Middle East Technical University
Department of Mechanical Engineering
ME310 Numerical Methods
Programming Assignment 1

Prepared by:
Bertan Özbay 2378545 Section 4
Yunus Çınar Kızıltepe 2378362 Section 1

Introduction

In this project, main aim was to calculate same functions in each functions own respective boundary by using different calculation techniques namely numerical methods. These numerical methods can be named as; Bisection method, False-position method, Secant method, Newton-Raphson method, and a technique that is derived from a polynomial that will be called as Polynomial Method.

In this assignment, our goal was to apply different numerical methods, see them in action and compare their performance through their produced error values and the iterations they took to reach the root with tolerable accuracy. We also derived a new method, which was named the polynomial method, by applying a polynomial instead of a line in the false position method. The five methods that we studied in this assignment are Newton-Rhapson Method, False-Position Method, Bisection Method, Polynomial Method, and Secant Method.

The Methods

False-Position Method

In the false-position method, we are basically creating a line to estimate the root, which gets more accurate with each iteration. Here the formula we used was as in the book:

$$x_r = x_u \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)}$$

Polynomial Method

The polynomial method is similar to the false position method, however here we are using a polynomial instead of a line to be better able to estimate the root of the function. As explained in the assignment, we calculate an intermediate point $x_i = (x_l + x_u)/2$ which we then use to create a second order polynomial passing through that point.

$$p(x) = a(x - x_i)^2 + b(x - x_i) + c,$$

$$f(x_l) = a(x_l - x_i)^2 + b(x_l - x_i) + c$$
$$f(x_u) = a(x_u - x_i)^2 + b(x_u - x_i) + c\,.$$
$$f(x_i) = c$$

Where the coefficients are:

$$a = \frac{f(x_l) - f(x_i)}{(x_l - x_i)(x_l - x_u)} + \frac{f(x_i) - f(x_u)}{(x_u - x_i)(x_l - x_u)}$$
$$b = \frac{(f(x_l) - f(x_i))(x_i - x_u)}{(x_l - x_i)(x_l - x_u)} - \frac{(f(x_i) - f(x_u))(x_l - x_i)}{(x_u - x_i)(x_l - x_u)}\,.$$
$$c = f(x_i)$$

and thus the new root estimate is:

$$x_r = x_i - \frac{2c}{b + sign(b)\sqrt{b^2 - 4ac}}$$

Bisection Method

In the bisection method, we simply take the middle value of the interval.
$x_r = (x_l + x_u)/2$
Then evaluate the new root through a series of sign evaluations. Simply replacing the boundary with the same sign as the root.

Secant Method

For the secant method we will use the equation

$$x_j = x_i - \frac{f(x_i) * (x_k - x_i)}{f(x_k) - f(x_i)}$$

where

$$j = i + 1 = k + 2$$

Id Est:

$$x_2 = x_1 - \frac{f(x_1) * (x_0 - x_1)}{f(x_0) - f(x_1)}$$

Newton-Rhapson Method

We will use the equation:

$$x_j = x_i - \frac{f(x_i)}{f'(x_i)}$$

where

$$j = i + 1$$

For each iteration given f, f' and the first guess initial x_i is defined.

For this example the function:

$$f(x) = \mathrm{e}^{-x} - x$$

will be used with the following derivative:

$$f'(x) = -\mathrm{e}^{-x} - 1$$

Error Calculation

In this assignment we used a simple method of relative approximate error calculation given by the formula:

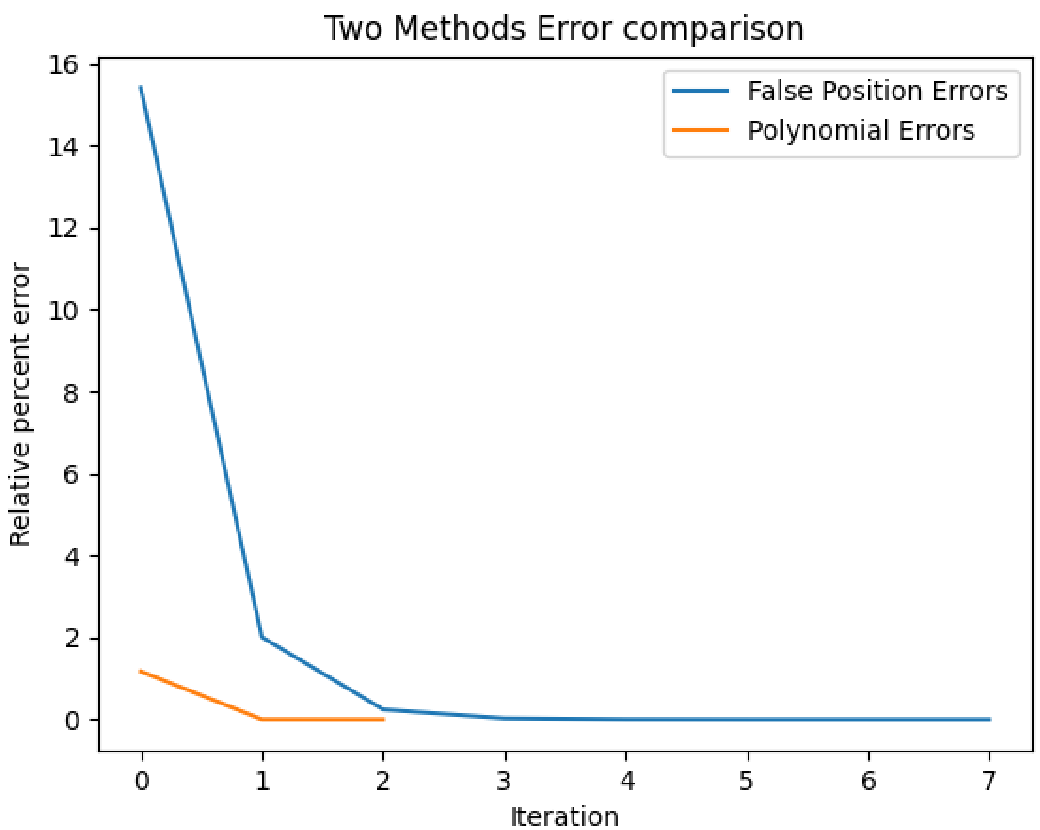
$$E_a = \frac{x_{i+1} - x_i}{x_{i+1}}$$

Graphs

For the graphs, we used a python graphing library called matplotlib. During operation, our program records root and error values of the functions to their designated arrays and after successful execution displays graphs for each method.

Discussion

Generally, our methods have reached a root for most of the functions. During our development phase, we encountered some errors such as some methods not working for some functions or problems with the domain of the functions. We have tried to solve them as best as we could, implementing warnings, error signs and checks for the methods. We have observed for some functions and methods, the function may converge very slowly, or it could converge to an incorrect root. For others we sometimes could not get convergence fast enough simply because the root value kept getting small and the error did not decrease fast enough.
Regarding our new polynomial method, we have observed that for most functions, it converges much faster and therefore shows a greater performance compared to its counterpart, the false-position method. A second-degree polynomial usually fits better with most functions than a line does, so this result is most natural. This can be observed in our graphs at the appendix section.



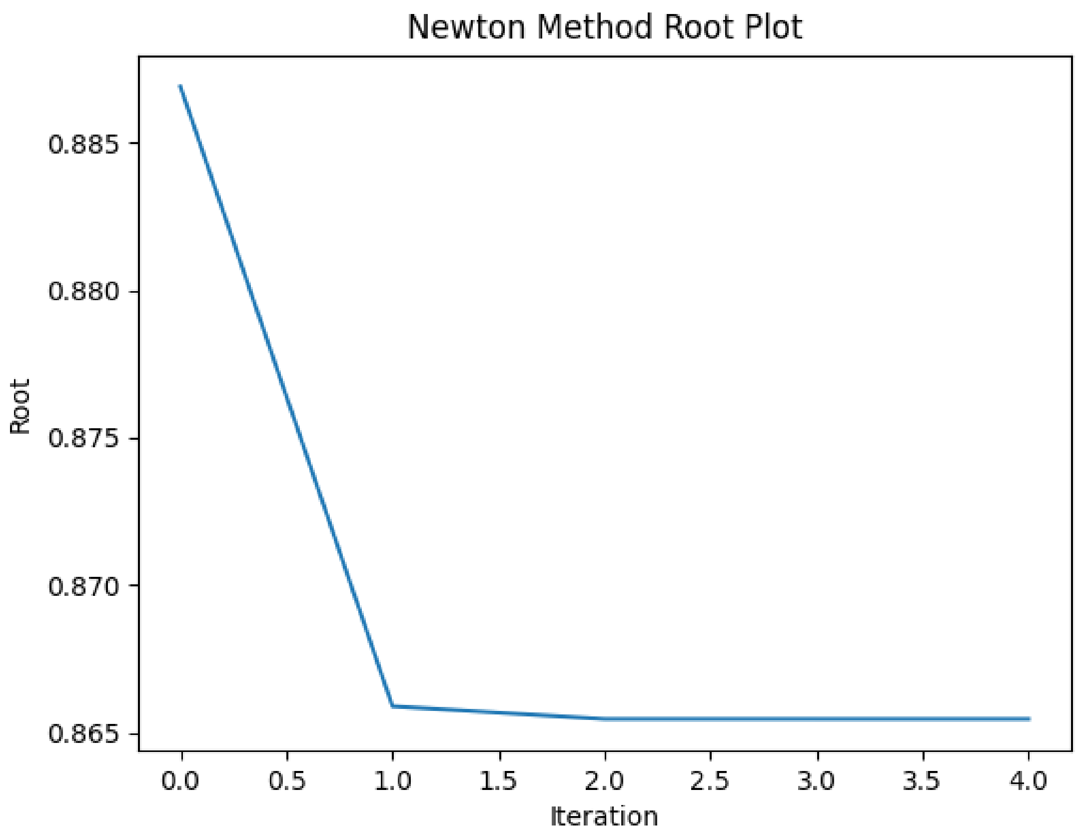
Conclusion

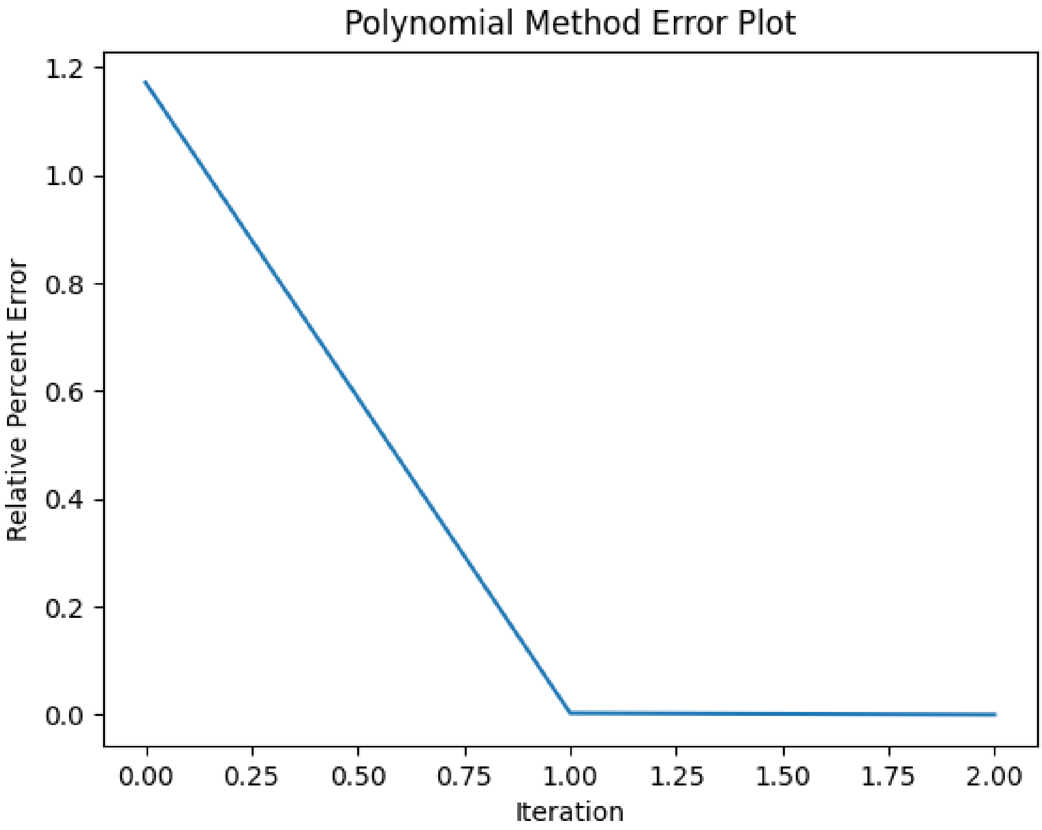
In this assignment we had a chance to see a wide range of different numerical methods in action, while also modifying an existing method as we tried to improve it. If we were to check the graphs, it is evident that the new polynomial method performed better for the given function. However, we have also been reminded that some methods work well with some functions whereas some methods may not be suitable at all. After all, this is the reason for the existence of this course and the existence of different numerical methods for different cases.

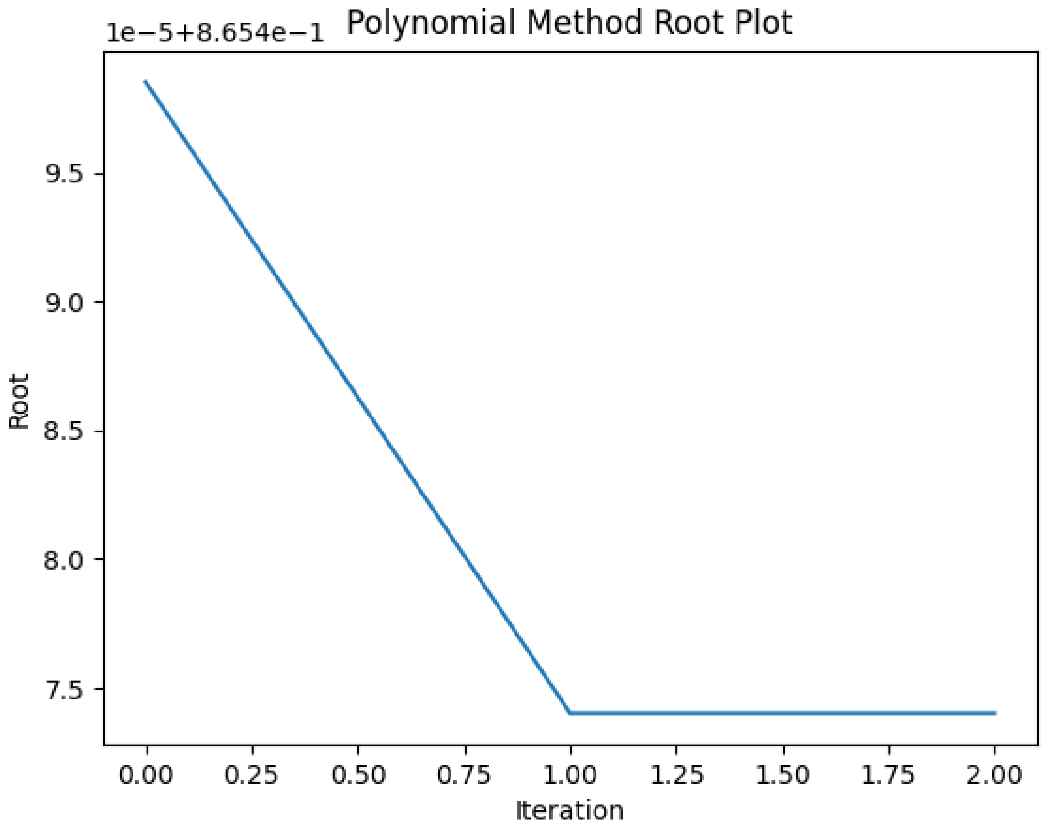
Appendix

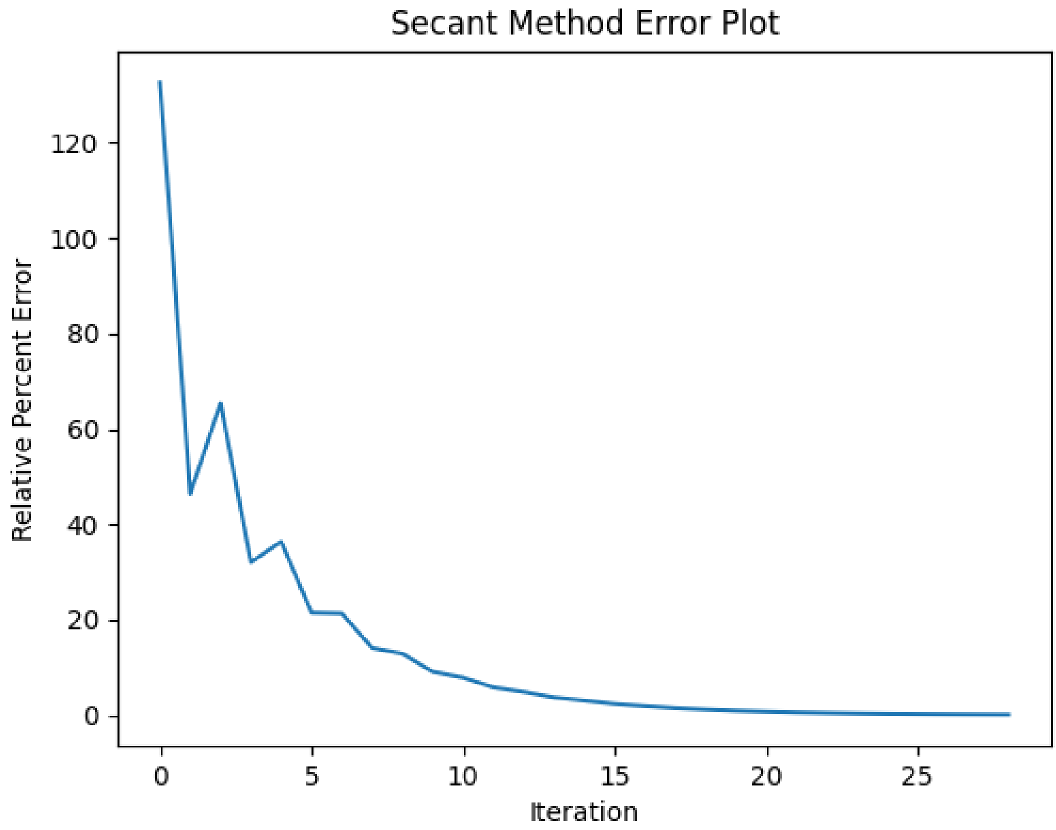
Our Graphs

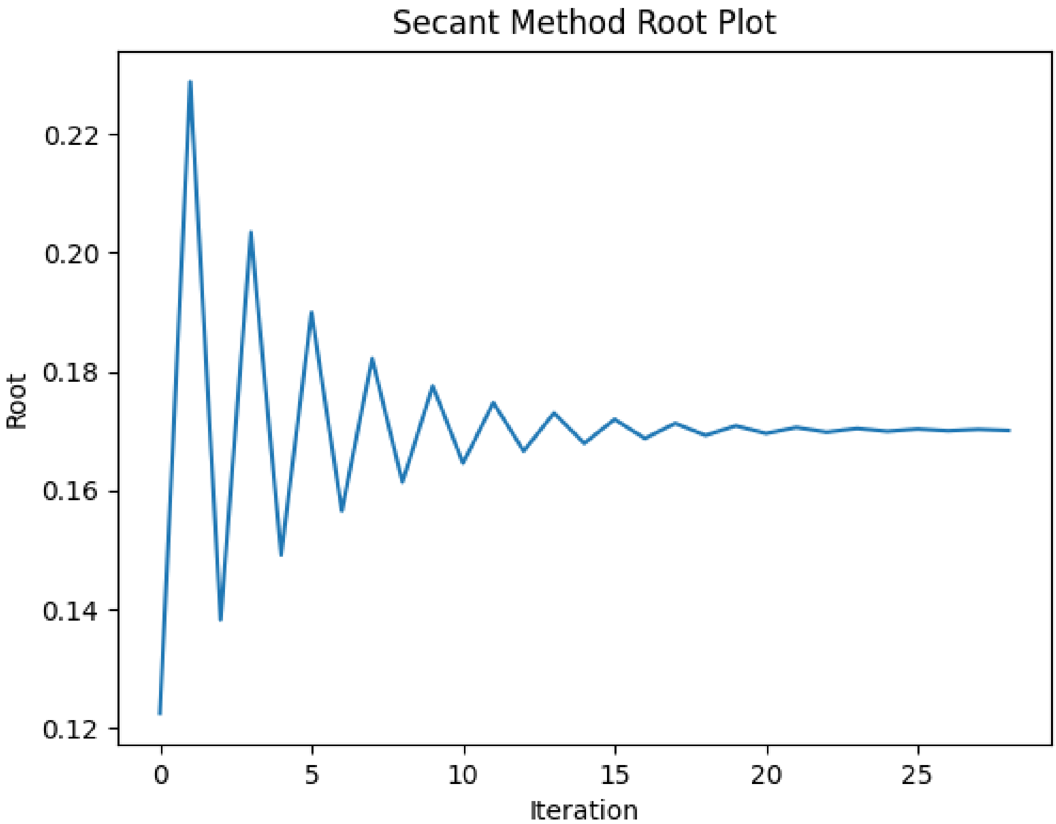
To use as an example for the graphs and output files, we used the function $f(x) = cos(x) - x^3$ within the bounds of $[0.1, 1]$

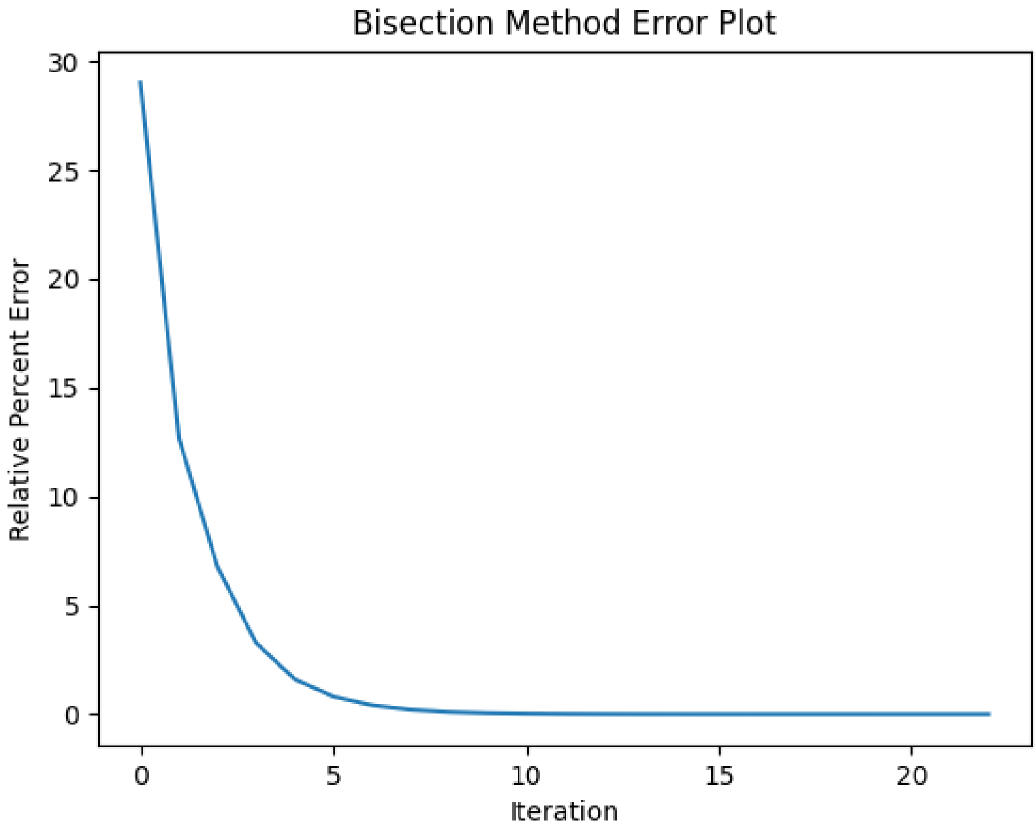


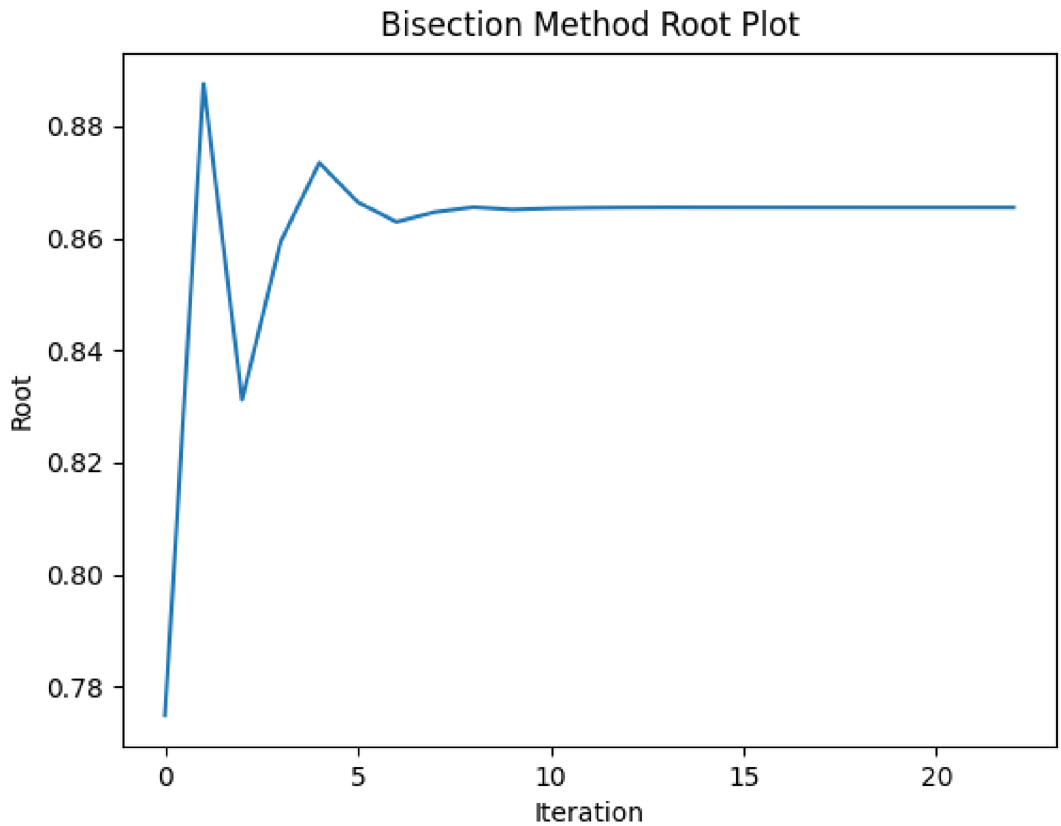

Polynomial Method Error Plot
Relative Percent Error
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
Iteration

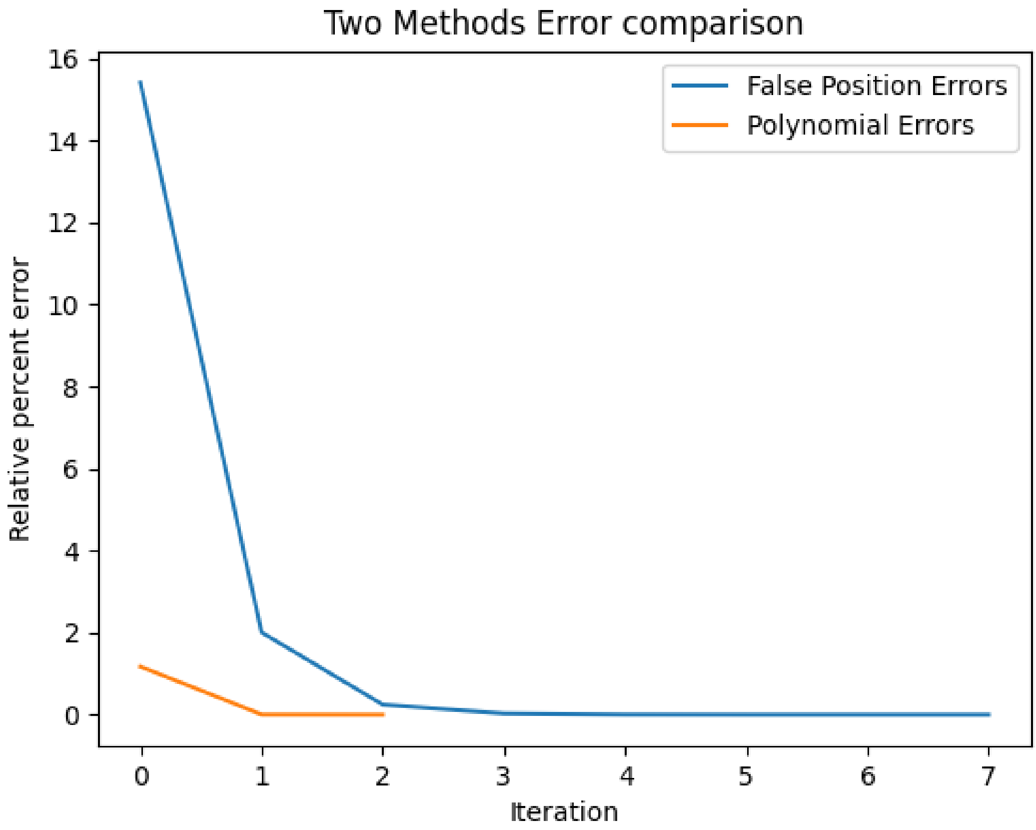

Polynomial Method Root Plot
1e−5+8.654e−1
Root
9.5
9.0
8.5
8.0
7.5
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
Iteration

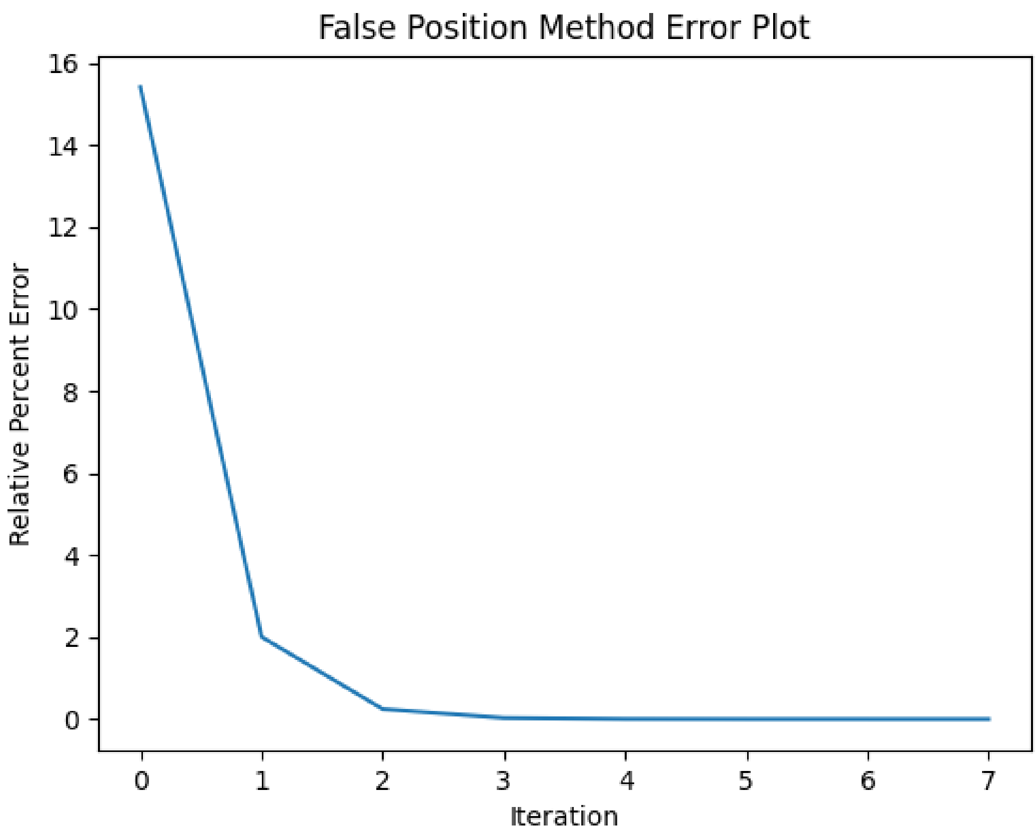

Secant Method Error Plot
Relative Percent Error
Iteration
0
20
40
60
80
100
120
0
5
10
15
20
25

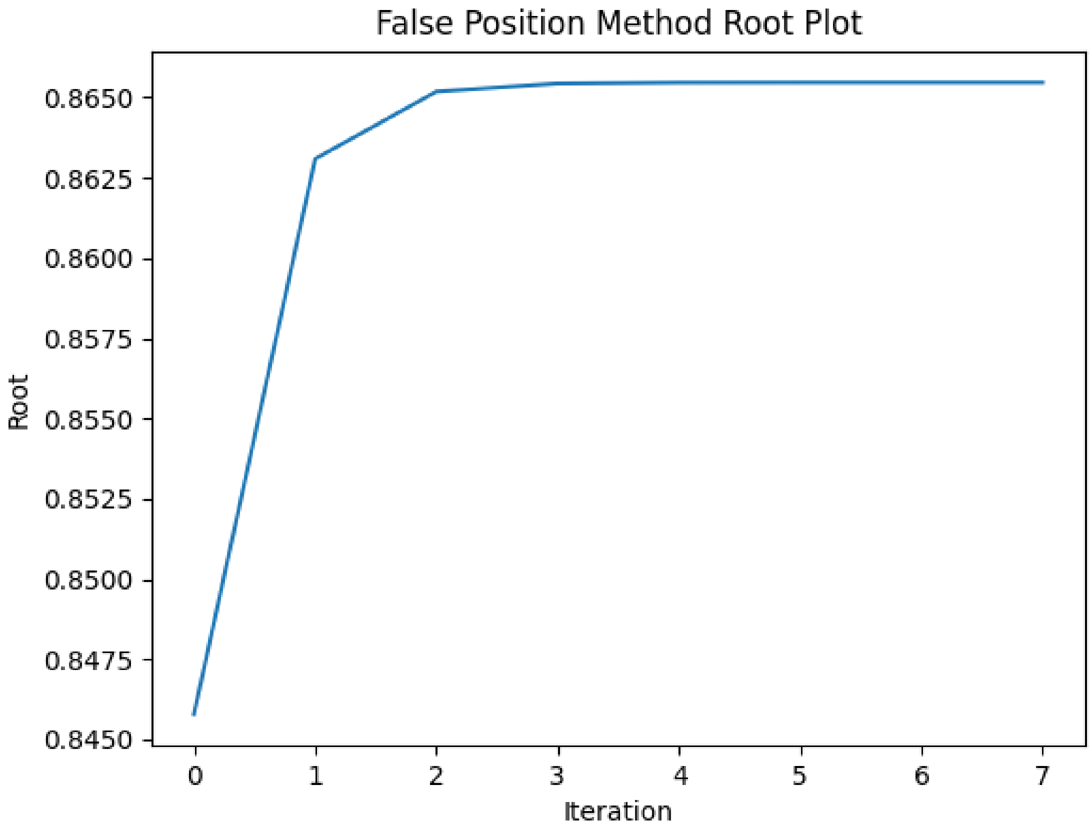

Secant Method Root Plot
Root
Iteration
0.12
0.14
0.16
0.18
0.20
0.22
0
5
10
15
20
25

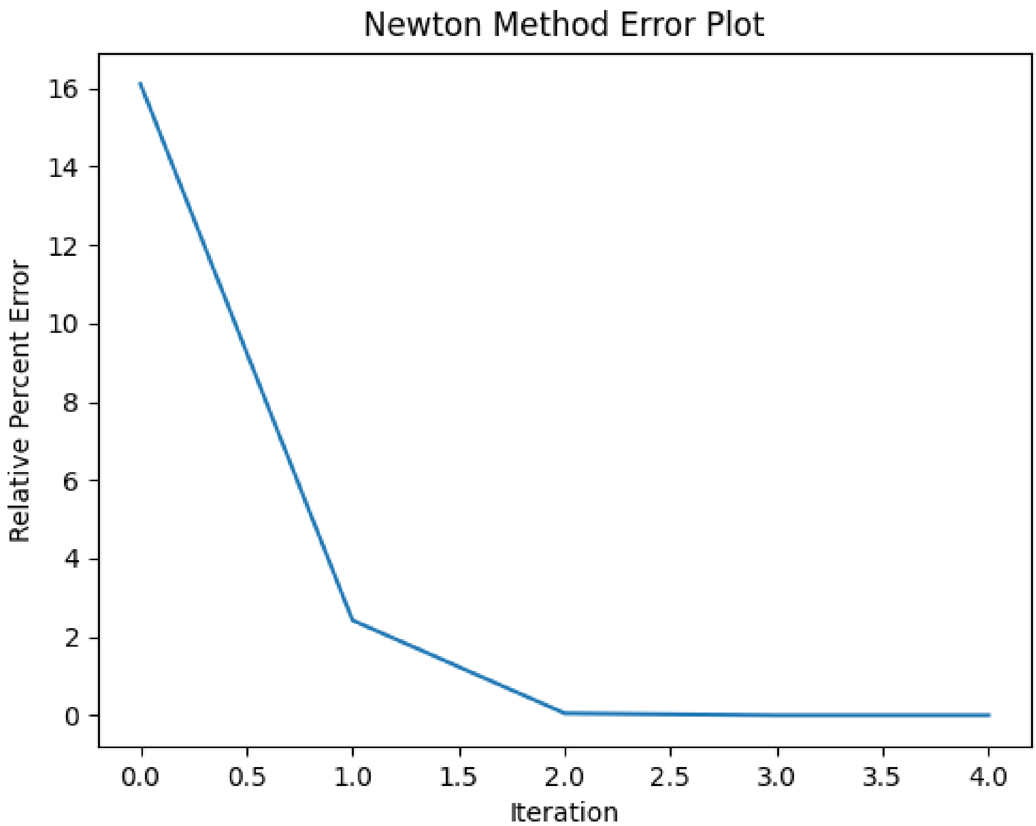

Bisection Method Error Plot
Relative Percent Error
Iteration
0
5
10
15
20
25
30


Bisection Method Root Plot
Root
Iteration
0.78
0.80
0.82
0.84
0.86
0.88
0
5
10
15
20


Two Methods Error comparison
False Position Errors
Polynomial Errors
Relative percent error
Iteration
0
2
4
6
8
10
12
14
16
0
1
2
3
4
5
6
7


False Position Method Error Plot
Relative Percent Error
Iteration
0
2
4
6
8
10
12
14
16
0
1
2
3
4
5
6
7


False Position Method Root Plot
0.8650
0.8625
0.8600
0.8575
0.8550
0.8525
0.8500
0.8475
0.8450
Root
0
1
2
3
4
5
6
7
Iteration


Newton Method Error Plot
16
14
12
10
8
6
4
2
0
Relative Percent Error
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Iteration

Our Code

```
#import functions
from hashlib import new
from tkinter.filedialog import Open
import f as f
import fp as fp
import math
import os
import matplotlib.pyplot as plt
# Structure for input.txt
# x_low
# x_up
# error_set
# iteration_max


# Reading input
 #Open the input file
input_file = open("input.txt", "r")

#read line by line
x_low = float(input_file.readline())
x_up = float(input_file.readline())
error_set = float(input_file.readline())
iteration_max = float(input_file.readline())

#debug
print(x_low, x_up, error_set, iteration_max)

#The arrays here will be used for easy and automatic plotting.
newton_Errors = []
secant_Errors = []
polynomial_Errors = []
bisection_Errors = []
```

```
falsepos_Errors = []
newton_Roots = []
secant_Roots = []
polynomial_Roots = []
bisection_Roots = []
falsepos_Roots = []

# Define the sign function to get sign of b
def sign(b):
  if b >= 0:
    return 1
  else:
    return -1

# Error Calculation
def calculate_error (x_current, x_previous):
  E_a = (x_current - x_previous)/(x_current)
  E_a = E_a * 100
  return abs(E_a)

# The Methods

def new_bisection(xl, xu, it_Max, error_Goal, xprev = 500, i = 0,
bisectionOutput = 0 ):

  #Initialization, open output file
  if i == 0:
    bisectionOutput = open("output_bisection.txt", "w")

  #Termination, close output file
  if it_Max == i or it_Max < i :
    print("Bisection Method, Max iteration reached")
    print("Last estimate= " + str(xprev) + "Apprx. % Relative Err: " +
str(bisection_Errors[-1]) + "# of iterations: " + str(i) + "f(xr) = " +
str(f.f(xprev)))
    bisectionOutput.close() #Close output file
    return 0 #Terminate function

  #Advance iteration
  i = i+1

  #Store function values to decrease function evaluations
  fxl = f.f(xl)
```

```
  fxu = f.f(xu)

  #Check if the bounding input values give different sign
  if sign(fxl) == sign(fxu):
    print("xl is " + str(xl))
    print("fxl is " + str(fxl))
    print("xu is " + str(xu))
    print("fxu is " + str(fxu))
    bisectionOutput.write(" No sign change between point, program 
terminated")
    raise Exception(" No sign change between the points ")

  #Calculate mid value
  xi = (xl + xu)/2
  fxi = f.f(xi)


  #Error not calculated for the first iteration
  if i == 1:
    approxError = "---"
  else:#Calculate error
    approxError = calculate_error(xi,xprev)
    bisection_Errors.append(approxError)
    bisection_Roots.append(xi)
    if approxError < error_Goal: #Stop process if error goal is reached
      print("Bisection Method, Error tolerance goal reached")
      print("Last estimate= " + str(xprev) + "Apprx. % Relative Err: " 
+ str(bisection_Errors[-1]) + "# of iterations: " + str(i) + "f(xr) = " 
+ str(f.f(xprev)))
      bisectionOutput.write(str(i) + " " + str(xi) + " " + str(fxi) + " 
" + str(approxError) + "\n")
      return 0

  #Record mid value for error calculation in the next iteration
  xprev = xi
  #debug
  #print("xi is " + str(xi))
  #print("xl was " + str(xl))
  #print("xu was " + str(xu))

  #Replace xi with whichever bound that gives the same sign
  if sign(fxi) == sign(fxl):
    xl = xi
```

```
  elif sign(fxi) == sign(fxu):
    xu = xi

  #Debug
  #print("xl is " + str(xl))
  #print("xu is " + str(xu))

  #Write to output file
  bisectionOutput.write(str(i) + " " + str(xi) + " " + str(fxi) + " " +
str(approxError) + "\n")

  #Perform Recursive function call
  new_bisection(xl,xu,it_Max,error_Goal,xprev,i,bisectionOutput)

########
#
#
# False Position

def falsePosition(xl, xu, it_Max, error_Goal, xprev = 500, i = 0,
falsePositionOutput = 0 ):

  #Initialization, open output file
  if i == 0:
    falsePositionOutput = open("output_falseposition.txt", "w")

  #Termination, close output file
  if it_Max == i or it_Max < i :
    print("False Position Method, Max iteration reached")
    print("Last estimate= " + str(xprev) + "Apprx. % Relative Err: " +
str(falsepos_Errors[-1]) + "# of iterations: " + str(i) + "f(xr) = " +
str(f.f(xprev)))
    falsePositionOutput.close() #Close output file
    return 0 #Terminate function

  #Advance iteration
  i = i+1

  #Store values to decrease the number of function calls
  fxl = f.f(xl)
  fxu = f.f(xu)

  #Check if the bounding input values give different sign
```

```
  if sign(fxl) == sign(fxu):
    print("xl is " + str(xl))
    print("fxl is " + str(fxl))
    print("xu is " + str(xu))
    print("fxu is " + str(fxu))
    falsePositionOutput.write(" No sign change between point, program
terminated")
    raise Exception(" No sign change between the points ")

  #Calculate xr
  xr = xu - ((fxu*(xl-xu))/(fxl - fxu))
  fxr = f.f(xr)

  #Error not calculated for the first iteration
  if i == 1:
    approxError = "---"
  else:#Calculate error
    approxError = calculate_error(xr,xprev)
    #Add error to the list
    falsepos_Errors.append(approxError)
    falsepos_Roots.append(xr)
    if approxError < error_Goal: #Stop process if error goal is reached
      print("False Position Method, Error tolerance goal reached")
      print("Last estimate= " + str(xprev) + "Apprx. % Relative Err: "
+ str(falsepos_Errors[-1]) + "# of iterations: " + str(i) + "f(xr) = "
+ str(f.f(xprev)))
      falsePositionOutput.write(str(i) + " " + str(xr) + " " +
str(f.f(xr)) + " " + str(approxError) + "\n")
      return 0

  #Record mid value for error calculation in the next iteration
  xprev = xr
  #debug
  #print("xr is " + str(xr))
  #print("xl was " + str(xl))
  #print("xu was " + str(xu))

  #Replace xr with whichever bound that gives the same sign
  if sign(fxr) == sign(fxl):
    xl = xr
  elif sign(fxr) == sign(fxu):
    xu = xr
```

```
  #Debug
  #print("xl is " + str(xl))
  #print("xu is " + str(xu))

  #Write to output file
  falsePositionOutput.write(str(i) + " " + str(xr) + " " + str(fxr) + "
" + str(approxError) + "\n")

  #Perform Recursive function call
  falsePosition(xl,xu,it_Max,error_Goal,xprev,i,falsePositionOutput)


#----------------------------

#Newton-Rhapson Method

def newton(xl, xu, it_Max, error_Goal, xprev = 500, i = 0, newtonOutput
= 0 ):

  #Initialization, open output file
  if i == 0:
    newtonOutput = open("output_newton.txt", "w")
    #Calculate the mean of the bounds for the single guess
    xprev = (xl + xu)/2

  #Termination, close output file
  if it_Max == i or it_Max < i :
    print("Newton Method, Max iteration reached")
    print("Last estimate= " + str(xprev) + "Apprx. % Relative Err: " +
str(newton_Errors[-1]) + "# of iterations: " + str(i) + "f(xr) = " +
str(f.f(xprev)))
    newtonOutput.close() #Close output file
    return 0 #Terminate function

  #Advance iteration
  i = i+1

  fxprev = f.f(xprev)
  fpxprev = fp.fp(xprev)
  if fpxprev == 0:
    newtonOutput.write(" The derivative equals zero, program
terminated")
    raise Exception("The derivative of f equals zero")
```

```
  #Calculate the next x
  xnext = xprev - ((fxprev)/(fpxprev))
  print("xnext is")
  print(xnext)
  fxnext = f.f(xnext)


  #Error not calculated for the first iteration
  if i == 1:
    approxError = "---"
  else:#Calculate error
    approxError = calculate_error(xnext,xprev)
    newton_Errors.append(approxError)
    newton_Roots.append(xnext)
    if approxError < error_Goal: #Stop process if error goal is reached
      print("Error tolerance goal reached")
      print("Last estimate= " + str(xprev) + "Apprx. % Relative Err: "
+ str(newton_Errors[-1]) + "# of iterations: " + str(i) + "f(xr) = " +
str(f.f(xprev)))
      newtonOutput.write(str(i) + " " + str(xnext) + " " + str(fxnext)
+ " " + str(approxError) + "\n")
      return 0

  #Record x with the next value
  xprev = xnext

  #Write to output file
  newtonOutput.write(str(i) + " " + str(xnext) + " " + str(fxnext) + "
" + str(approxError) + "\n")

  #Perform Recursive function call
  newton(xl,xu,it_Max,error_Goal,xprev,i,newtonOutput)


#--------------------------
#Secant Method

def secant(xold, xolder, it_Max, error_Goal, i = 0, secantOutput = 0 ):

  #Initialization, open output file
  if i == 0:
    secantOutput = open("output_secant.txt", "w")
```

```
  #Termination, close output file
  if it_Max == i or it_Max < i :
    print("Secant Method, Max iteration reached")
    print("Last estimate= " + str(xold) + "Apprx. % Relative Err: " +
str(secant_Errors[-1]) + "# of iterations: " + str(i) + "f(xr) = " +
str(f.f(xold)))
    secantOutput.close() #Close output file
    return 0 #Terminate function

  #Advance iteration
  i = i+1

  fxold = f.f(xold)
  fxolder = fp.fp(xolder)
  if fxold == fxolder:
    secantOutput.write(" Zero in the denominator, program terminated")
    raise Exception("Zero in the denominator")
  #Calculate the next x
  xnew = xold - (((fxold)*(xolder - xold))/(fxolder - fxold))
  #print(xnext)
  fxnew = f.f(xnew)


  #Error not calculated for the first iteration
  if i == 1:
    approxError = "---"
  else:#Calculate error
    approxError = calculate_error(xnew,xold)
    secant_Errors.append(approxError)
    secant_Roots.append(xnew)
    if approxError < error_Goal: #Stop process if error goal is reached
      print("Secant Method, error tolerance goal reached")
      print("Last estimate= " + str(xold) + "Apprx. % Relative Err: " +
str(secant_Errors[-1]) + "# of iterations: " + str(i) + "f(xr) = " +
str(f.f(xold)))
      secantOutput.write(str(i) + " " + str(xnew) + " " + str(fxnew) +
" " + str(approxError) + "\n")
      return 0

  #Record x with the next value
  xolder = xold
  xold = xnew
```

```
  #Write to output file
  secantOutput.write(str(i) + " " + str(xnew) + " " + str(fxnew) + " "
+ str(approxError) + "\n")

  #Perform Recursive function call
  secant(xold,xolder,it_Max,error_Goal,i,secantOutput)

#-----------------------------------
#New Poly Method

def polynomial(xl, xu, it_Max, error_Goal, xprev = 500, i = 0,
polynomialOutput = 0 ):

  #Initialization, open output file
  if i == 0:
    polynomialOutput = open("output_polynomial.txt", "w")

  #Termination, close output file
  if it_Max == i or it_Max < i :
    print("Polynomial Method, Max iteration reached")
    print("Last estimate= " + str(xprev) + "Apprx. % Relative Err: " +
str(polynomial_Errors[-1]) + "# of iterations: " + str(i) + "f(xr) = "
+ str(f.f(xprev)))
    polynomialOutput.close() #Close output file
    return 0 #Terminate function

  #Advance iteration
  i = i+1
  #calculate middle value
  xi = (xu + xl)/2

  #Store values to decrease the number of function calls
  fxl = f.f(xl)
  fxu = f.f(xu)
  fxi = f.f(xi)

  #Check if the bounding input values give different sign
  if sign(fxl) == sign(fxu):
    print("xl is " + str(xl))
    print("fxl is " + str(fxl))
    print("xu is " + str(xu))
    print("fxu is " + str(fxu))
```

```
    polynomialOutput.write(" No sign change between point, program
terminated")
    polynomialOutput.close()
    raise Exception(" No sign change between the points ")

  #Calculate polynomial coefficients
  poly_a = ((fxl - fxi)/((xl - xi)*(xl - xu))) + ((fxi - fxu)/((xu -
xi)*(xl-xu)))
  poly_b =
(((fxl-fxi)*(xi-xu))/((xl-xi)*(xl-xu)))-(((fxi-fxu)*(xl-xi))/((xu-xi)*(
xl-xu)))
  poly_c = fxi

  #Using the coefficients, calculate the root of the polynomial

  xr = xi - ((2*poly_c)/(poly_b + sign(poly_b)*(math.sqrt(poly_b*poly_b
- 4*poly_a*poly_c))))
  fxr = f.f(xr)

  #Error not calculated for the first iteration
  if i == 1:
    approxError = "---"
  else:#Calculate error
    approxError = calculate_error(xr,xprev)
    polynomial_Errors.append(approxError)
    polynomial_Roots.append(xr)
    if approxError < error_Goal: #Stop process if error goal is reached
      print("Polynomial Method, Error tolerance goal reached")
      print("Last estimate= " + str(xprev) + "Apprx. % Relative Err: "
+ str(polynomial_Errors[-1]) + "# of iterations: " + str(i) + "f(xr) =
" + str(f.f(xprev)))
      polynomialOutput.write(str(i) + " " + str(xr) + " " +
str(f.f(xr)) + " " + str(approxError) + "\n")
      return 0

  #Record mid value for error calculation in the next iteration
  xprev = xr

  #Replace xr with whichever bound that gives the same sign
  if sign(fxr) == sign(fxl):
    xl = xr
  elif sign(fxr) == sign(fxu):
    xu = xr
```

```
  #Write to output file
  polynomialOutput.write(str(i) + " " + str(xr) + " " + str(fxr) + " "
+ str(approxError) + "\n")

  #Perform Recursive function call
  polynomial(xl,xu,it_Max,error_Goal,xprev,i,polynomialOutput)



new_bisection(x_low,x_up, iteration_max, error_set)
falsePosition(x_low,x_up, iteration_max, error_set)
newton(x_low,x_up, iteration_max, error_set)
secant(x_low,x_up, iteration_max, error_set)
polynomial(x_low,x_up, iteration_max, error_set)

#For the plots

plt.plot(bisection_Errors)
plt.ylabel("Relative Percent Error")
plt.xlabel("Iteration")
plt.title("Bisection Method Error Plot")
plt.show()

plt.plot(bisection_Roots)
plt.ylabel("Root")
plt.xlabel("Iteration")
plt.title("Bisection Method Root Plot")
plt.show()

plt.plot(secant_Errors)
plt.ylabel("Relative Percent Error")
plt.xlabel("Iteration")
plt.title("Secant Method Error Plot")
plt.show()

plt.plot(secant_Roots)
plt.ylabel("Root")
plt.xlabel("Iteration")
plt.title("Secant Method Root Plot")
plt.show()

plt.plot(falsepos_Errors)
```

```
plt.ylabel("Relative Percent Error")
plt.xlabel("Iteration")
plt.title("False Position Method Error Plot")
plt.show()

plt.plot(falsepos_Roots)
plt.ylabel("Root")
plt.xlabel("Iteration")
plt.title("False Position Method Root Plot")
plt.show()

plt.plot(polynomial_Errors)
plt.ylabel("Relative Percent Error")
plt.xlabel("Iteration")
plt.title("Polynomial Method Error Plot")
plt.show()

plt.plot(polynomial_Roots)
plt.ylabel("Root")
plt.xlabel("Iteration")
plt.title("Polynomial Method Root Plot")
plt.show()


plt.plot(newton_Errors)
plt.ylabel("Relative Percent Error")
plt.xlabel("Iteration")
plt.title("Newton Method Error Plot")
plt.show()

plt.plot(newton_Roots)
plt.ylabel("Root")
plt.xlabel("Iteration")
plt.title("Newton Method Root Plot")
plt.show()

plt.plot(falsepos_Errors, label= "False Position Errors")
plt.plot(polynomial_Errors, label= "Polynomial Errors")
plt.ylabel("Relative percent error")
plt.xlabel("Iteration")
plt.title("Two Methods Error comparison")
plt.legend()
plt.show()
```

Our Outputs

Bisection Output

1 0.55 0.6861495220595056 ---
2 0.775 0.24893665905593132 29.032258064516125
3 0.8875 -0.06769218486766826 12.676056338028163
4 0.83125 0.09957855127284765 6.766917293233071
5 0.859375 0.01824073473747645 3.2727272727272676
6 0.8734375 -0.024144050758339586 1.610017889087654
7 0.86640625 -0.002807149244152929 0.8115419296663583
8 0.862890625 0.007752806125732037 0.4074241738343167
9 0.8646484375000001 0.002481846015988315 0.20329794443190288
10 0.8655273437500001 -0.00016039544181789545 0.10154575200270531
11 0.8650878906250001 0.0011612891078581766 0.05079866794603281
12 0.8653076171875 0.000500587816001152 3 0.025392884349508683
13 0.86541748046875 0.0001701314363103945 0.012694830382955585
14 0.8654724121093751 4.876809984954988e-06 0.0063470123202663984
15 0.8654998779296876 -7.775711267765661e-05 0.003173405451915697
16 0.8654861450195314 -3.643960054333 65e-05 0.0015867279026127326
17 0.8654792785644532 -1.5781257579172703e-05 0.0007933702456263095
18 0.8654758453369141 -5.452189372201488e-06 0.0003966866964038496
19 0.8654741287231447 -2.876810876184521e-07 0.00019834374159712886
20 0.8654732704162599 2.29456660028049e-06 9.917196915567553e-05
21 0.8654736995697023 1.0034432940120297e-06 4.958595999656525e-05
22 0.8654739141464235 3.5788123764479707e-07 2.479297384495162e-05
23 0.865474021434784 3.510010881946357e-08 1.239648537933316e-05
24 0.8654740750789643 -1.262904809617993e-07 6.198242311898438e-06

False-Position Output

1 0.7153969900770283 0.38869784123450635 ---
2 0.8457896795177269 0.05809615907631138 15.416680127268648
3 0.8630919847777031 0.007149924320576484 2.0046884416882382
4 0.865188775912602 0.0008579696222208444 0.24235070926425448
5 0.8654399158813488 0.00010263957330569617 0.02901876423056588
6 0.8654699532445378 1.2274358724750911e-05 0.0034706419415665877
7 0.8654735452268012 1.4677894933923241e-06 0.0004150308560229232
8 0.8654739747610124 1.755199410258257e-07 4.9629939631390865e-05
9 0.8654740261251834 2.0988861537674097e-08 5.934802140457278e-06

Newton Output

1 1.029762025684242 -0.5769469285931149 ---

2 0.8868970098382101 -0.06580080429195279 16.108411039980126
3 0.8659070730648281 -0.0013033651729218443 2.424040341775861
4 0.8654742150357275 -5.473558913893228e-07 0.050013971714069724
5 0.8654740331016466 -9.670042544485113e-14 2.1021321721197037e-05
6 0.8654740331016144 1.1102230246251565e-16 3.720096326720109e-12

Polynomial Output

1 0.8553566427615571 0.030140510133209175 ---
2 0.8654985107365776 -7.364365854745092e-05 1.1717949654690105
3 0.8654740327387511 1.0916882953182494e-09 0.0028282764011989905
4 0.8654740331016144 1.1102230246251565e-16 4.19265374914418e-08

Secant Output

1 0.28500844715008555 0.9365080942236688 ---
2 0.12252587802045806 0.9906636597874346 132.6108180204168
3 0.22873742169044667 0.961985731601199 46.43382918503212
4 0.13825735149811857 0.9878148646918985 65.44322541399136
5 0.20342389142854625 0.9709626713515692 32.03485071138646
6 0.1491636161908943 0.9855768614146967 36.37634741183237
7 0.18993197596395453 0.9751654441554155 21.464716283895918
8 0.1565384175801589 0.9839370000148494 21.332500289710495
9 0.18218900317032477 0.9774020705050459 14.079107489372264
10 0.16141453496966218 0.9827953350478866 12.870258681826357
11 0.1775729327960538 0.9786760557025805 9.099583800279893
12 0.16458473661267914 0.982028184622072 7.891494953107385
13 0.1747619668053241 0.9794304190903993 5.823481149065963
14 0.16662184151112638 0.9815247716875894 4.885389106478068
15 0.17302888441866707 0.979887502090486 3.7028747709185748
16 0.16792063501146387 0.9811995126129769 3.042061749501161
17 0.17195239113084246 0.9801683547737403 2.344693256583291
18 0.16874448276158927 0.9809914528079823 1.9010448915153497
19 0.17128069260273399 0.9803424134831862 1.4807330602213118
20 0.16926534747938968 0.9808592106862013 1.1906424754716578
21 0.1708603979873946 0.980450863280601 0.9335402040457654
22 0.1695939654343844 0.9807754987286348 0.7467438772166604
23 0.17059695352770787 0.9805186594832941 0.587928490270823
24 0.16980101732945668 0.9807226433657142 0.4687464249444831
25 0.17043164472382616 0.9805611297280754 0.3700177836054851
26 0.16993136443339385 0.9806893248158921 0.29440138499470375
27 0.17032784455853453 0.980587769457365 0.23277469762404387
28 0.17001337934102762 0.9806683431928406 0.18496498259477184
29 0.1702626389261728 0.9806044930547476 0.14639711137877104
30 0.17006496616778335 0.9806551389812835 0.11623367401515469