CSE 4746 Numerical Methods Lab
Lab Assignment-1

MD Faisal Hoque Rifat
ID: C221076

March 30, 2025

1. Count Number of Significant Digits

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

int countSignificantDigits(string number)
{
    int count = 0;
    bool significantDigitEncountered = false;

    for (char ch : number)
    {
        if (isdigit(ch))
        {
            if (ch != '0' || significantDigitEncountered)
            {
                significantDigitEncountered = true;
                count++;
            }
        }
    }

    return count;
}

int main()
{
    string number;
    cout << "Enter a number: ";
    cin >> number;
    cout << "Number of significant digits: " <<
        countSignificantDigits(number) << endl;
    return 0;
}
```

GitHub Link: Click here to view on GitHub

2. Round Off a Number with n Significant Figures Using Banker's Rule

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

double roundToSignificantFigures(double num, int n)
{
    if (num == 0.0) return 0.0;

    double d = ceil(log10(num < 0 ? -num : num));
    int power = n - static_cast<int>(d);

    double magnitude = pow(10.0, power);
    long shifted = round(num * magnitude);

    return shifted / magnitude;
}

int main()
{
    double number;
    int n;
    cout << "Enter a number: ";
    cin >> number;
    cout << "Enter number of significant figures: ";
    cin >> n;
    cout << "Rounded number: " << roundToSignificantFigures(number, n) << endl;
    return 0;
}
```

GitHub Link: Click here to view on GitHub

3. Evaluate Polynomial Using Horner's Rule

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

int evaluatePolynomial(int x)
{
    return (x*(x*(x-2)+5)+10);
}

```

```cpp
int main()
{
    int x = 5;
    cout << "f(5) = " << evaluatePolynomial(x) << endl;
    int a3 = 1;
    int a2 = -2;
    int a1 = 5;
    int a0 = 10;
    int p3 = a3;
    int p2 = p3*x + a2;
    int p1 = p2*x + a1;
    int p0 = p1*x + a0;
    cout << "f(5) = " << p0 << endl;
    return 0;
}
```

GitHub Link: Click here to view on GitHub

4. Root of Equation Using Bisection Method

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

double f(double x)
{
    return x * x * x - 9 * x + 1;
}

double bisection(double a, double b, double tol)
{
    double c;
    while ((b - a) / 2.0 > tol)
    {
        c = (a + b) / 2.0;
        if (f(c) == 0.0) break;
        else if (f(c) * f(a) < 0) b = c;
        else a = c;
    }
    return c;
}

int main()
{
    double a = 0, b = 1, tol = 0.001;
    cout << "Root: " << bisection(a, b, tol) << endl;
    return 0;
}
```

GitHub Link: Click here to view on GitHub

5. All Roots Using Bisection Method

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

double f(double x)
{
    return x * x * x - 6 * x + 4;
}

double bisection(double a, double b, double tol)
{
    double c;
    while ((b - a) / 2.0 > tol)
    {
        c = (a + b) / 2.0;
        if (f(c) == 0.0) break;
        else if (f(c) * f(a) < 0) b = c;
        else a = c;
    }
    return c;
}

int main()
{
    double tol = 0.001;
    double lower = -100, upper = 100, x = 1.0;///boundary and increment

    double x2 = lower, x1 = lower;
    int i = 1;
    while(x2 < upper)
    {
        x1 = lower, x2 = lower + x;
        double f1 = f(x1),f2 =f(x2);
        lower = x2 + 0.000001;
        if((f1 * f2) > 0)
        {
            continue;
        }
        cout << " Root "<< i <<" : "<< bisection(x1, x2, tol) << endl ;
        i++;
    }
    return 0;
}
```

GitHub Link: Click here to view on GitHub

6. Root Using Newton-Raphson Method

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

double f(double x)
{
    return x * x * x - 6 * x + 4;
}

double f_prime(double x)
{
    return 3 * x * x - 6;
}

double newtonRaphson(double x0, double tol)
{
    double x1;
    while (true)
    {
        x1 = x0 - f(x0) / f_prime(x0);
        if (fabs(x1 - x0) < tol)
            break;
        x0 = x1;
    }
    return x1;
}

int main()
{
    double x0 = 0, tol = 0.001;
    cout << "Root: " << newtonRaphson(x0, tol) << endl;
    return 0;
}
```

GitHub Link: Click here to view on GitHub

7. Root Using False Position Method

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

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

double falsePosition(double x1, double x2, double tol)
```

```
{
    double x0;
    while ((x2 - x1) > tol)
    {
        x0 = x1 - (f(x1) * (x2 - x1)) / (f(x2) - f(x1));
        if (f(x0) == 0.0)
            break;
        else if (f(x0) * f(x1) < 0)
            x2 = x0;
        else
            x1 = x0;
    }
    return x0;
}

int main()
{
    double x1 = -2, x2 = 2, tol = 0.001;
    cout << "Root: " << falsePosition(x1, x2, tol) << endl;
    return 0;
}
```

GitHub Link: Click here to view on GitHub

8. Root Using Secant Method

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

double f(double x)
{
    return x * x * x - 5 * x * x - 29;
}

double secant(double x1, double x2, double tol)
{
    double x3;
    while (true)
    {
        x3 = x2 - (f(x2) * (x2 - x1)) / (f(x2) - f(x1));
        if (fabs(x2 - x1) < tol)
            break;
        x1 = x2;
        x2 = x3;
    }
    return x2;
}

```

```cpp
int main()
{
    double x1 = 4, x2 = 2, tol = 0.001;
    cout << "Root: " << secant(x1, x2, tol) << endl;
    return 0;
}
```

GitHub Link: Click here to view on GitHub

9. Quotient Polynomial

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

vector<int> syntheticDivision(vector<int>& a, int root, int
    n)
{
    vector<int> q(n+1);
    q[0] = 0;
    for(int i = 1; i<=n; i++)
    {
        q[i] = a[i-1] + q[i-1]*root;
    }
    return q;
}

int main()
{
    int n = 3;
    vector<int> a = {1, -5, 10, -8}; // x^3 - 5x^2 + 10x - 8
    int root = 2;
    vector<int> q = syntheticDivision(a, root, n);

    cout << "Quotient polynomial: ";
    for (int coeff : q)
    {
        cout << coeff << " ";
    }
    cout << endl;
    return 0;
}
```

GitHub Link: Click here to view on GitHub

10. All Roots Using Newton-Raphson Method with Deflation

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

double evaluatePolynomial(const vector<double>& coeffs, double x)
{
    double result = 0.0;
    for (int i = 0; i < coeffs.size(); i++)
    {
        result += coeffs[i] * pow(x, coeffs.size() - 1 - i);
    }
    return result;
}

double evaluateDerivative(const vector<double>& coeffs, double x)
{
    double result = 0.0;
    for (int i = 0; i < coeffs.size() - 1; i++)
    {
        int power = coeffs.size() - 1 - i;
        result += power * coeffs[i] * pow(x, power - 1);
    }
    return result;
}

vector<double> syntheticDivision(const vector<double>& coeffs, double root)
{
    vector<double> newCoeffs(coeffs.size() - 1);
    newCoeffs[0] = coeffs[0];

    for (int i = 1; i < newCoeffs.size(); i++)
    {
        newCoeffs[i] = newCoeffs[i-1] * root + coeffs[i];
    }

    return newCoeffs;
}

double newtonRaphson(const vector<double>& coeffs, double x0, double E)
{
    double xr = x0;
    double prev_xr;
    int iterations = 0;
    const int max_iterations = 100;

    while (fabs(xr - prev_xr) > E)
```

```
    {
        prev_xr = xr;
        double fx = evaluatePolynomial(coeffs, xr);
        double dfx = evaluateDerivative(coeffs, xr);

        if (fabs(dfx) < 1e-10)
        {
            cout << "Derivative too small. Trying a
                different initial guess." << endl;
            xr += 0.5;
            continue;
        }

        xr = xr - fx / dfx;
        iterations++;

        if (iterations > max_iterations)
        {
            cout << "Maximum iterations reached." << endl;
            break;
        }
    }

    return xr;
}

int main()
{
    // Original polynomial: x^3 - 6x + 4 = 0
    vector<double> coefficients = {1, 0, -6, 4};

    double E = 0.001;
    double x0 = 1.0;

    vector<double> roots;
    vector<double> currentCoeffs = coefficients;
    int n = currentCoeffs.size() - 1;

    cout << fixed << setprecision(3);

    while (n > 1)
    {
        double root = newtonRaphson(currentCoeffs, x0, E);
        roots.push_back(root);

        currentCoeffs = syntheticDivision(currentCoeffs,
            root);
        n = currentCoeffs.size() - 1;

        x0 = root;
```

```
    }

    double lastRoot = -currentCoeffs[1] / currentCoeffs[0];
    roots.push_back(lastRoot);

    cout << "Roots of the polynomial x^3 - 6x + 4 = 0 are:"
        << endl;
    for (int i = 0; i < roots.size(); ++i)
    {
        cout << "Root " << i+1 << ": " << roots[i] << endl;
    }

    return 0;
}
```

GitHub Link: Click here to view on GitHub