Heaven's Light is Our Guide

RAJSHAHI UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Engineering

LAB REPORT

Course Code : CSE 2204

Course Name : Numerical Methods Sessional

Experiment No. : 04

Experiment Name : Numerical differentiation and Intregration

Submitted by:	**Submitted to:**
Name : MD Shehab Sarker Roll : 2103046 Section : A Session : 2021-2022	Shyla Afroge Associate Professor Computer Science & Engineering RUET

Date Of Submission : 07.12.2024

Problem 01: The numerical differentiation formulae is to differentiate by the interpolating polynomial.

Theory:

Numerical differentiation involves estimating the derivative of a function using discrete data points. This is particularly useful when the function is not available in a closed form, but only as a set of sampled values. The core idea is to approximate the derivative by utilizing the values of the function at certain points and applying mathematical formulations based on Taylor series expansions.

A simple approximation of the first derivative is

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

Derivatives using Newton's forward difference formula

Newton's forward interpolation formula (p. 274) is

$$y = y_0 + p\Delta y_0 + \frac{p(p-1)}{2!}\Delta^2 y_0 + \frac{p(p-1)(p-2)}{3!}\Delta^3 y_0 + \ldots$$

Differentiating both sides w.r.t. p, we have

$$\frac{dy}{dp} = \Delta y_0 + \frac{2p-1}{2!}\Delta^2 y_0 + \frac{3p^2 - 6p + 2}{3!}\Delta^3 y_0 + \ldots$$

Since $p = \frac{(x - x_0)}{h}$

Therefore $\frac{dp}{dx} = \frac{1}{h}$

Code:

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

int main() {
   float xp, h, sum = 0.0, term, first_derivative;
   int n, index = 0, flag = 0, sign = 1;
   double x[20],y[20][20];
   int i,j;
   //cout<<setprecision(7)<<fixed;
   //Input Section
   cout<<"Enter number of data:";
```

```
    cin>>n;

    cout<<"Enter Data: "<<endl;
    cout<<"x"<<"  "<<"y"<<endl;
    for(i=0;i<n;i++){
        cin>>x[i]>>y[i][0];
    }
// Reading the calculation point
    cout << "Enter at what value of x you want to calculate derivative: ";
    cin >> xp;
// Checking if the calculation point exists in the x data
    for (int i = 0; i < n; i++) {
        if (fabs(xp - x[i]) < 0.0001) {
            index = i;
            flag = 1;
            break;
        }
    }
  if (flag == 0) {
        cout << "Invalid calculation point. Program exiting..." << endl;
        exit(0);
    }
// Generating Forward Difference Table
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < n - i; j++) {
            y[j][i] = y[j + 1][i - 1] - y[j][i - 1];
        }
    }
    h = x[1] - x[0];
// Applying the formula for forward difference to calculate derivatives
    for (int i = 1; i < n - index; i++) {
        term = y[index][i]/ i;
        sum += sign * term;
sign = -sign;
    }
// Dividing by h
    first_derivative = sum / h;
// Displaying the result
    cout << "First derivative at x = " << xp << " is " << first_derivative << endl;
    return 0;
}
```

Output:

```
Enter number of data:7
Enter Data:
x  y
1 2.7183
1.2 3.3201
1.4 4.0552
```

1.6 4.9530
1.8 6.0496
2 7.3891
2.2 9.0250
Enter at what value of x you want to calculate derivative: 1.2
First derivative at x = 1.2 is 3.32032

Result Analysis:

The given code computes the first derivative of a function f(x) using the forward difference method. Based on the input data:

- x values: {1, 1.2 ,1.4 ,1.6 ,1.8 ,2 ,2.2}
- y values: {2.7183,3.3201,4.0552,4.9530,6.0496,7.3891,9.0250}

The derivative is calculated at x=1.2

Computed Result

The output of the program is:
First derivative at x=1.2 is 3.32032.
The program uses the **forward difference method**, a numerical differentiation technique, to approximate the derivative. The derivative is calculated using the difference table, where successive differences of y-values are computed and divided by iii, where iii is the order of the difference. The input data represents an exponential function (f(x)=e^x and the true derivative of e^x at x=1.2 is: f′(1.2) ≈3.3201. The computed derivative 3.32032 closely matches the analytical derivative, with a small error of approximately 0.00022.

Conclusion:

The forward difference method successfully computes the first derivative with high accuracy for the given data. The result 3.32032 is very close to the true derivative 3.3201 validating the reliability of the method for smooth functions.
The approach works well for evenly spaced data and smooth functions like exe^xex. However, for irregular spacing or functions with high variability, other numerical methods (e.g., central differences or higher-order schemes) may be more suitable.

Problem 02: Problem solve by Numerical Integration

Theory:

Numerical integration is the process of approximating the integral of a function when an exact analytical integration is difficult or impossible. Instead of finding a closed-form solution, we compute the integral by summing values of the function at discrete points. This is especially useful for functions that are complex, discontinuous, or represented as discrete data points.

The goal of numerical integration is to approximate the definite integral of a function f(x) over an interval [a,b] :

The idea of Newton-Cotes formulas is to replace a complicated function or tabulated data with an approximating function that is easy to integrate.

$$I = \int_a^b f(x)dx \approx \int_a^b f_n(x)dx$$

where $f_n(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$.

Code:

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

/* Define function here */
#define f(x) 1/(1+x)

using namespace std;
int main()
{
   double lower, upper, integration=0.0, stepSize, k;
   int i, subInterval;
   /* Input */

   cout<<"Enter lower limit of integration: ";
   cin>>lower;
   cout<<"Enter upper limit of integration: ";
   cin>>upper;

   int choice=-1;
   while(choice!=4){

      cout<<"Enter the value of h(stepsize()):";
      cin>>stepSize;
      cout<<endl;

      cout<<"Choose Intregration Method:"<<endl;
      cout<<"1. Trapizoidal formula\n2. Simpon's Rule(1/3)\n3.  Simpon's Rule(3/8)\n4. Exit\n";
      cout<<"Enter your choice: ";
      cin>>choice;

      cout << fixed << setprecision(3);

      switch(choice){
         case 1:
            /* Calculation */
            /* Finding step size */
            subInterval = (upper - lower)/stepSize;
            /* Finding Integration Value */
            integration = f(lower) + f(upper);

            for(i=1; i<= subInterval-1; i++)
            {
             k = lower + i*stepSize;
```

```
            integration = integration + 2 * (f(k));
            }

            integration = integration * stepSize/2;

            cout<< endl<<"In trapizoidale rule Required value of integration is: "<< integration;
            break;
        case 2:
            /* Calculation */
/* Finding step size */
            subInterval = (upper - lower)/stepSize;

            /* Finding Integration Value */
            integration = f(lower) + f(upper);

            for(i=1; i<= subInterval-1; i++)
            {
            k = lower + i*stepSize;

            if(i%2==0)
            {
               integration = integration + 2 * (f(k));
            }
            else
            {
               integration = integration + 4 * (f(k));
            }

            }
           integration = integration * stepSize/3;

            cout<< endl <<"In Simpon's Rule(1/3) value of integration is: "<< integration;
            break;
        case 3:
            /* Calculation */
            /* Finding step size */
            subInterval = (upper - lower)/stepSize;

            /* Finding Integration Value */
            integration = f(lower) + f(upper);

            for(i=1; i<= subInterval-1; i++)
            {
             k = lower + i*stepSize;
```

```
            if(i%3==0)
            {
              integration = integration + 2 * (f(k));
            }
            else
            {
              integration = integration + 3 * (f(k));
            }

            }

            integration = integration * stepSize*3.0/8.0;

            cout<< endl <<"In Simpon's Rule(3/8) value of integration is: "<< integration;
            break;
        case 4:
            cout<<"Exiting program."<<endl;
            break;
        default:
            cout<<"Invail choice.Please try again."<<endl;
            break;
      }
      if(choice!=4){
        cout<<"\nPress Enter to continue....";
        cin.ignore();
        cin.get();
      }
   }

   return 0;
}
```

Output:

Enter lower limit of integration: 0
Enter upper limit of integration: 1
Enter the value of h(stepsize()):0.5

Choose Intregration Method:
1. Trapizoidal formula
2. Simpon's Rule(1/3)
3. Simpon's Rule(3/8)
4. Exit
Enter your choice: 1

In trapizoidale rule Required value of integration is: 0.708

Press Enter to continue....

Enter the value of h(stepsize()):0.25

Choose Intregration Method:
1. Trapizoidal formula
2. Simpon's Rule(1/3)
3. Simpon's Rule(3/8)
4. Exit
Enter your choice: 2

In Simpon's Rule(1/3) value of integration is: 0.693
Press Enter to continue....

Enter the value of h (stepsize()):0.125

Choose Intregration Method:
1. Trapizoidal formula
2. Simpon's Rule(1/3)
3. Simpon's Rule(3/8)
4. Exit
Enter your choice: 3

In Simpon's Rule(3/8) value of integration is: 0.685
Press Enter to continue....
Enter the value of h(stepsize()):0.125

Choose Intregration Method:
1. Trapizoidal formula
2. Simpon's Rule(1/3)
3. Simpon's Rule(3/8)
4. Exit
Enter your choice: 4
Exiting program.

Result Analysis:

The program uses numerical integration methods to approximate the value of the definite integral of the function f(x)=1/(1+x) over a given interval [a, b] . The methods implemented are:

1. Trapezoidal Rule
2. Simpson's Rule (1/3)
3. Simpson's Rule (3/8)

The results are computed for different step sizes h and are as follows:
1. Using Trapezoidal Rule:
  Step Size h=0.5: The approximate integral value is 0.708.

2. Using Simpson's Rule (1/3):
   Step Size h=0.25: The approximate integral value is 0.693.
3. Using Simpson's Rule **(3/8)**:
   Step Size (h=0.125): The approximate integral value is 0.6850

The exact value of the integral is $\ln(2) \approx 0.693$

Conclusion:

1. Trapezoidal Rule:Simple to implement but less accurate for functions with curvature like $f(x)=1/(1+x)$ It is suitable for rough estimates or large step sizes when computational efficiency is prioritized.
2. Simpson's Rule (1/3): The most accurate among the tested methods for this problem, especially for small step sizes. It effectively balances computational effort and accuracy, making it the recommended choice for smooth functions.
3. Simpson's Rule (3/8): This method is effective for specific problems, but in this case, it performed slightly less accurately than Simpson's Rule (1/3). Its higher order of polynomial fitting may not always guarantee better results for every function or step size.