

3. WAP to implement Newton's Raphson Method.

Theory:

The Newton Raphson method is an iterative method (numeric technique used to find the root of a real-valued function. It is one of the fastest methods for solving non-linear equations of the form: $f(x)=0$

The Newton Raphson iterative formula to solve non-linear equation is:

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

x_{n+1} = next approximation

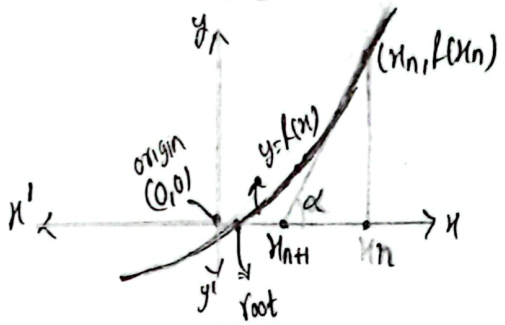


Fig:- Newton Raphson's Method.

Algorithm:

1. Start
2. Define the function for i.e $f(x)$ and Error tolerance.
3. Input initial guess value x_0.
4. Calculate $f_0 = f(x_0)$ and $G_0 = G(x_0) = f'(x)$
5. Calculate $X\_new = X_0 - f(X_0)/G(X_0)$ and $f(X\_new)$.
6. Test results and Accuracy:
   - if $F(X\_new == 0)$ then print x_new is root of given equation $f(x)=0$ and exist.
   - if $(fabs((X\_new - X_0)/X\_new) <= E)$ then, print the X_new as root of given equation and exit
   - else set $x_0 = X\_new$ and goto step-4.
7. Stop!

Program (Source code) of Newton Raphson Method in C.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define  f(x)  (x*x*x+5*x-3)
#define  G(x)  (3*x*x+5)
#define  E  0.001

void main()
{
float  x0, x_new, f0, g0, fx_new
printf("Enter the initial value x0.");
scanf("%f", &x0);

Label:
f0=F(x0);
g0=G(x0);

x_new=x0-f0/g0;
fx_new=f(x_new);
if (fx_new==0)
 {
printf("The root is %f.",x_new);
 return;
 }
if (fabs((x_new-x0)/x_new)<=E)
    {
    printf("The root is %f.", x_new);
    }
```

```
else
    {
    x0 = x_new;
    goto label;
    }
}
```

Output

```
Enter the initial guess value.
2
The root is 0.56410.
```

Conclusion:

In this lab, the Newton Raphson Method was implemented successfully. The non-linear equation $f(x) = x^3+5x-3 = 0$ was solved using the Newton Raphson numerical iterative technique and the solution with initial value 2 was computed as 0.56410.

4. WAP to Implement Secant Method.

Theory:

The secant method is an iterative method (numerical technique to find the root of a non-linear equation of the form $f(x)=0$. Unlike Newton-Raphson method, the Secant Method does not require derivate of the function. It is useful when the derivate of function is not easy to calculate.

Given two initial approximations x_0 and x_1, the next approximation x_2 is computed as x_{n+1}:

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

This formula uses the slope of the line (secant) connecting the points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$, and finds where this secant line intersects.

Algorithm:

1. Start
2. Define the function fx i.e. Fun1 and error tolerance
3. Input the initial guess values x0 and x1.
4. Calculate the value of f0=f(x0) and f1=f(x1).
5. Calculate x2= x1 - f1* (x1-x0)/(f1-f0); and f2=f(x2)
6. Test for result and accuracy:
   - if f2==0 then print root is x2 and return.
   - if (fabs((x2-x1)/x2) <= E then print the root as x2 and return.
   - else set x0=x1 and x1=x2 and goto step-4. and continue.
7. Stop!.

Program (C-program / Source Code) for Secant Method.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define f(x)  (x*x*x - 2*x - 5)
#define E  0.001

void main()
{
float x0, x1, x2, f1, f0, f2;
printf("Enter the initial guess values x0 and x1");
scanf("%f%f", &x0, &x1);

Label:
f0 = f(x0);
f1 = f(x1);

x2 = x1 - f1 * (x2 - x0) / (f1 - f0);
f2 = f(x2);

if (f2 == 0)
{
   printf("The root is %f.", x2);
return;
}

if ( fabs ((x2 - x1) / x2) <= E)
  {
  printf("The root is %f", x2);
}
```

```
        else
            {
            x0 = x1;    // x0 = x1;
            x1 = x2;    // x1 = x2;
            goto label;
            }
}
```

Output:

Enter the initial guess values.

1

2

The root is 2.094553.

Conclusion:

In this lab, the secant method was successfully implemented. The equation $f(x) = 0$ where $f(x) = x^3 - 2x - 5$ a non-linear equation was solved with initial guess values 1 and 2 with the result of root 2.094553.

5. WAP to implement fixed Point Method.

Theory:

The fixed point Iteration numerical technique used to find the root of non-linear equation of the form: $f(x) = 0$.

In this method, the equation is re written in the form of $x = g(x)$. A values x that satisfies $x = g(x)$ is called a fixed point and is the solution of the original equation.

formula:

Given an initial guess value x_0, the next approximation is given by:

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

This process is re-peated until the difference between successive iterations is within the desired tolerance:

$$\left| \frac{x_{n+1} - x_n}{x_{n+1}} \right| \leq \epsilon$$

Algorithm

1. Start.
2. Define a function $G(x)$ from the equation $f(x)=0$ such that $x = G(x)$ and error tolerance (E).
3. Input the initial guess value. x_0
4. Estimate the new value i.e $x1 = G(x_0)$. and $f(x1)$
5. Test for the result and accuracy:
   - if $(f1 == 0)$ then print the root as x1 and return.
   - if $(\text{fabs}((x1 - x0)/x1) <= E)$. then, print the root as $x1$ and exit.
   - else set, $x_0 = x1$ and goto step-6.
6. Stop!

Program (Source code) in C for Fixed point Iteration Method.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define F(x) (x*x*x - x*x - 1)
#define E 0.001
#define G(x) (1/sqrt(x+1))

void main()
{
   float x0, x1, f1;
   printf("Enter the initial guess value.\n");
   scanf("%f", &x0);
   Label:
   x1 = G(x0);
   f1 = F(x1);

   if(f1==0)
     {
       printf("The root is %f.", x1);
       return;
     }
   else if( fabs((x1-x0)/x1) <= E)
     {
       printf("The root is %f," x1);
     }

   else {
          x0 = x1;
          goto label;
        }
```

Output

Enter the initial guess value.

1

The root is 0.754774

Conclusion:

In this lab, the fixed point iteration method was implemented successfully to approximate the solution of non-linear equation $x^3 - x^2 - 1 = 0$. The result of the non-linear equation within 0.001 as desired tolerance is 0.754774.

6. WAP to implement linear Interpolation.

Theory:

Linear Interpolation is a method of estimating value of y at a point x lying between two known data points (x_0, y_0) and (x_1, y_1). by assumming a linear relationship between x and y. with that interval.

$$y = y_0 + (x - x_0)\left(\frac{y_1 - y_0}{x_2 - x_1}\right)$$

, where $y = f(x)$

x = value where the value of y to be estimated.

It is very simple and quick method.

Good Accuracy when data points are close and function is approximately linear.

Algorithm

1. Start.
2. Input (x_0, y_0) and (x_1, y_1) two known data points.
3. Input the value of x for which you want to find y where $x_0 < x < x_1$.
4. Apply the formula:

$$y = y_0 + \frac{(x - x_0)(y_1 - y_0)}{(x_1 - x_0)}$$

5. Output the interpolated value y.
6. Stop.

Program (Source code) for Linear Interpolation in C.

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

void main()
 {
 float x0, y0, x1, y1, x, y;
 printf("Enter the first known point:");
 scanf("%f %f", &x0, &y0);
 printf("Enter the second known point:");
 scanf("%f%f", &x1, &y1);

 Label:
 printf("Enter the value of x for which you want to
 find y:");
 scanf("%f", &x);
 if(x<x0 || x>x1)
  {
  printf("x is not between the range of x0 and
x1, so input again the value of x\n");
  goto Label;
  }

 y= y0+ ((x-x0) * (y1-y0))/ (x1-x0));

 printf("The interpolated value at x = %f is y=
  %f \n", x, y);
 }
```

Output

Enter the first known points.
2
3
Enter the second known points.
4
5
Enter the value of x for which you want to find y:
3
The interpolated value at x= 3.000000 is y=4.00000.

Conclusion:

In this lab, the linear Interpolation was implemented successfully by interpolating the value of y at given point x with the given that the x lies between x_0 and x_1 such that (x_0, y_0) and (x_1, y_1) are two known points. The value of y when x=3 is interpolated as 4 using linear interpolation method.

7. WAP to implement Lagrange's Interpolation.

Theory:

Lagrange's Interpolation is a powerful technique used to estimate the value of a function at any given point using a set of known data points. It is useful when the data points are unequally spaced.

This method constructs a polynomial of degree n (where n is one less than number of data points) which passes through all the given points exactly.

given., n+1 data points $(x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)$

Interpolating function $[P(x)] = \sum_{i=0}^{n} y_i L_i(x)$

where each $L_i(x)$ is the lagrangel basis polynomial delined as

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \left(\frac{y_i \cdot L_i(x)}{(x_i - x_j)} * (x - x_j) \right)$$

Each term $y_i \cdot L_i(x)$ contributes to the overall polynomial based on distane loom the interest x.

Algorithm

Start

Input the 'n'- numbers of points to be input

Input the the value of where interpolation is needed.

Input $(x_0, y_0), (x_1, y_1) \ldots (x_{n-1}, y_{n-1})$. (n-points)

5. Compute lagrange's basis polynomial.
   - for each $i = 0$ to $n-1$
     - Initialize $l_i = 1$
     - for each $j = 0$ to $n-1$ (except i)
       * compute:

$$L_i = l_i * \frac{x - x_j}{x_i - x_j}$$

6. Compute the interpolated value
   - Initialize $P(x) = 0$
   - for each $i = 0$ to $n-1$
     update

$$P(x) = P(x) + y_i * l_i$$

7. Output the interpolated value $P(x)$.
8. End!

Program: (Source code) implementing the lagrange's interpolation.

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

int main()
 {
 int n,i,j;
 printf(" Enter the value of n as number of
       points");
 scanf("%d",&n);

 float x[n], y[n], L[n]; //L[n] for lagrange's Basis.
//x[n] for all value of x; y[n] for values of y.
```

```
// Input data points
for (i=0; i<n; i++)
 {
printf ("x%d = ", i);
scanf ("%f", &x[i]);
printf("y%d = ", i);
scanf ("%f", &y[i]);
}
float X;
printf("Enter the point x at which value of y is
to be calculated: ");
scanf("%f ", &X);

// Calculate basis polynomials
for(i=0; i<n; i++)
  {
L[i]=0+1;
for(j=0; j<n; j++)
        {
        if (i!=j)
          {
          L[i] *= (X - x[j])/(x[i] - x[j]);
          }
        }
    }
// Compute Interpolated value
float S=0;
for (i=0; i<n; i++)
 {
S= S+ y[i]*L[i];
 }
printf("The value of y at point x=%f is %f\n", X,S);
}
```

```
return 0;
}
```

Output:

```
Enter the number of data points: 3
x0=1
y0=2
x1=2
y1=6
x2=6
y2=10
Enter the point X at which value of y
is to be calculated: 4
The value of y at the point y=4.00
is 10.4000.
```

Conclusion:

In this lab, the lagrange's Interpolation was implemented in C-programming taking tabulated functions as input $x_0=1$, $y_0=2$, $x1=2$, $y1=6$, $x2=6$ and $y2=10$ and at $X=4$ we had interpolated the value of y which was 10.400.

3. WAP to Implement Straight line curve fitting.

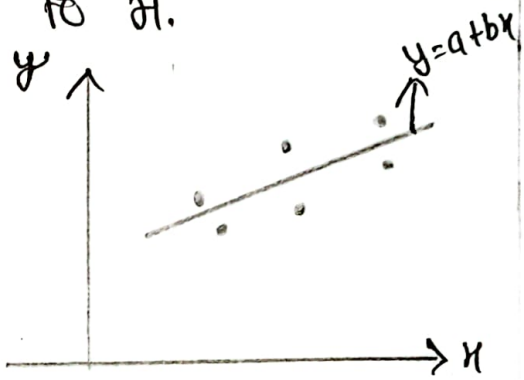
Theory:

Straight line curve fitting, often known as linear regression, is statistical method used to model the relationship between dependent variable y and independent variable x using a straight line. The goal is to find the best line (fit-line) in the form of:

$$y = a + bx$$

where,

a = intercept, repesenting y at x=0
b is the slope, increasing rate of which y changes with respect to x.



$$b = \frac{n\sum(x_i y_i) - (\Sigma(x_i).\Sigma(y_i))}{n\Sigma(x_i^2) - (\Sigma(x_i))^2}$$

$$a = \frac{\Sigma y_i - b\Sigma x_i}{n}$$

n = given no. of data points, and x_i and y_i are the individual data values.

Algorithm

- Start.
- Input the number of data points 'n'.
- Input each data elements associated with the data points x[i] and y[i].
- Compute:
  - Sum of $x_i.y_i$
  - Sum of x_i
  - Sum of y_i
  - Sum of $x_i.x_i$

&

b. - Compute b and a as:-

$$b = [(n \text{ sum}(x_i y_i) - (\text{sum } x_i \cdot \text{sum } y_i)) / (n \text{ sum}(x_i \cdot x_i)) - (\text{sum}(x_i) \cdot \text{sum}(x_i))];$$

$$a = (\text{sum } y_i - b * \text{sum } y_i) / n;$$

5. print the equation:-

$$y = a + b * x.$$

6. Stop!

Program for implementing the straight line curve fitting.

```
#include<stdio.h>
void main()
 {
int n, i;
float sum_x=0.00, sum_y=0.00, sum_xy=0.00, sum_x2
   =0.00;
float a, b;
 //Input data points:
printf for(i=0; i<n; i++)
     {
 printf("Enter x[%d] and y[%d]:", i,j);
 scanf("%f %f", &x[i], &y[i]);
 sum_x += x[i];
 sum_y += y[i];
 sum_xy += x[i]*y[i];
 sum_x2 += x[i] * x[i];
}
```

```
// Calculate slope (b) and intercept (a)
b = (n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x
                                    * sum_x);
a = (sum_y - b * sum_x) / n;

// Print the straight line equation
printf("The best fit line is y = %.2f + %.2f * x
    \n", a, b);

}
```

Output

```
Enter the number of data points: 5.
Enter x[0] and y[0]: 10 20
Enter x[1] and y[1]: 15 22
Enter x[2] and y[2]: 20 23
Enter x[3] and y[3]: 30 26
Enter x[4] and y[4]: 35 27
```

The best fit line $y = 17.51 + 0.28 * x$.

Conclusion:

In this lab, the fitting of curve in a straight line by using least square method was successfully implented. As we gave 5 data points as given and the best fit curve was calculated as $y = 17.51 + 0.28x$.

5. WAP to implement fitting of a trascendental Equation.

Theory:-

In curve fitting, sometimes the data follows non-linear pattern which cannot be accurately modeled by a straight line or polynomial. Such function are called transcendental functions and include exponetial, logarithmic and trigonometric equations. For eg $y = e^{bx}$, $y = a + b \log x$.

To fit these type of equations using least square method, we convert the non-linear equation into a linear form using suitable mathematical transformation. Once converted, we apply the standard regression techniques to find the values of constants.

let do the fitting curve that is formed by the transcendental Equation $y = ax^b$.

(not in linear)

$$\log_e y = \log_e a + \log_e x^b$$

$$Y = A + BX$$

converted into linear equation

now,

$$B = \frac{\Sigma \log x_i . \log y_i - \Sigma \log x_i * \Sigma \log y_i}{n * \Sigma \log x_i^2 - (\Sigma \log x_i)^2}$$

$$A = \frac{1}{n}(\Sigma \log y_i - b \Sigma \log x_i)$$

Algorithm

1. Start
2. Input the number of data points n. & each pair data (x_i, y_i) for $i=1$ to n.
3. Transform the given equation $y = ax^b$ into linear form $Y = A + bX$

   $Y = \log y$, $X = \log x$, $A = \log a$.
4. Compute the required sums.
   - ΣX
   - ΣY
   - ΣXY
   - ΣX^2
5. Apply least square formulas:

$$b = \frac{n\Sigma XY - \Sigma X \cdot \Sigma Y}{n\Sigma X^2 - (\Sigma X)^2}$$

$$A = \frac{1}{n}(\Sigma Y - b\Sigma X)$$

6. Compute a from $A = \log a \Rightarrow a = e^A$
7. Display the Result as the curved with value a and b.

$$y = ax^b$$

8. Stop!

Program to implementing fitting of Transcendental Equation $y = ax^b$.

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

void main()
{
int i, n;
```

```
//Input number of data points
printf("Enter the number of data points:");
scanf("%d", &n);

float x[n], y[n];
float sumX=0.0, sumY=0.0, sumXY=0.0, sumX2=0.0;
float a, b, A;

//Input the data points (x and y):\n");
printf("Enter the data points (x and y:\n");
for(i=0; i<n; i++)
    {
printf("x[%d] = ", i);
scanf("%f", &x[i]);
printf("y[%d] = ", i);
scanf("%f", &y[i]);
}

for(i=0; i<n; i++)
 {
 float X=log(x[i]);
 float Y = log(y[i]);
sumX += X;
sumY += Y
sumXY += X*Y;
sumX2 += X*X;
 }
b= (n* sumXY - sumX* sumY) / (n* sumX2 - sumX*sumX);
A =(sumY - b* sumX)/n;

a= exp(A);
printf("The fitted curve is y= %.4f * x^ %.4f\n",
    ,a,b);
return 0;  }
```

Output

Enter the number of data points: 6
Entere the data points (x and y):
x[0] = 2
y[0] = 16
x[1] = 4
y[1] = 17.1
x[2] = 6
y[2] = 8.7
x[3] = 8
y[3] = 6.4
x[4] = 10
y[4] = 4.7
x[5] = 12
y[5] = 2.6
The fitted curve is: y = 44.8653 * x^-0.9953.
~~Inclusion~~:

Inclusion:

In this lab, the fitting of transcendental equation was implemented by using C-programming language. The input 6 data points results the fitted curve as $y = 44.8653\, x^{-0.9953}$.

10. WAP to find the first & second derivative of a continuous function.

Theory:

Numerical differentiation is used to find approximate values of derivatives when a function is known but hard to differentiate analytically.

- First Derivative (slope):

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

- Second Derivative (curvature):

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

use small h eg $h = 0.0001$ or better accuracy.

Algorithm

1. Start
2. Define a function (continuous) $f(x)$
3. Enter the point at where the derivative is to be found.
4. Input a small step size. 'h'
5. Calculate the first derivative using
$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$
6. Calculate second Derivative using:-
$$f''(x) = \frac{f(x+h) + f(x-h) + 2f(x)}{h^2}$$
7. Display the values of first and second derivatives.
8. Stop!

C program for Numerical Differentiation

```
#include<stdio.h>
# define f(x)  (x*x +3*x+2)

void main()
{
double x, h;
double first_derivative, second_derivative;

printf("Enter the point x:");
scanf("%lf", &x);

printf("Enter small step size h (e.g., 0.001):");
scanf("%lf", &h);

first_derivative = (f(x+h) - f(x-h))/(2*h);
second_derivative= (f(x+h) - 2f(x) + f(x-h))/(h*h);

printf(" First Derivative at x= %lf is %.6lf\n", x,
        first_derivative);
printf(" Second_Derivative at x=%lf is %.6lf\n",
        x, second_derivative);
}
```

Output:

```
Enter the point x: 5
Enter the small step size h(e.g. 0.001): 0.01
First Derivative at x= 5.000 is 13.0000
Second Derivative at x= 5.000 is 2.0000
```

Conclusion:-

In this lab, the first and second derivatives of a continuous function was calculated by using C programming language.

The central difference method gives a simple and accurate way to find and second derivatives of a function using numerical calculations.

11. WAP to find integration using trapezoidal rule.

Theory:

The trapezoidal rule is a numerical method to approximate the definite integral of a function over an interval [a, b].

Formula:

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

where,

- $h = \frac{b-a}{n}$
- n be the number of sub-intervals.
- $x_i = a + i \cdot h$ for $i = 1, 2, 3, 4, \ldots n-1$

Algorithm

1. Start.
2. Define the function f(x).
3. Input the upper limit 'a' and lower limit 'b', and number of intervals n.
4. Compute: $h = \frac{b-a}{n}$
5. Initialize sum = f(a) + f(b)
6. loop from i=1 to n-1, compute x_i and add 2f(x_i) to sum
7. Multiply final sum by $\frac{h}{2}$
8. Display the result.
9. Stop!

C_program for the Trapezoidal Method

```
#include <stdio.h>
#define f(x)  (x*x)

void main() {
 double a, b, h, sum= 0.0;
 int n, i;

// Input a, b and n
 printf(" Enter Upper interval (a), lower interval b
        and interval size 'h': ");
 scanf("%lf %lf %lf", &a &b &n);

 h= (b-a)/n;
 sum = f(a) + f(b);
for(i=1; i<n; i++)
 {
 double x = a + i*h;
 sum +=2 * f(x);
  }
double result= (h/2)*sum;

printf(" Approximate value of the integral= %0.1f\n"
       , result);
 }
```

Output

Enter Upper limit (a), lower interval b, and interval size 'n' : 2 3 7

Approximate value of the integral = 0.336735.

Conclusion:

In this lab, Trapezoidal rule was implemented successfully for the numerical integration. The numerical integration of function $f(x) = x^2$ from 2 to 3 with taking 7 subinterval was 6.336735.

12. WAP to find integration using simpson's 1/3 rule.

Theory:

Simpson's $\frac{1}{3}$ Rule is a numerical method for approximating definite integrals. it is based on the assumption that the function can be ~~second~~ approximated by second-degree polynomial (parabola) over small subintervals.

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 4\sum_{odd} f(x_i) + 2\sum_{even} f(x_i) + f(x_n)\right]$$

$$\approx \frac{h}{3}\left[(f(x_0)+f(x_n)) + 4\sum_{odd} f(x_i) + 2\sum_{even} f(x_i)\right]$$

n = no. of sub-intervals (must be even)

$h = \frac{b-a}{n}$, $x_i = a + ih$.

Algorithm:-

1. Start
2. Define $f(x)$.
3. Input limits a, b, and even number of intervals n.
4. Compute $h = \frac{b-a}{n}$
5. Set sum = $f(a) + f(b)$
6. loop
   for (i = 1 to i = n-1)
   - Add $4f(x_i)$ if i is odd (to sum)
   - Add $2f(x_i)$ if i is even.
7. Multiply sum by h/3
8. Display result
9. Stop.

C program to find integration using Simpson's 1/3 rule.

```
#include <stdio.h>
#define f(x) (x*x)

void main()
{
double a, b, h, sum=0.0;
int n, i;

printf("Enter lower limit a, upper limit b and no. of subintervals:");
scanf("%lf%lf%d", &a, &b, &n);

if(n%2 !=0)
{
printf("n must be even!\n");
return;
}
h=(b-a)/n;
sum = f(a) + f(b);

for(i=1; i<n; i++)
{
double x = a + i*h;
if(i%2 == 0)
sum += 2*f(x);
else
sum += 4*f(x);
}
double result = (h/3) * sum;
```

```
printf("Approximate integral = %.4lf\n", result);
}
```

Output.

```
Enter lower limit a, upper limit b and even number
of intervals n: 2    3    8

Approximate Integral = 6.333.
```

Conclusion:

In this Lab, Simpson's $\frac{1}{3}$ Rule for numerical Intregation was implemented successfully. The value of $\int_a^b x^2 dx = \int_2^3 x^2 dx$ was evaluated or computed by simpson's $\frac{1}{3}$ method.

13. WAP to find Integration using Simpson's 3/8 method.

Theory:

Simpson's 3/8 Rule is another technique for numerical integration that approximates the integrand using a third-degree (cubic) polynomial. It is especially useful when the number of intervals is a multiple of 3.

The rule is given by:

$$\int_a^b f(x)\,dx \approx \frac{3h}{8}\left[f(x_0)+f(x_n)+3\sum f(x_i) \text{ where } i \bmod 3 \neq 0) + 2\sum f(x_i \text{ where } i \bmod 3 = 0)\right]$$

- $h = \frac{b-a}{n}$, n- is a multiple of 3.
- $x_i = a + ih$.

Algorithm:

1. Start.
2. Define f(x)
3. Input a, b and n (multiple of 3)
4. Compute h = (b-a)/n
5. Initialize sum = f(a) + f(b)
6. Loop:
   - if i mod 3 == 0, add 2f(x_i)
   - Else, add 3f(x_i)
7. Multiply final sum by $\frac{3h}{8}$.
8. Display result
9. Stop!

C. program for Simpson's $\frac{3}{8}$ Rule.

```
#include<stdio.h>
#define f(x) (x*x)

void main()
{
double a, b, h, sum=0.0;
int n, i;

printf("Enter lower limit a:");
scanf("%lf", &a);
printf("Enter upper limit b:");
scanf("%lf", &b);
printf("Enter the number of intervals(multiple of 3)
    :");
scanf("%d", &n);

if(n%3!=0) {
   printf("n must be a multiple of 3.\n");
   return;
           }

h= (b-a)/n;
sum = f(a) + f(b);
for(i=1; i<n; i++)
    {
double x = a+i*h;
if(i%3==0)
   sum += 2*f(x);
else
   sum +=3*f(x);
      }
```

```
double result = (3*h/8)* sum;
printf("Approximate integral = %.6lf\n", result);

}
```

Output

```
Enter lower limit a: 2
Enter upper limit b: 3
Enter the numbers of intervals (multiple of 3): 9
Approximate integral = 6.33333
```

Conclusion:

In this lab, the integration of a integral with in limit a to b was computed using simpson's $\frac{3}{8}$ rule. The function $f(x) = x^2$ having integral $\int_2^3 f(x)dx$ was computed successfully 6.3333.

4. WAP to solve system of linear equation using gauss elimination method.

Theory:

The Gauss Elimination method is a direct method used to solve a system of linear equations. It transforms the system into an upper triangular matrix form using elementary row operations and then solve it using back substitution.

Algorithm

Input:

Given a system of equation in augmented form of 'n' unknows with 'n' equation and Augmented matrix as [A:y] ⇒ A[n][n+1]

Output:-

Solution of system (x[1]. ---- x[n])

Steps

1. Start
2. Input the number of equations n and the augmented matrix A[n][n+1]
3. Forward Elimination (convert to upper triangular Matrix)
   - 3.1. For i=1 to n-1
     - 3.1.1. For j=i+1 to n
       - 3.1.1.1. Compute ratio = A[j][i] / A[i][i]
       - 3.1.1.2. For k=i to n+1
         - 3.1.1.2.1 A[j][k] = A[j][k] - ratio x A[i][k]

4. Backward Substitution (solve from last row upwards)

4.1. x[n] = A[n][n+1] / A[n][n]

4.2. for (i = n-1 down to 1)

4.2.1. x[i] = A[i][n+1]

4.2.2. for j = i+1 to n

4.2.3 x[i] = x[i] - A[i][j] * x[j]

4.2.3. x[i] = x[i] / A[i][i]

5. Display the solution x[1] to x[n]

End.

Program:-

```
#include<stdio.h>
#include<conio.h>
#define Max 10

int main() {
  int n, m, i, j, k;
  float A[MAX][MAX], X[MAX], ratio;

 printf("Enter no. of equations (n): ");
 scanf("%d", &n);
printf("Enter the. of Augmented Matrix (A|b): \n");
for(i=0; i<n; i++) {
 for (j=0; j<=n; j++) {
    printf("A[%d][%d]=", i, j);
   scanf("%f", &A[i][j]);
    }
 }
```

WAP to solve system of linear equation using Gauss Jordan method.

Theory: The Gauss - Jordan Method is an Extension of Gauss Elimination Method. It reduced the augmented matrix to the Reduced Row Echelon form (RREF) - a diagonal Matrix - from which solutions can be read directly without back substitution.

Algorithm:

Input: Number of equations 'n' and augmented matrix [A|b]

output: Solution of system x[1...n]

Steps:-

1. Start.
2. Input the number of equations 'n' and the augmented matrix A[n][n+1]
3. for k=0 to n-1.
   a. Make the pivot element A[k][k] =1 by dividing by the whole row by A[k][k]
   b. for each row (i ≠ k)
      (i) subtract a multiple of k^{th} row from i^{th} row to make A[i][k]=0.

After the above process the matrix becomes a diagonal matrix.

The last column of the matrix contains the solution of the system.

Display the values of unknowns x[1], x[2] ... x[n].

Stop!

15. WAP to solve system of linear equation using Gauss Jordan method.

Theory: The Gauss - Jordan Method is an Extension of Gauss Elimination Method. It reduced the augmented matrix to the Reduced Row Echelon form (RREF) - a diagonal Matrix - from which solutions can be read directly without back substitution.

Algorithm:

Input: Number of equations 'n' and augmented matrix [A|b]

output: Solution of system x[1...n]

Steps:-

1. Start.
2. Input the number of equations 'n' and the augmented matrix A[n][n+1]
3. for k=0 to n-1.
   a. Make the pivot element A[k][k] =1 by dividing by the whole row by A[k][k]
   b. for each row (i ≠ k)
      (i) subtract a mutiple of k^{th} row from i^{th} row to make A[i][k]=0.
4. After the above process the matrix becomes a diagonal matrix.
5. The last column of the matrix contains the solution of the system.
6. Display the values of unknowns x[1], x[2] ... x[n].
7. Stop!

C-program for Solving System of Equation using Gauss JORDAN METHOD.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
  {
int n, i, j, k, p, q;
float pivot, term, a[10][10];

printf("Enter Dimension of System of Equations\n");
scanf("%d", &n);

printf("Enter coefficients Augmented Matrix\n");
for(i=0; i<n; i++) {
for(j=0; j<n+1; j++) {
     scanf("%f", &a[i][j]);
       }
  }

for(k=0; k<n; k++) {
pivot = a[k][k];
   for(p=0; p<n+1; p++) {
    a[k][p] = a[k][p] / pivot;
    }

for(i=0; i<n; i++) {
term = a[i][k];
if(i!=k) {
   for(j=0; j<n+1; j++)
       {
      a[i][j] = a[i][j] - a[k][j] *term;
      }
```

```
        }
    }
}
printf("Solution: \n");
for(i=0; i<n; i++)
    {
    printf("X[%d] = %f\n", i+1, a[i][n]);
    }
getch();
return 0;
}
```

Output

```
Enter Dimension of system of Equations.
3
Enter coefficients Augmented Matrix
2     4     -6     -8
1     3      1      10
2    -4     -2     -12
Solution:
X1=1.00     X2.=2.000     X3=3.00
```

Conclusion:

In this lab, the system of linear equation was solved by using the Gauss JORDAN METHOD by using C- program.

The system of equation $2x_1 + 4x_2 - 6x_3 = -8$
$x_1 + 3x_2 + x_3 = 10$
$2x_1 - 4x_2 - 2x_3 = -1$ was solved and solution was computed as:.
$x_1 = 1.000$, $x_2 = 2.000$ and $x_3 = 3.000$.

WAP to solve system of linear equation using Jacobi iteration method.

Theory:

Jacobi iteration method is an iterative algorithm to solve the system of linear equation $Ax=b$, where A is a square matrix, x is the unknown (solution) and b is the constant vector.

Algorithm

Start

Input the system

a. Coefficient Matrix $A=[a_{ij}]$
b. Constant vector $b=[b_i]$
c. initial guess $x^{(0)} = [x_1^{(0)}, x_2^{(0)}, x_3^{(0)}, \dots x_n^{(0)}]$
d. Maximum number of iterations max_itera.
e. Tolerance level for stopping condition

Rearrange the equation (if necessary)

make sure each equation is solved for one variable

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_i \right)$$

Initialize

a. set iteration count $k=0$
b. let $x^{(0)}$ be the initial guess.

Repeat until convergence or maximum iterations

$k=0,1,2,3 \dots$ max_iteration

(a) each variable $i=1$ to n compute.

$$x_i^{(k+1)} = \frac{1}{a_{ij}} \left(b_i - \sum_{j=1, i\neq j}^{n} a_{ij} x_j^{(k)} \right)$$

(b) compute error: $error = \max(|x_1^{(k+1)} - x_1^{(k)}|, \dots |x_n^{(k+1)} - x_n^{(k)}|)$

(c) if error < (tol)

stop and return x^{k+1} as the solution.

else

update $x^{(k)} = x^{(k+1)}$ and continue.

6. Output:

Result the solution $x = [x[1] \ldots x[n]]$
If not converged, show a warning message

7. Stop!

Program (Source code in C language)

```
#include<stdio.h>
#include<math.h>
#define SIZE 10
void main() {
int n, max_iter;
float A[SIZE][SIZE], b[SIZE], x[SIZE], xnew[SIZE];
float tol; int i,j,max_iter, iter=0;
printf("Enter the number of equations (n): ");
scanf("%d", &n);
printf("Enter the coefficients of matrix A:\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("Enter constant vector b:\n");
for(i=0; i<n; i++) {
printf("b[%d]=", i+1);
scanf("%f", &b[i]);
}
```

```
printf("Enter the initial guess vector: x:\n");
for(int i=0; i<n; i++) {
printf("x[%d]:", i+1);
scanf("%lf", &x[i]);
    }

printf("Enter error tolerance:");
scanf("%lf", &tol);
printf("Enter the maximum number of iterations:");
scanf("%d", &max_iter);

while (iter<max_iter){
   float error=0.0;
     for(int i=0; i<n; i++) {
        float sum=0.0;
          for(j=0; j<n; j++) {
        if(j!=i) {
          sum+=A[i][j]*x[j];
                }
             }

 x_new[i]=(b[i]-sum)/A[i][i];
}


 for(i=0; i<n; i++) {
float diff=fabs(x_new[i]-x[i]);
if(diff>error) {
   error=diff;
         }

 x[i]=x_new[i];
   }
```

```
if(error < tol) {
printf("\n Converged in %d iterations.\n", iter+1);
  break;
        }
iter++;
  }
if(iter == max_iter) {
   printf("Did not converge\n");
                    }
printf("Solution:\n");
for(i=0; i<n; i++) {
printf("x[%d] = %.6f\n", i+1, x[i]);
        }

return 0;

}
```

Output

```
Enter the Dimension of System of Equations: 3
Enter the coefficient matrix.
2     4     -6
1     3      1
2     -4    -2
Ente constant vector b:
-8    10   -12
Enter the initial guess vector
x[1] = 0    x[2] = 0    x[3] = 0
Enter Error tolerance:
0.001
Enter the maximum number of iterations.");
10
```

Solution:

$x_1 = 1.0000$ $x_2 = 2.000$ $x_3 = 3.000$

Conclution:

In this lab, the system of linear algebraic equations was solved by using the Jacobi Iteration method.

The system of linear equation

$2x_1 + 4x_2 - 6x_3 = -8$

$x_1 + 3x_2 + x_3 = 10$

$2x_1 + (-4x_2) - 2x_3 = -12$

was solved as $x_1 = 1.00$, $x_2 = 2.00$ and $x_3 = 3.00$.

17. WAP to solve system of linear equation using Gauss seidel Method.

Theory:

The Gauss-Seidal Method is an iterative technique used to solve a system of linear equations of the form:

$$Ax = b.$$

where: A is square matrix of coefficients,
x is the column vector of variables,
b is the column vector of constants.

The system of equation is converted as:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} (b_i - a_{ij} x_2^{(k)} - a_{ij} x_3^{(k)}$$

Algorithm:

1. START
2. A Syste of linear equations $Ax = b$, ~~whe~~ is given.
3. Input the number of equations n.
4. Input the coefficient matrix and constants matrix.
5. Input the initial guess value for $x_1, x_2 \dots x_n$.
6. Input the allowed error tolerance ϵ.
7. Repeat the following steps (iterations) until convergence: for each equations i=1 to n:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

8. Check for convergence after each full iteration:
   (a) if $|x^{(k+1)} - x_i^{(k)}| < \epsilon$ for all i, then stop.
   (b) otherwise, set $k = k+1$ and repeat.
9. Output the values of $x_1, x_2 \dots x_n$ as the solution.

C program (Source code) to solve system of equation by Gauss-Seidal Method.

```
#include<stdio.h>
#include<math.h>
void main()
  {
int i,j,k,n=3,maxIteration = 100;
float a[3][4] = {
          { 4, -1, 1, 7},
          {2, 5, 2, -8},
          {1, 2, 4, 6}
                };
float x[3] ={0, 0, 0}, old[3];
float eps = 0.001, sum,error;

for(k=0; i<maxIteration; k++)  {
   for(i=0; i<n; i++)  old[i]=x[i];
for (i=0; i<n; i++) {
  sum=a[i][n];
   for (j=0; j<n; j++) {
     if (j!=i) sum -= a[i][j] * x[j];
        }

   x[i] = sum/ a[i][i];
}

error =0;
  for(i=0; i<n; i++) {
    if(fabs(x[i] - old[i])> error)
     error = fabs(x[i] - old[i]);
      }.
if (error <eps) break;
  }
```

```
printf("Solution: \n");
for(i=0; i<n; i++)
    printf("x[%d] = %.4f \n", i+1, x[i]);
}
```

Output:

Solution:

x[1] = 0.3190

x[2] = -2.8695

x[3] = 2.8550

Conclusion:

In this lab, the system of linear equation was solved using Gauss Seidal Method. The system of equation

$$4x_1 - x_2 - x_3 = 7$$
$$2x_1 + 5x_2 + 2x_3 = -8$$
$$x_1 + 2x_2 + 4x_3 = 6$$

was solved using Gauss Seidal Method (An Iterative method) and solution was computed as $x_1 = x[1] = 0.3190$, $x_2 = x[2] = -2.869$ and $[x_3) = x[3] = 2.8550$

18. WAP to implement Taylor's Series Method.

Theory:

Taylor's series Method is a numerical technique used to solve ordinary differential equations ODEs using the Taylor's series expansion of the solution.

It approximate the value of $y(x)$ using the Taylor's series expansion about an initial point.

$$\frac{dy}{dx} = f(x,y), \quad y(x_0) = y_0$$

Formula:-

$$y(x+h) = y(x) + hy'(x) + \frac{h^2}{2!}y''(x) + \frac{h^3}{3!}y'''(x) + \dots + \frac{h^n}{n!}y^n(x) + \dots$$

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

$$y''(x) = f''(x) = f'(f(x,y)) = \frac{dy}{dx}(f(x,y)).$$

Algorithm:

1. Start
2. Read the value of x_0 and y_0.
3. Read the value of which function to be evaluated, say x.
4. Compute value of y', y'', y''' and y^{10} .....
5. Comput $y(x) = y_0 + y'(x-x_0) + y''(x-x_0)^2/2! + (x-x_0)^3 \frac{y'''}{3!} + \dots$
6. Display the functional value at x i.e y.
7. Stop!

Source Code (C-program) to solve differential equation using Taylor's series method.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int fact(int n) {
  int (n==1)
      return 1;
      else
       return (n*fact(n-1));
         }

int main()
 {
float x, x0, y0, yx, fdy, sdy, tdy;
printf("Enter initial values of x & y.\n");
scanf("%f %f", &x0, &y0);

printf("Enter x at which function to be evaluated
       \n");
scanf("%f", &x);
fdy = (x0) + (y0); // first derivative.

sdy = 2*(x0) + 2*(y0) * fdy; //second derivative.
tdy = 2 + 2*y0 * sdy + 2*fdy* fdy; // Third derivative
yx = y0 + (x-x0)*fdy + (x-x0)*(x-x0)* sdy/fact(2)+
     (x-x0)(x-x0)*(x-x0)* tdy/fact(3);

printf" The value of function at x= %f is %f\n",
   x,yx);
getch();
```

```
return 0;
}
```

Output:

```
Enter initial value of x and y.
0    1
Enter x at which function to be evaluated
0.5
function value at x= 0.50000 is 1.916667
```

Conclusion:

In this Lab, the solution of ODE by using Taylor's series method was implemented by using the C-programming languages.

WAP to implement Euler's Method.

Theory:

Euler's Method is a simple and basic numeric technique used to find an approximate solution of a first-order ordinary differential Equation Equation (ODE).
It is used to solve $\frac{dy}{dx} = f(x,y)$, with initial condition $y(x_0) = y_0$. It gives approximate values of y at points near x_0 by using a small step size h.

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

x_n is current x-value, y_n is the value of y-approximate, h is step size., $f(x_n, y_n)$ is the derivative at the current point.

Algorithm:

Start
Read initial values of x and y say x_0 and y_0.
Read the value
at which functional value is required, say x_p.
Read step-size, say h.
set $y = x_0, y = y_0$
Compute value of y as below
for ($x = x_0$ to x_p)
$y = y + f(x,y)$
$x = x + h$
End for
Display functional value of y.
Terminate.

WAP to implement Euler's Method.

Theory:

Euler's Method is a simple and basic numerical technique used to find an approximate solution of a first-order ordinary differential Equation Equation (ODE).
It is used to solve $\frac{dy}{dx} = f(x,y)$, with initial condition $y(x_0) = y_0$. It gives approximate values of y at points near x_0 by using a small step size h.

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

x_n is current x-value, y_n is the value of y-approximate, h is step size., $f(x_n, y_n)$ is the derivative at the current point.

Algorithm:

1. Start
2. Read initial values of x and y say x_0 and y_0.
3. Read the value at which functional value is required, say x_p.
4. Read step size, say h.
5. Set $x = x_0, y = y_0$
6. Compute value of y as below

```
for (x = x0 to xp
    y = y + f(x,y)
    x = x + h
End for
```

7. Display functional value of y.
8. Terminate.

Program for implementing Euler's Method

```
#include<stdio.h>
float f(float x, float y) {
    return x+y;  // dy/dx = x+y;
        }
int main() {
 float x0, y0, h, xn, yn;
 int n,i;

 printf("Enter initial value of x(x0): ");
 scanf("%f", &x0);
 printf("Enter the initial value of y(y0):\n");
 scanf("%f", &y0);
 printf("Enter step-size h:");
 scanf("%f", &h);
 printf("Enter the number of steps: ");
 scanf("%d", &n);
 xn = x0;
 yn = y0;
 for(i= 0; i<n; i++)
     {
  yn = yn +h * f(xn, yn);
  xn = xn+h;
  printf("After step %d: x = %.4f, y= %.4f\n", i+1, xn, yn);
   }

printf("Approximate solution at x= %.4f is y= %.4f\n",
    xn, yn);
 return 0;
  }
```

Output

Enter initial value of x (x0): 0
Enter initial value of y (y0): 1
Enter step size (h): 0.1
Enter no. of steps: 5
After step-1: x = 0.10000, y = 1.1000
After step-2: x = 0.2000, y = 1.2200
After step-3: x = 0.3000, y = 1.3600
After step-4: x = 0.4000, y = 1.5282
After step-5 x = 0.5000, y = 1.7210

Approximate solution at x = 0.5000 is y = 1.7210.

Conclusion:

In this lab, The the EULER'S METHOD was Implemented successfully and the differential first order ODE $\frac{dy}{dx} = x + y$ was solved using EULER'S method at x = 0.500 taking step size h = 0.100 the final result was $y(x) = y(0.500) = 1.7210$.

20. WAP to implement Heun's Method.

Theory:

HEUN'S METHOD is a numerical technique to solve Ordinary Differential Equations (ODEs). It is an improvement over Euler's Method by using the average slope. It is improved version of Euler's Method. It is classified as predictor-corrector method where

$y_{i+1} = y_i + h \times f(x_i, y_i)$ → predictor

$y_{i+1} = y_i + \frac{h}{2}[f(x_i, y_i) + f(x_{i+1}, y_{i+1})]$ → corrector

$y_{i+1} = y_i + \frac{h}{2}(M_1 + M_2)$

where, $M_1 = f(x_i, y_i)$ and $M_2 = f(x_{i+1}, y_{i+2})$

final Heun's formula becomes:

$y_{i+1} = y_i + \frac{h}{2}[f(x_i, y_i) + f(x_{i+1}, (y_i) + hf(x_i, y_i)]$

Algorithm:

1. Start
2. Read initial value of x & y x0 and y0.
3. Read the value at which functional value is required, say xp.
4. Read step-size, say h.
5. Set x = x0 and y = y0.
6. Compute value of y(xp)

for x0 to xp:

```
m1 = f(x,y);
m2 = f(x+h, y+h*m1);
y = y + h/2 * (m1 + M2);
```

End for
7. Display functional value, y:
8. Terminate.

Program for Implementing HEUN's METHOD.

```
#include<stdio.h>
#include<conio.h>
#define f(x,y) (2*(y)+(x))

void main() {
    float x, xp, x0, y0, y, h, m1, m2;

    printf("Enter initial values of x0 and y0\n");
    scanf("%f%f", &x0, &y0);

    printf("Enter x at which function is to be evaluated: \n");
    scanf("%f", &xp);

    printf("Enter the step-size: \n");
    scanf("%f", &h);

x=x0;
y=y0;

while (x<xp) {
m1= f(x,y);
m2=f(x+h, y+h*m1);
y= y+ (h/2) * (m1+m2);
x= x+h;
printf("At x=%.4f, y= %.4f\n", x,y);
}
```

```
printf("\nFinal value at x=%.4f is y=%.4f\n", xp, y);
getch();
}
```

Output:

```
Enter initial values of x and y:
0
1
Enter x at which function is to be evaluated.
0.5
Enter step-size:
0.1

At x=0.1000, y=1.110
At x=0.2000, y=1.242
At x=0.3000, y=1.398
At x=0.4000, y=1.582
At x=0.5000, y=1.795
Final value at x=0.5000 is y=1.795.
```

Conclusion:

In this lab the first order ODE $\frac{dy}{dx} = x+y$ was solved Approximately at $x=0.500$ By using Heun's method and at $x=0.500$ Approximate value of $y=1.795$ was computed.

21. WAP to implement RK 4th order Method.

Theory:

Runge Kutta Fourth Order Method is widely used numerical technique for solving ordinary Differential Equation (ODEs). It is an extension of Heun's method, improving accuracy by using a weighted average of slopes at four different points within a given interval. Developed by C.D.T. Runge and M.W. Kutta, this method achieves an approximation error of order $O(h^4)$, making it significantly more precise than lower order methods like Euler's or Heun's.

Given initial value problem:

$$\frac{dy}{dx} = f(x, y), \quad y(x_0) = y_0.$$

where,

- $f(x, y)$ is the derivative function,
- h is the step size,
- (x_n, y_n) is the current point.

The RK4th order method approximates the next value y_{n+1} as:-

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where,

$$k_1 = f(x_n, y_n)$$
$$k_2 = f(x_n + h/2, y_n + h/2\, k_1)$$
$$k_3 = f(x_n + h/2, y_n + h/2\, k_2)$$
$$k_4 = (x_n + h, y_n + h k_3)$$

Algorithm:

1. Start.
2. Read initial value of x and y, say x0, y0.
3. Read the value at which functional value is required say xp.
4. Read step size, say h.
5. Set x=x0, y=y0.
6. Approximate value of y as below.

```
for x=x0 to xp
    k1 = f(x, y);
    k2 = f(x+h/2, h/2*k1+y);
    k3 = f(x+h/2, h/2*k2+y);
    k4 = f(x+h, y+k3);

    y = y + h/6 * (k1+2*k2+2*k3+k4);
End for
```

7. Display functional value y
8. End (Stop)

Program to implement the fourth order Runge-Kutta Method. (Source code in C)

```
#include<stdio.h>
int main()
    {
double x0, y0, h, k1, k2, k3, k4;
int steps;
printf("Enter initial x and y as x0 and y0:");
scanf("%lf%lf", &x0, &y0);
printf("Enter step size (h):");
scanf("%lf", &h);
```

```
printf("Enter number of steps: ");
scanf("%d", &steps);
int i;
for (i=1; i<=steps; i++) {
 k1 = x0+y0;                          //dy/dx = x0+y0
 k2 = (x0+h/2)+(y0+(h/2)*k1);
 k3 = (x0+h/2)+(y0+(h/2)*k2);
 k4 = (x0+h)+(y0+h*k3);

 y0 = y0+(h/6)*(k1+2*k2+2*k3+k4);
 x0 = x0+h;

 printf("%d \t %.4f \t %.6f \n", i, x0, y0);
   }

return 0;

}
```

Output

```
Enter intial x and y as x0 and y0:
0       1
Enter step size(h):  0.1
Enter number of steps : 5
1     0.1000     1.11034
2     0.2000     1.242805
3     0.3000     1.399317
4     0.4000     1.58364
5     0.5000     1.797441
```

Conclusion: In this lab, The Runge fourth order Method was implemented successfully to solve first order ODEs $\frac{dy}{dx} = x+y$ at $x=0.5$ with the value of y was computed as 1.797441.

22. WAP to implement Horner's Method for evaluating polynomials efficiently;

Theory:

Horners method is a method which is used to evaluate the polynomials equation. This method split the polynomial into its individual term and solves them incrementally.

eg:-

given $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = P(x)$

Horner rule makes above polynomial as

$P(x) = ((((a_4 * x + a_3)x + a_2)x + a_1)x + a_0$

similarly general formula for n^{th} degree of polynomial equation is coefficients $a_n, a_{n-1} \ldots a_2 a_1 a_0$

$b_n = a_n$

$b_{n-1} = a_{n-1} + b_n x_0$

$b_{n-2} = a_{n-2} + b_{n-1} x_0$

$b_0 = P(x_0)$

b_0 be the value of polynomial at given x_0

Algorithm

Start

Enter degree of polynomial, say n

Enter the value at which polynomial to be evaluated, x

Initially $b_n = a_n$

while $n > 0$

$b_{n-1} = a_{n-1} + b_n * x$

End while

Display the result at b_0 which the value of polynomial at x.

Terminate.

22. WAP to implement Horner's Method for evaluating polynomials efficiently;

Theory:

Horners method is a method which is used to evaluate the polynomials equation. This method split the polynomial into the its individual term and solves them incrementally.

eg:-

given $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = P(x)$

Horner rule makes above polynomial as

$P(x) = ((((a_4)*x + a_3)x + a_2)x + a_1)x + a_0$

similarly general formula for n^{th} degree of polynomial equation is coefficients $a_n, a_{n-1} \ldots a_2, a_1, a_0$

$b_n = a_n$

$b_{n-1} = a_{n-1} + b_n x_0$

$b_{n-2} = a_{n-2} + b_{n-1} x_0$

$b_0 = P(x_0)$

b_0 be the value of polynomial at given x_0

Algorithm

1. Start
2. Enter degree of polynomial, say n
3. Enter the value at which polynomial to be evaluated, x
4. Initially $b_n = a_n$
5. While $n > 0$

   $b_{n-1} = a_{n-1} + b_n * x$
6. End while
7. Display the result of b_0 which the value of polynomial at x.
8. Terminate.

C program for evaluating Polynomial using Horner's Method.

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

float a[10], b[10];
int main()
{
int i, n;
float x;

printf("Enter the degree of polynomial\n");
scanf("%d", &n);

printf("Enter coefficients of dividend polynomial\n");
for(i=n; i>=0; i--)
{
scanf("%f", &a[i]);
}
printf("Enter the value of x at which polynomial
to be evaluated\n");
scanf("%f", &x);
b[n]=a[n];
while (n>0)
{
b[n-1] = a[n-1]+b[n]*x;
n--;
}

printf("Value of polynomial P[%f] = %f", x, b[0]);
return 0;
}
```

Output:-

```
Enter the degree of polynomial
4
Enter the coefficients of dividend polynomial.
2
-1
3
-5
4
Enter the value at which polynomial to be evaluated.
2
```

value of polynomial $p(2.00) = 30.00$.

Conclusion:-

In this lab the Horner's Method was implemented successfully as we have input the polynomial $P(x) = 2x^4 - x^3 + 3x^2 - 5x + 4$ and at $x = 2$ we had computed the value of polynomial $P(2) = 30.0$ using C-program.