ME EN 2450 Assignment HW 2a

Name: Chrstopher Wall

I declare that the assignment here submitted is original except for source material explicitly acknowledged.

I also acknowledge that I am aware of University policy and regulations on honesty in academic work, and of the disciplinary guidelines and procedures applicable to breaches of such policy and regulations, as contained in the University website.

Christopher Wall	9/9/2024
Name	Date
CHRISTOPHER WALL	u1467634
Signature	Student ID

Score

Total: /18

You must submit this assignment to **gradescope**.

When submitting to Gradescope, please remember to select and indicate the specific pages for each question.

This also applies to all future HW assignments

Q1: Function Approximation

Consider a Taylor series approximation to **(Always use radian unless otherwise stated)**

$$f(x) = \cos(x)$$

a) (2 pts) Write a code to compute the Taylor Series approximation of $f(x)$. The code should take as input x_i, x_{i+1}, and the order, n. The code should output the normalized true error, ϵ_T, at x_{i+1}.

b) (2 pts) Run your code with $n = 0, 1, 2, 3, 4,$ and 5.
Each time, use $x_{i+1} = 1$ a nd $x_i = 0.25$. Plot ϵ_T v s. n.

c) (1 pt) Comment on the convergence of the Taylor Series approximation at $f(x_{i+1})$.

Q2: Finite Difference Method

You have measured the following displacements along the length of a cantilevered beam with a linearly increasing load.

x [m]	0.0	0.375	0.75	1.125	1.5	1.875	2.25	2.625	3
y [m]	0.0	-0.2571	-0.9484	-1.9689	-3.2262	-4.6414	-6.1503	-7.7051	-9.275

The slope, θ, of the beam is the first derivative of the displacement with respect to x, as follows:

$$\frac{dy}{dx} = \theta(x) \tag{1}$$

The first derivative of the slope, θ, with respect to x is related to the bending moment, $M(x)$, as follows:

$$\frac{d\theta}{dx} = \frac{d^2y}{dx^2} = \frac{M(x)}{EI} \tag{2}$$

where x is the distance along the beam, y is the displacement, E is the modulus of elasticity, and I is the moment of inertia. Use the following physical parameters:

- $E = 200$ [GPa]
- $I = 0.0003$ [m^4]

Write a Matlab or Python code to complete the following:

1. (1 pt) Numerically differentiate the provided data for displacement, $y(x)$, to produce a table of the slope, $\theta(x)$, data. You can choose formulas of any error order.
2. (2 pts) Use numerical differentiation to compute the moment, $M(x)$, using the following 2 approaches [reference Eqn. 2]:
   (a) Numerically differentiate the first derivative of the $\theta(x)$ a pproximations obtained in the first part.
   (b) Numerically differentiate the second derivative of the measured $y(x)$ data provided.
3. (2 pts) Plot the moment, $M(x)$, obtained using each of the methods from the previous part. Provide a comment regarding which method is more accurate and why.

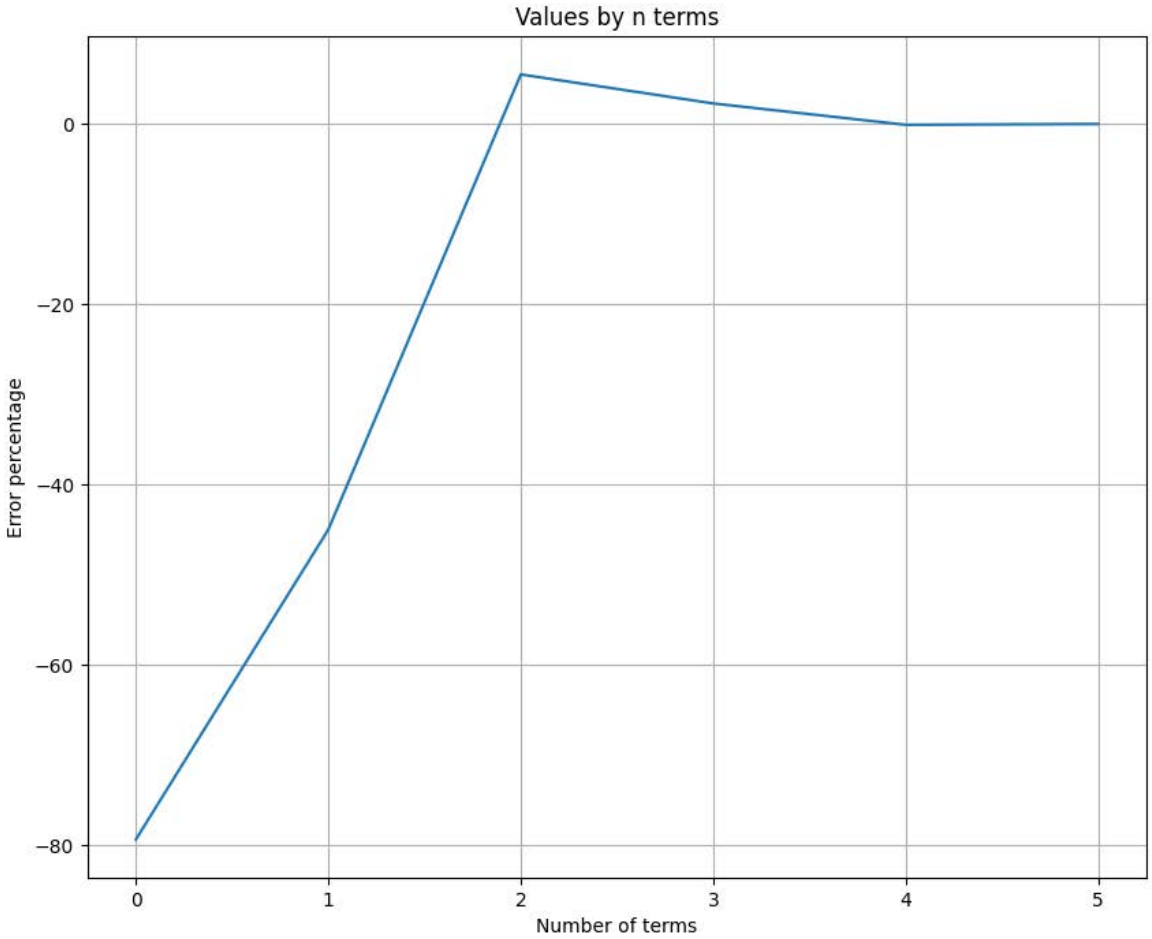
Q3: Root finding

Determine the real root of: $f(x) = x^3 - 13x - 12$

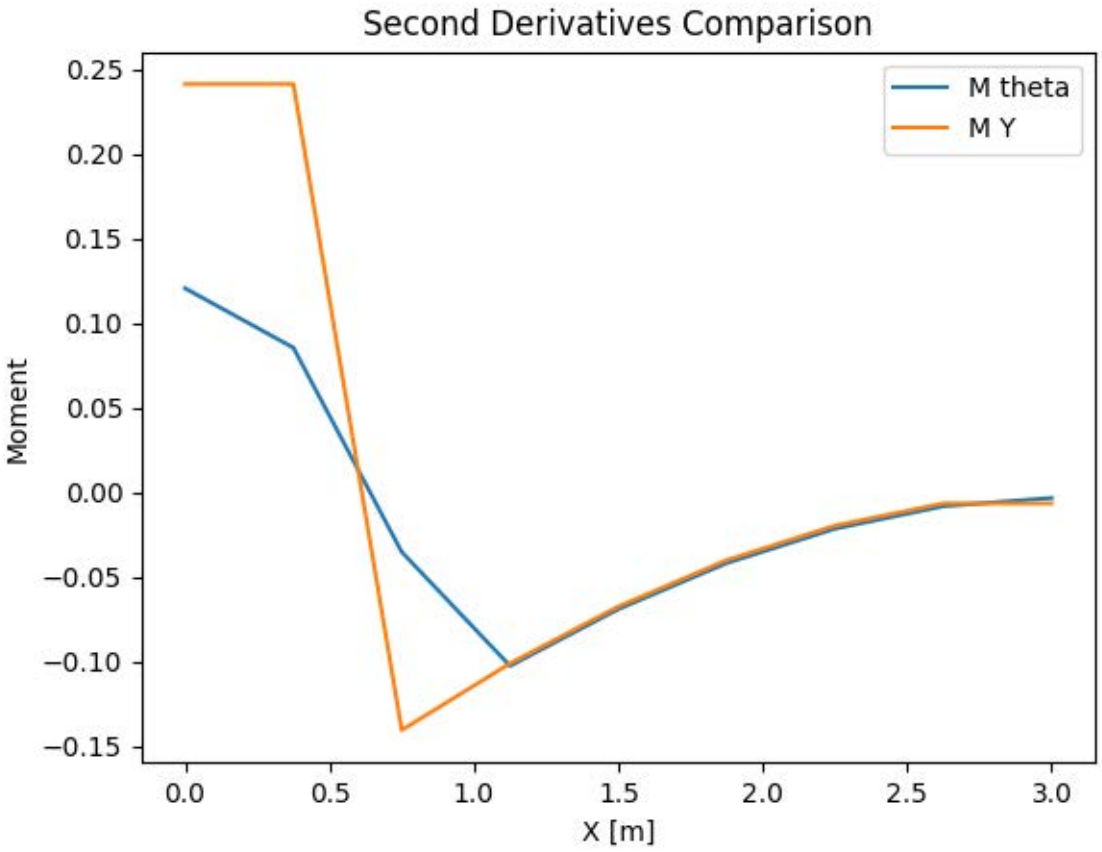
a. (1 pt) Graphically. Report your graph and your estimates of the roots from the graph.

b. (6 pts) Using matlab or python, write a script to determine one of the roots using the bisection method. Set $a = 2$ a nd $b = 7$. Print the normalized approximate percent error, $\epsilon_a(\%)$, over 5 iterations. Submit your code, along with a table of results. Your table should have 5 rows (1 for each itera-tion of the bisection method) and 3 columns: the iteration count, the approximate root at each iteration, and $\epsilon_a(\%)$ a t each iteration.

c. (1 pt) Compare your results of the previous step to the results from Müller's method given in Example 7.2 of the textbook (also attached on the next page). Report the difference in convergence between the two methods.
Here in particular, "convergence" means how fast the approximate percent error approaches zero


Values by n terms
Error percentage
0
−20
−40
−60
−80
0
1
2
3
4
5
Number of terms

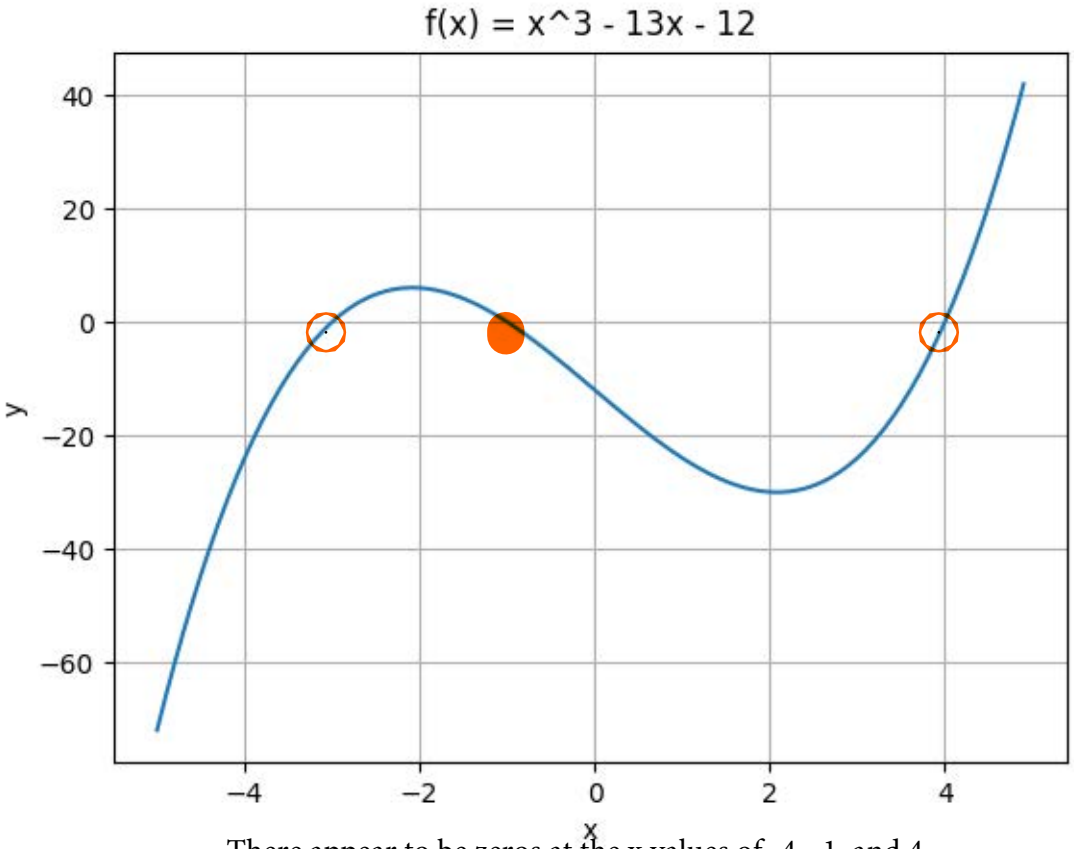
```
import numpy as np
import matplotlib.pyplot as plt
import math


# Q1: compute the taylor series approximation of f(x) = cos(x)

#finds percent error between two values
def error(val1, val2, abs=False):
    if abs:
        return np.abs(100 * (val2 - val1)/val2)
else:
        return 100 * (val2 - val1)/val2

#returns value for cosine derivatives
def derivativeCos(x, level):
    level = level % 4
    if(level == 0):
        return math.cos(x)
    elif( level == 1):
        return -math.sin(x)
    elif(level == 2):
        return -math.cos(x)
    elif( level == 3):
        return math.sin(x)


#use taylor series to estimate value
def taylor_approx(x, x1, n ):
    sum = 0
    h = x1 - x
    for i in range(0, n + 1 ):
        sum += (derivativeCos(x,i) * ((h)**i)) / math.factorial(i)
        return sum

#error from taylor approx method
def taylor_error(x, x1, n):
    return error( taylor_approx(x, x1, n), math.cos(x1))

#number of terms to compute
nVals = [i for i in range(0,6)]
trueErrors = []
values = []
#computing values at x values
for n in nVals:
    trueErrors.append(taylor_error(.25, 1, n))
    values.append(taylor_approx(.25, 1, n))

#plotting
print(math.cos(1))
print(values)

plt.figure(figsize=(10,8))
plt.plot(nVals,trueErrors)
plt.title("Values by n terms")
plt.ylabel("Error percentage")
plt.xlabel("Number of terms")
plt.grid("on")
plt.show()



#The error rapidly approaches zero. This approaches the true value of cos(x) in a much faster method than
#euler's method
```



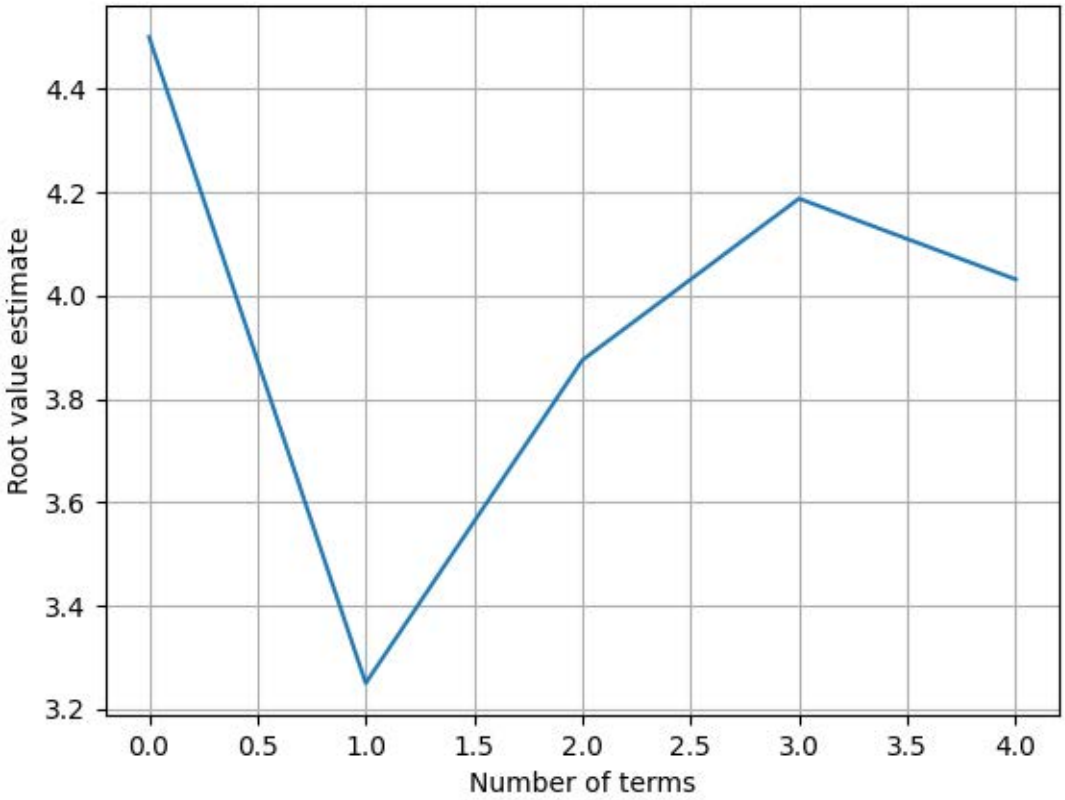
	X values	Y values	Slope	dTheta	dY2
1	0.000	1.0000	-3.352267	2.011733	4.023467
2	0.375	-0.2571	-2.597867	1.426489	4.023467
3	0.750	-0.9484	-2.282400	-0.585600	-2.340978
4	1.125	-1.9689	-3.037067	-1.707911	-1.683911
5	1.500	-3.2262	-3.563333	-1.148978	-1.122844
6	1.875	-4.6414	-3.898800	-0.695467	-0.666311
7	2.250	-6.1503	-4.084933	-0.356622	-0.326400
8	2.625	-7.7051	-4.166267	-0.135289	-0.107378
9	3.000	-9.2750	-4.186400	-0.053689	-0.107378

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#Question 2 Finite difference method

x = [0, 0.375, .75, 1.125, 1.5, 1.875, 2.25, 2.625, 3]
y = [1, -.2571, -0.9484, -1.9689, -3.2262, -4.6414, -6.1503, -7.7051, -9.275]


#numerical derivative definitions
def diff_forwards( x, y, i):
    return (y[i + 1] - y[i])/(x[i + 1] - x[i])

def diff_backwards( x, y, i):
    return (y[i] - y[i-1])/(x[i] - x[i-1])

def diff_center(x, y, i):
    return (y[i+1] - y[i-1])/(x[i+1] - x[i-1])

#second order derivatives
def diff2_forward(x, y, i):
    return (y[i+2] - 2 * y[i + 1] + y[i])/((x[i + 1] - x[i])**2)

def diff2_backward(x, y, i):
    return (y[i] - 2 * y[i - 1] + y[i - 2])/((x[i] - x[i-1])**2)

def diff2_center(x, y, i):
    return (y[i+1] - 2 * y[i] + y[i-1])/((x[i + 1] - x[i])**2)

#forward diff
slopes = [diff_forwards(x,y,0)]
#central diff
for i in range(1 , len(x) - 1):
    slopes.append(diff_center(x,y,i))
#backward diff
slopes.append(diff_backwards(x,y,-1))


#finding derivative of slope:
#forward difference for first value
dTheta = [diff_forwards(x,slopes,0)]
for i in range(1 , len(slopes) - 1):
    dTheta.append( diff_center(x, slopes, i))
#backwards difference for last value
dTheta.append( diff_backwards(x, slopes, -1))

#finding second y derivative:
dY2 = [diff2_forward(x,y,0)]
for i in range(1, len(y) - 1):
    dY2.append( diff2_center(x,y,i))
dY2.append(diff2_backward(x,y,-1))

#Finding bending moment
E = 200 #[GPa]
I = 0.0003 #[m^4]
My =  np.multiply( dY2, E * I)
Mt =  np.multiply( dTheta, E * I)

data = {
    ("X values") : x,
    ("Y values") : y,
    ("Slope") : slopes,
    ("dTheta") : dTheta,
    ("dY2") : dY2
}

df = pd.DataFrame(data)
df.index = range(1, len(x)+1)


print(df)
''''
plt.figure()
plt.plot(x, y)
plt.title("Profile of rod")
'''
plt.figure()
```

```
plt.plot(x, Mt, label='M theta')
plt.plot(x, My, label='M Y')
plt.title("Second Derivatives Comparison")
plt.xlabel("X [m]")
plt.ylabel("Moment")
plt.legend()
plt.show()

#The second derivative of the y values matches the expected curve for more values than the derivative of
#the slope. This derivative of the y values is a more accurate method
```



There appear to be zeros at the x values of -4, -1, and 4



Iteration count	approximate root	approximate error %
0	4.50000	NaN
1	3.25000	38.461538
2	3.87500	16.129032
3	4.18750	7.462687
4	4.03125	3.875969

```
    import matplotlib.pyplot as plt
    import numpy as np
    import pandas as pd
    #question 3
    #determine the real root of f(x) = x^3 - 13x - 12

    #part A
    def f(x):
        return x**3 - 13 * x - 12

    x = np.arange(-5, 5, .1)
    y = f(x)


    plt.figure()
    plt.plot(x,y)
    plt.grid("on")
    plt.title("f(x) = x^3 - 13x - 12")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.show()


    #Zeroes appear to be at -3, -1, and 4

    #part b
    a = 2
    b = 7
    tolerance = .001
    possibleRoots = []
    for i in range(0, 5):
        #if there isnt a root in the range, break
        if f(a)*f(b) > 0:
            break
        #possible root is the average of two guesses
        possibleRoot = (a+b)/2
        possibleRoots.append(possibleRoot)

        #redefining a - b interval
        if f(possibleRoot) * f(a) < 0:
            # root is to the left of possibleRoot
            b = possibleRoot
        elif f(possibleRoot) * f(b) < 0:
            # root is to the right of possibleRoot
            a = possibleRoot

    def error(val1, val2):
        return np.abs(100 * (val1 - val2)/val2)

    approximateErrors = [np.nan]
    for i in range(1, len(possibleRoots)):
        approximateErrors.append( error(possibleRoots[i-1], possibleRoots[i]))

    data = {
        ("approximate root") : possibleRoots,
        ("approximate error %") : approximateErrors

}

df = pd.DataFrame(data)
df.index.name = "Iteration count"
print(df)

plt.figure()
plt.plot(range(0, 5), possibleRoots, label="Root guesses")
plt.grid("On")
plt.xlabel("Number of terms")
plt.ylabel("Root value estimate")
plt.show()

#Müller's method approaches zero much more quickly because it is a higher order approximation than the bisection method
# The bisection method is a first order solution whereas Müller's method is a second order solution, which means Müller's
# converges much faster.
```