```python
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 16 16:42:10 2019
HW 7
ME EN 2450
@author: Ryan Dalby
"""
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

print("Exercise 1:")
#Exercise 1
func1 = lambda x : np.cos(x)
estimateX = np.pi/4.0
truedydx = -np.sin(estimateX)
h1 = np.pi / 3.0
h2 = np.pi / 6.0
centeredDifferenceInitialEstimate1 = (func1(estimateX + h1) - func1(estimateX - h1))/(2*h1)
centeredDifferenceInitialEstimate2 = (func1(estimateX + h2) - func1(estimateX - h2))/(2*h2)

#Second order accurate Richardson Extrapolation
def richardsonExtrapolation(h, estimateWithH, estimateWithHDividedBy2):
    """
    Given h, estimateWithH, estimateWithHDividedBy2 will give a second order accurate
    richardson extrapolation
    """
    return ((4 * estimateWithHDividedBy2) - estimateWithH)/3


richardsonExtrapolationdydx = richardsonExtrapolation(h1, centeredDifferenceInitialEstimate1, cente
trueRelError = ((truedydx - richardsonExtrapolationdydx) / truedydx) * 100
print("Using Richardson Extrapolation with step sizes of pi/3 and pi/6 for the derivative of cos(x)

#Exercise 2
print("\n\nExercise 2:")

E = 200 #GPa
I = 0.0003 #m^4
x = np.array([0.0, 0.375, 0.75, 1.125, 1.5, 1.875, 2.25, 2.625, 3.0])
y = np.array([0.0, -0.2571, -0.9484, -1.9689, -3.2262, -4.6414, -6.1503, -7.7051, -9.275])

def numericallyDifferentiate(xVals, yVals):
    """
    Given xVals and yVals will numerically differntiate using centered difference for inside values
    forward and backward difference for end values
    """
    dydx = np.empty_like(xVals)

    dydx[0] = (yVals[1] - yVals[0])/(xVals[1] - xVals[0]) #forward difference

    for i in range(1,xVals.shape[0]-1): #center difference
        dydx[i] = (yVals[i+1] - yVals[i-1])/(xVals[i+1] - xVals[i-1])

    dydx[-1] = (yVals[-1] - yVals[-2])/(xVals[-1] - xVals[-2]) #backward difference

```

```
    return dydx

theta = numericallyDifferentiate(x, y)
dthetadx1 = numericallyDifferentiate(x, theta)
Mestimate1 = dthetadx1 * E * I


def secondOrderNumericalDifferentiation(xVals, yVals):
    """
    Given xVals and yVals will numerically second order differntiate using centered difference for
    forward and backward difference for end values
    """

    d2ydx2 = np.empty_like(xVals)
    d2ydx2[0] = (yVals[2] - 2*yVals[1] + yVals[0])/((xVals[1] - xVals[0])**2) #forward difference

    for i in range(1,xVals.shape[0]-1): #center difference
        d2ydx2[i] = (yVals[i+1] - 2 * yVals[i] + yVals[i-1])/ ((xVals[i+1] - xVals[i])**2)

    d2ydx2[-1] = (yVals[-1] - 2*yVals[-2] + yVals[-3])/((xVals[-1] - xVals[-2])**2) #backward diffe

    return d2ydx2

dthetadx2 = secondOrderNumericalDifferentiation(x, y)
Mestimate2 = dthetadx2 * E * I
ex2Table = pd.DataFrame({"x": x, "y":y, "theta approx":theta, "M estimate 1":Mestimate1, "M estimat
print(ex2Table)
plt.plot(x, Mestimate1, label = "Mestimate1(a)")
plt.plot(x, Mestimate2, label = "Mestimate2(b)")
plt.legend()
plt.show()
print("It appears that the second estimate using the finite difference approximation is more accura


#Exercise 3
print("\n\nExercise 3:")

def gaussQuadrature(func, a, b, n = 2):
    """
    Given a function and a and b integration bounds will evaluate the integral using either n = 2 o
    """
    jacobian = (b - a) / 2.0
    firstTerm = (b + a) / 2.0
    transformFunc = lambda x : firstTerm + jacobian * x

    if(n == 3): #3 point gauss quadrature
        c = np.array([5.0/9.0, 8.0/9.0, 5.0/9.0])
        x = np.array([-np.sqrt(3.0/5.0), 0, np.sqrt(3.0/5.0)])
        return ((c[0] * func(transformFunc(x[0])) + c[1] * func(transformFunc(x[1])) + c[2] * func(
    else: # 2 point gauss quadrature
        c = np.array([1.0, 1.0])
        x = np.array([-1.0/np.sqrt(3.0), 1.0/np.sqrt(3.0)])
        return ((c[0] * func(transformFunc(x[0])) + c[1] * func(transformFunc(x[1]))) * jacobian)

#Testing
#Ex 22.3
```

```python
ex2234Func = lambda x: 0.2 + 25.0*x - 200.0*(x**2) + 675.0*(x**3) - 900.0*(x**4) + 400.0*(x**5)
print("Checking ex22.3.  Two point gauss quadrature:{}  Actual: {}".format(gaussQuadrature(ex2234Fu

#Ex 22.4
print("Checking ex22.4.  Three point gauss quadrature:{}  Actual: {}".format(gaussQuadrature(ex2234


#Exercise 4
print("\n\nExercise 4:")

def newtonsMethod(xInitial, dfdx, d2fdx2, maxIters, tolerence):
    """
    Given an initial x guess, a maximum number of iterations, and a termination tolerence in approx
    will attempt to find an extrema given df/dx and d2f/dx2 using Newton's method
    """
    x = xInitial
    lastX = xInitial
    eA = 100.0 #arbitrarily large approx relative error
    for _ in range(maxIters):
        x = x - (dfdx(x)/d2fdx2(x))
        eA = np.abs((x - lastX) / x) * 100
        lastX = x
        if(eA < tolerence):
            return x

    raise Exception("Did not find extrema in given number of iterations")


x0 = -1.0
eS = 1.0
ex1311 = lambda x: 3.0 + 6.0*x + 5.0*x**2 + 3.0*x**3 + 4.0*x**4
derivEx1311 = lambda x: 6.0 + 10.0*x + 9.0*x**2 + 16.0*x**3
secondDerivEx1311 = lambda x: 10.0 + 18*x + 48*x**2
predictedMinX = newtonsMethod(x0, derivEx1311, secondDerivEx1311, 1000, eS)
predictedMin = ex1311(predictedMinX)
print("13.11\na) Using Newton's method the estimated minimum is {} at x = {}".format(predictedMin,

def newtonsMethodFinite(xInitial, func, maxIters, tolerence):
    """
    Given an initial x guess, a maximum number of iterations, and a termination tolerence in approx
    will attempt to find an extrema given a function (will use finite differences with perturbation
    to find using Newton's method
    """
    x = xInitial
    lastX = xInitial
    eA = 100.0 #arbitrarily large approx relative error

    #Approximation of derivatives using finite differences
    dfdx = lambda x: (func(x + 0.01*x) - func(x - 0.01*x)) / (2*(0.01*x))
    d2fdx2 = lambda x: (func(x + 0.01*x) - 2 * func(x) + func(x - 0.01*x)) / (0.01*x)**2

    for _ in range(maxIters):
        x = x - (dfdx(x)/d2fdx2(x))
        eA = np.abs((x - lastX) / x) * 100
        lastX = x
        if(eA < tolerence):
```

```
            return x

    raise Exception("Did not find extrema in given number of iterations")

predictedFiniteMinX = newtonsMethodFinite(x0, ex1311, 1000, eS)
predictedFiniteMin = ex1311(predictedFiniteMinX)
print("b) Using Newton's method with finite differences the estimated minimum is {} at x = {}".forr
```