Index

"Heaven's Light is Our Guide"

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course Code:	CSE 2204
Course Name:	Numerical Methods Sessional
Experiment No:	01
Experiment Name:	Polynomial Root Finder with Bisection and False Position Methods
Date of Experiment:	
Date of Submission:	January 7, 2024

Submitted By:	Submitted To:
Name: Tonmoy	Shyla Afroge
Section: A	Assistant Professor
Roll: 2003027	CSE, RUET
Year: 2nd year, Even Semester	

Title: Polynomial Root Finder with Bisection and False Position Methods

Theoretical Background:

1. Bisection Method: The Bisection method, also known as the binary search method, is a simple and robust numerical technique to find the root of a real-valued function. It is based on the Intermediate Value Theorem, which states that if a continuous function changes sign over an interval, it must have at least one root within that interval. The Bisection method works as follows:

Start with an interval [a, b] where the function changes sign (f(a) * f(b) < 0).

Calculate the midpoint c = (a + b) / 2.

Check the sign of f(c):

If f(c) ≈ 0 (within a predefined tolerance), c is the root.

If f(c) has the same sign as f(a), replace a with c, else replace b with c.

Repeat the process until the interval becomes sufficiently small or the function value is close to zero.

2. False Position Method (Regula Falsi): The False Position method is another numerical technique for finding the root of a real-valued function. It's based on linear interpolation. This method is also known as the linear interpolation method. The False Position method works as follows:

Start with an interval [a, b] where the function changes sign (f(a) * f(b) < 0).

Calculate the point c where the line connecting (a, f(a)) and (b, f(b)) intersects the x-axis.

Check the sign of f(c):

If f(c) ≈ 0 (within a predefined tolerance), c is the root.

If f(c) has the same sign as f(a), replace a with c, else replace b with c.

Repeat the process until the interval becomes sufficiently small or the function value is close to zero.

Program:

```
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;
double arr[100];
long double polynom(long double x, int d)
{
    long double sum = 0;
    for (int i = d; i >= 0; i--)
        sum += (arr[i] * pow(x, i));
    return sum;
}
void method(int de, long double a, long double b, bool h)
{
    int i = 1;
    long double temp1 = polynom(a, de), temp2 = polynom(b, de), mid = (h ==
true) ? ((a * temp2) - (b * temp1)) / (temp2 - temp1) : ((a + b) / 2);
    cout << "step  "<< "a   "<< "b   "<< "Xr   "<< "f(xr)   "<< "error" <<
endl;
    cout << "0"<< "   " << a << "   " << b << "   " << mid << "   " <<
polynom(mid, de) << "   " << mid << endl;
    if (temp1 * temp2 < 0)
    {
        while (i > 0)
        {
            long double prev = 0;
            long double chk = polynom(mid, de);
            prev = mid;
            if (temp1 * chk < 0)
                b = mid;
            else if (temp1 * chk == 0)
                break;
            else if (temp1 * chk > 0)
                a = mid;
            temp1 = polynom(a, de);
            temp2 = polynom(b, de);
            mid = (h == true) ? ((a * temp2) - (b * temp1)) / (temp2 - temp1)
: ((a + b) / 2);
            long double err = (mid - prev);
            cout << i << "   " << a << "   " << b << "   " << mid << "   " <<
chk << "   " << abs(err) << endl;
            if (abs(err) < 0.0001)
                break;
```

```
            i++;
        }
    }
    cout << "ANSWER : " << setprecision(5) << mid << endl;
}
int main()
{
    long double a, b;
    int deg;
    bool h;
    cout << "degree, a, b value : ";
    cin >> deg >> a >> b;
    cout << "What method do you want: (0 for bisection 1 for false position)
";
    cin >> h;
    cout << "Give the " << deg + 1 << " Coefficients: ";
    for (int i = deg; i >= 0; i--)
        cin >> arr[i];
    method(deg, a, b, h);
}
```

Input & Output: Bisection

```
C:\Users\Tonmo\OneDrive\Documents\Work\RUET_CSE>cd  C:\Users\Tonmo\O
ork\RUET_CSE\CSE2204\Lab1\q1\"q1
degree, a, b value : 3 2 3
What method do you want: (0 for bisection 1 for false position) 0
Give the 4 Coefficients: 1 0 -2 -5
step  a    b    Xr    f(xr)    error
0   2   3   2.5   5.625   2.5
1   2   2.5   2.25   5.625   0.25
2   2   2.25   2.125   1.89062   0.125
3   2   2.125   2.0625   0.345703   0.0625
4   2.0625   2.125   2.09375   -0.351318   0.03125
5   2.09375   2.125   2.10938   -0.00894165   0.015625
6   2.09375   2.10938   2.10156   0.166836   0.0078125
7   2.09375   2.10156   2.09766   0.0785623   0.00390625
8   2.09375   2.09766   2.0957   0.0347143   0.00195312
9   2.09375   2.0957   2.09473   0.0128623   0.000976562
10   2.09375   2.09473   2.09424   0.00195435   0.000488281
11   2.09424   2.09473   2.09448   -0.00349515   0.000244141
12   2.09448   2.09473   2.0946   -0.000770775   0.00012207
13   2.09448   2.0946   2.09454   0.000591693   6.10352e-05
ANSWER : 2.0945
```

False Position:

```
C:\Users\Tonmo\OneDrive\Documents\work\RUET_CSE>cd "C:\Users\Tonmo\OneD
ork\RUET_CSE\CSE2204\Lab1\q1\"q1
degree, a, b value : 3 2 3
What method do you want: (0 for bisection 1 for false position) 1
Give the 4 Coefficients: 1 0 -2 -5
step  a   b   Xr   f(xr)   error
0   2   3   2.05882   -0.3908   2.05882
1   2.05882   3   2.08126   -0.3908   0.0224401
2   2.08126   3   2.08964   -0.147204   0.00837555
3   2.08964   3   2.09274   -0.0546765   0.00310036
4   2.09274   3   2.09388   -0.0202029   0.00114413
5   2.09388   3   2.09431   -0.00745051   0.000421743
6   2.09431   3   2.09446   -0.00274567   0.000155395
7   2.09446   3   2.09452   -0.00101157   5.72476e-05
ANSWER : 2.0945

c:\Users\Tonmo\OneDrive\Documents\work\RUET_CSE\CSE2204\Lab1\q1>
```

Discussion: The Bisection and False Position methods are essential tools in numerical analysis for finding the roots of single-variable polynomials. The Bisection method relies on the concept of interval halving and is guaranteed to converge to a root if the initial interval contains a sign change in the function. The False Position method, on the other hand, uses linear interpolation to iteratively narrow down the interval containing the root. Both methods are relatively simple to implement and can be effective even when dealing with complex or non-differentiable functions. Choosing between them often depends on the specific characteristics of the function and the desired convergence speed, as the Bisection method provides slower but more reliable convergence, while the False Position method can be faster but may encounter convergence issues with certain functions.

"Heaven's Light is Our Guide"

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course Code:	CSE 2204
Course Name:	Numerical Methods Sessional
Experiment No:	02
Experiment Name:	Finding root of x^3-2x-5 by Iteration and Newton-Raphson Method.
Date of Experiment:	
Date of Submission:	January 7, 2024

Submitted By:	Submitted To:
Name: Tonmoy	Shyla Afroge
Section: A	Assistant Professor
Roll: 2003027	CSE, RUET
Year: 2nd year, Even Semester	

Title: Finding root of x^3-2x-5 by Iteration and Newton-Raphson Method.

Theoretical Background:

Iteration Method:

The Iteration Method, also known as the Fixed-Point Iteration Method, is a numerical technique for finding the roots of an equation of the form f(x) = 0. It is based on the idea of rearranging the equation into the form $x = \phi(x_n)$, where g(x) is a function that is easier to work with.

The general iterative formula for the Iteration Method is given by:

$x_{n+1} = \phi(x_n)$

Here, x_{n+1} is the next approximation, and x_n is the current approximation. The process is repeated until the sequence x_n converges to the root.

For convergence to occur, the iteration function $\phi(x_n)$ must satisfy the conditions:

1. $\phi(x_n)$ must be continuous at an interval containing the root.

2. $|\phi'(x_n)| < 1$ for all x in the interval.

Newton-Raphson Method:

The Newton-Raphson method is another iterative technique for finding the roots of a real-valued function f(x) = 0. It is based on linear approximation of the function around an initial guess.

The iteration formula for the Newton-Raphson method is given by:

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

Here, f'(x) is the derivative of f(x), and x_{n+1} is the next approximation.

The conditions for the Newton-Raphson method to converge include:

1. The initial guess should be close to the actual root.

2. $f'(x_n)$ should not be close to zero to avoid division by a very small number.

3. The function f(x) and its derivative f'(x) should be continuous.

Program:

```
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;
double arr[100];
long double polynom2(long double x, int d, int h)
{
    long double sum = 0;

    if (h == 0 || h == 2)
    {
        for (int i = (d - 2); i >= -1; i--)
            sum += (arr[i + 1] * pow(x, i));
    }
    else if (h == 1)
    {
        for (int i = d; i >= 0; i--)
            sum += (arr[i] * pow(x, i));
    }
    if (h == 0)
        return sqrt(-sum);
    else if (h == 1)
        return (x - (sum / ((3 * pow(x, 2)) - 2)));
    else if (h == 2)
        return round(abs(-2.5 * pow(x, -2) * pow(-sum, -0.5)) * 10) / 10;
    return 0;
}
void method(int de, long double a, long double b, int h)
{
    int i = 1;
    long double prev = a, chk, err, k = polynom2(b, de, 2);
    long double e = (h == 0) ? ((1 - k) / k) * 0.0001 : 0.0001;
    (h == 0) ? cout << "Iteration Method" << endl : cout << "Newton Raphson
Method" << endl;
    cout << "step  "
         << "Xr   "
         << "error" << endl;
    while (i > 0)
    {
        chk = polynom2(prev, de, h);
        err = (chk - prev);
        prev = chk;
        cout << i << "    " << chk << "    " << abs(err) << endl;
```

```
        if (abs(err) < e)
            break;
        i++;
    }

    cout << "ANSWER : " << prev << endl;
}
int main()
{
    long double a, b;
    int deg;
    int h;
    cout << "degree, a, b value : ";
    cin >> deg >> a >> b;
    cout << "What method do you want: (0 for Iteration 1 for Newton Raphson) ";
    cin >> h;
    cout << "Give the " << deg + 1 << " Coefficients: ";
    for (int i = deg; i >= 0; i--)
        cin >> arr[i];
    method(deg, a, b, h);
}
```

Input & Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Tonmo\OneDrive\Documents\RUET_CSE> cd "c:\Users\Tonmo\OneDrive\Documents\RUET_CSE\CSE2204\Lab2\" ;
sk }
degree, a, b value : 3 2 3
What method do you want: (0 for Iteration 1 for Newton Raphson) 0
Give the 4 Coefficients: 1 0 -2 -5
Iteration Method
step  Xr   error
1   2.12132   0.12132
2   2.08735   0.0339721
3   2.09652   0.00916883
4   2.09402   0.00249989
5   2.0947   0.000679723
ANSWER : 2.0947
PS C:\Users\Tonmo\OneDrive\Documents\RUET_CSE\CSE2204\Lab2> cd "c:\Users\Tonmo\OneDrive\Documents\RUET_CSE\CSE2
f ($?) { .\task }
degree, a, b value : 3 2 3
What method do you want: (0 for Iteration 1 for Newton Raphson) 1
Give the 4 Coefficients: 1 0 -2 -5
Newton Raphson Method
step  Xr   error
1   2.1   0.1
2   2.09457   0.00543188
3   2.09455   1.66394e-05
ANSWER : 2.0946
PS C:\Users\Tonmo\OneDrive\Documents\RUET_CSE\CSE2204\Lab2>
```

Comparisons:

Convergence Rate:

- The Newton-Raphson method typically converges faster than the Iteration Method.
- Newton-Raphson has quadratic convergence (approximately doubles the number of correct digits with each iteration) when close to the root.

Derivative Requirement:

- Newton-Raphson requires the derivative of the function, which may not always be readily available.
- Iteration Method only requires the function itself.

Ease of Implementation:

- The Iteration Method is generally easier to implement since it only requires the function $\phi(x_n)$ without the need for derivatives.

"Heaven's Light is Our Guide"

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course Code:	CSE 2204
Course Name:	Numerical Methods Sessional
Experiment No:	03
Experiment Name:	Exploring Functional Values through Newton's Forward and Backward Interpolation Formula.
Date of Experiment:	
Date of Submission:	January 7, 2024

Submitted By:	Submitted To:
Name: Tonmoy	Shyla Afroge
Section: A	Assistant Professor
Roll: 2003027	CSE, RUET
Year: 2nd year, Even Semester	

Title: Exploring Functional Values through Newton's Forward and Backward Interpolation Formula.

Theoretical Background:

Newton's Forward Interpolation Formula:

The forward interpolation method estimates the value of a function at a specified point within the range of given values. It involves constructing a forward difference table based on known data points and their respective function values. Using these divided difference values, the formula predicts the value of the function at a specified point.

Given n+1 equally spaced data points (x_0, y_0), (x_1, y_1), ..., (x_n, y_n) with equal intervals h, setting $x=x_0+ph$ we get the formula for forward interpolation is:

$$y_n(x) = y_0 + p\Delta y_0 + \frac{p(p-1)}{2!}\Delta^2 y_0 + \frac{p(p-1)(p-2)}{3!}\Delta^3 y_0 + \ldots. + \frac{p(p-1)(p-2)\ldots\ldots(p-n+1)}{n!}\Delta^n y_0$$

Newton's Backward Interpolation Formula:

Like forward interpolation, backward interpolation estimates the value of a function within the range of given values. However, it works backward from the data points to compute the divided differences and subsequently predicts the value at the specified point.

Given n+1 equally spaced data points (x_0, y_0), (x_1, y_1), ..., (x_n, y_n) with equal intervals h, setting $x=x_n+ph$ we get the formula for forward interpolation is:

$$y_n(x) = y_n + p\nabla y_n + \frac{p(p+1)}{2!}\nabla^2 y_n + \frac{p(p+1)(p+2)}{3!}\nabla^3 y_n + \ldots + \frac{p(p+1)(p+2)\ldots\ldots(p+n-1)}{n!}\nabla^n y_n$$

These methods offer a means to estimate the value of a function at points where the actual function values are not explicitly given but can be interpolated from the available data. They find applications in numerical analysis, engineering, and scientific computations for approximating values within a set of given data points.

Program:

```
#include <iostream>
#include <windows.h>
using namespace std;
double calculateCombination(double n, double r, bool h)
{
    double result = 1;
    for (int i = 1; i <= r; ++i)
    {
        if (h == true)
            result *= (n - i + 1) / i;
        else if (h == false)
            result *= (n + i - 1) / i;
    }
    return result;
}

void NewtonForward(double x, int n, double ys[], double xs[], double h)
{
    double sum = ys[0];
    cout << "x   ";
    for (int i = 0; i < n; i++)
        cout << xs[i] << " ";
    cout << endl;
    cout << "y   ";
    double ys1[n];
    for (int i = 0; i < n; i++)
    {
        ys1[i] = ys[i];
        cout << ys1[i] << " ";
    }

    cout << endl;
    for (int j = 0; j < n - 1; j++)
    {
        cout << "\u0394" << j + 1 << "y  ";
        for (int i = 0; i < n - (j + 1); i++)
        {
            cout << ys1[i + 1] - ys1[i] << " ";
            ys1[i] = ys1[i + 1] - ys1[i];
        }
        cout << endl;
        double p = (x - xs[0]) / h;
        sum += (calculateCombination(p, j + 1, true) * ys1[0]);
```

```
    }
    cout << "The Answer is: " << sum << endl;
}
void NewtonBackward(double x, int n, double ys[], double xs[], double h)
{
    double sum = ys[n - 1];
    cout << "x    ";
    for (int i = n - 1; i >= 0; i--)
        cout << xs[i] << " ";
    cout << endl;
    cout << "y    ";
    for (int i = n - 1; i >= 0; i--)
        cout << ys[i] << " ";
    cout << endl;
    for (int j = 0; j < n - 1; j++)
    {
        cout << "\u0394" << j + 1 << "y  ";
        for (int i = n - 2; i >= j; i--)
        {
            cout << ys[i + 1] - ys[i] << " ";
            ys[i + 1] = ys[i + 1] - ys[i];
        }
        cout << endl;
        double p = (x - xs[n - 1]) / h;
        // cout << p << endl;
        // cout << calculateCombination(p, j + 1, false) << " " << y[n - 1] <<
endl;
        sum += (calculateCombination(p, j + 1, false) * ys[n - 1]);
        // cout << sum << endl;
    }
    cout << "The Answer is: " << sum << endl;
}
int main()
{
    SetConsoleOutputCP(CP_UTF8);
    int n;
    cout << "How many points: ";
    cin >> n;
    double x[n], y[n];
    cout << "Enter x0 Values: ";
    for (int i = 0; i < n; i++)
        cin >> x[i];
    cout << "Enter y0 Values: ";
    for (int i = 0; i < n; i++)
        cin >> y[i];
```

```
    cout << "Enter x value: ";
    double x1;
    cin >> x1;
    double h = x[1] - x[0];
    while (1)
    {
        cout << "InterPolation\n1. Newton's Forward\n2. Newton's Backward\nplease
choose:" << endl;
        int choice;
        cin >> choice;
        switch (choice)
        {
        case 1:
            NewtonForward(x1, n, y, x, h);
            break;
        case 2:
            NewtonBackward(x1, n, y, x, h);
            break;
        }
    }
}
```

Input and Output:

```
PS C:\Users\Tonmo\OneDrive\Documents\RUE
How many points: 4
Enter x0 Values: 1 3 5 7
Enter y0 Values: 24 120 336 720
Enter x value: 8
InterPolation
1. Newton's Forward
2. Newton's Backward
please choose:
1
x   1 3 5 7
y   24 120 336 720
Δ1y  96 216 384
Δ2y  120 168
Δ3y  48
The Answer is: 990
InterPolation
1. Newton's Forward
2. Newton's Backward
please choose:
2
x    7 5 3 1
y    720 336 120 24
Δ1y  384 216 96
Δ2y  168 120
Δ3y  48
The Answer is: 990
InterPolation
1. Newton's Forward
2. Newton's Backward
please choose:
```

Discussion:

Newton's Forward and Backward Interpolation Formulas are distinct methods used to estimate unknown function values between known data points. Newton's Forward Interpolation moves in a forward direction from known data towards the desired point, employing forward differences to calculate values. It is suitable for predicting future values within the given range of data points. Conversely, Newton's Backward Interpolation moves in a reverse direction from known data towards the desired point, using backward differences to estimate values. This method is effective for estimating past values or values beyond the provided data range. Both techniques involve constructing difference tables but operate in opposite directions, catering to different scenarios based on the directionality of required estimations in relation to the available data points. The choice between these methods depends on whether future or past values need to be predicted and their relation to the existing data range.

"Heaven's Light is Our Guide"

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course Code:	CSE 2204
Course Name:	Numerical Methods Sessional
Experiment No:	04
Experiment Name:	Find the best values of a_0 and a_1 if the straight line $Y = a_0 + a_1 x$ fits to the data of (x_i, y_i).
Date of Experiment:	
Date of Submission:	January 7, 2024

Submitted By:	Submitted To:
Name: Tonmoy	Shyla Afroge
Section: A	Assistant Professor
Roll: 2003027	CSE, RUET
Year: 2nd year, Even Semester	

Title: Find the best values of a_0 and a_1 if the straight line $Y = a_0 + a_1 x$ fits to the data of (x_i, y_i).

Theoretical Background: (Least Squares Curve Fitting)

Let the set of data points be (x_i, y_i), i = 1, 2, ... m and let the curve given by $y = f(x)$ fits to this data. At $x = x_i$ the given ordinate is y and the corresponding value on the fitting curve is f(x). If e, is the error of approximation at $x = x_i$, then we have $e_i = y_i - f(x_i)$, if we write,

$$S = [y_1 - f(x_1)]^2 + [y_2 - f(x_2)]^2 + \cdots + [y_m - f(x_m)]^2 = e_1^2 + e_2^2 + \cdots + e_m^2$$

Then the method of least squares consists of minimizing S, i.e., the sum of the square of errors. The given values of (x_i, y_i) are $(1, 0.6), (2, 2.4), (3, 3.5), (4, 4.8), (5, 5.7)$.

x_i	y_i	x_i^2	$x_i y_i$	$(y_i - \bar{y})$	$(y_i - a_0 - a_i x)^2$
1	0.6	1	0.6	7.84	0.0784
2	2.4	4	4.8	1.00	0.0676
3	3.5	9	10.5	0.01	0.0100
4	4.8	16	19.2	1.96	0.0196
5	5.7	25	28.5	5.29	0.0484
15	17.0	55	63.6	16.10	0.2240

From the Table given above we find $\bar{x} = 3,\ \bar{y} = 3.4$.

We know, for least square curve fitting for straight line,

$$a_1 = \frac{m\sum_{i=1}^{m} x_i y_i - (\sum_{i=1}^{m} x_i \sum_{i=1}^{m} y_i)}{m\sum_{i=1}^{m} x_i^2 - \left(\sum_{i=1}^{m} x_i^2\right)^2} = \frac{5 * 63.6 - 15 * 17}{5 * 55 - 22.5} = 1.26$$

$$a_0 = \bar{y} - a_0 \bar{x} = -0.38$$

Program:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    int n;
    cin >> n;
    double x[n], y[n], sumx = 0, sumy = 0;
    double avg1 = 0, avg2 = 0, a = 0, b = 0, c = 0, sq1 = 0;
    cout << "xi "
         << "xi2\n";
    for (int i = 0; i < n; i++)
    {
        cin >> x[i];
        sumx += x[i];
        sq1 += pow(x[i], 2);
        cout << x[i] << " " << pow(x[i], 2) << endl;
    }
    cout << "yi "
         << "xiyi\n";
    for (int i = 0; i < n; i++)
    {
        cin >> y[i];
        sumy += y[i];
        a += (x[i] * y[i]);
        cout << y[i] << " " << (x[i] * y[i]) << endl;
    }
    avg1 = sumx / n;
    avg2 = sumy / n;
    double a1 = ((n * a) - (sumx * sumy)) / ((n * sq1) - pow(sumx, 2));
    double a0 = avg2 - (a1 * avg1);
    cout << "m = " << a1 << " c = " << a0 << endl;
}
```

Input & Output:

Input:

```
5
1 2 3 4 5
0.6 2.4 3.5 4.8 5.7
```

Received Output:

```
xi xi2
1 1
2 4
3 9
4 16
5 25
yi xiyi
0.6 0.6
2.4 4.8
3.5 10.5
4.8 19.2
5.7 28.5
m = 1.26 c = -0.38
```

Discussion:

Least squares curve fitting of a straight line involves finding the line those best fits a given set of data points by minimizing the sum of the squared vertical distances between each data point and the line. This technique uses the principle of least squares to determine the optimal slope and intercept of the line, ensuring that it approximates the data as closely as possible. By minimizing the squared errors, this method calculates the parameters that define the straight line, allowing for accurate predictions and a reliable representation of the relationship between variables. Its widespread application stems from its ability to efficiently model linear relationships within datasets across various disciplines, making it a cornerstone of regression analysis and data interpretation.

"Heaven's Light is Our Guide"

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course Code:	CSE 2204
Course Name:	Numerical Methods Sessional
Experiment No:	05
Experiment Name:	Finding integration of $\frac{1}{1+x}$ by numerical methods Trapezoidal Rule, Simpson's 1/3-Rule & Simpson's 3/8-Rule.
Date of Experiment:	
Date of Submission:	January 7, 2024

Submitted By:	Submitted To:
Name: Tonmoy	Shyla Afroge
Section: A	Assistant Professor
Roll: 2003027	CSE, RUET
Year: 2nd year, Even Semester	

Title: Finding integration of $\frac{1}{1+x}$ by numerical methods Trapezoidal Rule, Simpson's 1/3-Rule & Simpson's 3/8-Rule.

Theoretical Background:

Let the interval $[a, b]$ be divided into n equal subintervals such that $a = x_0 < x_1 < x_2 < \cdots x_n = b$. Clearly, $x_n = x_0 + nh$. Hence the integral becomes,

$$I = \int_{x_1}^{x_0} y dx$$

On simplification,

$$\int_{x_1}^{x_0} y dx = nh[y_0 + \frac{n}{2}\Delta y_0 + \frac{n(2n-3)}{12}\Delta^2 y_0 + n(n-2)^2\Delta^3 y_0 + \cdots]$$

Trapezoidal Rule: Setting $n = 1$ and simplifying,

$$\int_{x_1}^{x_0} y dx = \frac{h}{2}[y_0 + 2(y_1 + y_2 + \cdots + y_{n-1}) + y_n]$$

Simpson's 1/3-Rule: Setting $n = 2$ and simplifying,

$$\int_{x_1}^{x_0} y dx = \frac{h}{3}[y_0 + 4(y_1 + y_3 + y_5 + \cdots + y_{n-1}) + 4(y_2 + y_4 + y_6 + \cdots + y_{n-2}) + y_n]$$

Simpson's 3/8-Rule: Setting $n = 3$ and simplifying,

$$\int_{x_1}^{x_0} y dx = \frac{3h}{8}[(y_0 + 3y_1 + 3y_2 + y_3) + (y_3 + 3y_4 + 3y_5 + y_6) + \cdots + (y_{n-3} + 3y_{n-2} + 3y_{n-1} + y_n)]$$

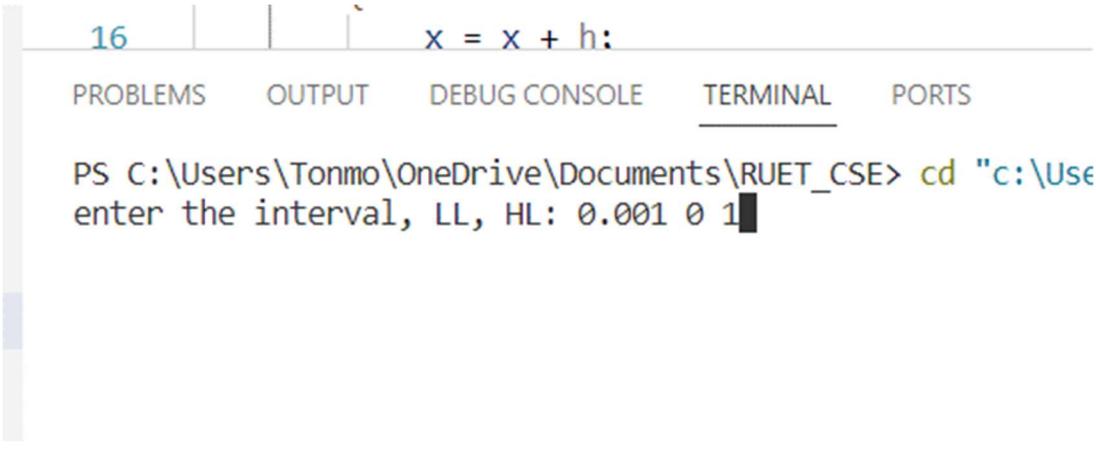
Program:

```
#include <iostream>
using namespace std;
double funk(double x)
{
    return 1 / (1 + x);
}

double integration(double h, int y, int a, int b)
{
    double n = ((b - a) / h), x = a, sum = funk(x);
    for (int i = 1; i <= n; i++)
    {
        if (y == 1)
        {
            x = x + h;
            cout << x << " " << funk(x) << endl;
            sum += 2 * funk(x);
        }
        else if (y == 2)
        {
            if (i % 2 == 0)
            {
                x = x + h;
                cout << x << " " << funk(x) << endl;
                sum += 2 * funk(x);
            }
            else if (i % 2 != 0)
            {
                x = x + h;
                cout << x << " " << funk(x) << endl;
                sum += 4 * funk(x);
            }
        }
        else if (y == 3)
        {
            if (i % 3 == 0)
            {
                x = x + h;
                cout << x << " " << funk(x) << endl;
                sum += 2 * funk(x);
            }
            else if (i % 3 != 0)
            {
```

```
                x = x + h;
                cout << x << " " << funk(x) << endl;
                sum += 3 * funk(x);
            }
        }
    }
    if (y == 1)
        sum = (h / 2) * (sum + funk(x + n));
    else if (y == 2)
        sum = (h / 3) * (sum + funk(x + n));
    else if (y == 3)
        sum = (3 * h / 8) * (sum + funk(x + n));
    return sum;
}
int main()
{
    double h, a, b;
    cout << "enter the interval, LL, HL: ";
    cin >> h >> a >> b;
    double re = integration(h, 1, a, b);
    double re2 = integration(h, 2, a, b);
    double re3 = integration(h, 3, a, b);
    cout << "The area value of 1/(1+x) by trapizoidal is = " << re << endl;
    cout << "The area value of 1/(1+x) by simpson's 1/3 rule is = " << re2 <<
endl;
    cout << "The area value of 1/(1+x) by simpson's 3/8 rule is = " << re3 <<
endl;
}
```

Input & Output:



```
0.997 0.500751
0.998 0.500501
0.999 0.50025
1 0.5
The area value of 1/(1+x) by trapizoidal is = 0.693398
The area value of 1/(1+x) by simpson's 1/3 rule is = 0.693314
The area value of 1/(1+x) by simpson's 3/8 rule is = 0.693398
PS C:\Users\Tonmo\OneDrive\Documents\RUET_CSE\CSE2204\Lab5\Q1>
```

Discussion: Numerical integration of the function $\frac{1}{1+x}$ using the Trapezoidal Rule, Simpson's 1/3-Rule, and Simpson's 3/8-Rule involves approximating the definite integral by dividing the interval into smaller segments. The Trapezoidal Rule calculates the area using trapezoids, while Simpson's 1/3-Rule and 3/8-Rule utilize quadratic and cubic polynomial approximations, respectively. These methods improve accuracy compared to simple geometric shapes. The choice between them depends on the number of subintervals: Simpson's 1/3-Rule requires an even number, while Simpson's 3/8-Rule demands a multiple of 3. Adjusting the number of subintervals allows for a balance between computational efficiency and accuracy in estimating the integral.