Numerical Methods - MAT202E

Homework I

Emre Anıl OĞUZ - 110190170

I. Programming language and used libraries

As a programming language Python is chosen due to my experience with it, more readable code for the review, and easy to code graphical figures. In addition to that, to work more efficiently, object-oriented programming methods are used.

```python
from math import sin,cos
import matplotlib.pyplot as plt
import numpy as np
import random
```

Figure 1. Used libraries

As seen in Figure 1 different libraries were used for this assignment. Sin and Cos from the math library are used to describe the function which is given in the homework document. For the graphical illustration, the matplotlib library is used. Finally, to assign random colors for graphs, and declare random initial values for the functions NumPy and random libraries are used. Any of the libraries not used for the computation as described in the homework document.

II. Question-1

Calculate sin (0.3π) to 8 significant figures using the Maclaurin series expansion of sin(x).

Solution

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

Equation 1.

For 8 significant figures we need $Es = (0.5\ x\ 10^{2-8})\% = (0.5\ x\ 10^{-6})\%$

```python
def maclaurin_serie(self,x,n):
    '''
    param:: x = x value  ---> sin(0.3*pi) -> x=0.3*pi

    param:: n = significant figure variable ---> n= 8
    '''

    func_string = ""
    error_significant = (0.5 * 10**(2-n))
```

Figure 2. Declaring of Question 1 Function and Error Significant

With the for loop, the equation we need (Equation 1.) is created and then approximation error is calculated.

```python
for i in range(0,8,1):

    def f(x):
        f = eval(func_string)
        return f

    if i !=0:
        previous_approximation = f(x)


    func_string += "+(" + str((-1)**i) +" * "  +"((x**" + str(1+(2*i)) +")"+ " / " + str(self.factorial(1+(2*i)))+ ")) "
    #sign = False


    current_approximation = f(x)

    approximation_error = abs((current_approximation - previous_approximation) / current_approximation)*100


    print(f"{i+1} \t\t\t\t\t {current_approximation} \t\t {approximation_error}")
    print("-------------------------------------------------------")

    if error_significant > approximation_error:
        print(f"The root was found to be at {current_approximation} after {i+1} iterations, actual value is {sin(x)}")
        break
```

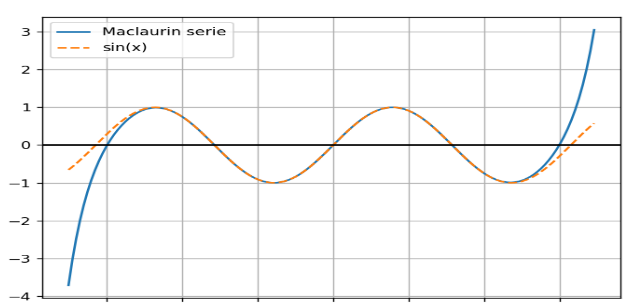
Figure 3. Actual loop of Question 1

As a result, as seen below (Figure 4.), at the 7th iteration, 11 significant digit of my approximation is correct.

```
n                    Result                    Ea
-------------------------------------------------------
1                    0.9424777960769379        100.0
-------------------------------------------------------
2                    0.8029495510155887        17.376962834697487
-------------------------------------------------------
3                    0.8091464496324907        0.7658562451477817
-------------------------------------------------------
4                    0.8090153904799282        0.016199834280625925
-------------------------------------------------------
5                    0.8090170073574143        0.00019985704520635l8
-------------------------------------------------------
6                    0.809016994300917     1.6138718246614999e-06
-------------------------------------------------------
7                    0.8090169943752608        9.189393287477434e-09
-------------------------------------------------------
The root was found to be at 0.8090169943752608 after 7 iterations, actual value is 0.8090169943749475
```

Figure 4. Function Output in a table format

Finally, there is a graphical illustration for sin function:



III. Question 2-A

There is a root of the equation $f(x) = \ln(x) - \cos(x) - e^{x}$ that lies between x ε {1,2}.

Calculate this root by using the bisection method, the false-position method, the Newton-Raphson method and the secant method with an error tolerance εs=0.05%.

A. Bisection Method

Step 1: Choosing xl and xu. In our example xl = 1 and, xu = 2

Step 2: Find estimate of the root → (xl+xu)/2

```
xr = (xl+xu)/2
```

Figure 5. Estimation of the root

Step 3: If f(xl) f(xr)<0 set xu=xr otherwise (f(xl) f(xr)>0) set xl=xr.
If f(xl) f(xr)=0 xr is the root, terminate the computation.

```
if function(xl) * function(xu) >= 0 :
    print("Error")
    break

elif function(xr) * function(xl) < 0:
    xu = xr
    error = abs((xu-xl)/xu)*100
elif function(xl) * function(xu) < 0:
    xl = xr
    error = abs((xu-xl)/xl)*100
```

Figure 6. Setting boundaries of the function

Step 4: Return to Step 2

Result in table format:

The root was found to be at 1.44091796875 after 11 iterations

Iteration	xl	Xu	Xr	Ea (%)
1	1	2	1.5	33.3333
2	1	1.5	1.25	20
3	1.25	1.5	1.375	9.09091
4	1.375	1.5	1.4375	4.34783
5	1.4375	1.5	1.46875	2.12766
6	1.4375	1.46875	1.45312	1.07527
7	1.4375	1.45312	1.44531	0.540541
8	1.4375	1.44531	1.44141	0.271003
9	1.4375	1.44141	1.43945	0.135685
10	1.43945	1.44141	1.44043	0.0677966
11	1.44043	1.44141	1.44092	0.0338868

Figure 7. Bisection Output Table

Result in graphs:

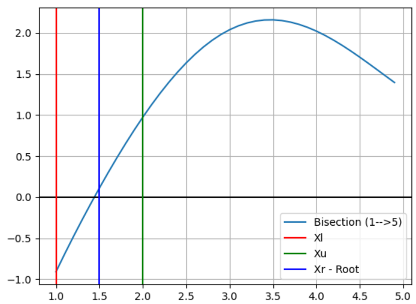


Figure 8. First iteration

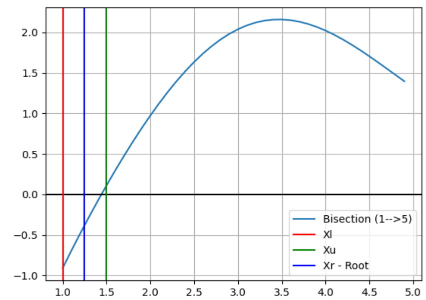


Figure 9. Second iteration

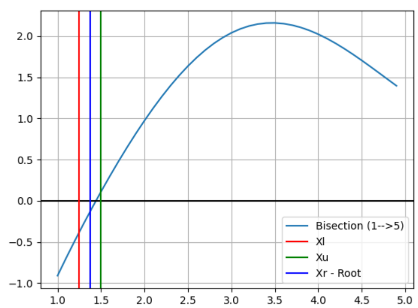


Figure 10. Sixth iteration

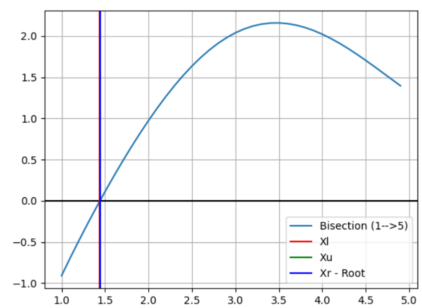


Figure 11. Last iteration

B. False Position Method

Step 1: Choose xl and xu with f(xl)*f(xu) < 0

Step 2: An estimate of the root is xr = xu $-\frac{f(xu)}{f(xl)-f(xu)}(xl - xu)$

```
c = (xl * function(xu)- xu * function(xl))/(function(xu)-function(xl))
```

Figure 11. Representation of the step 2

Step 3: If f(xl)*f(xr)<0, set xu=xr, otherwise f(xl)*f(xr)>0, set xl=xr. If f(xl)*f(xr) = 0, xr is the root

```
xr = (xl * function(xu)- xu * function(xl))/(function(xu)-function(xl))

if function(xl)*function(xu) >=0 :
    print("Error: The function does not change sign in the given interval,
    return

elif function(xr)* function(xl) <0:
    error = (abs(xr-xu)/xr)*100
    temp=xu
    xu = xr
    table += [[cnt,xl,temp,xr,error]]

elif function(xr) * function(xu) <0 :
    error = (abs(xr- xl)/xr)*100
    temp = xl
    xl = xr
    table += [[cnt,temp,xu,xr,error]]
```

Figure 12. Setting boundaries of the function

Step 4: Return to Step 2

Result in table format:

The root was found to be at 1.4413857837808024 after 4 iterations

Iteration	Xl	Xu	Xr	Ea (%)
1	1	2	1.48246	34.9105
2	1	1.48246	1.44408	2.65772
3	1	1.44408	1.44155	0.175669
4	1	1.44155	1.44139	0.0115586

Figure 13. False Position Output table

Result in graphs:

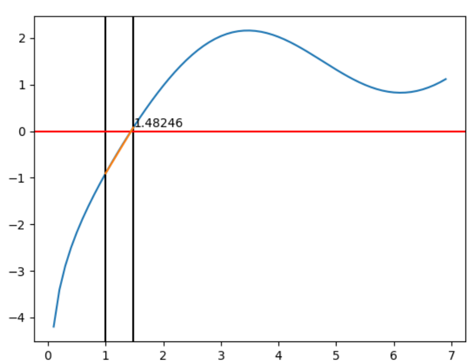


Figure 14. First iteration

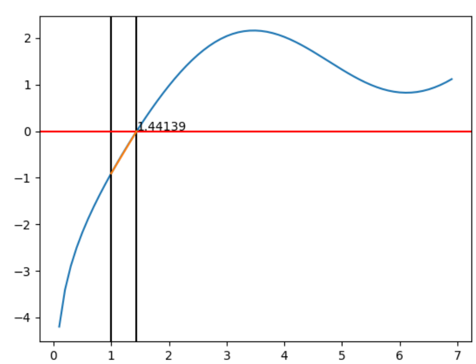


Figure 15. Last iteration

C. Newton-Raphson Method

Step 1: Find the derivative of f(x)

```
def calculate_derivative(self,function,x):

    h = 1e-7
    top = function(x+h) - function(x)
    bottom = h
    slope = top/bottom

    return float("%.3f"%slope)
```

Figure 16. Derivative calculator for the Newton Raphson

Step 2: Find $x_{i+1} = x_i - \frac{f(xi)}{f'(xi)}$

```
ix = xi - (function(xi)/self.calculate_derivative(function,xi))

error = abs((ix-xi)/ix)*100

table += [[i,xi,error]]

if error_significant > error:
    print(f"The root was found to be at {ix} after {i} iterations\n\n")
    table = tabulate(table, headers=['Iteration', 'Xi', 'Ea (%)'], tablefmt='orgtbl')
    print(table)
    break

xi = ix
```

Figure 17. Newton Raphson Loop

Step 3: Equalize $x_i = x_{i+1}$

Step 4: Return to Step 1

Result in table format:

```
The root was found to be at 1.441374070281983 after 3 iterations


|   Iteration |      Xi |      Ea (%) |
|-------------+---------+-------------|
|           1 | 1.81026 | 27.9348     |
|           2 | 1.41499 |  1.82313    |
|           3 | 1.44127 |  0.0075283  |
```

Figure 18. Newton Raphson output table

Result in graphs:

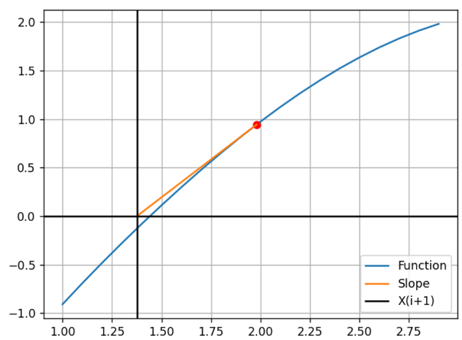


Figure 19. First iteration

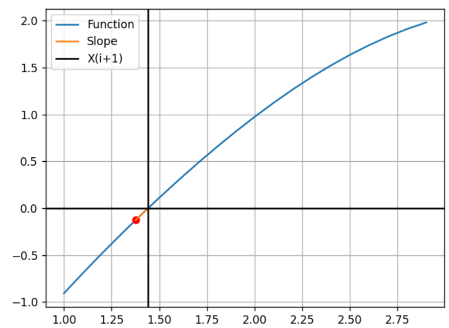


Figure 20. Last iteration

D. Secant Method

Step 1: Estimate the root $x_{i+1} = x_i - \frac{f(xi)}{\frac{f(xi)-f(x(i-1))}{xi-x(i-1)}}$

```
xi = guess_1 - (function(guess_1)/((function(guess_1)-function(guess_2))/(guess_1-guess_2)))
```

Figure 21. Representation of the step 1

Step 2: Equalize $x_{i-1} = x_i$, $x_i = x_{i+1}$

```
xi = guess_1 - (function(guess_1)/((function(guess_1)-function(guess_2))/(guess_1-guess_2)))


error = abs((xi-guess_2)/xi)*100
table += [[i,xi,error]]


if error_significant > error:
    print(f"The root was found to be at {xi} after {i} iterations\n\n")
    table = tabulate(table, headers=['Iteration', 'Xi', 'Ea (%)'], tablefmt='orgtbl')
    print(table)
    break

guess_1 = guess_2
guess_2 = xi
```

Figure 22. Loop of the Secant Method

Step 3: Return to Step 1

Result in table:

```
The root was found to be at 1.4413737311885542 after 2 iterations


|   Iteration |      Xi |     Ea (%) |
|-------------+---------+-----------|
|           1 | 1.44104 | 0.415824  |
|           2 | 1.44137 | 0.0230625 |
```

Figure 23. Secant Method output table

Question 2-B

I. $x = ln(x) - cos(x) - e^{-x} + x$

With the for loop seen in below, the equation is diverges, as seen in the graphs

```
for i in range(1,number_of_steps+1):
    xi = function(x0)

    xi_list.append(xi)

    error = (abs((xi-x0))/xi)*100

    table += [[i,xi,error]]

    x0_list.append(x0)

    x0 = xi
```

Figure 24. Fixed-point iteration loop

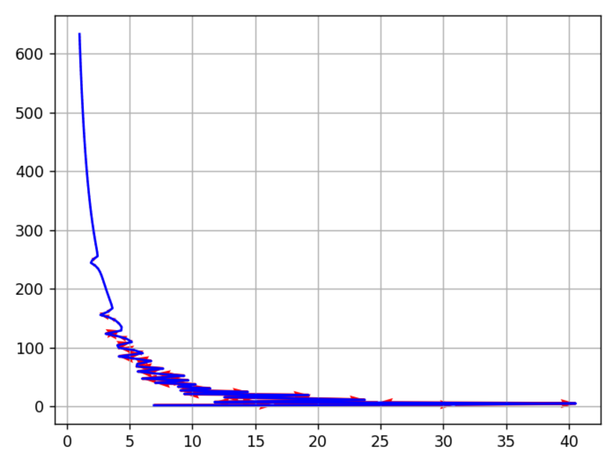


Figure 25. Final graph for the given function

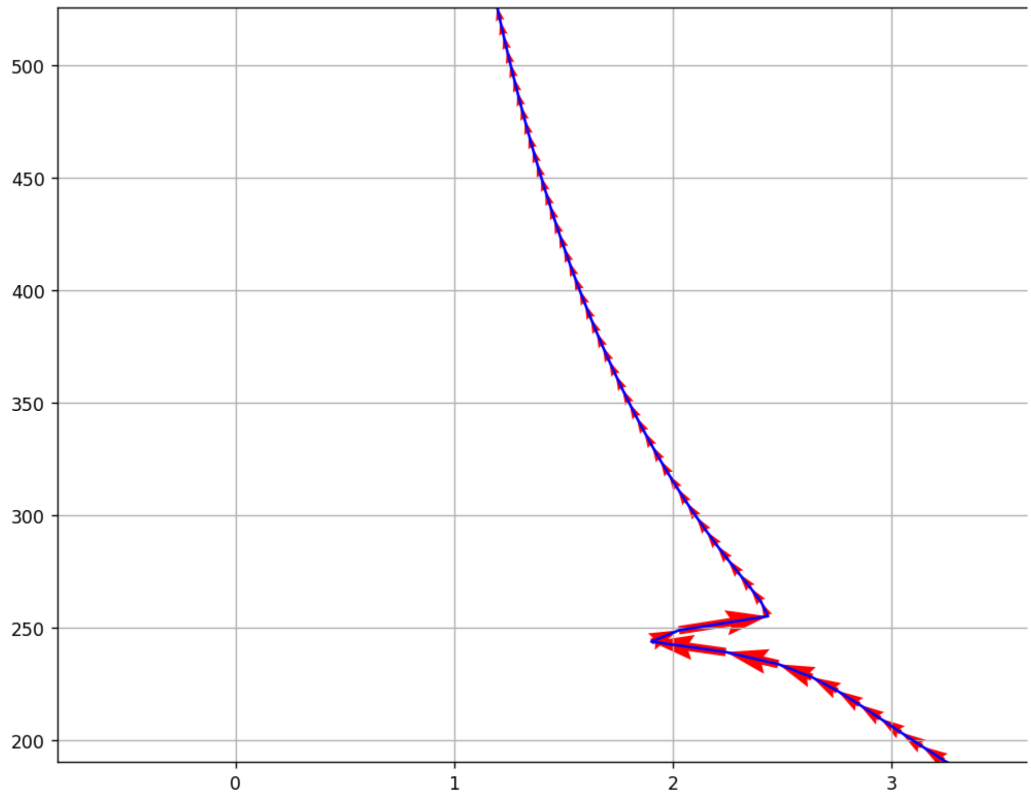


Figure 26. Representation of the divergence for the given function

Result in the table format, where the columns represent, iteration number, xi, and the Ea.

```
|         110 | 570.271    |  1.10377  |
|         111 | 576.565    |  1.0917   |
|         112 | 582.86     |  1.07989  |
|         113 | 589.154    |  1.06834  |
|         114 | 595.448    |  1.05703  |
|         115 | 601.742    |  1.04595  |
|         116 | 608.036    |  1.03511  |
|         117 | 614.329    |  1.02449  |
|         118 | 620.623    |  1.01409  |
|         119 | 626.917    |  1.00389  |
|         120 | 633.21     |  0.993901 |
The root was found to be at 633.2101368752659 after 120 iterations
```

Figure 27. Fixed-point iteration output table

II. $x = -ln(x) + cos(x) + e^{-x} + x$

With same loop proposed in figure 3, this function converges at the root of the equation with 64 iterations, as seen in the below graphs.

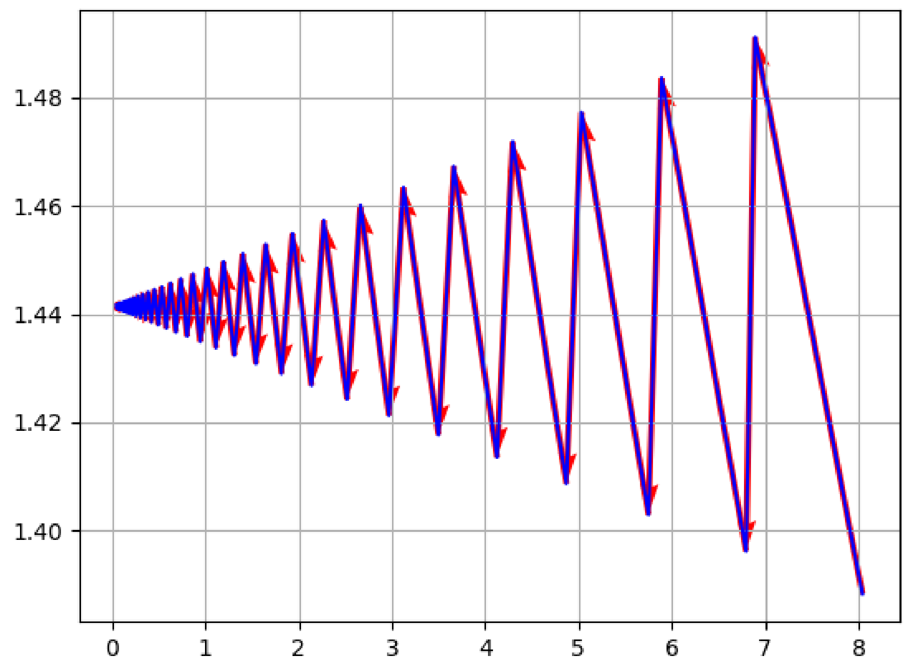


Figure 28. Final graph for the given function

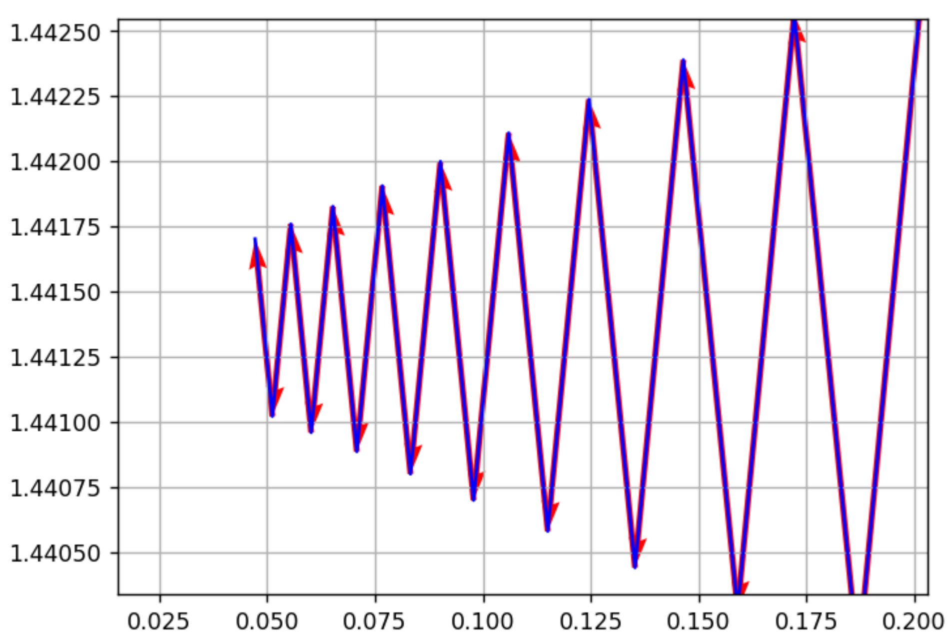


Figure 29. Representation of the convergence for the given function

Result in the table format, where the columns represent, iteration number, xi, and the Ea.

38	1.44405	0.386175
39	1.43891	0.357458
40	1.44365	0.32857
41	1.43928	0.303971
42	1.44331	0.279545
43	1.43959	0.258498
44	1.44302	0.237827
45	1.43986	0.219834
46	1.44277	0.202329
47	1.44008	0.186959
48	1.44257	0.172125
49	1.44028	0.159004
50	1.44239	0.146426
51	1.44044	0.135232
52	1.44224	0.124562
53	1.44058	0.115015
54	1.44211	0.105961
55	1.4407	0.0978229
56	1.442	0.0901367
57	1.4408	0.0832014
58	1.4419	0.0766746
59	1.44088	0.0707661
60	1.44183	0.0652224
61	1.44096	0.0601899
62	1.44176	0.0554802
63	1.44102	0.0511948
64	1.4417	0.0471929

Figure 30. Fixed-point iteration output table

IV. Guide

The code that was prepared for this homework, was written with OOP. To use a function, commented lines should be uncommented (Questions and their functions described in the code, below the if __name__ == "__main__" line). In addition to that, if variables that are used in the functions wanted to know, the line shown in Figure 31 should be called.

```
print(hw.bisection.__doc__)
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
[Running] python -u "c:\Users\emrea\Desktop\'22 Courses\Num-Methods\hw-1\hw-1.py"

        :param function: function to be evaluated
        :param xl: lower bound
        :param xu: upper bound
        :return: root of the function
```

Figure 31. Getting parameter information.

Lastly, all of the codes which written or going to be written will be published on my GitHub account.