Index

Lab Report 1

Error & Best Approximation Analysis

by **Kefaet Ullah (2103011)**

to **Shyla Afroge(Associate Professor, CSE, RUET)**

Task 1

Problem Statement: Calculate the absolute error (E_a), relative error (E_r), and percentage error (E_p) when estimating the sum of $\sqrt{2}+\sqrt{5}+\sqrt{7}$ using approximate values of $\sqrt{2}$, $\sqrt{5}$, and $\sqrt{7}$. The task requires to understand error propagation in numerical approximations.

Theory: In Numerical Analysis, errors occur when exact values are approximated for ease of computation. The errors can be classified as follows:

1. Absolute Error (Ea): $E_a=|S-S_1|$,
   where S is the true value and S_1 is the approximated value.
2. Relative Error (Er): $E_r=\frac{Ea}{S}$,
   This measures the size of the error relative to the true value.
3. Percentage Error (Ep): $E_p=E_r\times 100$,
   It is the relative error expressed as a percentage.

Code:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    float s = sqrt(2) + sqrt(5) + sqrt(7);
    float x1 = 1.414213;
    float x2 = 2.236068;
    float x3 = 2.645751;
    float s1 = x1 + x2 + x3;
    float Ea = abs(s - s1);
    float Er = Ea / s;
    float Ep = Er * 100;
    cout << fixed << setprecision(6) << "Absoluter Error : " << Ea << endl << "Relative
Error : " << Er << endl << "Percentage Error : " << Ep << "%" << endl;
    return 0;
}
```

Output:

```
Output
Absoluter Error : 0.000001
Relative Error : 0.000000
Percentage Error : 0.000015%
```

Discussion & Conclusion: This program demonstrates the application of **error analysis** in numerical computation. It highlights the importance of choosing precise approximations to minimize errors. The absolute error shows the magnitude of the deviation, while the relative and percentage errors provide normalized measures to compare the errors relative to the true value.

Task 2

Problem Statement: Given the exact value of $\frac{1}{3}$ identify the best approximation from a set of predefined values *[0.30,0.31,0.32,0.33,0.34]* by calculating the absolute error for each approximation. Determine the smallest absolute error and its corresponding approximation.

Theory: Approximations are commonly used in numerical computations to represent values that are irrational or difficult to handle precisely. The **absolute error** quantifies the difference between the exact value and its approximation:

$$Absolute\ Error = |x - approximation|\ ,$$

Where:

- x is the exact value ($\frac{1}{3}$ in this case).
- approximation is the value being compared to x.

The smallest absolute error indicates the best approximation among the given values. This method is useful in selecting the most accurate representation of a value under constrained precision.

Code:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    float x = (1*1.0)/3;
    vector <float> approx = {0.30,0.31,0.32,0.33,0.34};
    vector <float> error;
    for (float i : approx){
        error.push_back(abs(x-i));
    }
    cout << "Minimum approx error: " << *min_element(error.begin(),error.end()) << endl;
    int index = 0 ;
    for (float i : error)
    {
        if (i == *min_element(error.begin(),error.end()))
        {
            break;
        }
        index++;
    }
    cout << "Best approximation: " << approx[index] << endl;
    return 0;
}
```

Output:

```
Output
Minimum approx error: 0.00333333
Best approximation: 0.33
```

Discussion & Conclusion: From the given set of approximations, the program calculates the absolute error for each value and identifies the approximation that is closest to $\frac{1}{3}$. The smallest error value highlights the most accurate approximation among the candidates. This approach ensures an optimal choice for representing the exact value.

Lab Report 2

Root-Finding Techniques: Bisection, False Position, and Iterative Method

by **Kefaet Ullah (2103011)**

to **Shyla Afroge(Associate Professor, CSE, RUET)**

Task 1

Problem Statement: Solve a polynomial equation $P(x)=0$ using three numerical methods: **Bisection Method**, **False Position Method**, and **Iteration Method**. Implement the methods to find the root of the polynomial with a precision of ≤ 0.001.

Theory: Finding roots of a polynomial involves solving P(x)=0, where P(x) is a polynomial of degree n. The following numerical methods are used:

1. Bisection Method: This method works by repeatedly halving an interval [a,b] such that P(a)·P(b)<0. The root is approximated as:

   $$x_r = \frac{a+b}{2},$$

   The process stops when the error $|x_r^{(i)} - x_r^{(i-1)}| \leq 0.001$
1. False Position Method: This is similar to the Bisection Method but uses a weighted approximation for x_r based on the values of P(a) and P(b):

   $$x_r = \frac{a.P(b) - b.P(a)}{P(b) - P(a)},$$

2. Iteration Method: An iterative formula $x_{i+1} = g(x_i)$ is derived from the polynomial, where:

   $$x_{i+1} = \left(\frac{-c}{P'(xi)}\right)^{1/n},$$

   The process iterates until the error $|x_{i+1} - x_i| \leq 0.001$.

Code:

```
#include <bits/stdc++.h>
using namespace std;
float poly(const vector<float> &coeff, int n, float x){
    float result = 0.0;
    for (int i = 0; i <= n; i++){
        result += coeff[i] * pow(x, n - i);
    }
    return result;
}
float bisection(vector<float> coeff, int n){
    float a, b;
    cout << "Approximation" << endl;
    while (1){
        cout << "a : ";
        cin >> a;
        cout << "b : ";
        cin >> b;
        if (poly(coeff, n, a) * poly(coeff, n, b) < 0){
            break;
        }
        else{
            cout << "Invalid input of a or b" << endl << "Try again" << endl;
        }
    }
    float xr = 0;
    int count = 0;
    float error = 0.0;
```

```
    int max_iterations = 100;
    while (count < max_iterations){
        float temp = xr;
        xr = (a + b) / 2;
        if (count > 0){
            error = abs(xr - temp);
        }
        cout << count << " " << a << " " << b << " " << xr << " " << error << endl;
        count++;
        float f_xr = poly(coeff, n, xr);
        if (f_xr == 0 || (error <= 0.001 && count > 1)){
            break;
        }
        if (poly(coeff, n, a) * f_xr < 0){
            b = xr;
        }
        else{
            a = xr;
        }
    }
    return xr;
}
float false_position(vector<float> coeff, int n){
    float a, b;
    cout << "Approximation" << endl;
    while (1){
        cout << "a : ";
        cin >> a;
        cout << "b : ";
        cin >> b;
        if (poly(coeff, n, a) * poly(coeff, n, b) < 0){
            break;
        }
        else{
            cout << "Invalid input of a or b" << endl << "Try again" << endl;
        }
    }
    float xr = a;
    int count = 0;
    float error = 0.0;
    int max_iterations = 100;
    while (count < max_iterations){
        float f_a = poly(coeff, n, a);
        float f_b = poly(coeff, n, b);
        float temp = xr;
        xr = (a * f_b - b * f_a) / (f_b - f_a);
        if (count > 0){
```

```
            error = abs(xr - temp);
        }
        cout << count << " " << a << " " << b << " " << xr << " " << error << endl;
        count++;
        float f_xr = poly(coeff, n, xr);
        if (f_xr == 0 || (error <= 0.001 && count > 1)){
            break;
        }
        if (f_a * f_xr < 0){
            b = xr;
        }
        else{
            a = xr;
        }
    }
    return xr;
}
float func(vector <float> coeff, int n, float x){
    float result = 0.0;
    for (int i = 1; i < n; i++){
        result += coeff[i] * pow(x, n - i);
    }
    return pow((coeff[coeff.size()-1]/result),1/n);
}
float iteration(vector<float> coeff, int n){
    float initial;
    cout << "Initial guess : ";
    cin >> initial;
    float xr = initial;
    int count = 0;
    float error = 0.0;
    int max_iterations = 100;
    while (count < max_iterations){
        float temp = xr;
        xr = func(coeff,n,temp);
        error = abs(xr-temp);
        cout << count << " " << temp << " " << xr << " " << error << endl;
        count++;
        if (error <= 0.001) {
            break;
        }
    }
    return xr;
}
int main(){
    cout << "Order : ";
    int n;
```

```
    cin >> n;
    vector<float> coeff(n + 1);
    cout << "The form of the equation" << endl
         << "ax^(n) + bx^(n-1) + ..... + ex + z = 0" << endl
         << "Enter the co-efficients" << endl;
    for (int i = 0; i <= n; i++){
        cin >> coeff[i];
    }
    while (1){
        cout << "Choose your method" << endl
             << "1. Bisection\n2. False Position\n3. Iteration Method\n0. Exit" << endl;
        int choose;
        cin >> choose;
        if (choose == 1){
            cout << "Root : " << bisection(coeff, n) << endl;
        }
        else if (choose == 2){
            cout << "Root : " << false_position(coeff, n) << endl;
        }
        else if (choose == 3){
            cout << "Root : " << iteration(coeff,n) << endl;
        }
        else{
            break;
        }
    }
    return 0;
}
```

Output:

```
Output
Order : 3
The form of the equation
ax^(n) + bx^(n-1) + ..... + ex + z = 0
Enter the co-efficients
1 0 -2 -5
Bisection:
Approximation
a : 2
b : 3
0 2 3 2.5 0
1 2 2.5 2.25 0.25
2 2 2.25 2.125 0.125
3 2 2.125 2.0625 0.0625
4 2.0625 2.125 2.09375 0.03125
5 2.09375 2.125 2.10938 0.015625
6 2.09375 2.10938 2.10156 0.0078125
7 2.09375 2.10156 2.09766 0.00390625
8 2.09375 2.09766 2.0957 0.00195312
9 2.09375 2.0957 2.09473 0.000976562
Root : 2.09473
```

```
False Position:
Approximation
a : 2
b : 3
0 2 3 2.05882 0
1 2.05882 3 2.08126 0.0224402
2 2.08126 3 2.08964 0.00837541
3 2.08964 3 2.09274 0.0031004
4 2.09274 3 2.09388 0.00114417
5 2.09388 3 2.09431 0.000421762
Root : 2.09431

Iteration:
Initial guess : 0.75
0.755929
0 0.75 0.755929 0.00592893
0.754652
1 0.755929 0.754652 0.00127727
0.754926
2 0.754652 0.754926 0.000274599
Root : 0.754926
```

Discussion & Conclusion: The three methods are effective for finding roots of a polynomial:

- The Bisection Method is simple and guarantees convergence, though it may be slower.
- The False Position Method converges faster for some problems but may fail in cases of skewed intervals.
- The Iteration Method is fast but requires careful derivation of *g(x)* and an appropriate initial guess to ensure convergence.

This approach allows flexibility in choosing a method based on the polynomial and the desired precision.

Lab Report 3

Root-Finding Techniques: Newton-Raphson and Ramanujan's Method

by **Kefaet Ullah (2103011)**

to **Shyla Afroge(Associate Professor, CSE, RUET)**

Task 1

Problem Statement: Given a polynomial equation, find its root using two numerical techniques: the Newton-Raphson method and Ramanujan's method.

Theory:

1. **Newton-Raphson Method:** The Newton-Raphson method is an iterative approach to finding the root of a real-valued function. Given a function *f(x)*, the iteration formula is:

$$x_{r+1}=x_r-\frac{f(xr)}{f'(xr)},$$

x_r is the current approximation, $f'(x)$ is the derivative of $f(x)$.
The method converges quadratically, provided the initial guess is close to the actual root and $f'(x)\neq 0$.

3. **Ramanujan's Method**: This method leverages coefficients of the polynomial to iteratively approximate the roots. For a polynomial $f(x)=c_0+c_1x+c_2x^2+\ldots+c_nx^n$, Ramanujan's approach involves:
    - Rearranging coefficients to form sequences b_n using a recurrence relation.
    - Computing the ratios b_{n-1}/b_n, which converge to the root as iterations progress.

   Both methods aim to achieve convergence within a specified tolerance ϵ.

Code:

```
import sympy as sp
def newton_raphson(f_x, tol=1e-3, max_iter=100):
    x0 = float(input("Enter the initial guess: "))
    x = sp.symbols('x')
    f = sp.lambdify(x, f_x)
    df_expr = sp.diff(f_x, x)
    df = sp.lambdify(x, df_expr)
    x_r = x0
    print(f"Iteration | x_r")
    for i in range(max_iter):
        x_new = x_r - f(x_r) / df(x_r)
        print(f"{i+1}  {x_r:.6f}")
        if abs(x_new - x_r) < tol:
            return x_new
        x_r = x_new
    return x_r
def ramanujan(f_x,tol=1e-3,max_iter=100):
    x=sp.symbols('x')
    coeff = sp.Poly(f_x, x).all_coeffs()
    coeff = [float(coef) for coef in coeff]
    # print(coeff)
    c = coeff[-1]
    a = [float(coef/c*-1) for coef in coeff]
    partial_a = a[:-1]
    reverse_partial_a = partial_a[::-1]
    b = [1.0]
    ratio_list = [0.0]
    print(reverse_partial_a)
    i=1
```

```
    while i <= max_iter:
        new_b = 0
        for j in range(i):
            if i - j - 1 < len(reverse_partial_a):
                new_b += reverse_partial_a[i - j - 1] * b[j]
        b.append(new_b)
        ratio = b[i-1] / b[i]
        ratio_list.append(ratio)
        print(f"{i}  {ratio:.6f}")
        if abs(ratio_list[i]-ratio_list[i-1]) < tol :
            return ratio
        i += 1
x = sp.symbols('x')
f_x = x**3 - 6*x**2 + 11*x - 6
while 1:
    print("The method you want\n1.Newton Raphson\n2.Ramanujan\n3.Exit")
    choice = int(input())
    if choice==1:
        root = newton_raphson(f_x)
        print(f"\nRoot is approximately: {root:.6f}")
    elif choice == 2:
        root =  ramanujan(f_x)
        print(f"\nRoot is approximately: {root:.6f}")
    else:
        break
```

Output:

```
Output
Absoluter Error : 0.000001
Relative Error : 0.000000
Percentage Error : 0.000015%
```

Discussion & Conclusion: The Newton-Raphson and Ramanujan's methods effectively approximate roots of polynomial equations. Newton-Raphson provides rapid convergence when a good initial guess is supplied, while Ramanujan's method systematically leverages polynomial coefficients, offering an alternative for specific cases. Both approaches highlight the power of iterative numerical techniques in solving real-world mathematical problems.

Lab Report 4

Least Square Curve Fitting

by **Kefaet Ullah (2103011)**

to **Shyla Afroge(Associate Professor, CSE, RUET)**

Task 1

Problem Statement: Given a dataset of randomly generated *x* and *y* values, determine the relationship between them by fitting:

1. A linear regression model of the form $y=a_0+a_1x$.
2. A non-linear regression model of the form $y=ae^{a_1x}$

Use matrix methods for solving the linear regression equation and logarithmic transformation for the non-linear model.

Theory:

1. Linear Regression: Linear regression models a straight-line relationship between *x* and *y*, expressed as:

   $$y=a_0+a_1x$$

   To determine a_0 (intercept) and a_1 (slope), the normal equations are derived as follows:

   $$n\cdot a_0+(\Sigma x)\cdot a_1=\Sigma y$$
   $$(\Sigma x)\cdot a_0+(\Sigma x^2)\cdot a_1=\Sigma(x\cdot y)$$

   These equations are solved using matrix algebra for a_0 and a_1.
2. Non-Linear Regression (Exponential Model): The exponential model is expressed as:

   $$y=ae^{a_1x}$$

   Taking the natural logarithm on both sides linearizes the model:

   $$lny = lna+a_1x$$

   let $y_2=lny$. This reduces the problem to a linear regression on y_2:

   $$y_2=a_0+a_1x$$

   Where $a_0=lna$. After determining a_0, compute a as:

   $$a = e^{a_0}$$

Both methods provide insights into the relationship between *x* and *y*, with the exponential model capturing non-linear growth patterns.

Code:

```
import random
import math
import numpy as np
n = int(input("Enter the size of the lists: "))
x = [random.uniform(1, 50) for _ in range(n)]
y = [random.uniform(1, 50) for _ in range(n)]
x_sum = sum(x)
y_sum = sum(y)
x_square = [i**2 for i in x]
x_square_sum = sum(x_square)
x_y = [x[i]*y[i] for i in range(n)]
x_y_sum = sum(x_y)
a1=n
b1=x_sum
c1=y_sum
a2=x_sum
b2=x_square_sum
c2=x_y_sum
A = np.array([[a1, b1], [a2, b2]])
C = np.array([c1, c2])
```

```
if np.linalg.det(A) != 0:
    solution = np.linalg.solve(A, C)
    x, yx = solution
    print(f"y = {x} + ({yx})x")
else:
    print("No Solution.")
y2 = [math.log(i) for i in y]
sum_y2 = sum(y2)
avg_y = sum_y2/n
avg_x = x_sum/n
a_1 = ((n*x_y_sum)-(x_sum*avg_y))/(n*x_square_sum-x_sum**2)
a_0 = avg_y - a_1*avg_x
a = math.exp(a_0)
print(f"y = {a}e^{a_1}x")
```

Output:

```
Output
Enter the size of the lists: 10
y = 34.08504317078141 + (-0.3257482741222965)x
y = 1.8332493122539838e-38e^3.504640957340287x
```

Discussion & Conclusion: The program implements two regression techniques:

1. **Linear Regression:** Solves for a_0 and a_1 using matrix algebra, yielding a simple linear relationship between x and y.
2. **Non-Linear Regression:** Fits an exponential growth model using logarithmic transformation, demonstrating the suitability of the model for non-linear data patterns.

These regression models highlight the flexibility of mathematical techniques in analyzing diverse relationships in data.

Lab Report 5

Numerical Integration

by **Kefaet Ullah (2103011)**

to **Shyla Afroge(Associate Professor, CSE, RUET)**

Task 1

Problem Statement: Given a function *f(x),* integrate it numerically over the interval [a,b] using the following methods:

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

The program approximates the definite integral by dividing the interval *[a,b]* into *n* subintervals of width *h*, then applies the respective rules for numerical integration.

Theory: Numerical integration techniques are used to approximate definite integrals when an exact analytical solution is challenging to compute.

1. **Trapezoidal Rule:** The Trapezoidal Rule approximates the area under the curve by dividing it into trapezoids:

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right]$$

   where $h = \frac{b-a}{n}$ is the width of each subinterval, and $x_i = a + i \cdot h$

2. **Simpson's 1/3 Rule:** This rule uses parabolic segments to approximate the curve and requires an even number of subintervals:

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(a) + 4\sum_{\text{odd indices}} f(x_i) + 2\sum_{\text{even indices}} f(x_i) + f(b)\right]$$

   This method provides higher accuracy for smooth functions.

3. **Simpson's 3/8 Rule:** This rule divides the interval into segments of three subintervals (multiple of 3):

$$\int_a^b f(x)dx \approx \frac{3h}{8}\left[f(a) + 3\sum_{\text{not divisible by 3}} f(x_i) + 2\sum_{\text{divisible by 3}} f(x_i) + f(b)\right]$$

   It is particularly useful when the number of subintervals is a multiple of 3.

Code:

```
a=float(input())
b=float(input())
h=float(input())
n=int((b-a)/h)
x_values = [a+i*h for i in range (n+1)]
y_values = [1 / (x + 1) if (x + 1) != 0 else None for x in x_values]
def trapezoidal_rule(x_values, y_values, h):
    result = y_values[0] + 2 * sum(y_values[1:-1]) + y_values[-1]
    result = result * 0.5 * h
    return result
def simson_rule_1(x_values, y_values, h):
    # Sum the odd-indexed and even-indexed terms correctly
    odd_sum = sum(y_values[1:-1:2])  # terms at index 1, 3, 5, ...
    even_sum = sum(y_values[2:-1:2])  # terms at index 2, 4, 6, ...
    result = y_values[0] + 4*odd_sum + 2*even_sum + y_values[-1]
    result *= h / 3  # Multiply by h/3 as per the formula
    return result
```

```
def simson_rule_2(x_values,y_values,h):
    div_3 = 0
    div_not_3  = 0
    for i in range(1,len(y_values)):
        if i % 3 == 0:
            div_3 += y_values[i]
        else:
            div_not_3 += y_values[i]
    result = y_values[0]+ 3*div_not_3 + 2*div_3 + y_values[-1]
    result = result * (3/8) * h
    return result
way_1 = trapezoidal_rule(x_values, y_values,h)
way_2 = simson_rule_1(x_values, y_values,h)
way_3 = simson_rule_2(x_values, y_values,h)
print(way_1)
print(way_2)
print(way_3)
```

Output:

Input

```
0
1
0.5
```

Output

```
0.783333333333
0.694444444443
0.9375
```

Discussion & Conclusion: The program demonstrates the application of three popular numerical integration methods:

1. **Trapezoidal Rule** provides a simple and quick approximation, though it might be less accurate for functions with high curvature.
2. **Simpson's 1/3 Rule** improves accuracy by approximating the curve with parabolas, ideal for even subintervals.
3. **Simpson's 3/8 Rule** offers another robust approach, particularly suited for cases where the interval count is a multiple of 3.

These methods highlight the trade-offs between computational effort and accuracy, offering versatile tools for practical problems in engineering, physics, and other sciences.

Lab Report 7

Numerical Differentiation

by **Kefaet Ullah (2103011)**

to **Shyla Afroge(Associate Professor, CSE, RUET)**

Task 1

Problem Statement: Given a set of equally spaced data points (x_i,y_i), estimate the **first derivative** and **second derivative** of a function $f(x)$ at a specific point x_{target}. Use **Newton's Forward Difference Method** to approximate these derivatives based on forward difference tables.

Theory: Numerical differentiation is a method to approximate the derivative of a function based on discrete data points. When the function f(x) is not explicitly known, we use difference formulas derived from interpolating polynomials.

Newton's Forward Difference Formula:

The forward difference table is constructed to represent the difference relations among the y-values of the data points. Let h represent the spacing between consecutive x-values $(h=x_{i+1}-x_i)$.

The derivatives can then be approximated using the forward differences:

1. First Derivative $(f'(x))$: The approximation at $x=x_0+u\cdot hx$ (where u = $\frac{x-x_0}{h}$) is:

$$f'(x) \approx \frac{1}{h}\left[\Delta y_0 + \frac{(2u-1)}{2!}\Delta^2 y_0 + \frac{(3u^2-6u+2)}{3!}\Delta^3 y_0\right]$$

2. Second Derivative $(f''(x))$: The approximation at $x=x_0+u\cdot hx$ is:

$$f''(x) \approx \frac{1}{h^2}\left[\Delta^2 y_0 + \frac{(6u-6)}{3!}\Delta^3 y_0\right]$$

Where: $\Delta y_0, \Delta^2 y_0, \Delta^3 y_0 \ldots$ are the forward differences derived from the difference table.

This method is most effective when:

1. Data points are equally spaced.
2. The target point x_{target} is near the start of the interval (close to x_0).

Code:

```
import numpy as np
def forward_difference_table(y_values):
    n = len(y_values)
    table = np.zeros((n, n))
    table[:, 0] = y_values
    for j in range(1, n):
        for i in range(n - j):
            table[i, j] = table[i + 1, j - 1] - table[i, j - 1]
    return table
def newton_forward_derivative(x_values, y_values, x):
    h = x_values[1] - x_values[0]
    table = forward_difference_table(y_values)
    u = (x - x_values[0]) / h
    derivative = (1 / h) * (table[0, 1] + (2 * u - 1) * table[0, 2] / 2 + (3 * u**2 - 6 *
u + 2) * table[0, 3] / 6)
    second_derivative = (1 / h**2) * (table[0, 2] + (6 * u - 6) * table[0, 3] / 6)
    result = [derivative,second_derivative]
    return result
x = [1, 1.2, 1.4, 1.6, 1.8,2.0,2.2]
y = [2.7183, 3.3201, 4.0552, 4.9530, 6.0496,7.3891,9.0250]
```

```
x_target = 1.2
derivatives = newton_forward_derivative(x, y, x_target)

print(derivatives)
```

Output:

```
Output
[3.3177500000000006, 3.3325000000000067]
```

Discussion & Conclusion: The code implements **Newton's Forward Difference Method** to calculate the first and second derivatives of a function represented by discrete data points.

- The **first derivative** provides the slope or rate of change of the function at a given point.
- The **second derivative** describes the concavity or the rate of change of the slope.

This method is particularly useful for approximating derivatives when analytical methods are infeasible. The results highlight the power of numerical techniques in analyzing real-world problems with discrete data.