1. **WAP to Implement Bisection Method and find root of the equation *f(x)* = *x3-2x-5*.**

#include <stdio.h>

#include <math.h>

double f(double x) { *// Function representing the equation x^3 - 2x - 5*

    return (x \* x \* x - 2 \* x - 5);

}

void bisection(double a, double b, double tolerance) { *// Bisection Method*

    double c;

    if (f(a) \* f(b) >= 0) { *// Ensure that the function changes sign at the endpoints*

        printf("The function must have opposite signs at a and b.\n");

        return;

    }

    while ((b - a) / 2 > tolerance) {

        c = (a + b) / 2;

        if (f(c) == 0) {

            printf("Root found at x = %.6lf\n", c);

            return;

        }

        if (f(a) \* f(c) < 0) b = c;

        else a = c;

    }

    c = (a + b) / 2;

    printf("Root approximation: %.6lf\n", c);

}

int main() {

    double a, b, tolerance;

    printf("Enter the value of a (lower bound of the interval): ");

    scanf("%lf", &a);

    printf("Enter the value of b (upper bound of the interval): ");

    scanf("%lf", &b);

    printf("Enter the tolerance for root approximation: ");

    scanf("%lf", &tolerance);

    bisection(a, b, tolerance); *// Call the bisection method function*

    return 0;

}

**OUTPUT**

S:\WorkSpace\Numerical-Methods> gcc bisection.c

S:\WorkSpace\Numerical-Methods> ./a.exe

Enter the value of a (lower bound of the interval): 2

Enter the value of b (upper bound of the interval): 3

Enter the tolerance for root approximation: 0.00001

Root approximation: 2.094551

1. **WAP for the Regula Falsi Method with the Equation *f(x)=x3−5x+1*:**

#include <stdio.h>

#include <math.h>

double f(double x) { *// Function representing the equation x^3 - 5x + 1*

    return (x \* x \* x - 5 \* x + 1);

}

void regulaFalsi(double a, double b, double tolerance) { *// Regula Falsi Method*

    double c;

    if (f(a) \* f(b) >= 0) { *// Ensure that the function has opposite signs at the endpoints*.

        printf("The function must have opposite signs at a and b.\n");

        return;

    }

    while (fabs(f(a)) > tolerance) {

        c = b - (f(b) \* (b - a)) / (f(b) - f(a)); *// Find the point where the line intersects the x-axis*

        if (fabs(f(c)) < tolerance) { *// If the root is found or the approximation is within tolerance*

            printf("Root found at x = %.6lf\n", c);

            return;

        }

        if (f(a) \* f(c) < 0)        *// Narrow down the interval*

            b = c;

        else

            a = c;

    }

    printf("Root approximation: %.6lf\n", c);

}

int main() {

    double a, b, tolerance;

    printf("Enter the value of a (lower bound of the interval): ");

    scanf("%lf", &a);

    printf("Enter the value of b (upper bound of the interval): ");

    scanf("%lf", &b);

    printf("Enter the tolerance for root approximation: ");

    scanf("%lf", &tolerance);

    regulaFalsi(a, b, tolerance); *// Call the regula falsi method function*

    return 0;

}

**OUTPUT**

S:\WorkSpace\Numerical-Methods> gcc regula-falsi.c

S:\WorkSpace\Numerical-Methods> ./a.exe

Enter the value of a (lower bound of the interval): 0

Enter the value of b (upper bound of the interval): 1

Enter the tolerance for root approximation: 0.0001

Root found at x = 0.201654

1. **WAP for the Newton-Raphson Method to find the root of the Equation *f(x)=x3−3x-5*:**

#include <stdio.h>

#include <math.h>

double f(double x) { // Function representing the equation x^3 - 3x - 5

return (x \* x \* x - 3 \* x - 5);

}

double f\_prime(double x) { // Derivative of the function f(x) = x^3 - 3x - 5

return (3 \* x \* x - 3); // Derivative: 3x^2 - 3

}

// Newton-Raphson Method

void newtonRaphson(double x0, double tolerance) {

double x1;

while (1) {

x1 = x0 - f(x0) / f\_prime(x0);

- if (fabs(x1 - x0) < tolerance) {

printf("Root found at x = %.6lf\n", x1);

return;

}

x0 = x1;

}

}

int main() {

double x0, tolerance;

printf("Enter the initial guess for the root: ");

scanf("%lf", &x0);

printf("Enter the tolerance for root approximation: ");

scanf("%lf", &tolerance);

newtonRaphson(x0, tolerance); // Call the Newton-Raphson method

return 0;

}

**OUTPUT**

S:\WorkSpace\Numerical-Methods> gcc newton-raphson.c

S:\WorkSpace\Numerical-Methods> ./a.exe

Enter the initial guess for the root: 2

Enter the tolerance for root approximation: 0.00001

Root found at x = 2.279019

1. **WAP to Implement Lagrange’s Interpolation. For the given example data set. And find the value of y for x=1, and for x=4.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **X** | **-1** | **0** | **2** | **3** |
| **Y** | **-8** | **3** | **1** | **2** |

#include <stdio.h>

*// Function to calculate the Lagrange basis polynomial l\_i(x)*

double lagrange\_basis(double x, double x\_data[], int i, int n) {

    double result = 1.0;

    for (int j = 0; j < n; j++) {

        if (j != i) {

            result \*= (x - x\_data[j]) / (x\_data[i] - x\_data[j]);

        }

    }

    return result;

}

*// Function to perform Lagrange interpolation*

double lagrange\_interpolation(double x, double x\_data[], double y\_data[], int n) {

    double result = 0.0;

    for (int i = 0; i < n; i++) {

        result += y\_data[i] \* lagrange\_basis(x, x\_data, i, n);

    }

    return result;

}

int main() {

    int n;

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

    scanf("%d", &n);

    double x\_data[n], y\_data[n];

    printf("Enter the x and y values for the data points:\n");

    for (int i = 0; i < n; i++) {

        printf("x[%d] = ", i);

        scanf("%lf", &x\_data[i]);

        printf("y[%d] = ", i);

        scanf("%lf", &y\_data[i]);

    }

    double x, y;

    while(1){

        printf("Enter the value of x for which you want to find the corresponding y: ");

        scanf("%lf", &x);

*// Perform the Lagrange interpolation to find y*

        y = lagrange\_interpolation(x, x\_data, y\_data, n);

*// Output the result*

        printf("The estimated value of y for x = %.6lf is: %.6lf\n", x, y);

        printf("\nPress [CTRL+C] to Terminate the Program.\*\*\*\*\*\*\n");

    }

    return 0;

}

**OUTPUT**

S:\WorkSpace\Numerical-Methods> gcc .\lagrange.c

S:\WorkSpace\Numerical-Methods> ./a.exe

Enter the number of data points: 4

Enter the x and y values for the data points:

x[0] = -1

y[0] = -8

x[1] = 0

y[1] = 3

x[2] = 2

y[2] = 1

x[3] = 3

y[3] = 2

Enter the value of x for which you want to find the corresponding y: 1

The estimated value of y for x = 1.000000 is: 3.666667

Press [CTRL+C] to Terminate the Program.\*\*\*\*\*\*

Enter the value of x for which you want to find the corresponding y: 4

The estimated value of y for x = 4.000000 is: 13.666667

Press [CTRL+C] to Terminate the Program.\*\*\*\*\*\*

Enter the value of x for which you want to find the corresponding y: 2.5

The estimated value of y for x = 2.500000 is: 0.604167

1. **WAP to Implement Gauss Seidel Method. And find value of x,y,z for the following system of equation.**

**2x-y+0z=7**

**-x+2y-z=1**

**0x-y+2z=1**

#include <stdio.h>

#include <math.h>

*// Function to solve the system of equations using Gauss-Seidel method*

void gaussSeidel(double a[3][3], double b[3], double x[3], double tolerance, int maxIter) {

    double x\_old[3];

    int iter = 0;

    double error;

*// Initialize x with initial guess (0,0,0)*

    for (int i = 0; i < 3; i++) {

        x[i] = 0.0;

    }

    do {

*// Store the current values of x in x\_old*

        for (int i = 0; i < 3; i++) {

            x\_old[i] = x[i];

        }

*// Update x, y, z using Gauss-Seidel iterative formulas*

        x[0] = (b[0] - a[0][1] \* x[1] - a[0][2] \* x[2]) / a[0][0];

        x[1] = (b[1] - a[1][0] \* x[0] - a[1][2] \* x[2]) / a[1][1];

        x[2] = (b[2] - a[2][0] \* x[0] - a[2][1] \* x[1]) / a[2][2];

*// Calculate the error (max difference between old and new values)*

        error = 0.0;

        for (int i = 0; i < 3; i++) {

            error = fmax(error, fabs(x[i] - x\_old[i]));

        }

        iter++;

    } while (error > tolerance && iter < maxIter);

*// Print the result*

    if (error <= tolerance) {

        printf("Solution converged after %d iterations.\n", iter);

        printf("x = %.6lf, y = %.6lf, z = %.6lf\n", x[0], x[1], x[2]);

    } else {

        printf("Solution did not converge after %d iterations.\n", iter);

    }

}

int main() {

    double a[3][3], b[3], x[3];

    double tolerance;

    int maxIter;

*// Input the coefficients of the system of equations*

    printf("Enter the coefficients for the system of equations (3 equations, 3 variables):\n");

    for (int i = 0; i < 3; i++) {

        printf("Equation %d:\n", i + 1);

        printf("a%d1, a%d2, a%d3: ", i + 1, i + 1, i + 1);

        scanf("%lf %lf %lf", &a[i][0], &a[i][1], &a[i][2]);

        printf("b%d: ", i + 1);

        scanf("%lf", &b[i]);

    }

*// Input tolerance and maximum number of iterations*

    printf("Enter the tolerance for convergence: ");

    scanf("%lf", &tolerance);

    printf("Enter the maximum number of iterations: ");

    scanf("%d", &maxIter);

*// Solve the system of equations using Gauss-Seidel method*

    gaussSeidel(a, b, x, tolerance, maxIter);

    return 0;

}

**OUTPUT:**

S:\WorkSpace\Numerical-Methods> gcc .\gauss-seidel.c

S:\WorkSpace\Numerical-Methods> ./a.exe

Enter the coefficients for the system of equations (3 equations, 3 variables):

Equation 1:

a11, a12, a13: 2 -1 0

b1: 7

Equation 2:

a21, a22, a23: -1 2 -1

b2: 1

Equation 3:

a31, a32, a33: 0 -1 2

b3: 1

Enter the tolerance for convergence: 0.00001

Enter the maximum number of iterations: 100

Solution converged after 20 iterations.

x = 5.999995, y = 4.999995, z = 2.999997

1. **Implementing Simpson's 1/3 Rule for Numerical Integration. Where,**

***f(x) = 1/(1+x2) and a = 0, b=1, taking a appropriate value of h.***

#include <stdio.h>

#include <math.h>

double f(double x) {

*// taking function: f(x) = 1/(1+x2)*

    return 1/(1+x\*x);

}

*// Function to perform Simpson's 1/3 rule*

double simpson13(double a, double b, int n) {

    if (n % 2 != 0) {

        printf("Number of intervals (n) must be even for Simpson's 1/3 rule.\n");

        return -1; *// Return error code*

    }

*// Step size (h)*

    double h = (b - a) / n;

*// Apply Simpson's 1/3 rule formula*

    double sum = f(a) + f(b);

*// Apply the 4\*f(x\_i) terms (odd indices)*

    for (int i = 1; i < n; i += 2) {

        sum += 4 \* f(a + i \* h);

    }

*// Apply the 2\*f(x\_i) terms (even indices)*

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

        sum += 2 \* f(a + i \* h);

    }

*// Final result*

    double result = sum \* h / 3;

    return result;

}

int main() {

    double a, b, result;

    int n;

*// Ask user for the limits of integration*

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

    scanf("%lf", &a);

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

    scanf("%lf", &b);

*// Ask for the number of subintervals (n must be even)*

    printf("Enter the number of subintervals (n, must be even): ");

    scanf("%d", &n);

*// Perform Simpson's 1/3 rule integration*

    result = simpson13(a, b, n);

*// Output the result if valid*

    if (result != -1) {

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

    }

    return 0;

}

**OUTPUT**

S:\WorkSpace\Numerical-Methods> gcc simpson1by3.c

S:\WorkSpace\Numerical-Methods> ./a.exe

Enter the lower limit (a): 0

Enter the upper limit (b): 1

Enter the number of subintervals (n, must be even): 100

The estimated value of the integral is: 0.785398

S:\WorkSpace\Numerical-Methods> ./a.exe

Enter the lower limit (a): 0

Enter the upper limit (b): 3

Enter the number of subintervals (n, must be even): 10

The estimated value of the integral is: 1.249014