1. Write a program to implement bisection method for solving non-linear equations. Source Source Code

#include <stdio.h>

#include <math.h>

#include <stdbool.h>

int a0,a1,a2,a3;

float f(float x) {

return (a3 \* x \* x \* x + a2 \* x \* x + a1 \* x + a0);

}

int main()

{

float x1.xm, xu, fxl, fxu, fxm.e.era;

printf("Enter coefficient of eqn,");

scanf("%d,%d,%d,%d", &a3, &a2, &a1, &a0);

printf("Enter initial conditons: ");

scanf("%f%f", &xl, &xu);

printf("Enter error margin: ");

scanf("%f", &e); while (true)

{

fx1 = f(x1); fxu = f(xu);

xm = (x1 + xu) / 2;

fxm = f(xm);

iif((fx1 \* fxm) = 0)

{

printf("Root=%f\n", xm);

break;

}

else if ((fx1 \* fxm) < 0)

xu = xm;

elsex1 = xm;

era = (xu - x1) / xu;

if (era < 0)

era = -era;

if (era < e)

{

printf("Root=%f\n", xm);

break;

}

}

}

Write a program to implement Newton Raphson method for solving non-linear equation.

Source Code:

#include <stdio.h>  
#include <math.h>  
float f(float x)  
{  
return (x \* log(x) - 1.2);  
}  
float f1(float x)  
{  
return(1+log(x));  
}  
int main()  
{  
float h.x0,x1,err;  
int itr,maxitr=1000;  
printf("enter the initial gues:");  
scanf("%f",&err);  
for(itr=1;itr<maxitr;itr++)  
{  
h=f(x0)/f1(x0);  
x1=x0-h;  
printf("at iteration %d, x=%f",itr,x1);  
printf("\n");  
if(fabs(x1-x0)/x1<err)  
{  
print f("\nroot is%f\n",x1);  
printf("no of iteration is %d",itr);  
break;  
}x0=x1;  
}  
return 0;  
}

Write a program to implement Secant method for sloving non-linear equations.  
  
#include <stdio.h>  
#include <math.h>  
float f(float x)  
{  
return (3 \* x - cos(X));  
}  
int main()  
{  
float g, h, x0, x1, err;  
int itr, maxitr - 100;  
printf("Enter two initial guesses:");  
scanf("%f%f", &x0, &x1);  
printf("Enter the error margin:");  
scanf("%f", &err);  
for (itr = 1; itr < maxitr; itr++)  
{  
h = (x1 - x0) \* (f(x1) / f(x1) - f(x0));  
g = x1;  
x1 = x1 - h;  
printf("at iteration%d,x=%f", itr, x1);  
printf("\n");  
if (fabs((x1 - g) / x1) < err)  
{  
printf("\nroot is %f\n", x1);  
printf("no of iteration is %d", itr);  
break;  
}  
x0 = g;  
}  
return 0;  
}

WAP to implement fixed point iteration method for solving non linear equations.

Introduction

The Fixed Point Iteration Method is a numerical technique used to solve non-linear equations of the form \( f(x) = 0 \). It involves rearranging the equation into the form \( x = g(x) \) and iteratively substituting the initial guess into \( g(x) \) to get closer to the root. The method converges when \( |g'(x)| < 1 \) near the root.

Source Code

#include <stdio.h>

#include <math.h>

#define EPSILON 0.0001

#define MAX\_ITER 100

double g(double x)

{ return (cos(x));

}

double g\_derivative(double x) {

return (-sin(x));

} int main() {

double x0, x1;

int iteration = 0;

printf("Enter initial guess: ");

scanf("%lf", &x0);

if (fabs(g\_derivative(x0)) >= 1)

{

printf("The method may not converge with this initial guess.\n");

return -1; }

printf("\nIteration\t x0\t\t x1\t\t Error\n");

do { x1 = g(x0);

double error = fabs(x1 - x0);

printf("%d\t\t%.6lf\t%.6lf\t%.6lf\n", iteration, x0, x1, error);

x0 = x1;

iteration++;

if (iteration > MAX\_ITER)

{

printf("Exceeded maximum iterations. The method did not converge.\n"); return -1;

}

} while (fabs(g(x1) - x1) > EPSILON);

printf("\nThe root of the equation is approximately: %.6lf\n", x1); printf("Total iterations: %d\n", iteration);

return 0;

}

Conclusion

The Fixed Point Iteration Method successfully approximates the root of a non-linear equation. The method's convergence depends on the choice of the initial guess and the function \( g(x) \). If \( |g'(x)| < 1 \), the method converges to a root. However, divergence may occur for improper choices of \( g(x) \) or \( x\_0 \). This example demonstrates the effectiveness of the method when applied correctly.

WAP in C to find the solution using Gauss Elimination Method for system of linear equation.

Introduction:

The Gauss Elimination Method is a systematic approach to solving a system of linear equations. It involves transforming the system into an upper triangular matrix (row echelon form) and then performing back substitution to find the solutions.

#include <stdio.h>

#include <math.h>

#define MAX 10

void gaussElimination(int n, double a[MAX][MAX + 1])

{ int i, j, k; double factor, x[MAX], sum;

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

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

{ factor = a[k][i] / a[i][i];

for (j = i; j <= n; j++) { a[k][j] -= factor \* a[i][j];

}

}

}

x[n - 1] = a[n - 1][n] / a[n - 1][n - 1];

for (i = n - 2; i >= 0; i--) { sum = 0.0;

for (j = i + 1; j < n; j++) { sum += a[i][j] \* x[j];

}

x[i] = (a[i][n] - sum) / a[i][i]; }

printf("\nThe solution is:\n");

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

{ printf("x%d = %.6lf\n", i + 1, x[i]); }

}

int main()

{ double a[MAX][MAX + 1];

int n, i, j;

printf("Enter the number of equations: ");

scanf("%d", &n);

printf("Enter the augmented matrix (coefficients and constants):\n");

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

{

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

{

scanf("%lf", &a[i][j]);

}

}

gaussElimination(n, a);

return 0;

}

**Conclusion\***

The Gauss Elimination Method is an effective and systematic way to solve a system of linear equations. It is widely used in numerical computations. The accuracy of the solution depends on the precision of calculations, and partial pivoting can be used to avoid division by small numbers for better stability.

WAP a program to find solution using Gauss Jordan Method for system of linear equations.

#include <stdio.h>

#define MAX 10\

void gaussJordan(int n, double a[MAX][MAX + 1])

{

int i, j, k;

double factor;

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

{

double pivot = a[i][i];

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

{ a[i][j] /= pivot;

}

for (k = 0; k < n; k++)

{

if (k != i)

{

factor = a[k][i];

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

{ a[k][j] -= factor \* a[i][j];

}

}

}

}

printf("\nThe solution is:\n");

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

{ printf("x%d = %.6lf\n", i + 1, a[i][n]);

}

} int main()

{ double a[MAX][MAX + 1];

int n, i, j;

printf("Enter the number of equations: ");

scanf("%d", &n);

printf("Enter the augmented matrix (coefficients and constants):\n");

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

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

{ scanf("%lf", &a[i][j]); } }

gaussJordan(n, a);

return 0;

}

**Conclusion**

The Gauss-Jordan Method provides a systematic approach to solving a system of linear equations by directly reducing the matrix to its simplest form. This eliminates the need for back substitution, making the solution readily available from the final matrix.

WAP to calculate inverse matrix.

Introduction

The inverse of a square matrix \( A \) (denoted as \( A^{-1} \)) exists if and only if the determinant of \( A \) is non-zero. The Gauss-Jordan method can be used to find the inverse of a matrix by augmenting the matrix \( A \) with the identity matrix and performing row operations to convert \( A \) into the identity matrix, leaving the augmented part as the inverse.

Source Code:

#include <stdio.h>

#include <math.h>

#define MAX 10

void printMatrix(int n, double a[MAX][2 \* MAX])

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

{ for (int j = 0; j < 2 \* n; j++)

{ printf("%.6lf ", a[i][j]);

}

printf("\n");

}

}

void findInverse(int n, double a[MAX][2 \* MAX])

{

int i, j, k; double factor;

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

{

double pivot = a[i][i];

if (fabs(pivot) < 1e-9)

{ printf("Matrix is singular, inverse does not exist.\n");

return;

}

for (j = 0; j < 2 \* n; j++)

{ a[i][j] /= pivot;

}

for (k = 0; k < n; k++)

{ if (k != i)

{ factor = a[k][i];

for (j = 0; j < 2 \* n; j++)

{ a[k][j] -= factor \* a[i][j];

}

}

}

}

printf("\nThe inverse matrix is:\n");

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

{

for (j = n; j < 2 \* n; j++)

{

printf("%.6lf ", a[i][j]);

}

printf("\n"); }

}

int main()

{

double a[MAX][2 \* MAX];

int n, i, j;

printf("Enter the size of the matrix: ");

scanf("%d", &n);

printf("Enter the elements of the matrix:\n");

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

{

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

{

scanf("%lf", &a[i][j]);

}

}

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

{

for (j = n; j < 2 \* n; j++)

{

a[i][j] = (i == (j - n)) ? 1.0 : 0.0; }

}

printf("\nAugmented matrix:\n");

printMatrix(n, a);

findInverse(n, a);

return 0;

}

**Conclusion**

This program demonstrates how to compute the inverse of a matrix using the Gauss-Jordan method. The inverse exists only if the matrix is non-singular (determinant \( \neq 0 \)). For singular matrices, the program will notify that the inverse does not exist.

WAP in C to implement Jacobi Iterative Method.

Introduction:

The Jacobi Iterative Method is an iterative numerical technique for solving a system of linear equations. It assumes an initial guess and refines it iteratively until a desired level of accuracy is achieved.

Source Code:

#include <stdio.h>

#include <math.h>

#define MAX 10

#define TOLERANCE 1e-6

#define MAX\_ITER 1000

void jacobiMethod(int n, double a[MAX][MAX], double b[MAX], double x[MAX])

{ double new\_x[MAX], diff, max\_diff; int i, j, iter;

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

{

x[i] = 0.0;

}

printf("\nIterative results:\n");

for (iter = 1; iter <= MAX\_ITER; iter++)

{ max\_diff = 0.0;

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

{ new\_x[i] = b[i];

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

{ if (i != j)

{ new\_x[i] -= a[i][j] \* x[j];

}

}

new\_x[i] /= a[i][i];

diff = fabs(new\_x[i] - x[i]);

if (diff > max\_diff) { max\_diff = diff; }

for (i = 0; i < n; i++) { x[i] = new\_x[i];

printf("Iteration %d: ", iter);

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

{ printf("x%d = %.6lf ", i + 1, x[i]);

}

printf("\n

if (max\_diff < TOLERANCE)

{

printf("\nThe system converged in %d iterations.\n", iter);

return;

}

}

printf("\nThe system did not converge within the maximum number of iterations.\n");

}

int main()

{

double a[MAX][MAX], b[MAX], x[MAX]; int n, i, j;

printf("Enter the number of equations: ");

scanf("%d", &n);

printf("Enter the coefficients of the matrix:\n");

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

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

}

}

printf("Enter the constant terms: ");

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

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

}

jacobiMethod(n, a, b, x);

return 0;

}

**Conclusion**

The Jacobi Iterative Method provides an efficient way to solve large systems of linear equations. However, it converges only if the coefficient matrix is diagonally dominant or symmetric positive definite. For large systems, this method is easy to implement and computationally inexpensive.

WAP in C to implement Newton’s forward interpolation

Introduction

Newton's Forward Interpolation is a method used to find an approximate value of a function \( f(x) \) for a given value of \( x \) that lies within a tabulated set of points. It uses the forward differences of \( y \)-values in an equally spaced data set.

**Source Code\***

#include <stdio.h> /

double factorial(int n)

{

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

return 1;

return n \* factorial(n - 1);

}

double newtonForwardInterpolation(double x[], double y[], int n, double value)

{

double forwardDiff[n][n];

double u, result;

int i, j;

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

{

forwardDiff[i][0] = y[i];

}

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

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

{

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

}

}

printf("\nForward Difference Table:\n");

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

{

printf("%lf", x[i]);

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

{

printf("\t%lf", forwardDiff[i][j]);

}

printf("\n");

}

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

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

result = forwardDiff[0][0];

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

{

double term = forwardDiff[0][i];

for (j = 0; j < i; j++)

{

term \*= (u - j);

}

term /= factorial(i);

result += term;

}

return result;

}

int main()

{

int n, i;

double x[10], y[10], value, result;

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

scanf("%d", &n);

printf("Enter the values of x and y:\n");

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

{ printf("x[%d] = ", i);

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

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

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

printf("Enter the value of x for interpolation: ");

scanf("%lf", &value);

result = newtonForwardInterpolation(x, y, n, value);

printf("\nThe interpolated value at x = %.2lf is %.6lf\n", value, result);

return 0;

}

WAP to implement Gauss-Seidal Method

**Introduction**

The Gauss-Seidel Method is an iterative numerical technique for solving a system of linear equations. Unlike the Jacobi method, Gauss-Seidel updates each variable as soon as it is computed, using the most recent values. This often leads to faster convergence compared to the Jacobi method.

Source Code:

#include <stdio.h>

#include <math.h>

#define MAX 10

#define TOLERANCE 1e-6

#define MAX\_ITER 1000

void gaussSeidel(int n, double a[MAX][MAX], double b[MAX], double x[MAX])

{ int i, j, iter;

double sum, diff, max\_diff;

for (i = 0; i < n; i++) { x[i] = 0.0;

}

printf("\nIterative results:\n");

for (iter = 1; iter <= MAX\_ITER; iter++)

{ max\_diff = 0.0;

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

{ sum = b[i];

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

{

if (j != i)

{

sum -= a[i][j] \* x[j];

}

}

for x[i] double new\_x = sum / a[i][i];

diff = fabs(new\_x - x[i]);

if (diff > max\_diff)

{

max\_diff = diff;

}

x[i] = new\_x;

}

printf("Iteration %d: ", iter);

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

{

printf("x%d = %.6lf ", i + 1, x[i]);

}

printf("\n");

if (max\_diff < TOLERANCE)

{ printf("\nThe system converged in %d iterations.\n", iter); return;

}

}

printf("\nThe system did not converge within the maximum number of iterations.\n"); }

int main()

{

double a[MAX][MAX], b[MAX], x[MAX];

int n, i, j;

printf("Enter the number of equations: ");

scanf("%d", &n);

printf("Enter the coefficients of the matrix:\n");

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

{

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

{

scanf("%lf", &a[i][j]);

}

}

printf("Enter the constant terms: ");

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

{

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

}

gaussSeidel(n, a, b, x);

return 0;

}

WAP in C to implement Lagrange interpolation Method.

**Introduction**

Lagrange Interpolation is a polynomial interpolation method that provides an approximate value of a function \( f(x) \) for a given value of \( x \). It constructs a polynomial passing through all given points \((x\_0, y\_0), (x\_1, y\_1), \dots, (x\_n, y\_n)\). The Lagrange interpolation formula is: \[ f(x) = \sum\_{i=0}^{n} y\_i \prod\_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x\_j}{x\_i - x\_j} \] where \( n \) is the number of given data points.

**Source Code**

#include <stdio.h>

double lagrangeInterpolation(double x[], double y[], int n, double value)

{

double result = 0.0;

int i, j;

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

{

double term = y[i];

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

{

if (i != j)

{

term \*= (value - x[j]) / (x[i] - x[j]);

}

}

result += term;

}

return result;

}

int main()

{

int n, i; double x[10], y[10], value, result;

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

scanf("%d", &n

printf("Enter the values of x and y:\n");

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

{

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

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

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

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

}

printf("Enter the value of x for interpolation: ");

scanf("%lf", &value);

result = lagrangeInterpolation(x, y, n, value

printf("\nThe interpolated value at x = %.2lf is %.6lf\n", value, result);

return 0;

}

**Conclusion**

The Lagrange Interpolation method is effective for estimating values of a function at any given point within the range of the input data. It is simple to implement and does not require constructing a difference table. However, for large datasets, it can become computationally expensive.

WAP to implement linear integration model.

Introduction

Linear integration approximates the area under a curve \( f(x) \) over a given interval \([a, b]\) by dividing it into small intervals and summing the areas of trapezoids formed by linear approximations of \( f(x) \).

Source Code:

#include <stdio.h>

#include <math.h

if(double x)

{

f(x) = x^2 return x \* x;

}

linearIntegration(double a, double b, int n)

{ double h = (b - a) / n;

sum = f(a) + f(b);

for i = 1 to n-1

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

{

double x = a + i \* h;

sum += 2 \* f(x); }

return (h / 2) \* sum;

}

int main()

{

double a, b, result;

int n

printf("Enter the lower limit (a): ");

scanf("%lf", &a);

printf("Enter the upper limit (b): ");

scanf("%lf", &b);

printf("Enter the number of intervals (n): ");

scanf("%d", &n);

result = linearIntegration(a, b, n);

printf("\nThe approximate value of the integral is: %.6lf\n", result);

return 0;

}

**Conclusion**

The linear integration model using the Trapezoidal Rule is simple and effective for approximating definite integrals. While it is accurate for linear functions, higher-order functions may require smaller step sizes (\( h \)) or advanced methods for better precision.

WAp in C to implement linear regression model.

Introduction:

Linear Regression is a statistical method for modeling the relationship between a dependent variable \( y \) and an independent variable \( x \).

**Source Code**

#include <stdio.h>

void linearRegression(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];

}

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

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

}

int main()

{

int n;

double x[100], y[100], m, c;

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

scanf("%d", &n);

printf("Enter the values of x and y:\n");

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

{

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

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

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

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

linearRegression(x, y, n, &m, &c);

printf("\nThe linear regression line is: y = %.2lfx + %.2lf\n", m, c);

return 0;

}

**Conclusion**

The linear regression model calculates the best-fit line for a given dataset. This implementation assumes a linear relationship between \( x \) and \( y \). It is effective for small datasets and forms the foundation for more advanced regression techniques.

WAP to find the integration of a function using trapezoidal rule.

**Introduction**

The Trapezoidal Rule is a numerical method used to estimate the definite integral of a function. It approximates the area under the curve of a function by dividing the area into trapezoids rather than rectangles, offering a better approximation than the simple rectangular rule.

**Source Code**

#include <stdio.h>

Double f(double x)

{

f(x) = x^2 (can be changed as needed)

return x \* x;

}

trapezoidalRule(double a, double b, int n)

{

double h = (b - a) / n;

sum = f(a) + f(b);

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

{

double x = a + i \* h;

sum += 2 \* f(x);

}

return (h / 2) \* sum;

}

int main()

{

double a, b, result;

int n;

printf("Enter the lower limit (a): ");

scanf("%lf", &a); printf("Enter the upper limit (b): ");

scanf("%lf", &b); printf("Enter the number of subintervals (n): ");

scanf("%d", &n);

result = trapezoidalRule(a, b, n);

printf("\nThe approximate value of the integral is: %.6lf\n", result);

return 0;

}

**Conclusion**

The Trapezoidal Rule is a simple and effective numerical integration method. It works well when the function is reasonably smooth over the interval. As \( n \) (the number of subintervals) increases, the accuracy of the approximation improves. However, for highly oscillatory functions or functions with steep gradients, more sophisticated numerical methods may be required.

Write a program to find the integration of a function using simson 1/3 rule.

**Introduction**

Simpson's 1/3 Rule is a method for numerical integration, offering a higher degree of accuracy than the Trapezoidal Rule. It approximates the integral of a function by fitting parabolas through the points of the function.

**Source Code**

#include <stdio.h>

double f(double x)

{

f(x) = x^2 (can be changed as needed) return x \* x; }

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

{

if (n % 2 != 0)

{ printf("Error: n must be even for Simpson's 1/3 Rule.\n");

return -1;

}

double h = (b - a) / n;

double sum = f(a) + f(b);

odd indices (4 \* f(x\_i)) and even indices (2 \* f(x\_i))

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

{ double x = a + i \* h;

if (i % 2 == 0)

{ sum += 2 \* f(x);

index: 2 \* f(x\_i)

}

else { sum += 4 \* f(x);

index: 4 \* f(x\_i)

}

}

return (h / 3) \* sum; }

int main()

{ double a, b, result;

int n;

printf("Enter the lower limit (a): ");

scanf("%lf", &a);

printf("Enter the upper limit (b): ");

scanf("%lf", &b);

printf("Enter the number of subintervals (n): ");

scanf("%d", &n);

result = simpsonsRule(a, b, n);

if (result != -1)

{

printf("\nThe approximate value of the integral is: %.6lf\n", result);

}

return 0;

}

**Conclusion**

Simpson's 1/3 Rule provides a more accurate estimate of the integral of a function compared to the Trapezoidal Rule, especially when the function is smooth. The accuracy improves as the number of subintervals \( n \) increases, but \( n \) must always be even for the rule to work. For highly oscillatory functions, increasing \( n \) or using other advanced methods may be necessary for better precision.

WAP to find Simpson's 3/8 Rule for Numerical Integration in C.

**Introduction**

Simpson's 3/8 Rule is a method for numerical integration that is an extension of Simpson's 1/3 Rule. This method approximates the integral of a function by fitting cubic polynomials through the points of the function.

**Source Code**

#include <stdio.h>

double f(double x) {

function: f(x) = x^2

return x \* x;

}

double simpsons38Rule(double a, double b, int n)

{

if (n % 3 != 0)

{

printf("Error: n must be a multiple of 3 for Simpson's 3/8 Rule.\n");

return -1;

}

double h = (b - a) / n;

double sum = f(a) + f(b);

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

{

double x = a + i \* h;

if (i % 3 == 0)

{

sum += 2 \* f(x);

index (x\_3, x\_6, x\_9,...): 2 \* f(x\_i)

}

else if (i % 3 == 1)

{

sum += 3 \* f(x);

3 \* f(x\_i)

}

else

{

sum += 3 \* f(x);

3 \* f(x\_i)

}

}

return (3 \* h / 8) \* sum;

}

int main()

{ double a, b, result;

int n;

printf("Enter the lower limit (a): ");

scanf("%lf", &a);

printf("Enter the upper limit (b): ");

scanf("%lf", &b);

printf("Enter the number of subintervals (n) [must be a multiple of 3]: ");

scanf("%d", &n);

result = simpsons38Rule(a, b, n);

if (result != -1)

{ printf("\nThe approximate value of the integral is: %.6lf\n", result);

}

return 0;

}

**Conclusion**

Simpson's 3/8 Rule provides a higher degree of accuracy than the Trapezoidal Rule and Simpson's 1/3 Rule. It is particularly useful when the function is smooth and the number of subintervals is a multiple of 3. The rule achieves better precision with fewer intervals compared to the other methods, but requires a multiple of 3 subintervals. As the number of subintervals increases, the accuracy of the method improves significantly.