EE1103: Numerical Methods

Programming Assignment # 1

Shreya S Ramanujam, EE21B126

February 19, 2022

Contents

List of Figures

List of Tables

1 Problem 1

Determine the positive real root of

$$f(x) = \ln(x^4) - 0.7 \tag{1}$$

(a) Plot the function by choosing a reasonable range for x to understand the nature of the function

(b) Find its root using three iterations of the bisection method, with initial guesses of $xl = 0.5$ and $xu = 2$, and

(c) Use three iterations of the false-position method, with the same initial guesses as in (b) and find the root.

(d) Summarize your results in a table - against each iteration, record the approximate value and approximate error in both the methods.

1.1 Approach

For part (b) of this problem, we use a `for` loop to iterate over the Bisection algorithm (described in subsection 1) 3 times.

In each iteration, we calculate our new root approximation as mid-point of upper and lower bounds (xu and xl respectively), and update our bounds after checking where the root lies (using Intermediate Value Theorem condition).

For part (c) of this problem, we use a `for` loop to iterate over the False Position algorithm (described in subsection 2) 3 times.

In each iteration, we calculate our new root approximation by finding where the chord connecting $(xu, f(xu))$ and $(xl, f(xl))$ cuts the x-axis, and then update upper and lower bounds (xu and xl respectively) by checking condition given by Intermediate Value Theorem.

Additionally, these programs also print their runtimes.

1.2 Algorithm

The pseudocode for Bisection method solution is provided in Algorithm 1.

Algorithm 1: Approximating root of $f(x)$ using Bisection Method

1. Include necessary header files
2. Initialize time t to clock time at beginning of program for runtime calculation
3. Initialize $xl = 0.5$ and $xu = 2.0$ as our initial guesses for lower and upper bounds respectively
4. Declare $xr = 0, fxl = 0, fxr = 0, approxerror, xrold = 0$
5. Calculate $xr = \frac{xl+xu}{2}$
6. Calculate $approxerror = |\frac{xr-xrold}{xr} \times 100|$
7. Calculate fxl and fxr ($f(x) = \ln(x^4) - 0.7$ values at $x = xl$ and $x = xr$ respectively)
8. If $fxl \times fxr < 0$, then set xu = xr else if $fxl \times fxr > 0$ set xl = xr
9. Update $xrold = xr$
10. Display Iteration No (i), approximate value of root (xr) and approximate error % ($approxerror$)
11. If $i <= 3$ then return to step 5, else go to step 12
12. Declare answer as xr
13. Calculate and display runtime
14. Stop

The pseudocode for False Position method solution is provided in Algorithm 2.

Algorithm 2: Approximating root of $f(x)$ using False Position Method

1. Include necessary header files
2. Create function $float\ func(float\ x)$ to calculate the value of $f(x) = \ln{(x^4)} - 0.7$ when called. Include its function prototype at beginning of program, right after header files
3. Initialize time t to clock time at beginning of *main()* for runtime calculation
4. Initialize $xl = 0.5$ and $xu = 2.0$ as our initial guesses for lower and upper bounds respectively
5. Declare $fxu, fxl, xf, fxf, approxerror, xfold = 0.0$
6. Calculate $fxu = func(xu)$ and $fxl = func(xl)$
7. Calculate $xf = \frac{fxu \times xl - fxl \times xu}{fxu - fxl}$, $fxf = func(xf)$
8. Calculate $approxerror\% = |\frac{xf - xfold}{xf} \times 100|$
9. If $fxf \times fxl < 0$, then set xu = xf else if $fxf \times fxl > 0$ set xl = xf
10. Update $xfold = xf$
11. Display Iteration No (i), approximate value of root (xf) and approximate error% $(approxerror)$
12. If $i <= 3$ then return to step 6, else go to step 13
13. Declare answer as xf
14. Calculate and display runtime
15. Stop

1.3 Results

The results of the Bisection Method are summarized in Table 1.

Table 1: Bisection Method

Iteration no.	Approximate value of root	Approximate Error(%)
1	1.250000	100.000000
2	0.875000	42.857143
3	1.062500	17.647058

The results of the False Position Method are summarized in Table 2.

Table 2: False Position Method

Iteration no.	**Approximate value of root**	**Approximate Error(%)**
1	1.439354	100.000000
2	1.271271	13.221585
3	1.217534	4.413657

The Approximate Error % v/s Iteration No. graphs are given below in Figure 1 for Bisection Method and Figure 2 for False Position Method.

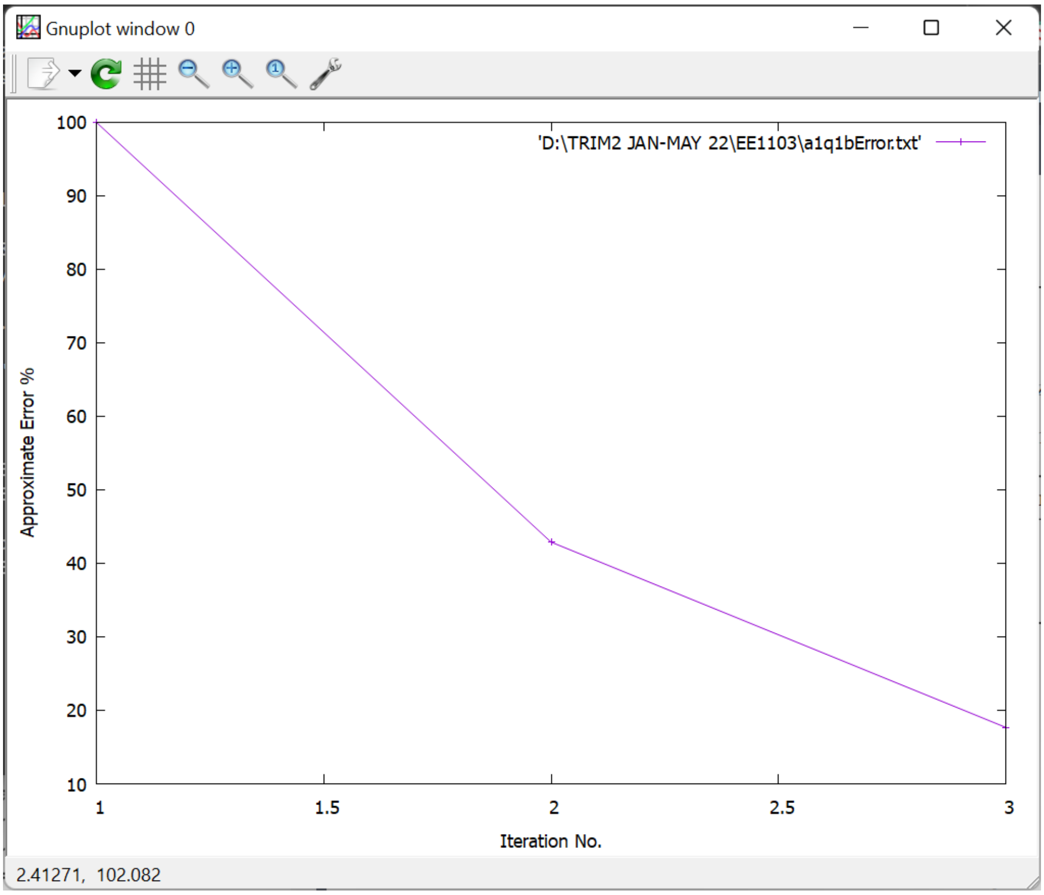


Figure 1: Approximate error(%) v/s Iteration no. for finding root by Bisection method

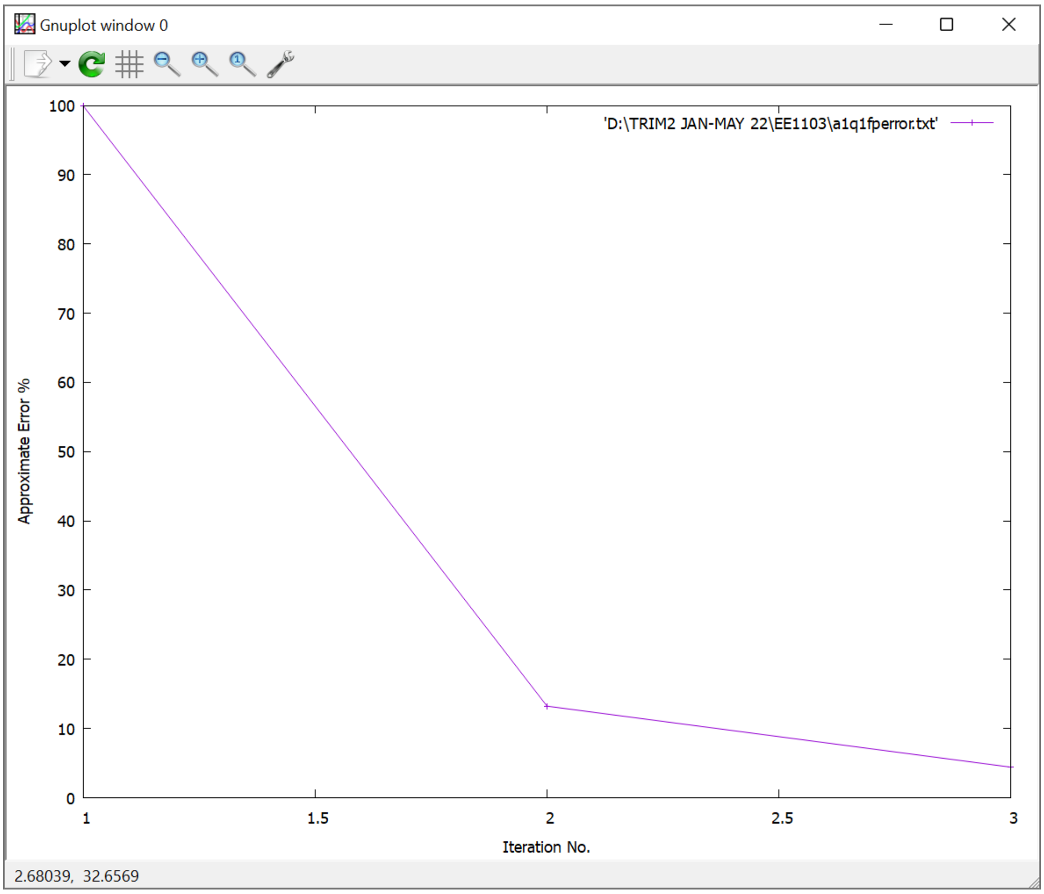


Figure 2: Approximate error(%) v/s Iteration no. for finding root by False Position method

1.4 Inferences

We deduce the following inferences from Problem 1:

- Approximate value of root calculated by Bisection method is 1.062500 with an approximate error of 17.647058% and an average runtime of 0.001000000000000 seconds
- Approximate value of root calculated by False Position method is 1.217534 with an approximate error of 4.413657% and an average runtime of 0.007000000000000 seconds
- We observe that False Position method gives a better approximation of the true root (1.91) than Bisection method for the given function, likely due to the shape of the graph of $f(x)$, which favours the False Position method (atleast in the initial iterations).
- Thus, for the three iterations that we have recorded, False Position gets an answer closer to the true root because the shape of the log function allows False position method to approach the root quicker than the Bisection method.
- The average runtime of Bisection method is lesser than that of False Position method, due to lesser computations and variables involved in Bisection method.

1.5 Code

The code used for solving Problem 1 using Bisection Method is mentioned in Listing 1.

Listing 1: Code snippet used in solving Problem 1 using Bisection Method.

```c
#include <stdio.h>
#include <math.h>
#include <time.h>

int main (void)
{
    //initializing time at beginning of program for runtime calculation
    clock_t t;
    t = clock();

    float xl = 0.5f , xu = 2.0f , xr = 0 , fxl = 0 , fxr = 0, approxerror, xrold = 0;
    printf ("Iteration No. \t Approximate Value \t Approximate Error \n\n");

    //loop for bisection iterations
    for (int i = 1; i <= 3; i++)
    {
        //calculating new root approximation "xr"
        xr = (xl + xu)/2;

        //calculating approximate error in current iteration
        approxerror = fabs(((xr - xrold)/xr) * 100);

        fxl = 4 * log(xl) - 0.7;
        fxr = 4 * log(xr) - 0.7;

        //checking condition and updating bounds accordingly
        if ((fxl * fxr) < 0)
            xu = xr;
        else if ((fxl * fxr) > 0)
            xl = xr;

        //update "xrold" for next iteration
        xrold = xr;

        //printing iteration no, approx value and approx error of current iteration
        printf ("%d \t\t %f \t\t %f%\n\n" , i, xr, approxerror);
    }
    //print answer
    printf ("Root by bisection method is %f\n" , xr);

    //runtime calculation and display
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC;
    printf("Time taken = %.15f\n", time_taken);
```

```

    return 0;
}
```

The code used for solving Problem 1 using False Position Method is mentioned in Listing 2.

Listing 2: Code snippet used in solving Problem 1 using False Position Method.

```
#include <stdio.h>
#include <math.h>
#include <time.h>

//function prototype
float func (float x);

int main (void)
{
    //initializing time at beginning of program for runtime calculation
    clock_t t;
    t = clock();

    float xl = 0.5f, xu = 2.0f, fxu, fxl, xf, fxf, approxerror, xfold =
    ↪ 0.0f;
    printf ("Iteration No. \t Approximate Value \t Approximate Error \n");

    //loop for false position method iterations
    for (int i = 1; i <= 3; i++)
    {
        fxu = func(xu);
        fxl = func(xl);

        //calculating new root approximation "xf"
        xf = (fxu * xl - fxl * xu)/(fxu - fxl);
        fxf = func(xf);

        //calculating approximate error in current iteration
        approxerror = fabs(((xf - xfold)/xf) * 100);

        //check given condition and update bounds accordingly for next
        ↪ iteration
        if (fxf * fxl < 0)
        {
            xu = xf;
        }
        else if (fxf * fxl > 0)
        {
            xl = xf;
        }

        //variable updation for next iteration
        xfold = xf;
```

```

        //print iteration no, approx value and approx error of current iteration
        printf ("%d \t\t %f \t\t %f%\n\n" , i, xf, approxerror);
    }
    //print result
    printf ("Root by False Position Method is %f\n", xf);

    //runtime calculation and display
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC;
    printf("Time taken = %.15f\n", time_taken);

    return 0;
}

float func (float x)
{
    //function calculates and returns value of f(x)
   float ans = 4 * log(x) - 0.7;
   return (ans);
}
```

1.6 Contributions

Individual submission. All items included in this subsection were done by me.

2 Problem 2

Sketch the following function in a sensible range to understand its nature and then employ the Newton-Raphson method to determine a real root for

$$f(x) = -2 + 6x - 4x^2 + 0.5x^3 \tag{2}$$

using initial guesses of (a) 4.2 (b) 4.43. Run the algorithm for 50 iterations in each case and plot the approximate error percentages as a function of iteration number. Comment on your results and discuss any peculiarities you observe in your results.

2.1 Approach

In this problem, we use a `for` loop to iterate over the Newton-Raphson algorithm (described in subsections 3 and 4) 50 times for any given initial guess.

In each iteration, we calculate the new root approximation as $x2 = x1 - \frac{fx1}{derx1}$. We also calculate the approximate error(%) and display it. When the loop ends after completing specified no. of iterations (here, 50), we declare $x2$ as our root and display it.

Additionally, this program also displays its runtime.

We can reuse the same code for both (a) and (b) subparts by just changing initialization of $x1$ when it is declared.

2.2 Algorithm

The pseudocode for Problem 2(a) is provided in Algorithm 3.

Algorithm 3: Determining root of $f(x)$ using Newton-Raphson method and an initial guess of 4.2

1. Include necessary header files.
2. Initialize time t to clock time at beginning of *main()* for runtime calculation.
3. Declare $fx1, derx1, x2, approxerror$.
4. Initialize $x1 = 4.2$ as our initial guess for root.
5. Calculate $fx1 = -2 + 6(x1) - 4(x1)^2 + 0.5(x1)^3$ and $derx1 = 1.5(x1)^2 - 8(x1) + 6$.
6. Calculate $x2 = x1 - \frac{fx1}{derx1}$.
7. Calculate $approxerror = |\frac{x2-x1}{x2} \times 100|$.
8. Update $x1 = x2$ for next iteration.
9. Display iteration no.(i) and approximate error ($approxerror$).
10. If $i <= 50$ then go to step 5 else go to step 11.
11. Declare Root as $x2$.
12. Calculate and display runtime.
13. Stop.

The pseudocode for Problem 2(b) is provided in Algorithm 4.

Algorithm 4: Determining root of $f(x)$ using Newton-Raphson method and an initial guess of 4.43

1. Include necessary header files.
2. Initialize time t to clock time at beginning of *main()* for runtime calculation.
3. Declare $fx1, derx1, x2, approxerror$.
4. Initialize $x1 = 4.43$ as our initial guess for root.
5. Calculate $fx1 = -2 + 6(x1) - 4(x1)^2 + 0.5(x1)^3$ and $derx1 = 1.5(x1)^2 - 8(x1) + 6$.
6. Calculate $x2 = x1 - \frac{fx1}{derx1}$.
7. Calculate $approxerror = |\frac{x2-x1}{x2} \times 100|$.
8. Update $x1 = x2$ for next iteration.
9. Display iteration no.(i) and approximate error ($approxerror$).
10. If $i <= 50$ then go to step 5 else go to step 11.
11. Declare Root as $x2$.
12. Calculate and display runtime.
13. Stop.

2.3 Results

The graph of the function

$$f(x) = -2 + 6x - 4x^2 + 0.5x^3 \tag{3}$$

is given in Figure 3

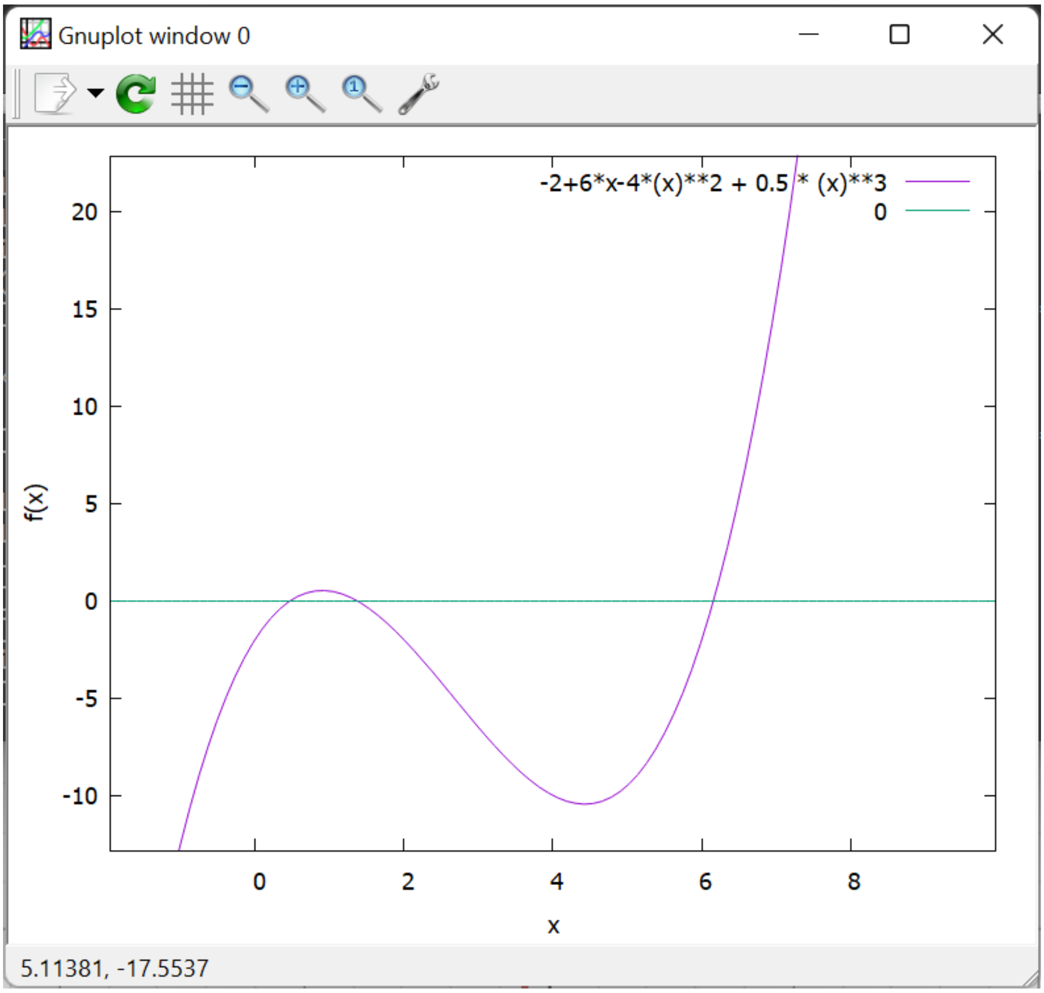


Figure 3: Graph of $f(x) = -2 + 6x - 4x^2 + 0.5x^3$

For both data sets (with initial guesses of 4.2 and 4.43), we plot the graph showing the approximate error percentage as a function of iteration number, as shown in Figure 4 (for Q2(a)) and 5 (for Q2(b)).

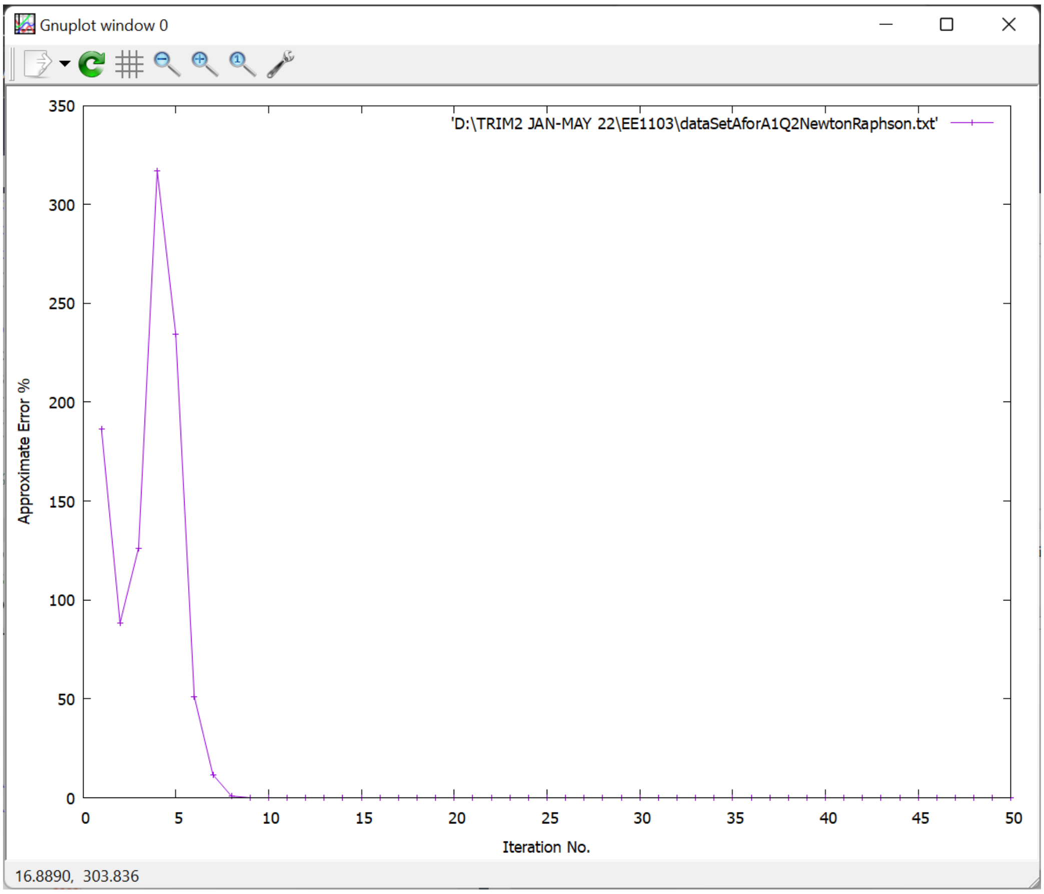


Figure 4: Approximate error(%) v/s Iteration no. for finding root by Newton-Raphson method with initial guess as 4.2

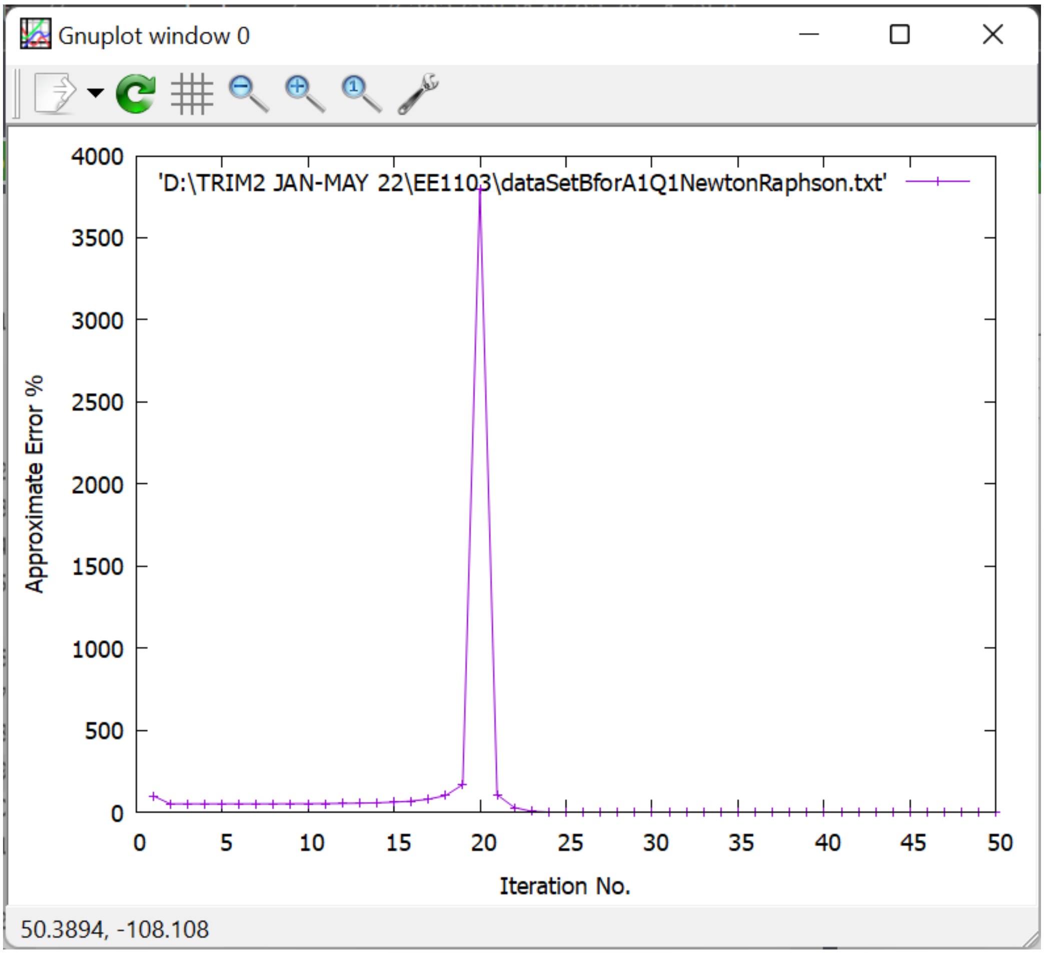


Figure 5: Approximate error(%) v/s Iteration no. for finding root by Newton-Raphson method with initial guess as 4.43

2.4 Inferences

We deduce the following inferences from Problem 2:

- Both initial guesses give the same root approximation (upto 15 decimal points) in 50 iterations, namely *0.474572449922562*.
- Runtime of 2(a) lies in the range of *0.008-0.009 seconds*, while runtime of 2(b) lies in the range of *0.007 to 0.009 seconds*
- Some pecularities can be observed in the approximate error percentage v/s iteration number graphs, as shown in Figure 4 (for Q2(a)) and 5 (for Q2(b)).
- In Figure 4, there is a dip in the approximate error % in Iteration 2 (88.39%), and a peak in the approximate error % in Iteration 4 (317.07%), and then the error decreases to become 0.000000000000000 from Iteration 10 onwards.
- In Figure 5, there is a steep peak in the approximate error% in Iteration 20 (3797.285%), and it becomes 0.000000000000000 from Iteration 26 onwards.
- From this data, we can see that an initial guess of 4.2 lets us converge to 0.000000000000000% approximate error faster.

- This observation can be explained using the graphical approach. (The graph for $f(x)$ is given in Figure 3)
- We see that 4.2 lies to the left of the minima of $f(x)$ while 4.43 lies to its right.
- So, when we start with 4.2 as our initial guess, the shape of the graph on the left of the minima is what guides our next root approximation. Here, the shape is such that the tangent at this point lets us get quite close to the root (0.4745) in the first iteration itself (first approximate root is -4.849116)
- But, with an initial guess of 4.43, we depend on the graph on the right side of the minima to guide us to the next root approximation, which takes us much farther from the root (0.4745) than we initially were (first approximate root is -3937.783447). This is why the convergence to the root is much slower for an initial guess of 4.43 when compared to 4.2.
- Thus, initial guess of 4.2 is much more efficient here.

2.5 Code

The code used for Problem 2(a) with initial guess as 4.2 is mentioned in Listing 4.

```
#include <stdio.h>
#include <math.h>
#include <time.h>

int main (void)
{
    //initializing time at beginning of program for runtime calculation
    ↪ later
    clock_t t;
    t = clock();

    float x1 = 4.2f, fx1, derx1, x2, approxerror;
    printf ("Iteration No. \t Approximate Error \n");

    //loop for Newton Raphson iterations
    for (int i = 1; i <= 50; i++)
    {
        fx1 = (0.5 * pow(x1, 3)) - 4 * pow(x1, 2) + 6 * x1 - 2;
        derx1 = (1.5 * pow(x1, 2)) - 8 * x1 + 6;

        //calculating new root approximation
        x2 = x1 - (fx1/derx1);

        //calculating approximate error % of current iteration
        approxerror = fabs(((x2 - x1)/x2) * 100);

        //updating x1 for next iteration
        x1 = x2;

        //printing iteration no and approximate error
```

```
        printf ("%d \t\t %.15f% \n\n" , i, approxerror);
    }
    //displaying answer
    printf ("Root by Newton Raphson Method is %.15f\n", x2);

    //runtime calculation and display
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC;
    printf("Time taken = %.15f\n", time_taken);

    return 0;
}
```

Listing 3: Code snippet used in Q2(a).

The code used for Problem 2(b) with initial guess as 4.43 is mentioned in Listing 4.

```
#include <stdio.h>
#include <math.h>
#include <time.h>

int main (void)
{
     //initializing time at beginning of program for runtime calculation later
    clock_t t;
    t = clock();

    float x1 = 4.43f, fx1, derx1, x2, approxerror;
    printf ("Iteration No. \t Approximate Error \n");

    //loop for Newton Raphson iterations
    for (int i = 1; i <= 50; i++)
    {
        fx1 = (0.5 * pow(x1, 3)) - 4 * pow(x1, 2) + 6 * x1 - 2;
        derx1 = (1.5 * pow(x1, 2)) - 8 * x1 + 6;

        //calculating new root approximation
        x2 = x1 - (fx1/derx1);

        //calculating approximate error % of current iteration
        approxerror = fabs(((x2 - x1)/x2) * 100);

        //updating x1 for next iteration
        x1 = x2;

        //printing iteration no and approximate error
        printf ("%d \t\t %.15f% \n\n" , i, approxerror);
    }
```

```c
    //displaying answer
    printf ("Root by Newton Raphson Method is %.15f\n", x2);

    //runtime calculation and display
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC;
    printf("Time taken = %.15f\n", time_taken);

    return 0;
}
```

Listing 4: Code snippet used in Q2(b).

2.6 Contributions

Individual submission. All items included in this subsection were done by me.

3 Problem 3

The concentration of pollutant bacteria (c units) in a lake decreases with time (t days) according to

$$c = 75e^{-1.5t} + 20e^{-0.075t} \tag{4}$$

Determine the time required in days, for the bacteria concentration to be reduced to 15 units using the Newton-Raphson method with an initial guess of $t = 6$ and a stopping criterion of 0.5%. Check your result.

3.1 Approach

In this problem, we use a `while(true)` loop to iterate over Newton Raphson algorithm (described in subsection 3.2). In each iteration, we calculate the new root approximation ($t2$), approximate error, and check whether our exit clause,

$$\text{approximate error} < 0.5\%$$

is satisfied or not. Whenever it is satisfied, we print out the current value of $t2$ as the number of days required.

Additionally, this program also prints its runtime.

3.2 Algorithm

The pseudocode for Problem 3 is provided in Algorithm 5.

Algorithm 5: Finding days required for concentration of bacteria to become 15 units (using Newton-Raphson method)

1. Include necessary header files and define constant $e = 2.718281828459045$
2. Initialize time t to clock time at beginning of *main()* for runtime calculation
3. Declare $ct1, dert1, t2, approxerror$
4. Initialize $t1 = 6.0$ as our initial guess for root
5. Calculate $ct1 = 75e^{-1.5t1} + 20e^{-0.075t1}$ and $dert1 = -1.5 \times (75e^{-1.5t1} + e^{-0.075t1})$
6. Calculate $t2 = t1 - \frac{ct1}{dert1}$
7. Calculate $approxerror = |\frac{t2-t1}{t2} \times 100|$
8. Update $t1 = t2$ for next iteration
9. If $approxerror < 0.5\%$, then go to step 10, else return to step 5
10. Declare "Days Required" as $t2$. Calculate and display runtime ($time\_taken$).
11. Stop

3.3 Results

The plot of the concentration of pollutant bacteria (c) v/s time (in days) is as shown in Figure 6.

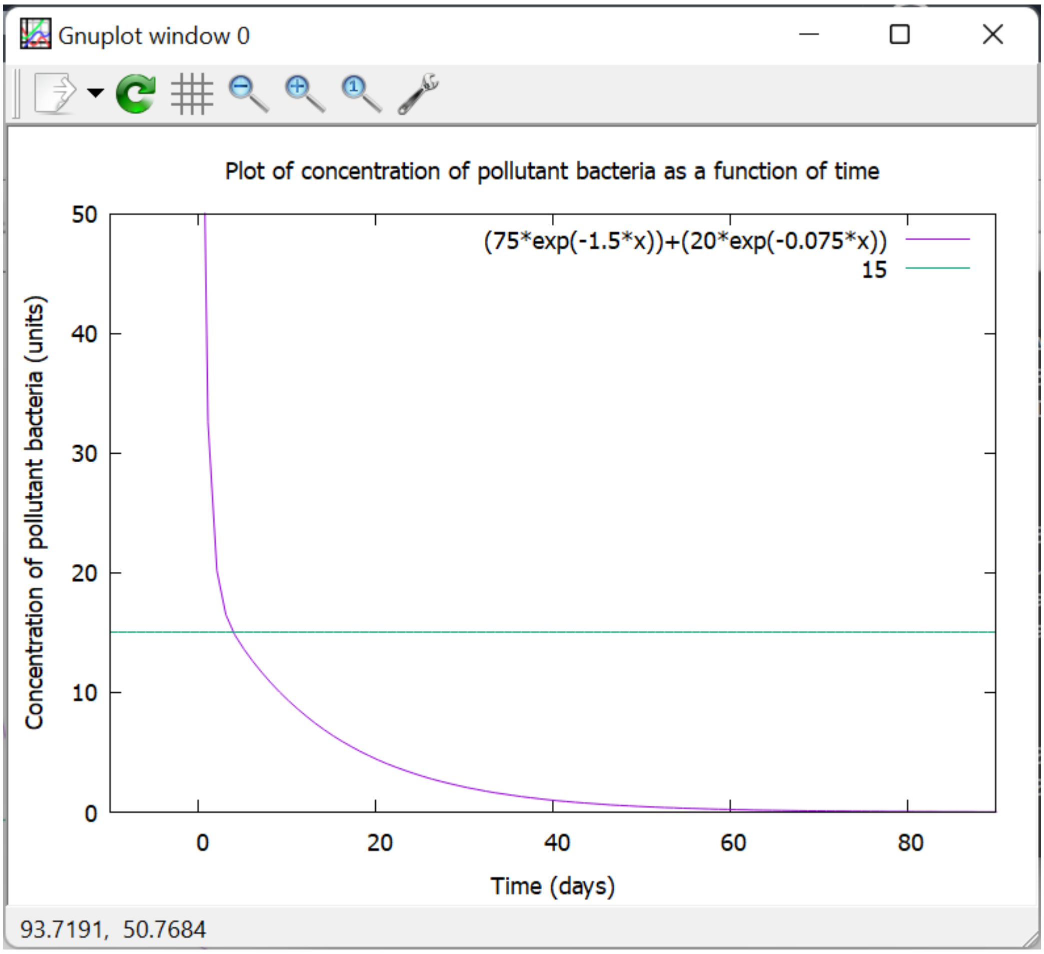


Figure 6: Concentration of bacteria v/s time (in days) using Newton-Raphson method with initial guess as t=6

3.4 Inferences

We deduce the following inferences from Problem 3:

- The result of this program is *"Days required for the bacteria concentration to be reduced to 15 units using Newton-Raphson method is 4.001563072204590"* and this result has an approximate error of 0.491480708122253%
- The average runtime of this program is *0.008000000000000* seconds
- This result can be confirmed by using the graphical approach. The intersection of the green line ($c = 15$) with the purple graph (*c as a function of time*) in Figure 6 is our required answer, which comes out to be the same as the value calculated by our program.

3.5 Code

The code used for Problem 3 is mentioned in Listing 5.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
```

```c

#define e 2.718281828459045

int main (void)
{
    //initializing time at beginning of program for runtime calculation
    ↪ later
    clock_t t;
    t = clock();

    float t1 = 6.0f, ct1, dert1, t2, approxerror;

    //Loop for Newton Raphson method iterations.
    while (1 > 0)
    {
        ct1 = 75 * pow(e, (-1.5 * t1)) + 20 * pow(e, (-0.075 * t1)) - 15;
        dert1 = -1.5 *(75 * pow(e, (-1.5 * t1)) + pow(e, (-0.075 * t1)));

        //finding new root approximation t2 (which is where tangent at
        ↪ (t1,c(t1)) cuts x axis)
        t2 = t1 - (ct1/dert1);

        //calculating approximate error % in this iteration
        approxerror = fabs(((t2 - t1)/t2) * 100);

        //variable updation for next iteration
        t1 = t2;

        //Checking program termination condition of approxerror < 0.5% and
        ↪ terminating program when it is satisfied
        if (approxerror < 0.5)
        {
            printf ("Days required for the bacteria concentration to be
            ↪ reduced to 15 units using Newton-Raphson method is %.15f\n"
            ↪ , t2);
            //runtime calculation and display
            t = clock() - t;
            double time_taken = ((double)t)/CLOCKS_PER_SEC;
            printf("Time taken = %.15f\n", time_taken);
            return 0;
        }
    }
}
```

Listing 5: Code snippet used in Problem 3.

3.6 Contributions

Individual submission. All items included in this subsection were done by me.

4 Problem 4

Figure 7 shows a circuit with a resistor, an inductor, and a capacitor in parallel. Kirchhoff's rules can be used to express the impedance of the system as

$$\frac{1}{Z} = \sqrt{\frac{1}{R^2} + \left(\omega C - \frac{1}{\omega L}\right)^2} \tag{5}$$

where Z = impedance (Ω) and ω = the angular frequency. Find the ω that results in an impedance of 75 Ω using both bisection and false position with initial guesses of 1 and 1000 for the following parameters: R = 225 Ω , $C = 0.6 \times 10^{-6}$ F, and L = 0.5 H. Determine how many iterations of each technique are necessary to determine the answer to $\epsilon_s = 0.1\%$. Use the graphical approach to explain any difficulties that arise.

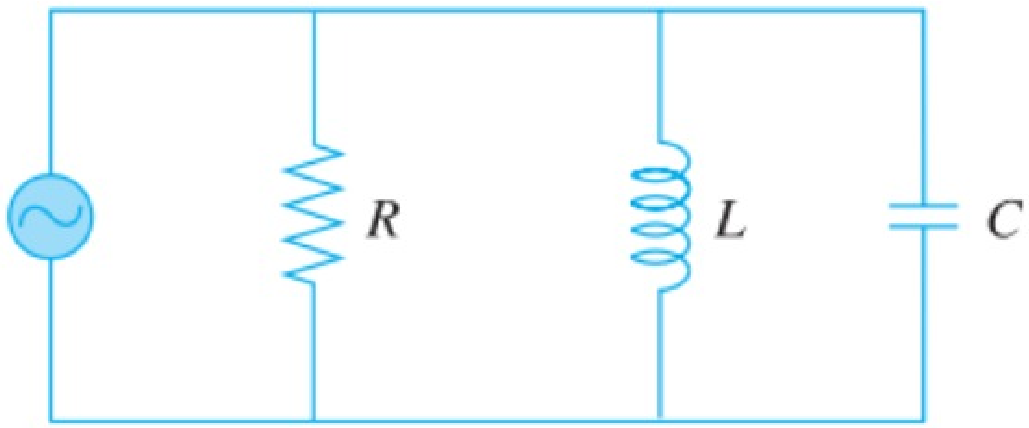


Figure 7: Resistor, Inductor and Capacitor in parallel.

4.1 Approach

For finding the solution to Problem 4 using Bisection method, we use a `while (true)` loop to iterate over the Bisection algorithm (described in subsection 6) until our exit condition of *approxerror* < *0.1* is satisfied.

In each iteration, we calculate new root approximation as mid-point of upper and lower bounds (wu and wl respectively) , calculate the approximate error, check it against the exit condition and exit the loop if satisfied, else update bounds and $wrold$ appropriately and go to next iteration.

For finding the solution to Problem 4 using False Position method, we use a `while (true)` loop to iterate over the False Position algorithm (described in subsection 7) until our exit condition of *approxerror* < *0.1* is satisfied.

In each iteration, we calculate new root approximation as point where chord connecting *(wu,f(wu)) and (wl, f(wl))* cuts x-axis, calculate approximate error %, check exit condition (whether *approxerror* < *0.1*) and declare answer if it is satisfied, else update bounds and $wfold$ appropriately and go to next iteration.

Additionally, both these programs also tabulate iteration no, approximate root, approximate error% and display their runtimes.

4.2 Algorithm

The pseudocode for Bisection Method of Problem 4 is provided in Algorithm 6.

Algorithm 6: Bisection Method for Problem 4

1. Include necessary header files
2. Define constants $R = 225, L = 0.5, C = 0.6 \times 10^{-6}$
3. Create function *float func (float x)* which calculates and returns value of

$$f(x) = \frac{1}{\sqrt{\frac{1}{R^2} + \left(\omega C - \frac{1}{\omega L}\right)^2}} - 75 \tag{6}$$

   Include its prototype right after defining constants.

4. Initialize time t to clock time at beginning of *main(void)* for runtime calculation
5. Initialize $wl = 1.0$ and $wu = 1000.0$ as our initial guesses for lower and upper bounds respectively
6. Declare $wr = 0, fwl = 0, fwr = 0, approxerror, wrold = 0$
7. Initialize *int i = 0* as our counter variable
8. Calculate $wr = \frac{wl+wu}{2}$
9. Calculate $approxerror = |\frac{wr-wrold}{wr} \times 100|$
10. Increment value of i by 1 to store current iteration no
11. Display Iteration no (i), Approximate Root (wr) and Approximate Error% (approxerror)
12. If *approxerror < 0.1*, go to step 16, else go to step 13
13. Calculate fwl and fwr ($f(x)$ values at xl and xr respectively) by calling *func (float x)*
14. If $fwl \times fwr < 0$, then set $wu = wr$ else if $fwl \times fwr > 0$ set $wl = wr$
15. Update $wrold = wr$ for next iteration, then go back to step 8
16. Display no of iterations required as i
17. Calculate and display runtime
18. Stop

The pseudocode for False Position method solution of Problem 4 is provided in Algorithm 7.

Algorithm 7: False Position Method for Problem 4

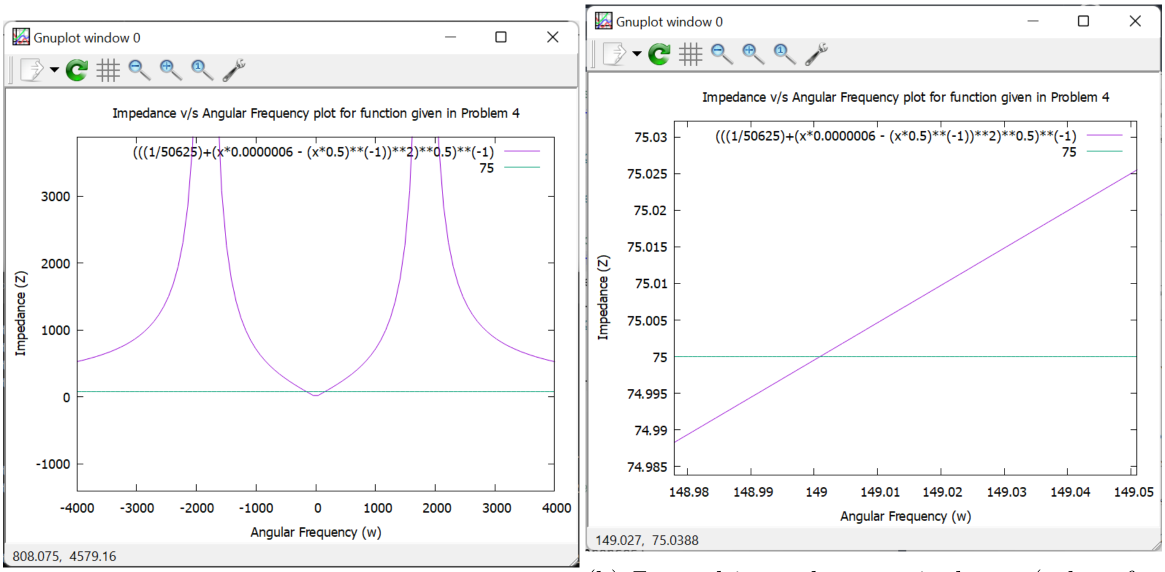
1. Include necessary header files
2. Define constants $R = 225, L = 0.5, C = 0.6 \times 10^{-6}$
3. Create function $float\ func(float\ x)$ to calculate the value of

$$f(x) = \frac{1}{\sqrt{\frac{1}{R^2} + \left(\omega C - \frac{1}{\omega L}\right)^2}} - 75 \tag{7}$$

   when called. Include its function prototype at beginning of program, right after defining constants.
4. Initialize time t to clock time at beginning of *main()* for runtime calculation
5. Initialize $wl = 1.0$ and $wu = 1000.0$ as our initial guesses for lower and upper bounds respectively
6. Declare *fwu, fwl, wf, fwf, approxerror, wfold=0.0*
7. Initialize *int i = 0* as counter variable
8. Calculate $fwu = func(wu)$ and $fwl = func(wl)$
9. Calculate $wf = \frac{fwu \times wl - fwl \times wu}{fwu - fwl}$, $fwf = func(wf)$
10. Calculate $approxerror\% = |\frac{wf - wfold}{wf} \times 100|$
11. Update counter i to current iteration no. by incrementing it by 1
12. Display Iteration No (i), approximate value of root (wf) and approximate error% $(approxerror)$
13. If *approxerror < 0.1*, go to step 16, else go to step 14
14. If $fwf \times fwl < 0$, then set wu = wf else if $fwf \times fwl > 0$ set wl = wf
15. Update $wfold = wf$ for next iteration, then go back to step 8
16. Declare No of Iterations required as i
17. Calculate and display runtime
18. Stop

4.3 Results

We plot the graphs showing the Impedance(Z) as a function of the angular frequency(ω), as shown in by the purple graph in Figure 8. The green line represents $Z = 75$. The intersection of Green and Purple graphs gives us our required value of Angular Frequency ω



(a) Plot of Z vs ω

(b) Zoomed in to show required root (value of ω such that Z = 75)

Figure 8: Graphs of Impedance (Z) vs Angular Frequency (purple) and Z = 75 (green)

4.4 Inferences

We deduce the following inferences from Problem 4:

- If we use Bisection method, we need *13 iterations* to get our answer as $\omega = 157.947388$ with an approximate error of 0.077208%. The average runtime of this program is 0.008000000000000 seconds.

- However, if we use False Position method, we need *6 iterations* to get our answer as $\omega = 157.911560$ with an approximate error of 0.015750%. The average runtime of this program is 0.004000000000000 seconds.

- We can clearly observe that the False Position Method is extremely efficient here, since it takes less than half the no of iterations of Bisection method to reach our desired predetermined error. It also has half the average runtime of Bisection method, and much lesser approximate error.

- The reason for this can be explained using the graphical approach.

- As seen in Figure 8, the shape of the graph of "Z vs ω" (shown in purple) is such that it allows for rapid progress in root approximation(towards "true root" i.e. value of ω for which Z = 75) in False Position Method, but in Bisection method, progression towards true root in each successive approximation is not this big. Thus, False Position method is preferred for this particular problem.

- In this problem, we have a choice of considering *f(x)* as either

$$f(x) = \frac{1}{\sqrt{\frac{1}{R^2} + \left(\omega C - \frac{1}{\omega L}\right)^2}} - 75 \tag{8}$$

$$or \tag{9}$$

$$f(x) = \sqrt{\frac{1}{R^2} + \left(\omega C - \frac{1}{\omega L}\right)^2} - \frac{1}{75} \tag{10}$$

This choice doesn't make any difference in Bisection Method, but in False Position method, it makes a huge difference.

- Choosing the first option, as I have done, gives 6 iterations and 0.004s average time.
- However, choosing the second option gives us 578 iterations with 0.117000000000000s runtime.
- This huge difference can once again be explained by the graphical approach. The Z v/s ω graph in needed for the first option, which is highly optimised for False Position method.
- The second option, however, requires the $\frac{1}{Z}$ v/s ω graph, which is extremely unoptimised for False Position method.
- Thus, the optimal solution for this Problem would be choosing the first option for $f(x)$ and applying False Position.

4.5 Code

The code used for Problem 4 is mentioned in Listing 6 (Bisection Method) and Listing 7 (False Position Method).

```
#include <stdio.h>
#include <math.h>
#include <time.h>

//define constants for calculations
#define R 225
#define L 0.5
#define C 0.0000006

//function prototype
float func (float x);

int main (void)
{
    //initializing time at beginning of program for runtime calculation
    ↪ later
    clock_t t;
    t = clock();

    float wl = 1.0f , wu = 1000.0f , wr = 0 , fwl = 0 , fwr = 0,
    ↪ approxerror, wrold = 0;

    //initialize int i as counter variable to count no of iterations of
    ↪ loop
    int i = 0;

```

```c
    printf ("Iteration No. \t Approximate Value \t Approximate Error \n\n");

    //while(true) loop to compute iterations of Bisection method. Control exits loop and terminates program when exit condition "approxerror < 0.1%" is satisfied
    while (1 > 0)
    {
        //calculating new root approximation as mid point of upper and lower bounds
        wr = (wl + wu)/2;

        //calculating approximate error % in current iteration
        approxerror = fabs(((wr - wrold)/wr) * 100);

        //increment loop variable to current iteration no
        i++;

        //display Iteration No, Approx value and Approx Error of current iteration
        printf ("%d \t\t %f \t\t %f%\n\n" , i, wr, approxerror);

        //Exit clause: if approxerror < 0.1% then display "i" as required no of iterations, calculate and display runtime of program, then stop
        if (approxerror < 0.1)
        {
            printf ("Number of iterations required (using Bisection Method) is %d\n", i);

            //runtime calculation and display
            t = clock() - t;
            double time_taken = ((double)t)/CLOCKS_PER_SEC;
            printf("Time taken = %.15f\n", time_taken);

            return 0;
        }

        //if exit condition not met, then update lower/upper bound accordingly after checking condition "fwl * fwr < 0"
        fwl = func(wl);
        fwr = func(wr);
        if ((fwl * fwr) < 0)
            wu = wr;
        else if ((fwl * fwr) > 0)
            wl = wr;

        //update wrold for next iteration
        wrold = wr;
    }
```

```
}

//function for calculating and returning the value of f(x) when called
float func (float x)
{
    float ans = pow((pow((pow(R, -2) + pow ((x*C - pow(x*L, -1)), 2)), 0.5)),-1)- (75.0);
    return (ans);
}
```

Listing 6: Code snippet used in Bisection method of Problem 4.

```
#include <stdio.h>
#include <math.h>
#include <time.h>

//define constants for calculations
#define R 225
#define L 0.5
#define C 0.0000006

//function prototype
float func (float x);

int main (void)
{
    //initializing time at beginning of program for runtime calculation later
    clock_t t;
    t = clock();

    float wl = 1.0f, wu = 1000.0f, fwu, fwl, wf, fwf, approxerror, wfold = 0.0f;

    printf ("Iteration No. \t Approximate Value \t Approximate Error \n");

    //initialize counter variable "i" to count iteration no of loop
    int i = 0;

    //here, we use a while (true) loop to execute iterations of False Position method. This loop keeps running until exit clause of "approxerror < 0.1%" is satisfied, at which point it displays result and terminates the program
    while (1 > 0)
    {
        //calling function "func" to calculate values of f(wu) and f(wl)
        fwu = func(wu);
        fwl = func(wl);

```

```
        //calculating new root approximation as point where chord connecting (wu,f(wu)) and (wl, f(wl)) cuts x-axis
        wf = (fwu * wl - fwl * wu)/(fwu - fwl);

        //again, calling function "func" to calculate value of f(wf)
        fwf = func(wf);

        //calculating approximate error %
        approxerror = fabs(((wf - wfold)/wf) * 100);

        //updating counter to current iteration no.
        i++;

        //printing Iteration No, Approximate Value and Approximate Error (in table format)
        printf ("%d \t\t %f \t\t %f%\n\n" , i, wf, approxerror);

        //checking whether exit clause of (approxerror < 0.1%) is satisfied or not
         if (approxerror < 0.1)
        {
           //if exit clause satisfied, then display number of iterations required as "i"
           printf ("Number of iterations needed for getting an impedence of 75 (using False Position Method) is %d\n" , i);

           //runtime calculation and display
           t = clock() - t;
           double time_taken = ((double)t)/CLOCKS_PER_SEC;
           printf("Time taken = %.15f\n", time_taken);

          return 0;
        }

        //if exit clause not satisfied, then update bounds accordingly (after checking condition "fwf * fwl < 0") for next iteration
         if (fwf * fwl < 0)
            wu = wf;
         else if (fwf * fwl > 0)
            wl = wf;

        //update wfold to current wf value for next iteration
        wfold = wf;
    }
}

//function which, when called, calculates and returns value of f(x)
float func (float x)
{
```

```
    float ans = pow((pow((pow(R, -2) + pow ((x*C - pow(x*L, -1)), 2)),
    ↪  0.5)),-1)- (75.0);
    return (ans);
}
```

Listing 7: Code snippet used in False Position method of Problem 4.

4.6 Contributions

Individual submission. All items included in this subsection were done by me.