RAJSHAHI UNIVERSITY OF ENGINEERING AND TECHNOLOGY

LABORATORY REPORTS

Numerical Methods Experiments

Submitted to:
Shyla Afroge
Assistant Professor, Department of CSE
Rajshahi University of Engineering And Technology

Submitted by:
Md Naim Parvez
Section: A
Roll : 2003015
Department of CSE
Rajshahi University of Engineering And Technology

Lab- 1

Name of the Experiment

Root Finding Methods - Bisection and False Position

Theoretical Background

Bisection Method

The Bisection method is based on the Intermediate Value Theorem, which states that if a continuous function $f(x)$ changes sign over an interval $[a, b]$, then there exists at least one root in that interval. The algorithm iteratively narrows down the interval by calculating the midpoint x_{mid} and checking for a sign change. The process continues until the interval becomes sufficiently small.

Formula:

$$x_{\text{mid}} = \frac{a+b}{2}$$

False Position Method

The False Position method, also known as the Linear Interpolation method, utilizes linear interpolation between the function values at the interval endpoints. It estimates the root by finding the x-intercept of the linear segment connecting $(a, f(a))$ and $(b, f(b))$.

Formula:

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

Source Code

```
#include <iostream>
#include <cmath>

using namespace std;

double f(double x) {
    return x * x * x - 2 * x - 5;
}

double false_position(double a, double b, double tol, int max_iterations) {
    double x, error;

    for (int n = 1; n <= max_iterations; ++n) {
        x = (a * f(b) - b * f(a)) / (f(b) - f(a));
        error = fabs(x - a);

        cout << n << "\t" << a << "\t" << b << "\t" << x << "\t" << f(x) << "\t" << error
    << endl;

        if (error <= tol)
            return x;

        if (f(x) * f(a) < 0)
            b = x;
        else
            a = x;
    }

    return -1.0; // Return -1 to indicate failure to converge
}

int main() {
    double a, b, tol;
    int max_iterations = 1000; // Maximum number of iterations for both methods
```

```
    cout << "Enter a: ";
    cin >> a;
    cout << "Enter b: ";
    cin >> b;
    cout << "Enter the tolerance (e.g., 0.0002): ";
    cin >> tol;

    if (f(a) * f(b) >= 0) {
        cout << "The chosen bounds do not have opposite signs.
         Bisection and false position methods may not converge." << endl;
        return 1;
    }

    int choice;
    cout << "Choose a method:" << endl;
    cout << "1. Bisection" << endl;
    cout << "2. False Position" << endl;
    cin >> choice;

    if (choice == 1) {
        cout << "n\t  a\t\t  b\t\t  x\t\t  f(x)\t\t  Error" << endl;
        cout << "--------------------------------------------------------" << endl;

        // Bisection method
        double x_bisection, error_bisection;
        for (int n = 1; n <= max_iterations; ++n) {
            x_bisection = (a + b) / 2.0;
            error_bisection = fabs(b - a) / 2.0;

            cout << n << "\t" << a << "\t" << b << "\t" << x_bisection << "\t"
            << f(x_bisection) << "\t" << error_bisection << endl;

            if (error_bisection <= tol)
                break;

            if (f(x_bisection) * f(a) < 0)
                b = x_bisection;
            else
                a = x_bisection;
        }

        cout << "--------------------------------------------------------" << endl;
        cout << "Approximate root using bisection: " << x_bisection << endl;
    } else if (choice == 2) {
        cout << "n\t  a\t\t  b\t\t  x\t\t  f(x)\t\t  Error" << endl;
        cout << "--------------------------------------------------------" << endl;

        // False position method
        double x_false_position = false_position(a, b, tol, max_iterations);

        if (x_false_position != -1.0) {
            cout << "Approximate root using false position: " << x_false_position << endl;
        } else {
            cout << "False position method did not converge within
             the maximum number of iterations." << endl;
        }
    } else {
        cout << "Invalid choice." << endl;
    }

    return 0;
}
```

Output (Screenshot)

```
Enter degree of the polynomial: 3
Enter interval [a, b]: 2 3
Enter coefficients from highest to lowest degree: 1 0 -2 -5
Bisection Method:
 Step           a           b           x        f(x)       Error
    1           2           3         2.5       5.625         2.5
    2           2         2.5        2.25     1.89062        0.25
    3           2        2.25       2.125    0.345703       0.125
    4           2       2.125      2.0625   -0.351318      0.0625
    5      2.0625       2.125     2.09375 -0.00894165     0.03125
    6     2.09375       2.125     2.10938    0.166836    0.015625
    7     2.09375     2.10938     2.10156   0.0785623   0.0078125
    8     2.09375     2.10156     2.09766   0.0347143  0.00390625
    9     2.09375     2.09766      2.0957   0.0128623  0.00195312
   10     2.09375      2.0957     2.09473  0.00195435 0.000976562
   11     2.09375     2.09473     2.09424 -0.00349515 0.000488281
   12     2.09424     2.09473     2.09448-0.000770775 0.000244141
   13     2.09448     2.09473      2.0946 0.000591693  0.00012207
   14     2.09448      2.0946     2.09454-8.95647e-0056.10352e-005
Root found: 2.0945
False Position Method:
 Step           a           b           x        f(x)       Error
    1           2           3      2.0588     -0.3908    0.035781
    2      2.0588           3      2.0813     -0.1472     0.02244
    3      2.0813           3      2.0896   -0.054677   0.0083756
    4      2.0896           3      2.0927   -0.020203   0.0031004
    5      2.0927           3      2.0939  -0.0074505   0.0011441
    6      2.0939           3      2.0943  -0.0027457  0.00042174
    7      2.0943           3      2.0945  -0.0010116  0.00015539
    8      2.0945           3      2.0945 -0.00037265 5.7248e-005
Root found: 2.0945
```

Discussion

The Bisection method converges slowly but is guaranteed to converge for continuous functions. The False Position method, while potentially faster, may suffer from convergence issues if the chosen interval is not well-behaved. Analyze the impact of the initial interval and tolerance on convergence. Discuss situations where each method excels or struggles.

Lab- 2

Name of the Experiment

Newton's Interpolation Methods - Forward and Backward

Theoretical Background

Newton's Forward Interpolation

Newton's Forward Interpolation constructs an interpolating polynomial using forward divided differences. The method involves computing the differences of function values at consecutive data points and using them to build up the polynomial.

Formula:

$$P_n(x) = f(x_0) + (x - x_0)\frac{\Delta f(x_0)}{h} + \frac{(x - x_0)(x - x_1)}{2!}\frac{\Delta^2 f(x_0)}{h^2} + \dots$$

Newton's Backward Interpolation

Newton's Backward Interpolation, similar to forward interpolation, builds the interpolating polynomial using backward divided differences.

Formula:

$$P_n(x) = f(x_n) + (x - x_n)\frac{\Delta f(x_n)}{h} + \frac{(x - x_n)(x - x_{n-1})}{2!}\frac{\Delta^2 f(x_n)}{h^2} + \dots$$

Source Code

```cpp
#include <iostream>
#include <vector>

using namespace std;

double newtonForwardInterpolation(double x,const vector<double> &xValues,
const vector<double> &yValues)
{
    int n = xValues.size();
    double result = yValues[0];
    double h = xValues[1] - xValues[0];
    double u = (x - xValues[0]) / h;

    vector<vector<double>> dividedDifferences(n, vector<double>(n));

    for (int i = 0; i < n; i++)
    {
        dividedDifferences[i][0] = yValues[i];
    }

    for (int i = 1; i < n; i++)
    {
        for (int j = 0; j < n - i; j++)
        {
            dividedDifferences[j][i] = dividedDifferences[j + 1][i - 1]
             - dividedDifferences[j][i - 1];
        }
    }
    cout<<"Divided Differences Table for Forward method: "<<endl;
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            cout<<dividedDifferences[i][j]<<" ";
        }
        cout<<endl;
    }

```

```
    for (int i = 1; i < n; i++)
    {
        double term = dividedDifferences[0][i];
        for (int j = 0; j < i; j++)
        {
            term *= (u - j) / (j + 1);
        }
        result += term;
    }

    return result;
}

double newtonBackwardInterpolation(double x, const vector<double> &xValues,
const vector<double> &yValues)
{
    int n = xValues.size();
    double result = yValues[n - 1];
    double h = xValues[1] - xValues[0];
    double u = (x - xValues[n - 1]) / h;

    vector<vector<double>> dividedDifferences(n, vector<double>(n));

    for (int i = 0; i < n; i++)
    {
        dividedDifferences[i][0] = yValues[i];
    }

    for (int i = 1; i < n; i++)
    {
        for (int j = n - 1; j >= i; j--)
        {
            dividedDifferences[j][i] = dividedDifferences[j][i - 1]
            - dividedDifferences[j - 1][i - 1];
        }
    }
    cout<<"Divided Differences Table for backward method: "<<endl;
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            cout<<dividedDifferences[i][j]<<" ";
        }
        cout<<endl;
    }

    for (int i = 1; i < n; i++)
    {
        double term = dividedDifferences[n - 1][i];
        for (int j = 0; j < i; j++)
        {
            term *= (u + j) / (j + 1);
        }
        result += term;
    }

    return result;
}

int main()
{
    vector<double> xValues;
    vector<double> yValues;
    double x;

```

```
    cout << "Enter the number of data points: ";
    int n;
    cin >> n;

    cout << "Enter the x-values: ";
    for (int i = 0; i < n; i++)
    {
        double value;
        cin >> value;
        xValues.push_back(value);
    }

    cout << "Enter the y-values: ";
    for (int i = 0; i < n; i++)
    {
        double value;
        cin >> value;
        yValues.push_back(value);
    }

    cout << "Enter the value of x: ";
    cin >> x;

    int choice;
    do
    {
        cout << "Menu:" << endl;
        cout << "1. Forward Interpolation" << endl;
        cout << "2. Backward Interpolation" << endl;
        cout << "0. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        double backwardInterpolatedValue;
        double forwardInterpolatedValue;
        switch (choice)
        {
        case 1:
            forwardInterpolatedValue = newtonForwardInterpolation(x, xValues, yValues);
            cout << "Forward Interpolated value at x = " << x << " is: "
             << forwardInterpolatedValue << endl;
            break;
        case 2:
             backwardInterpolatedValue = newtonBackwardInterpolation(x, xValues, yValues);
            cout << "Backward Interpolated value at x = " << x << " is: "
            << backwardInterpolatedValue << endl;
            break;
        case 0:
            cout << "Exiting..." << endl;
            break;
        default:
            cout << "Invalid choice. Please try again." << endl;
            break;
        }
    } while (choice != 0);

    return 0;
}
```

Output (Screenshot)

```
Enter the number of data points: 5
Enter the x-values: 0 1 2 3 4
Enter the y-values: 1 2 3 4 5
Enter the value of x: 2
Menu:
1. Forward Interpolation
2. Backward Interpolation
0. Exit
Enter your choice: 1
Divided Differences Table for Forward method:
1 1 0 0 0
2 1 0 0 0
3 1 0 0 0
4 1 0 0 0
5 0 0 0 0
Forward Interpolated value at x = 2 is: 3
Menu:
1. Forward Interpolation
2. Backward Interpolation
0. Exit
Enter your choice: 2
Divided Differences Table for backward method:
1 0 0 0 0
2 1 0 0 0
3 1 0 0 0
4 1 0 0 0
5 1 0 0 0
Backward Interpolated value at x = 2 is: 3
```

Discussion

Discuss the accuracy of Newton's interpolation methods and their suitability for different datasets. Explore how the number of data points affects the accuracy of interpolation. Consider the computational cost of these methods and any observed oscillations in the interpolated results.

Lab- 3

Name of the Experiment

Linear Regression - Data Fitting

Theoretical Background

Linear Regression aims to find the best-fitting linear relationship between a set of independent (x) and dependent (y) variables. The method minimizes the sum of squared differences between the observed and predicted values.

Formula:

$$y = a + bx$$

where:

$$a : \text{Intercept of the linear fit line}$$
$$b : \text{Slope of the linear fit line}$$

To find the parameters a and b, use the following formulas:

$$b = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$
$$a = \frac{\sum y - b(\sum x)}{n}$$

where:

$$n : \text{Number of data points}$$
$$\sum : \text{Summation notation}$$
$$x, y : \text{Data points (independent and dependent variables)}$$

Source Code

```cpp
#include<bits/stdc++.h>
#include<fstream>
#include<iomanip>
#include<cmath>
#define ll long long
#define fr02m(m) for(int i=0; i<m; i++)
#define fr12m(m) for(int i=1; i<m; i++)
#define fr02j(m) for(int j=0; j<m; j++)
#define frn20(n) for(int i=n; i>=0; i--)
#define frxy(x,y) for(int i=x;i<=y;i++)
#define pb push_back
#define pf push_front
using namespace std;
int main()
{
    int i,j,k,n;
    cout<<"\nEnter the no. of data: \n";
    cin>>n;
    vector<double>x(n);
    vector<double>y(n);
    double a,b,c;

    cout<<"\nEnter the x-axis values: \n";
    for (i=0;i<n;i++){
        cin>>x[i];
    }
    cout<<"\nEnter the y-axis values: \n";
```

```cpp
    for (i=0;i<n;i++){
        cin>>y[i];
    }
    double sum_x=0,sum_x2=0,sum_y=0,xysum=0;
    for (i=0;i<n;i++)
    {
        sum_x=sum_x+x[i];
        sum_y=sum_y+y[i];
        sum_x2=sum_x2+pow(x[i],2);
        xysum=xysum+x[i]*y[i];
    }
    a=(n*xysum-sum_x*sum_y)/(n*sum_x2-sum_x*sum_x);
    c= (sum_y/n) - (a*(sum_x/n));
    cout<<"\nThe linear fit line is of the form:\n\n"<<a<<" + x("<<c<<")"<<endl;
    return 0;
}
```

Output (Screenshot)

```
Enter the no. of data:
5

Enter the x-axis values:
1 2 3 4 5

Enter the y-axis values:
.6 2.4 3.5 4.8 5.7

The linear fit line is of the form:

1.26 + x(-0.38)
PS C:\Users\ASUS\OneDrive\Documents\2-2 Labs\Numerical\LAB 3>
```

Discussion

Discuss the obtained linear fit line, including the slope (b) and intercept (a). Examine how well the linear fit represents the given data points. Consider the significance of the linear regression parameters, their interpretation in the context of the specific dataset, and potential limitations of the linear model.

Lab- 4

Name of the Experiment

Numerical Integration Techniques - Trapezoidal, Simpson's 1/3, and 3/8 Rules

Theoretical Background

Trapezoidal Rule

The Trapezoidal Rule approximates the definite integral by dividing the interval into trapezoids and summing their areas. It is derived from the geometric interpretation of the integral.

Formula:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}[f(a) + 2f(x_1) + 2f(x_2) + \ldots + f(b)]$$

Simpson's 1/3 Rule

Simpson's 1/3 Rule uses quadratic interpolating polynomials to approximate the integral. It provides higher accuracy than the Trapezoidal Rule.

Formula:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}[f(a) + 4f(x_1) + 2f(x_2) + \ldots + 4f(x_{n-1}) + f(b)]$$

Simpson's 3/8 Rule

Simpson's 3/8 Rule is an extension of Simpson's 1/3 Rule, using cubic interpolating polynomials. It offers further improvement in accuracy.

Formula:

$$\int_a^b f(x)\,dx \approx \frac{3h}{8}[f(a) + 3f(x_1) + 3f(x_2) + 2f(x_3) + \ldots + 3f(x_{n-2}) + 3f(x_{n-1}) + f(b)]$$

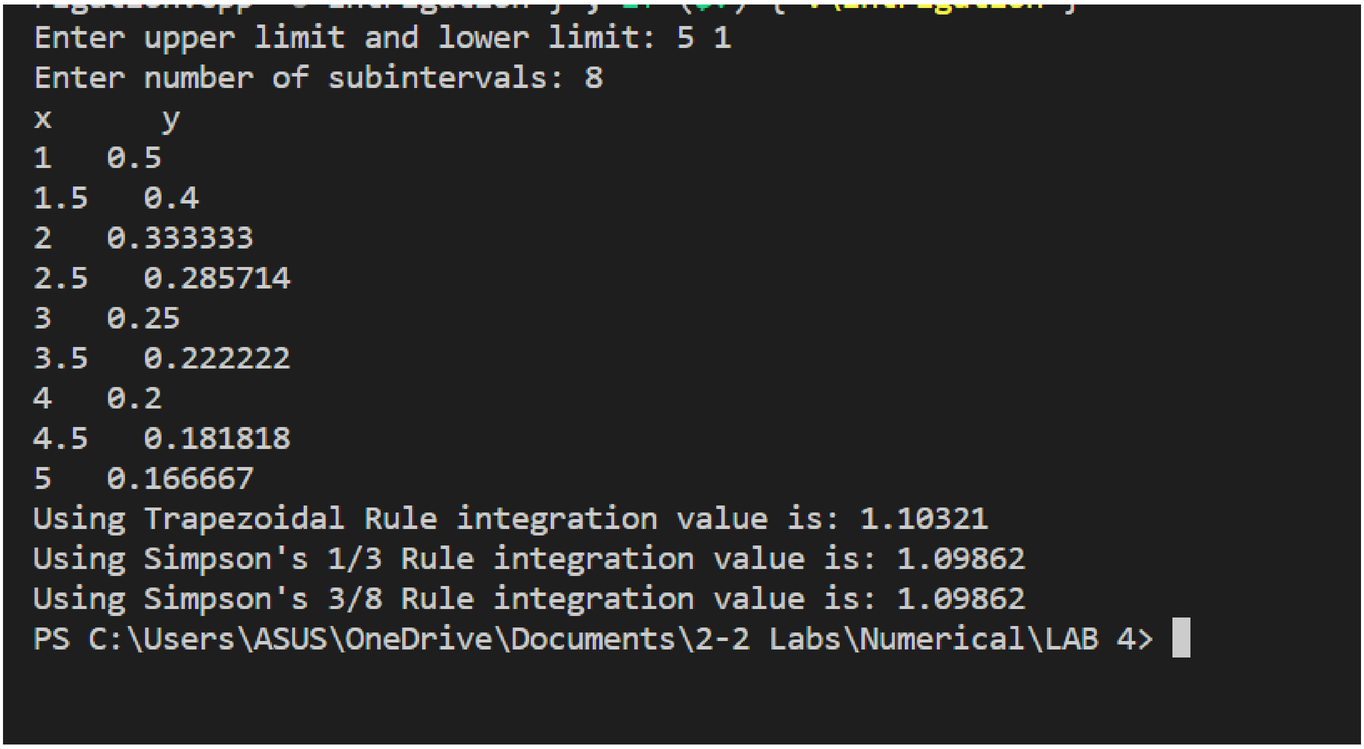
Source Code

```cpp
#include <bits/stdc++.h>
#include <fstream>
#define ll long long
#define fr02m(m) for (int i = 0; i < m; i++)
#define fr12m(m) for (int i = 1; i < m; i++)
#define fr02j(m) for (int j = 0; j < m; j++)
#define frn20(n) for (int i = n; i >= 0; i--)
#define frxy(x, y) for (int i = x; i <= y; i++)
#define pb push_back
#define pf push_front
using namespace std;

float calc(float x)
{
    return (1 / (1 + x));
}
float trapezoidal(float a, float b, int n)
{
    float h = (b - a) / n;
    float sum = 0.0;
    cout<<"x       y"<<endl;
    for (int i = 0; i <= n; i++)
    {
        if (i == 0 || i == n)
        {
            sum += calc(a + i * h);
            cout<<a+i*h<<"    "<<calc(a+i*h)<<endl;
```

```cpp
        }
        else
        {
            sum += 2 * calc(a + i * h);
            cout<<a+i*h<<"    "<<calc(a+i*h)<<endl;
        }
    }
    return (h / 2) * sum;
}

float simpsons_one_third(float a, float b, int n)
{
    float h = (b - a) / n;
    float sum = 0.0;
    for (int i = 0; i <= n; i++)
    {
        if (i == 0 || i == n)
        {
            sum += calc(a + i * h);
        }
        else if (i % 2 != 0)
        {
            sum += 4 * calc(a + i * h);
        }
        else
        {
            sum += 2 * calc(a + i * h);
        }
    }
    return (h / 3) * sum;
}

float simpsons_three_eight(float a, float b, int n)
{
    float h = (b - a) / n;
    float sum = 0.0;
    for (int i = 0; i <= n; i++)
    {
        if (i == 0 || i == n)
        {
            sum += calc(a + i * h);
        }
        else if (i % 3 == 0)
        {
            sum += 2 * calc(a + i * h);
        }
        else
        {
            sum += 3 * calc(a + i * h);
        }
    }
    return (3 * h / 8) * sum;
}

int main()
{
    float a, b;
    int n;
    cout << "enter upper limit and lower limit: ";
    cin >> b >> a;
    cout << "enter number of sub interval: ";
    cin >> n;
    cout << "Using Trapezoidal Rule intrigation value is : " << trapezoidal(a, b, n)<<"
```

```
    Error- "
    << endl;
    cout<<"Using simpson's one_third Rule intrigation value is :"<<simpsons_one_third(a, b
    , n)<<"Error- "  << endl;
    cout<<"Using simpson's three_eight Rule intrigation value is:"<<simpsons_three_eight(a
    , b, n)
    <<"Error- " ;

    return 0;
}
```

Output (Screenshot)



```
Enter upper limit and lower limit: 5 1
Enter number of subintervals: 8
x      y
1   0.5
1.5   0.4
2   0.333333
2.5   0.285714
3   0.25
3.5   0.222222
4   0.2
4.5   0.181818
5   0.166667
Using Trapezoidal Rule integration value is: 1.10321
Using Simpson's 1/3 Rule integration value is: 1.09862
Using Simpson's 3/8 Rule integration value is: 1.09862
PS C:\Users\ASUS\OneDrive\Documents\2-2 Labs\Numerical\LAB 4>
```

Discussion

Discuss the accuracy and efficiency of each numerical integration method. Analyze the impact of the chosen parameters (number of subintervals) on the results. Compare the results obtained from different integration techniques. Consider the convergence properties and potential sources of error in numerical integration. Discuss situations where each method is most suitable.