MIDDLE EAST TECHNICAL UNIVERSITY

DEPARTMENT OF MECHANICAL ENGINEERING

ME 310 NUMERICAL METHODS

FALL 2022

PROGRAMMING PROJECT 1

Murat Berk Buzluk 2377653

Gökberk Çiçek 2377810

Contents

Abstract

ME310 lecture students create this report to develop proper programs to solve seven equations using five other methods. Python is used in programming because the broader application area of Python makes it preferable for students to learn. We have learned how to code and deal with different problems caused by other functions. All the mistakes that we have done while writing the code lead us to a better understanding of the course objectives. For example, we experienced that when we forgot to change initial boundaries, we got dramatic errors in different methods. Also, we have seen that the purpose of the numerical method is to solve problems that can not be solved analytically. Thus, linear equations are not worth the effort to solve by numerical methods.

1. INTRODUCTION

Numerical methods are used when functions are not suitable for finding their roots by using analytical methods. Numerical methods do not give the exact result but provide very close approximate results. Therefore, numerical methods are preferred for challenging root-finding operations. Using iterative methods might be beneficial for calculating non-linear equations.

In this project, numerical methods are **polynomial method**, **bisection method**, **false-position method**, **Newton-Raphson method**, and **secant method** are applied separately to find the roots of seven different functions. For all functions there are given proper initial upper and lower borders so project did not focused on border selection details.

2. HAND CALCULATIONS

Consider the function $f(x) = x^2 - (1-x)^5$, [0.1,1]

$x_l = 0.1, x_u = 1.0, x_i = \frac{(x_u + x_l)}{2} = \frac{0.1 + 1.0}{2} = 0.55$

$f(x_l) = -0.5805, f(x_U) = 1.0$

Since $f(x_l) * f(x_u) < 0$, any bracketing method can be applied. For instance;

- According to the polynomial method:

$$p(x) = a(x - x_\text{i})^2 + b(x - x_i) + c$$

$$f(x_l) = a(x_l - x_i)^2 + b(x_u - x_i)^2 + c \quad \textbf{(1)}$$

$$f(x_u) = a(x_u - x_i)^2 + b(x_u - x_i) + c \quad \textbf{(2)}$$

$$f(x_i) = c \quad \textbf{(3)}$$

- Combining equations 1, 2, and 3:

$$a = \frac{f(x_l) - f(x_i)}{(x_l - x_i) * (x_l - x_u)} + \frac{f(x_i) - f(x_u)}{(x_u - x_i) * (x_l - x_u)} \quad \textbf{(4)}$$

$$b = \frac{\left(f(x_l) - f(x_i)\right) * (x_i - x_u)}{(x_l - x_i) * (x_l - x_u)} + \frac{\left(f(x_i) - f(x_u)\right) * (x_l - x_i)}{(x_u - x_i) * (x_l - x_u)} \quad \textbf{(5)}$$

then;

a= -0.367

b= 1.75

c= 0.284

$$x_r = x_i - \frac{2c}{b + sign(b)\sqrt{(b^2 - 4 * a * c)}} \quad \textbf{(6)}$$

Here in this example, sign(b) is positive. Then;

$$x_r = 0.55 + \frac{2 * 0.284}{1.75 + \sqrt{(1.75^2 - 4 * -0.367 * 0.284)}}$$

$x_r = 0.393$

and $f(x_r) = 0.0726$

Since $f(x_l)*f(x_r) < 0$, the new upper bound is $x_u = 0.393$

And new $x_i = \frac{(0.393+0.1)}{2} = 0.2467$

$$\varepsilon_a = \frac{|0.2467 - 0.55|}{0.2467} * 100 = 123\%$$

The same process will be applied till getting $\varepsilon_a < \varepsilon_s$ where $\varepsilon_s = 0.000001\%$.

3. NUMERICAL RESULTS

Bracketing methods:

Polynomial method results:

$\boldsymbol{Iteration}$	x_i	$f(x_i)$	$\boldsymbol{Approximate\ error}(\%)$
1	0.55	0.28404719	100
2	0.24668802	-0.18173585	122.95
3	0.36770305	0.03413945	32.91
4	0.34399758	-0.00315194	6.89
5	0.34595995	0.00000825	0.57
6	0.34595995	0.00000825	0

False position method results:

$Iteration$	x_i	$f(x_i)$	$Approximate\ error(\%)$
1	0.43055635	0.12550263	160
2	0.37179413	0.040392341	16
3	0.3541122	0.012990503	5
4	0.34855002	0.00415811	1.6
5	0.34678229	0.001328413	0.51
6	0.34621883	0.000424118	0.16
7	0.34603907	0.000135378	0.052
8	0.34598171	432.09569	0.017
9	0.3459634	137.91209	0.0053
10	0.34595756	44.017129	0.0017
11	0.34595569	14.048829	0.00054
12	0.34595509	4.4839239	0.00017
13	0.3459549	1.4311206	0.000055
14	0.34595484	0.45676647	0.000018
15	0.34595482	0.14578478	0.0000056
16	0.34595482	0.046529689	0.0000018
17	0.34595482	0.014850741	0.00000057

Bisection method results

$Iteration$	x_i	$f(x_i)$	$Approximate\ error(\%)$
1	0.55	0.28404719	100
2	0.325	-0.034501045	69
3	0.4375	0.13509274	26
4	0.38125	0.054658084	15
5	0.353125	0.011430673	8
6	0.3390625	-0.01116185	4,1
7	0.34609375	0.000223214	2
8	0.34257813	-0.005446558	1
9	0.34433594	-0.002606053	0.51
10	0.34521484	-0.001190023	0.25
11	0.3456543	-0.000483057	0.13
12	0.34587402	-0.000129834	0.064
13	0.34598389	467.11513	0.032
14	0.34592896	-415.56029	0.016
15	0.34595642	25.790981	0.0079
16	0.34594269	-194.88126	0.004
17	0.34594955	-84.544294	0.002
18	0.34595299	-29.376444	0.00099
19	0.3459547	-1.7926788	0.0005
20	0.34595556	11.999164	0.00025
21	0.34595513	0.5103246	0.00012
22	0.34595492	1.6552844	0.000062
23	0.34595481	-0.068697007	0.000031
24	0.34595487	0.79329375	0.000016
25	0.34595484	0.36229839	0.0000078
26	0.34595482	0.14680069	0.0000039
27	0.34595482	0.039051843	0.0000019
28	0.34595481	-0.014822583	0.00000097

Open methods:

Newton-Raphson method results:

$Iteration$	x_i	$f(x_i)$	$Approximate\ error(\%)$
1	0.33234453	-0.02221385	65
2	0.34574077	-0.000344032	3.9
3	0.34595476	-0.82416547	0.062
4	0.34595482	$-4.7462 * 10^{-8}$	0.000015
5	0.34595482	$-6.93889 * 10^{-10}$	$8.5 * 10^{-18}$

Secant method results:

$Iteration$	x_i	$f(x_i)$	$Approximate\ error(\%)$
1	0.43055635	0.12550263	77
2	0.37179413	0.040392341	16
3	0.34390628	-0.003299312	8.1
4	0.34601219	$9.2183 * 10^{-5}$	0.61
5	0.34595495	$2.11827 * 10^{-7}$	0.017
6	0.34595482	$-1.35962 * 10^{-11}$	0.000038
7	0.34595482	$-6.93889 - 10^{-17}$	$2.4 * 10^{-9}$

4. GRAPHICS OF NUMERICAL RESULTS

Approximate error values vs iteration number:

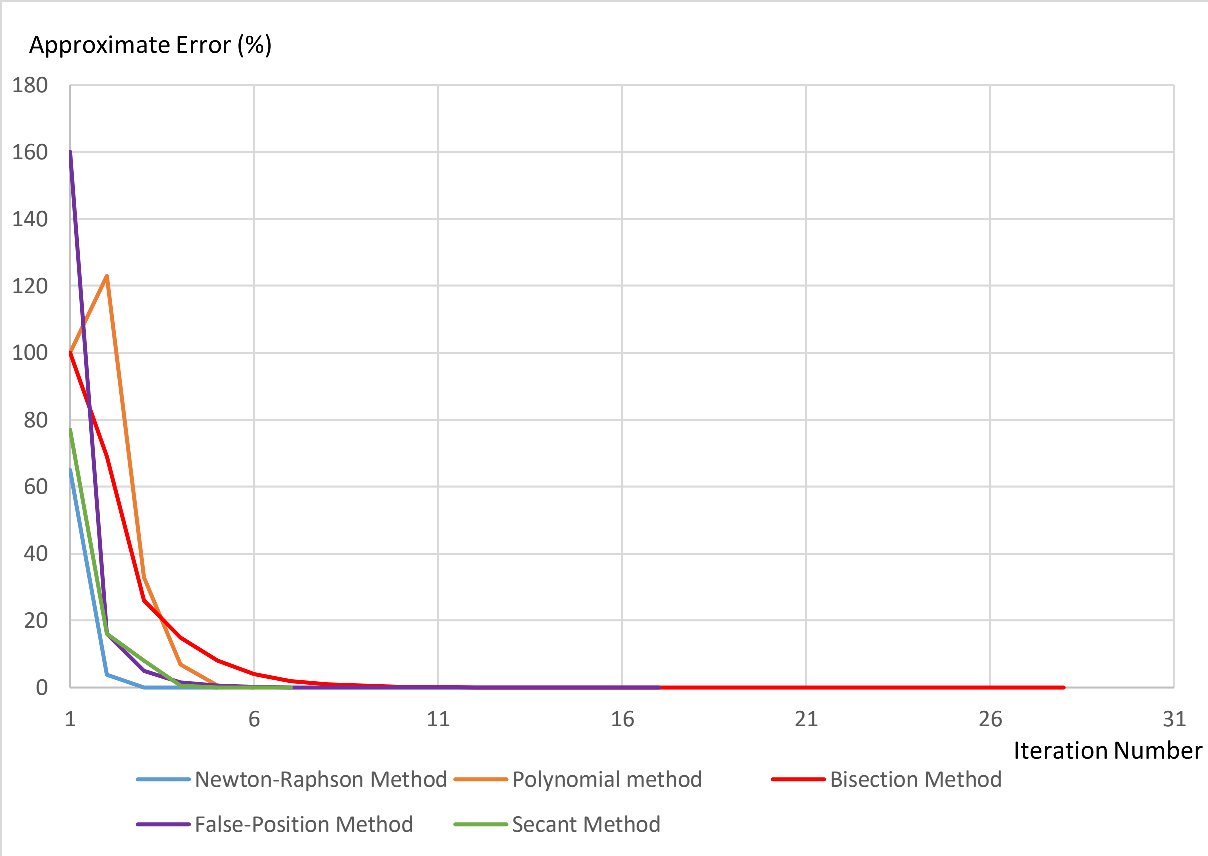

Figure 1 Approximate error values for each method vs. number of iterations

Estimated root value, x_i, iteration number:

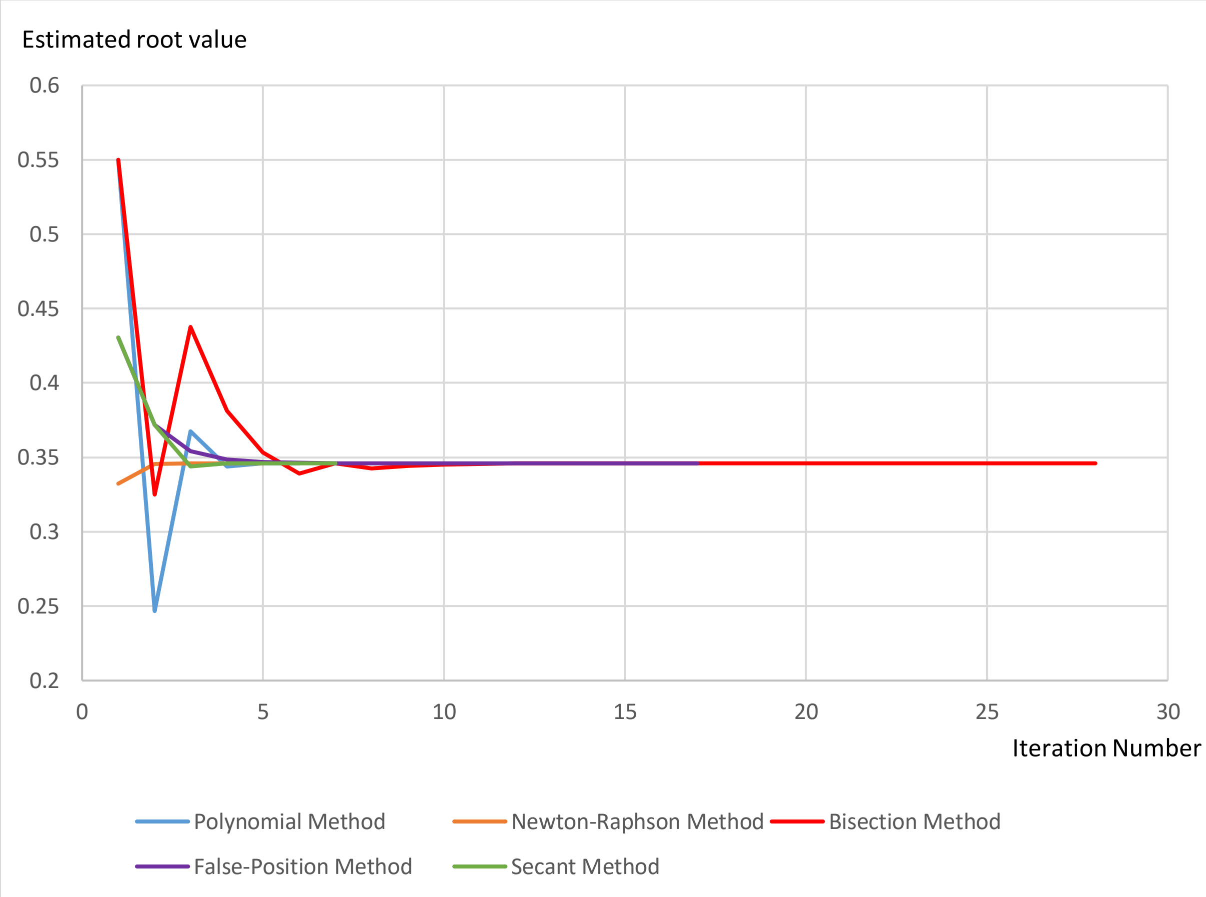

Figure 2 Estimated root values for each method vs. number of iterations

5. PLOT OF $\boldsymbol{\varepsilon_s}$ VALUES

- Comparing ε_s values of polynomial method and Newton-Raphson method vs. the number of iterations.

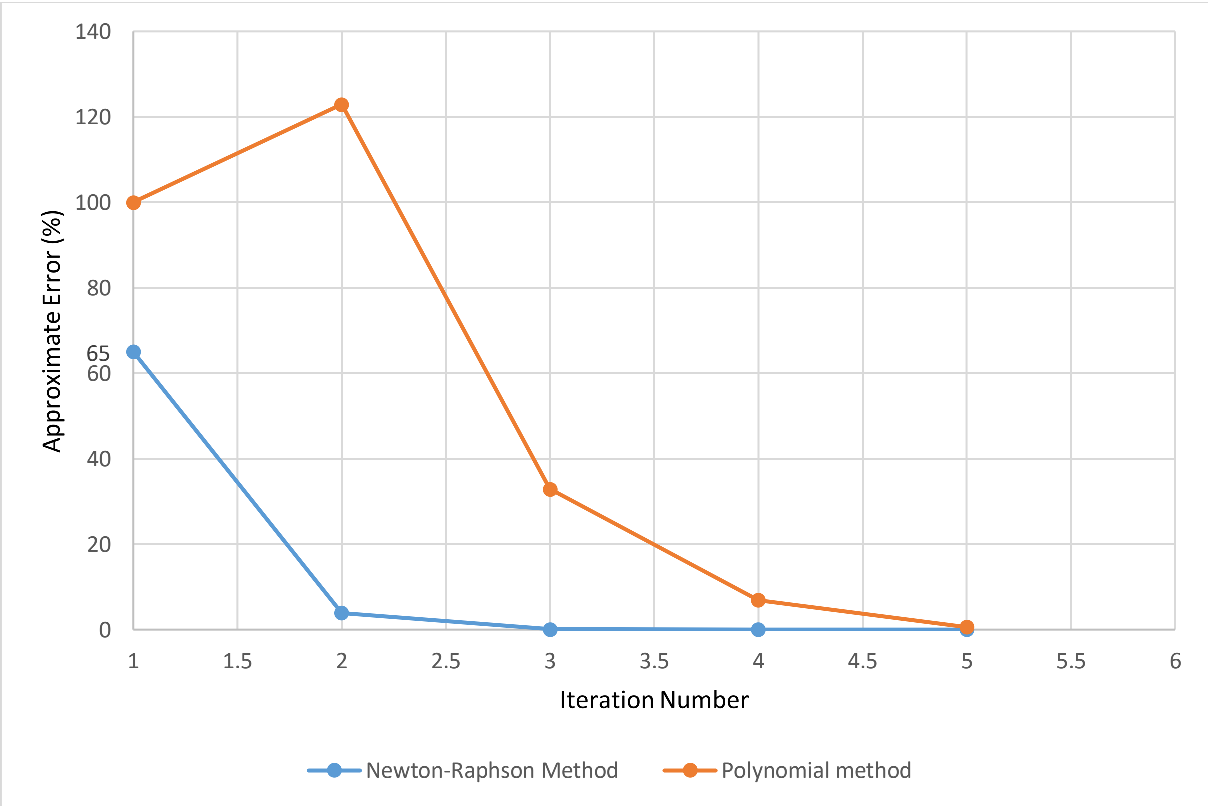

Figure 3 ε_s values to compare Newton-Raphson method vs. polynomial method

6. DISCUSSION AND CONCLUSION

The purpose of the code is to find the one root of given functions by using different numerical methods. According to the results, iteration numbers may be change in different methods. In other words, it can be interpreted that the convergence speed of every method is different from another that we use in our program. We learned that, generally, open methods converge to the approximate result faster than bracketing methods. The Polynomial method is an exception to that.

For example, consider the function f(x)=x2-(1-x)5. Using newton raphson method provides less iteration than the bisection method. Therefore newton raphson method is preferable. Furthermore, even though the number of iterations is the same for the Newton-Raphson method and the polynomial method, the Newton-Raphson method provides faster convergence as shown in the previous plot.

However, the disadvantages of open methods are that they may diverge or cause an error due to dividing by zero in calculations. For bracketing methods, if the multiplication of f(xL) and f(xU) is positive, then the method fails. For example, the functions with logarithmic operations such as ln(x) can't be solved by Newton-Raphson Method. Additionally, some other methods also couldn't manage to solve x^3 . However, this does not create any problem, after all, x^3 is not a tricky equation that couldn't be solved analytically.

To sum up, in our program, all methods are applicable to the given seven functions, and for each method, the code is working till the approximate error becomes less than the tolerance value. When the approximate error is less than the tolerance value, the program stops and provides a root closest to the real root.

7. Appendices

Code of the Program

```
"""
ME310 Project 1

@author: Murat Berk Buzluk
         2377653
         Gökberk Çiçek


"""
import numpy
import math
import sys
"""
We retrieve the wanted function with "from f import f" line
If we change f with fx, our input function become fx
f1 = x^2 - (1-x)^5
f2 = x*e^x - 1
f3 = cos(x) - x^3
f4 = ln(x)
f5 = x^5
f6 = e^(x^2+7x-30) - 1
f7 = x^-1 - sin(x) + 1

initial1 = [0.1,1]
initial2 = [-1,1]
initial3 = [0.1,1]
initial4 = [0.5,50]
initial5 = [-0.5,1/3]
initial6 = [2.8,3.1]
initial7 = [-1.3,-0.5]

"""
from f import f
initial = []     # initial values coming from input file

with open('input.txt','r') as function:          ### input taking section
    content = function.readlines()
    for x in content :
        row = x.split()
        initial.append(float(row[0]))              ### From there you can
change which function gonna be solved
        """
        row[0] = [0.1,1]
        row[1] = [-1,1]
        row[2] = [0.1,1]
        row[3] = [0.5,50]
        row[4] = [-0.5,1/3]
```

```
        row[5] = [2.8,3.1]
        row[6] = [-1.3,-0.5]
        """
"""
If we change the input.txt file and imported function(f and fd) we can get
results of other equations

"""
xl = initial[0]             # We are using first equation and first initial
guesses accordingly.
xu = initial[1]             # To use other equations initial1 can be changed
to initialx
errors = initial[2]         # To use other equations initial1 can be changed
to initialx
Niter = initial[3]          # To use other equations initial1 can be changed
to initialx
Niter = int(Niter)       #turn Niter from float to integer

# define a small number for zero check
EPS = 100*(numpy.finfo(numpy.float64).tiny)

# Check validity of data. It are enough to only do once
if (f(xl)*f(xu) > 0 or abs(xl - xu) < EPS):
    print('Please correct initial estimates, exiting...\n')
    sys.exit()

######################              POLYNOMIAL METHOD

print('POLYNOMIAL METHOD:\n')
print('Iter \t\t','xl \t\t\t', 'f(xl) \t\t' ,'xu \t\t\t', 'f(xu) \t\t', 'xi
\t\t\t','f(xi) \t\t', 'err(%) \t\t', sep='\t', end = "\n")    # Headers

file = open("output_polynomial.txt", "wt")                  # Output .txt
file opened

xi = (xu + xl) ## For initilazing to make first approximate error to 100
for i in range(0,Niter):
    x0i = xi
    xi = (xu + xl)/2
    approxE = (abs((xi - x0i)/xi))*100

    print(i,'\t\t', format(xl, '.6e'), format(f(xl), '.6e'), format(xu,
'.6e'), \
      format(f(xu), '.6e'), format(xi, '.10e'), format(f(xi), '.6e'), \
      format(approxE, '.6e'), sep='\t', end = "\n")

    a = (f(xl)-f(xi))/((xl-xi)*(xl-xu)) + (f(xi)-f(xu))/((xu-xi)*(xl-xu))
# a value to calculate roots
    b = ((f(xl)-f(xi))*(xi-xu))/((xl-xi)*(xl-xu)) - (f(xi)-f(xu))*(xl-
xi)/((xu-xi)*(xl-xu))   # b value to calculate roots
    c = f(xi)
# c value to calculate roots
    xr = xi - 2*c/(b+numpy.sign(b)*math.sqrt(b**2-4*a*c))
# New upper or lower bound calculation
    if math.isnan(f(xi)):
        print('This function cannot be solved by Newton-Rapshon Method')
```

```
        break
    """
    Below are the lines for writing output file

    """
    outpoly = (i+1, format(xi, '1.8f'), format(f(xi), '1.8f'),
format(approxE, '1.2f'))          # Tuple created for output.txt file
    for item in outpoly:
        s = str(item)
        file.write(s+ '\t\t')
    file.write('\n')

    if numpy.sign(f(xl))*numpy.sign(f(xr)) < 0:              ### New bound
calculation if statement
        xu = xr
    else :
        xl = xr

    if approxE < errors :              ### if statement for deciding stop or
continue
        break

file.close()   ### closes the temporary file

######################              BISECTION METHOD

file = open("output_bisection.txt", "wt")                  # Output .txt
file opened
xl = initial[0]        # initilazing
xu = initial[1]        # initilazing
errors = initial[2]    # initilazing
Niter = initial[3]     # initilazing
Niter = int(Niter)      # float to integer trasformation
xi = (xu + xl)
print('\n BISECTION METHOD:\n')
print('Iter \t\t','xl \t\t\t', 'f(xl) \t\t' ,'xu \t\t\t', 'f(xu) \t\t', 'xi
\t\t\t','f(xi) \t\t', 'err(%) \t\t', sep='\t', end = "\n")    # Headers

for i in range(0,Niter):
    # copy previous estimate
    x0i = xi
    # update the estimate with bisection and compute error
    xi = (xu + xl)/2
    approxE = (abs((xi-x0i)/xi))*100
    # damp out info
    print(i,'\t\t', format(xl, '.6e'), format(f(xl), '.6e'), format(xu,
'.6e'), \
      format(f(xu), '.6e'), format(xi, '.6e'), format(f(xi), '.6e'), \
      format(approxE, '.6e'), sep='\t', end = "\n")
    if math.isnan(f(xi)):
        print('This function cannot be solved by Newton-Rapshon Method')
        break

    """
    Below are the lines for writing output file
```

```
    """
    outpoly = (i+1, format(xi, '1.8f'), format(f(xi), '1.8g'),
format(approxE, '.2g'))          # Tuple created for output.txt file
    for item in outpoly:
        s = str(item)
        file.write(s+ '\t\t')
    file.write('\n')

    if (approxE < errors or abs(f(xi))< EPS):               # check the
error and terminate iterations if necessary
      break
    # decide new bounds
    if f(xl)*f(xi) < 0:
      xu = xi
    else :
      xl = xi
file.close()
######################               FALSE POSITION METHOD
file = open("output_falseposition.txt", "wt")                 # Output
.txt file opened
from f import fp                                ### Different from open
method we used derrivative of the equation
xl = initial[0]          # initilazing
xu = initial[1]          # initilazing
errors = initial[2]      # initilazing
Niter = initial[3]       # initilazing
Niter = int(Niter)        # float to integer trasformation
xi = (xu + xl)
print('\n FALSE POSITION METHOD:\n')
print('Iter \t\t','xl \t\t\t', 'f(xl) \t\t' ,'xu \t\t\t', 'f(xu) \t\t', 'xi
\t\t\t','f(xi) \t\t', 'err(%) \t\t', sep='\t', end = "\n")        # Headers

for i in range(0,Niter):
    # copy previous estimate
    xi0 = xi
    # update the estimate with false-position and compute approxEor
    xi = xu  -f(xu)*(xl-xu)/(f(xl) -f(xu))
    approxE = abs((xi-xi0)/xi)*100

    print(i,'\t\t', format(xl, '.6e'), format(f(xl), '.6e'), format(xu,
'.6e'), \
          format(f(xu), '.6e'), format(xi, '.6e'), format(f(xi), '.6e'), \
          format(approxE, '.6e'), sep='\t', end = "\n")
    """
    Below are the lines for writing output file

    """
    if math.isnan(f(xi)):
        print('This function cannot be solved by Newton-Rapshon Method')
        break

    outpoly = (i+1, format(xi, '1.8g'), format(f(xi), '1.8g'),
format(approxE, '1.2g'))
    for item in outpoly:
        s = str(item)
        file.write(s+ '\t\t')
```

```
    file.write('\n')

    # check approxEor and terminate iterations if necessary
    if (approxE < errors or abs(f(xi))< EPS):
      break
    # Decide on new bounds
    if f(xl)*f(xi) < 0:
        xu = xi
    else :
        xl = xi
file.close()

######################                NEWTON RAPSHON METHOD
file = open("output_newton.txt", "wt")                    # Output .txt file
opened
xl = initial[0]         # initilazing
xu = initial[1]         # initilazing
errors = initial[2]     # initilazing
Niter = initial[3]      # initilazing
Niter = int(Niter)       # float to integer trasformation
xi = (xu + xl)/2
print('\n NEWTON RAPSHON METHOD:\n')
print('Iter \t\t', 'xi \t\t\t','f(xi) \t\t', 'err(%) \t\t', sep='\t', end =
"\n")
for i in range(0,Niter):
    # copy previous estimate
    xi0 = xi
    # update the estimate
    xi = xi  - f(xi)/fp(xi)
    approxE = (abs((xi-xi0)/xi))*100  # Calculate error

    if math.isnan(f(xi)):
        print('This function cannot be solved by Newton-Rapshon Method')
        break

    print(i,'\t\t', format(xi, '.6e'), format(f(xi), '.6e'),
format(approxE, '.6e'),\
          sep='\t', end = "\n")
    """
    Below are the lines for writing output file

    """
    outpoly = (i+1, format(xi, '1.8g'), format(f(xi), '1.8g'),
format(approxE, '.2g'))
    for item in outpoly:
        s = str(item)
        file.write(s+ '\t\t')
    file.write('\n')

    # Check error and terminate iterations if necessary
    if (approxE < errors or abs(f(xi))< EPS):
      break

file.close()

######################                SECANT METHOD
```

```
file = open("output_secant.txt", "wt")                    # Output .txt file
opened

xi0 = initial[0]          # initilazing
xi1 = initial[1]          # initilazing
errors = initial[2]     # initilazing
Niter = initial[3]      # initilazing
Niter = int(Niter)        # float to integer trasformation
print('\n SECANT METHOD:\n')
print('Iter \t\t', 'xi \t\t\t','f(xi) \t\t', 'err(%) \t\t', sep='\t', end =
"\n")    # Headers

for i in range(0,Niter):
    # update the estimate
    xi = xi0 - f(xi0)*(xi1 - xi0)/(f(xi1) - f(xi0))
    # compute error
    approxE = abs((xi-xi0)/xi)*100

    if math.isnan(f(xi)):
        print('This function cannot be solved by Newton-Rapshon Method')
        break

    """
    Below are the lines for writing output file

    """
    outpoly = (i+1, format(xi, '1.8g'), format(f(xi), '1.8g'),
format(approxE, '1.2g'))
    for item in outpoly:
        s = str(item)
        file.write(s+ '\t\t')
    file.write('\n')

    # check error and terminate iterations if necessary
    if (approxE < errors):
      break
    # discard oldest data
    xi1  = xi0
    # update the new estimate
    xi0 = xi

    print(i,'\t\t', format(xi, '.6e'), format(f(xi), '.6e'),
format(approxE, '.6e'), \
            sep='\t', end = "\n")

file.close()
```