**Lab 3**

**Experiment No: 1**

**TITLE: To Perform Trapezoidal method**

**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.

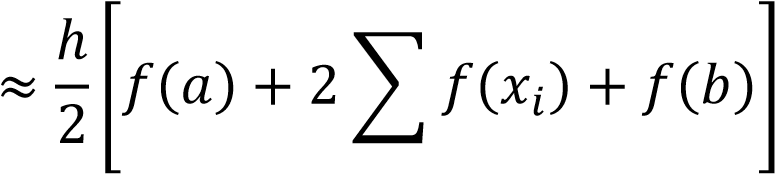
**THEORY**

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 n for high precision.

**Algorithm**

1. **Input:**

Function f(x) to integrate.

Limits of integration a and b.

Number of trapezoids n.

1. **Compute Step Size:**

Calculate h= (b−a ) / n .

1. **Sum Function Values:**

Initialize sum= (f(a)+f(b)) / 2

1. **For i=1 to n−1:**

Compute xi=a+ih .

Add f(xi) to sum.

1. **Approximate Integral:**

Multiply sum by h to get the integral value.

1. **Output:**

The approximate value of the integral.

**DEMOSTRATION**

*#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;*

*}*

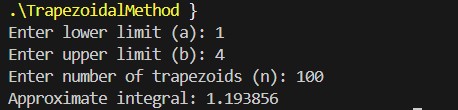
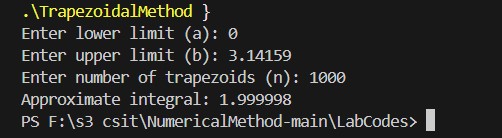
*double result = trapezoidal(a, b, n);*

*printf("Approximate integral: %.6f\n", result);*

*return 0;*

*}*

**Output 1: Output 2:**



**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.

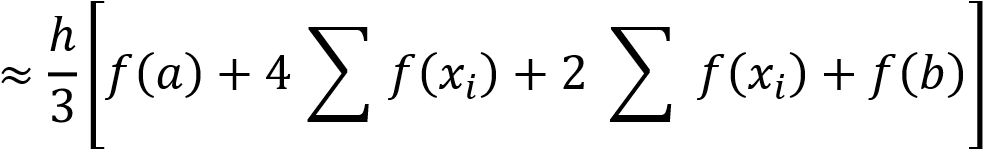
1.3. To analyze the error and convergence properties of the method.

# 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;

}

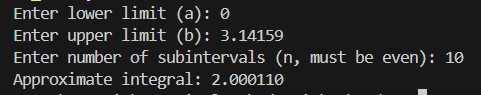
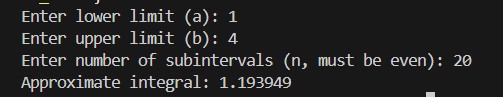
double result = simpsons\_1\_3(a, b, n);

printf("Approximate integral: %.6f\n", result);

return 0;

}

**Output: Output 2:**



# 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.

# 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.**

# 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.

# 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:

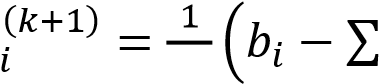
* A is the coefficient matrix,
* x is the solution vector,
* b is the right-hand side vector.

**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).

**DEMONSTRATION**

#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 vector

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++) {

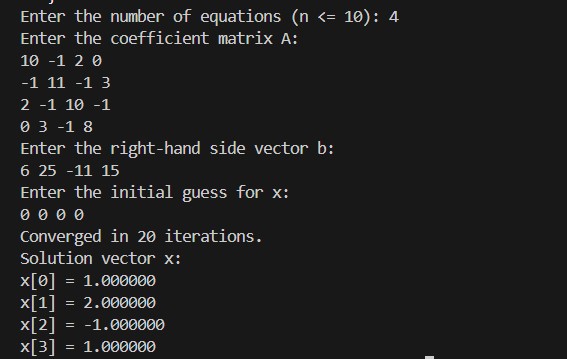
printf("x[%d] = %.6f\n", i, x[i]);

}

return 0;

}

**Output:**



# 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.**

# 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.

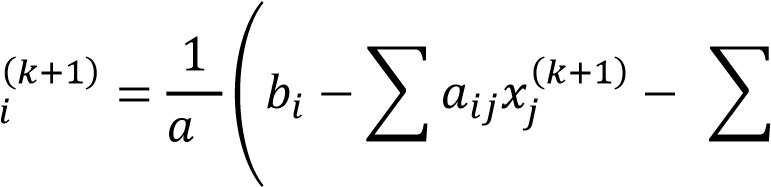
# THEORY

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

**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.

**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]; // Use latest values (Gauss-Seidel characteristic)*

*}*

*}*

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

*error += fabs(x\_new[i] - x[i]);*

*x[i] = x\_new[i]; // Immediate update in 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");*

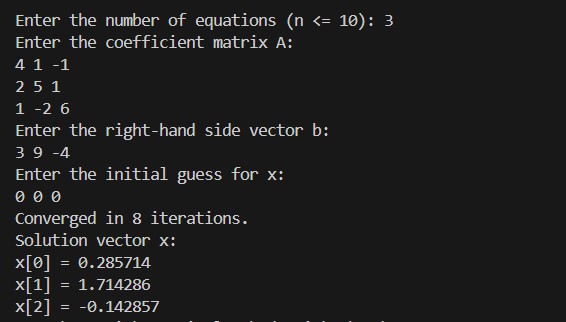
*for (int i = 0; i < n; i++) {*

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

*}*

*return 0;*

*}*



**Output 1:**

**System to Solve:**

𝟒𝒙𝟏+𝒙𝟐−𝒙𝟑 =𝟑

𝟐𝒙𝟏+𝟓𝒙𝟐+𝒙=𝟗 𝒙𝟏−𝟐𝒙𝟐+𝟔𝒙𝟑 =−𝟒

# 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.

# 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.