Haven's Light is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course Code:	CSE 2204
Course Title:	Numerical Methods Sessional
Experiment No:	01
Experiment Name:	Solving Algebric and Transcendental Equations using Bisection and False Position Method and Comparison of their Efficency

Date:

December 15, 2023

Submitted By:	Submitted To:
Name: Md. Abdullah Al Mamun	Shyla Afroge
Section: A	Assistant Professor
Roll No: 2003028	Computer Science & Engineering
Year: 2nd Year Odd Semester	Rajshahi University of Engineering & Technology

Experiment No: 01

Experiment Name: Solving Algebric and Transcendental Equations using Bisection and False Position Method and Comparison of their Efficency

Theory:

Bisection Method:

The bisection method is a numerical approach employed for finding the root of a real-valued function within a specified interval. The method begins by selecting an initial interval $[a, b]$ such that the function has opposite signs at the endpoints ($f(a) \cdot f(b) < 0$), adhering to the Intermediate Value Theorem. The process iteratively narrows down the interval by evaluating the function at the midpoint $c = \frac{a+b}{2}$. If $f(c) = 0$, c is the root; otherwise, the interval containing the root is determined based on the signs of $f(a)$ and $f(c)$. This iterative process continues until the interval becomes sufficiently small or a predetermined number of iterations is reached, ensuring convergence. While the bisection method is straightforward and guaranteed to converge, it may converge slowly for certain functions due to its halving of the interval at each step. Nevertheless, its reliability and simplicity make it a widely used numerical technique in various scientific and engineering applications.

Algorithm:

1. Interval Selection: Choose an initial interval $[a, b]$ such that the function has opposite signs at the endpoints ($f(a) \cdot f(b) < 0$), adhering to the Intermediate Value Theorem.

2. Iterative Process: Divide the interval in half and evaluate the function at the midpoint $c = \frac{a+b}{2}$. If $f(c) = 0$, c is the root. Otherwise, determine the subinterval $[a, c]$ or $[c, b]$ in which the root must lie based on the signs of $f(a)$ and $f(c)$ (or $f(b)$ and $f(c)$).

3. Convergence Criteria: Repeat the process iteratively until the interval becomes sufficiently small or until a predetermined number of iterations is reached. The Bisection method converges to the root because it progressively narrows down the interval containing the root.

4. Accuracy Improvement: The accuracy of the root approximation improves with each iteration, and the method typically converges linearly. The precision can be controlled by specifying a tolerance level or a maximum number of iterations.

False Position Method:

The False Position method, also known as the Regula Falsi method, is a numerical technique used for approximating the root of a real-valued function within a specified interval. Similar to the bisection method, it relies on the Intermediate Value Theorem, but instead of bisecting the interval, it employs linear interpolation between function values. The method iteratively updates the interval based on the function values at the endpoints, aiming to reduce the width of the interval containing the root. The False Position method can converge faster than the bisection method, but it may encounter convergence issues if the initial interval is poorly chosen or if the function exhibits erratic behavior. Despite these considerations, the method remains a valuable tool for numerical root-finding, particularly when a faster convergence rate is desired.

Algorithm:

1. Interval Selection: Choose an initial interval $[a, b]$ such that the function has opposite signs at the endpoints ($f(a) \cdot f(b) < 0$), adhering to the Intermediate Value Theorem.

2. Linear Interpolation: Interpolate linearly between function values at the endpoints to find the point c where the line intersects the x-axis. Evaluate $f(c)$.

3. Interval Update: Determine the subinterval $[a, c]$ or $[c, b]$ in which the root must lie based on the signs of $f(a)$ and $f(c)$ (or $f(b)$ and $f(c)$).

4. Convergence Criteria: Repeat the process iteratively until the interval becomes sufficiently small or until a predetermined number of iterations is reached. The False Position method aims to reduce the width of the interval containing the root.

5. Accuracy Improvement: The accuracy of the root approximation improves with each iteration, and the method typically converges faster than the Bisection method. However, it may encounter convergence issues if the initial interval is poorly chosen or if the function exhibits erratic behavior.

Program:

Listing 1: Bisection Method

```
double rootByBisection(double a, double b, bool print = false)
{
    int i = 0;
    double error = abs(a - b), c;

    while (abs(error) > 0.0001)
    {
        if (print)
            cout << i + 1 << "\t|\t" << a << "\t|\t" << b << "\t|\t" <<
                c << "\t\t|\t" << function(c) << "\t\t|\t" << error <<
                "\n";

        c = (a + b) / 2;
        if (function(a) * function(c) < 0)
            b = c;
        else
            a = c;

        error = abs(a - b);

        if (i > MAX_ITERATION)
            break;
        i++;
    }
    return c;
}
```

Listing 2: False Position Method

```
double rootByFalsePosition(double a, double b, bool print = false)
{
    int i = 0;
    double c;

    while (abs(function(c)) > 0.0001)
    {
```

```
            if (print)
                cout << i + 1 << "\t|\t" << a << "\t|\t" << b << "\t|\t" <<
                    c << "\t\t|\t" << function(c) << "\t\t|\t" <<
                    function(c) * 100 << "%\n";

            c = b - function(b) * (b - a) / (function(b) - function(a));

            if (function(a) * function(c) < 0)
                b = c;
            else
                a = c;

            if (i > MAX_ITERATION)
                break;
            i++;
        }
        return c;
    }
```

Listing 3: Main Program

```
    #include <iostream>
    #define MAX_ITERATION 100
    using namespace std;

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

    int main()
    {
        int choice;
        double a = -1, b = 1;

        while (function(a) * function(b) > 0)
        {
            function(b) > function(a) ? b++ : a--;
        }

        // cout << "a: " << a << "\tb: " << b << "\n";

        cout << "Menu Program: \n";
        cout << "\t1. Bisection Method\n";
        cout << "\t2. False Position Method\n";
        cout << "\t3. Compare Both Methods\n";
        cout << "Enter Your Choice: ";
        cin >> choice;

        cout << "\n\nn\t|\ta\t|\tb\t|\tx\t\t|\tf(x)\t\t|\terror\n";
        cout <<
            "-------------------------------------------------------------\n";

        switch (choice)
        {
        case 1:
            rootByBisection(a, b, true);
            break;
        case 2:
            rootByFalsePosition(a, b, true);
            break;
        case 3:
            cout << "Bisection Method: " << rootByBisection(a, b) << " with
                error: " << function(rootByBisection(a, b)) * 100 << "%\n";
            cout << "False Position Method: " << rootByFalsePosition(a, b)
                << " with error: " << function(rootByFalsePosition(a, b)) *
```

```
                    100 << "%\n";
43                break;
44            }
45        }
```

Result:

The Bisection and False Position methods are numerical techniques employed for finding the roots of a real-valued function. In the Bisection method, the interval containing the root is successively halved until a sufficiently accurate root approximation is achieved. While it guarantees convergence, it may be slow for certain functions. On the other hand, the False Position method, also known as the Regula Falsi method, utilizes linear interpolation between function values to approximate the root. Although it typically converges faster than Bisection, it can encounter convergence issues if the initial interval is poorly chosen or if the function exhibits erratic behavior. Both methods have their strengths and limitations, with the choice between them often dependent on the specific characteristics of the function being analyzed.

```
┌──(shohag㉿Shohag-Ubuntu)-[~/Documents/LABs/CSE-2204]
└─$ cd "/home/shohag/Documents/LABs/CSE-2204/" && g++ lab01.cpp -o lab01 && "/home/shohag/Documents/LABs/CSE-2204/"lab01
Menu Program:
        1. Bisection Method
        2. False Position Method
        3. Compare Both Methods
Enter Your Choice: 3

n       |       a       |       b       |       x               |       f(x)            |       error
-------------------------------------------------------------------------------------------------------------
Bisection Method:  with error: 0.0762014%
1       |       -2      |       3       |       6.9191e-310     |       -5              |       5
2       |       0.5     |       3       |       0.5             |       -5.875          |       2.5
3       |       1.75    |       3       |       1.75            |       -3.14062        |       1.25
4       |       1.75    |       2.375   |       2.375           |       3.64648         |       0.625
5       |       2.0625  |       2.375   |       2.0625          |       -0.351318       |       0.3125
6       |       2.0625  |       2.21875 |       2.21875         |       1.48508         |       0.15625
7       |       2.0625  |       2.14062 |       2.14062         |       0.527683        |       0.078125
8       |       2.0625  |       2.10156 |       2.10156         |       0.0785623       |       0.0390625
9       |       2.08203 |       2.10156 |       2.08203         |       -0.138761       |       0.0195312
10      |       2.0918  |       2.10156 |       2.0918          |       -0.0306977      |       0.00976562
11      |       2.0918  |       2.09668 |       2.09668         |       0.0237823       |       0.00488281
12      |       2.09424 |       2.09668 |       2.09424         |       -0.00349515     |       0.00244141
13      |       2.09424 |       2.09546 |       2.09546         |       0.0101342       |       0.0012207
14      |       2.09424 |       2.09485 |       2.09485         |       0.00331719      |       0.000610352
15      |       2.09454 |       2.09485 |       2.09454         |       -8.95647e-05    |       0.000305176
16      |       2.09454 |       2.0947  |       2.0947          |       0.00161367      |       0.000152588
False Position Method:  with error: -0.00997117%
1       |       -2      |       3       |       6.9191e-310     |       -5              |       -500%
2       |       -0.2    |       3       |       -0.2            |       -4.608          |       -460.8%
3       |       0.515528        |       3       |       0.515528        |       -5.89404        |       -589.404%
4       |       1.18437 |       3       |       1.18437         |       -5.70739        |       -570.739%
5       |       1.66174 |       3       |       1.66174         |       -3.73478        |       -373.478%
6       |       1.915   |       3       |       1.915           |       -1.80723        |       -180.723%
7       |       2.02512 |       3       |       2.02512         |       -0.745012       |       -74.5012%
8       |       2.06849 |       3       |       2.06849         |       -0.286605       |       -28.6605%
9       |       2.08488 |       3       |       2.08488         |       -0.107306       |       -10.7306%
10      |       2.09098 |       3       |       2.09098         |       -0.0397673      |       -3.97673%
11      |       2.09324 |       3       |       2.09324         |       -0.0146816      |       -1.46816%
12      |       2.09407 |       3       |       2.09407         |       -0.00541269     |       -0.541269%
13      |       2.09437 |       3       |       2.09437         |       -0.00199447     |       -0.199447%
14      |       2.09449 |       3       |       2.09449         |       -0.00073478     |       -0.073478%
15      |       2.09453 |       3       |       2.09453         |       -0.000270681    |       -0.0270681%
```

Figure 1: Output of the Program

Haven's Light is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course Code:	CSE 2204
Course Title:	Numerical Methods Sessional
Experiment No:	02
Experiment Name:	Solving Equations using Iteration and Newton-Rapson Method

Date:

December 15, 2023

Submitted By:	Submitted To:
Name: Md. Abdullah Al Mamun	Shyla Afroge
Section: A	Assistant Professor
Roll No: 2003028	Computer Science & Engineering
Year: 2nd Year Odd Semester	Rajshahi University of Engineering & Technology

Experiment No: 02

Experiment Name: **Solving Equations using Iteration and Newton-Rapson Method**

Theory:

Iteration Method:

Iteration is a fundamental concept in mathematics and numerical methods, involving the process of repeatedly applying a rule or procedure to achieve a desired outcome. In the context of numerical analysis, iteration is commonly used to approximate solutions to equations, particularly when analytical solutions are challenging or impossible to find. The iterative process starts with an initial guess and refines it through successive iterations until a sufficiently accurate solution is obtained.

Algorithm:

1. Initial Guess: Begin with an initial guess or estimate for the solution.
2. Iterative Process: Apply a rule or procedure to refine the estimate through successive iterations.
3. Convergence Criteria: Continue the iterations until a specified convergence criterion is met, indicating that the solution is sufficiently accurate.

Newton-Raphson Method:

The Newton-Raphson Method is a powerful iterative technique for finding successively better approximations to the roots of a real-valued function. It is based on linear approximation and tangent lines. Starting with an initial guess x_0, the method iteratively refines the approximation using the formula $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, where $f(x)$ is the function and $f'(x)$ is its derivative. The method converges rapidly when the initial guess is close to the actual root, making it an efficient tool for root-finding. However, it may fail to converge or converge to an undesired root under certain conditions or if the function has peculiar characteristics. Despite these considerations, the Newton-Raphson Method is widely utilized in various fields for its speed and effectiveness in finding solutions to nonlinear equations.

Algorithm:

1. Initial Guess: Start with an initial guess x_0 for the root.
2. Iterative Formula: Apply the iterative formula $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, where $f(x)$ is the function and $f'(x)$ is its derivative.
3. Successive Refinement: Repeat the iterative process to successively refine the approximation to the root.
4. Convergence Criteria: Continue the iterations until a specified convergence criterion is met, indicating that the root approximation is sufficiently accurate.

Program:

Listing 1: Newton-Rapson Method

```
double newtonRapsonMethod(double x, double error)
{
    double x1 = x - function(x) / functionDerivative(x);
    double x2 = x1 - function(x1) / functionDerivative(x1);

    cout << "x0: " << x << "\tx1: " << x1 << "\n";

    int i = 1;
    while (abs(x2 - x1) > error)
    {
        x1 = x2;
        x2 = x1 - function(x1) / functionDerivative(x1);
        cout << "x" << i << ": " << x1 << "\tx" << (i + 1) << ": " <<
            x2 << "\n";
        i++;
    }

    return x2;
}
```

Listing 2: Iteration Method

```
double phiFunction(double x, int coeff[], int degree)
{
    double sum = 0;
    for (int i = 0; i <= degree; i++)
    {
        sum += coeff[i] * pow(x, i);
    }
    return sum;
}

double iterarionMethod(double x, double error)
{
    int degree;
    cout << "Enter degree for phi function: ";
    cin >> degree;
    int coeff[degree + 1];
    cout << "Enter " << degree + 1 << " Coefficients: ";
    for (int i = 0; i <= degree; i++)
        cin >> coeff[i];

    double x1 = phiFunction(x, coeff, degree);
    double x2 = phiFunction(x1, coeff, degree);

    while (abs(x2 - x1) > error)
    {
        x1 = x2;
        x2 = phiFunction(x1, coeff, degree);
        cout << "x1: " << x1 << "\tx2: " << x2 << "\n";
    }

    return x2;
}
```

Listing 3: functionInput.h helper file

```
#ifndef FUNCTIONINPUT_H
#define FUNCTIONINPUT_H
#include <iostream>
#include <cmath>
```

```
5
6       using namespace std;
7
8       int degree;
9       int *coefficients = new int[degree + 1];
10
11      void takeInputForFunction()
12      {
13          int deg;
14          cout << "Enter the degree of the polynomial: ";
15          cin >> deg;
16          int *coef = new int[deg + 1];
17          cout << "Enter " << deg + 1 << " coefficients of the polynomial: ";
18          for (int i = 0; i <= deg; i++)
19              cin >> coef[i];
20          delete[] coefficients; // free the previously allocated memory
21          coefficients = coef;
22          degree = deg;
23      }
24
25      double function(double x)
26      {
27          double sum = 0;
28          for (int i = 0; i <= degree; i++)
29          {
30              sum += coefficients[i] * pow(x, i);
31          }
32          return sum;
33      }
34
35      double functionDerivative(double x)
36      {
37          double sum = 0;
38          for (int i = 1; i <= degree; i++)
39              sum += i * coefficients[i] * pow(x, i - 1);
40          return sum;
41      }
42
43      #endif // !FUNCTIONINPUT_H
```

Listing 4: Main Program

```
1       #include <iostream>
2       #include "functionInput.h"
3       using namespace std;
4
5       int main()
6       {
7           takeInputForFunction();
8           double a;
9
10          cout << "Enter guess: ";
11          cin >> a;
12
13          cout << "\n1. Newton Rapson Method\n2. Iteration Method\n";
14          cout << "Choose the method you want to use: ";
15
16          int choice;
17          cin >> choice;
18          switch (choice)
19          {
20              {
21              case 1:
22              {
23                  double sol = newtonRapsonMethod(a, 0.0001);
24                  cout << "The solution using Newton-Rapson Method: " << sol
```

```
                    << endl;
                break;
            }

            case 2:
            {
                double sol = iterarionMethod(a, 0.0001);
                cout << "The solution using Iteration Method: " << sol <<
                    endl;
                break;
            }

            default:
                break;
            }
        }
    }
```

Result:

The Newton-Raphson Method and the Iteration Method are both iterative techniques used for approximating solutions to mathematical problems. The Newton-Raphson Method utilizes function derivatives for rapid convergence, especially when initial guesses are close to roots, but it may be sensitive to specific conditions. In contrast, the Iteration Method is a more general approach, suitable for a wide range of functions, although it tends to converge more slowly. The choice between these methods depends on the specific characteristics of the problem and the quality of the initial guess.

```
┌──(shohag㉿Shohag-Ubuntu)-[~/Documents/LABs/CSE-2204]
└─$ cd "/home/shohag/Documents/LABs/CSE-2204/" && g++ lab02.cpp -o lab02 && "/home/shohag/Documents/LABs/CSE-2204/"lab02
Enter the degree of the polynomial: 2
Enter 3 coefficients of the polynomial: 1 -2 1
Enter guess: 0.6

1. Newton Rapson Method
2. Iteration Method
Choose the method you want to use:1
x0: 0.6 x1: 0.8
x1: 0.9 x2: 0.95
x2: 0.95        x3: 0.975
x3: 0.975       x4: 0.9875
x4: 0.9875      x5: 0.99375
x5: 0.99375     x6: 0.996875
x6: 0.996875    x7: 0.998438
x7: 0.998438    x8: 0.999219
x8: 0.999219    x9: 0.999609
x9: 0.999609    x10: 0.999805
x10: 0.999805   x11: 0.999902
The solution using Newton-Rapson Method: 0.999902
```

Figure 1: Output of the Program

Haven's Light is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course Code:	CSE 2204
Course Title:	Numerical Methods Sessional
Experiment No:	03
Experiment Name:	Exploring Functional Values through Newton's Forward and Backward Interpolation Formula.

Date:

December 15, 2023

Submitted By:	Submitted To:
Name: Md. Abdullah Al Mamun	Shyla Afroge
Section: A	Assistant Professor
Roll No: 2003028	Computer Science & Engineering
Year: 2nd Year Odd Semester	Rajshahi University of Engineering & Technology

Experiment No: 03

Experiment Name: **Exploring Functional Values through Newton's Forward and Backward Interpolation Formula.**

Theory:

Newton's Forward Interpolation Formula:

Given a set of data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ with a common difference $h = x_1 - x_0$, Newton's Forward Interpolation Formula estimates y at a point x within the range (x_0, x_n) as follows:

$$y = y_0 + t\Delta y_0 + \frac{t(t-1)}{2!}\Delta^2 y_0 + \ldots + \frac{t(t-1)\ldots(t-n+1)}{n!}\Delta^n y_0$$

where $t = \frac{x-x_0}{h}$ and $\Delta y_i = y_{i+1} - y_i$ are the finite differences.

Algorithm:

1. **Calculate Finite Differences:**
   - Calculate first-order differences: $\Delta y_i = y_{i+1} - y_i$ for $i = 0, 1, \ldots, n-1$.
   - Calculate second-order differences: $\Delta^2 y_i = \Delta y_{i+1} - \Delta y_i$ for $i = 0, 1, \ldots, n-2$.
   - Continue this process up to the n-th order differences.
2. **Calculate Coefficients:**
   - Initialize coefficients $a_0 = y_0$.
   - For $k = 1, 2, \ldots, n$, calculate $a_k = \frac{\Delta^k y_0}{k!}$.
3. **Interpolation Formula:**
   - The interpolated value at a point x is given by the formula:

     $$P(x) = a_0 + a_1 t + a_2 t(t-1) + \ldots + a_n t(t-1)\ldots(t-n+1)$$

     where $t = \frac{x-x_0}{h}$.

Newton's Backward Interpolation Formula:

When the data points are given in equally spaced intervals but in reverse order, Newton's Backward Interpolation Formula estimates y at a point x within the range (x_0, x_n) as follows:

$$y = y_n + t\Delta y_n + \frac{t(t+1)}{2!}\Delta^2 y_n + \ldots + \frac{t(t+1)\ldots(t+n-1)}{n!}\Delta^n y_n$$

where $t = \frac{x-x_n}{h}$ and $\Delta y_i = y_i - y_{i-1}$. These interpolation formulas are valuable tools when evenly spaced data points are available, and the choice between forward and backward interpolation depends on the arrangement of the given data points.

Algorithm:

1. **Calculate Finite Differences:**
   - Calculate first-order differences: $\Delta y_i = y_i - y_{i-1}$ for $i = 1, 2, \ldots, n$.
   - Calculate second-order differences: $\Delta^2 y_i = \Delta y_i - \Delta y_{i-1}$ for $i = 2, 3, \ldots, n$.
   - Continue this process up to the n-th order differences.
2. **Calculate Coefficients:**
   - Initialize coefficients $a_n = y_n$.
   - For $k = 1, 2, \ldots, n$, calculate $a_{n-k} = \frac{\Delta^k y_n}{k!}$.
3. **Interpolation Formula:**
   - The interpolated value at a point x is given by the formula:

$$P(x) = a_n + a_{n-1}t + a_{n-2}t(t+1) + \ldots + a_0 t(t+1)\ldots(t+n-1)$$

   where $t = \frac{x - x_n}{h}$.

Program:

Listing 1: Newton's Forward Interpolation

```
double NewtonForward(double x[], double y[], int n, double p)
{
    double dely[n][n], a = x[0], h = x[1] - x[0], u, y0;

    for (int i = 0; i < n; i++)
        dely[i][0] = y[i];

    for (int i = 1; i < n; i++)
        for (int j = 0; j < n - i; j++)
            dely[j][i] = dely[j + 1][i - 1] - dely[j][i - 1];

    int k = 0;
    while (a - p > h)
    {
        a = x[k];
        k++;
    }
    u = (p - a) / h;
    y0 = dely[0][0];
    double tempU = u;

    for (int i = 1; i < (n - k); i++)
    {
        if (i != 1)
            tempU = tempU * (u - i + 1);

        tempU = tempU / (double)i;
        y0 += tempU * dely[k][i];
    }
    return y0;
}
```

Listing 2: Newton's Backward Interpolation

```
double NewtonBackward(double x[], double y[], int n, double p)
{
    double dely[n][n], a = x[n - 1], h = x[1] - x[0], u, y0;

```

```
        for (int i = 0; i < n; i++)
            dely[i][0] = y[i];

        for (int i = 1; i < n; i++)
            for (int j = 0; j < n - i; j++)
                dely[j][i] = dely[j + 1][i - 1] - dely[j][i - 1];

        int k = 0;
        while (a - p > h)
        {
            a = x[k];
            k++;
        }
        u = (p - a) / h;
        y0 = dely[n - 1][0];
        double tempU = u;

        for (int i = 1; i < (n - k); i++)
        {
            if (i != 1)
                tempU = tempU * (u + i - 1);

            tempU = tempU / (double)i;
            y0 += tempU * dely[n - k - i - 1][i];
        }
        return y0;
    }
```

Listing 3: Main Program

```
    #include <iostream>
    using namespace std;

    void takeInput(double *x, double *y, int &n, double &x0)
    {
        cout << "Enter the number of data points: ";
        cin >> n;
        cout << "Enter the values of x: ";
        for (int i = 0; i < n; i++)
            cin >> x[i];

        cout << "Enter the values of y: ";
        for (int i = 0; i < n; i++)
            cin >> y[i];
        cout << endl;

        cout << "Enter the value of x for which y is to be found: ";
        cin >> x0;
    }

    int main()
    {
        int input = 0;
        while (input != 3)
        {
            // take all the inputs
            int n = 0;
            double x[10000], y[10000], x0;

            // take the choice
            cout << "1. Newton Forward" << endl;
            cout << "2. Newton Backward" << endl;
            cout << "3. Exit" << endl
                 << endl;
            cout << "Note: All the data must be equally spaced" << endl;
            cout << "Enter your choice: ";
            cin >> input;
            cout << endl;

            switch (input)
            {
            case 1:
```

```
            {
                takeInput(x, y, n, x0);
                double y0 = NewtonForward(x, y, n, x0);
                cout << "The value of y at x = " << x0 << " is " << y0 <<
                    endl
                      << endl
                      << endl;
                break;
            }
            case 2:
            {
                takeInput(x, y, n, x0);
                double y0 = NewtonBackward(x, y, n, x0);
                cout << "The value of y at x = " << x0 << " is " << y0 <<
                    endl
                      << endl
                      << endl;
                break;
            }
            case 3:
            {
                return 0;
            }
            default:
            {
                cout << "Invalid choice" << endl;
                break;
            }
                cout << endl
                      << endl;
            }
        }
    }
```

Result:

Newton's Forward and Backward Interpolation Formulas are methods for estimating values between known data points through polynomial interpolation. Forward Interpolation assumes equally spaced data from left to right, using forward finite differences. Backward Interpolation is for right-to-left data with backward finite differences. Both involve systematic finite difference and coefficient calculations, providing efficient interpolation. The choice depends on data arrangement, with forward for left-to-right and backward for right-to-left data. Despite directional distinctions, both share the same principles and are valuable for accurately estimating values within a given data range.

```
┌──(shohag㉿Shohag-Ubuntu)-[~/Documents/LABs/CSE-2204]
└─$ cd "/home/shohag/Documents/LABs/CSE-2204/" && g++ lab03.cpp -o lab03 && "/home/shohag/Documents/LABs/CSE-2204/"lab03
1. Newton Forward
2. Newton Backward
3. Exit

Note: All the data must be equally spaced
Enter your choice: 1

Enter the number of data points: 6
Enter the values of x: 15 20 25 30 35 40
Enter the values of y: 0.2588190 0.3420201 0.4226183 0.5 0.5735764 0.6427876

Enter the value of x for which y is to be found: 17
The value of y at x = 17 is 0.292372


1. Newton Forward
2. Newton Backward
3. Exit

Note: All the data must be equally spaced
Enter your choice: 2

Enter the number of data points: 6
Enter the values of x: 15 20 25 30 35 40
Enter the values of y: 0.2588190 0.3420201 0.4226183 0.5 0.5735764 0.6427876

Enter the value of x for which y is to be found: 38
The value of y at x = 38 is 0.615661
```

Figure 1: Output of the Program

Haven's Light is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course Code:	CSE 2204
Course Title:	Numerical Methods Sessional
Experiment No:	04
Experiment Name:	Find the straight line that best fits some given data using Least Square Method.

Date:

December 15, 2023

Submitted By:	Submitted To:
Name: Md. Abdullah Al Mamun	Shyla Afroge
Section: A	Assistant Professor
Roll No: 2003028	Computer Science & Engineering
Year: 2nd Year Odd Semester	Rajshahi University of Engineering & Technology

Experiment No: 04

Experiment Name: **Find the straight line that best fits some given data using Least Square Method.**

Theory:

Least Square Method:

The least squares method is a statistical technique used to determine the optimal parameters of a linear regression model by minimizing the sum of the squared differences between observed and predicted values. It aims to find the coefficients (intercept a_0 and slope a_1) that minimize the overall squared residuals in the data. The optimization involves solving the normal equations, resulting in formulas for a_0 and a_1 that provide the best-fitting line. This method is widely applied in various fields for modeling and predicting relationships between variables, as it provides a systematic way to estimate parameters that yield the most accurate linear approximation to the given data.

Algorithm:

Given a set of data points (x_i, y_i) for $i = 1, 2, \ldots, n$:

1. Formulate the linear regression model: $Y = a_0 + a_1 x + \varepsilon$, where Y is the dependent variable, x is the independent variable, and ε is the error term.

2. Define the cost function: $J(a_0, a_1) = \sum_{i=1}^{n}(y_i - (a_0 + a_1 x_i))^2$, representing the sum of squared differences between observed y_i and predicted values.

3. Find the partial derivatives of J with respect to a_0 and a_1:

$$\frac{\partial J}{\partial a_0} = -2\sum_{i=1}^{n}(y_i - (a_0 + a_1 x_i)), \quad \frac{\partial J}{\partial a_1} = -2\sum_{i=1}^{n} x_i(y_i - (a_0 + a_1 x_i))$$

4. Set the derivatives to zero and solve for a_0 and a_1:

$$a_1 = \frac{n\sum xy - (\sum x)(\sum y)}{n\sum x^2 - (\sum x)^2}, \quad a_0 = \frac{\sum y - a_1 \sum x}{n}$$

5. The resulting a_0 and a_1 values represent the optimal coefficients for the best-fitting line through the given data points.

Program:

Listing 1: Least Square Method Model

```
double leastSquare(double x[], double y[], int n, double *a1)
{
    double sumX = 0, sumY = 0, sumXY = 0, sumX2 = 0;
    for (int i = 0; i < n; i++)
    {
        sumX += x[i];
        sumY += y[i];
        sumXY += x[i] * y[i];
        sumX2 += x[i] * x[i];
    }
    *a1 = (n * sumXY - sumX * sumY) / (n * sumX2 - sumX * sumX);
    return (sumY - *a1 * sumX) / n;
}
```

Listing 2: Main Program

```
#include <iostream>
using namespace std;

int main()
{
    int n;
    cout << "Enter the number of values: ";
    cin >> n;
    double x[n], y[n];

    cout << "Enter the values of X: ";
    for (int i = 0; i < n; i++)
        cin >> x[i];

    cout << "Enter the values of Y: ";
    for (int i = 0; i < n; i++)
        cin >> y[i];

    double a0, a1;
    a0 = leastSquare(x, y, n, &a1);

    cout << "\nThe value of a0 is " << a0 << " and the value of a1 is "
       << a1 << endl;
    cout << "The equation is: y = " << a0 << " + " << a1 << "x" << endl;
    return 0;
}
```

Result:

The resulting coefficients a_0 and a_1 represent the best-fitting line that minimizes the sum of squared differences between the observed and predicted values. This line provides a linear relationship that can be used for prediction or inference. In summary, the least squares method is a powerful and widely used approach for fitting linear models to data, providing a systematic way to estimate the parameters that best describe the relationship between variables.

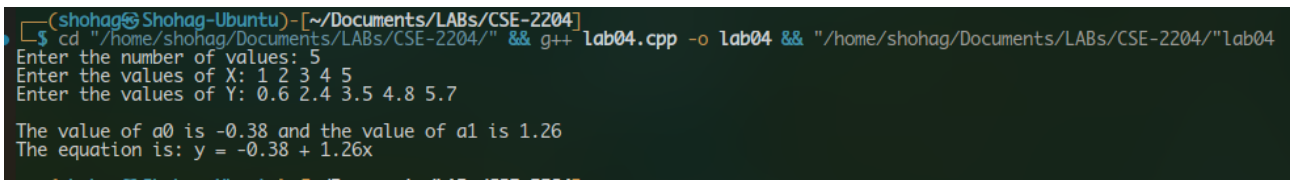

Figure 1: Output of the Program