# ENS409 Project 1 Report

## Question 1

Find the solution for $4x^3 - 8x^2 + 3x - 10 = 0$ on [1, 3] with an absolute approximate error < $10^{-4}$. You should select appropriate initial approximate solution for open methods

## Solution

We are now given the function and number of iterations to be used to find the root by methods and then calculate the relative error in accordance with the root given by the function fzero. Hence, I am now going to iterate 6 times in the methods then find the relative error.

Absolute Approximate error can be founded as | Present Approx. – Previous Approx. |

Hence, let me start with one of the bracketing methods, Bisection Method

### *Bisection Method*

The bisection method is a root-finding method that can be applied to any continuous function for which two values with opposite signs are known. The technique involves repeatedly bisecting the interval defined by these values and then selecting the subinterval in which the function changes sign, indicating that it must contain a root. Thus, we can state it uses Intermediate Value Theorem (IVT) to show that there is a root in an interval of a continuous function.

I am calculating absolute approximate error by storing the p values. Then there is a counter I such that if I is bigger than 1 I start to calculate the error.

```
function [root, finalerror, iteration] = BisectionMethod(lowerbound,
upperbound, maxiter, tol)
    tolerance = 1.0*10^-tol; % given tolerance
    myfunc = @(x)4*x^3 - 8*x^2 + 3*x - 10; %first function
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    error = 1000;
    i = 0;
    value(1) = 0;
    p = -1;
    while(error > tolerance && count < maxiter)
```

```
        i = i + 1;
        FA = myfunc(bound(1));
        FB = myfunc(bound(2));
        p = (bound(1) + bound(2)) /2;
        FP = myfunc(p);
        value(i) = p;
        if(FA*FP > 0)
           bound(1) = p;
        elseif (FA*FP < 0)
           bound(2) = p;
        else
           break;
        end
        if(i >1)
           error = abs(value(i) - value(i-1));
        end
        count = count + 1;
    end
    root = p;
    finalerror = error;
    iteration = count;
end
```

I am getting inputs from the user for lower bound, upper bound, number of iterations and tolerance. Then, I am printing the founded root, absolute approximate error and iteration number.

```
%% Bisection Method
fprintf("Welcome to the Bisection Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the tolerance 1.0 x e-n:');
[root, error, iteration] = BisectionMethod(lowerbound, upperbound,
maxiter, tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The absolute approximate error is %.6f', error);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')
```

Here, you will see a demonstration of the bisection method function.

```
Welcome to the Bisection Method function
Please enter the lower bound:1
Please enter the upper bound:3
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18157959
The absolute approximate error is 0.000061
The iteration number is 15
```

As you can see, we find the root with a tolerance which is less than the specified one.

### *False Position Method*

The algorithm is like the Bisection method. However, if you review the q2_BisectionMethod function in the below, the difference comes in the step of calculating the new p (root) value. Now, you are not using the mean of lower bound and upper bound, you are using the weighted average to find the new p.

Here, I am assigning tolerance and writing the function which I am going to use in the method. Then, I am calculating p and FP for progressing with the method. Then, I am calculating absolute approximate error by subtracting previous approximation from the current approximation in absolute brackets.

```
function [root, finalerror, iteration] =
FalsePositionMethod(lowerbound, upperbound, maxiter, tol)
    tolerance = 1.0*10^-tol; % given tolerance
    myfunc = @(x)4*x^3 - 8*x^2 + 3*x - 10; %first function
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    error = 1000;
    i = 0;
    value(1) = 0;
    p = -1;
    while(error >= tolerance && count < maxiter)
        i = i + 1;
        FA = myfunc(bound(1));
        FB = myfunc(bound(2));
        p = bound(1) + ((FA*(bound(1)- bound(2)))/(FB-FA));
        FP = myfunc(p);
        value(i) = p;
        if(FA*FP > 0)
           bound(1) = p;
        elseif(FA*FP < 0)
           bound(2) = p;
        else
           break;
        end
        if(i >1)
           error = abs(value(i) - value(i-1));
```

```
        end
        count = count + 1;
    end
    root = p;
    finalerror = error;
    iteration = count;
end
```

I am getting inputs from the user for lower bound, upper bound, number of iterations and tolerance. Then, I am printing the founded root, absolute approximate error and iteration number.

```
%% False Position Method
fprintf("Welcome to the False Position Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the tolerance 1.0 x e-n:');
[root, error, iteration] = FalsePositionMethod(lowerbound, upperbound,
maxiter, tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The absolute approximate error is %.8f', error);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')
```

Here, you can see the result of the method

```
Welcome to the False Position Method function
Please enter the lower bound:1
Please enter the upper bound:3
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18145950
The absolute approximate error is 0.00008705
The iteration number is 12
```

### *Modified False Position Method*

It is generated from modified false position method. Hence, it is quite similar. However, there are some to find the root faster if possible. I am now using two counters one of the for lower bound and the other one is for upper bound. The reason we are using two additional counters to detect a stuck bound in the method. It differentiates at this point from the false position

method because we were not checking the bounds for detecting a stuck bound in the false position method.

Here, as you can see I have additional two counters in the function. I decided to use the approach that Mustafa Hocam talked about it in the class. If a counter value bigger or equal to 2. I am dividing the function's value at that boundary

```
function [root, finalerror, iteration] =
ModifiedFalsePositionMethod(lowerbound, upperbound, maxiter, tol)
    tolerance = 1.0*10^-tol; % given tolerance
    myfunc = @(x)4*x^3 - 8*x^2 + 3*x - 10; %first function
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    leftcount = 0;
    rightcount = 0;
    error = 1000;
    i = 0;
    value(1) = 0;
    p = -1;
    FA = myfunc(bound(1));
    FB = myfunc(bound(2));
    while(error > tolerance && count < maxiter)
        i = i +1;
        %p = (abs(FB)*bound(1) + abs(FA)*bound(2)) / (abs(FA) +
abs(FB));
        p = bound(1) + ((FA*(bound(1)- bound(2)))/(FB-FA));
        FP = myfunc(p);
        value(i) = p;
        if(FA*FP > 0)
           leftcount = 0;
           rightcount = rightcount + 1;
           bound(1) = p;
           FA = myfunc(bound(1));
           if(rightcount >= 2)
               FB = FB/2;
           end
        elseif(FA*FP < 0)
           rightcount = 0;
           leftcount = leftcount + 1;
           bound(2) = p;
           FB = myfunc(bound(2));
           if(leftcount >= 2)
               FA = FA/2;
           end
        else
            break;
        end
        if(i >1)
           error = abs(value(i) - value(i-1));
        end
        count = count + 1;
    end
    root = p;
    finalerror = error;
    iteration = count;
end
```

I am getting inputs from the user for lower bound, upper bound, number of iterations and tolerance. Then, I am printing the founded root, absolute approximate error and iteration number.

```
%% Modified False Position Method
fprintf("Welcome to the Modified False Position Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the tolerance 1.0 x e-n:');
[root, error, iteration] = ModifiedFalsePositionMethod(lowerbound,
upperbound, maxiter, tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The absolute approximate error is %.12f', error);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')
%%%%% BRACKETING METHODS %%%%%
```

Here, you can see the result of the method

```
Welcome to the Modified False Position Method function
Please enter the lower bound:1
Please enter the upper bound:3
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18153442
The absolute approximate error is 0.000029246340
The iteration number is 7
```

## *Newton Method*

Newton's method is a root-finding algorithm that produces successively better approximations to a real-valued function's roots. The method begins with a single-variable function f defined for a real variable x, its derivative f', and an initial guess x0 for a root of the function. We found the roots by the following equation.

$$p_{n+1} = p_n - \frac{f(p_n)}{f'(p_n)}$$

In the function below, we now have another function which is for the derivative. Then, basically, I am the using the formula for finding roots in Newton Method. Furthermore, I am calculating absolute approximate error by using the current and previous approximation values.

```
function [root, finalerror, iteration] = NewtonMethod(initial,
maxiter, tol)
    tolerance = 1.0*10^-tol; % given tolerance
    myfunc = @(x)4*x^3 - 8*x^2 + 3*x - 10; %first function
    derivative = @(x)12*x^2 - 16*x + 3; % derivative of the function
    bound(1) = initial; % boundaries
    count = 0;
    error = 1000;
    i = 2;
    while(error > tolerance && count < maxiter)
        bound(i) = bound(i-1) - (myfunc(bound(i-
1))/derivative(bound(i-1)));
        count = count + 1;
        error = abs(bound(i)- bound(i-1));
        i = i + 1;
    end
    root = bound(i-1);
    finalerror = error;
    iteration = count;
end
```

Here, I am taking the inputs for the function which are initial value, maximum iteration number and tolerance. Then, I am printing the values that came from the function

```
%% Newton Method
fprintf("Welcome to the Newton Method function\n");
p0 = input('Please enter the initial value:');

maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the tolerance 1.0 x e-n:');

[root, error, iteration] = NewtonMethod(p0, maxiter, tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The absolute error is %.8f', error);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')
```

Here, you can see the results of the Newton Method

```
Welcome to the Newton Method function
Please enter the initial value:3
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18152015
The absolute approximate error is 0.00000084
The iteration number is 5
Welcome to the Newton Method function
Please enter the initial value:1
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18152015
The absolute approximate error is 0.00000409
The iteration number is 23
Welcome to the Newton Method function
Please enter the initial value:2
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18152015
The absolute approximate error is 0.00000025
The iteration number is 4
>>
```

Here as you can see with different initial value we get the correct approximation bound with the given tolerance in different iteration scores for each value. The worst performance came with the initial value of 1.

### *Secant Method*

The secant method is a root-finding algorithm that uses a succession of the secant lines can be drawn from its approximation points to better approximate the root of a function f. The following function can be used to find the next approximate root.

$$p_{n+1} = p_n + \frac{f(p_n)p_{n-1} - f(p_n)p_n}{f(p_n) - f(p_{n-1})}$$

Furthermore, I am calculating absolute approximate error by using the current and previous approximation values.

```
function [root, finalerror, iteration] = SecantMethod(initial,
second, maxiter, tol)
    tolerance = 1.0*10^-tol; % given tolerance
    myfunc = @(x)4*x^3 - 8*x^2 + 3*x - 10; %first function
    bound = [initial second]; % boundaries
    count = 0;
    error = 1000;
    i = 3;
    while(error > tolerance && count < maxiter)
        bound(i) = bound(i-1) + (myfunc(bound(i-1))*(bound(i-2)-
bound(i-1)))/ (myfunc(bound(i-1))- myfunc(bound(i-2)));
        count = count + 1;
        error = abs(bound(i) - bound(i-1));
        i = i +1;
    end
    finalerror = error;
    root = bound(i-1);
    iteration = count;
end
```

Here, I am taking the inputs for the function which are initial approximation, second approximation, maximum iteration number and tolerance. Then, I am printing the values that came from the function

```
%% Secant Method
fprintf("Welcome to the Secand Method function\n");
p0 = input('Please enter the p0:');
p1 = input('Please enter the p1:');
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the tolerance 1.0 x e-n:');
[root, error, iteration] = SecantMethod(p0, p1, maxiter, tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The absolute approximate error is %.8f', error);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')
```

The results are as follows

```
Welcome to the Secand Method function
Please enter the p0:1
Please enter the p1:3
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18152012
The absolute approximate error is 0.00002761
The iteration number is 8
Welcome to the Secand Method function
Please enter the p0:2
Please enter the p1:3
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18152008
The absolute approximate error is 0.00004876
The iteration number is 5
```

As you can see with different initial values, we came up with different roots as well as number of iterations differ.

### *Fixed Point Iteration*

As we have seen in the class, root-finding problem can be converted into a fixed-point finding problem. Now we are going to define a function with a fixed point at x as where

$$g(x) = x - f(x)$$

I am taking inputs for the initial value and number of iterations. Then, I define function g function as x – function f. I am using the following formula to find the root.

$$x_{i+1} = g(x_i)$$

Furthermore, I am calculating absolute approximate error by using the current and previous approximation values.

```
function [root, finalerror, iteration] = FixedPointIteration(initial,
maxiter, tol)
    tolerance = 1.0*10^-tol; % given tolerance
    func_g = @(x)(10 + 8*x^2)/(4*x^2 + 3); %I converted this from the main
function
    myfunc = @(x)4*x^3 - 8*x^2 + 3*x - 10;
    value(1) = initial;
    count = 0;
    error = 1000;
```

```
    i = 2;
    while(error > tolerance && count < maxiter)
        value(i) = func_g(value(i-1));
        count = count + 1;
        error = abs(value(i) - value(i-1));
        i = i + 1;
    end
    root = value(i-1);
    finalerror = error;
    iteration = count;
end
```

Here, I am taking the inputs for the function which are initial value, maximum iteration number and tolerance. Then, I am printing the values that came from the function

```
%% Fixed Point Iteration
fprintf("Welcome to the Fixed Point Iteration function\n");
p0 = input('Please enter the initial value:');
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the tolerance 1.0 x e-n:');
[root, error, iteration] = FixedPointIteration(p0, maxiter, tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The absolute approximate error is %.8f', error);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')
```

The results are as follows

```
Welcome to the Fixed Point Iteration function
Please enter the initial value:2
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18153237
The absolute approximate error is 0.00009717
The iteration number is 5
Welcome to the Fixed Point Iteration function
Please enter the initial value:1
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18152302
The absolute approximate error is 0.00002283
The iteration number is 7
Welcome to the Fixed Point Iteration function
Please enter the initial value:3
Please enter the maximum iteration number:100
Please enter the tolerance 1.0 x e-n:4
The root is 2.18152520
The absolute approximate error is 0.00004012
The iteration number is 6
>>
```

As you can see with different initial values we get different roots and number of iteration differ. However, each error is less than tolerance.

## Question 2

Compute the root of the equation

$$x^2 - x - e^{-x} = 0$$

in the interval [1, 2]. For each method, perform six iterations and compute the relative errors with respect to the true solution obtained using the built-in fzero function in MATLAB.

## Solution

We are now given the function and number of iterations to be used to find the root by methods and then calculate the relative error in accordance with the root given by the function fzero. Hence, I am now going to iterate 6 times in the methods then find the relative error.

Relative error can be founded as follows. | True Error | / | True Value |
True error can be founded as | True Value – Approximate Value |

Hence, let me start with one of the bracketing methods, Bisection Method

### *Bisection Method*

In the function, I have given 3 parameters as input which are lower bound, upper bound and number of iterations. For the given question number of iterations is 6 and lower bound 1 while upper bound is 2. I have 3 outputs that I am going to use in the main script that I have prepared, *root* stands the value that I have found using by Bisection method. *finalerror* is the relative error between the actual root calculated and given by the built-in function fzero in matlab. Since the number of iterations is given I am not using the a tolerance range in this question which is different from the first question.

```
function [root, finalerror, iteration] =
q2_BisectionMethod(lowerbound, upperbound, maxiter)
    myfunc = @(x)x^2 - x - exp(-x); %first function
    bound = [lowerbound upperbound]; % boundaries
```

```
    count = 0; %iteration count

    p = -1;
    score = fzero(myfunc, bound);

    while(count < maxiter)
        FA = myfunc(bound(1));
        FB = myfunc(bound(2));
        p = (bound(1) + bound(2)) /2;
        FP = myfunc(p);
        if(FA*FP > 0)
           bound(1) = p;
        elseif(FA*FP < 0)
           bound(2) = p;
        else
            break;
        end
        count = count + 1;
    end
    root = p;
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

In the q2_main.m, I am getting inputs from the user for lower bound, upper bound and number of iterations. Then, I am printing the founded root, fzero root and the relative error.

```
%% Bisection Method
fprintf("Welcome to the Bisection Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
[root, error, iteration] = q2_BisectionMethod(lowerbound,
upperbound, maxiter);
fprintf('The founded root is %.6f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.6f', iteration);
fprintf('\n')
fprintf('The relative error is %.6f', error);
fprintf('\n')
```

Here, you will see a demonstration of the bisection method function.

```
Welcome to the Bisection Method function
Please enter the lower bound:1
Please enter the upper bound:2
Please enter the maximum iteration number:6
The founded root is 1.234375
The root given by fzero function is 1.235346
The relative error is 0.000786
>>
```

### *False Position Method*

I've given three parameters as input to the function: lower bound, upper bound, and number of iterations. The number of iterations for the given question is 6, with a lower bound of 1 and an upper bound of 2. I have three outputs that I plan to use in the main script that I've written, root represents the value that I discovered using the False Position method. finalerror is the relative error between the actual root calculated and the value returned by matlab's built-in function fzero.

```
function [root, finalerror, iteration] = q2_FalsePositionMethod(lowerbound,
upperbound, maxiter)

    myfunc = @(x)x^2 - x - exp(-x); %first function
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    score = fzero(myfunc, bound);
    p = -1;
    while(count < maxiter)
        FA = myfunc(bound(1));
        FB = myfunc(bound(2));
        p = bound(1) + ((FA*(bound(1)- bound(2)))/(FB-FA));
        FP = myfunc(p);
        if(FA*FP > 0)
           bound(1) = p;
        elseif(FA*FP < 0)
           bound(2) = p;
        else
            break;
        end
        count = count + 1;
    end
    root = p;
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

The method's section in the main script is very similar to the Bisection Method.

I'm getting user input for the lower bound, upper bound, and number of iterations. Then I print the founded root, the fzero root, and the relative error.

```
%% False Position Method
fprintf("Welcome to the False Position Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
[root, error, iteration] = q2_FalsePositionMethod(lowerbound,
upperbound, maxiter);
fprintf('The founded root is %.6f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.6f', iteration);
fprintf('\n')
fprintf('The relative error is %.6f', error);
fprintf('\n')
```

Here, you will see a demonstration of the False Position method function.

```
Welcome to the False Position Method function
Please enter the lower bound:1
Please enter the upper bound:2
Please enter the maximum iteration number:6
The founded root is 1.235226
The root given by fzero function is 1.235346
The relative error is 0.000097
```

### *Modified False Position Method*

Here, as you can see I have additional two counters in the function. I decided to use the approach that Mustafa Hocam talked about it in the class. If a counter value bigger or equal to 2. I am dividing the function's value at that boundary. If counter is reset, then I am actually calculating the new bound's function value for specific side if occurs.

```
function [root, finalerror, iteration] =
q2_ModifiedFalsePositionMethod(lowerbound, upperbound, maxiter)

    myfunc = @(x)x^2 - x - exp(-x); %first function
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    leftcount = 0;
    rightcount = 0;
```

```
    score = fzero(myfunc, bound);
    p = -1;
    FA = myfunc(bound(1));
    FB = myfunc(bound(2));
    while(count < maxiter)
        %p = (abs(FB)*bound(1) + abs(FA)*bound(2)) / (abs(FA) +
abs(FB));
        p = bound(1) + ((FA*(bound(1)- bound(2)))/(FB-FA));
        FP = myfunc(p);
        if(FA*FP > 0)
           leftcount = 0;
           rightcount = rightcount + 1;
           bound(1) = p;
           FA = myfunc(bound(1));
           if(rightcount >= 2)
               FB = FB/2;
           end
        elseif(FA*FP < 0)
           rightcount = 0;
           leftcount = leftcount + 1;
           bound(2) = p;
           FB = myfunc(bound(2));
           if(leftcount >= 2)
               FA = FA/2;
           end
        else
           break;
        end
        count = count + 1;
    end
    root = p;
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

The method's section in the main script is very similar to the False Position Method.

I'm getting user input for the lower bound, upper bound, and number of iterations. Then I print the founded root, the fzero root, and the relative error.

```
%% Modified False Position Method
fprintf("Welcome to the Modified False Position Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
[root, error, iteration] =
q2_ModifiedFalsePositionMethod(lowerbound, upperbound, maxiter);
fprintf('The founded root is %.6f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.6f', iteration);
fprintf('\n')
```

```
fprintf('The relative error is %.12f', error);
fprintf('\n')
%%%%% BRACKETING METHODS %%%%%
```

Here, you will see a demonstration of the Modified False Position method function.

```
Welcome to the Modified False Position Method function
Please enter the lower bound:1
Please enter the upper bound:2
Please enter the maximum iteration number:6
The founded root is 1.235347
The root given by fzero function is 1.235346
The relative error is 0.000000262535
```

The Bracketing method have finished. Now, I will continue with the Open Methods.

### *Newton Method*

Newton's method is a root-finding algorithm that produces successively better approximations to a real-valued function's roots. The method begins with a single-variable function f defined for a real variable x, its derivative f', and an initial guess x0 for a root of the function. We found the roots by the following equation.

In the function below, we now have another function which is for the derivative. Then, basically, I am the using the formula for finding roots in Newton Method.

```
function [root, finalerror, iteration] = q2_NewtonMethod(initial,
maxiter)
    myfunc = @(x)x^2 - x - exp(-x); %first function
    derivative = @(x)2*x + exp(-x) -1; % derivative of the function
    bound(1) = initial; % boundaries
    count = 0;
    score = fzero(myfunc, [1 2]);
    i = 2;
    while(count < maxiter)
        bound(i) = bound(i-1) - (myfunc(bound(i-
1))/derivative(bound(i-1)));
        count = count + 1;
        i = i + 1;
    end
    root = bound(i-1);
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

Here, I am getting input for initial value and number of iteration. Then I print the founded root, the fzero root, and the relative error.

```
%% Newton Method
fprintf("Welcome to the Newton Method function\n");
p0 = input('Please enter the initial value:');
maxiter = input('Please enter the maximum iteration number:');


[root, error, iteration] = q2_NewtonMethod(p0, maxiter);
fprintf('The founded root is %.6f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.6f', iteration);
fprintf('\n')
fprintf('The relative error is %.12f', error);
fprintf('\n')
```

Here, you will see a demonstration of the Newton Method.

```
Welcome to the Newton Method function
Please enter the initial value:1.5
Please enter the maximum iteration number:6
The founded root is 1.235346
The root given by fzero function is 1.235346
The relative error is 0.000000000000
Welcome to the Newton Method function
Please enter the initial value:2
Please enter the maximum iteration number:6
The founded root is 1.235346
The root given by fzero function is 1.235346
The relative error is 0.000000000000
Welcome to the Newton Method function
Please enter the initial value:1
Please enter the maximum iteration number:6
The founded root is 1.235346
The root given by fzero function is 1.235346
The relative error is 0.000000000000
```

As you can see, the result in the below shows that, this function converges beautifully.

## *Secant Method*

The secant method is a root-finding algorithm that uses a succession of the secant lines can be drawn from its approximation points to better approximate the root of a function f. The following function can be used to find the next approximate root.

$$p_{n+1} = p_n + \frac{f(p_n)p_{n-1} - f(p_n)p_n}{f(p_n) - f(p_{n-1})}$$

Here I am getting inputs as initial and second approximation values and the number of iterations. The outputs are similar to other functions. Founded root, root by fzero and relative error between founded error and fzero's value.

```
function [root, finalerror, iteration] = q2_SecantMethod(initial,
second, maxiter)

    myfunc = @(x)x^2 - x - exp(-x); %first function
    bound = [initial second]; % boundaries
    count = 0;
    score = fzero(myfunc, [1 2]);
    i = 3;
    while(count < maxiter)
        bound(i) = bound(i-1) + (myfunc(bound(i-1))*(bound(i-2)-
bound(i-1)))/ (myfunc(bound(i-1))- myfunc(bound(i-2)));
        count = count + 1;
        i = i +1;
    end
    root = bound(i-1);
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

Here I am getting inputs. Then I print the founded root, the fzero root, and the relative error.

```
%% Secant Method
fprintf("Welcome to the Secand Method function\n");
p0 = input('Please enter the p0:');
p1 = input('Please enter the p1:');
maxiter = input('Please enter the maximum iteration number:');
[root, error, iteration] = q2_SecantMethod(p0, p1, maxiter);
fprintf('The founded root is %.8f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.8f', iteration);
fprintf('\n')
fprintf('The relative error is %.12f', error);
fprintf('\n')
```

Here, you will see a demonstration of the Newton Method.

```
Welcome to the Secand Method function
Please enter the p0:1
Please enter the p1:2
Please enter the maximum iteration number:6
The founded root is 1.23534623
The root given by fzero function is 1.23534623
The relative error is 0.000000000000
Welcome to the Secand Method function
Please enter the p0:1.5
Please enter the p1:1.9
Please enter the maximum iteration number:6
The founded root is 1.23534623
The root given by fzero function is 1.23534623
The relative error is 0.000000000000
```

It also converges beautifully. Further tests can be done to check its convergence behaviour.

### *Fixed Point Iteration*

As we have seen in the class, root-finding problem can be converted into a fixed-point finding problem. Now we are going to define a function with a fixed point at x as where

$$g(x) = x - f(x)$$

I am taking inputs for the initial value and number of iterations. Then, I define function g function as x – function f. I am using the following formula to find the root.

$$x_{i+1} = g(x_i)$$

```
function [root, finalerror, iteration] = q2_FixedPointIteration(initial,
maxiter)
    myfunc = @(x)x^2 - x - exp(-x); %first function
    func_g = @(x)x - x^2 + x + exp(-x); %I converted this from the main
function
    x(1) = initial;
    count = 0;
    score = fzero(myfunc, [1 2]);
    i = 2;
    while(count < maxiter)
        x(i) = func_g(x(i-1));
        count = count + 1;
        i = i + 1;
    end
    root = x(i-1);
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

Here I am getting inputs. Then I print the founded root, the fzero root, and the relative error.

```
%% Fixed Point Iteration
fprintf("Welcome to the Fixed Point Iteration function\n");
p0 = input('Please enter the initial value:');
maxiter = input('Please enter the maximum iteration number:');

[root, error, iteration] = q2_FixedPointIteration(p0, maxiter);
fprintf('The founded root is %.8f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.8f', iteration);
fprintf('\n')
fprintf('The relative error is %.12f', error);
fprintf('\n')

%%%%% OPEN METHODS %%%%%
```

Here, you will see a demonstration of the Newton Method.

```
Welcome to the Fixed Point Iteration function
Please enter the initial value:1
Please enter the maximum iteration number:6
The founded root is 1.19933433
The root given by fzero function is 1.23534623
The relative error is 0.029151266296
Welcome to the Fixed Point Iteration function
Please enter the initial value:2
Please enter the maximum iteration number:6
The founded root is 1.20113807
The root given by fzero function is 1.23534623
The relative error is 0.027691156184
Welcome to the Fixed Point Iteration function
Please enter the initial value:1.5
Please enter the maximum iteration number:6
The founded root is 1.28337905
The root given by fzero function is 1.23534623
The relative error is 0.038882064878
>>
```

Here, as you can see it gets close to the actual root. However, the convergence behavior to the real root is not as good as other open methods if we want to compare them.

## Question 3

Solve the equation $e^{x^3} - 8 = 0$ in the interval [0, 3]. Compare the values after 10, 50 and 100 iterations. Comment on the convergence of the methods.

## Solution

We are now given the function and different number of iterations to be used to find the root by methods and then understand the convergence behaviour of each method for the given function. Hence, I am now going to iterate 10, 50 and 100 times in the methods then evaluate the its convergence behaviour.

### *Bisection Method*

In the function, I have given 3 parameters as input which are lower bound, upper bound and number of iterations. For the given question number of iterations will be 100 since I need to analyze $10^{th}$, $50^{th}$ and $100^{th}$ iterations. Lower bound is 0 while upper bound is 3. I have 6 outputs that I am going to use in the main script that I have prepared, *root* stands the value that I have found using by Bisection method. Fzero is the root given by the built-in function. Itervalue(i) is the root approximate for the given iteration while iterscore it the output with the approximate root of the function. iterabserror is the absolute error between true value of the function which is 0 and function value of the approximate root.

In the function I am calculating the absolute errors of the each given iteration number. Additionally, I am calculating the relative error of the last iteration's root with the real root.

```
function [root, finalerror, iteration, iterscore, itervalue,
iterABSerror] = q3_BisectionMethod(lowerbound, upperbound, maxiter)
    myfunc = @(x)exp(x^3) - 8; %first function
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    a = 1;
    p = -1;
    score = fzero(myfunc, bound);

    while(count < maxiter)
        count = count + 1;
        FA = myfunc(bound(1));
        FB = myfunc(bound(2));
        p = (bound(1) + bound(2)) /2;
        FP = myfunc(p);
        if(count == 10 || count == 50 || count == 100)
            iterscore(a)= FP;
            itervalue(a) = p;
```

```
            iterABSerror(a) = abs(0-FP);
            a = a+1;
        end
        if(FA*FP > 0)
           bound(1) = p;
        elseif(FA*FP < 0)
           bound(2) = p;
        else
            break;
        end

    end
    root = p;
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

In the main script for question3 I am taking the inputs then printing the outputs that are coming from the bisection method function.

```
%% Bisection Method
fprintf("Welcome to the Bisection Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
[root, error, Fzero, iterscore, itervalue, iterabserror] =
q3_BisectionMethod(lowerbound, upperbound, maxiter);
fprintf('The founded root is %.6f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.6f', Fzero);
fprintf('\n')
fprintf('The relative error with the root given by fzero function is
%.12f', error);
fprintf('\n')


fprintf('The founded root at 10th iteration is %.12f with function
value %.12f', itervalue(1), iterscore(1));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(1));
fprintf('\n')
fprintf('The founded root at 50th iteration is %.30f with function
value %.22f', itervalue(2), iterscore(2));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(2));
fprintf('\n')
fprintf('The founded root at 100th iteration is %.30f with function
value %.22f', itervalue(3), iterscore(3));
fprintf('\n')
```

```
fprintf('The absolute error is %.12f', iterabserror(3));
fprintf('\n')
```

Here, you will see a demonstration of the Bisection Method for the following question.

```
Welcome to the Bisection Method function
Please enter the lower bound:0
Please enter the upper bound:3
Please enter the maximum iteration number:100
The founded root is 1.276387
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is 0.000000000000
The founded root at 10th iteration is 1.274414062500 with function value -0.076637735679
The absolute error is 0.076637735679
The founded root at 50th iteration is 1.276386607154200270031196851050 with function value 0.000000000000870414851
The absolute error is 0.000000000000
The founded root at 100th iteration is 1.276386607154198049585147600737 with function value -0.000000000000017763568
The absolute error is 0.000000000000
```

As you can see it converges beautifully to the real root before 50$^{th}$ iteration. In the 10$^{th}$ iteration the absolute error is not as small as the following iteration scores.

## *False Position Method*

I've given three parameters as input to the function: lower bound, upper bound, and number of iterations. The number of iterations for the given question will be 100 because I need to analyze the 10th, 50th, and 100th iterations. The lower bound is 0 and the upper bound is 3. I have 6 outputs that I will use in the main script that I have written, and root represents the value that I discovered using the False Position method. The root provided by the built-in function is fzero. Itervalue(i) is the approximate root of the function for the given iteration, whereas iterscore is the output with the approximate root of the function. iterabserror is the absolute error between true value of the function which is 0 and function value of the approximate root. In the function I am calculating the absolute errors of the each given iteration number. Additionally, I am calculating the relative error of the last iteration's root with the real root.

```
function [root, finalerror, iteration, iterscore, itervalue,
iterABSerror] = q3_FalsePositionMethod(lowerbound, upperbound,
maxiter)
    myfunc = @(x)exp(x^3) - 8; %first function
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    score = fzero(myfunc, bound);
    a = 1;
    p = -1;
    while(count < maxiter)
        count = count + 1;
        FA = myfunc(bound(1));
        FB = myfunc(bound(2));
        %p = bound(1) + ((FA*(bound(1)- bound(2)))/(FB-FA));
```

```
        p = (abs(FB)*bound(1) + abs(FA)*bound(2)) / (abs(FA) +
abs(FB));
        FP = myfunc(p);
        if(count == 10 || count == 50 || count == 100)
            iterscore(a)= FP;
            itervalue(a) = p;
            iterABSerror(a) = abs(0-FP);
            a = a+1;
        end
        if(FA*FP > 0)
           bound(1) = p;
        elseif(FA*FP < 0)
           bound(2) = p;
        else
            break;
        end
    end
    root = p;
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

In the main script for question3 I am taking the inputs then printing the outputs that are coming from the False Position method.

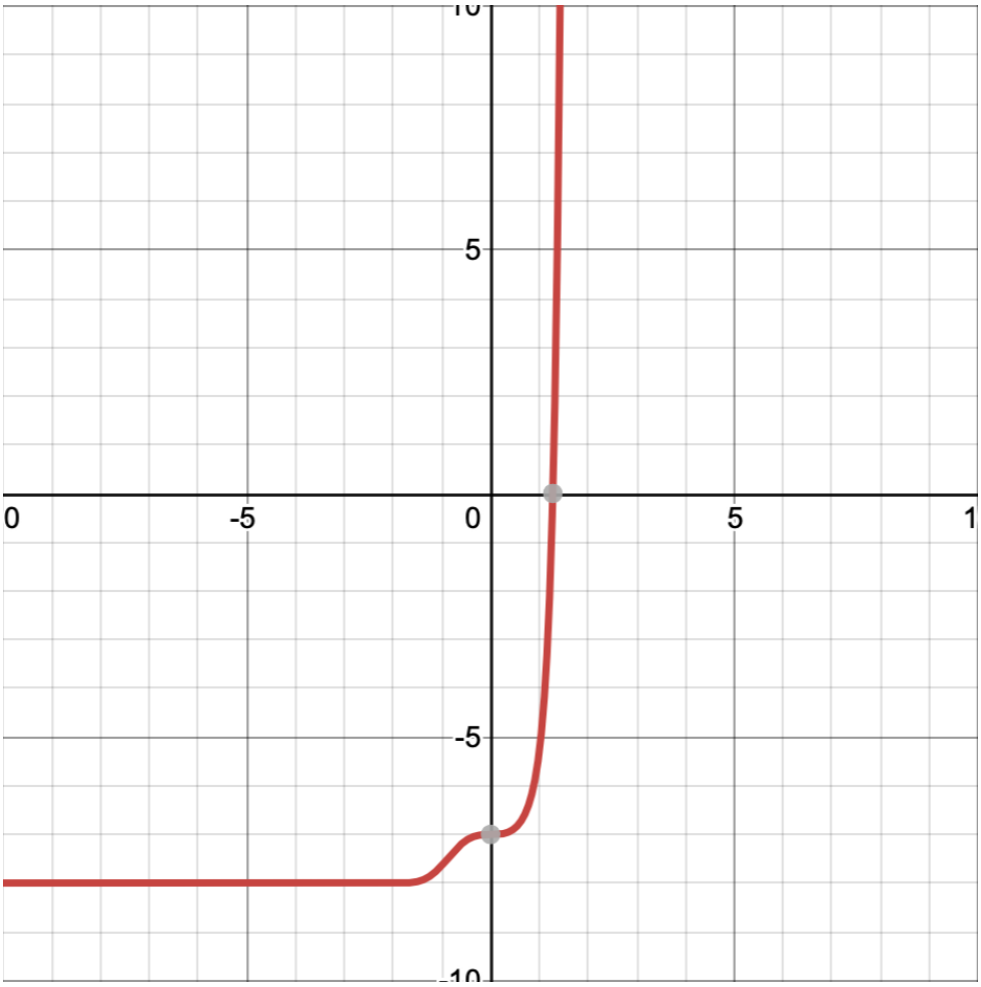
```
%% False Position Method
fprintf("Welcome to the False Position Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
[root, error, Fzero, iterscore, itervalue, iterabserror] =
q3_FalsePositionMethod(lowerbound, upperbound, maxiter);
fprintf('The founded root is %.6f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.6f', Fzero);
fprintf('\n')
fprintf('The relative error with the root given by fzero function is
%.12f', error);
fprintf('\n')


fprintf('The founded root at 10th iteration is %.12f with function
value %.12f', itervalue(1), iterscore(1));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(1));
fprintf('\n')
fprintf('The founded root at 50th iteration is %.30f with function
value %.22f', itervalue(2), iterscore(2));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(2));
fprintf('\n')
```

```
fprintf('The founded root at 100th iteration is %.30f with function
value %.22f', itervalue(3), iterscore(3));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(3));
fprintf('\n')
```

Here, you will see a demonstration of the False Position Method for the following question.

```
Welcome to the False Position Method function
Please enter the lower bound:0
Please enter the upper bound:3
Please enter the maximum iteration number:100
The founded root is 0.000000
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is 0.999999996908
The founded root at 10th iteration is 0.000000000395 with function value -7.000000000000
The absolute error is 7.000000000000
The founded root at 50th iteration is 0.00000000197350525673360905593 with function value -7.000000000000000000000
The absolute error is 7.000000000000
The founded root at 100th iteration is 0.000000003947010512168979647170 with function value -7.000000000000000000000
The absolute error is 7.000000000000
```

As you can see does not converge to the real root. Thus, we could not get the proper solution from the false position method.




This is the $e^{x^3} - 8$ function's plot taken by Desmos Graph Calculator. After also carefully reviewing the plot, this is a case where False Position method works poorly.

## *Modified False Position Method*

I've given three parameters as input to the function: lower bound, upper bound, and number of iterations. The number of iterations for the given question will be 100 because I need to analyze the 10th, 50th, and 100th iterations. The lower bound is 0 and the upper bound is 3. I have 6 outputs that I will use in the main script that I have written, and root represents the value that I discovered using the Modified False Position method. The root provided by the built-in function is Fzero. Itervalue(i) is the approximate root of the function for the given iteration, whereas iterscore is the output with the approximate root of the function. iterabserror is the absolute error between true value of the function which is 0 and function value of the approximate root.

In the function I am calculating the absolute errors of the each given iteration number. Additionally, I am calculating the relative error of the last iteration's root with the real root. Here, as you can see, I have additional two counters in the function. I decided to use the approach that Mustafa Hocam talked about it in the class. If a counter value bigger or equal to 2. I am dividing the function's value at that boundary. If counter is reset, FB or FA value are reset accordingly.

```
function [root, finalerror, iteration, iterscore, itervalue,
iterABSerror] = q3_ModifiedFalsePositionMethod(lowerbound,
upperbound, maxiter)
    myfunc = @(x)exp(x^3) - 8; %first function
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    leftcount = 0;
    rightcount = 0;
    score = fzero(myfunc, bound);
    a = 1;
    p = -1;
    FA = myfunc(bound(1));
    FB = myfunc(bound(2));
    while(count < maxiter)
        %p = (abs(FB)*bound(1) + abs(FA)*bound(2)) / (abs(FA) +
abs(FB));
        count = count + 1;
        p = bound(1) + ((FA*(bound(1)- bound(2)))/(FB-FA));
        FP = myfunc(p);
        if(count == 10 || count == 50 || count == 100)
            iterscore(a)= FP;
            itervalue(a) = p;
            iterABSerror(a) = abs(0-FP);
            a = a+1;
        end
        if(FA*FP > 0)
           leftcount = 0;
           rightcount = rightcount + 1;
```

```
            bound(1) = p;
            FA = myfunc(bound(1));
            if(rightcount >= 2)
                FB = FB/2;
            end
        elseif(FA*FP < 0)
            rightcount = 0;
            leftcount = leftcount + 1;
            bound(2) = p;
            FB = myfunc(bound(2));
            if(leftcount >= 2)
                FA = FA/2;
            end
        else
            break;
        end
    end
    root = p;
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

In the main script for question3 I am taking the inputs then printing the outputs that are coming from the Modified False Position method.

```
%% Modified False Position Method
fprintf("Welcome to the Modified False Position Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
[root, error, Fzero, iterscore, itervalue, iterabserror] =
q3_ModifiedFalsePositionMethod(lowerbound, upperbound, maxiter);
fprintf('The founded root is %.6f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.6f', Fzero);
fprintf('\n')
fprintf('The relative error with the root given by fzero function is
%.12f', error);
fprintf('\n')

fprintf('The founded root at 10th iteration is %.12f with function
value %.12f', itervalue(1), iterscore(1));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(1));
fprintf('\n')
fprintf('The founded root at 50th iteration is %.36f with function
value %.22f', itervalue(2), iterscore(2));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(2));
```

```
fprintf('\n')
fprintf('The founded root at 100th iteration is %.36f with function
value %.22f', itervalue(3), iterscore(3));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(3));
fprintf('\n')

%%%%% BRACKETING METHODS %%%%%
```

Now, you will see a demonstration of the Modified False Position Method for the following question.

```
Welcome to the Modified False Position Method function
Please enter the lower bound:0
Please enter the upper bound:3
Please enter the maximum iteration number:100
The founded root is 1.276387
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is 0.000000000000
The founded root at 10th iteration is 0.000000020209 with function value 0.000000020209
The absolute error is 7.000000000000
The founded root at 50th iteration is 1.276386607154198049585147600737400353 with function value -0.000000000000000017763568
The absolute error is 0.000000000000
The founded root at 100th iteration is 1.276386607154198049585147600737400353 with function value -0.000000000000000017763568
The absolute error is 0.000000000000
>>
```

Thus, solution converges to the real root after some iterations. The behavior was like False Position Method at $10^{th}$ step. However, the behavior changed, and it converges to the real root. So, modification was satisfactory to find the root.

## *Newton Method*

I've given two parameters as input to the function: initial approximation and number of iterations. The number of iterations for the given question will be 100 because I need to analyze the 10th, 50th, and 100th iterations I have 6 outputs that I will use in the main script that I have written, and root represents the value that I discovered using the Newton method. The root provided by the built-in function is Fzero. Itervalue(i) is the approximate root of the function for the given iteration, whereas iterscore is the output with the approximate root of the function. iterabserror is the absolute error between true value of the function which is 0 and function value of the approximate root.

In the function I am calculating the absolute errors of the each given iteration number. I have also the derivative of the function since it will be used the find the root. Additionally, I am calculating the relative error of the last iteration's root with the real root.

```
function [root, finalerror, iteration, iterscore, itervalue,
iterABSerror] = q3_NewtonMethod(initial, maxiter)
    myfunc = @(x)exp(x^3) - 8; %first function
```

```
    derivative = @(x)3*(x^2)*exp(x^3); % derivative of the function
    bound(1) = initial; % boundaries
    count = 0;
    score = fzero(myfunc, [0 3]);
    a = 1;
    i = 2;
    while(count < maxiter)
        bound(i) = bound(i-1) - (myfunc(bound(i-
1))/derivative(bound(i-1)));
        count = count + 1;
        if(count == 10 || count == 50 || count == 100)
            iterscore(a)= myfunc(bound(i));
            itervalue(a) = bound(i);
            iterABSerror(a) = abs(0-iterscore(a));
            a = a+1;
        end
        i = i + 1;
    end
    root = bound(i-1);
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

I am taking the inputs then printing the outputs that are coming from the Newton method.

```
%% Newton Method
fprintf("Welcome to the Newton Method function\n");
p0 = input('Please enter the initial value:');
maxiter = input('Please enter the maximum iteration number:');

[root, error, Fzero, iterscore, itervalue, iterabserror] =
q3_NewtonMethod(p0, maxiter);
fprintf('The founded root is %.6f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.6f', Fzero);
fprintf('\n')
fprintf('The relative error with the root given by fzero function is
%.12f', error);
fprintf('\n')
fprintf('The founded root at 10th iteration is %.12f with function
value %.12f', itervalue(1), iterscore(1));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(1));
fprintf('\n')
fprintf('The founded root at 50th iteration is %.36f with function
value %.22f', itervalue(2), iterscore(2));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(2));
fprintf('\n')
fprintf('The founded root at 100th iteration is %.36f with function
value %.22f', itervalue(3), iterscore(3));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(3));
fprintf('\n')
```

Now, you will see a demonstration of the Newton Method for the following question.

```
Welcome to the Newton Method function
Please enter the initial value:2
Please enter the maximum iteration number:100
The founded root is 1.276387
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is 0.000000000000
The founded root at 10th iteration is 1.276386632563 with function value 0.000000993478
The absolute error is 0.000000993478
The founded root at 50th iteration is 1.27638660715419804958514760073740035 3 with function value -0.00000000000000017763568
The absolute error is 0.000000000000
The founded root at 100th iteration is 1.27638660715419804958514760073740035 3 with function value -0.00000000000000017763568
The absolute error is 0.000000000000
>>
```

```
Welcome to the Newton Method function
Please enter the initial value:0.66
Please enter the maximum iteration number:100
The founded root is 1.276387
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is 0.000000000000
The founded root at 10th iteration is 4.333361025359 with function value 2184642303497555504741860441713541 12.000000000000
The absolute error is 2184642303497555504741860441713541 12.000000000000
The founded root at 50th iteration is 3.464804816716284374678025415050797164 with function value 1159522292921283072.000000000000000000000
The absolute error is 1159522292921283072.000000000000
The founded root at 100th iteration is 1.27638660715419804958514760073740035 3 with function value -0.00000000000000017763568
The absolute error is 0.000000000000
```

```
Welcome to the Newton Method function
Please enter the initial value:0.2
Please enter the maximum iteration number:100
The founded root is NaN
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is NaN
The founded root at 10th iteration is NaN with function value NaN
The absolute error is NaN
The founded root at 50th iteration is NaN with function value NaN
The absolute error is NaN
The founded root at 100th iteration is NaN with function value NaN
The absolute error is NaN
```

In the following 3 demonstration of the Newton's method, method performance changes with respect to the initial value is given. When the initial value is close to the 0, it does not give a solution. When the number is close to the 3 it gives a good convergence behaviour. In the second where the initial value was 0.66 the convergence behaviour was bad at $10^{th}$ and $50^{th}$ iteration. It converges successfully at $100^{th}$ iteration. The slope of the function and the value of the function are significant parameters to analyze the convergence behaviour of the function when using Newton's method.

## *Secant Method*

I've given three parameters as input to the function: first initial approximation, second initial approximation and number of iterations. The number of iterations for the given question will be 100 because I need to analyze the 10th, 50th, and 100th iterations.

```
function [root, finalerror, iteration, iterscore, itervalue,
iterABSerror] = q3_SecantMethod(initial, second, maxiter)
    myfunc = @(x)exp(x^3) - 8; %first function
    bound = [initial second]; % boundaries
```

```
    count = 0;
    score = fzero(myfunc, [0 3]);
    a = 1;
    i = 3;
    while(count < maxiter)
        bound(i) = bound(i-1) + (myfunc(bound(i-1))*(bound(i-2)-
bound(i-1)))/ (myfunc(bound(i-1))- myfunc(bound(i-2)));
        count = count + 1;
        if(count == 10 || count == 50 || count == 100)
            iterscore(a)= myfunc(bound(i));
            itervalue(a) = bound(i);
            iterABSerror(a) = abs(0-iterscore(a));
            a = a+1;
        end
        i = i +1;
    end
    root = bound(i-1);
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

As usual, I am giving inputs and then printing the outputs.

```
%% Secant Method
fprintf("Welcome to the Secand Method function\n");
p0 = input('Please enter the p0:');
p1 = input('Please enter the p1:');
maxiter = input('Please enter the maximum iteration number:');
[root, error, Fzero, iterscore, itervalue, iterabserror] =
q3_SecantMethod(p0, p1, maxiter);
fprintf('The founded root is %.6f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.6f', Fzero);
fprintf('\n')
fprintf('The relative error with the root given by fzero function is
%.12f', error);
fprintf('\n')


fprintf('The founded root at 10th iteration is %.12f with function
value %.12f', itervalue(1), iterscore(1));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(1));
fprintf('\n')
fprintf('The founded root at 50th iteration is %.36f with function
value %.22f', itervalue(2), iterscore(2));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(2));
fprintf('\n')
fprintf('The founded root at 100th iteration is %.36f with function
value %.22f', itervalue(3), iterscore(3));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(3));
fprintf('\n')
```

```
Welcome to the Secand Method function
Please enter the p0:1.5
Please enter the p1:1.9
Please enter the maximum iteration number:100
The founded root is NaN
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is NaN
The founded root at 10th iteration is 1.276386607154 with function value 0.000000000000
The absolute error is 0.000000000000
The founded root at 50th iteration is NaN with function value NaN
The absolute error is NaN
The founded root at 100th iteration is NaN with function value NaN
The absolute error is NaN
Welcome to the Secand Method function
Please enter the p0:0
Please enter the p1:3
Please enter the maximum iteration number:100
The founded root is NaN
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is NaN
The founded root at 10th iteration is NaN with function value NaN
The absolute error is NaN
The founded root at 50th iteration is NaN with function value NaN
The absolute error is NaN
The founded root at 100th iteration is NaN with function value NaN
The absolute error is NaN
Welcome to the Secand Method function
Please enter the p0:2
Please enter the p1:3
Please enter the maximum iteration number:100
The founded root is NaN
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is NaN
The founded root at 10th iteration is 1.424442541363 with function value 9.997736230143
The absolute error is 9.997736230143
The founded root at 50th iteration is NaN with function value NaN
The absolute error is NaN
The founded root at 100th iteration is NaN with function value NaN
The absolute error is NaN
```

As you can see in the figure above, the behavior of convergence changes with initial approximations and number of iterations. When we give initial values as 1.5 and 1.9 at $10^{th}$ iteration, the root was converging to the real root beautifully. However, after further iterations it diverges. In the second example, the function diverges at each given iteration. In the third example, the function was converging to the real root at the $10^{th}$ step. However, it diverged at $50^{th}$ and $100^{th}$ step. Hence, the selection of initial values and the decided number of iterations can give you good results or divergent results.

## *Fixed Point Iteration*

I am taking inputs for the initial value and number of iterations. Then, I define function g function as x – function f. I am using the following formula to find the root.

$$x_{i+1} = g(x_i)$$

```
function [root, finalerror, iteration, iterscore, itervalue,
iterABSerror] = q3_FixedPointIteration(initial, maxiter)
    myfunc = @(x)exp(x^3) - 8; %first function
    %func_g = @(x)x - exp(x^3) - 8; %I converted this from the main
function
    func_g = @(x)log(8)^(1/3);
    x(1) = initial;
    count = 0;
    score = fzero(myfunc, [0 3]);
    a = 1;
    i = 2;
    while(count < maxiter)
        x(i) = func_g(x(i-1));
        count = count + 1;
        if(count == 10 || count == 50 || count == 100)
            iterscore(a)= myfunc(x(i));
            itervalue(a) = x(i);
            iterABSerror(a) = abs(0-iterscore(a));
            a = a+1;
        end
        i = i + 1;
    end
    root = x(i-1);
    relative = abs(score - root)/abs(score);
    finalerror = relative;
    iteration = score;
end
```

As usual, I am giving inputs and then printing the outputs.

```
%% Fixed Point Iteration
fprintf("Welcome to the Fixed Point Iteration function\n");
p0 = input('Please enter the initial value:');
maxiter = input('Please enter the maximum iteration number:');


[root, error, Fzero, iterscore, itervalue, iterabserror] =
q3_FixedPointIteration(p0, maxiter);
fprintf('The founded root is %.6f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.6f', Fzero);
fprintf('\n')
fprintf('The relative error with the root given by fzero function is
%.12f', error);
fprintf('\n')


fprintf('The founded root at 10th iteration is %.12f with function
value %.12f', itervalue(1), iterscore(1));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(1));
fprintf('\n')
fprintf('The founded root at 50th iteration is %.36f with function
value %.22f', itervalue(2), iterscore(2));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(2));
fprintf('\n')
```

```
fprintf('The founded root at 100th iteration is %.36f with function
value %.22f', itervalue(3), iterscore(3));
fprintf('\n')
fprintf('The absolute error is %.12f', iterabserror(3));
fprintf('\n')

%%%%% OPEN METHODS %%%%%
```

I am using 2 different g functions to see the difference in the selection of the g. We can get x from $e^{x^3} - 8$ as $\ln 8^{(1/3)}$ and I can also write the g function as x - $e^{x^3} + 8$

```
Welcome to the Fixed Point Iteration function
Please enter the initial value:2
Please enter the maximum iteration number:100
The founded root is 1.276387
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is 0.000000000000
The founded root at 10th iteration is 1.276386607154 with function value -0.000000000000
The absolute error is 0.000000000000
The founded root at 50th iteration is 1.27638660715419804958514760073740035 3 with function value -0.00000000000000017763568
The absolute error is 0.000000000000
The founded root at 100th iteration is 1.27638660715419804958514760073740035 3 with function value -0.00000000000000017763568
The absolute error is 0.000000000000
```

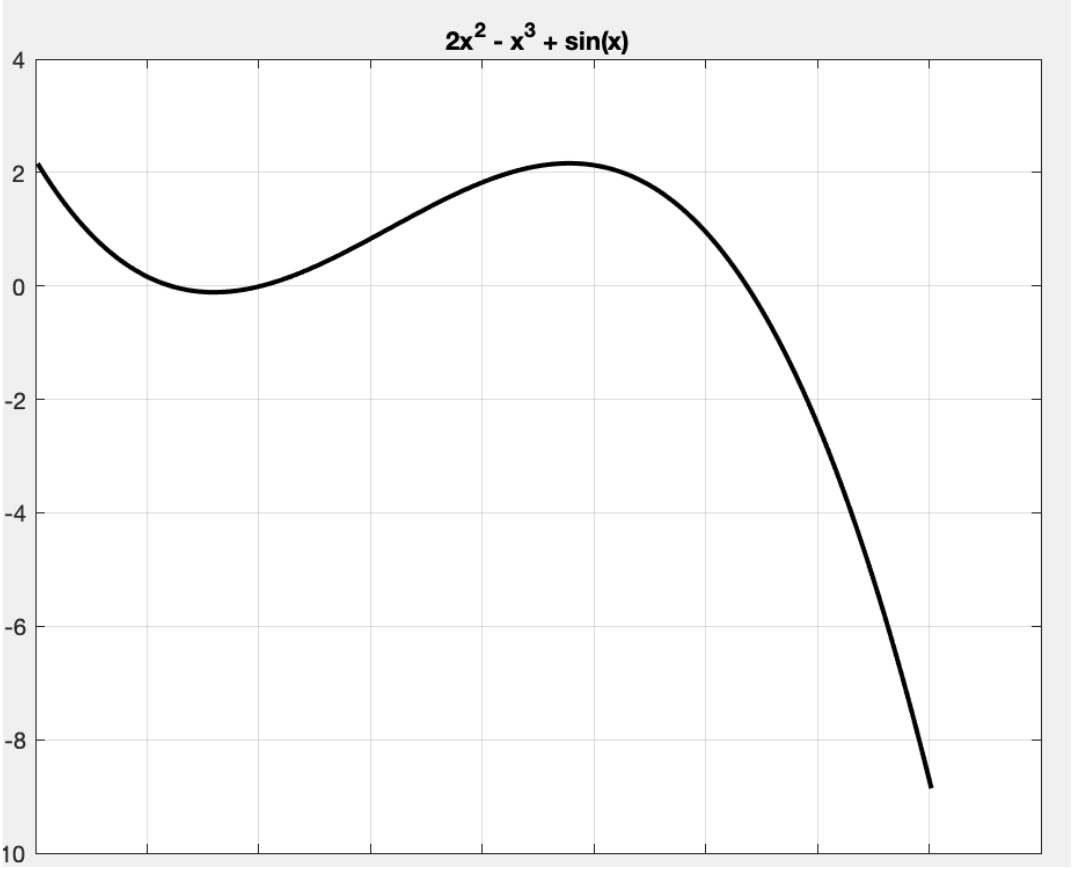
When it is a constant function and when we get x only from the equation we are finding the root of the function indeed. However, there are significant digit difference between the root by fzero and founded root, when we investigate deeper we get some difference. If we keep the significant digit value between 5 or 6 it would give the difference zero.

```
Welcome to the Fixed Point Iteration function
Please enter the initial value:2
Please enter the maximum iteration number:100
The founded root is -2178.957987
The root given by fzero function is 1.276387
The relative error with the root given by fzero function is 1708.130092738816
The founded root at 10th iteration is -2898.957987041728 with function value -8.000000000000
The absolute error is 8.000000000000
The founded root at 50th iteration is -2578.95798704172830184688791632652282 7148 with function value -8.00000000000000000000
The absolute error is 8.000000000000
The founded root at 100th iteration is -2178.95798704172830184688791632652282 7148 with function value -8.00000000000000000000
The absolute error is 8.000000000000
```

When I change the g function to x - $e^{x^3} + 8$, you can see that the result does not converge to the real function. Hence, the choice of g is as significant as choosing the initial value.

## Question 4

The equation $2x^2 - x^3 + \sin(x) = 0$ has three roots. Compute all of them by selecting an appropriate initial interval $[a_k, b_k]$ from the plot of the function and an initial starting point as $(a_k + b_k)/2$. In each case, find the roots with an accuracy of six significant digits. The following graph is plotted in MATLAB with respect to the material taught in the second tutorial.




As it can be seen there are 3 roots for this equation. So, let's find all roots by the bracketing methods and open methods.

## *Bisection Method*

I chose the tolerance as 0.05 x $10^{-k}$ to make sure that we have 6 significant digit accuracy. I used the relative approximate error since it was stated in the second slide set of the lecture.

```
function [root, iteration, score] = q4_BisectionMethod(lowerbound,
upperbound, maxiter, tol)
    myfunc = @(x)2*x^2 - x^3 + sin(x); %first function
    tolerance = 0.05*10^-tol; % given tolerance
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    score = fzero(myfunc, bound);
    error = 1000;
    value(1) = 0;
    i = 0;
    p = -1;
    while(error > tolerance && count < maxiter)
        i = i + 1;
        count = count + 1;
        FA = myfunc(bound(1));
        FB = myfunc(bound(2));
        p = (bound(1) + bound(2)) /2;
        FP = myfunc(p);
        value(i) = p;
```

```
        if(FA*FP > 0)
           bound(1) = p;
        elseif (FA*FP < 0)
           bound(2) = p;
        else
           break;
        end
        if(i > 1)
            error = abs(value(i) - value(i-1))/ abs(value(i));
        end
    end
    root = p;
    iteration = count;
end
```

Here, we are getting inputs for lower bound, upper bound, maximum number of iteration and tolerance. Then, I am printing the values given as the output of the function.

```
%% Bisection Method
fprintf("Welcome to the Bisection Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the wanted number accuracy in significant
digits:');
[root,  iteration, score] = q4_BisectionMethod(lowerbound,
upperbound, maxiter, tol);
fprintf('The root is %.10f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.10f', score);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')
```

I give the boundaries as [-1, -0.2], [-0.2, 0.2] and [1, 3] for finding the roots of the function.

Here, you can see the approximate roots where at least 6 significant digit accuracy is satisfied.

```
Welcome to the Bisection Method function
Please enter the lower bound:-1
Please enter the upper bound:-0.2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is -0.4046128392
The root given by fzero function is -0.4046128360
The iteration number is 26
Welcome to the Bisection Method function
Please enter the lower bound:-0.2
Please enter the upper bound:0.2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 0.0000000000
The root given by fzero function is -0.0000000000
The iteration number is 1
Welcome to the Bisection Method function
Please enter the lower bound:1
Please enter the upper bound:3
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 2.1741901040
The root given by fzero function is 2.1741901205
The iteration number is 25
```

### *False Position Method*

I chose the tolerance similar to the approach I have in Bisection Method to make sure that we have 6 significant digit accuracy. I used the relative approximate error since it was stated in the second slide set of the lecture.

```
function [root, iteration, score] = q4_FalsePositionMethod(lowerbound,
upperbound, maxiter, tol)
    tolerance = 0.05*10^-tol; % given tolerance
    myfunc = @(x)2*x^2 -x^3 + sin(x); %first function
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    error = 1000;
    score = fzero(myfunc, bound);
    value(1) = 0;
    i = 0;
    p = -1;
    while(error > tolerance && count < maxiter)
        i = i + 1;
        count = count + 1;
        FA = myfunc(bound(1));
        FB = myfunc(bound(2));
        p = bound(1) + ((FA*(bound(1)- bound(2)))/(FB-FA));
        FP = myfunc(p);
        value(i) = p;
        if(FA*FP > 0)
           bound(1) = p;
        elseif(FA*FP < 0)
           bound(2) = p;
```

```
        else
           break;
        end
        if(i > 1)
            error = abs(value(i) - value(i-1))/ abs(value(i));
        end
    end
    root = p;
    iteration = count;
end
```

Here, we are getting inputs for lower bound, upper bound, maximum number of iteration and tolerance. Then, I am printing the values given as the output of the function.

```
%% False Position Method
fprintf("Welcome to the False Position Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the wanted number accuracy in significant
digits:');
[root, iteration, score] = q4_FalsePositionMethod(lowerbound, upperbound,
maxiter, tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.8f', score);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')
```

I give the boundaries as [-1 , -0.2], [-0.2, 0.2] and [1, 3] for finding the roots of the function.

Here, you can see the approximate roots where at least 6 significant digit accuracy is satisfied.

```
Welcome to the False Position Method function
Please enter the lower bound:-1
Please enter the upper bound:-0.2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is -0.40461280
The root given by fzero function is -0.40461284
The iteration number is 41
Welcome to the False Position Method function
Please enter the lower bound:-0.2
Please enter the upper bound:0.2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is -0.00000000
The root given by fzero function is -0.00000000
The iteration number is 100
Welcome to the False Position Method function
Please enter the lower bound:1
Please enter the upper bound:3
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 2.17419007
The root given by fzero function is 2.17419012
The iteration number is 22
```

Here, you can see the approximate roots where at least 6 significant digit accuracy is satisfied. The performance to find the second root of the function was performed poorly in False Position Method. However, it performed I believe above average for finding the other roots.

### *Modified False Position Method*

I used the same approach I used in False Position Method. I just modified the algorithm according to the lecture talks with Mustafa Hoca that I have used in q1, q2 and q3 respectively.

```
function [root, iteration, score] =
q4_ModifiedFalsePositionMethod(lowerbound, upperbound, maxiter, tol)
    tolerance = 0.05*10^-tol; % given tolerance
    myfunc = @(x)2*x^2 -x^3 + sin(x); %first function
    bound = [lowerbound upperbound]; % boundaries
    count = 0; %iteration count
    leftcount = 0;
    rightcount = 0;
    error = 1000;
    score = fzero(myfunc, bound);
    value(1) = 0;
    i = 0;
    p = -1;
    FA = myfunc(bound(1));
    FB = myfunc(bound(2));
    while(error > tolerance && count < maxiter)
        i = i + 1;
        count = count + 1;
        %p = (abs(FB)*bound(1) + abs(FA)*bound(2)) / (abs(FA) +
abs(FB));
        p = bound(1) + ((FA*(bound(1)- bound(2)))/(FB-FA));
        FP = myfunc(p);
        value(i) = p;
        if(FA*FP > 0)
           leftcount = 0;
           rightcount = rightcount + 1;
           bound(1) = p;
           FA = myfunc(bound(1));
           if(rightcount >= 2)
               FB = FB/2;
           end
        elseif(FA*FP < 0)
           rightcount = 0;
           leftcount = leftcount + 1;
           bound(2) = p;
           FB = myfunc(bound(2));
           if(leftcount >= 2)
               FA = FA/2;
           end
        else
            break;
        end
        if(i > 1)
```

```
            error = abs(value(i) - value(i-1))/ abs(value(i));
        end
    end
    root = p;
    iteration = count;
end
```

Here, we are getting inputs for lower bound, upper bound, maximum number of iteration and tolerance. Then, I am printing the values given as the output of the function.

```
%% Modified False Position Method
fprintf("Welcome to the Modified False Position Method function\n");
lowerbound = input('Please enter the lower bound:');
upperbound = input('Please enter the upper bound:');
while(upperbound < lowerbound)
    fprintf("Upper bound is smaller than the lower bound\n");
    upperbound = input('Please enter the upper bound: ');
end
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the wanted number accuracy in significant
digits:');
[root, iteration, score] =
q4_ModifiedFalsePositionMethod(lowerbound, upperbound, maxiter,
tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.8f', score);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')
%%%%% BRACKETING METHODS %%%%%
```

I give the boundaries as [-1 , -0.2], [-0.2, 0.2] and [1, 3] for finding the roots of the function.

Here, you can see the approximate roots where at least 6 significant digit accuracy is satisfied.

```
Welcome to the Modified False Position Method function
Please enter the lower bound:-1
Please enter the upper bound:-0.2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is -0.40461284
The root given by fzero function is -0.40461284
The iteration number is 9
Welcome to the Modified False Position Method function
Please enter the lower bound:-0.2
Please enter the upper bound:0.2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 0.00000000
The root given by fzero function is -0.00000000
The iteration number is 13
Welcome to the Modified False Position Method function
Please enter the lower bound:1
Please enter the upper bound:3
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 2.17419012
The root given by fzero function is 2.17419012
The iteration number is 10
```

I believe the performance of Modified False position was better than False Position method for the given function.

### *Newton Method*

I used the derivative and the normal function I assigned the tolerance like the previous ones

```
function [root, iteration, score] = q4_NewtonMethod(initial,
maxiter, tol)
    tolerance = 0.05*10^-tol; % given tolerance
    myfunc = @(x)2*x^2 -x^3 + sin(x); %first function
    derivative = @(x)4*x - 3*x^2 + cos(x); % derivative of the
function
    bound(1) = initial; % boundaries
    count = 0;
    score = fzero(myfunc, [1 3]);
    error = 1000;
    i = 1;
    while(error > tolerance && count < maxiter)
        i = i + 1;
        bound(i) = bound(i-1) - (myfunc(bound(i-
1))/derivative(bound(i-1)));
        count = count + 1;
        if(i > 1)
            error = abs(bound(i) - bound(i-1))/ abs(bound(i));
        end
    end
    root = bound(i);
    iteration = count;
end
```

Here, we are getting inputs for initial approximation, maximum number of iteration and tolerance. Then, I am printing the values given as the output of the function.

```
%% Newton Method
fprintf("Welcome to the Newton Method function\n");
p0 = input('Please enter the initial value:');
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the wanted number accuracy in significant
digits:');

[root, iteration, score] = q4_NewtonMethod(p0, maxiter, tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.8f', score);
fprintf('\n')

fprintf('The iteration number is %d', iteration);
fprintf('\n')
```

I give the boundaries as [-1 , -0.2], [-0.2, 0.2] and [1, 3] for finding the roots of the function.

Middle points for each root that are going to be used are as follows: -0.6, 0, 2

Here, you can see the approximate roots where at least 6 significant digit accuracy is satisfied.

```
Welcome to the Newton Method function
Please enter the initial value:-0.6
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is -0.40461284
The root given by fzero function is -0.40461284
The iteration number is 6
Welcome to the Newton Method function
Please enter the initial value:0
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 0.00000000
The root given by fzero function is -0.00000000
The iteration number is 1
Welcome to the Newton Method function
Please enter the initial value:2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 2.17419012
The root given by fzero function is 2.17419012
The iteration number is 5
>>
```

The iteration number to find each of the roots demonstrate that this function converges beautifully. A very good solution with very less number of iterations.

## *Secant Method*

```
function [root, iteration, score] = q4_SecantMethod(initial, second,
maxiter, tol)
    tolerance = 0.05*10^-tol; % given tolerance
    myfunc = @(x)2*x^2 -x^3 + sin(x); %first function
    bound = [initial second]; % boundaries
    count = 0;
    score = fzero(myfunc, [-1 -0.2]);
    error = 1000;
    i = 2;
    while(error > tolerance && count < maxiter)
        i = i +1;
        bound(i) = bound(i-1) + (myfunc(bound(i-1))*(bound(i-2)-
bound(i-1)))/ (myfunc(bound(i-1))- myfunc(bound(i-2)));
        count = count + 1;
        if(i > 2)
            error = abs(bound(i) - bound(i-1))/ abs(bound(i));
        end


    end
    root = bound(i);
    iteration = count;
end
```

Here, we are getting inputs for initial approximation, second approximation, maximum number of iteration and tolerance. Then, I am printing the values given as the output of the function.

```
%% Secant Method
fprintf("Welcome to the Secand Method function\n");
p0 = input('Please enter the p0:');
p1 = input('Please enter the p1:');
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the wanted number accuracy in significant
digits:');
[root, iteration, score] = q4_SecantMethod(p0, p1, maxiter, tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.8f', score);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')
```

I give the boundaries as [-1 , -0.2], [-0.2, 0.2] and [1, 3] for finding the roots of the function. Thus, I use the bound values as initial and second approximation values.

```
Welcome to the Secand Method function
Please enter the p0:-1
Please enter the p1:-0.2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is -0.40461284
The root given by fzero function is -0.40461284
The iteration number is 13
Welcome to the Secand Method function
Please enter the p0:-0.2
Please enter the p1:0.2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 0.00000000
The root given by fzero function is -0.00000000
The iteration number is 11
Welcome to the Secand Method function
Please enter the p0:1
Please enter the p1:3
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 2.17419012
The root given by fzero function is 2.17419012
The iteration number is 12
>>
```

When I changed the tolerance range to $0.5 \times 10^{-k}$ I get the following results which are like the previous one. I just wanted to show it as another example.

```
Welcome to the Secand Method function
Please enter the p0:1
Please enter the p1:3
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 2.17419012
The root given by fzero function is 2.17419012
The iteration number is 11
Welcome to the Secand Method function
Please enter the p0:-0.2
Please enter the p1:0.2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 0.00000000
The root given by fzero function is -0.00000000
The iteration number is 11
Welcome to the Secand Method function
Please enter the p0:-1
Please enter the p1:-0.2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is -0.40461284
The root given by fzero function is -0.40461284
The iteration number is 13
```

## *Fixed Point Iteration*

I assigned the g function from the f function. Then calculate the tolerance accordingly.

```
function [root, iteration, score] = q4_FixedPointIteration(initial,
maxiter, tol)
    tolerance = 0.05*10^-tol; % given tolerance
    func_g = @(x)(2*x^2 + sin(x))^(1/3); %I converted this from the
main function
    x(1) = initial;
    myfunc = @(x)2*x^2 -x^3 + sin(x); %first function
    count = 0;
    score = fzero(myfunc, [1 3]);
    error = 1000;
    i = 1;
    while(error > tolerance && count < maxiter)
        i = i + 1;
        x(i) = func_g(x(i-1));
        count = count + 1;
        if(i > 1)
            error = abs(x(i) - x(i-1))/ abs(x(i));
        end
    end
    root = x(i);
    iteration = count;
end
```

Here, we are getting inputs for initial approximation, maximum number of iteration and tolerance. Then, I am printing the values given as the output of the function.

```
%% Fixed Point Iteration
fprintf("Welcome to the Fixed Point Iteration function\n");
p0 = input('Please enter the initial value:');
maxiter = input('Please enter the maximum iteration number:');
tol = input('Please enter the wanted number accuracy in significant
digits:');
[root, iteration, score] = q4_FixedPointIteration(p0, maxiter, tol);
fprintf('The root is %.8f', root);
fprintf('\n')
fprintf('The root given by fzero function is %.8f', score);
fprintf('\n')
fprintf('The iteration number is %d', iteration);
fprintf('\n')

%%%%% OPEN METHODS %%%%%
```

I give the boundaries as [-1 , -0.2], [-0.2, 0.2] and [1, 3] for finding the roots of the function.

Middle points for each root that are going to be used are as follows: -0.6, 0, 2

Here, you can see the approximate roots where at least 6 significant digit accuracy is satisfied.

`func_g = @(x)(2*x^2 + sin(x))^(1/3) is used the intervals[1, 3] and [-0.2 0.2].`

```
Welcome to the Fixed Point Iteration function
Please enter the initial value:2
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 2.17419003
The root given by fzero function is 2.17419012
The iteration number is 26
Welcome to the Fixed Point Iteration function
Please enter the initial value:0
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is 0.00000000
The root given by fzero function is -0.00000000
The iteration number is 1
```

`func_g = @(x)sin(x)/(x^2 - 2*x); is used fort he interval [-1, -0.2].`

```
Welcome to the Fixed Point Iteration function
Please enter the initial value:-0.6
Please enter the maximum iteration number:100
Please enter the wanted number accuracy in significant digits:6
The root is -0.40461284
The root given by fzero function is -0.40461284
The iteration number is 12
>>
```

We used different function g to find the roots of the function.