Numerical-Analysis-Root-Methods

Overview

- Purpose: A curated set of classic numerical methods implemented in C++ with clear theory notes, sample inputs, and generated outputs.
- What's inside: Linear systems (Gauss, Gauss–Jordan, LU, Matrix Inversion), nonlinear root finding (Bisection, False Position, Secant, Newton–Raphson), interpolation (Newton forward/backward/divided differences), numerical differentiation (forward/backward), numerical integration (Simpson's 1/3 and 3/8 rules), ODE solving (Runge–Kutta 4th order), and curve fitting (linear, polynomial, transcendental).
- How to run: Each method folder contains an `input.txt` and writes to `output.txt`. Build the corresponding C++ file (e.g., with MinGW g++ or Code::Blocks) and run in the same directory.
- Requirements: A C++ compiler (e.g., g++/MinGW) and standard library; no external dependencies.

Methods at a Glance

Domain	Methods	When to use	Input/Notes
Linear Systems	Gauss Elimination, Gauss–Jordan, LU, Matrix Inversion	Solve AX = B; LU best for repeated solves with same A; Gauss–Jordan for inverse/RREF	Augmented matrix in input.txt; writes echelon/solution to output.txt
Nonlinear Roots (Bracketing)	Bisection, False Position	$f(a) \cdot f(b) < 0$; guaranteed convergence; robust but slower	Polynomial coefficients and interval [a, b]
Nonlinear Roots (Open)	Secant, Newton–Raphson	Faster; need good initial guess(es); Newton needs derivative; may fail if poorly seeded	Coefficients and initial guess(es)

Domain	Methods	When to use	Input/Notes
Interpolation	Newton Forward, Newton Backward, Divided Difference	Estimate y at intermediate x; forward/backward for equally spaced ends; divided difference for uneven spacing	Tabulated (x, y) pairs
Numerical Differentiation	Forward/Backward (from interpolation)	Approximate f'(x), f''(x) from tables; requires step size h	Equally spaced data; uses Δ/ tables
Numerical Integration	Simpson's 1/3, Simpson's 3/8	Definite integrals; 1/3: n even; 3/8: n multiple of 3	Polynomial + [a, b], n subintervals
ODE (Initial Value Problem)	Runge–Kutta (RK4)	y' = f(x, y) with step h; good accuracy per step	x0, y0, h, target xn
Curve Fitting	Least Squares (Linear, Polynomial, Transcendental)	Fit models to data; transform when needed (log/ln)	(x, y) data pairs

Table of Contents

        * Code
        * Input
        * Output
- Solution of Non-Linear Equations
  - Bisection Method
    * Theory
    * Code
    * Input
    * Output
  - False Position Method
    * Theory
    * Code
    * Input
    * Output
  - Secant Method
    * Theory
    * Code
    * Input
    * Output
  - Newton Raphson Method
    * Theory
    * Code
    * Input
    * Output
- Solution of Interpolation
  - Newton's Forward Interpolation Method
    * Theory
    * Code
    * Input
    * Output
  - Newton's Backward Interpolation Method
    * Theory
    * Code
    * Input
    * Output
  - Divided Difference Method
    * Theory
    * Code
    * Input
    * Output
- Solution of Curve Fitting Model
  - Least Square Regression Method For Linear Equations
    * Theory
    * Code
    * Input
    * Output

  - Least Square Regression Method For Transcendental Equations
    - Theory
    - Code
    - Input
    - Output
  - Least Square Regression Method For Polynomial Equations
    - Theory
    - Code
    - Input
    - Output
- Solution of Differential Equations
  - Runge Kutta Method
    - Theory
    - Code
    - Input
    - Output
  - Numerical Differentiation by Forward Interpolation Method
    - Theory
    - Code
    - Input
    - Output
  - Numerical Differentiation by Backward Interpolation Method
    - Theory
    - Code
    - Input
    - Output
- Solution of Numerical Integrations
  - Simpson's One-Third Rule
    - Theory
    - Code
    - Input
    - Output
  - Simpson's Three-Eighths Rule
    - Theory
    - Code
    - Input
    - Output

Solution of Linear Equations

Gauss Elimination Method

Gauss Elimination Theory

Method used Gauss Elimination Method

Objective To solve a system of linear algebraic equations by transforming it into an **upper triangular system**, followed by **back substitution**.

Data Requirement A system of `n` linear equations:

```
a x  + a x  + ... + a x  = b
a x  + a x  + ... + a x  = b
...
a x  + a x  + ... + a x  = b
```

Matrix form:

```
AX = B
```

Notation

- `A = [a ]` : coefficient matrix of order `n × n`
- `X = [x , x , ..., x ]` : vector of unknowns
- `B = [b , b , ..., b ]` : constant vector

Core Idea The system is simplified by eliminating variables using **elementary row operations** to obtain an upper triangular matrix.

Elimination Approach (Formula) To eliminate element `a` (where `j < i`):

```
R ← R - (a  / a ) R
```

Phases Involved

Forward Elimination Transforms the augmented matrix `[A | B]` into an **upper triangular form**.

Back Substitution Solutions are obtained using:

```
x  = b  / a
x  = (1 / a ) [ b  - Σ (a  x ) ], j = i+1 to n
```

Accuracy Considerations

- Exact in theory
- Rounding errors may occur in floating-point arithmetic
- Pivoting improves numerical stability

Applicability

- Suitable for small to medium-sized systems
- Widely used due to conceptual simplicity

Gauss Elimination Code

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

int main() {

    // File Handling
    ifstream in("input.txt");
    ofstream out("output.txt");

    if (!in) {
        cerr << "Error: input.txt not found\n";
        return 1;
    }

    int n;
    in >> n;

    vector<vector<float>> a(n, vector<float>(n + 1));

    // augmented matrix
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= n; j++) {
            in >> a[i][j];
        }
    }

    // Copying original matrix for echelon form
    vector<vector<float>> echelon = a;

    //Forward Elimination (Echelon Form)
    for (int i = 0; i < n; i++) {

        int maxRow = i;
        for (int k = i + 1; k < n; k++) {
            if (fabs(echelon[k][i]) > fabs(echelon[maxRow][i]))
                maxRow = k;
        }

        swap(echelon[i], echelon[maxRow]);

        if (fabs(echelon[i][i]) < 1e-9)
            continue;

        for (int k = i + 1; k < n; k++) {
```

```
        float factor = echelon[k][i] / echelon[i][i];
        for (int j = i; j <= n; j++) {
            echelon[k][j] -= factor * echelon[i][j];
        }
    }
}

// Printing Echelon Form
out << "Echelon Form (Upper Triangular):\n";
for (int i = 0; i < n; i++) {
    for (int j = 0; j <= n; j++)
        out << fixed << setprecision(4) << echelon[i][j] << " ";
    out << "\n";
}
out << "\n";


int rankA = 0, rankAug = 0;
const float EPS = 1e-9;

for (int i = 0; i < n; i++) {
    bool nonZeroCoeff = false;
    bool nonZeroAug = false;

    for (int j = 0; j < n; j++) {
        if (fabs(echelon[i][j]) > EPS)
            nonZeroCoeff = true;
    }

    if (fabs(echelon[i][n]) > EPS)
        nonZeroAug = true;

    if (nonZeroCoeff)
        rankA++;

    if (nonZeroCoeff || nonZeroAug)
        rankAug++;
}

out << "System Classification:\n";

if (rankA < rankAug) {
    out << "→ No Solution (Inconsistent System)\n";
    return 0;
}
else if (rankA < n) {
```

```
            out << "→ Infinite Solutions (Dependent System)\n";
            return 0;
        }
        else {
            out << "→ Unique Solution Exists\n\n";
        }


        for (int i = 0; i < n; i++) {

            int maxRow = i;
            for (int k = i + 1; k < n; k++) {
                if (fabs(a[k][i]) > fabs(a[maxRow][i]))
                    maxRow = k;
            }

            swap(a[i], a[maxRow]);

            float pivot = a[i][i];

            if (fabs(pivot) < EPS) {
                out << "Numerical instability detected.\n";
                return 1;
            }

            for (int j = 0; j <= n; j++)
                a[i][j] /= pivot;

            for (int k = 0; k < n; k++) {
                if (k != i) {
                    float factor = a[k][i];
                    for (int j = 0; j <= n; j++)
                        a[k][j] -= factor * a[i][j];
                }
            }
        }

        // Output Solution
        out << "Solution:\n";
        for (int i = 0; i < n; i++) {
            out << "x" << i + 1 << " = " << a[i][n] << "\n";
        }

        return 0;
    }
```

Gauss Elimination Input

```
3
2 1 -1 8
-3 -1 2 -11
-2 1 2 -3
```

Gauss Elimination Output

```
Echelon Form (Upper Triangular):
-3.0000 -1.0000 2.0000 -11.0000
0.0000 1.6667 0.6667 4.3333
0.0000 0.0000 0.2000 -0.2000

System Classification:
→ Unique Solution Exists

Solution:
x1 = 2.0000
x2 = 3.0000
x3 = -1.0000
```

Back to Contents

Gauss Jordan Elimination Method

Gauss Jordan Theory

Method used Gauss–Jordan Elimination Method

Objective To solve a system of linear equations by reducing the augmented matrix directly to **Reduced Row Echelon Form (RREF)**.

Data Requirement Augmented matrix form:

```
[A | B]
```

Notation

- `a` : element of coefficient matrix
- `b` : element of constant vector

Core Idea Each pivot element is made **unity**, and all other elements in its column are eliminated, producing an identity matrix.

Elimination Approach (Formulae) **Normalization of pivot row:**

`R ← R / a`

Elimination of other rows:

`R ← R - a R (j i)`

Matrix Form Obtained

`[I | X]`

where `I` is the identity matrix and `X` contains the solution.

Evaluation Process The solution is obtained directly as:

`x = b`

Accuracy Considerations

- More computationally expensive than Gauss Elimination
- Sensitive to rounding errors for large systems

Applicability

- Used when a direct solution or matrix inverse is required

Gauss Jordan Code

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

int main() {

    // File Handling
    ifstream in("input.txt");
    ofstream out("output.txt");

    if (!in) {
        cerr << "Error: input.txt not found\n";
        return 1;
    }

    int n;
    in >> n;

    vector<vector<float>> a(n, vector<float>(n + 1));

    // Reading augmented matrix
```

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j <= n; j++) {
        in >> a[i][j];
    }
}

//Copy of matrix for echelon form
vector<vector<float>> echelon = a;

// Forward Elimination (Echelon Form)
for (int i = 0; i < n; i++) {

    int maxRow = i;
    for (int k = i + 1; k < n; k++) {
        if (fabs(echelon[k][i]) > fabs(echelon[maxRow][i]))
            maxRow = k;
    }

    swap(echelon[i], echelon[maxRow]);

    if (fabs(echelon[i][i]) < 1e-9)
        continue;

    for (int k = i + 1; k < n; k++) {
        float factor = echelon[k][i] / echelon[i][i];
        for (int j = i; j <= n; j++) {
            echelon[k][j] -= factor * echelon[i][j];
        }
    }
}

// Printing Echelon Form
out << "Echelon Form (Upper Triangular):\n";
for (int i = 0; i < n; i++) {
    for (int j = 0; j <= n; j++)
        out << fixed << setprecision(4) << echelon[i][j] << " ";
    out << "\n";
}
out << "\n";


int rankA = 0, rankAug = 0;
const float EPS = 1e-9;

for (int i = 0; i < n; i++) {
    bool nonZeroCoeff = false;
```

```
    bool nonZeroAug = false;

    for (int j = 0; j < n; j++) {
        if (fabs(echelon[i][j]) > EPS)
            nonZeroCoeff = true;
    }

    if (fabs(echelon[i][n]) > EPS)
        nonZeroAug = true;

    if (nonZeroCoeff)
        rankA++;

    if (nonZeroCoeff || nonZeroAug)
        rankAug++;
}

out << "System Classification:\n";

if (rankA < rankAug) {
    out << "→ No Solution (Inconsistent System)\n";
    return 0;
}
else if (rankA < n) {
    out << "→ Infinite Solutions (Dependent System)\n";
    return 0;
}
else {
    out << "→ Unique Solution Exists\n\n";
}


for (int i = 0; i < n; i++) {

    int maxRow = i;
    for (int k = i + 1; k < n; k++) {
        if (fabs(a[k][i]) > fabs(a[maxRow][i]))
            maxRow = k;
    }

    swap(a[i], a[maxRow]);

    float pivot = a[i][i];
    if (fabs(pivot) < EPS) {
        out << "Numerical instability detected.\n";
        return 1;
```

```
        }

        // Normalize pivot row
        for (int j = 0; j <= n; j++)
            a[i][j] /= pivot;

        // Eliminate other rows
        for (int k = 0; k < n; k++) {
            if (k != i) {
                float factor = a[k][i];
                for (int j = 0; j <= n; j++)
                    a[k][j] -= factor * a[i][j];
            }
        }
    }

    //Output Solution
    out << "Solution:\n";
    for (int i = 0; i < n; i++) {
        out << "x" << i + 1 << " = " << a[i][n] << "\n";
    }

    return 0;
}
```

Gauss Jordan Input

```
5
2 1 -1 3 2 9
1 3 2 -1 1 8
3 2 4 1 -2 20
2 1 3 2 1 17
1 -1 2 3 4 15
```

Gauss Jordan Output

```
Echelon Form (Upper Triangular):
3.0000 2.0000 4.0000 1.0000 -2.0000 20.0000
0.0000 2.3333 0.6667 -1.3333 1.6667 1.3333
0.0000 0.0000 -3.5714 2.1429 3.5714 -4.1429
0.0000 0.0000 0.0000 2.4000 7.0000 7.9600
0.0000 0.0000 0.0000 0.0000 -1.0833 -1.2833

System Classification:
→ Unique Solution Exists
```

```
Solution:
x1 = 5.1538
x2 = -1.0000
x3 = 2.2615
x4 = -0.1385
x5 = 1.1846
```

Back to Contents

LU Decomposition Method

LU Decomposition Theory

Method used **LU Decomposition Method**

Objective To solve a system of linear equations by factorizing the coefficient matrix into lower and upper triangular matrices.

Data Requirement A square matrix with non-zero pivots.

Core Idea (Formula)

`A = LU`

Notation

- `L = [l ]` : lower triangular matrix
- `U = [u ]` : upper triangular matrix
- `A` : coefficient matrix
- `X` : solution vector
- `B` : constant vector

Solution Process **Step 1:** Solve

`LY = B`

using forward substitution:

`y  = b  - Σ (l  y ), j = 1 to i-1`

Step 2: Solve

`UX = Y`

using back substitution:

`x  = (1 / u ) [ y  - Σ (u  x ) ], j = i+1 to n`

Accuracy Considerations

- More efficient than repeated Gauss Elimination
- Numerical stability improves with pivoting

Applicability

- Ideal for solving multiple systems with the same coefficient matrix

LU Decomposition Code

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

int main() {

    // File Handling
    ifstream in("input.txt");
    ofstream out("output.txt");

    if (!in) {
        cerr << "Error: input.txt not found\n";
        return 1;
    }

    int n;
    in >> n;

    vector<vector<double>> a(n + 1, vector<double>(n + 2));

    // Reading augmented matrix A|b
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n + 1; j++) {
            in >> a[i][j];
        }
    }

    vector<vector<double>> u(n + 1, vector<double>(n + 1, 0));
    vector<vector<double>> l(n + 1, vector<double>(n + 1, 0));

    for (int i = 1; i <= n; i++) {
        l[i][i] = 1;
    }

    //  LU Decomposition
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
```

```
        if (i <= j) {
            u[i][j] = a[i][j];
            for (int k = 1; k < i; k++)
                u[i][j] -= l[i][k] * u[k][j];
        }
        else {
            l[i][j] = a[i][j];
            for (int k = 1; k < j; k++)
                l[i][j] -= l[i][k] * u[k][j];

            if (u[j][j] == 0) {
                out << "Matrix is singular. Cannot compute LU decomposition.\n";
                return 0;
            }

            l[i][j] /= u[j][j];
        }
    }
}

// Printing U Matrix
out << "U Matrix:\n";
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        out << u[i][j] << " ";
    }
    out << "\n";
}

// Printing L Matrix
out << "\nL Matrix:\n";
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        out << l[i][j] << " ";
    }
    out << "\n";
}

// Checking Solution Type
bool noSolution = false;
bool infiniteSolution = false;

for (int i = 1; i <= n; i++) {
    bool allZero = true;
```

```
    for (int j = 1; j <= n; j++) {
        if (fabs(u[i][j]) > 1e-9) {
            allZero = false;
            break;
        }
    }

    if (allZero) {
        double rhs = a[i][n + 1];

        if (fabs(rhs) > 1e-9) {
            noSolution = true;
        }
        else {
            infiniteSolution = true;
        }
    }
}

if (noSolution) {
    out << "\nThe system has NO SOLUTION (Inconsistent equations).\n";
    return 0;
}

if (infiniteSolution) {
    out << "\nThe system has INFINITE SOLUTIONS (Dependent equations).\n";
    return 0;
}

out << "\nThe system has a UNIQUE SOLUTION.\n";

// Forward Substitution: Ly = b
vector<double> y(n + 1, 0);

for (int i = 1; i <= n; i++) {
    y[i] = a[i][n + 1];
    for (int k = 1; k < i; k++) {
        y[i] -= l[i][k] * y[k];
    }
}

// Backward Substitution: Ux = y
vector<double> ans(n + 1, 0);

for (int i = n; i >= 1; i--) {
    ans[i] = y[i];
```

```
        for (int k = i + 1; k <= n; k++) {
            ans[i] -= u[i][k] * ans[k];
        }
        ans[i] /= u[i][i];
    }

    // Printing Solution
    out << "\nFinal Solution (x values):\n";
    for (int i = 1; i <= n; i++) {
        out << "x" << i << " = " << ans[i] << "\n";
    }

    return 0;
}
```

LU Decomposition Input

```
5
2 1 -1 3 2 9
1 3 2 -1 1 8
3 2 4 1 -2 20
2 1 3 2 1 17
1 -1 2 3 4 15
```

LU Decomposition Output

```
U Matrix:
2 1 -1 3 2
0 2.5 2.5 -2.5 0
0 0 5 -3 -5
0 0 0 1.4 3
0 0 0 0 1.85714

L Matrix:
1 0 0 0 0
0.5 1 0 0 0
1.5 0.2 1 0 0
1 0 0.8 1 0
0.5 -0.6 0.8 1.71429 1

The system has a UNIQUE SOLUTION.

Final Solution (x values):
x1 = 5.15385
x2 = -1
x3 = 2.26154
```

```
x4 = -0.138462
x5 = 1.18462
```

Back to Contents

Matrix Inversion

Matrix Inversion Theory

Method used Matrix Inversion Method

Objective To solve a system of linear equations using the inverse of the coefficient matrix.

Data Requirement A square, non-singular matrix (`det(A)  0`).

Core Idea (Formula) Given:

`AX = B`

The solution is:

`X = A ¹ B`

Notation

- `A ¹` : inverse of matrix `A`
- `I` : identity matrix

Inversion Approach The inverse is computed using Gauss–Jordan elimination:

`[A | I] → [I | A ¹]`

Evaluation Process Once `A ¹` is obtained, the solution vector is computed using matrix multiplication.

Accuracy Considerations

- Computationally expensive
- Sensitive to rounding errors
- Not recommended for large systems

Applicability

- Useful for theoretical analysis
- Suitable for small systems only

Matrix Inversion Code

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

// Cofactor
void getCofactor(const vector<vector<double>>& A,
                 vector<vector<double>>& temp,
                 int p, int q, int n)
{
    int i = 1, j = 1;
    for (int row = 1; row <= n; row++) {
        for (int col = 1; col <= n; col++) {
            if (row != p && col != q) {
                temp[i][j++] = A[row][col];
                if (j == n) {
                    j = 1;
                    i++;
                }
            }
        }
    }
}

//Recursive Determinant
double determinant(const vector<vector<double>>& A, int n)
{
    if (n == 1)
        return A[1][1];

    double det = 0;
    int sign = 1;
    vector<vector<double>> temp(n, vector<double>(n, 0));

    for (int i = 1; i <= n; i++) {
        getCofactor(A, temp, 1, i, n);
        det += sign * A[1][i] * determinant(temp, n - 1);
        sign = -sign;
    }
    return det;
}

int main()
{
    ifstream fin("input.txt");
    ofstream fout("output.txt");
```

```
if (!fin) {
    cout << "Error opening input file.\n";
    return 1;
}

int n;
fin >> n;

vector<vector<double>> aug(n+1, vector<double>(n+2, 0));
vector<vector<double>> a(n+1, vector<double>(n+1, 0));
vector<vector<double>> B(n+1, vector<double>(2, 0));
vector<vector<double>> C(n+1, vector<double>(n+1, 0));
vector<vector<double>> C1(n+1, vector<double>(n+1, 0));
vector<vector<double>> res(n+1, vector<double>(2, 0));

for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n + 1; j++)
        fin >> aug[i][j];

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++)
        a[i][j] = aug[i][j];
    B[i][1] = aug[i][n+1];
}

double detA = determinant(a, n);

if (detA == 0) {
    bool noSol = false, infiniteSol = true;

    for (int r = 1; r <= n; r++) {
        bool allZero = true;
        for (int c = 1; c <= n; c++)
            if (aug[r][c] != 0)
                allZero = false;

        if (allZero && aug[r][n+1] != 0) {
            noSol = true;
            infiniteSol = false;
            break;
        }
        if (!allZero)
            infiniteSol = false;
    }
```

```
        if (noSol)
            fout << "Determinant = 0 → No Solution (Inconsistent System)\n";
        else
            fout << "Determinant = 0 → Infinite Solutions (Dependent System)\n";

        return 0;
    }

    fout << "Determinant = " << detA << "\n\n";

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            vector<vector<double>> temp(n, vector<double>(n, 0));
            getCofactor(a, temp, i, j, n);
            C[i][j] = pow(-1, i + j) * determinant(temp, n - 1);
        }
    }

    fout << "Inverse Matrix:\n";
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            C1[i][j] = C[j][i] / detA;
            fout << C1[i][j] << " ";
        }
        fout << "\n";
    }

    for (int i = 1; i <= n; i++) {
        for (int t = 1; t <= n; t++)
            res[i][1] += C1[i][t] * B[t][1];
    }

    fout << "\nSolution Vector:\n";
    for (int i = 1; i <= n; i++)
        fout << "x" << i << " = " << res[i][1] << "\n";

    fin.close();
    fout.close();

    return 0;
}
```

Matrix Inversion Input

```
5
2 1 -1 3 2 9
```

```
1 3 2 -1 1 8
3 2 4 1 -2 20
2 1 3 2 1 17
1 -1 2 3 4 15
```

Matrix Inversion Output

```
Determinant = 65

Inverse Matrix:
0.384615 0.384615 1.92308 -3.76923 1.61538
-0 0 -1 2 -1
-0.246154 -0.0461538 -0.230769 0.692308 -0.153846
-0.0461538 -0.446154 -1.23077 2.69231 -1.15385
0.0615385 0.261538 0.307692 -0.923077 0.538462

Solution Vector:
x1 = 5.15385
x2 = -1
x3 = 2.26154
x4 = -0.138462
x5 = 1.18462
```

Back to Contents

Solution of Non-Linear Equations

Bisection Method

Bisection Theory

Objective To find a root of a nonlinear algebraic equation.

Data Requirement A polynomial equation of `n` degree:

- $a\,x\ + a\ \ x^{\ 1} + \ldots + a\,x^2 + a\,x + a\ = 0$

Core Idea It repeatedly bisects an interval and then selects a subinterval in which a root must lie. Always converges if $f(a)\ \cdot\ f(b) < 0$
Formula:

```
x   = (a + b) / 2
```

Bisection Code

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

```
double f(double x, vector<double>& coef) {
    int n = coef.size() - 1;
    double s = 0;
    for (int i = 0; i <= n; i++) s += coef[i] * pow(x, n - i);
    return s;
}
void printEq(ofstream &out, vector<double>& coef) {
    int n = coef.size() - 1;
    out << "Equation: ";
    for (int i = 0; i <= n; i++) {
        out << coef[i] << "*x^" << (n - i);
        if (i != n) out << " + ";
    }
    out << endl;
}
int main() {
    ifstream in("input.txt");
    ofstream out("output.txt");

    int deg, maxit, it;
    double lowl, upl, err, root, prev;

    in >> deg;
    vector<double> coef(deg + 1);
    for (int i = 0; i <= deg; i++) in >> coef[i];
    in >> lowl >> upl >> err >> maxit;
    printEq(out,coef);
    if (f(lowl, coef) * f(upl, coef) >= 0) {
        out << "Invalid interval\n";
        return 0;
    }

    for (it = 1; it <= maxit; it++) {
        prev = root;
        root = (lowl + upl) / 2.0;

        if (fabs(root - prev) < err || fabs(f(root, coef)) < err) break;

        f(lowl, coef) * f(root, coef) < 0 ? upl = root : lowl = root;
    }

    out << "Bisection Root = " << root << "\n";
    out << "Iterations = " << it << "\n";
    return 0;
}
```

Bisection Input

```
3
1 0 -4 -9
0 5 .0001 30
```

Bisection Output

```
Equation: 1*x^3 + 0*x^2 + -4*x^1 + -9*x^0
Bisection Root = 2.70653
Iterations = 16
```

Back to Contents

False Position Method

False Position Theory

Objective To solve nonlinear algebraic equation using a bracketing method based on linear interpolation.

Data Requirement A polynomial equation of `n` degree:

- $a\,x\ + a\ \ x^{\ 1} + \ldots + a\,x^2 + a\,x + a\ = 0$

Core Idea Uses linear interpolation between two points where the function has opposite signs. Draws a straight line connecting the function values at the endpoints and finds where it crosses the x-axis. More efficient than bisection method.

Formula:

```
x   = (a · f(b) - b · f(a)) / (f(b) - f(a))
```

False Position Code

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

double f(double x, vector<double>& coef) {
    int n = coef.size() - 1;
    double s = 0;
    for (int i = 0; i <= n; i++)
        s += coef[i] * pow(x, n - i);
    return s;
}

void printEq(ofstream &out, vector<double>& coef) {
```

```
    int n = coef.size() - 1;
    out << "Equation: ";
    for (int i = 0; i <= n; i++) {
        out << coef[i] << "*x^" << (n - i);
        if (i != n) out << " + ";
    }
    out << endl;
}

int main() {
    ifstream in("input.txt");
    ofstream out("output.txt");

    int deg, maxit,it;
    double lowl, upl, err, root;
    in >> deg;

    vector<double> coef(deg + 1);
    for (int i = 0; i <= deg; i++) in >> coef[i];

    in >> lowl >> upl >> err >> maxit;

    printEq(out, coef);

    if (f(lowl, coef) * f(upl, coef) >= 0) {
        out << "Invalid interval"<< endl;
        return 0;
    }

    double pre_root; root=lowl;
    for (it = 1; it <= maxit; it++) {
        pre_root = root;
        root = (lowl * f(upl, coef) - upl * f(lowl, coef)) / (f(upl, coef) - f(lowl, coef));

        if (fabs(root - pre_root) < err || fabs(f(root, coef)) < err) break;

        f(lowl, coef) * f(root, coef) < 0 ? upl = root : lowl = root;
    }

    out << "False Position Root = " << root << endl;
    out << "iterations = " << it << endl;
    return 0;
}
```

False Position Input

```
3
1 0 -4 -9
0 5 .0001 30
```

False Position Output

```
Equation: 1*x^3 + 0*x^2 + -4*x^1 + -9*x^0
False Position Root = 2.70642
iterations = 21
```

Back to Contents

Secant Method

Secant Theory

Objective To solve nonlinear algebraic equation using an iterative method that approximates the derivative.

Data Requirement A polynomial equation of `n` degree:

- $a\,x\ + a\ \ x^{\ 1} + \ldots + a\,x^2 + a\,x + a\ = 0$

Core Idea Uses two initial approximations and draws a secant line through them to find the next approximation. Does not require derivative computation unlike Newton-Raphson.
Formula:

```
x   = x  - (f(x )(x  - x  )) / (f(x ) - f(x  ))
```

Secant Code

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

double f(double x, vector<double>& coef) {
    int n = coef.size() - 1;
    double s = 0;
    for (int i = 0; i <= n; i++)
        s += coef[i] * pow(x, n - i);
    return s;
}
void printEq(ofstream &out, vector<double>& coef) {
    int n = coef.size() - 1;
    out << "Equation: ";
    for (int i = 0; i <= n; i++) {
        out << coef[i] << "*x^" << (n - i);
```

```
        if (i != n) out << " + ";
    }
    out << endl;
}
int main() {
    ifstream in("input.txt");
    ofstream out("output.txt");

    int deg, maxit, it;
    double x0, x1, root, err;

    in >> deg;
    vector<double> coef(deg + 1);
    for (int i = 0; i <= deg; i++) in >> coef[i];
    in >> x0 >> x1 >> err >> maxit;
    printEq(out,coef);
    for (it = 1; it <= maxit; it++) {
        root = x1 - f(x1, coef) * (x1 - x0) / (f(x1, coef) - f(x0, coef));
        if (fabs(root - x1) < err || fabs(f(root, coef)) < err) break;
        x0 = x1;
        x1 = root;
    }
    out << "Secant Root = " << root << "\n";
    out << "Iterations = " << it << "\n";
    return 0;
}
```

Secant Input

```
3
1 0 -1 -2
1 2 0.0001 20
```

Secant Output

```
Equation: 1*x^3 + 0*x^2 + -1*x^1 + -2*x^0
Secant Root = 1.52138
Iterations = 5
```

Back to Contents

Newton Raphson Method

Newton Raphson Theory

Objective To solve nonlinear algebraic equation using tangent line approximation at each iteration.

Data Requirement A polynomial equation of `n` degree:

- $a\,x\ + a\ \ x^{\ 1} + \ldots + a\,x^2 + a\,x + a\ = 0$

Core Idea Starts with an initial guess and uses the tangent line at that point to find a better approximation. Requires both function and its derivative. Converges quadratically when close to root.

Formula:

```
x   = x  - f(x ) / f'(x )
```

Newton Raphson Code

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

double f(double x, vector<double>& coef) {
    int n = coef.size() - 1;
    double s = 0;
    for (int i = 0; i <= n; i++)
        s += coef[i] * pow(x, n - i);
    return s;
}

double df(double x, vector<double>& coef) {
    int n = coef.size() - 1;
    double s = 0;
    for (int i = 0; i < n; i++)
        s += coef[i] * (n - i) * pow(x, n - i - 1);
    return s;
}
void printEq(ofstream &out, vector<double>& coef) {
    int n = coef.size() - 1;
    out << "Equation: ";
    for (int i = 0; i <= n; i++) {
        out << coef[i] << "*x^" << (n - i);
        if (i != n) out << " + ";
    }
    out << endl;
}
int main() {
    ifstream in("input.txt");
    ofstream out("output.txt");

    int deg, maxit, it;
    double x0, root, err;
```

```
    in >> deg;
    vector<double> coef(deg + 1);
    for (int i = 0; i <= deg; i++) in >> coef[i];
    in >> x0 >> err >> maxit;
    printEq(out,coef);
    for (it = 1; it <= maxit; it++) {
        if (fabs(df(x0, coef)) < 1e-12) {
            out << "Derivative too small\n";
            return 0;
        }
        root = x0 - f(x0, coef) / df(x0, coef);
        if (fabs(root - x0) < err || fabs(f(root, coef)) < err) break;
        x0 = root;
    }
    out << "Newton-Raphson Root = " << root << "\n";
    out << "Iterations = " << it << "\n";
    return 0;
}
```

Newton Raphson Input

```
3
1 0 -1 -2
1.5 0.0001 20
```

Newton Raphson Output

```
Equation: 1*x^3 + 0*x^2 + -1*x^1 + -2*x^0
Newton-Raphson Root = 1.52138
Iterations = 2
```

Back to Contents

Solution of Interpolation

Newton's Forward Interpolation Method

Newton's Forward Interpolation Theory

Method used Newton's Forward Difference Interpolation

Objective To approximate function values at intermediate points using forward differences. Supports multiple data points with automatic polynomial order detection.

NEWTON FORWARD INTERPOLATION FORMULA $f(x) = f(x_0) + u\Delta f(x_0) + [u(u-1)/2!]\ \Delta^2 f(x_0) + [u(u-1)(u-2)/3!]\ \Delta^3 f(x_0) + \ldots$

where

$u = (x - x_0) / h$
h = step size $(x_1 - x_0)$
$\Delta f(x_0)$ = nth forward difference at x_0

FORWARD DIFFERENCE TABLE $\Delta f(x_0) = f(x_1) - f(x_0)$
$\Delta^2 f(x_0) = \Delta f(x_1) - \Delta f(x_0)$
$\Delta f(x_0) = \Delta^1 f(x_1) - \Delta^1 f(x_0)$

Data Requirement

- Tabulated values $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$
- Equal spacing between x values

Features

- Best suited for interpolation near the beginning of the data table.
- Requires equally spaced x values.

Newton's Forward Interpolation Code

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


int fact(int n)
{
    int f = 1;
    for(int i = 2; i <= n; i++)
        f *= i;
    return f;
}


int detectOrder( vector<vector<double>>& diff, double eps = 1e-9)
{
    int n = diff.size();

    for(int j = 1; j < n; j++)
    {
        bool allZero = true;
        for(int i = 0; i < n - j; i++)
        {
```

```
            if (fabs(diff[i][j]) > eps)
            {
                allZero = false;
                break;
            }
        }
        if (allZero)
            return j - 1;
    }
    return n - 1;
}

int main()
{
    ifstream inputFile("Input.txt");
    ofstream outputFile("Output.txt");

    if (!inputFile.is_open())
    {
        cout << "Error opening Input.txt\n";
        return 1;
    }

    int testCases;
    inputFile >> testCases;

    for (int tc = 1; tc <= testCases; tc++)
    {
        int n;
        inputFile >> n;

        vector<double> x(n), y(n);
        for (int i = 0; i < n; i++) inputFile >> x[i];
        for (int i = 0; i < n; i++) inputFile >> y[i];

        double X;
        inputFile >> X;

        cout << "Test Case " << tc << "\n";
        outputFile << "Test Case " << tc << "\n";

        double h = x[1] - x[0];
        double u = (X - x[0]) / h;


        vector<vector<double>> diff(n, vector<double>(n, 0.0));
```

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

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

        cout << "Forward Difference Table:\n";
        outputFile << "Forward Difference Table:\n";

       cout << fixed << setprecision(2);
outputFile << fixed << setprecision(2);

for (int i = 0; i < n; i++)
{

    cout << setw(10) << diff[i][0];
    outputFile << setw(10) << diff[i][0];


    for (int j = 1; j < n - i; j++)
    {
        cout << setw(10) << diff[i][j];
        outputFile << setw(10) << diff[i][j];
    }

    cout << "\n";
    outputFile << "\n";
}


        double fx = diff[0][0];
        double u_term = 1.0;

        for (int k = 1; k < n; k++)
        {
            u_term *= (u - (k - 1));
            fx += (u_term * diff[0][k]) / fact(k);
        }

        int order = detectOrder(diff);


        double error = 0.0;
        if (order + 1 < n)
        {
```

```
            double u_err = 1.0;
            for (int i = 0; i <= order; i++)
                u_err *= (u - i);

            error = fabs((u_err * diff[0][order + 1]) / fact(order + 1));
        }

        cout << "Step size (h) = " << h << "\n";
        cout << "u = " << u << "\n";
        cout << "Detected Polynomial Order = " << order << "\n";
        cout << "Interpolated value f(" << X << ") = "
             << fixed << setprecision(2) << fx << "\n";
        cout << "Estimated Forward Interpolation Error = "
             << fixed << setprecision(2) << error << "\n\n";

        outputFile << "Step size (h) = " << h << "\n";
        outputFile << "u = " << u << "\n";
        outputFile << "Detected Polynomial Order = " << order << "\n";
        outputFile << "Interpolated value f(" << X << ") = "
                   << fixed << setprecision(2) << fx << "\n";
        outputFile << "Estimated Forward Interpolation Error = "
                   << fixed << setprecision(2) << error << "\n\n";
    }

    inputFile.close();
    outputFile.close();
    return 0;
}
```

Newton's Forward Interpolation Input

```
2
5
0 1 2 3 4
1 2 5 10 17
0.5
6
1 2 3 4 5 6
1 8 27 64 125 216
2.5
```

Newton's Forward Interpolation Output

```
Test Case 1
Forward Difference Table:
      1.00      1.00      2.00      0.00      0.00
```

```
      2.00      3.00      2.00      0.00
      5.00      5.00      2.00
     10.00      7.00
     17.00
Step size (h) = 1.00
u = 0.50
Detected Polynomial Order = 2
Interpolated value f(0.50) = 1.25
Estimated Forward Interpolation Error = 0.00

Test Case 2
Forward Difference Table:
      1.00      7.00     12.00      6.00      0.00      0.00
      8.00     19.00     18.00      6.00      0.00
     27.00     37.00     24.00      6.00
     64.00     61.00     30.00
    125.00     91.00
    216.00
Step size (h) = 1.00
u = 1.50
Detected Polynomial Order = 3
Interpolated value f(2.50) = 15.62
Estimated Forward Interpolation Error = 0.00
```

Back to Contents

Newton's Backward Interpolation Method

Newton's Backward Interpolation Theory

Method used Newton's Backward Difference Interpolation

Objective To approximate function values at intermediate points using backward differences. Ideal when interpolating near the end of the data table.

NEWTON BACKWARD INTERPOLATION FORMULA $f(x) = f(x) + u\,f(x) + [u(u+1)/2!]\ {}^2f(x) + [u(u+1)(u+2)/3!]\ {}^3f(x) + \ldots$

where

$u = (x - x) / h$
h = step size $(x - x)$
$f(x)$ = nth backward difference at x

BACKWARD DIFFERENCE TABLE $f(x_) = f(x_) - f(x_{\ })$

$^2f(x_) = f(x_) - f(x_{\ })$

$f(x_) = {}^1f(x_) - {}^1f(x_{\ })$

Data Requirement

- Tabulated values $(x_, y_)$, $(x_, y_)$, ..., $(x_, y_)$
- Equal spacing between x values

Features

- Best suited for interpolation near the end of the data table.
- Requires equally spaced x values.
- Particularly useful when new data points are appended at the end.

Newton's Backward Interpolation Code

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

int fact(int n)
{
    int f = 1;
    for(int i = 2; i <= n; i++)
        f *= i;
    return f;
}

int detectOrderBackward(const vector<vector<double>>& diff, double eps = 1e-9)
{
    int n = diff.size();

    for(int j = 1; j < n; j++)
    {
        bool allZero = true;
        for(int i = j; i < n; i++)
        {
            if (fabs(diff[i][j]) > eps)
            {
                allZero = false;
                break;
            }
        }
        if (allZero)
            return j - 1;
    }
    return n - 1;
```

```
}

int main()
{
    ifstream inputFile("Input.txt");
    ofstream outputFile("Output.txt");

    if (!inputFile.is_open())
        return 1;

    int testCases;
    inputFile >> testCases;

    for (int tc = 1; tc <= testCases; tc++)
    {
        int n;
        inputFile >> n;

        vector<double> x(n), y(n);
        for (int i = 0; i < n; i++) inputFile >> x[i];
        for (int i = 0; i < n; i++) inputFile >> y[i];

        double X;
        inputFile >> X;

        double h = x[1] - x[0];
        double u = (X - x[n-1]) / h;

        vector<vector<double>> diff(n, vector<double>(n, 0.0));
        for (int i = 0; i < n; i++)
            diff[i][0] = y[i];

        for (int j = 1; j < n; j++)
            for (int i = n-1; i >= j; i--)
                diff[i][j] = diff[i][j-1] - diff[i-1][j-1];

        cout << fixed << setprecision(2);
        outputFile << fixed << setprecision(2);

        cout << "Test Case " << tc << "\n";
        outputFile << "Test Case " << tc << "\n";

        cout << "Backward Difference Table:\n";
        outputFile << "Backward Difference Table:\n";

        for (int i = 0; i < n; i++)
```

```
    {
        for (int j = 0; j <= i; j++)
        {
            cout << setw(10) << diff[i][j];
            outputFile << setw(10) << diff[i][j];
        }
        cout << "\n";
        outputFile << "\n";
    }

    double fx = diff[n-1][0];
    double u_term = 1.0;

    for (int k = 1; k < n; k++)
    {
        u_term *= (u + (k - 1));
        fx += (u_term * diff[n-1][k]) / fact(k);
    }

    int order = detectOrderBackward(diff);

    double error = 0.0;
    if (order + 1 < n)
    {
        double u_err = 1.0;
        for (int i = 0; i <= order; i++)
            u_err *= (u + i);

        error = fabs((u_err * diff[n-1][order + 1]) / fact(order + 1));
    }

    cout << "h = " << h << "\n";
    cout << "u = " << u << "\n";
    cout << "Detected Polynomial Order = " << order << "\n";
    cout << "Interpolated value f(" << X << ") = " << fx << "\n";
    cout << "Estimated Backward Interpolation Error = " << error << "\n\n";

    outputFile << "h = " << h << "\n";
    outputFile << "u = " << u << "\n";
    outputFile << "Detected Polynomial Order = " << order << "\n";
    outputFile << "Interpolated value f(" << X << ") = " << fx << "\n";
    outputFile << "Estimated Backward Interpolation Error = " << error << "\n\n";
}

inputFile.close();
outputFile.close();
```

```
    return 0;
}
```

Newton's Backward Interpolation Input

```
1
7
0 1 2 3 4 5 6
0 1 8 27 64 125 300
5.3
```

Newton's Backward Interpolation Output

```
Test Case 1
Backward Difference Table:
      0.00
      1.00      1.00
      8.00      7.00      6.00
     27.00     19.00     12.00      6.00
     64.00     37.00     18.00      6.00      0.00
    125.00     61.00     24.00      6.00      0.00      0.00
    300.00    175.00    114.00     90.00     84.00     84.00     84.00
h = 1.00
u = -0.70
Detected Polynomial Order = 6
Interpolated value f(5.30) = 156.75
Estimated Backward Interpolation Error = 0.00
```

Back to Contents

Divided Difference Method

Divided Difference Theory

Method used Newton's Divided Difference Interpolation

Objective To construct interpolating polynomials for unequally spaced data points. Works with both equal and unequal spacing.

DIVIDED DIFFERENCES FORMULA First-order divided difference

f[x , x] = [f(x) − f(x)] / (x − x)

Higher-order divided differences (recursive)

f[x , x , … , x] = [f[x , … , x] − f[x , … , x]] / (x − x)

NEWTON DIVIDED DIFFERENCE POLYNOMIAL $P(x) = f[x] + (x - x) f[x, x] + (x - x)(x - x) f[x, x, x] + \ldots + (x - x)(x - x)\ldots(x - x) f[x, \ldots, x]$

Data Requirement

- Distinct data points (x, y), (x, y), …, (x, y)
- Works with equal or unequal spacing

Features

- Works for both equally and unequally spaced nodes.
- Coefficients can be updated easily when new data points are added.
- Numerically stable when nodes are distinct and well separated.

Divided Difference Code

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

int main()
{
    ifstream inputFile("Input.txt");
    ofstream outputFile("Output.txt");

    if(!inputFile.is_open())
        return 1;

    int T;
    inputFile >> T;

    cout << fixed << setprecision(2);
    outputFile << fixed << setprecision(2);

    for(int tc = 1; tc <= T; tc++)
    {
        int n;
        inputFile >> n;

        vector<double> x(n), y(n);
        for(int i = 0; i < n; i++) inputFile >> x[i];
        for(int i = 0; i < n; i++) inputFile >> y[i];

        double x_extra, y_extra;
        inputFile >> x_extra >> y_extra;

        double X;
```

```
inputFile >> X;

vector<double> x_all = x;
vector<double> y_all = y;
x_all.push_back(x_extra);
y_all.push_back(y_extra);

vector<vector<double>> dd(n+1, vector<double>(n+1, 0.0));

for(int i = 0; i <= n; i++)
    dd[i][0] = y_all[i];

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

double fx = dd[0][0];
double term = 1.0;

for(int i = 1; i < n; i++)
{
    term *= (X - x[i-1]);
    fx += term * dd[0][i];
}

double prod = 1.0;
for(int i = 0; i < n; i++)
    prod *= (X - x[i]);

double error = fabs(dd[0][n] * prod);

cout << "Test Case " << tc << "\n";
cout << "Divided Difference Table:\n";

for(int i = 0; i <= n; i++)
{
    for(int j = 0; j <= n - i; j++)
        cout << setw(12) << dd[i][j];
    cout << "\n";
}

cout << "Interpolated value f(" << X << ") = " << fx << "\n";
cout << "Truncation Error = " << error << "\n\n";

outputFile << "Test Case " << tc << "\n";
outputFile << "Interpolated value f(" << X << ") = " << fx << "\n";
```

```
        outputFile << "Truncation Error = " << error << "\n\n";
    }

    inputFile.close();
    outputFile.close();
    return 0;
}
```

Divided Difference Input

```
1
4
0 1 2 3
0 1 8 27
4
70
2.5
```

Divided Difference Output

```
Test Case 1
Interpolated value f(2.50) = 15.62
Truncation Error = 0.23
```

Back to Contents

Solution of Curve Fitting Model

Least Square Regression Method For Linear Equations Method

Least Square Regression Method For Linear Equations Theory

Method used **Least Squares Regression – Linear Equation**

Objective To fit a straight line of the form

`y = a + bx`

that best represents the given experimental data.

Data Requirement A set of `n` observed data points:

`(x1, y1), (x2, y2), ..., (xn, yn)`

Core Idea The best-fitting straight line is obtained by minimizing the **sum of squares of vertical deviations** (errors) between the observed data points and the assumed line.

Assumed Model

$y = a + bx$

Least Squares Conditions To minimize error:

$(y - a - bx)^2 \rightarrow \text{minimum}$

This leads to the **normal equations**:

$$y = na + b\,x$$
$$xy = a\,x + b\,x^2$$

Evaluation Process The two normal equations are solved simultaneously to determine constants `a` and `b`.

Accuracy Considerations

- Simple and effective for linear trends
- Not suitable for nonlinear data
- Accuracy depends on data distribution

Applicability

- Widely used for trend analysis
- Useful when data follows an approximately linear pattern

Least Square Regression Method For Linear Equations Code

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

int main() {

    // File Handling
    ifstream in("input.txt");
    ofstream out("output.txt");

    if (!in) {
        cerr << "Error: input.txt not found\n";
        return 1;
    }

    int n;
    in >> n;
```

```
    vector<double> x(n), y(n);

    for (int i = 0; i < n; i++) {
        in >> x[i] >> y[i];
    }

    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];
    }

    // Least squares coefficients
    double b = (n * sumxy - sumx * sumy) / (n * sumx2 - sumx * sumx);
    double a = (sumy - b * sumx) / n;

    // Output
    out << "Linear Fit Equation:\n";
    out << "y = " << a << " + " << b << "x\n";

    return 0;
}
```

Least Square Regression Method For Linear Equations Input

```
5
1 50
2 80
3 96
4 120
5 45
```

Least Square Regression Method For Linear Equations Output

```
Linear Fit Equation:
y = 69.2 + 3x
```

Back to Contents

Least Square Regression Method For Transcendental Equations

Least Square Regression Method For Transcendental Equations Theory

Method used **Least Squares Regression – Transcendental Equation**

Objective To fit nonlinear relationships by transforming them into linear form so that least squares method can be applied.

Common Transcendental Forms

1. **Exponential model**

```
y = ae^(bx)
```

2. **Power model**

```
y = ax^b
```

Linearization Technique

Exponential Equation Taking natural logarithm:

```
ln y = ln a + bx
```

Let:

```
Y = ln y , A = ln a
```

Then:

```
Y = A + bx
```

Power Equation Taking logarithm on both sides:

```
log y = log a + b log x
```

Let:

```
Y = log y , X = log x , A = log a
```

Then:

```
Y = A + bX
```

Evaluation Process

- Transform the given data
- Apply linear least squares regression
- Compute constants
- Convert back to original form

Accuracy Considerations

- Transformation may amplify errors
- Requires positive data values
- Fit quality depends on correct model assumption

Applicability

- Used in population growth, decay processes, and empirical laws
- Suitable for nonlinear experimental data

Least Square Regression Method For Transcendental Equations Code

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

int main() {

    // File Handling
    ifstream in("input.txt");
    ofstream out("output.txt");

    if (!in) {
        cerr << "Error: input.txt not found\n";
        return 1;
    }

    int n;
    in >> n;

    vector<double> x(n), y(n);
    for (int i = 0; i < n; i++) {
        in >> x[i] >> y[i];
    }

    // Take log of y
    vector<double> Y(n);
    for (int i = 0; i < n; i++) {
        Y[i] = log(y[i]);   // ln(y)
    }

    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];
    }

    // Least squares for Y = A + b x
    double b = (n * sumxY - sumx * sumY) / (n * sumx2 - sumx * sumx);
```

```
    double A = (sumY - b * sumx) / n;
    double a = exp(A);

    // Output
    out << "Transcendental (Exponential) Fit:\n";
    out << "y = " << a << " * e^(" << b << "x)\n";

    return 0;
}
```

Least Square Regression Method For Transcendental Equations Input

```
5
1 50
2 80
3 96
4 120
5 45
```

Least Square Regression Method For Transcendental Equations Output

```
Transcendental (Exponential) Fit:
y = 68.8608 * e^(0.0194744x)
```

Back to Contents

Least Square Regression Method For Polynomial Equations

Least Square Regression Method For Polynomial Equations Theory

Method used Least Squares Regression – Polynomial Equation

Objective To fit a polynomial curve when linear regression is insufficient to represent data trends.

Assumed Model (Second Order Polynomial)

$y = a + bx + cx^2$

Data Requirement A set of experimental observations:

`(x1, y1), (x2, y2), ..., (xn, yn)`

Core Idea The coefficients of the polynomial are determined by minimizing the sum of squared deviations between observed and computed values.

Normal Equations For a second-degree polynomial:

$$y = na + b\,x + c\,x^2$$
$$xy = a\,x + b\,x^2 + c\,x^3$$
$$x^2y = a\,x^2 + b\,x^3 + c\,x$$

Evaluation Process

- Compute required summations from data table
- Solve the system of equations
- Substitute coefficients into polynomial

Accuracy Considerations

- Higher degree improves fit but may cause overfitting
- Computational complexity increases with degree
- Sensitive to data errors

Applicability

- Used when data shows curvature
- Suitable for engineering and experimental modeling

Least Square Regression Method For Polynomial Equations Code

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

// Solve nXn linear system using Gauss-Jordan elimination
vector<double> solveN(vector<vector<double>> A, vector<double> B) {

    int n = A.size();

    for (int i = 0; i < n; i++) {

        // Pivot selection
        int pivot = i;
        for (int j = i + 1; j < n; j++) {
            if (fabs(A[j][i]) > fabs(A[pivot][i])) {
                pivot = j;
            }
        }

        swap(A[i], A[pivot]);
        swap(B[i], B[pivot]);

        double div = A[i][i];
```

```
        if (fabs(div) < 1e-12) {
            cout << "Singular system detected. No unique solution." << endl;
            return {};
        }

        // Normalize pivot row
        for (int j = 0; j < n; j++)
            A[i][j] /= div;
        B[i] /= div;

        // Eliminate other rows
        for (int j = 0; j < n; j++) {
            if (j != i) {
                double factor = A[j][i];
                for (int k = 0; k < n; k++)
                    A[j][k] -= factor * A[i][k];
                B[j] -= factor * B[i];
            }
        }
    }

    return B;
}


int main() {

    // File Handling
    ifstream in("input.txt");
    ofstream out("output.txt");

    if (!in) {
        cerr << "Error: input.txt not found\n";
        return 1;
    }

    int n;
    in >> n;

    vector<double> x(n), y(n);
    for (int i = 0; i < n; i++)
        in >> x[i] >> y[i];

    // Required summations
    double sx = 0, sx2 = 0, sx3 = 0, sx4 = 0;
    double sy = 0, sxy = 0, sx2y = 0;
```

```
    for (int i = 0; i < n; i++) {
        sx   += x[i];
        sx2  += x[i] * x[i];
        sx3  += x[i] * x[i] * x[i];
        sx4  += x[i] * x[i] * x[i] * x[i];
        sy   += y[i];
        sxy  += x[i] * y[i];
        sx2y += x[i] * x[i] * y[i];
    }

    // Normal equations
    vector<vector<double>> A = {
        { double(n), sx,  sx2 },
        { sx,  sx2, sx3 },
        { sx2, sx3, sx4 }
    };

    vector<double> B = { sy, sxy, sx2y };

    vector<double> sol = solveN(A, B);

    if (sol.empty()) {
    cout << "Solution could not be computed.\n";
    return 0;
    }

    // ---- Output ----
    out << "Quadratic Polynomial Fit:\n";
    out << "y = " << sol[0]
        << " + " << sol[1] << "x"
        << " + " << sol[2] << "x^2\n";

    return 0;
}
```

Least Square Regression Method For Polynomial Equations Input

```
5
1 6
2 11
3 18
4 27
5 38
```

Least Square Regression Method For Polynomial Equations Output

```
Quadratic Polynomial Fit:
y = 3 + 2x + 1x^2
```

Back to Contents

Solution of Differential Equations

Runge Kutta Method

Runge Kutta Theory

Method used Runge–Kutta Method (Classical 4th Order RK)

Objective To solve initial value problems of the form `dy/dx = f(x, y)` with initial condition `y(x0) = y0`.

INITIAL VALUE PROBLEM:

`dy/dx = f(x, y), y(x0) = y0`

Goal: Advance solution from `xn` to `x(n+1) = xn + h` with high accuracy.

RUNGE-KUTTA 4TH ORDER FORMULA: Compute slope estimates at different points:

```
k1 = f(xn, yn)
k2 = f(xn + h/2, yn + h*k1/2)
k3 = f(xn + h/2, yn + h*k2/2)
k4 = f(xn + h, yn + h*k3)
```

Update to next step:

`y(n+1) = yn + (h/6)(k1 + 2*k2 + 2*k3 + k4)`

Data Requirement

- Initial condition: `(x0, y0)`
- Differential equation: `dy/dx = f(x, y)`
- Step size: `h`
- Final x value: `xn`

Features

- Local truncation error of order `O(h^5)`; global error `O(h^4)`
- Does not require evaluation of higher derivatives
- Widely used as a standard one-step method for ODEs.

Runge Kutta Code

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

double f(double x, double y)
{
    return x + y;
}

double rungeKutta(double x0, double y0, double h, double xn)
{
    double x = x0;
    double y = y0;

    while (x  < xn)
    {
        double k1 = h * f(x, y);
        double k2 = h * f(x + h/2.0, y + k1/2.0);
        double k3 = h * f(x + h/2.0, y + k2/2.0);
        double k4 = h * f(x + h, y + k3);

        y = y + (k1 + 2*k2 + 2*k3 + k4) / 6.0;
        x = x + h;
    }

    return y;
}

int main()
{
    ifstream fin("Input.txt");
    ofstream fout("Output.txt");

    if (!fin.is_open())
    {
        cout << "Error: Input.txt not found\n";
        return 0;
    }

    double x0, y0, h, xn;
    fin >> x0 >> y0 >> h >> xn;

    double result = rungeKutta(x0, y0, h, xn);
```

```
    cout << "Runge-Kutta 4th Order Method\n";
    cout << "----------------------------\n";
    cout << "Differential equation: dy/dx = x + y\n";
    cout << "Initial condition: y(" << x0 << ") = " << y0 << "\n";
    cout << "Step size (h): " << h << "\n";
    cout << "Required value at x = " << xn << "\n";
    cout << "Computed solution: y(" << xn << ") = " << result << "\n";

    fout << "Runge-Kutta 4th Order Method\n";
    fout << "----------------------------\n";
    fout << "Differential equation: dy/dx = x + y\n";
    fout << "Initial condition: y(" << x0 << ") = " << y0 << "\n";
    fout << "Step size (h): " << h << "\n";
    fout << "Required value at x = " << xn << "\n";
    fout << "Computed solution: y(" << xn << ") = " << result << "\n";

    fin.close();
    fout.close();

    return 0;
}
```

Runge Kutta Input

```
0 1 0.1 1
```

Runge Kutta Output

```
Runge-Kutta 4th Order Method
----------------------------
Differential equation: dy/dx = x + y
Initial condition: y(0) = 1
Step size (h): 0.1
Required value at x = 1
Computed solution: y(1) = 3.43656
```

Back to Contents

Numerical Differentiation By Forward Interpolation Method

Numerical Differentiation By Forward Interpolation Theory

Method used Numerical Differentiation using Forward Interpolation

Objective To approximate derivatives f'(x), f''(x), … from tabulated values of f(x) when an explicit analytic form is not available.

Data Requirement

- Tabulated values (x , y) with equal spacing: h = x - x

Basic Idea

1. Construct an interpolating polynomial for f(x) using either:
   - Newton's forward interpolation (when differentiating near the beginning of the table), or
   - Newton's backward interpolation (when differentiating near the end of the table).
2. Differentiate the interpolating polynomial analytically and then evaluate the derivative at the required point.

Example: First Derivative – Forward Formula at x Using the Newton forward polynomial and differentiating, one obtains a formula of the form

f'(x) (1/h)(a Δy + a Δ²y + a Δ³y +)

where Δ y are forward differences and a are known constants depending on the chosen order of approximation.

Second Derivative Approximation By differentiating the interpolating polynomial twice, we obtain formulas for the second derivative.

- **At the beginning of the table (forward / using forward differences):** f''(x) Δ²y /h² = (y - 2y + y)/h²

Higher-order formulas (involving more difference terms) can be derived in the same way when greater accuracy is required.

Interpolation Formulas Used For reference, the interpolation polynomials used in numerical differentiation are:

- **Forward interpolation (around x):** f(x) y + uΔy + [u(u-1)/2!]Δ²y + [u(u-1)(u-2)/3!]Δ³y + where u = (x - x)/h

Features

- Allows estimation of derivatives using only function values.
- Accuracy depends on the step size h and smoothness of f(x)

Numerical Differentiation By Forward Interpolation Code

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

long long fact(int n)
{
```

```
    if(n == 0 || n == 1) return 1;
    return n * fact(n - 1);
}

double g(double x)
{
    return exp(x) + x*x*x + sin(x);
}

double g_prime(double x)
{
    return exp(x) + 3*x*x + cos(x);
}

double g_double_prime(double x)
{
    return exp(x) + 6*x - sin(x);
}

vector<vector<double>> buildDiffTable(vector<double>& values)
{
    int sz = values.size();
    vector<vector<double>> table(sz, vector<double>(sz, 0.0));

    for(int i = 0; i < sz; i++)
        table[i][0] = values[i];

    for(int j = 1; j < sz; j++)
        for(int i = 0; i < sz - j; i++)
            table[i][j] = table[i + 1][j - 1] - table[i][j - 1];

    return table;
}

void process(int caseNo, ifstream& fin, ofstream& fout)
{
    int intervalCount;
    fin >> intervalCount;

    double startPoint, endPoint;
    fin >> startPoint >> endPoint;

    double evalPoint;
    fin >> evalPoint;

    double step = (endPoint - startPoint) / intervalCount;
```

```
vector<double> grid(intervalCount), funcValues(intervalCount);
for(int i = 0; i < intervalCount; i++)
{
    grid[i] = startPoint + i * step;
    funcValues[i] = g(grid[i]);
}

vector<vector<double>> diff = buildDiffTable(funcValues);

double u = (evalPoint - grid[0]) / step;

double approxFirst =
    ( diff[0][1]
    + (2*u - 1) * diff[0][2] / fact(2)
    + (3*u*u - 6*u + 2) * diff[0][3] / fact(3)
    ) / step;

double approxSecond =
    ( diff[0][2]
    + (u - 1) * diff[0][3]
    ) / (step * step);

double exactFirst = g_prime(evalPoint);
double exactSecond = g_double_prime(evalPoint);

double errorFirst = fabs((exactFirst - approxFirst) / exactFirst) * 100.0;
double errorSecond = fabs((exactSecond - approxSecond) / exactSecond) * 100.0;

cout << "\nTEST CASE #" << caseNo << "\n";
cout << "Difference Table:\n";
for(int i = 0; i < intervalCount; i++)
{
    for(int j = 0; j < intervalCount; j++)
        cout << setw(12) << diff[i][j];
    cout << "\n";
}

cout << fixed << setprecision(6);
cout << "Numerical g'(x)  = " << approxFirst << "\n";
cout << "Exact g'(x)      = " << exactFirst << "\n";
cout << "Numerical g''(x) = " << approxSecond << "\n";
cout << "Exact g''(x)     = " << exactSecond << "\n";
cout << "Error in g'(x)   = " << errorFirst << "%\n";
cout << "Error in g''(x)  = " << errorSecond << "%\n";
```

```
    fout << "\nTEST CASE #" << caseNo << "\n";
    fout << fixed << setprecision(6);
    fout << "Numerical g'(x)  = " << approxFirst << "\n";
    fout << "Exact g'(x)      = " << exactFirst << "\n";
    fout << "Numerical g''(x) = " << approxSecond << "\n";
    fout << "Exact g''(x)     = " << exactSecond << "\n";
    fout << "Error in g'(x)   = " << errorFirst << "%\n";
    fout << "Error in g''(x)  = " << errorSecond << "%\n";
}

int main()
{


    ifstream fin("input.txt");
    ofstream fout("output.txt");



    int totalCases;
    fin >> totalCases;

    cout << "Total Test Cases: " << totalCases << "\n";
    fout << "Total Test Cases: " << totalCases << "\n";

    for(int i = 1; i <= totalCases; i++)
        process(i, fin, fout);

    fin.close();
    fout.close();


    return 0;
}
```

Numerical Differentiation By Forward Interpolation Input

```
1
6
1.0 1.6
1.15
```

Numerical Differentiation By Forward Interpolation Output

```
Total Test Cases: 1
```

```
TEST CASE #1
Numerical g'(x)  = 7.534179
Exact g'(x)      = 7.534180
Numerical g''(x) = 9.153914
Exact g''(x)     = 9.145429
Error in g'(x)   = 0.000022%
Error in g''(x)  = 0.092775%
```

Back to Contents

Numerical Differentiation By Backward Interpolation Method

Numerical Differentiation By Backward Interpolation Theory

Method used Numerical Differentiation using Backward Interpolation

Objective To approximate derivatives f'(x), f''(x), … from tabulated values of f(x) when an explicit analytic expression of the function is not available.

Data Requirement

- Tabulated values (x , y) with equal spacing: h = x − x

Basic Idea

1. Construct an interpolating polynomial for f(x) using Newton's backward interpolation, which is suitable when the required derivative is near the end of the data table.
2. Differentiate the backward interpolation polynomial analytically and evaluate the derivative at the required point.

Example: First Derivative – Backward Formula at x By differentiating the Newton backward interpolation polynomial, the first derivative at x can be approximated as

f'(x) (1/h)(b y + b ²y + b ³y +)

where y are backward differences and b are known constants depending on the order of approximation.

Second Derivative Approximation Differentiating the backward interpolation polynomial twice yields an expression for the second derivative.

- At the end of the table (backward / using backward differences):

f''(x) ²y /h² = (y − 2y + y)/h²

Higher-order derivative formulas can be obtained by including additional backward difference terms to improve accuracy.

Interpolation Formulas Used For reference, the backward interpolation polynomial used in numerical differentiation is:

f(x) y + u y + [u(u+1)/2!] ^{2}y + [u(u+1)(u+2)/3!] ^{3}y +

where u = (x − x)/h

Features

- Enables numerical differentiation when only discrete data values are available.
- Provides better accuracy when the evaluation point is close to the last tabulated value.
- Accuracy depends on step size h and the smoothness of the underlying function.

Numerical Differentiation By Backward Interpolation Code

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

long long fact(int n)
{
    if(n == 0 || n == 1) return 1;
    return n * fact(n - 1);
}

double f(double x)
{
    return x*x + sin(x);
}

double f1(double x)
{
    return 2*x + cos(x);
}

double f2(double x)
{
    return 2 - sin(x);
}

vector<vector<double>> backwardDiffTable(vector<double>& values)
{
    int n = values.size();
    vector<vector<double>> table(n, vector<double>(n, 0.0));
```

```
    for(int i = 0; i < n; i++)
        table[i][0] = values[i];

    for(int j = 1; j < n; j++)
        for(int i = n - 1; i >= j; i--)
            table[i][j] = table[i][j - 1] - table[i - 1][j - 1];

    return table;
}

void solve(int tc, ifstream& fin, ofstream& fout)
{
    int n;
    fin >> n;

    double a, b;
    fin >> a >> b;

    double X;
    fin >> X;

    double h = (b - a) / n;

    vector<double> x(n), y(n);
    for(int i = 0; i < n; i++)
    {
        x[i] = a + i * h;
        y[i] = f(x[i]);
    }

    vector<vector<double>> diff = backwardDiffTable(y);

    double u = (X - x[n - 1]) / h;

    double dydx =
        ( diff[n - 1][1]
        + (2*u + 1) * diff[n - 1][2] / fact(2)
        + (3*u*u + 6*u + 2) * diff[n - 1][3] / fact(3)
        ) / h;

    double d2ydx2 =
        ( diff[n - 1][2]
        + (u + 1) * diff[n - 1][3]
        ) / (h * h);

    double exact1 = f1(X);
```

```
    double exact2 = f2(X);

    double error1 = fabs((exact1 - dydx) / exact1) * 100.0;
    double error2 = fabs((exact2 - d2ydx2) / exact2) * 100.0;

    cout << "\nTEST CASE #" << tc << "\n";
    cout << "Backward Difference Table:\n";
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
            cout << setw(12) << diff[i][j];
        cout << "\n";
    }

    cout << fixed << setprecision(6);
    cout << "Numerical f'(x)  = " << dydx << "\n";
    cout << "Exact f'(x)      = " << exact1 << "\n";
    cout << "Numerical f''(x) = " << d2ydx2 << "\n";
    cout << "Exact f''(x)     = " << exact2 << "\n";
    cout << "Error in f'(x)   = " << error1 << "%\n";
    cout << "Error in f''(x)  = " << error2 << "%\n";

    fout << "\nTEST CASE #" << tc << "\n";
    fout << fixed << setprecision(6);
    fout << "Numerical f'(x)  = " << dydx << "\n";
    fout << "Exact f'(x)      = " << exact1 << "\n";
    fout << "Numerical f''(x) = " << d2ydx2 << "\n";
    fout << "Exact f''(x)     = " << exact2 << "\n";
    fout << "Error in f'(x)   = " << error1 << "%\n";
    fout << "Error in f''(x)  = " << error2 << "%\n";
}

int main()
{


    ifstream fin("input.txt");
    ofstream fout("output.txt");



    int t;
    fin >> t;

    cout << "Total Test Cases: " << t << "\n";
    fout << "Total Test Cases: " << t << "\n";
```

```
    for(int i = 1; i <= t; i++)
        solve(i, fin, fout);

    fin.close();
    fout.close();


    return 0;
}
```

Numerical Differentiation By Backward Interpolation Input

```
1
6
1.0 1.6
1.15
```

Numerical Differentiation By Backward Interpolation Output

```
Total Test Cases: 1

TEST CASE #1
Numerical f'(x)  = 2.709370
Exact f'(x)      = 2.708487
Numerical f''(x) = 1.070054
Exact f''(x)     = 1.087236
Error in f'(x)   = 0.032598%
Error in f''(x)  = 1.580300%
```

Back to Contents

Solution of Numerical Integrations

Simpson's One-Third Rule

Simpson's One-Third Rule Theory

Objective To approximate the definite integral of a function using parabolic interpolation.

Data Requirement A polynomial equation of `n` degree:

- $a\,x\ + a\ \ x^{\ 1} + \ldots + a\,x^2 + a\,x + a\ = 0$

Integration limits: `[a, b]` and number of subintervals `n` (must be even)

Core Idea Fit a parabola → to approximate the curve
Integrate the parabola → to approximate the area

Divides the interval into an even number of subintervals and fits parabolas through groups of three consecutive points. Once a parabola is fitted through three points, that parabola can be integrated exactly using basic calculus. **Formula:**

$$\int f(x)dx \approx \frac{h}{3}[f(x_0) + 4\sum_{i=odd} f(x_i) + 2\sum_{i=even} f(x_i) + f(x_n)]$$

where h = (b-a)/n

Simpson's One-Third Rule Code

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

double f(double x, vector<double>& coef) {
    int n = coef.size() - 1;
    double s = 0;
    for (int i = 0; i <= n; i++)
        s += coef[i] * pow(x, n - i);
    return s;
}
void printEq(ofstream &out, vector<double>& coef) {
    int n = coef.size() - 1;
    out << "Equation: ";
    for (int i = 0; i <= n; i++) {
        out << coef[i] << "*x^" << (n - i);
        if (i != n) out << " + ";
    }
    out << endl;
}
int main() {
    ifstream in("input.txt");
    ofstream out("output.txt");

    int deg, n;
    double a, b, h, result = 0;

    in >> deg;
    vector<double> coef(deg + 1);
    for (int i = 0; i <= deg; i++) in >> coef[i];

    in >> a >> b >> n;
    printEq(out,coef);
```

```
    if (n % 2 != 0) {
        out << "Number of subintervals must be even\n";
        return 0;
    }

    h = (b - a) / n;

    result = f(a, coef) + f(b, coef);

    for (int i = 1; i < n; i++) {
        double x = a + i * h;
        if (i % 2 == 0) result += 2 * f(x, coef);
        else result += 4 * f(x, coef);
    }

    result *= h / 3.0;

    out << "Simpson 1/3 Rule Result = " << result << endl;
    return 0;
}
```

Simpson's One-Third Rule Input

```
3
1 2 0 1
0 1 6
```

Simpson's One-Third Rule Output

```
Equation: 1*x^3 + 2*x^2 + 0*x^1 + 1*x^0
Simpson 1/3 Rule Result = 1.91667
```

Back to Contents

Simpson's Three-Eighths Rule

Simpson's Three-Eighths Rule Theory

Objective To approximate the definite integral of a function using cubic interpolation.

Data Requirement A polynomial equation of `n` degree:

- $a\,x\ + a\ \ x^{\ 1} + \ldots + a\,x^2 + a\,x + a\ = 0$

Integration limits: `[a, b]` and number of subintervals `n` (must be multiple of 3)

Core Idea Divides the interval into subintervals (multiple of 3) and fits cubic polynomials through groups of four consecutive points. Provides higher accuracy for functions with higher-order derivatives.
Formula:

$$\int f(x)dx \approx (3h/8)[f(x_0) + 3\Sigma f(x_i) + 2\Sigma f(x_i) + f(x_n)]$$
$$(i \neq 3k) \qquad (i=3k)$$

where h = (b-a)/n

Simpson's Three-Eighths Rule Code

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

double f(double x, vector<double>& coef) {
    int n = coef.size() - 1;
    double s = 0;
    for (int i = 0; i <= n; i++)
        s += coef[i] * pow(x, n - i);
    return s;
}
void printEq(ofstream &out, vector<double>& coef) {
    int n = coef.size() - 1;
    out << "Equation: ";
    for (int i = 0; i <= n; i++) {
        out << coef[i] << "*x^" << (n - i);
        if (i != n) out << " + ";
    }
    out << endl;
}
int main() {
    ifstream in("input.txt");
    ofstream out("output.txt");

    int deg, n;
    double a, b, h, result = 0;

    in >> deg;
    vector<double> coef(deg + 1);
    for (int i = 0; i <= deg; i++) in >> coef[i];

    in >> a >> b >> n;
    printEq(out,coef);
    if (n % 3 != 0) {
        out << "Number of subintervals must be multiple of 3\n";
        return 0;
```

```
    }

    h = (b - a) / n;

    result = f(a, coef) + f(b, coef);

    for (int i = 1; i < n; i++) {
        double x = a + i * h;
        if (i % 3 == 0) result += 2 * f(x, coef);
        else result += 3 * f(x, coef);
    }

    result *= 3 * h / 8.0;

    out << "Simpson 3/8 Rule Result = " << result << endl;
    return 0;
}
```

Simpson's Three-Eighths Rule Input

```
4
1 0 0 1 1
0 3 6
```

Simpson's Three-Eighths Rule Output

```
Equation: 1*x^4 + 0*x^3 + 0*x^2 + 1*x^1 + 1*x^0
Simpson 3/8 Rule Result = 56.1562
```

Back to Contents

Download PDF

- Download this README as a PDF for offline viewing: README.pdf