Numerical Analysis

Phase 1 – Root Finding Algorithms

Dahlia Chehata Mahmoud 28

Moamen Raafat El-Baroudy 52

Mohamed Ayman Farrag 61

Mahmoud Mohamed Hussein 70

Moustafa Mahmoud Reda El-Deen 75

1. Algorithms & Pseudo Code:

a. Bisection:

i. Description:

The Bisection method is an algorithm the repeatedly bisects an interval into two halves and then select a half in which a root must lie in for further processing.

ii. Pros:

1. The Basic idea is easy and robust.
2. It'll always converge to a root if the interval is valid.
3. Number of the iterations required to attain an absolute error can be computed theoretically.

iii. Cons:

1. The Method may appear to be fast due to its logarithmic time complexity but It's slow compared to other methods like Newton Raphson or Regular Falsi.
2. It's required to find a valid interval which contains an odd number of roots to begin the algorithm.
3. It doesn't use the fact that which side has the minimum absolute value is closer to the root.

iv. Pseudo Code:

```
Input: Function f, endpoint values a, b, tolerance EPS, maximum iterations MAX_ITER
Condition: a < b, f(a) * f(b) < 0
Output: value not far by more than EPS from the root

ITER ← 1
While ITER ≤ MAX_ITER # limit iterations to prevent infinite loop
prevC ← c
 c ← (a + b)/2 # new midpoint
 If then # solution found
  Output(c)
  return
 End if
 ITER ← ITER + 1 # increment step counter
 If sign(f(c)) = sign(f(a)) then
        a ← c
 else
        b ← c # new interval
 End if
EndWhile
```

b. Regular Falsi:

i. Description:

Regula Falsi assumes that $f(x)$ is linear even though these methods are needed only when $f(x)$ is not linear and usually work well anyway.
The ratio of the change in x, to the result change in y is:

$$(x2 - x1)/(y2 - y1)$$

Then we try to interpolate the position of the root on the x-axis but since the function doesn't represent a linear function, in most cases it won't be the root so we choose a new interval with the new point and one of the previous two points which makes the interval valid.

ii. Pros:

1. It'll always converge to a root if the interval is.
2. The method converges quickly and fast compared to the Bisection in most cases.

iii. Cons:

1. The Method may become slow or get stuck at all in case of convex and concave curves near the x-axis if they aren't handled separately.
2. It's required to find a valid interval which contains an odd number of roots to begin the algorithm.

iv. Pseudo Code:

Input: Function f, endpoint values a, b, tolerance EPS, maximum iterations MAX_ITER

Condition: a < b, f(a) * f(b) < 0

Output: value not far by more than EPS from the root

```
ITER ← 1
While ITER ≤ MAX_ITER # limit iterations to prevent infinite loop
prevC ← c
 c ←  # new root approximation
 If f(c) = 0 < EPS || abs(c – prevC) < EPS then # solution found
  Output(c)
  return
 End if
 ITER ← ITER + 1 # increment step counter
 If sign(f(c)) = sign(f(a)) then
        a ← c
 else
        b ← c # new interval
 End if
EndWhile
```

c. Fixed Point:

i. Description:

a fixed point (sometimes shortened to fix point, also known as an invariant point) of a function is an element of the function's domain that is mapped to itself by the function. That is to say, c is a fixed point of the function f(x) if and only if f(c) = c.

Not all functions have fixed points: for example, if f is a function defined on the real numbers as f(x) = x + 1, then it has no fixed points, since x is never equal to x + 1 for any real number. In graphical terms, a fixed point means the point (x, f(x)) is on the line y = x, or in other words the graph of f has a point in common with that line.

ii. Pros:

1. There can be multiple valid g(x) for a function f(x) where they can give multiple options and some converge faster.

iii. Cons:

1. The Method doesn't guarantee convergence where upon the g(x) it may sometimes diverge.

iv. Pseudo Code:

```
Input: Function f and g, initial guess a, tolerance EPS, maximum iterations MAX_ITER
Output: value not far by more than EPS from the root

ITER ← 1
While ITER ≤ MAX_ITER # limit iterations to prevent infinite loop
C ← a
 a ← g(c) # new root approximation
 If f(c) = 0 < EPS || abs(a – c) < EPS then # solution found
  Output(c)
  return
 End if
 ITER ← ITER + 1 # increment step counter
EndWhile
```

d. Newton Raphson:

i. Description:

The Newton–Raphson method in one variable is implemented as follows:

The method starts with a function F defined over the real numbers x, the function's derivative F', and an initial guess $x0$ for a root of the function f. If the function satisfies the assumptions made in the derivation of the formula and the initial guess is close, then a better approximation x1 is

$$X1 = X0 - F(X0)/F'(X0)$$

Geometrically, (x1, 0) is the intersection of the x-axis and the tangent of the graph of F at $(x0, f(x0))$.

ii. Pros:

1. When the initial guess is near the root, the method usually converges.
2. The Method converges Quadratically when it doesn't diverge.
3. Knowledge of the multiplicity of the roots can speed the convergence for a non-1 value multiplicity.
4. A good algorithm can identify if the iterative process is diverging or not.

iii. Cons:

1. The calculation of the derivative of the function isn't always possible or straight forward.
2. There's no general convergence criteria, It depends on the nature and accuracy of the initial guess.
3. If the first derivative isn't well behaved near the initial guess the method may overshoot and diverge.
4. The Method may misbehave by going into a cycle or jump to another loop.
5. Slow convergence for roots of multiplicity other than 1.
6. Zero slope causes division by zero because of the derivative which needs to be handled separately in the code.

iv. Pseudo Code:

Input: Function f, initial guess a, tolerance EPS, maximum iterations MAX_ITER

Output: value not far by more than EPS from the root

ITER ← 1

While ITER ≤ MAX_ITER # limit iterations to prevent infinite loop

C ← a

If F'(c) = 0 then

  Output("Error division by zero")

  return

End if

$a \leftarrow c - F(c)/F'(c)$ # new root approximation

$If\ f(c) = 0 < EPS$ || $abs(a - c) < EPS$ then # solution found

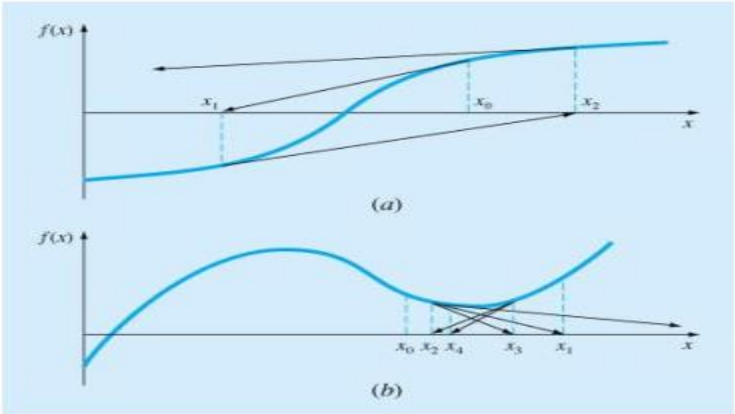
  Output(c)

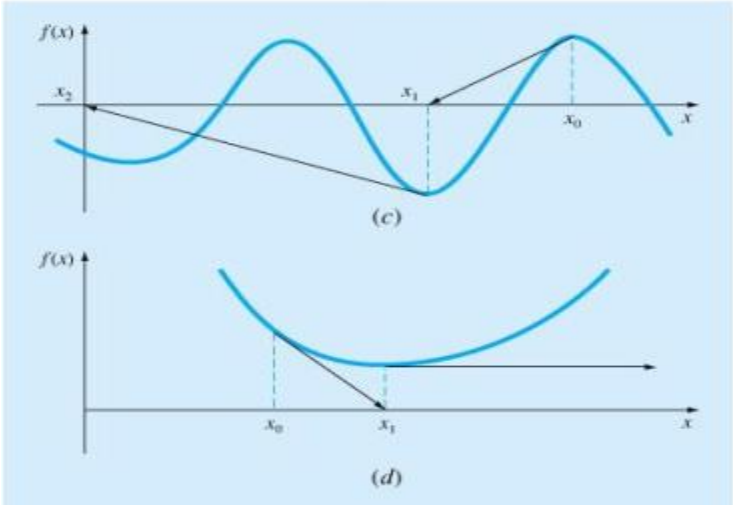
  return

 End if

 ITER ← ITER + 1 # increment step counter

End while





e. Secant:

i. Description:

is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function f. The secant method can be thought of as a finite difference approximation of Newton's method. However, the method was developed independently of Newton's method, and predates it by over 3,000 years.

ii. Pros:

1. Knowledge of the multiplicity of the roots can speed the convergence for a non-1 value multiplicity.
2. It's very fast compared to bisection and other slow algorithms.

iii. Cons:

1. There's no general convergence criteria, It depends on the nature and accuracy of the initial guess.
2. It requires two initial guesses instead of one in Newton Raphson.
3. It's slower than Newton Raphson.

iv. Pseudo Code:

Input: Function f, initial guess a and b, tolerance EPS, maximum iterations MAX_ITER

Output: value not far by more than EPS from the root

ITER ← 1

X(0) = a

X(1) = b

While ITER ≤ MAX_ITER # limit iterations to prevent infinite loop

$$X(ITER + 1) = X(ITER) - F(X(ITER)) * (X(ITER) - X(ITER - 1))/(F(X(ITER)) - F(X(ITER - 1))$$

If F'(c) = 0 then

Output("Error division by zero")

return

End if

new root approximation

$If\ f(c) = 0 < EPS$ || $abs(X(ITER + 1) - X(ITER)) < EPS$ then # solution found

Output(c)

return

End if

ITER ← ITER + 1 # increment step counter

End while

f. Birge Vieta:

i. Description:

This is an iterative method to find a real root of the **n**th degree polynomial equation.

ii. Pseudo Code:

Input: Coefficients of polynomial coeff, initial guess a, tolerance eps, maximum iterations MAX_ITER

Output: value not far by more than EPS from the root

ITER ← 1

```
x(1) ← a
condition ← true
cnt ← 1
order ← OrderOf(coeff)
abs_error ← nan
while cnt <= MAX_ITER
    b(order) ← a(order)
    c(order) ← a(order)
    for i ← order - 1 : −1 : 1
        b(i) ← a(i) + x(cnt) * b(i + 1)
        if i > 1
            c(i) ← b(i) + x(cnt) * c(i + 1)
        End if
    End loop
    if abs(b(1)) <= eps
        break
    end if
    if cnt > 1
        abs_eror ← abs(x(cnt + 1) − x(cnt))
    end if
    cnt ← cnt + 1
End while
print("The root is", x(cnt))
```

g. General Algorithm(Illinois):

i. Introduction:

In practical root-finding problems and the industry where the initial estimates of the roots are known or at least the range where you can find the initial guess there's a wide range of computationally efficient algorithms to use.

However, when using these algorithms there's two points to consider.

First Not every Method converges for example such methods as Secant, Newton Raphson will sometime fail to converge and may diverge, cycle itself, or get stuck. Second Although some Methods will always converge to a solution like Bisection but It's very slow compared to the number of the required iterations by other methods like Newton Raphson, Secant. That was the reason of using Regula Falsi.

We consider the equation

F(x) = 0

We'll start with two approximation of the root where their function values must have an opposite sign to be able to continue. then we compute $x(i + 1)$. Then for the next iteration we will use $x(i + 1)$ and either $x(i)$ or $x(i - 1)$ whichever

gives us a valid interval containing an odd number of roots. The iterative process will continue until some criterion is satisfied.

For example, reaching a certain small value of the error or a certain previously specified number of iterations.

Although Regula Falsi may seem to be an excellent Method but it isn't optimal due to one main draw back. In case of convex or concave curves near the x-axis Regular Falsi starts to get slow reducing its asymptotic convergence to Linear convergence and may in some extreme cases get stuck and stop changing its value at all.

We'll introduce a modification to handle such case.

ii. Modification:

The first solution that comes to mind to handle the slow convergence might be using Bisection, when detecting that the change in the absolute approximate error reaches a certain limit so we can instead switch for the Bisection method for 3 or 4 Iterations until maybe the part covering the Convex or concave part may have passed.

Such solution is straight forward but It has a binding that when trying to escape slow convergence of Bisection, we still used it again and we will be bound by the logarithmic complexity.

1. Detection:

We will detect the convex or concave curve in the situations required as a non-changing side for the interval where if $x(i+1)$ doesn't change for two consecutive iteration then we may consider using the modification later specified in this section.

2. Modification:

Assuming that we have two initial Guesses a and b.

C: the new approximation of the root.

```
if F(c) * Fb < 0 then
(a, Fa) := (b,Fb)
end if
if F(c) * Fb > 0 then
(a, Fa) := (a, Fa/2)
end if
(b, Fb) := (c, F(c))
```

The above pseudo code describes the iterative process.

This method was used because Regula Falsi is just an interpolation and in case of convex or concave, we can identify the side which doesn't change and by dividing it by two we balance the ratio of the cut in x-axis and enhance the converging speed.

In the first Picture is the normal Regula Falsi and in the second picture we can see that the change of the value of F(b) resulted in decreasing the slope of the line and speeding the algorithm.
It can be noted that the first step will be the same but in case of the second step, the Illinois got an approximation which was attained in the third iteration in the normal Regula Falsi.

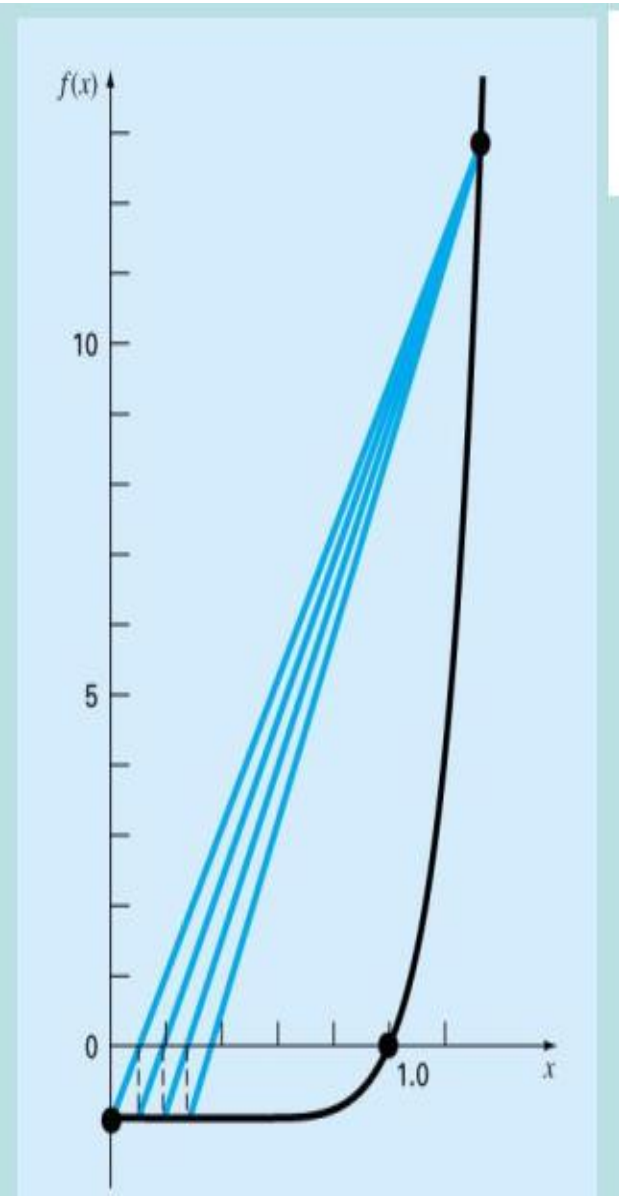


Figure 1(Regula Falsi)

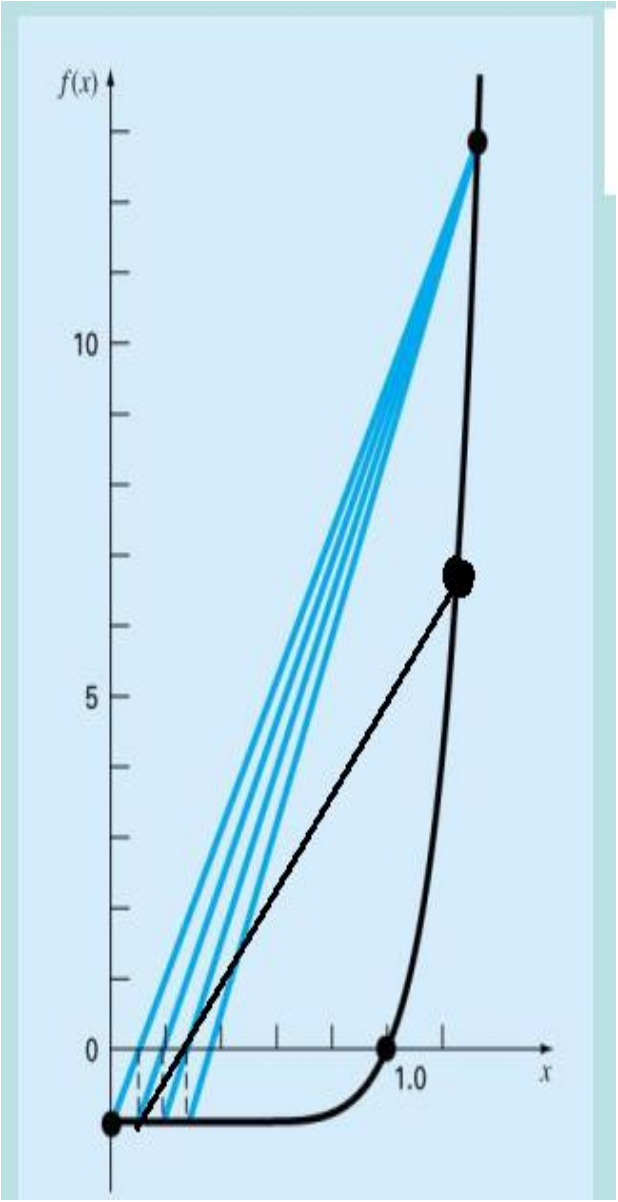


Figure 2(Illinois)

iii. Pseudo code:

There was no data structure required in the Algorithm other than normal data value storage represented in variables used to store the value

Input: Function F, initial Guesses a and b, the maximum number of iterations maxIterations, and tolerance EPS.
output: a value not far by more than EPS from the Root.

```
if F(a) * F(b) > 0.0
error('Function has same sign at end points');
return
end
Fa ← F(a)
Fb ← F(b)
for i = 1 : maxIterations
  appRoot ← b – Fb * (b – a) / (Fb – Fa)
  appRoot_val ← F(appRoot)
  if i > 1
    abs_error ← abs((appRoot – prev_root))
  end
  if xroot_val == 0.0
    break;
  elseif F(appRoot) * F(a) < 0
    b ← a
    b_val ← Fa
    a ← appRoot;
    Fa ← F(appRoot)
  else
    a ← appRoot
    Fa ← F(appRoot)
    Fb ← Fb / 2.0
  end
  if i > 1 && abs_error < eps
    break;
  end
  prev_root ← appRoot
end
print("The root is", appRoot)
```

iv. Difference:

These functions have a sloppy curvature that slows down the Regular Falsi on the interval of [0.01,1] with a tolerance of 10^{-6}

$$F(x) = x^2 - (1 - x)^n$$

n\Method	Bisection	Regular Falsi	Illinois
2	**1**	**1**	**1**
5	**20**	**16**	**6**
10	**20**	**32**	**9**
15	**20**	**47**	**10**

$$F(x) = (n * x - 1)/((n - 1) * x)$$

n\Method	Bisection	Regular Falsi	Illinois
2	**20**	**457**	**13**
5	**20**	**208**	**13**
10	**20**	**110**	**13**
15	**20**	**74**	**12**

2. Problematic Functions:

1. Regular Falsi:

One of the biggest disadvantages of the Regular Falsi is that in the Convex and concave curves near the x-axis its convergence tends to become slower and also may even get stuck and the method won't converge at all.
There're two solutions to this problem:

a. We can try to detect when the convergence become slower and then use bisection for 3~5 iterations until the Regular Falsi method can converge again.

b. We can use the Illinois modification where the value of the non-changing end of the interval can be divided by two to increase the convergence speed that it may even has faster convergence than the normal Regular Falsi in normal cases.

2. Fixed point:

The main problems with the Fixed Point Method.

a. The function G(x) may converge very slow or fast or may even diverge depending on the nature of the initial guess and the function G(x).
The solution of this problem is to change the function G(x) or try to use different initial guesses and we can use the rules describing the rates of convergence for the algorithm for this G(x) to avoid using the method on a G(x) that may diverge.

3. Newton Raphson:

a. One main problem of Newton Raphson is that it may diverge if the initial guess isn't near a root for this function or may converge slowly. By calculating the rate of convergence successively a good software can identify if the algorithm is diverging or not.
b. It may go into a cycle iterating over the same set of points. To avoid this pitfall, we can save the previous points in an array or any data structure and in case of repeating a previous pattern we identify the current state as a loop.
c. Local maximum or minimum can cause Oscillation of the Newton Raphson Method around the local maximum or minimum. To avoid this pitfall, we can try to pick the guess as far as possible from the local maximum or minimum and as near as possible from the expected root.
d. A Local minimum point would have a derivative function equaling zero which would cause division by zero, so we should check the value of the derivative first before calculating the next approximation of the root to know when to stop before causing division by zero error.

e. Newton Raphson doesn't converge Quadratically when it converges where the root is a multiple root. We can try to guess or specify the multiplicity of the root and use modified Newton Raphson which will speed the algorithm.

4. Secant:

a. It requires two guesses instead of one guess as in Newton Raphson. To overcome this problem, we can use the modified secant method which will require only one initial guess if the delta is selected properly because if it's too small it can cause subtractive cancelation in the denominator and if it's too big it can cause the method to slow or even diverge in the worst case scenario.

3. Sample Runs and Analytics:

1- Bisection

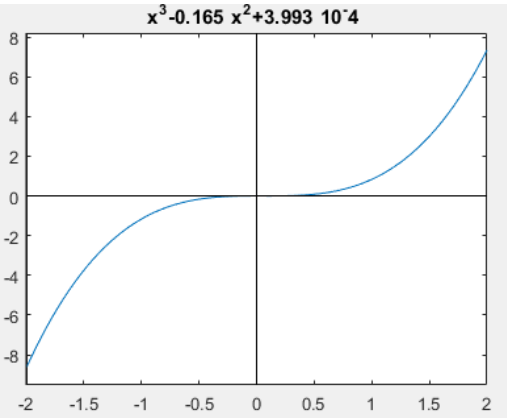
Inputs:

- $x^3 - 0.165 * x^2 + 3.993 * 10^{-4} = 0$
- Initial guesses: $\{0,\ 0.11\}$
- Precision: 0.00001
- Max iterations: 50

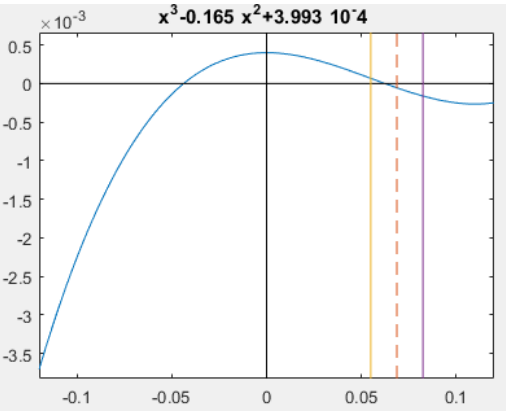
Outputs:

- Root:0.0623785
- Time: 0.18179
- Iterations: 14
- Precision: $-8.54208 * 10^{-9}$

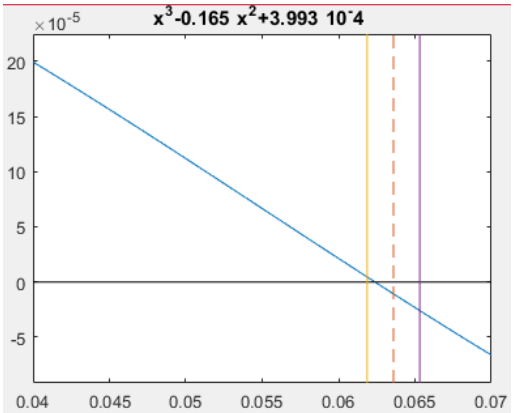
Function plotting & step simulation.



Function plotting



Bisection 3rd iteration



Bisection 6th iteration

2- False-Position

- **First Test Case**

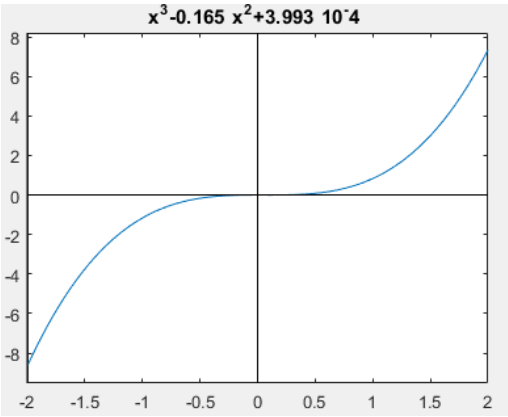
Inputs:

- $x^3 - 0.165 * x^2 + 3.993 * 10^{-4} = 0$
- Initial guesses: $\{0,\ 0.11\}$
- Precision: 0.00001
- Max iterations: 50

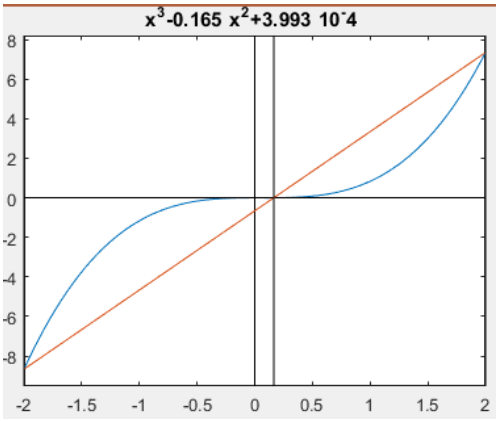
Outputs:

- Root: 0.0623776
- Time: 0.0694955
- Iterations: 5
- Precision: $3.74479 * 10^{-8}$

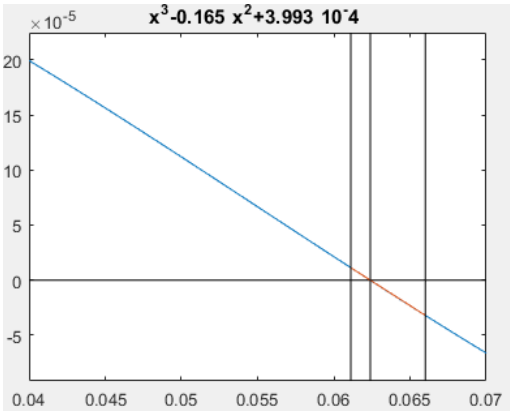
Function plotting & step simulation.



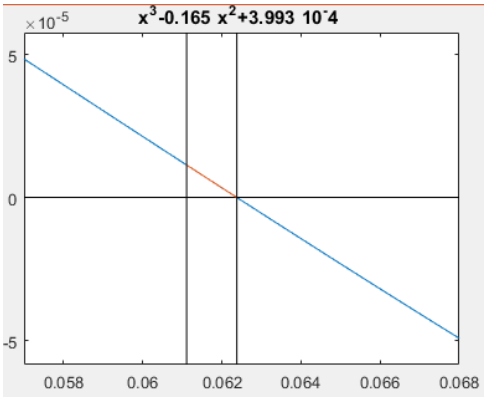
Function plotting



False-Position 1st iteration



False-Position 3th iteration



False-Position 5th (Last) iteration

- **Second Test Case**

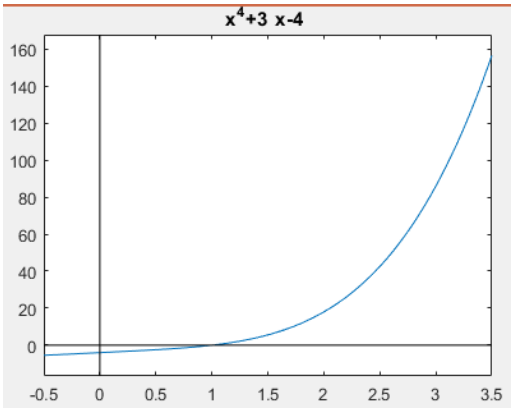
Inputs:

- $x^3 - 0.165 * x^2 + 3.993 * 10^{-4} = 0$
- Initial guesses: $\{0,\ 0.11\}$
- Precision: 0.00001
- Max iterations: 50

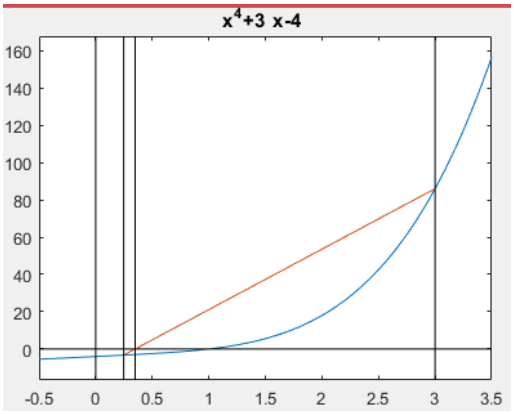
Outputs:

- Root: 0.999793
- Time: 1.94616
- Iterations: 50
- Precision: $4.03109 * 10^{-5}$

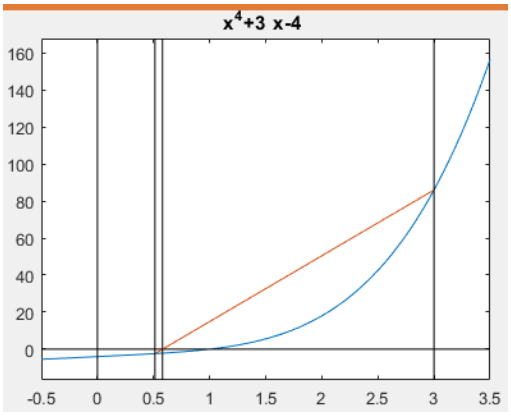
Function plotting & step simulation.



Function plotting



False-Position 3th iteration



False-Position 6th iteration

Node: The difference between the plot for the 3rd and the 6th is very small because this method is very slow for this test case.

3- Fixed Point

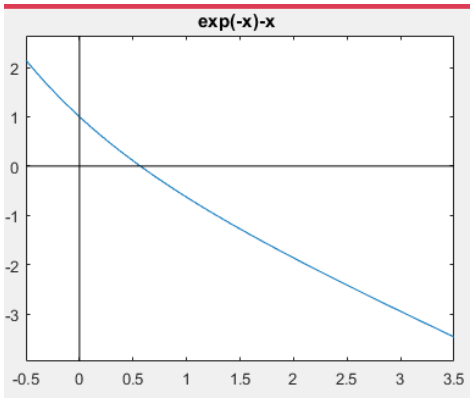
Inputs:

- $e^{-x} - x = 0$
- Initial guesses: $\{0\}$
- $G(x) = e^{-x}$
- Precision: 0.00001
- Max iterations: 50

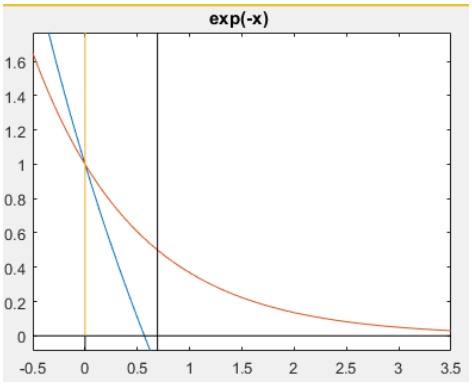
Outputs:

- Root: 0.567141
- Time: 1.95749
- Iterations: 22
- Precision: $3.3189 * 10^{-6}$

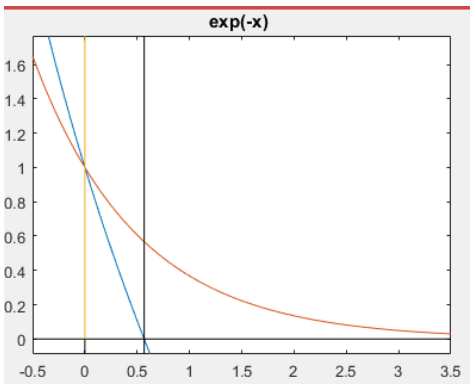
Function plotting & step simulation.



Function Plotting



3rd iteration plotting



12th iteration plotting

4- Newton-Raphson

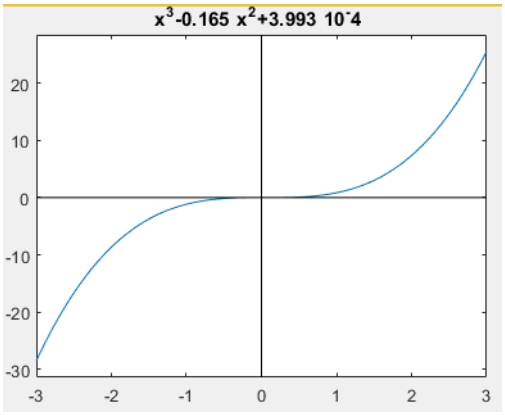
Inputs:

- Input equation: $x^3 - 0.165 * x^2 + 3.993 * 10^{-4} = 0$
- Initial guesses: $\{0.05\}$
- Precision: 0.00001
- Max iterations: 50

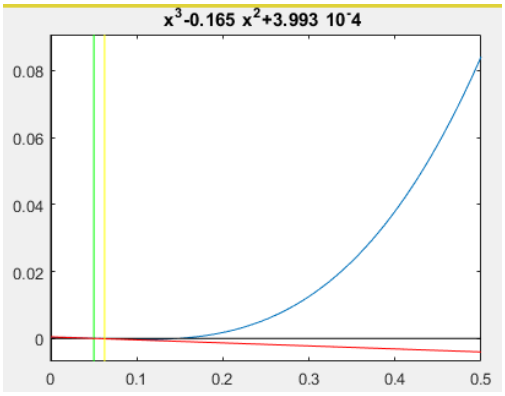
Outputs:

- Root: 0.0623776
- Time: 0.0633101
- Iterations: 3
- Precision: $5.24341 * 10^{-19}$

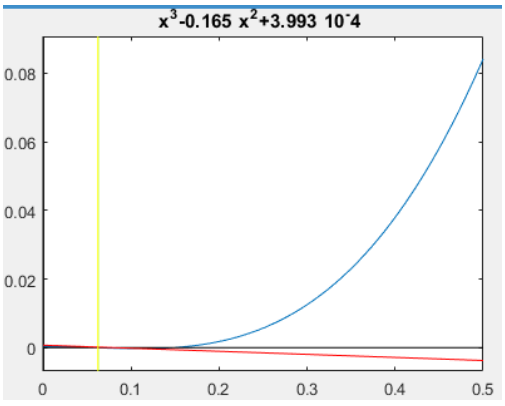
Function plotting & step simulation.



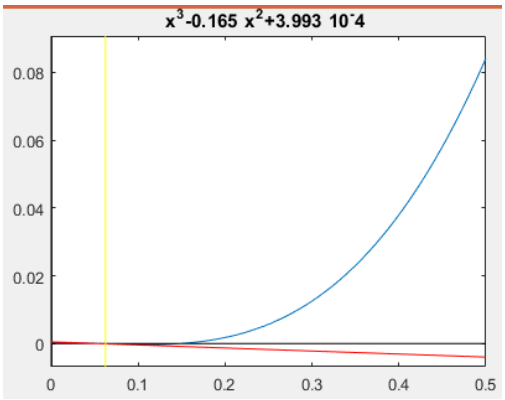
Function plotting



First iteration



Second iteration



Third (last) iteration

5- Secant Method

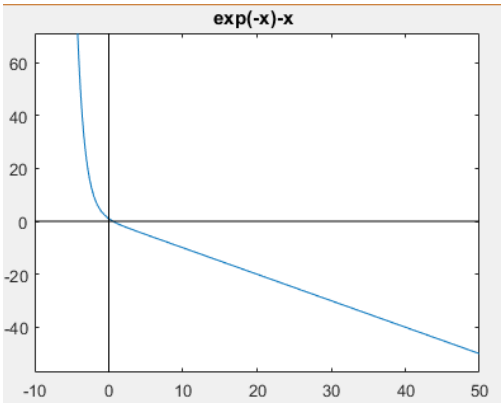
Inputs:

- Input equation: $e^{-x} - x = 0$
- Initial guesses: $\{0,\ 1\}$
- Precision: 0.00001
- Max iterations: 50

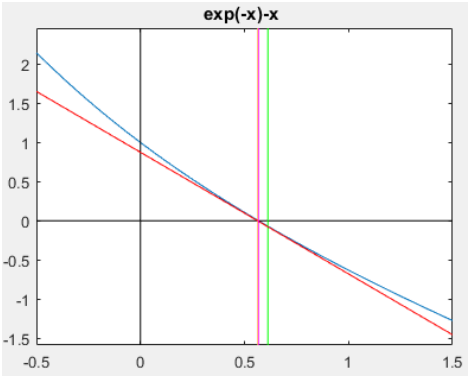
Outputs:

- Root: 0.567143
- Time: 0.609435
- Iterations: 4
- Precision: $-2.53802 * 10^{-8}$

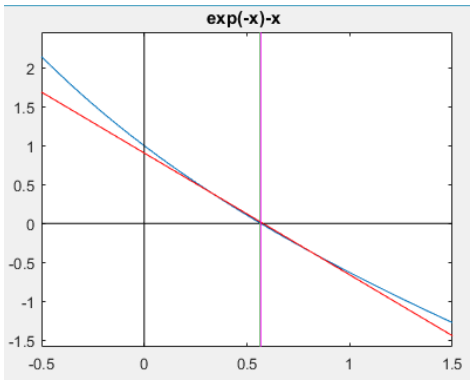
Function plotting & step simulation.



Function plotting



3rd iteration plotting



4rd (Last) iteration plotting

6- Birge Vieta:

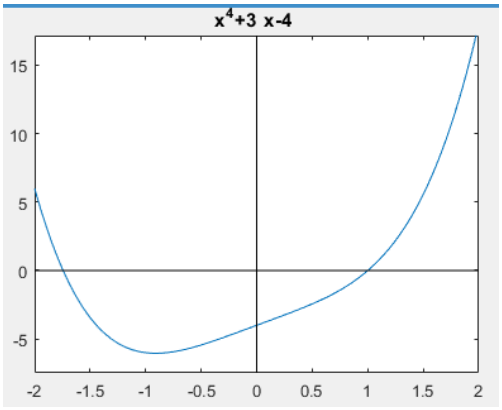
Inputs:

- Input equation: $x^4 + 3 * x - 4 = 0$
- Initial guesses: $\{0\}$
- Precision: 0.00001
- Max iterations: 50

Outputs:

- Root: 1
- Time: 0.040305
- Iterations: 6
- Precision: 0

Function plotting.



Function plotting

7- All Methods:

- **Test 1**

  Inputs:

  Equation: $e^{-x} - x = 0$

  Bisection interval: $\{0, 3\}$

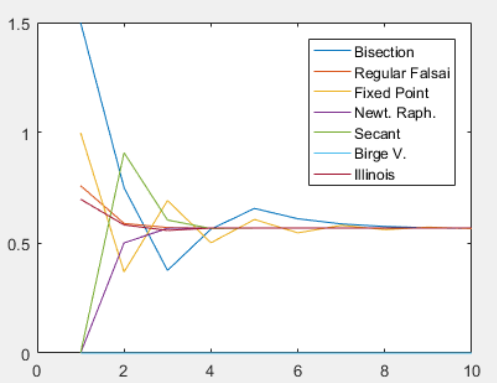
  False-Position interval: $\{0, 3\}$

  $G(x) = e^{-x}$

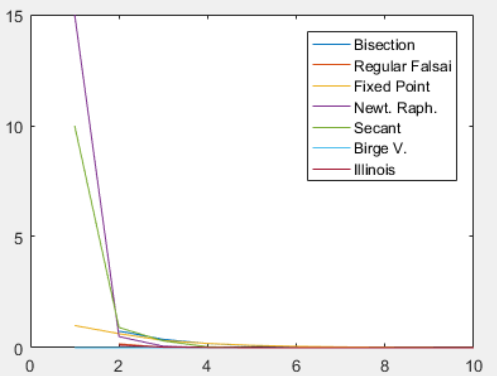
  Fixed point initial guess = 0

  Newton's Raphson initial guess = $\{15\}$

  Secant initial guess = $\{15, 10\}$



Root/iterations



Absolute error/iterations

- **Test 2**

Inputs:

Equation: $\mathrm{x}^4 + 3 * \mathrm{x} - 4 = 0$

Bisection interval: $\{0, 3\}$

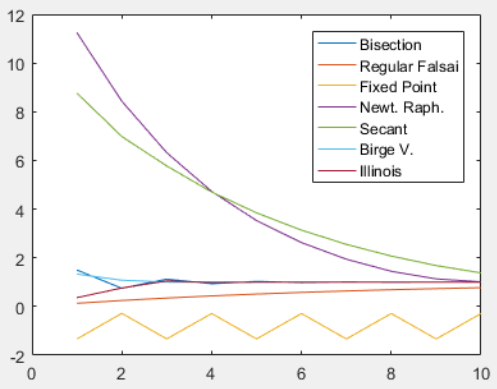
False-Position interval: $\{0, 3\}$

$G(x) = \dfrac{\mathrm{x}^4 - 4}{3}$

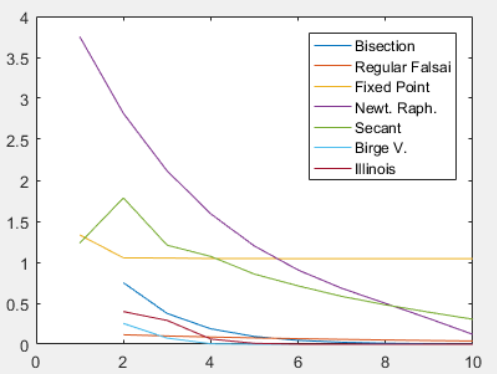
Fixed point initial guess = 0

Newton's Raphson initial guess = $\{15\}$

Secant initial guess = $\{15, 10\}$



Root/iterations



Absolute error/iterations

Note: the function $G(x) = \frac{\mathrm{x}^4 - 4}{3}$ is a bad $g(x)$ because in this case it will diverge.