

International Islamic University Chittagong

Lab Report

Course Title : Numerical Methods Lab
Autumn 2024
Section : 7BF

Lab Report on Final Lab Codes

Submitted From,

Name : Jannatul Ferdous Maisha

Student's ID : C213232

Section : 7BF

Semester : 7th

Submitted to,

Name : Sanjida sharmin
Lecturer, Dept of CSE,
IIUC

Date of Submission:
16.01.2025

Marks :

Index

01)Matrix Inversion-----	2-6
02)Jacobi & Least Square Method-----	6-10
03)Trapezoidal & simpson's 1/3 rule-----	10-12
04)Euler series and Runge kutta-----	13-15
05)Taylor series method-----	16-18
06)First & Second derivatives using Newton's forward interpolation formula----- -----	18-20
07)Basic Gauss Elimination Method-----	21-22

01)Name of Labwork : Matrix Inversion

1. Introduction:

Matrix inversion is a fundamental concept in numerical methods and linear algebra, widely used in solving systems of linear equations, data analysis, and computational modeling. The inverse of a matrix, denoted as A^{-1} , plays a crucial role in many applications, such as determining unique solutions to linear systems of equations of the form $Ax=b$, where A is a square matrix, x is the solution vector, and b is the constant vector.

In numerical methods, the computation of a matrix inverse is often challenging, especially for large or ill-conditioned matrices, as it requires precise algorithms to maintain stability and accuracy. Practical approaches for matrix inversion include direct methods such as Gaussian elimination, LU decomposition, and adjoint-based calculation, as well as iterative methods like Jacobi and Gauss-Seidel iterations for specific cases. The efficiency and reliability of these methods depend on the properties of the matrix, such as its determinant, condition number, and sparsity.

2. Code:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int s[10][10], s1[10][10], transpose[10][10], adj[10][10];
    double inv[10][10], determinant;
    int r, c, r1, c1;

    cout << "Enter the row number of the first matrix:" << endl;
    cin >> r;
    cout << "Enter the column number of the first matrix:" << endl;
    cin >> c;
    cout << "\n2D Array Input:\n";

    for (int i = 0; i < r; i++)
    {
        for (int j = 0; j < c; j++)
        {
            cin >> s[i][j];
        }
    }

    cout << "\nThe 2-D Array is:\n";
    for (int i = 0; i < r; i++)
    {
        for (int j = 0; j < c; j++)
```

```

    {
        cout << "\t" << s[i][j];
    }
    cout << endl;
}

cout << "Enter the row number of the second matrix:" << endl;
cin >> r1;
cout << "Enter the column number of the second matrix:" << endl;
cin >> c1;
cout << "\n2D Array Input:\n";

for (int k = 0; k < r1; k++)
{
    for (int l = 0; l < c1; l++)
    {
        cin >> s1[k][l];
    }
}
cout << "\nThe 2-D Array is:\n";
for (int k = 0; k < r1; k++)
{
    for (int l = 0; l < c1; l++)
    {
        cout << "\t" << s1[k][l];
    }
    cout << endl;
}

if (r == 3 && c == 3)
{
    // Calculate determinant for 3x3 matrix
    int x = (s[1][1] * s[2][2]) - (s[2][1] * s[1][2]);
    int y = (s[1][0] * s[2][2]) - (s[2][0] * s[1][2]);
    int z = (s[1][0] * s[2][1]) - (s[2][0] * s[1][1]);

    determinant = (s[0][0] * x) - (s[0][1] * y) + (s[0][2] * z);

    cout << "\nThe Determinant of the Matrix = " << determinant;

    if (determinant == 0)
    {
        cout << "\nThe matrix is singular and cannot have an inverse.";
        return 0;
    }
    // Computing transpose of the matrix

```

```

for (int i = 0; i < r; ++i)
    for (int j = 0; j < c; ++j)
    {
        transpose[j][i] = s[i][j];
    }

// Printing the transpose
cout << "\nTranspose of Matrix: " << endl;
for (int i = 0; i < c; ++i)
{
    for (int j = 0; j < r; ++j)
    {
        cout << transpose[i][j] << " ";
    }
    cout << endl;
}

// Compute adjoint
adj[0][0] = transpose[1][1] * transpose[2][2] - transpose[1][2] * transpose[2][1];
adj[0][1] = -(transpose[1][0] * transpose[2][2] - transpose[1][2] * transpose[2][0]);
adj[0][2] = transpose[1][0] * transpose[2][1] - transpose[1][1] * transpose[2][0];
adj[1][0] = -(transpose[0][1] * transpose[2][2] - transpose[0][2] * transpose[2][1]);
adj[1][1] = transpose[0][0] * transpose[2][2] - transpose[0][2] * transpose[2][0];
adj[1][2] = -(transpose[0][0] * transpose[2][1] - transpose[0][1] * transpose[2][0]);
adj[2][0] = transpose[0][1] * transpose[1][2] - transpose[0][2] * transpose[1][1];
adj[2][1] = -(transpose[0][0] * transpose[1][2] - transpose[0][2] * transpose[1][0]);
adj[2][2] = transpose[0][0] * transpose[1][1] - transpose[0][1] * transpose[1][0];

cout << "\nAdjoint of Matrix: " << endl;
for (int i = 0; i < r; ++i)
{
    for (int j = 0; j < c; ++j)
    {
        cout << adj[i][j] << " ";
    }
    cout << endl;
}

// Compute inverse
for (int i = 0; i < c; ++i)
{
    for (int j = 0; j < r; ++j)
    {
        inv[i][j] = (1 / determinant) * (adj[i][j]);
    }
}

cout << "\nInverse of Matrix: " << endl;

```

```

for (int i = 0; i < r; ++i)
{
    for (int j = 0; j < c; ++j)
    {
        cout << inv[i][j] << " ";
    }
    cout << endl;
}
// Multiply s1 with inverse of s
if (r1 == 3 && c1 == 1)
{
    double x1 = s1[0][0] * inv[0][0] + s1[1][0] * inv[0][1] + s1[2][0] * inv[0][2];
    double y1 = s1[0][0] * inv[1][0] + s1[1][0] * inv[1][1] + s1[2][0] * inv[1][2];
    double z1 = s1[0][0] * inv[2][0] + s1[1][0] * inv[2][1] + s1[2][0] * inv[2][2];

    cout << "\nResult of multiplication:" << endl;
    cout << x1 << endl;
    cout << y1 << endl;
    cout << z1 << endl;
}
else
{
    cout << "\nThe second matrix must be 3x1 for multiplication with the inverse of the first
matrix.";
}
}
else
{
    cout << "\nThe first matrix must be 3x3 to calculate its determinant and inverse.";
}
return 0;
}

```

3.Code Explanation:

In this program demonstrates several matrix operations, including calculating the determinant, transpose, adjoint, inverse, and matrix multiplication for specific cases. The user is prompted to input two matrices along with their dimensions. The first matrix must be a 3x3 matrix to compute its determinant and inverse. The program calculates the determinant using the cofactor expansion formula and checks if it is non-zero, as a zero determinant indicates that the matrix is singular and cannot have an inverse. The program then computes the transpose by swapping rows and columns, followed by calculating the adjoint using cofactor values from the transpose matrix. The inverse is determined using the formula $A^{-1} = 1/\text{determinant} \times \text{adjoint}$. If the second matrix is a 3x1 matrix, it is multiplied by the inverse of the first matrix, and the result is displayed. The program includes validations to ensure the correct dimensions for operations and

provides error messages if the conditions are not met. This implementation highlights key concepts of numerical methods and matrix algebra in an interactive and user-friendly way.

4. Input & Output:

Input	
1	3 3
2	3 1 2
3	2 -3 -1
4	1 2 1
5	3 1
6	3
7	-3
8	4

stdout	stderr	compile output	scribble
19			
20			The Determinant of the Matrix = 8
21			Transpose of Matrix:
22			3 2 1
23			1 -3 2
24			2 -1 1
25			
26			Adjoint of Matrix:
27			-1 3 5
28			-3 1 7
29			7 -5 -11
30			
31			Inverse of Matrix:
32			-0.125 0.375 0.625
33			-0.375 0.125 0.875
34			0.875 -0.625 -1.375
35			
36			Result of multiplication:
37			1
38			2
39			-1

Figure 1: Input & Output Figure

02)Name of Labwork : Jacobi & Least Square Method

1.Introduction:

The Jacobi method is an iterative numerical technique for solving systems of linear equations of the form $Ax = b$, where A is a square matrix, x is the solution vector, and b is the constant vector. This method involves breaking the system into individual equations and iteratively updating the values of the solution vector based on the results from the previous iteration. The Jacobi method is particularly useful for large, sparse systems where direct methods, such as Gaussian elimination, may be computationally expensive.

The convergence of the Jacobi method is guaranteed for diagonally dominant matrices, where the absolute value of each diagonal element is greater than the sum of the absolute values of the non-diagonal elements in the corresponding row. In this experiment, the Jacobi method is implemented to solve a system of linear equations, and the results of each iteration are observed to understand the convergence behavior and accuracy of the solution.

2.Code:

Jacobi

```
#include <bits/stdc++.h>
using namespace std;
void jacobi(vector<vector<double>>& A, vector<double>& b, int maxit, vector<double>& x) {
    int n = A.size(); // Determine the size of the system
    vector<double> x_new(n);
    cout << "Iteration Results:" << endl;
    for (int iter = 1; iter <= maxit; iter++) {
        for (int i = 0; i < n; i++) {
            x_new[i] = b[i];
            for (int j = 0; j < n; j++) {
                if (j != i) {
                    x_new[i] -= A[i][j] * x[j];
                }
            }
            x_new[i] /= A[i][i];
        }
        x = x_new; // Update the solution vector
        cout << "Iteration " << iter << ": "; // Display results for the current iteration
        for (int i = 0; i < n; i++) {
            cout << fixed << setprecision(4) << x[i] << " ";
        }
        cout << endl;
    }
}
int main() {
    int n;
    cout << "Enter the number of variables: ";
    cin >> n;
    vector<vector<double>> A(n, vector<double>(n));
    vector<double> b(n);
    vector<double> x(n, 0); // Initialize the solution vector with zeros
    cout << "Enter the coefficient matrix A:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> A[i][j];
        }
    }
}
```



```

    }
    cout << "Enter the constant vector b:" << endl;
    for (int i = 0; i < n; i++) {
        cin >> b[i];
    }
    int maxit;
    cin >> maxit;
    jacobi(A, b, maxit, x);
    return 0;
}

```

Least Square:

```

#include<bits/stdc++.h>
using namespace std;

int main()
{
    int n,i;
    float x[50], y[50], m, c;

    cout<<"Enter the total points " << endl;
    cin >> n;

    for(i=1; i<=n; i++)
    {
        cout<<"Co-ordinates : "<<i<<" : ";
        cin>>x[i];
        cin>>y[i];
    }
    float sumX=0, sumX2=0, sumY=0, sumXY=0;
    for(i=1; i<=n; i++)
    {
        sumX += x[i];
        sumX2 += ( x[i] * x[i] );
        sumY += y[i];
        sumXY += ( x[i]*y[i] );
    }
    cout<<"sumX = "<<sumX<<endl;
    cout<<"sumX2 = "<<sumX2<<endl;
    cout<<"sumY = "<<sumY<<endl;
    cout<<"sumXY = "<<sumXY<<endl;
    m = (n * sumXY - sumX * sumY) / (n * sumX2 - sumX * sumX);
    c = ((sumY*sumX2)-(sumX*sumXY))/((n*sumX2)-(sumX * sumX));
    cout << "Values are: c = " << c << " and m = " << m;
}

```

```
return(0);
}
```

3.Explanation:

This C++ program solves a system of linear equations using the **Jacobi iterative method**, a numerical technique. The user inputs the coefficient matrix A, the constant vector b, and the number of variables (n). An initial guess for the solution vector (x) is initialized as zero.

The number of iterations is determined by the user. At each iteration, the program calculates the updated values of x using the current values of x from the previous iteration. The results of each iteration are displayed for better understanding. This method is suitable for diagonally dominant systems, ensuring convergence.

4.Input & Output:

Input	
1	3
2	5 2 1
3	1 4 2
4	1 2 5
5	12
6	15
7	20
8	7
stdout	stderr compile output scribble
1	Enter the number of variables: Enter the coefficient matrix A:
2	Enter the constant vector b:
3	Iteration Results:
4	Iteration 1: 2.4000 3.7500 4.0000
5	Iteration 2: 0.1000 1.1500 2.0200
6	Iteration 3: 1.5360 2.7150 3.5200
7	Iteration 4: 0.6100 1.6060 2.6068
8	Iteration 5: 1.2362 2.2941 3.2356
9	Iteration 6: 0.8352 1.8231 2.8351
10	Iteration 7: 1.1037 2.1236 3.1037

Figure 2: Input & Output of Jacobi

Input	
1	4
2	0 -1
3	2 5
4	5 12
5	7 20
stdout	stderr compile output scribble
1	Enter the total points
2	Co-ordinates : 1 : Co-ordinates : 2 : Co-ordinates : 3 : Co-ordinates : 4 : sumX = 14
3	sumX2 = 78
4	sumY = 36
5	sumXY = 210
6	Values are: c = -1.13793 and m = 2.89655

Figure 3: Input & Output of Least Square Method

03)Name of Labwork : Trapezoidal & simpson's 1/3 rule

1.Introduction:

The Trapezoidal Rule is a numerical integration technique used to approximate the definite integral of a function over a given range. It is particularly useful when the function is complex or its analytical integration is difficult or impossible. The method works by dividing the integration range into smaller intervals and approximating the area under the curve as a series of trapezoids. The total area of these trapezoids provides an estimate of the integral.

This experiment focuses on implementing the Trapezoidal Rule for numerical integration. By dividing the interval into equal subintervals, the function values at the endpoints of each interval are used to calculate the area under the curve. The accuracy of the result is influenced by the number of intervals, with smaller intervals yielding better approximations. This study aims to evaluate the effectiveness and precision of the Trapezoidal Rule in numerical computations.

2.Code:

Trapezoidal:

```
#include<bits/stdc++.h>
#define f(x) x/(1+x)
using namespace std;
int main()
{
    float l, u, k;
    int i, n;
    cout<<"Lower limit of integration: ";
    cin>>l;
    cout<<"Upper limit of integration: ";
    cin>>u;
```

```

cout<<"Intervals: ";
cin>>n;
float h = (u - l)/n;
float integration = f(l) + f(u);
for(i=1; i<= n-1; i++)
{
    k = l + i*h;
    integration = integration + 2 * (f(k));
    cout<<"value of x:"<<k<<" "<<"value of Y:"<<f(k)<<endl;
}
integration = integration * h/2;
cout<<endl;
cout<<fixed<<setprecision(3)<<"Final Inegral Value"<<integration<<endl;
return 0;
}

```

Simpson's 1/3:

```

#include<bits/stdc++.h>
#define f(x) 1/(1+x*x)
using namespace std;
int main()
{
    float l, u;
    int n;
    cout<<"lower limit of integration: ";
    cin>>l;
    cout<<"upper limit of integration: ";
    cin>>u;
    cout<<"number of intervals: ";
    cin>>n;
    float h = (u - l)/n;
    float integration = f(l) + f(u);
    int i;
    float j;
    for(i= 1; i<=n-1; i++)
    {
        j = l+ i*h;

        if(i%2==0)
        {
            integration = integration + 2 * (f(j));
        }
        else
        {
            integration = integration + 4 * (f(j));
        }
    }
    cout<<"value of x:"<<j<<" "<<"value of Y:"<<f(j)<<endl;
}

```

```

integration = integration * h/3;
cout<< endl <<"Required value of integration is: "<< integration;
return 0;
}

```

3.Explanation:

The two provided programs numerically calculate the definite integral of a given function using **Trapezoidal Rule** and **Simpson's 1/3 Rule**, respectively.

The **Trapezoidal Rule** program takes the lower and upper limits of integration, divides the range into equal intervals, and calculates the area under the curve by approximating it as a series of trapezoids. The function values at each interval are evaluated, and the sum of these values (adjusted by weights) is used to compute the integral.

The **Simpson's 1/3 Rule** program, on the other hand, uses a more accurate method by dividing the range into intervals and approximating the curve with parabolic segments. Depending on the position of the interval (even or odd index), the function values are given different weights (4 for odd and 2 for even). The weighted sum is then used to compute the integral.

Both programs display intermediate values of x and $f(x)$ during the calculation, providing a step-by-step overview of the numerical integration process.

4.Input & Output:

```

Input
1 0
2 1
3 6
Saved

stdout stderr compile output scribble
1 Lower limit of integration: Upper limit of integration: Intervals: value of x:0.166667 value of Y:0.142857
2 value of x:0.333333 value of Y:0.25
3 value of x:0.5 value of Y:0.333333
4 value of x:0.666667 value of Y:0.4
5 value of x:0.833333 value of Y:0.454545
6
7 Final Inegral Value0.305

```

Figure 4: Input & Output of Trapezoidal Method

```

Input
1 0
2 1
3 6
Saved

stdout stderr compile output scribble
1 lower limit of integration: upper limit of integration: number of intervals: value of x:0.166667 value of Y:0.9
2 value of x:0.333333 value of Y:0.9
3 value of x:0.5 value of Y:0.8
4 value of x:0.666667 value of Y:0.692308
5 value of x:0.833333 value of Y:0.590164
6
7 Required value of integration is: 0.785398

```

Figure 5: Input & Output of Simpson's 1/3 Method

04)Name of Labwork : Euler's method and Runge-Kutta method.

1.Introduction:

The Trapezoidal Rule is a fundamental numerical method for approximating the definite integral of a function over a given interval. By dividing the integration range into smaller subintervals, it approximates the area under the curve as trapezoids, summing their areas to estimate the integral. This method is especially useful for functions that are difficult to integrate analytically or when only discrete data points are available.

In this experiment, the Trapezoidal Rule is applied to numerically evaluate definite integrals. The accuracy of the method is influenced by the number of subintervals, with smaller intervals yielding more precise results. By implementing the Trapezoidal Rule, the objective is to understand its application in numerical integration and to evaluate its accuracy and efficiency in approximating the area under a curve. This technique is an essential tool in numerical analysis and has wide applications in engineering, physics, and computational mathematics.

2.Code:

Euler's method:

```
#include <bits/stdc++.h>
using namespace std;
float func(float x, float y)
{
    return 1-y;
}
int main()
{
    float x0, y, h, x;
    cout<<"enter the initial value for x0 : ";
    cin>>x0; //0
    cout<<"enter the initial value for y0 : ";
    cin>>y; //0
    cout<<"enter the value for h : ";
    cin>>h; //0.1
    cout<<"enter the approximation value for x : ";
    cin>>x; //0.3
    while (x0 < x) {
        y = y + h * func(x0,y);
        x0 = x0 + h;
        cout << "Approximate solution at x = " << x0 << " is " << y << endl;
    }
    return 0;
}
```

Runge-Kutta:

```

#include<bits/stdc++.h>
using namespace std;

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

int main()
{
    float x, y0, h ;
    cout<<"enter the initial value of x"<<endl;
    cin>>x;//0
    cout<<"enter the initial value of y"<<endl;
    cin>>y0;//0
    cout<<"enter the value of h"<<endl;
    cin>>h;//0.2
    float xn;
    cout<<"enter the last value of x"<<endl;
    cin>>xn;//0.4
    int n = (int)((xn - x) / h);
    cout<<endl<<"Result"<<endl<<endl;
    float y=y0;
    float m1,m2,m3,m4;
    for (int i=1; i<=n; i++)
    {
        m1 = f(x, y);
        cout<<"m1 ="<<m1<<endl;
        m2 = f(x + 0.5*h, y + 0.5*m1);
        cout<<"m2 ="<<m2<<endl;
        m3 = f(x + 0.5*h, y + 0.5*m2*h);
        cout<<"m3 ="<<m3<<endl;
        m4 = f(x + h, y + m3*h);
        cout<<"m4 ="<<m4<<endl;
        y = y + (1.0/6.0)*(m1 + 2*m2 + 2*m3 + m4)*h;
        x = x + h;
        cout<<"\nValue of y at x = "<<x<<" is " <<y<<endl;
    }
    return 0;
}

```

3.Explanation:

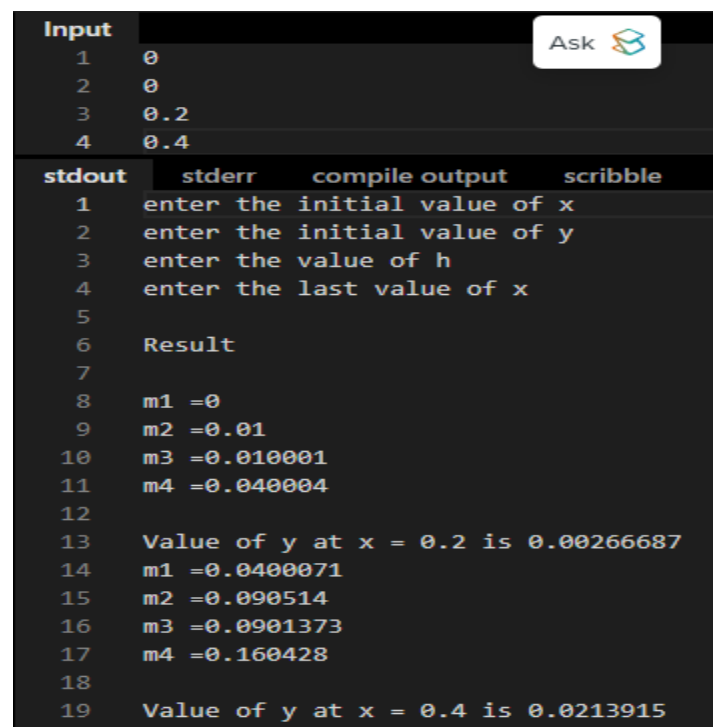
The first program implements **Euler's Method** to solve a first-order differential equation numerically. Starting from an initial condition (x_0, y_0) , it calculates the approximate solution for y at subsequent values of x by iteratively updating y using a small step size h . The program

repeatedly evaluates the function at the current point and updates y until the target x is reached, printing the solution at each step.

The second program uses the **Runge-Kutta Method (4th order)**, a more accurate numerical technique for solving first-order differential equations. It calculates intermediate slopes (m_1, m_2, m_3, m_4) at different points within each interval and uses their weighted average to update y . This method provides better accuracy compared to Euler's Method for the same step size h . The program outputs the intermediate slopes and the computed value of y at each iteration, making it suitable for approximating solutions of complex differential equations.

4.Input & Output:

Runge-Kutta:



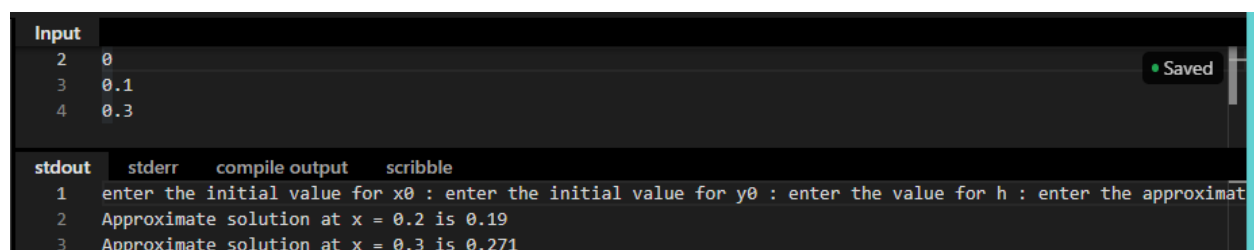
```

Input
1 0
2 0
3 0.2
4 0.4

stdout stderr compile output scribble
1 enter the initial value of x
2 enter the initial value of y
3 enter the value of h
4 enter the last value of x
5
6 Result
7
8 m1 =0
9 m2 =0.01
10 m3 =0.010001
11 m4 =0.040004
12
13 Value of y at x = 0.2 is 0.00266687
14 m1 =0.0400071
15 m2 =0.090514
16 m3 =0.0901373
17 m4 =0.160428
18
19 Value of y at x = 0.4 is 0.0213915
  
```

Figure 6: Input & Output of Runge Kutta Method

Euler's method:



```

Input
2 0
3 0.1
4 0.3

stdout stderr compile output scribble
1 enter the initial value for x0 : enter the initial value for y0 : enter the value for h : enter the approximat
2 Approximate solution at x = 0.2 is 0.19
3 Approximate solution at x = 0.3 is 0.271
  
```

Figure 7: Input & Output of Euler's Method

5)Name of Labwork : Taylor series method

1.Introduction:

The Taylor Series Method is a powerful numerical technique used to approximate the solution of first-order differential equations. By expanding the solution as a series, this method incorporates higher-order derivatives to improve accuracy, making it suitable for problems where precision is critical. The Taylor series provides a way to approximate the value of a function at a point by using its value and derivatives at a nearby point.

This experiment focuses on implementing the Taylor Series Method to solve a first-order differential equation. The method is applied iteratively, where the solution at each step is computed using the function and its derivatives up to a specified order. The step size and range of integration are chosen to balance accuracy and computational effort. The objective of this study is to understand the application of the Taylor Series Method in numerical solutions and evaluate its effectiveness for solving differential equations.

2.Code:

```
#include <bits/stdc++.h>
using namespace std;
double f(double x, double y) {
    return x + y;
}
double f1(double x, double y) {
    return 1 + f(x, y);
}
double f2(double x, double y) {
    return f1(x, y);
}
double f3(double x, double y) {
    return 0;
}
int main() {
    double x0, y0, h, x_target;
    cout << "Enter initial value of x (x0): ";
    cin >> x0;
    cout << "Enter initial value of y (y0): ";
    cin >> y0;
    cout << "Enter step size (h): ";
    cin >> h;
    cout << "Enter the target x value: ";
    cin >> x_target;
    int n = (x_target - x0) / h;
    cout<<"value of n"<<n<<endl;
    cout << "\nInitial values:\n";
    cout << "x0 = " << fixed << setprecision(6) << x0 << ", y0 = " << y0 << endl;
    cout << "Step size h = " << h << endl;
```

```

double k1_step1 = f(x0, y0);
double k2_step1 = f1(x0, y0);
double k3_step1 = f2(x0, y0);
double k4_step1 = f3(x0, y0);
cout << "\nStep 1 (Current x = " << x0 << ", y = " << y0 << "):\n";
cout << " f(x, y) = " << k1_step1 << " (dy/dx)\n";
cout << " f'(x, y) = " << k2_step1 << " (d^2y/dx^2)\n";
cout << " f''(x, y) = " << k3_step1 << " (d^3y/dx^3)\n";
cout << " f'''(x, y) = " << k4_step1 << " (d^4y/dx^4)\n";
y0 += h * k1_step1 + (h * h / 2) * k2_step1 + (h * h * h / 6) * k3_step1 + (h * h * h * h / 24) * k4_step1;
x0 += h;
cout << " New y = " << fixed << setprecision(6) << y0 << " (after Taylor expansion)\n";
double k1_step2 = f(x0, y0);
double k2_step2 = f1(x0, y0);
double k3_step2 = f2(x0, y0);
double k4_step2 = f3(x0, y0);
cout << "\nStep 2 (Current x = " << x0 << ", y = " << y0 << "):\n";
cout << " f(x, y) = " << k1_step2 << " (dy/dx)\n";
cout << " f'(x, y) = " << k2_step2 << " (d^2y/dx^2)\n";
cout << " f''(x, y) = " << k3_step2 << " (d^3y/dx^3)\n";
cout << " f'''(x, y) = " << k4_step2 << " (d^4y/dx^4)\n";
y0 += h * k1_step2 + (h * h / 2) * k2_step2 + (h * h * h / 6) * k3_step2 + (h * h * h * h / 24) * k4_step2;
x0 += h;
cout << " New y = " << fixed << setprecision(6) << y0
    << " (after Taylor expansion)\n";
cout << "\nFinal value: y(" << x_target << ") = " << fixed << setprecision(6) << y0 << endl;
return 0;
}

```

3.Explanation:

The **Taylor series method** to solve a first-order differential equation numerically. It computes the value of $y(x)$ for a given target x , starting from an initial condition (x_0, y_0) . The program calculates the first four derivatives ($y', y'', y''', y^{(4)}$) at each step using defined functions. The Taylor series expansion formula is used iteratively to compute the new value of y at each step, updating x and y accordingly. The user inputs the initial values (x_0, y_0) , step size (h) , and the target x -value. The program outputs the intermediate values of x, y , and derivatives at each step, along with the final computed value of y . This method provides an accurate numerical approximation of the solution to the differential equation.

4.Input & Output:

```

Input
1  1
2  0
3  0.1
4  1.2

stdout  stderr  compile output  scribble
1  Enter initial value of x (x0): Enter initial value of y (y0)
2
3  Initial values:
4  x0 = 1.000000, y0 = 0.000000
5  Step size h = 0.100000
6
7  Step 1 (Current x = 1.000000, y = 0.000000):
8      f(x, y) = 1.000000 (dy/dx)
9      f'(x, y) = 2.000000 (d²y/dx²)
10     f''(x, y) = 2.000000 (d³y/dx³)
11     f'''(x, y) = 0.000000 (d⁴y/dx⁴)
12     New y = 0.110333 (after Taylor expansion)
13
14  Step 2 (Current x = 1.100000, y = 0.110333):
15     f(x, y) = 1.210333 (dy/dx)
16     f'(x, y) = 2.210333 (d²y/dx²)
17     f''(x, y) = 2.210333 (d³y/dx³)
18     f'''(x, y) = 0.000000 (d⁴y/dx⁴)
19     New y = 0.242787 (after Taylor expansion)
20
21  Final value: y(1.200000) = 0.242787

```

Figure 8: Input & Output of Taylor Series

(6) **Name of Labwork** : First & Second derivatives using Newton's forward interpolation formula

1.Introduction:

Numerical methods for calculating derivatives are essential in situations where analytical differentiation is impractical or when dealing with discrete data points. The **Newton Forward Difference Method** is a commonly used technique for approximating derivatives based on tabulated data. This method relies on constructing a forward difference table and utilizing difference coefficients to approximate the first and second derivatives of a function.

This experiment focuses on implementing the Newton Forward Difference Method to calculate the first and second derivatives of a function at a specific point. The step size (h) and normalized

value (u) are used to compute the derivatives with the help of forward difference formulas. The objective of this study is to understand the application of numerical differentiation and evaluate the accuracy and efficiency of the forward difference method in approximating derivatives.

2.Code:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    cout << "Total number of data: " << endl;
    cin >> n;
    double x[20], y[20][20];
    int i,j;
    for(i = 0; i < n ; i++)
    {
        cout << "x[" << i << "] = ";
        cin >> x[i];
        cout << "y[" << i << "] = ";
        cin >> y[i][0];
    }
    for(i = 1; i < n; i++)
    {
        for(j = 0; j < n-i; j++)
        {
            y[j][i] = y[j+1][i-1] - y[j][i-1];
        }
    }
    cout << endl;
    for(i = 0; i < n; i++)
    {
        cout << x[i];
        for(j = 0; j < n-i ; j++)
        {
            cout << "\t" << y[i][j];
        }
        cout << endl;
    }
    double d1, d2,s,u;
    cout<<"enter the value of x: ";
    cin>>s;
    double h = x[1] - x[0];
    cout<<"h = "<<h<<endl;
    u =( s - x[0] )/h;
```

```

cout<<"u = "<<u<<endl;
d1 = (1/h)*(y[0][1]+(((2*u -1)*y[0][2])/2.0)+(( (3*u*u + 6*u +2)*y[0][3])/(3.0*2.0)));
cout<<d1<<endl;
d2= (1/(h*h))*(y[0][2]+((u-1)*y[0][3])+(((6*u*u-18*u+11)*y[0][4])/12.0));
cout<<d2<<endl;
return 0;
}

```

3.Explanation:

This C++ program calculates the first and second derivatives of a function at a given point using the **Newton Forward Difference Method**. It takes as input the x and y values of a dataset, constructs the forward difference table, and uses it to compute the derivatives. The user provides the value of x at which the derivatives are to be approximated. The program calculates the step size (h) and a normalized value (u) to use in the difference formulas. The first and second derivatives are then calculated using forward difference coefficients, which involve the differences from the constructed table. Finally, the program outputs the computed values of the first and second derivatives.

4.Input & Output:

Input	
1	6
2	0 0
3	0.2 0.12
4	0.4 0.48
5	0.6 1.10
6	0.8 2.0
7	1.0 3.20
8	0.7

stdout	stderr	compile output	scribble
1		Total number of data:	
2		x[0] = y[0] = x[1] = y[1] = x[2] = y[2] = x[3] = y[3] = x[4] = y[4] = x[5] = y[5] =	
3		0 0 0.12 0.24 0.02 -4.44089e-16 1.22125e-15	
4		0.2 0.12 0.36 0.26 0.02 7.77156e-16	
5		0.4 0.48 0.62 0.28 0.02	
6		0.6 1.1 0.9 0.3	
7		0.8 2 1.2	
8		1 3.2	
9		enter the value of x: h = 0.2	
10		u = 3.5	
11		5.19583	
12		7.25	

Figure 9: Input & Output of Derivatives using Newton's Forward Interpolation

(7)Name of Labwork : Basic Gauss Elimination Method

1.Introduction:

The **Gauss Elimination Method** is a direct numerical technique used to solve systems of linear equations. It is one of the most fundamental and widely used methods in numerical linear algebra. This method involves two main steps: forward elimination, where the system of equations is transformed into an upper triangular form, and back substitution, where the values of the unknowns are computed starting from the last equation. This experiment focuses on implementing the Gauss Elimination Method to solve a system of linear equations represented in augmented matrix form. By systematically eliminating variables and simplifying the equations, this method provides an efficient and accurate solution.

2.Code:

```
#include<bits/stdc++.h>
#define SIZE 10
using namespace std;
int main()
{
    float a[SIZE][SIZE], x[SIZE], r;
    int i,j,k,n;
    cout<<"Enter number of unknowns: "; //x y z 3ta
    cin>>n; //3
    cout<<"Enter Coefficients of Augmented Matrix: "<< endl;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n+1;j++)
        {
            cout<<"a["<< i<<"]["<< j<<"]=" ";
            cin>>a[i][j]; //constant value soho dite hobe
        }
    }
    for(i=1;i<=n-1;i++) /* Applying Gauss Elimination */ //i=1,2
    {
        if(a[i][i] == 0.0)
        {
            cout<<"Mathematical Error!";
            exit(0);
        }
        for(j=i+1;j<=n;j++)//j=2, 3
        {
            r = a[j][i]/a[i][i]; //a[2][1]/a[1][1]=3/2

            for(k=1;k<=n+1;k++) //k=1,2,3,4
            {
                a[j][k] = a[j][k] - r*a[i][k]; //a[2][1]=a[2][1]-3/2 a[1][1]=3-3/2*2=0
            }
        }
    }
}
```

```

    }
}
x[n] = a[n][n+1]/a[n][n]; /* Obtaining Solution by Back Substitution Method */
//x[3]=a[3][4]/a[3][3]=16 er ghor/9z er ghor

for(i=n-1;i>=1;i--) //i=2,1
{
    x[i] = a[i][n+1]; //x[2]=a[2][4]=18 ,x[1]=a[1][4]=10
    for(j=i+1;j<=n;j++)//j=3,2
    {
        x[i] = x[i] - a[i][j]*x[j]; //x[2] = x[2]- a[2][3]* x[3]
    }
    x[i] = x[i]/a[i][i];
}
cout<< endl<<"Solution: "<< endl;
for(i=1;i<=n;i++)
{
    cout<<"x["<<i<<"] = "<<x[i]<< endl;
}
return(0);
}

```

3.Explanation:

This C++ program implements the **Gauss Elimination Method** to solve a system of linear equations. It first takes the coefficients of the augmented matrix (including constants) as input. The program then performs forward elimination to convert the system into an upper triangular matrix by eliminating lower triangular elements. After this, the back substitution method is applied to calculate the values of the unknowns starting from the last equation. This iterative process uses the values of previously solved variables to compute the current one. Finally, the solution for all unknowns is displayed.

4.Input & Output:

Input	stdout	stderr	compile output	scribble
1 3	1 Enter number of unknowns: Enter Coefficients of Augmented Matrix:			
2 2 1 1 10	2 a[1][1]= a[1][2]= a[1][3]= a[1][4]= a[2][1]= a[2][2]= a[2][3]= a[2][4]= a[3][1]= a[3][2]= a[3][3]= a[3][4]=			
3 3 2 3 18	3 Solution:			
4 1 4 9 16	4 x[1] = 7			
5	5 x[2] = -9			
	6 x[3] = 5			
	7			

Figure 10: Input & Output of Gauss Elimination