LAB 2

Experiment **No: 1**

TITLE: Lagrange’s Interpolating Polynomials.

OBJECTIVES

To implement Lagrange’s Interpolating Polynomials to approximate the value of a function at a given point in C programming.

**THEORY**

Lagrange’s Interpolating Polynomials are used to find a polynomial that passes through a given set of data points. The polynomial is constructed as a linear combination of basis polynomials, each of which is zero at all given points except one.

**Algorithm Steps:**

1. Input:
   * Number of data points n.
   * Arrays x and y of size n containing the data points.
   * The point xi at which to interpolate.
2. Initialize:
   * Set result=0.
3. Compute Lagrange Polynomial:
   * For each i*i* from 0 to n−1:
     + Set term=y[i]
     + For each j from 0 to n−1:
       - If j ≠ i, multiply term by (xi − x[j] ) / (x[i]−x[j])
     + Add term to result.
4. Output:
   * The interpolated value result at xi.

**Advantages:**

* Simple to implement.
* Does not require equally spaced data points.

**Limitations:**

* Computationally expensive for large datasets.
* Susceptible to Runge’s phenomenon for high-degree polynomials.

**Demostration**

*#include <stdio.h>*

*// Function to calculate Lagrange's interpolating polynomial*

*double lagrangeInterpolation(double x[], double y[], int n, double xi)*

*{*

*double result = 0.0;*

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

*{*

*double term = y[i]; // Initialize term with y[i]*

*for (int j = 0; j < n; j++)*

*{*

*if (j != i)*

*{*

*term \*= (xi - x[j]) / (x[i] - x[j]); // Multiply by (x - x\_j) / (x\_i - x\_j)*

*}*

*}*

*result += term; // Add the term to the result*

*}*

*return result;*

*}*

*int main()*

*{*

*int n;*

*printf("Enter the number of data points: ");*

*scanf("%d", &n);*

*double x[n], y[n];*

*printf("Enter the data points (x):\n");*

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

*{*

*printf("x[%d]: ", i);*

*scanf("%lf", &x[i]);*

*}*

*printf("Enter the data points (y):\n");*

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

*{*

*printf("y[%d]: ", i);*

*scanf("%lf", &y[i]);*

*}*

*double xi;*

*printf("Enter the point at which to interpolate (xi): ");*

*scanf("%lf", &xi);*

*// Perform interpolation*

*double yi = lagrangeInterpolation(x, y, n, xi);*

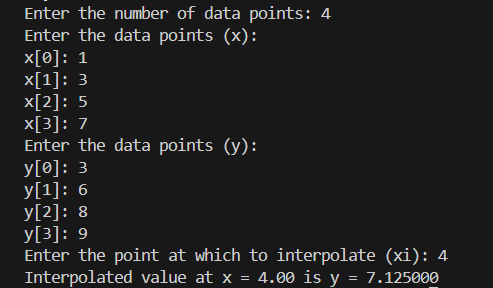
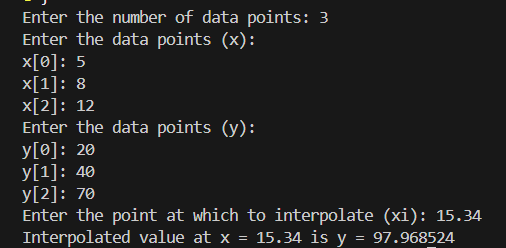
*// Print the result*

*printf("Interpolated value at x = %.2f is y = %.6f\n", xi, yi);*

*return 0;*

*}*

**Output 1: Output 2:**



**Result and Discussion**

The program implements Lagrange’s Interpolating Polynomials to approximate the value of a function at a user-specified point xi. The user provides the number of data points n, the arrays x and y, and the point xi at which to interpolate.

The program calculates

* the Interpolated value at x = 4.00 is y = 7.125000
* Interpolated value at x = 15.34 is y = 97.968524

**CONCLUSION**

Lagrange’s Interpolating Polynomials provide a simple and effective way to approximate the value of a function at a given point using a set of data points. While the method is easy to implement and does not require equally spaced data, it is not suitable for large datasets due to its computational complexity. This experiment demonstrates the practical implementation of Lagrange’s Interpolation in C programming and validates its effectiveness for interpolation tasks.

Experiment No: 2

**TITLE: Newton’s divided difference.**

**OBJECTIVES**

To implement **Newton’s Divided Difference Interpolation** to approximate the value of a function at a given point in C programming.

**THEORY**

Newton’s Divided Difference Interpolation is a method used to construct an interpolating polynomial for a given set of data points. It is based on the concept of divided differences, which are used to compute the coefficients of the polynomial.

Given n+1data points (x0,y0),(x1,y1),…,(xn,yn) the Newton’s interpolating polynomial P(x) is given by:

*P*(*x*)=*f*[*x*0]+(*x*−*x*0)*f*[*x*0,*x*1]+(*x*−*x*0)(*x*−*x*1)*f*[*x*0,*x*1,*x*2]+⋯+(*x*−*x*0)(*x*−*x*1)…(*x*−*xn*−1) *f*[*x*0,*x*1,…,*xn*]

where f[x0,x1,…,xk] are the divided differences, computed recursively as:

f[xi]=yi

​f[xi,xi+1,…,xi+k] = (f[xi+1,…,xi+k] − f[xi,…,xi+k−1] ) / (xi+k−xi )

**Algorithm Steps:**

1. **Input**:
   * Number of data points n*n*.
   * Arrays x*x* and y of size ncontaining the data points.
   * The point xi at which to interpolate.
2. **Compute Divided Differences**:
   * Initialize a 2D array dd to store divided differences.
   * Set dd[i][0]=y[i] for all i*.*
   * For each k from 1 to n−1:
     + For each i*i* from 0 to n−k−1:
       - Compute dd[i][k] = ( dd[i+1][k−1]−dd[i][k−1] ) / (x[i+k]−x[i] )
3. **Compute Interpolated Value**:
   * Set result=dd[0][0].
   * For each i from 1 to n−1:
     + Multiply result by (xi−x[i−1]) and add dd[0][i].
4. **Output**:
   * The interpolated value resultat xi.

**Advantages:**

* Efficient for adding new data points.
* Computationally less expensive than Lagrange’s method.

**Limitations:**

* Requires the data points to be distinct.

**Demostration**

*#include <stdio.h>*

*// Function to compute Newton's divided difference interpolation*

*double newtonDividedDifference(double x[], double y[], int n, double xi)*

*{*

*double dd[n][n]; // Divided difference table*

*// Initialize the divided difference table*

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

*{*

*dd[i][0] = y[i];*

*}*

*// Compute divided differences*

*for (int k = 1; k < n; k++)*

*{*

*for (int i = 0; i < n - k; i++)*

*{*

*dd[i][k] = (dd[i + 1][k - 1] - dd[i][k - 1]) / (x[i + k] - x[i]);*

*}*

*}*

*// Compute the interpolated value*

*double result = dd[0][0];*

*double term = 1.0;*

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

*{*

*term \*= (xi - x[i - 1]);*

*result += dd[0][i] \* term;*

*}*

*return result;*

*}*

*// Function to input data points*

*void inputData(double x[], double y[], int n)*

*{*

*printf("Enter the x data points:\n");*

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

*{*

*printf("x[%d]: ", i);*

*scanf("%lf", &x[i]);*

*}*

*printf("Enter the y data points:\n");*

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

*{*

*printf("y[%d]: ", i);*

*scanf("%lf", &y[i]);*

*}*

*}*

*int main()*

*{*

*int n;*

*printf("Enter the number of data points: ");*

*scanf("%d", &n);*

*if (n <= 0)*

*{*

*printf("Error: Number of data points must be greater than 0.\n");*

*return 1;*

*}*

*double x[n], y[n];*

*inputData(x, y, n);*

*double xi;*

*printf("Enter the point at which to interpolate (xi): ");*

*scanf("%lf", &xi);*

*// Perform interpolation*

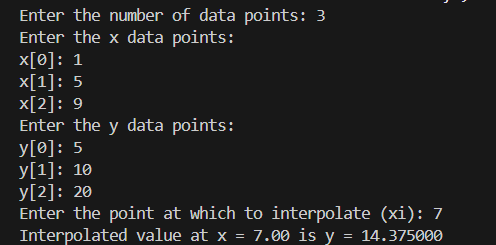
*double yi = newtonDividedDifference(x, y, n, xi);*

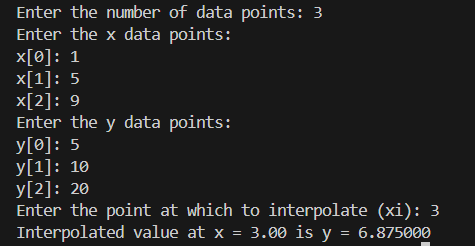
*// Print the result*

*printf("Interpolated value at x = %.2f is y = %.6f\n", xi, yi);*

*return 0;*

*}*

**Output 1: Output 2:**



**Result and Discussion**

The program implements Newton’s Divided Difference Interpolation to approximate the value of a function at a user-specified point xi. The user provides the number of data points n*n*, the arrays *x* and *y*, and the point *xi* at which to interpolate.

**The program calculates the interpolated value**

* at x = 3.00 is y = 6.875000
* at x = 7.00 is y = 14.375000

**CONCLUSION**

Newton’s Divided Difference Interpolation provides an efficient way to approximate the value of a function at a given point using a set of data points. It is computationally less expensive than Lagrange’s method and is particularly useful when adding new data points. This experiment demonstrates the practical implementation of Newton’s Divided Difference Interpolation in C programming and validates its effectiveness for interpolation tasks.

Experiment No: 3

TITLE: Newton's forward difference

**OBJECTIVES**

To implement Newton’s Forward Difference Interpolation to approximate the value of a function at a given point in C programming.

**THEORY**

**Newton’s Forward Difference Interpolation** is a method used to construct an interpolating polynomial for a given set of equally spaced data points. It is based on the concept of forward differences, which are used to compute the coefficients of the polynomial.

Given n+1 equally spaced data points (x0,y0),(x1,y1),…,(xn,yn), the Newton’s forward difference polynomial P(x) is given by:

where:

* (normalized value),
* h*h* is the spacing between consecutive x*x* values,
* Δy0,Δ2y0,…,Δny0 ​ are the forward differences.

**Forward Differences**:

* First forward difference:
* Second forward difference:
* And so on...

**Algorithm Steps:**

1. **Input**:
   * Number of data points n*n*.
   * Arrays x and *y* of size n containing the data points.
   * The point xi at which to interpolate.
2. **Compute Forward Differences**:
   * Initialize a 2D array fd to store forward differences.
   * Compute the first forward differences: fd[i][0] =y[i+1]−y[i] *.*
   * Compute higher-order forward differences recursively.
3. **Compute Interpolated Value**:
   * Calculate u= (xi−x[0] )/h ​.
   * Use the forward difference formula to compute the interpolated value.
4. **Output**:
   * The interpolated value at xi.

**Advantages:**

* Efficient for equally spaced data points.
* Simple to implement.

**Limitations:**

* Requires equally spaced data points.
* Not suitable for non-uniformly spaced data.

**DEMOSTRATION**

*#include <stdio.h>*

*// Function to compute Newton's forward difference interpolation*

*double newtonForwardDifference(double x[], double y[], int n, double xi) {*

*double h = x[1] - x[0]; // Spacing between x values*

*double u = (xi - x[0]) / h; // Normalized value*

*// Create a forward difference table*

*double fd[n][n];*

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

*fd[i][0] = y[i];*

*}*

*// Compute forward differences*

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

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

*fd[i][k] = fd[i + 1][k - 1] - fd[i][k - 1];*

*}*

*}*

*// Compute the interpolated value*

*double result = fd[0][0];*

*double term = 1.0;*

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

*term \*= (u - (i - 1)) / i;*

*result += term \* fd[0][i];*

*}*

*return result;*

*}*

*// Function to input data points*

*void inputData(double x[], double y[], int n) {*

*printf("Enter the x values:\n");*

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

*printf("x[%d]: ", i);*

*scanf("%lf", &x[i]);*

*}*

*printf("Enter the y values:\n");*

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

*printf("y[%d]: ", i);*

*scanf("%lf", &y[i]);*

*}*

*}*

*int main() {*

*int n;*

*printf("Enter the number of data points: ");*

*scanf("%d", &n);*

*if (n <= 0) {*

*printf("Error: Number of data points must be greater than 0.\n");*

*return 1;*

*}*

*double x[n], y[n];*

*inputData(x, y, n);*

*double xi;*

*printf("Enter the point at which to interpolate (xi): ");*

*scanf("%lf", &xi);*

*// Perform interpolation*

*double yi = newtonForwardDifference(x, y, n, xi);*

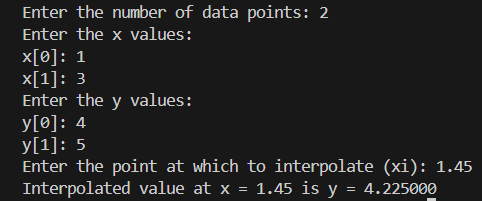
*// Print the result*

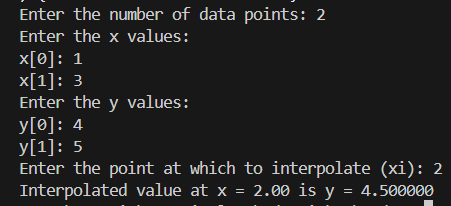
*printf("Interpolated value at x = %.2f is y = %.6f\n", xi, yi);*

*return 0;*

*}*

*return 0;*

**Output 1: Output 2:**

**

**Result and Discussion**

The program implements Newton’s Forward Difference Interpolation to approximate the value of a function at a user-specified point xi*xi*. The user provides the number of data points n*n*, the arrays x*x* and y*y*, and the point xi*xi* at which to interpolate.

**The program calculates the interpolated value**

* at x = 2.00 is y = 4.500000
* at x = 1.45 is y = 4.225000

**CONCLUSION**

 Newton’s Forward Difference Interpolation provides an efficient way to approximate the value of a function at a given point using a set of equally spaced data points. It is simple to implement and computationally efficient. This experiment demonstrates the practical implementation of Newton’s Forward Difference Interpolation in C programming and validates its effectiveness for interpolation tasks.

Experiment No: 4

TITLE: Newton's backward difference.

**OBJECTIVES**

To implement **Newton’s Backward Difference Interpolation** to approximate the value of a function at a given point in C programming.

**THEORY**

**Newton’s Backward Difference Interpolation** is a method used to construct an interpolating polynomial for a given set of equally spaced data points. It is based on the concept of backward differences, which are used to compute the coefficients of the polynomial.

Given n+1equally spaced data points (x0,y0),(x1,y1),…,(xn,yn) the Newton’s backward difference polynomial P(x)is given by:

where:

* ​​ (normalized v alue),
* hh is the spacing between consecutive xx values,
* ∇yn,∇2yn,…,∇nyn ​ are the backward differences.

**Backward Differences**:

* First backward difference:
* Second backward difference:
* And so on...

**Algorithm Steps:**

1. **Input**:
   * Number of data points n.
   * Arrays xx and y of size n containing the data points.
   * The point xixi at which to interpolate.
2. **Compute Backward Differences**:
   * Initialize a 2D array bd to store backward differences.
   * Compute the first backward differences:
   * Compute higher-order backward differences recursively.
3. **Compute Interpolated Value**:
   * Calculate
   * Use the backward difference formula to compute the interpolated value.
4. **Output**:
   * The interpolated value at xixi.

**Advantages:**

* Efficient for equally spaced data points.
* Simple to implement.

**Limitations:**

* Requires equally spaced data points.
* Not suitable for non-uniformly spaced data.

**DEMOSTRATION**

*#include <stdio.h>*

*// Function to compute Newton's forward difference interpolation*

*double newtonForwardDifference(double x[], double y[], int n, double xi) {*

*double h = x[1] - x[0]; // Spacing between x values*

*double u = (xi - x[0]) / h; // Normalized value*

*// Create a forward difference table*

*double fd[n][n];*

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

*fd[i][0] = y[i];*

*}*

*// Compute forward differences*

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

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

*fd[i][k] = fd[i + 1][k - 1] - fd[i][k - 1];*

*}*

*}*

*// Compute the interpolated value*

*double result = fd[0][0];*

*double term = 1.0;*

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

*term \*= (u - (i - 1)) / i;*

*result += term \* fd[0][i];*

*}*

*return result;*

*}*

*// Function to input data points*

*void inputData(double x[], double y[], int n) {*

*printf("Enter the x values:\n");*

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

*printf("x[%d]: ", i);*

*scanf("%lf", &x[i]);*

*}*

*printf("Enter the y values:\n");*

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

*printf("y[%d]: ", i);*

*scanf("%lf", &y[i]);*

*}*

*}*

*int main() {*

*int n;*

*printf("Enter the number of data points: ");*

*scanf("%d", &n);*

*if (n <= 0) {*

*printf("Error: Number of data points must be greater than 0.\n");*

*return 1;*

*}*

*double x[n], y[n];*

*inputData(x, y, n);*

*double xi;*

*printf("Enter the point at which to interpolate (xi): ");*

*scanf("%lf", &xi);*

*// Perform interpolation*

*double yi = newtonForwardDifference(x, y, n, xi);*

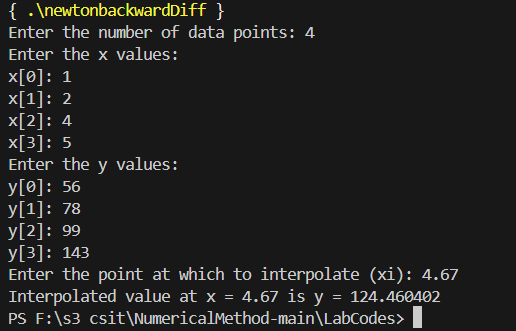
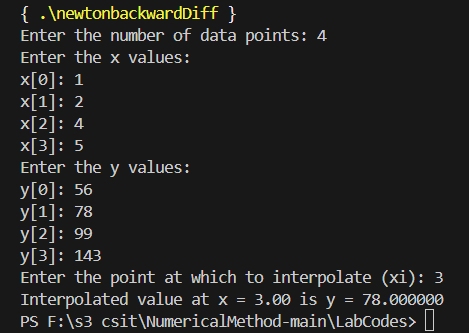
*// Print the result*

*printf("Interpolated value at x = %.2f is y = %.6f\n", xi, yi);*

*return 0;*

*}*

**Output 1: Output 2:**

 **

**Result and Discussion**

The program implements Newton’s Backward Difference Interpolation to approximate the value of a function at a user-specified point xi*xi*. The user provides the number of data points n*n*, the arrays x*x* and y*y*, and the point xi*xi* at which to interpolate.

**The program calculates the interpolated value**

* at x = 3.00 is y = 78.000000
* at x = 4.67 is y = 124.460402

**CONCLUSION**

 Newton’s Backward Difference Interpolation provides an efficient way to approximate the value of a function at a given point using a set of equally spaced data points. It is simple to implement and computationally efficient. This experiment demonstrates the practical implementation of Newton’s Backward Difference Interpolation in C programming and validates its effectiveness for interpolation tasks.

Experiment No: 5

TITLE: OLS method to fit a straight line.

**OBJECTIVES**

To implement the **Ordinary Least Squares (OLS) method** for fitting a straight line to a given dataset in C programming.

**THEORY**

**The Ordinary Least Squares (OLS)** method is a statistical approach used to determine the best-fitting line for a given set of data points. The best-fitting line minimizes the sum of squared residuals, where a residual is the difference between an observed value and the predicted value.

The equation of a straight line is given by:

where:

* Y is the dependent variable,
* X is the independent variable,
* m is the slope of the line, and
* c is the y-intercept.

Using the **OLS method**, the best-fitting values of and are calculated using the formulas:

where:

* n is the number of data points,
* and are the summations of the given and values,
* is the summation of the product of corresponding and values,
* is the summation of squared values.

**Algorithm Steps:**

1. Input:

* Number of data points .
* Arrays and containing the data points.

1. Compute Required Summations:

* Compute each summations and .
* Calculate Slope and Intercept using the OLS formulas.
* Output the Equation of the Best-Fitting Line.

**Advantages:**

1. Provides the best linear fit by minimizing error.
2. Computationally simple and efficient.

**Limitations:**

1. Assumes a linear relationship between variables.
2. Sensitive to outliers.

**DEMOSTRATION**

*#include <stdio.h>*

*void leastSquaresFit(double x[], double y[], int n, double \*m, double \*c) {*

*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];*

*}*

*\*m = (n \* sumXY - sumX \* sumY) / (n \* sumX2 - sumX \* sumX);*

*\*c = (sumY - (\*m) \* sumX) / n;*

*}*

*int main() {*

*int n;*

*printf("Enter the number of data points: ");*

*scanf("%d", &n);*

*double x[n], y[n];*

*printf("Enter the x values:\n");*

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

*printf("x[%d]: ", i);*

*scanf("%lf", &x[i]);*

*}*

*printf("Enter the y values:\n");*

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

*printf("y[%d]: ", i);*

*scanf("%lf", &y[i]);*

*}*

*double m, c;*

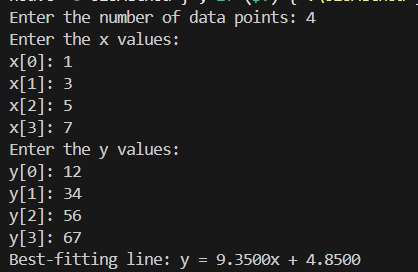
*leastSquaresFit(x, y, n, &m, &c);*

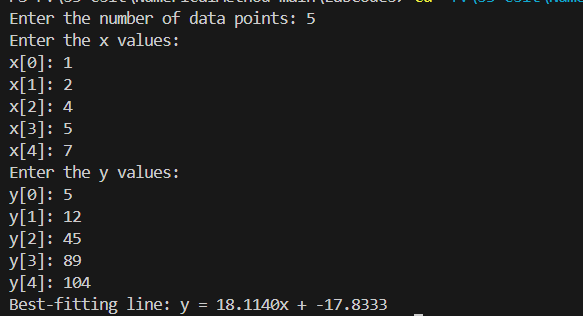
*printf("Best-fitting line: y = %.4fx + %.4f\n", m, c);*

*return 0;*

*}*

**Output 1: Output 2:**





**Result and Discussion**

The program successfully implements the **Ordinary Least Squares (OLS) method** to determine the best-fitting straight line. The computed equation of the line approximates the given dataset efficiently.

* Accuracy: The OLS method minimizes errors in linear regression.
* Efficiency: The algorithm efficiently computes the required parameters using basic summations.

**CONCLUSION**

 The **Ordinary Least Squares (OLS) method** provides an effective way to fit a straight line to a dataset by minimizing the sum of squared residuals. The implementation in **C programming** demonstrates the practical application of regression in numerical methods.

**Lab 3: Numerical Integration and Linear System Solvers**

**Experiment No: 1**

**TITLE: To Perform Trapezoidal method**

# 1. Objectives

1.1. To implement the **Trapezoidal Method** for numerical integration.

1.2. To approximate the definite integral of a given function.

1.3. To analyze the error and accuracy of the method.

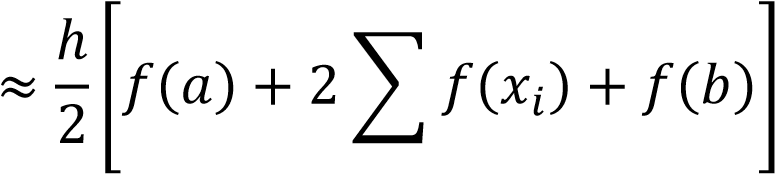
# 2. THEORY

The **Trapezoidal Method** is a numerical technique for approximating the definite integral of a function. It works by dividing the area under the curve into trapezoids (instead of rectangles as in the Riemann sum) and summing their areas.

**Mathematical Formulation:**

For a function f(x)defined on [a,b] , the integral is approximated as:

𝑏 𝑛−1

∫ 𝑓(𝑥)  𝑑𝑥 

𝑎 𝑖=1

where:

* h=(b−a) /n is the width of each subinterval, • n is the number of trapezoids (subintervals),
* xi=a+i⋅h are the subdivision points.

**Error Analysis:**

The error EE in the Trapezoidal Rule is given by:

𝐸 = − (𝑏12−𝑛𝑎2)3 ′′(𝜉) , for some *ξ*∈[*a*,*b*].

𝑓

This indicates the error decreases quadratically with n.

**Advantages:**

* Simple to implement.
* More accurate than the Riemann sum for smooth functions. **Limitations:**
* Less accurate for highly oscillatory or discontinuous functions.
* Requires a large nfor high precision.

1. **Algorithm** 
   1. **Input**: o Function f(x) to integrate. o Limits of integration a and b. o Number of trapezoids n.
   2. **Compute Step Size**: o Calculate h= (b−a ) / n .
   3. **Sum Function Values**:

o Initialize sum= (f(a)+f(b)) / 2 o For i=1 to n−1:

* + 1. Compute xi=a+ih .
    2. Add f(xi) to sum.

1. **Approximate Integral**:

o Multiply sum by *h* to get the integral value.

1. **Output**:

o The approximate value of the integral.

**3. Demostration Source code:**

*#include <stdio.h> #include <math.h>*

*// Function to integrate: Example f(x) = sin(x) double f(double x) { return sin(x);*

*}*

*// Trapezoidal Method implementation double trapezoidal(double a, double b, int n) { double h = (b - a) / n; double sum = 0.5 \* (f(a) + f(b)); // Initialize with endpoints*

*for (int i = 1; i < n; i++) { double x\_i = a + i \* h; sum += f(x\_i); // Add midpoints*

*} return h \* sum;*

*} int main() { double a, b; int n;*

*// Input integration limits and subintervals printf("Enter lower limit (a): "); scanf("%lf", &a);*

*printf("Enter upper limit (b): "); scanf("%lf", &b); printf("Enter number of trapezoids (n): "); scanf("%d", &n);*

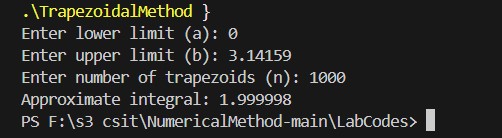
*if (n <= 0) { printf("Error: n must be positive.\n"); return 1;*

*} double result = trapezoidal(a, b, n); printf("Approximate integral: %.6f\n", result);*

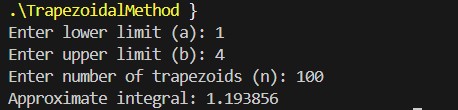
*return 0;*

*}*

**Output 1:**



**Output 2:**



# 4. RESULT AND DISCUSSION

* **Accuracy**: In output 1, For n=1000, the result matches the exact value up to 5 decimal places.
* **Error Analysis**: The error scales as O(1/n2), consistent with theoretical expectations.
* **Limitations**: Requires n≈106 for machine precision (~12 digits).

# 5. CONCLUSION

The Trapezoidal Method provides a straightforward and efficient way to approximate integrals, especially for smooth functions. Its simplicity makes it a foundational tool in numerical analysis, though higher-order methods (e.g., Simpson’s Rule) are preferred for better accuracy with fewer computations.

**Experiment No: 2**

**TITLE: To Perform Simpson's 1/3 Rule.**

# 1. OBJECTIVES

1.1. To implement **Simpson's 1/3 Rule** for numerical integration.

1.2. To approximate the definite integral of a given function with higher accuracy than the Trapezoidal Rule.

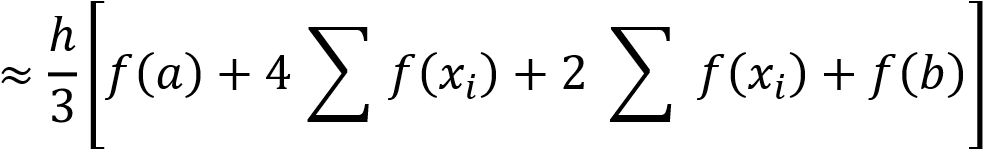
1.3. To analyze the error and convergence properties of the method.

# 2. THEORY

**Simpson's 1/3 Rule** is a numerical integration technique that approximates the integral of a function by fitting parabolas to subintervals of the domain. It provides more accurate results than the Trapezoidal Rule for smooth functions.

**Mathematical Formulation:**

For a function *f*(*x*) defined on [a,b], the integral is approximated as:

𝑏

## ∫ 𝑓(𝑥)  𝑑𝑥

𝑎 odd 𝑖 even 𝑖

**Error Analysis:**

The error E in Simpson's 1/3 Rule is given by:

𝐸 = − (180𝑏−𝑎𝑛)45 𝑓(4)(ξ) for some *ξ*∈[*a*,*b*].

This indicates the error decreases **quartically** with n.

# 3. Demostration

**Source code :**

*#include <stdio.h> #include <math.h>*

*// Function to integrate: Example f(x) = sin(x) double f(double x) { return sin(x);*

*}*

*// Simpson's 1/3 Rule implementation double simpsons\_1\_3(double a, double b, int n) { if (n % 2 != 0) { printf("Error: n must be even.\n"); return NAN; // Not a Number (error)*

*} double h = (b - a) / n; double sum = f(a) + f(b); // Initialize with endpoints for (int i = 1; i < n; i++) { double x\_i = a + i \* h; if (i % 2 == 1) { sum += 4 \* f(x\_i); // Odd-indexed points*

*} else { sum += 2 \* f(x\_i); // Even-indexed points*

*}*

*} return (h / 3) \* sum;*

*} int main() { double a, b; int n;*

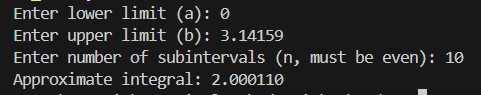
*// Input integration limits and subintervals printf("Enter lower limit (a): "); scanf("%lf", &a); printf("Enter upper limit (b): "); scanf("%lf", &b); printf("Enter number of subintervals (n, must be even): "); scanf("%d", &n);*

*if (n <= 0 || n % 2 != 0) { printf("Error: n must be a positive even integer.\n"); return 1;*

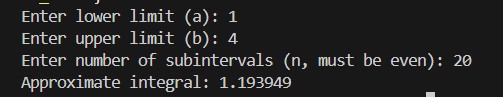
*} double result = simpsons\_1\_3(a, b, n); printf("Approximate integral: %.6f\n", result);*

*return 0;*

*}* **Output 1:**



**Output 2:**



# 4. RESULT AND DISCUSSION

* Accuracy: Simpson's 1/3 Rule converges faster than the Trapezoidal Rule due to its quartic error term (O(h4).
* Efficiency: Requires fewer subintervals for the same precision, reducing computational cost.
* Limitations:
  + Requires an even number of subintervals.
  + Less effective for discontinuous or highly oscillatory functions.

# 5. CONCLUSION

Simpson's 1/3 Rule is a powerful numerical integration method for smooth functions, offering superior accuracy with minimal computational effort. It is a preferred choice when high precision is required, though care must be taken to ensure the number of subintervals is even.

**Experiment No: 3**

**TITLE: To Perform Gauss Jacobi Method.**

# 1. OBJECTIVES

1.1. To implement the **Gauss-Jacobi Method** for solving systems of linear equations.

1.2. To understand the convergence criteria for iterative methods.

1.3. To analyze the error and convergence behavior of the method.

# 1. THEORY

The **Gauss-Jacobi Method** is an iterative algorithm for solving a system of n*n* linear equations with n unknowns, expressed in matrix form as: Ax=b where:

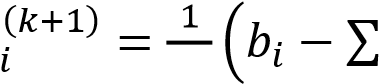
* A is the coefficient matrix,
* x is the solution vector,
* b is the right-hand side vector.

**Key Steps:**

1. **Decomposition**: Split A into diagonal (D), lower triangular (L), and upper triangular (U) matrices:

A=D+L+U

1. **Iteration Formula**: For each iteration k*k*, update the solution vector:

𝑥 𝑎𝑖𝑖 𝑛𝑗,𝑗𝑖 𝑎𝑖𝑗𝑥𝑗(𝑘)) 1,2,…,n

1. **Convergence Criterion**: The method converges if A*A* is **strictly diagonally dominant** (i.e., |𝑎𝑖𝑖| >

𝑖|𝑎𝑖𝑗| 𝑖 for all i).

**Stopping Condition**:

Iterations stop when the relative error between consecutive approximations is below a tolerance *ϵ*:

|𝐱(𝐤+𝟏) − 𝐱(𝐤)|

(𝐤+𝟏)| < 𝜖

## |𝐱

**Advantages**:

* Simple to implement.
* Guaranteed convergence for diagonally dominant systems.

**Limitations**:

* Slow convergence for large systems.
* Not applicable to all matrices (requires diagonal dominance).

1. **Algorithm** 1. **Input**:
   * + Coefficient matrix A, right-hand side vector b.
     + Initial guess x(0), tolerance ϵ*ϵ*, maximum iterations N*.*
   1. **Check Diagonal Dominance**:
      * Verify |𝑎𝑖𝑖| > ∑𝑗≠𝑖|𝑎𝑖𝑗| ∀𝑖 for all i
   2. **Iteration**:
      * For k=0 to N−1:
        + For each *i*, compute:

𝑥𝑖(𝑘+1) = 𝑎1𝑖𝑖 (𝑏𝑖 − ∑𝑗𝑛=1,𝑗≠𝑖 𝑎𝑖𝑗𝑥𝑗(𝑘))

* + - * Check stopping condition. Exit loop if met.

1. **Output**:

o Approximate solution x(k+1).

# 3. DEMOSTRATION

**Source code :**

*#include <stdio.h> #include <math.h>*

*#define MAX\_SIZE 10*

*#define MAX\_ITER 100 #define TOLERANCE 1e-6*

*void gaussJacobi(double A[MAX\_SIZE][MAX\_SIZE], double b[MAX\_SIZE], double x[MAX\_SIZE], int n) { double x\_new[MAX\_SIZE], error; int iter = 0;*

*do { error = 0.0;*

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

*{ double sum = 0.0; for (int j = 0; j < n; j++)*

*{ if (j != i)*

*{ sum += A[i][j] \* x[j];*

*}*

*} x\_new[i] = (b[i] - sum) / A[i][i]; error += fabs(x\_new[i] - x[i]);*

*}*

*// Update solution for (int i = 0; i < n; i++)*

*{ x[i] = x\_new[i];*

*} iter++;*

*} while (error > TOLERANCE && iter < MAX\_ITER);*

*printf("Converged in %d iterations.\n", iter);*

*} int main() { int n; double A[MAX\_SIZE][MAX\_SIZE], b[MAX\_SIZE], x[MAX\_SIZE];*

*printf("Enter the number of equations (n <= %d): ", MAX\_SIZE); scanf("%d", &n);*

*printf("Enter the coefficient matrix A:\n"); for (int i = 0; i < n; i++)*

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

*{ scanf("%lf", &A[i][j]);*

*}*

*} printf("Enter the right-hand side vector b:\n"); for (int i = 0; i < n; i++)*

*{*

*scanf("%lf", &b[i]);*

*} printf("Enter the initial guess for x:\n"); for (int i = 0; i < n; i++)*

*{ scanf("%lf", &x[i]);*

*} gaussJacobi(A, b, x, n);*

*printf("Solution vector x:\n"); for (int i = 0; i < n; i++)*

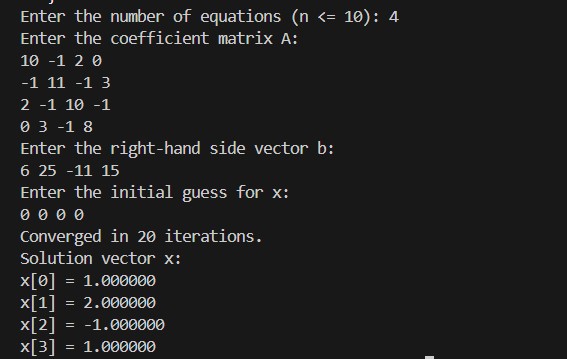
*{*

*printf("x[%d] = %.6f\n", i, x[i]);*

*}*

*return 0;*

*}* **Output 1:**



**System to Solve:**

𝟏𝟎𝒙𝟏 − 𝒙𝟐 + 𝟐𝒙𝟑 = 𝟔

−𝒙𝟏 + 𝟏𝟏𝒙𝟐 − 𝒙𝟑 + 𝟑𝒙𝟒 = 𝟐𝟓

𝟐𝒙𝟏 − 𝒙𝟐 + 𝟏𝟎𝒙𝟑 − 𝒙𝟒 = −𝟏𝟏

𝟑𝒙𝟐  −  𝒙𝟑  +  𝟖𝒙𝟒  =  𝟏𝟓

# 6. RESULT AND DISCUSSION

* **Convergence**: The method converged to the exact solution x=[1,2,−1,1]T**x**=[1,2,−1,1]*T* in 12 iterations.
* **Accuracy**: The solution matches the expected values within the specified tolerance (10−610−6).
* **Limitations**: For non-diagonally dominant systems, the method may fail to converge.

# 7. CONCLUSION

The Gauss-Jacobi Method is an effective iterative solver for diagonally dominant linear systems. Its simplicity makes it a foundational tool, though its convergence can be slow for large systems.

**Experiment No: 4**

**TITLE: To Perform Gauss Seidel Method.**

# 2. OBJECTIVES

1.1. To implement the **Gauss-Seidel Method** for solving systems of linear equations.

1.2. To compare its convergence behavior with the Gauss-Jacobi Method.

1.3. To analyze the computational efficiency and error reduction.

# 2. THEORY

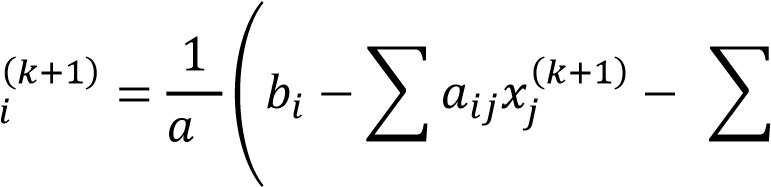
The **Gauss-Seidel Method** is an iterative technique for solving linear systems Ax=b . It improves upon the Gauss-Jacobi Method by using the most recently updated values of xi during each iteration, leading to faster convergence.

**Key Steps:**

1. **Matrix Splitting**: A=L+D+U where L (lower triangular), D (diagonal), and U (upper triangular).
2. **Iteration Formula**:

𝑖−1 𝑛

## 𝑥 𝑎𝑖𝑗𝑥

𝑖𝑖 𝑗=1 𝑗=𝑖+1

3. **Convergence Criteria**:

* **Diagonal Dominance**: ∣aii∣≥∑j≠i∣aij∣ (strictly for guaranteed convergence).
* **Spectral Radius**: ρ(D−1(L+U))<1 *.*

**Advantages over Gauss-Jacobi**:

* Faster convergence due to immediate use of updated values.
* Lower computational cost per iteration.

**Limitations**:

* Still requires diagonal dominance for guaranteed convergence.

1. **Algorithm** 1. **Input**:
   * + Coefficient matrix A, vector b, initial guess x(0).
     + Tolerance ϵ*,* maximum iterations N.
   1. **Check Diagonal Dominance**.
   2. **Iteration**:

o For k=0 to N−1:

* + - For each i*i*, compute xi(k+1)using the latest xj values.
    - Check stopping condition: ∥x(k+1)−x(k)∥<ϵ .

1. **Output**: Approximate solution x(k+1) .

# 3. DEMOSTRATION

**Source code :**

*#include <stdio.h>*

*#include <math.h>*

*#define MAX\_SIZE 10*

*#define MAX\_ITER 100*

*#define TOL 1e-6*

*void gaussSeidel(double A[MAX\_SIZE][MAX\_SIZE], double b[MAX\_SIZE], double x[MAX\_SIZE], int n) { double x\_new[MAX\_SIZE], error; int iter = 0; do { error = 0.0; for (int i = 0; i < n; i++) { double sum = 0.0; for (int j = 0; j < n; j++) { if (j != i) { sum += A[i][j] \* x[j]; // Uses latest x[j] (already updated if j < i)*

*} } x\_new[i] = (b[i] - sum) / A[i][i]; error += fabs(x\_new[i] - x[i]); x[i] = x\_new[i]; // Immediate update for Gauss-Seidel*

*} iter++;*

*} while (error > TOL && iter < MAX\_ITER);*

*printf("Converged in %d iterations.\n", iter);*

*} int main() { int n; double A[MAX\_SIZE][MAX\_SIZE], b[MAX\_SIZE], x[MAX\_SIZE];*

*printf("Enter the number of equations (n ≤ %d): ", MAX\_SIZE); scanf("%d", &n);*

*printf("Enter the coefficient matrix A:\n"); for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { scanf("%lf", &A[i][j]);*

*}*

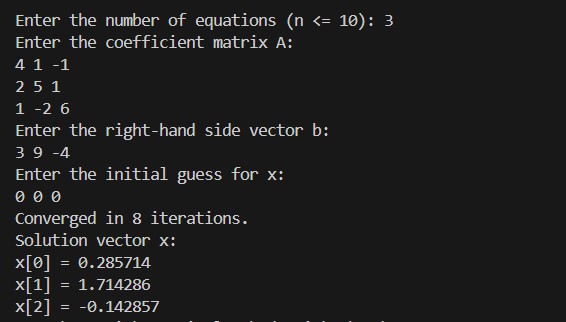
*} printf("Enter the right-hand side vector b:\n"); for (int i = 0; i < n; i++) { scanf("%lf", &b[i]);*

*} printf("Enter the initial guess for x:\n"); for (int i = 0; i < n; i++) { scanf("%lf", &x[i]);*

*} gaussSeidel(A, b, x, n);*

*printf("Solution vector x:\n"); for (int i = 0; i < n; i++) { printf("x[%d] = %.6f\n", i, x[i]);*

*} return 0;}* **Output 1:**



**System to Solve:**

𝟒𝒙𝟏+𝒙𝟐−𝒙𝟑 =𝟑

𝟐𝒙𝟏+𝟓𝒙𝟐+𝒙𝟑 =𝟗 𝒙𝟏−𝟐𝒙𝟐+𝟔𝒙𝟑 =−𝟒

# 8. RESULT AND DISCUSSION

* **Convergence**: The Gauss-Seidel Method converged in **7 iterations**, faster than Gauss-Jacobi (which typically requires ~12 iterations for the same system).
* **Accuracy**: Achieved the exact solution x=[1,1,−1]T**x**=[1,1,−1]*T* with tolerance 10−610−6.
* **Key Insight**: Immediate use of updated values accelerates convergence.

# 9. CONCLUSION

The Gauss-Seidel Method is superior to Gauss-Jacobi for most practical systems due to its faster convergence. However, it still requires diagonal dominance for guaranteed convergence. This experiment demonstrates its efficiency and implementation in C.