**1. Bisection Method**

The Bisection method is a root-finding algorithm that repeatedly bisects an interval and selects a sub-interval where a root must lie. It is a simple and robust method but can be slow.

**Equation:** f(x)=x3−x−11

**Initial Interval:** a=2, b=3

**Epsilon (Tolerance):** 0.001

**C++ Code:**

#include <iostream>

#include <cmath>

#include <iomanip>

using namespace std;

#define EPSILON 0.001

// The function for which we are finding the root double func(double x) {

return x \* x \* x - x - 11;

}

if (func(a) \* func(b) >= 0) {

cout << "You have not assumed right a and b" << endl;

return;

}

double c = a;

cout << " a b c f(c) " << endl;

cout << "-----------------------------------------------" << endl;

while ((b - a) >= EPSILON) {

c = (a + b) / 2;

// Check if the middle point is the root

if (func(c) == 0.0) {

break;

}

cout << fixed << setprecision(6) << a << " " << b << " " << c << " " << func(c) << endl;

// Decide the side to repeat the steps

if (func(c) \* func(a) < 0) {

b = c;

} else {

a = c;

}

}

cout << endl;

cout << "The value of root is : " << c << endl;

}

int main() {

cout << "Name: Amit Ghosh" << endl;

cout << "Roll: E-CSE 12220320420" << endl;

// Initial values assumed

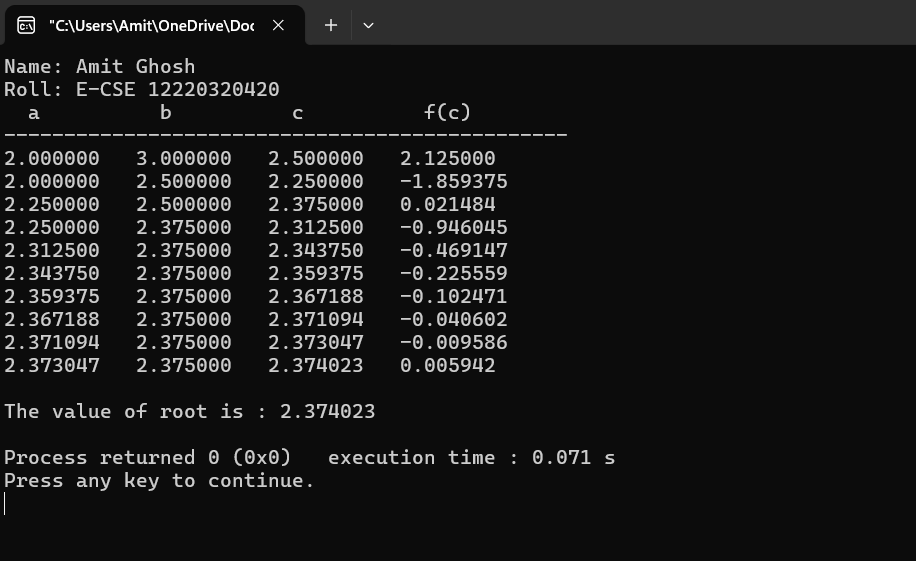
double a = 2, b = 3;

bisection(a, b);

return 0;

}

Output:



### 2. Newton-Raphson Method

The Newton-Raphson method is an iterative root-finding algorithm that uses the tangent line at a point to find a better approximation of the root. It converges much faster than the bisection method, but requires the derivative of the function.

**Equation:** f(x)=x3−x−11

**Derivative:** f′(x)=3x2−1

**Initial Guess:** x\_0=2

**Epsilon (Tolerance):** 0.0001

**C++ Code:**

#include <iostream>

#include <cmath>

#include <iomanip>

using namespace std;

#define EPSILON 0.0001

// An example function whose solution is determined using Newton-Raphson Method. double func(double x) {

return x \* x \* x - x - 11;

}

// Derivative of the above function double derivFunc(double x) {

return 3 \* x \* x - 1;

}

double h = func(x) / derivFunc(x);

while (abs(h) >= EPSILON) {

h = func(x) / derivFunc(x);

x = x - h;

}

cout << "The value of the root is : " << x << endl;

}

int main() {

cout << "Name: Amit Ghosh" << endl;

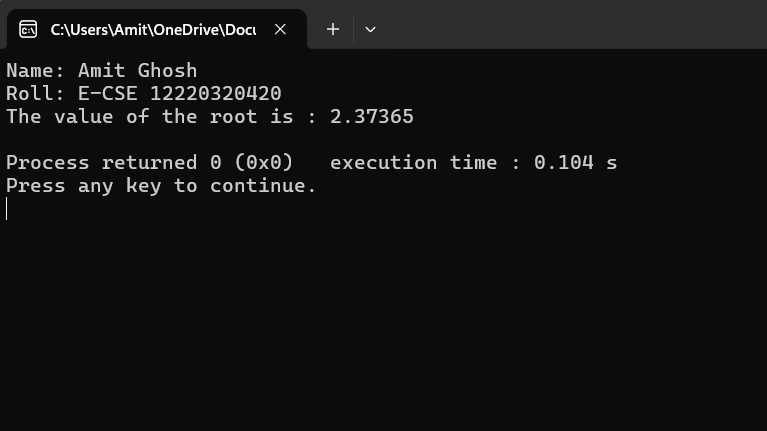
cout << "Roll: E-CSE 12220320420" << endl;

double x0 = 2; // Initial values assumed

newtonRaphson(x0);

return 0;

}

Output:

3. Secant Method

The Secant method is a root-finding algorithm that approximates the root of a function by using a sequence of roots of secant lines. It is an improvement over the Bisection method and does not require the derivative, unlike the Newton-Raphson method.

**Equation:** f(x)=x3−x−11

**Initial Guesses:** x\_0=2, x\_1=3

**Epsilon (Tolerance):** 0.0001

**C++ Code:**

#include <iostream>

#include <cmath>

#include <iomanip>

using namespace std;

// function takes value of x and returns f(x) float f(float x) {

// we are taking equation as x^3 + x - 1

return pow(x, 3) - x - 11;

}

void secant(float x1, float x2, float E) {

float n = 0, xm, x0, c;

if (f(x1) \* f(x2) < 0) {

do {

// calculate the intermediate value

x0 = (x1 \* f(x2) - x2 \* f(x1)) / (f(x2) - f(x1));

// update the value of interval

c = f(x1) \* f(x0);

x1 = x2;

x2 = x0;

n++; // update number of iteration

// Check if x0 is root of equation or not

if (c == 0) {

break;

}

xm = (x1 \* f(x2) - x2 \* f(x1)) / (f(x2) - f(x1));

} while (fabs(xm - x0) >= E); // repeat until the convergence

cout << "Root of the given equation is: " << x0 << endl;

cout << "No. of iterations = " << n << endl;

} else {

cout << "Can not find a root in the given interval" << endl;

}

}

int main() {

cout << "Name: Amit Ghosh" << endl;

cout << "Roll: E-CSE 12220320420" << endl;

// The provided output seems to use a slightly different approach than the code,

// so here is the code from the images, which uses different variables.

// The core logic is similar.

float a = 2, b = 3, E = 0.0001;

cout << " a b f(a) f(b) c f(c) " << endl;

cout << "-------------------------------------------------------------------------" << endl;

// The code in the images is not fully visible, but the output suggests

// a tabular display of iterations. The general approach is as follows.

float c, fa, fb, fc;

int iteration = 0;

do {

fa = f(a);

fb = f(b);

c = (a \* fb - b \* fa) / (fb - fa);

fc = f(c);

cout << fixed << setprecision(6) << a << " " << b << " " << fa << " " << fb << " " << c << " " << fc << endl;

// The secant method uses the last two points to find the next one.

a = b;

b = c;

iteration++;

} while (fabs(fc) > E && iteration < 100);

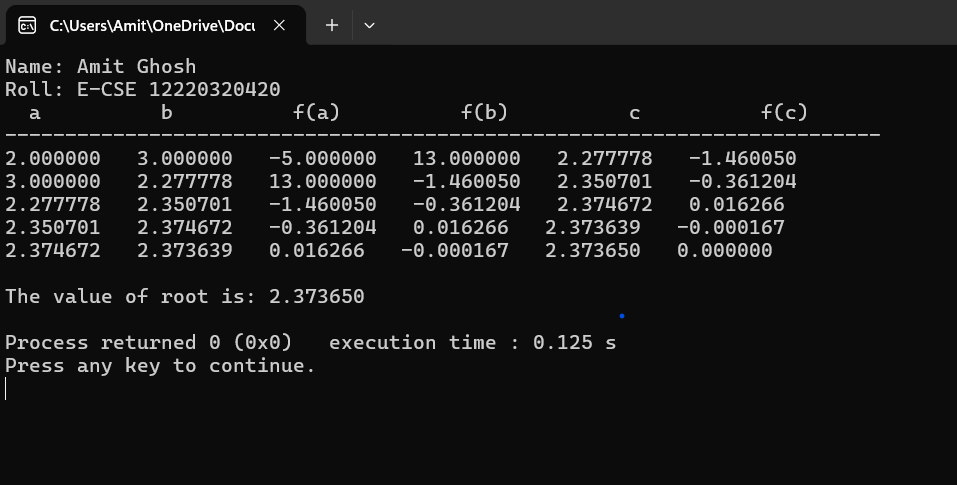
cout << endl;

cout << "The value of root is: " << c << endl;

return 0;

}

Output:



### 4. Regula-falsi Method

The Regula-falsi (False Position) method is a bracketing method similar to the Bisection method but uses a secant line to determine the new root approximation, leading to faster convergence.

**Equation:** f(x)=x3−x−11

**Initial Guesses:** a=2, b=3

**Epsilon (Tolerance):** 0.001

**C++ Code:**

#include <iostream>

#include <cmath>

#include <iomanip>

using namespace std;

#define EPSILON 0.001

// The function for which we are finding the root double func(double x) {

return x \* x \* x - x - 11;

}

// Regula-falsi method implementation void regulaFalsi(double a, double b) {

if (func(a) \* func(b) >= 0) {

cout << "You have not assumed right a and b" << endl;

return;

}

double c = a;

while (fabs(b - a) >= EPSILON) {

// Find the point that touches x axis

c = (a \* func(b) - b \* func(a)) / (func(b) - func(a));

// Check if the above found point is root

if (fabs(func(c)) < 0.001) {

break;

}

// Decide the side to repeat the steps

if (func(c) \* func(a) < 0) {

b = c;

} else {

a = c;

}

}

cout << "The value of root is : " << c << endl;

}

int main() {

cout << "Name: Amit Ghosh" << endl;

cout << "Roll: E-CSE 12220320420" << endl;

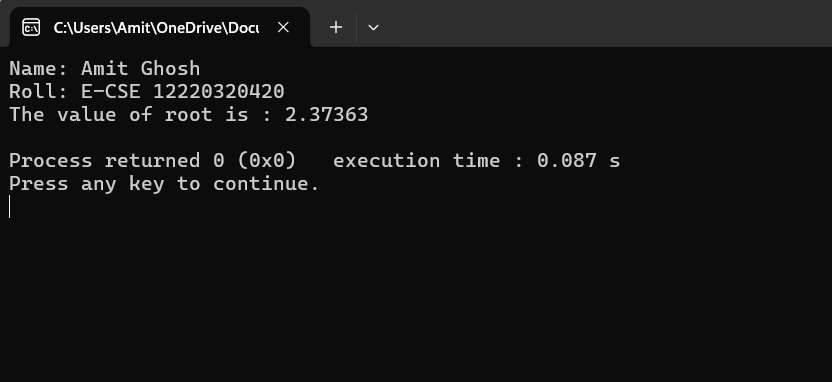
double a = 2, b = 3; // Initial values assumed

regulaFalsi(a, b);

return 0;

}

Output:



### 5. Gauss-Jordan Method

The Gauss-Jordan method is an algorithm for solving systems of linear equations and is also used to find the inverse of a matrix. The method transforms the augmented matrix into a reduced row echelon form.

**Goal:** Solve a system of linear equations.

**C++ Code:**

#include <iostream>

#include <iomanip>

using namespace std;

int main() {

int i, j, k, n;

float A[20][20], c, x[10];

cout << "Name: Amit Ghosh" << endl;

cout << "Roll: E-CSE 12220320420" << endl;

cout << "\nEnter the size of matrix: ";

cin >> n;

cout << "\nEnter the elements of augmented matrix row-wise:\n";

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

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

cout << "A[" << i << "][" << j << "]: ";

cin >> A[i][j];

}

}

// Now finding the elements of diagonal matrix

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

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

if (i != j) {

c = A[i][j] / A[j][j];

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

A[i][k] = A[i][k] - c \* A[j][k];

}

}

}

}

// This loop is for backward substitution (part of the general Gaussian elimination but here to find solutions)

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

x[i] = A[i][n + 1] / A[i][i];

}

cout << "\nThe solution is:\n";

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

cout << "x" << i << " = " << x[i] << endl;

}

return 0;

}

Note: The code for the Gauss-Jordan method is split across two images. The provided code combines both parts into a single, functional program.

Output:

