**Numerical Analysis And Computation Lab Final**

Algorithm: Gauss Forward Interpolation

1. Input:
   * A set of data points (x0,y0),(x1,y1),…,(xn,yn) are equidistant.
   * The value x where you want to estimate y(x).
2. Preprocessing:
   * Compute the forward differences Δky0 using: Δyi = yi+1−yi .
   * Repeat this process to compute higher-order differences.
3. Determine the value of uuu:
   * Compute u=x−x0/ hu​​, where h is the uniform interval between xi.
4. Apply the Gauss Forward Interpolation Formula:

y(x)=y0+ uΔy0+ u(u−1)/ 2! \* Δ^2y0+u(u−1)(u−2)/ 3! \*Δ^3y0+ …………

Here:

* + u is the normalized difference.
  + Δ^ky0 are the forward differences.

1. Terminate:
   * Stop the computation once the required accuracy or the highest difference level (based on the number of data points) is reached.
2. Output:
   * The interpolated value y(x).

Applications

* Estimating values for functions where direct computation is difficult.
* Solving problems in physics, engineering, and other fields where interpolation is required.

**Code for guass forward interpolation:**

#include <iostream>

#include <vector>

#include <iomanip> // For formatting output

#include <cmath> // For factorial function

using namespace std;

// Function to calculate factorial of a number

int factorial(int n) {

if (n == 0 || n == 1) return 1;

return n \* factorial(n - 1);

}

int main() {

// Number of data points

int n;

cout << "Enter the number of data points: ";

cin >> n;

// Input x and y values

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

cout << "Enter the x and y values:\n";

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

cout << "x[" << i << "]: ";

cin >> x[i];

cout << "y[" << i << "]: ";

cin >> y[i];

}

// Input the value of x for which y(x) is to be estimated

double xp;

cout << "Enter the value of x to interpolate: ";

cin >> xp;

// Calculate forward differences

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

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];

}

}

// Display forward difference table (optional)

cout << "\nForward Difference Table:\n";

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

cout << setw(10) << x[i];

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

cout << setw(10) << diff[i][j];

}

cout << endl;

}

// Calculate u

double h = x[1] - x[0]; // Assuming equidistant x values

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

// Apply Gauss Forward Interpolation formula

double yp = y[0]; // Initial value

double uTerm = u;

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

yp += (uTerm \* diff[0][i]) / factorial(i);

uTerm \*= (u - i); // Update u term for next iteration

}

// Output the result

cout << "\nEstimated value of y at x = " << xp << " is: " << yp << endl;

return 0;

}

**Stirlings interpolation:**

**Algorithm** :

1. **Input:**
   * A set of nnn data points (x0,y0),(x1,y1),…,(xn−1,yn−1), where xi are equidistant.
   * The value x for which y(x) needs to be interpolated.
2. **Preprocessing:**
   * Calculate the forward differences Δy, Δ^2y, Δ^3y..., up to the highest-order difference.
3. **Identify the Central Point:**
   * Find the central point xm such that xm ​ is the closest to x.
4. **Calculate the Value of uuu:**
   * Compute u=x−xm/ h ,where h is the uniform interval between xi.
5. **Apply Stirling’s Interpolation Formula:**
   * Stirling’s formula is given by:
   * y(x)=ym+ u/1! \*Δym+ u2/2!\* Δ^2ym+ u(u^2−1)/3!\*(Δ^3ym−1 + Δ^3ym)/2 + ……………
   * Alternate the terms from the central point xm and use both forward and backward differences.
6. **Compute Higher Terms (if needed):**
   * Include additional terms until the required precision is achieved.
7. **Output:**
   * The interpolated value y(x).

**Explanation of Terms:**

* Ym : Value of y at the central point.
* Δ^ky: Forward differences up to the k-th order.
* u: The normalized distance from the central point.

1. Stirling’s method alternates between forward and backward differences, combining them symmetrically around the central point.
2. This method is highly accurate for points near the center but may lose accuracy near the boundaries.

**Code for stirlings:**

#include <iostream>

#include <vector>

#include <cmath>

#include <iomanip> // For formatting output

using namespace std;

// Function to calculate factorial of a number

int factorial(int n) {

if (n == 0 || n == 1) return 1;

return n \* factorial(n - 1);

}

int main() {

// Number of data points

int n;

cout << "Enter the number of data points: ";

cin >> n;

// Input x and y values

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

cout << "Enter the x and y values:\n";

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

cout << "x[" << i << "]: ";

cin >> x[i];

cout << "y[" << i << "]: ";

cin >> y[i];

}

// Input the value of x to interpolate

double xp;

cout << "Enter the value of x to interpolate: ";

cin >> xp;

// Step 1: Compute the forward differences table

vector<vector<double>> diff(n, vector<double>(n, 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];

}

}

// Display forward difference table (optional)

cout << "\nForward Difference Table:\n";

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

cout << setw(10) << x[i];

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

cout << setw(10) << diff[i][j];

}

cout << endl;

}

// Step 2: Find the central point (closest to xp)

int m = 0;

double minDiff = fabs(xp - x[0]);

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

if (fabs(xp - x[i]) < minDiff) {

minDiff = fabs(xp - x[i]);

m = i;

}

}

// Step 3: Calculate u and h

double h = x[1] - x[0]; // Assuming equidistant x values

double u = (xp - x[m]) / h;

// Step 4: Apply Stirling's Interpolation Formula

double yp = y[m]; // Start with y\_m

double uTerm = u;

int k = 1;

// Add terms involving even-order differences

for (int i = 1; i <= n / 2; ++i) {

if (m - i >= 0 && m + i < n) {

yp += (uTerm \* (diff[m - i][2 \* i - 1] + diff[m][2 \* i - 1]) / 2) / factorial(2 \* i - 1);

uTerm \*= (u \* u - (i \* i));

}

}

// Add terms involving odd-order differences

uTerm = u \* (u \* u - 1);

for (int i = 1; i <= n / 2; ++i) {

if (m - i >= 0 && m + i < n) {

yp += (uTerm \* diff[m - i][2 \* i]) / factorial(2 \* i);

uTerm \*= (u \* u - i \* i);

}

}

// Step 5: Output the interpolated value

cout << "\nInterpolated value of y at x = " << xp << " is: " << yp << endl;

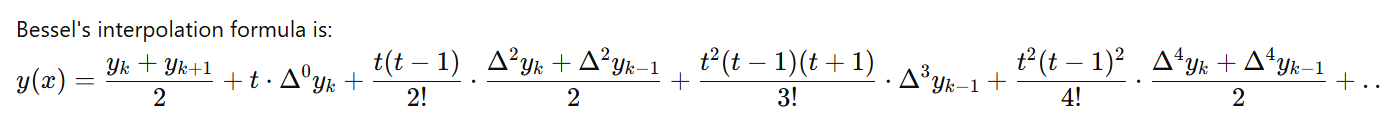
return 0;

}

**Bessels Interpolation:**

**Algorithm:**

1. **Given Data**:
   * A set of evenly spaced data points (x0,y0), (x1,y1),…, (xn,yn).
   * The value x at which y(x) needs to be interpolated.
   * Step size h=x1−x0
2. **Find the central index**:
   * Identify the central point xk​ is closest to x.
   * Let t=x−xk/ h ​​ (normalized distance from the central point).
3. **Compute central differences**:
4. **Apply Bessel’s formula**:



1. **Stop when the terms become negligible**:
   * Continue adding terms from the formula until the desired accuracy is achieved or the additional terms become negligible.
2. **Return the interpolated value y(x).**

**Steps in Practice:**

1. Construct the central difference table.
2. Identify k such that xk​ is closest to x.
3. Use Bessel’s formula step by step to calculate the interpolated value.

**Code for bessels**

#include <iostream>

#include <vector>

#include <iomanip>

using namespace std;

// Function to calculate central difference table

void calculateCentralDifferences(vector<vector<double>> &differenceTable, int n) {

for (int j = 1; j < n; ++j) {

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

differenceTable[i][j] = differenceTable[i + 1][j - 1] - differenceTable[i][j - 1];

}

}

}

// Bessel's interpolation function

double besselsInterpolation(const vector<double> &x, const vector<double> &y, double targetX, double h) {

int n = x.size();

// Find the central index

int k = 0;

for (int i = 0; i < n - 1; ++i) {

if (x[i] <= targetX && x[i + 1] >= targetX) {

k = i;

break;

}

}

if (k == n - 1) --k; // Handle edge case

// Calculate t

double t = (targetX - x[k]) / h;

// Prepare the central difference table

vector<vector<double>> differenceTable(n, vector<double>(n, 0));

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

differenceTable[i][0] = y[i];

}

calculateCentralDifferences(differenceTable, n);

// Bessel's interpolation formula

double result = (differenceTable[k][0] + differenceTable[k + 1][0]) / 2;

result += t \* differenceTable[k][1];

result += t \* (t - 1) \* (differenceTable[k][2] + differenceTable[k - 1][2]) / (2 \* 2);

result += t \* t \* (t - 1) \* (t + 1) \* differenceTable[k - 1][3] / (3 \* 2);

result += t \* t \* (t - 1) \* (t - 1) \* (differenceTable[k][4] + differenceTable[k - 1][4]) / (24);

return result;

}

int main() {

// Input data

vector<double> x = {1, 2, 3, 4, 5};

vector<double> y = {2.5, 3.1, 3.9, 5.0, 6.2};

double targetX;

cout << "Enter the value of x to interpolate: ";

cin >> targetX;

// Step size (assumes evenly spaced x values)

double h = x[1] - x[0];

// Perform Bessel's interpolation

double interpolatedValue = besselsInterpolation(x, y, targetX, h);

// Output result

cout << fixed << setprecision(6);

cout << "The interpolated value at x = " << targetX << " is: " << interpolatedValue << endl;

return 0;

}

**Numerical differentiation using forward difference:**

**Algorithm:**

1. **Input**:
   * A function f(x) for a table of values for f(x).
   * The value x0 at which the derivative is to be computed.
   * The step size h (spacing between consecutive x-values).
2. **Forward Difference Formula**:
   * **First-order derivative**: f′(x0)≈f(x1)−f(x0)/h, where x1= x0 + h.
   * **Second-order derivative**: f′′(x0)≈f(x2)−2f(x1)+f(x0)/ h^2, where x2=x0 + 2h
3. **Procedure**:
   * For the first derivative, calculate the difference between the next point and the current point, divided by h.
   * For the second derivative, use the values at the next two points and the current point.
   * Ensure the step size h is consistent and small enough for accuracy.
4. **Output**:
   * Return the approximated derivative(s).

**Code for forward difference:**

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

// Function to calculate the first derivative using the forward difference formula

double firstDerivative(const vector<double>& x, const vector<double>& y, double x0) {

double h = x[1] - x[0]; // Assume equally spaced points

int index = -1;

// Find the index of x0 in the array

for (int i = 0; i < x.size(); ++i) {

if (fabs(x[i] - x0) < 1e-6) { // Tolerance for floating-point comparison

index = i;

break;

}

}

if (index == -1 || index + 1 >= x.size()) {

throw invalid\_argument("x0 is out of range or forward difference cannot be applied.");

}

return (y[index + 1] - y[index]) / h;

}

// Function to calculate the second derivative using the forward difference formula

double secondDerivative(const vector<double>& x, const vector<double>& y, double x0) {

double h = x[1] - x[0]; // Assume equally spaced points

int index = -1;

// Find the index of x0 in the array

for (int i = 0; i < x.size(); ++i) {

if (fabs(x[i] - x0) < 1e-6) {

index = i;

break;

}

}

if (index == -1 || index + 2 >= x.size()) {

throw invalid\_argument("x0 is out of range or forward difference cannot be applied.");

}

return (y[index + 2] - 2 \* y[index + 1] + y[index]) / (h \* h);

}

int main() {

// Example data: x and f(x)

vector<double> x = {1, 1.5, 2, 2.5, 3};

vector<double> y = {2.718, 4.4817, 7.3891, 12.1825, 20.0855};

double x0 = 1.5;

try {

double first = firstDerivative(x, y, x0);

double second = secondDerivative(x, y, x0);

cout << "First derivative at x = " << x0 << ": " << first << endl;

cout << "Second derivative at x = " << x0 << ": " << second << endl;

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

**Newton backward differentiation:**

**Algo:**

1. **Input**:
   * A set of evenly spaced x-values and their corresponding f(x)-values.
   * The value x0​ where the derivative needs to be calculated.
   * The step size h (difference between consecutive x-values).
2. **Backward Difference Formulas**:
   * **First-order derivative**: f′(x0)≈f(x0)−f(x−1)/h,where x−1=x0 − h.
   * **Second-order derivative**: f′′(x0)≈f(x0)−2f(x−1)+f(x−2)/h^2,where x−2=x0 − 2h
3. **Procedure**:
   * Identify the index iii of x0 in the dataset.
   * For the first derivative, compute the difference between the current point and the previous point, divided by hhh.
   * For the second derivative, use values at x0x​, x−1, and x−2.
4. **Output**:
   * Return the approximated derivative(s).

**Code backward difference:**

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

// Function to calculate the first derivative using the backward difference formula

double firstDerivative(const vector<double>& x, const vector<double>& y, double x0) {

double h = x[1] - x[0]; // Assume equally spaced points

int index = -1;

// Find the index of x0 in the array

for (int i = 0; i < x.size(); ++i) {

if (fabs(x[i] - x0) < 1e-6) { // Tolerance for floating-point comparison

index = i;

break;

}

}

if (index == -1 || index - 1 < 0) {

throw invalid\_argument("x0 is out of range or backward difference cannot be applied.");

}

return (y[index] - y[index - 1]) / h;

}

// Function to calculate the second derivative using the backward difference formula

double secondDerivative(const vector<double>& x, const vector<double>& y, double x0) {

double h = x[1] - x[0]; // Assume equally spaced points

int index = -1;

// Find the index of x0 in the array

for (int i = 0; i < x.size(); ++i) {

if (fabs(x[i] - x0) < 1e-6) {

index = i;

break;

}

}

if (index == -1 || index - 2 < 0) {

throw invalid\_argument("x0 is out of range or backward difference cannot be applied.");

}

return (y[index] - 2 \* y[index - 1] + y[index - 2]) / (h \* h);

}

int main() {

// Example data: x and f(x)

vector<double> x = {1, 1.5, 2, 2.5, 3};

vector<double> y = {2.718, 4.4817, 7.3891, 12.1825, 20.0855};

double x0 = 2.5;

try {

double first = firstDerivative(x, y, x0);

double second = secondDerivative(x, y, x0);

cout << "First derivative at x = " << x0 << ": " << first << endl;

cout << "Second derivative at x = " << x0 << ": " << second << endl;

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

**Numerical differentiation using sterlings formula:**

**Algo:**

1. Input Data:
   * Collect evenly spaced xxx-values and their corresponding y=f(x) values.
   * Identify the point x0​ where the derivative needs to be calculated.
2. Find the Central Point:
   * Determine the middle value in the x-values (this is the central point).
   * Compute the parameter t, which represents the relative position of x0​ to the central point.
3. Construct the Difference Table:
   * Create a table where each row calculates the difference between consecutive values of y.
   * Continue until you’ve computed all higher-order differences.
4. Combine Differences:
   * Use the differences from the table to calculate approximations of the first and second derivatives.
   * Focus on the rows around the central point for better accuracy.
5. Stop When Accurate:
   * Continue combining terms until adding more differences has no significant impact on the result.
6. Output the Results:
   * Return the approximated values of the first and second derivatives at x0.

**Code:**

#include <iostream>

#include <vector>

#include <cmath>

#include <iomanip>

using namespace std;

// Function to construct the central difference table

vector<vector<double>> computeCentralDifferenceTable(const vector<double>& y) {

int n = y.size();

vector<vector<double>> diffTable(n, vector<double>(n, 0));

// Initialize the first column with y values

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

diffTable[i][0] = y[i];

}

// Compute the central difference table

for (int j = 1; j < n; ++j) {

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

diffTable[i][j] = diffTable[i + 1][j - 1] - diffTable[i][j - 1];

}

}

return diffTable;

}

// Stirling's formula for numerical differentiation

pair<double, double> stirlingDifferentiation(const vector<double>& x, const vector<double>& y, double x0) {

int n = x.size();

double h = x[1] - x[0]; // Step size

// Find the central point

int k = n / 2; // Assume n is odd for simplicity

double t = (x0 - x[k]) / h;

// Compute central difference table

vector<vector<double>> diffTable = computeCentralDifferenceTable(y);

// First derivative

double fPrime = 0.0;

fPrime += t \* diffTable[k][1];

fPrime += (diffTable[k][2] + diffTable[k - 1][2]) / (2 \* h);

fPrime += ((t \* t - 1) \* diffTable[k][3]) / (2 \* h);

fPrime /= h;

// Second derivative

double fDoublePrime = 0.0;

fDoublePrime += diffTable[k][2];

fDoublePrime += (t \* diffTable[k - 1][3]) / 2.0;

fDoublePrime /= (h \* h);

return {fPrime, fDoublePrime};

}

int main() {

// Example data: x and f(x)

vector<double> x = {1, 2, 3, 4, 5}; // Evenly spaced x values

vector<double> y = {2.0, 4.1, 8.3, 16.2, 32.5};

double x0 = 3.0; // Point of differentiation

try {

auto [firstDerivative, secondDerivative] = stirlingDifferentiation(x, y, x0);

cout << fixed << setprecision(6);

cout << "First derivative at x = " << x0 << ": " << firstDerivative << endl;

cout << "Second derivative at x = " << x0 << ": " << secondDerivative << endl;

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

**Numerical Differentiation Using Bessel's Formula**

**Algorithm**

1. **Input Data**:
   * Collect evenly spaced x-values and their corresponding y=f(x) values.
   * Specify the point x0​ where the derivative is to be calculated.
2. **Identify the Central Interval**:
   * Find the interval of x-values around x0 such that x0​ lies as close to the middle as possible.
3. **Construct the Central Difference Table**:
   * Calculate the differences (Δy) for the given y-values:
     + First differences (Δ^1y): Difference between adjacent yyy-values.
     + Second differences (Δ^2y): Difference between adjacent first differences.
     + Continue for higher-order differences.
4. **Compute Derivatives Using Bessel's Formula**:
   * Use the central differences to approximate:
     + The **first derivative** at x0.
     + The **second derivative** at x0​.
   * Focus on terms involving values near x0​ to achieve better accuracy.
5. **Stop When Accurate**:
   * Include terms from the difference table until the result stabilizes (adding more terms makes no significant change).
6. **Output the Results**:
   * Return the approximated first and second derivatives at x0​.

**Code:**

#include <iostream>

#include <vector>

#include <cmath>

#include <iomanip>

using namespace std;

// Function to construct the central difference table

vector<vector<double>> computeCentralDifferenceTable(const vector<double>& y) {

int n = y.size();

vector<vector<double>> diffTable(n, vector<double>(n, 0));

// Initialize the first column with y values

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

diffTable[i][0] = y[i];

}

// Compute the central difference table

for (int j = 1; j < n; ++j) {

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

diffTable[i][j] = diffTable[i + 1][j - 1] - diffTable[i][j - 1];

}

}

return diffTable;

}

// Bessel's formula for numerical differentiation

pair<double, double> besselsDifferentiation(const vector<double>& x, const vector<double>& y, double x0) {

int n = x.size();

double h = x[1] - x[0]; // Step size

// Find the central index

int k = n / 2; // Assume n is odd for simplicity

double t = (x0 - x[k]) / h;

// Compute central difference table

vector<vector<double>> diffTable = computeCentralDifferenceTable(y);

// Bessel's formula for first derivative

double fPrime = (diffTable[k][1] + diffTable[k - 1][1]) / 2.0;

fPrime += (t - 0.5) \* diffTable[k][2];

fPrime += (t \* (t - 1) \* (diffTable[k][3] + diffTable[k - 1][3])) / 6.0;

fPrime /= h;

// Bessel's formula for second derivative

double fDoublePrime = diffTable[k][2];

fDoublePrime += t \* (diffTable[k][3] + diffTable[k - 1][3]) / 2.0;

fDoublePrime /= h \* h;

return {fPrime, fDoublePrime};

}

int main() {

// Example data: x and f(x)

vector<double> x = {1, 2, 3, 4, 5}; // Evenly spaced x values

vector<double> y = {2.5, 3.1, 3.9, 5.0, 6.2};

double x0 = 3.0; // Point of differentiation

try {

auto [firstDerivative, secondDerivative] = besselsDifferentiation(x, y, x0);

cout << fixed << setprecision(6);

cout << "First derivative at x = " << x0 << ": " << firstDerivative << endl;

cout << "Second derivative at x = " << x0 << ": " << secondDerivative << endl;

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

**Trapezoidal formula:**

**Algo:**

 **Input**:

* Define the function f(x) you want to integrate.
* Specify the interval [a,b] over which to integrate.
* Choose the number of subintervals n to divide the range [a,b].

 **Divide the Interval**:

* Compute the width of each subinterval as h=b−a/n
* Determine the x-values at the boundaries of each subinterval.

 **Compute Area**:

* Calculate the function values at each x point.
* Add the first and last function values directly.
* Add the remaining intermediate function values twice (since they are shared between two trapezoids).

 **Multiply by the Step Size**:

* Multiply the sum by h/2 to get the final approximate integral.

 **Output**:

* Display the computed integral value.

Code:

#include <iostream>

#include <cmath>

#include <iomanip>

using namespace std;

// Function to integrate

double f(double x) {

return x \* x; // Example: f(x) = x^2

}

// Trapezoidal Rule function

double trapezoidalRule(double a, double b, int n) {

double h = (b - a) / n; // Width of each subinterval

double sum = 0.0;

// Calculate the sum of function values

sum += f(a); // First term

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

sum += 2 \* f(a + i \* h); // Intermediate terms

}

sum += f(b); // Last term

// Final integration result

return (h / 2) \* sum;

}

int main() {

double a = 0.0, b = 2.0; // Integration range [a, b]

int n = 4; // Number of subintervals

double result = trapezoidalRule(a, b, n);

cout << fixed << setprecision(6);

cout << "The approximate integral using the trapezoidal rule is: " << result << endl;

return 0;

}

**Simpsons formula :**

**Algo:**

 **Input**:

* Define the function f(x) to integrate.
* Specify the interval [a,b] over which integration is needed.
* Choose the number of subintervals nnn (must be even).

 **Divide the Interval**:

* Compute the width of each subinterval as h=b−a/n.
* Determine the x-values at the boundaries of each subinterval.

 **Calculate Function Values**:

* Compute f(x) at all x-points:
  + Add the function values at the first and last points directly.
  + Add the function values at odd-indexed points (multiplied by 4).
  + Add the function values at even-indexed points (multiplied by 2), excluding the first and last.

 **Multiply by the Step Size**:

* Multiply the sum by h/3​ to get the final integral.

 **Output**:

* Display the computed integral value.

**Code:**

#include <iostream>

#include <cmath>

#include <iomanip>

using namespace std;

// Function to integrate

double f(double x) {

return x \* x; // Example: f(x) = x^2

}

// Simpson's Rule function

double simpsonsRule(double a, double b, int n) {

if (n % 2 != 0) {

throw invalid\_argument("Number of subintervals (n) must be even.");

}

double h = (b - a) / n; // Width of each subinterval

double sum = f(a) + f(b); // First and last terms

// Compute the sum for odd-indexed points

for (int i = 1; i < n; i += 2) {

sum += 4 \* f(a + i \* h);

}

// Compute the sum for even-indexed points

for (int i = 2; i < n; i += 2) {

sum += 2 \* f(a + i \* h);

}

// Final integration result

return (h / 3) \* sum;

}

int main() {

double a = 0.0, b = 2.0; // Integration range [a, b]

int n = 6; // Number of subintervals (must be even)

try {

double result = simpsonsRule(a, b, n);

cout << fixed << setprecision(6);

cout << "The approximate integral using Simpson's rule is: " << result << endl;

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

**Euler’s Method:**

**Algorithm:**Start with initial values x0,y0;

Calculate yn+1 using the formula iteratively for n steps.

**Code:**

f = @(x,y) -2\*x\*y;

x0 = 0;

y0 = 1;

h = 0.1;

n = 10;

x = x0;

y = y0;

for i = 1:n

y = y+h\*f(x,y);

x = x + h;

disp([‘Step ‘, num2str(i), ‘ : x = ‘, num2str(x), ‘, y = ’, num2str(y)]);

end

**Rungge Kutta Method:**

**Code:**

f = @(x,y) -2\*x\*y;

x0 = 0;

y0 = 1;

h = 0.1;

n = 10;

x = x0;

y = y0;

for i = 1:n

k1 = f(x,y);

k2 = f(x + h/2, y + h/2 \* k1);

k3 = f(x + h/2, y + h/2 \* k2);

k4 = f(x + h, y + h \* k3);

y = y + h/6 \* (k1 + 2\*k2 + 2\*k3 + k4);

x = x + h;

disp([‘Step ‘, num2str(i), ‘ : x = ‘, num2str(x), ‘, y = ’, num2str(y)]);

end

**Plotting of a Graph(3 functions on a single figure):**

% Define the x values

x = linspace(-2\*pi, 2\*pi, 100); % 100 points from -2π to 2π

% Define the functions

y1 = sin(x); % Function 1: sin(x)

y2 = cos(x); % Function 2: cos(x)

y3 = sin(x) .\* cos(x); % Function 3: sin(x) \* cos(x)

% Create the figure

figure;

% Plot the first function

plot(x, y1, 'r--', 'LineWidth', 1.5); % Red dashed line

hold on; % Hold the plot to overlay other plots

% Plot the second function

plot(x, y2, 'b-', 'LineWidth', 1.5); % Blue solid line

% Plot the third function

plot(x, y3, 'g:', 'LineWidth', 1.5); % Green dotted line

% Add grid lines

grid on;

% Add labels and title

xlabel('x-axis'); % Label for x-axis

ylabel('y-axis'); % Label for y-axis

title('Comparison of Three Functions'); % Plot title

% Add legend

legend('sin(x)', 'cos(x)', 'sin(x) \* cos(x)', 'Location', 'best');

% Set axis limits for better visibility (optional)

xlim([-2\*pi, 2\*pi]);

ylim([-1.5, 1.5]);

% Display the figure

hold off;

**Matrix Addition, Subtraction, Product, and Transpose**

% Define two matrices

A = [1 2 3; 4 5 6; 7 8 9];

B = [9 8 7; 6 5 4; 3 2 1];

% Matrix addition

C\_add = A + B;

% Matrix subtraction

C\_sub = A - B;

% Matrix multiplication (standard matrix product)

C\_mul = A \* B;

% Transpose of matrices

A\_transpose = A';

B\_transpose = B';

% Display results

disp('Matrix A:');

disp(A);

disp('Matrix B:');

disp(B);

disp('A + B:');

disp(C\_add);

disp('A - B:');

disp(C\_sub);

disp('A \* B:');

disp(C\_mul);

disp('Transpose of A:');

disp(A\_transpose);

disp('Transpose of B:');

disp(B\_transpose);

**Determinant of a Matrix**

% Define a matrix

M = [4 2 1; 3 5 7; 8 9 6];

% Compute determinant

determinant = det(M);

% Display the determinant

disp('Matrix M:');

disp(M);

disp(['Determinant of M: ', num2str(determinant)]);

### **Check if a Matrix is Invertible and Compute Its Inverse**

% Define a matrix

M = [4 2 1; 3 5 7; 8 9 6];

% Check if the determinant is non-zero

determinant = det(M);

if determinant ~= 0

disp('The matrix M is invertible.');

% Compute the inverse

M\_inverse = inv(M);

% Display the inverse

disp('Inverse of M:');

disp(M\_inverse);

else

disp('The matrix M is not invertible.');

end