1. To find the roots of non-linear equation using Bisection method.

Solution:

Algorithm of Bisection Method for finding root:

1. **Input**: Function $f(x)$, interval $[a, b]$, and a tolerance value.
2. Check if $f(a) \times f(b) \geq 0$. If true, stop: no root exists within this interval.
3. Set $c = \frac{a+b}{2}$.
4. While $|b - a| \geq \text{tolerance}$:
   a. Evaluate $f(c)$.
   b. If $f(c) = 0$, then c is the root. Stop the process.
   c. If $f(c) \times f(a) < 0$, set $b = c$.
   d. If $f(c) \times f(b) < 0$, set $a = c$.
   e. Update $c = \frac{a+b}{2}$.
5. Output the final value of c as the approximate root.

Suppose we have a function: $f(x) = 3x - cos(x) - 1$

Now we need a and b. [0,1]
এখানে a এবং b এর মান নেয়ার সময় একটা কন্ডিশন মাথায় রাখতে হবে। তা হলোঃ $f(a) * f(b) < 0$;

এখানে আমরা a এবং b এর জন্য এমন মান নিবো যাতে ২টা ফাংশন গুন করলে ০ এর ছোট হয়।
এ জন্য আমরা [0,1] এটা না হলে [1,2] এটা না হলে [2,3] এভাবে চলতে থাকবে I তাও না হলে মাইনেস মান দিয়েও আমরা চেক করে দেখবো।

$a = 0 \qquad f(a) = f(0) = 3 * 0 - cos\,0 - 1 = -2$

$b = 1 \qquad f(b) = f(1) = 3 * 1 - cos\,1 - 1 = 1.46$

$\therefore f(a) * f(b) < 0$

$\Rightarrow -2 * 1.46$

$\Rightarrow -2.92$ [Note: এখানে আমরা দেখতে পারছি শর্ত মেনেছে তাই আমরা ধরে নইতে পারি আমাদের রুট ০ আর ১ এর মাঝে আছে]

Let's Find the root:

Befor jump we need to know 1 thing:
$If\ f(a) * f(c) = positive\ value\ then\ a = c;$

$If\ f(a) * f(c) = negative\ value\ then\ b = c;$

Now Lets go:

Iteration	a	b	$f(a)$	$f(b)$	$c = \frac{a+b}{2}$	$f(c)$
1	0	1	-2	1.4597	0.5	-0.377583
2	0.5	1	-0.377583	1.4597	0.75	0.518311
3	0.5	0.75	-0.377583	0.518311	0.625	0.0640369
4	0.5	0.625	-0.377583	0.0640369	0.5625	-0.158424
5	0.5625	0.625	-0.158424	0.0640369	0.59375	-0.0475985
6	0.59375	0.625	-0.0475985	0.0640369	0.609375	0.0081191
7	0.59375	0.609375	-0.0475985	0.0081191	0.601562	-0.0197649
8	0.601562	0.609375	-0.0197649	0.0081191	0.605469	-0.00582915
9	0.605469	0.609375	-0.00582915	0.0081191	0.607422	0.00114341
10	0.605469	0.607422	-0.00582915	0.00114341	0.606445	-0.00234326
11	0.606445	0.607422	-0.00234326	0.00114341	0.606445	-0.00234326

Solve with iteration :

```
#include <bits/stdc++.h>

using namespace std;

double equation(double x) {

    // Define your equation here

    // For example, let's solve 3*x - cos(x)  - 1

    return 3*x - cos(x)  - 1;

}


double bisectionMethod(double a, double b, double tolerance) {

    double c;
```

```
int n=1;
    while (fabs(b - a    ) >= tolerance) {

        c = (a + b) / 2;
cout<<"Iteration: "<<n<< " a = " << a << " b = " << b << " f(a) "<<equation(a)<< " f(b) "<<equation(b)<<" c = "<<c<<" f(c) "<<equation(c)<<endl;

        if (equation(c) == 0.0)
            return c;

        if (equation(c) * equation(a) < 0)
            b = c;
        else
            a = c;
n++;
    }
    cout<<"Iteration: "<<n<< " a = " << a << " b = " << b << " f(a) "<<equation(a)<< " f(b) "<<equation(b)<<" c = "<<c<<" f(c) "<<equation(c)<<endl;

    return c;
}

int main() {
```

```
    double a, b, tolerance;

    cout << "Enter the interval [a, b]: ";
    cin >> a >> b;

    cout << "Enter the tolerance: ";
    cin >> tolerance;

    double root = bisectionMethod(a, b, tolerance);

    cout << "Approximate root: " << root << endl;

    return 0;
}
```

Additional info:

The condition `while (fabs(b - a) >= tolerance)` in the code ensures that the bisection method keeps running until the interval between the two values (let's call them `a` and `b`) becomes smaller than the desired level of accuracy, which is defined as `tolerance`.

Imagine you're trying to find where a number is on a line, but you can only see a range on that line (from `a` to `b`). To determine the number more precisely, you need to keep reducing the range until it's very small. The `while` condition does just that – it keeps the method running until the range (the difference between `a` and `b`) is tinier than what you consider acceptable (tolerance). This helps to pinpoint the location of the number you're seeking.

So, the smaller the `tolerance`, the more precise the final result will be, because it forces the method to keep refining the range until it's very, very small, giving a more accurate approximation of the number you're looking for.

Certainly! Let's consider the quadratic function:

$f(x) = (x - a)(x - b)$

where a and b are the roots of the function. You can choose any values for a and b. For example:

$f(x) = (x - 2)(x + 3)$

In this case, the roots are $x = 2$ and $x = -3$. You can replace 2 and -3 with any other values to create a quadratic function with different roots.

MD

multiply (x−2)(x+3)

Sure, let's multiply $(x - 2)(x + 3)$:

$$(x - 2)(x + 3) = x \cdot x + x \cdot 3 - 2 \cdot x - 2 \cdot 3$$

Simplifying the expression:

$$= x^2 + 3x - 2x - 6$$

Combining like terms:

$$= x^2 + x - 6$$

So, the product of $(x - 2)(x + 3)$ is $x^2 + x - 6$.

Regenerate

2. To find the roots of non-linear equation using False-Position method.

Algorithm:

1. **Define the function:**
   - $f(x)$ is the function for which the root is to be found.
2. **Initialize variables:**
   - a and b are the endpoints of the interval.
   - TOL is the tolerance for the solution.
   - c is the current approximation to the root.
3. **Iterate until convergence:**
   - Repeat until $|f(c)| < TOL$:
     - Calculate the function values at the endpoints: $fa = f(a)$ and $fb = f(b)$.
     - Calculate the approximate root using the false position formula: $c = \frac{a \cdot fb - b \cdot fa}{fb - fa}$
     - If $f(c) = 0.0$, return c as the exact root.
     - Update the interval based on the signs of function values:
       - If $f(c) \cdot fa < 0$, set $b = c$.
       - Else, set $a = c$.
4. **Output the result:**
   - Return c as the final approximation to the root of $f(x)$ within the specified tolerance.

Iteration	a	b	$f(a)$	$f(b)$	$c = \frac{a*f(b)-b*f(a)}{f(b)-f(a)}$	$f(c)$
1	0	1	-2	1.4597	0.578085	-0.103255
2	0.578085	1	-0.103255	1.4597	0.605959	-0.0040808
3	0.605959	1	-0.0040808	1.4597	0.607057	-0.000159047
4	0.607057	1	-0.000159047	1.4597	0.607057	-0.000159047

<u>Solution:</u>

```
#include <bits/stdc++.h>

using namespace std;


double equation(double x) {
```

```
    // Define your equation here
    // For example, let's solve 3x-cos(x)-1
    return pow(x,2)+x-6;
}

double falsePositionMethod(double a, double b, double tolerance) {
    double c;

  while (fabs(equation(c)) >= tolerance){
        // Calculate the function values at the endpoints
        double fa = equation(a);
        double fb = equation(b);

        // Calculate the approximate root using the false position formula
        c = (a * fb - b * fa) / (fb - fa);
cout<<"Iteration: "<<1<< " a = " << a << " b = " << b << " f(a) "<<equation(a)<< " f(b)
"<<equation(b)<<" c = "<<c<<" f(c) "<<equation(c)<<endl;
        // Check if c is the root
        if (equation(c) == 0.0){
            return c;
        }
        // Update the interval based on the signs of function values
        if (equation(c) * fa < 0)
            b = c;
        else
            a = c;
```

```
    }

    return c;
}

int main() {
    double a, b, tolerance;

    cout << "Enter the interval [a, b]: ";
    cin >> a >> b;

    cout << "Enter the tolerance: ";
    cin >> tolerance;

    double root = falsePositionMethod(a, b, tolerance);

    cout << "Approximate root: " << root << endl;

    return 0;
}
```

3. To find the roots of non-linear equation using Newton's method.

$$f(x) = 3x - cosx - 1$$

<u>Newton Raphson Method:</u>

<u>Tangent formula:</u> $y - y' = \frac{dy}{dx}(x - x_1)$

Now: $y - f(x_0) = f'(x_0)(x_1 - x_0)$

$\Rightarrow 0 - f(x_0) = f'(x_0)(x_1 - x_0)$

$\Rightarrow x_1 - x_0 = -\frac{f(x_0)}{f'(x_0)}$

$\Rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$

So,That , $x_{n+1} = x_n - \frac{f(x_0)}{f'(x_0)}$

<u>Given function:</u>

$$f(x) = 3x - cosx - 1$$

$$f'(x) = 3 + sinx$$

Iteration	x_n	$f(x_n)$	$f'(x_n)$	$x_{n+1} = x_n - \frac{f(x_0)}{f'(x_0)}$
1	0	0.214113	3	0.666667
2	0.666667	0.00139686	3.61837	0.607493
3	0.607493	6.28295e-08	3.57081	0.607102

So, The Approximater root is: 0.607102

Solution:

```
#include<bits/stdc++.h>


using namespace std;


// Define your function here 3x-cos(x)-1
double equation(double x) {
    return 3*x-cos(x)-1;
}


// Define the derivative of your function here
double derivative(double x) {
    return 3+sin(x);
}




double newtonRaphson(double x0, double epsilon, int maxIterations) {
    double x = x0;
    int iterations = 0;


    while (fabs(equation(x)) > epsilon ) {
        cout<<"x = "<<x;
        x = x - (equation(x) / derivative(x));
        cout<< " f(x) = "<<equation(x)<<"  f'(x) = "<<derivative(x)<<"  Xn = "<<x<<endl;
        iterations++;
```

```
    }

    return x;
}

int main() {
    double initialGuess = 0;
    double epsilon = 0.001;
    int maxIterations = 100;

    double root = newtonRaphson(initialGuess, epsilon, maxIterations);

    cout << "Approximate root: " << root << endl;

    return 0;
}
```

Or,

<u>version-2:</u>

```
#include <bits/stdc++.h>

using namespace std;

double equation(double x) {
    return 3 * x - cos(x) - 1; // Change this function as needed
```

```
}

double numericalDerivative(double x, double h) {
    return (equation(x + h) - equation(x - h)) / (2 * h);
}

double newtonRaphson(double x0, double epsilon, int maxIterations, double h) {
    double x = x0;
    int iterations = 0;

    while (fabs(equation(x)) > epsilon && iterations < maxIterations) {
        cout << "x = " << x;
        double derivative = numericalDerivative(x, h);
        x = x - (equation(x) / derivative);
        cout << " f(x) = " << equation(x) << "  f'(x) = " << derivative << "  Xn = " << x << 
endl;
        iterations++;
    }
    return x;
}

int main() {
    double initialGuess = 0;
    double epsilon = 0.001;
    int maxIterations = 100;
```

```
    double h = 0.0001; // Step size for numerical derivative

    double root = newtonRaphson(initialGuess, epsilon, maxIterations, h);

    cout << "Approximate root: " << root << endl;

    return 0;
}
```

11. To integrate numerically using the trapezoidal rule.

$\int_{1}^{2} \frac{1}{x} dx \quad and \quad n = 10$

Solution:

Formula:

$i) \int_{a}^{b} f(x)dx = \frac{\Delta x}{2} * \left[f(x_0) + 2f(x_1) + 2f(x_2) + 2f(x_3) + 2f(x_4) + ... + 2f(x_{n-1}) + f(x_n)\right]$

$ii) \Delta x = \frac{b-a}{n} \quad note: \left[\frac{upper\ limit - Lower\ limit}{n}\right]$

$ii) x_i = a + i\Delta x$

Here, $\Delta x = \frac{b-a}{n} = \frac{2-1}{10} = 0.1$

$x_i = a + i\Delta x$

So,

$x_0 = 1 + 0 \times 0.1 = 1$

$x_1 = 1 + 1 \times 0.1 = 1.1$

$x_2 = 1 + 2 \times 0.1 = 1.2$

$x_3 = 1 + 3 \times 0.1 = 1.3$

$x_4 = 1 + 4 \times 0.1 = 1.4$

$x_5 = 1 + 5 \times 0.1 = 1.5$

$x_6 = 1 + 6 \times 0.1 = 1.6$

$x_7 = 1 + 7 \times 0.1 = 1.7$

$x_8 = 1 + 8 \times 0.1 = 1.8$

$x_9 = 1 + 9 \times 0.1 = 1.9$

$x_{10} = 1 + 10 \times 0.1 = 2$

Now,

$$\int_{a}^{b} f(x)dx = \frac{\Delta x}{2} * \left[f(x_0) + 2f(x_2) + 2f(x_3) + 2f(x_3) + 2f(x_4) + ... + 2f(x_{n-1}) + f(x_n)\right]$$

$$= \int_{1}^{2} \frac{1}{x}dx = \frac{0.1}{2}[\frac{1}{1} + 2 \times \frac{1}{1.1} + 2 \times \frac{1}{1.2} + 2 \times \frac{1}{1.3} + 2 \times \frac{1}{1.4} + 2 \times \frac{1}{1.5} + 2 \times \frac{1}{1.6} + 2 \times \frac{1}{1.7}$$

$$+ 2 \times \frac{1}{1.8} + 2 \times \frac{1}{1.9} + \frac{1}{2}]$$

$= \frac{0.1}{2}[1 + 12.374 + 0.5]$

= 0.6937

Code:

```
#include <bits/stdc++.h>

using namespace std;

// Define the function to integrate, f(x) = 1/x
double f(double x) {
   return 1/x;
}

// Implement the trapezoidal rule
double trapezoidalRule(double a, double b, int n) {
   double h = (b - a) / n;  // Step size
   double integral = f(a) + f(b);  // Sum the first and last terms

   for (int i = 1; i < n; i++) {  // Using post-increment here
      integral += 2 * f(a + i * h);  // Sum the interior terms with weight 2
   }

   integral *= h/2;  // Multiply by the step size divided by 2

   return integral;
}

int main() {
   double a = 1, b = 2;  // Limits of integration
   int n = 10;  // Number of subdivisions

   double result = trapezoidalRule(a, b, n);

   // Set precision for output to 5 decimal places
   cout << fixed << setprecision(5);
   cout << "The integral is approximately: " << result << endl;

   return 0;
}
```

12. To integrate numerically using Simpson's 1/3 rule.

$\int_{1}^{2} \frac{1}{x}dx \quad and \quad n = 10$

<u>Solution:</u>

<u>Formula:</u>

$i)\ \int_{a}^{b} f(x)dx = \frac{\Delta x}{3} = \left[f(x_0) + 4f(x_1) + 2f(x_2) + 2f(x_3) + 2f(x_4) + ... + 4f(x_{n-1}) + f(x_n)\right]$

$ii)\ \Delta x = \frac{b-a}{n} \quad note: \left[\frac{upper\ limit - Lower\ limit}{n}\right]$

$ii)\ x_i = a + i\Delta x$

Here, $\Delta x = \frac{b-a}{n} = \frac{2-1}{10} = 0.1$

$x_i = a + i\Delta x$

So,

$x_0 = 1 + 0 \times 0.1 = 1$

$x_1 = 1 + 1 \times 0.1 = 1.1$

$x_2 = 1 + 2 \times 0.1 = 1.2$

$x_3 = 1 + 3 \times 0.1 = 1.3$

$x_4 = 1 + 4 \times 0.1 = 1.4$

$x_5 = 1 + 5 \times 0.1 = 1.5$

$x_6 = 1 + 6 \times 0.1 = 1.6$

$x_7 = 1 + 7 \times 0.1 = 1.7$

$x_8 = 1 + 8 \times 0.1 = 1.8$

$x_9 = 1 + 9 \times 0.1 = 1.9$

$x_{10} = 1 + 10 \times 0.1 = 2$

Now,

$\int_{a}^{b} f(x)dx = \frac{\Delta x}{3} = \left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + ... + 4f(x_{n-1}) + f(x_n)\right]$

$= \int_{1}^{2} \frac{1}{x}dx = \frac{0.1}{3}[\frac{1}{1} + 4 \times \frac{1}{1.1} + 2 \times \frac{1}{1.2} + 2 \times \frac{1}{1.3} + 2 \times \frac{1}{1.4} + 2 \times \frac{1}{1.5} + 2 \times \frac{1}{1.6} + 2 \times \frac{1}{1.7}$

$+ 2 \times \frac{1}{1.8} + 4 \times \frac{1}{1.9} + \frac{1}{2}]$

$= \frac{0.1}{3} \times 20.7947$
= 0.69315

Code:
```
#include <bits/stdc++.h>
using namespace std;

// Define the function to be integrated
double f(double x) {
    if (x == 0) {
        return numeric_limits<double>::infinity(); // Avoid division by zero
    }
    return 1 / x;
}

// Implement Simpson's 1/3 rule
double simpsonsOneThirdRule(double a, double b, int n) {
    double h = (b - a) / n; // Calculate the interval size
    double sum = f(a) + f(b); // f(x_0) + f(x_n)

    // Apply Simpson's 1/3 rule
    for (int i = 1; i < n; i++) {
        double x_i = a + i * h;
        if (i % 2 == 0) {
            sum += 2 * f(x_i); // Even index terms are multiplied by 2
        } else {
            sum += 4 * f(x_i); // Odd index terms are multiplied by 4
        }
    }
    return (h / 3) * sum;
}

int main() {
    double lower_limit = 1;
    double upper_limit = 2;
    int n = 10; // Number of intervals

    // Calculate the integral
    double result = simpsonsOneThirdRule(lower_limit, upper_limit, n);

    // Output the result
    cout << " using Simpson's 1/3 rule is: "
         << setprecision(5) << fixed << result << endl;

    return 0;
}
```