

General instructions:

- You will write **three Python programs** named **hw2a.py**, **hw2b.py**, and **hw2c.py** to fulfill the requirements given in parts a, b, and c below. You will place those three python files into a single compressed file named hw2.zip. You will upload hw2.zip to canvas for submission.
- Use **docstrings** in ALL functions and write a clear description of the arguments/parameters to functions.
- In this assignment, you must use variables, loops, if statements, your own function definitions, callbacks and function calls. For now, you may not use any of the powerful functions available in python modules, with these exceptions: you may import and use functions from the **math**, **copy**, and **random** modules.
- See your MAE 3013 textbook for reminders of:
 - The Simpson's 1/3 rule for numerical integration (§19.5, p831 & Table 19.4)
 - The Secant Method for finding the solution (root or zero) of a nonlinear equation (§19.2, p805)
 - The use of cofactors and minor matrices for finding determinant of a matrix (§7.7)
 - The Cramer's method for solving a matrix equation (§7.7)

Problems:

a) In HW1 problem 3, we used `random.normalvariate(μ , σ)` to produce a list of numbers from a normally distributed population. Here, we need to write a function defined as: `def Probability(PDF, args, c, GT=True)`: that integrates the Gaussian normal density function using Simpson's rule.

PDF: is a callback function for the Gaussian/normal probability density function

$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$, which takes 1 argument in the form of a tuple containing values for x , μ (population mean), and σ (population standard deviation).

args: is a tuple containing μ and σ .

c: is a floating point value representing the upper limit of integration.

GT: is a Boolean indicating if we want the probability of x being greater than **c** (**GT==True**) or less than **c** (**GT==False**)

To find the probability of $x < c$ you should use the Simpson's 1/3 rule to integrate **PDF** between the limits of $a = \mu - 5 \cdot \sigma$ to **c**.

Write and call a **main()** function that uses your **Probability** function to find:

$P(x < 1 | N(0, 1))$: probability $x < 1$ given a normal distribution of x with $\mu = 0$, $\sigma = 1$

$P(x > \mu + 2\sigma | N(175, 3))$

$P(-2.00 < x < 2.00 | N(0, 1))$

Print you findings to the console (CLI) in the following format:

$P(x < 1.00 | N(0, 1)) = X.XX$

$P(x > 181.00 | N(175, 3)) = Y.YY$

$P(-2.00 < x < 2.00 | N(0, 1)) = Z.ZZ$

Programs need to be work on 2a

```
# region imports
from math import sqrt, exp, pi
# endregion
```

```

# region methods
def Probability(PDF, args, c, GT=True):
    """
    Use Simpson rule to numerically integrate PDF. Result is probability  $P(x < c)$ .
    If I want probability  $x > c$  (GT=True), calculate  $1-P$ 
    :param PDF: the probability density function (a callback function)
    :param args: contains (mean, standard deviation) in that order
    :param c: value for calculating  $P(x < c)$ 
    :param GT: boolean for deciding if we want  $P(x > c)$  (GT=True) or  $P(x < c)$  (GT=False)
    :return:  $P(x < c)$  or  $P(x > c)$ 
    """
    m, s = # $JES Missing Code # unpack mean and standard deviation
    a = # $JES Missing Code # compute left limit for integral as 5 standard deviations
    below (to left of) mean
    P = # $JES Missing Code # use Simpson function to integrate PDF with 100 points

    return P if not GT else (1-P) # return desired probability based on GT

def GNPDF(args):
    """
    This is the Gaussian Normal Probability Density Function with parameters mean (mu),
    standard deviation (sigma).
    :param args: contains x, mean, standard deviation
    :return: f(x)
    """
    x, mu, sigma = # $JES Missing Code # unpack from args
    f = # $JES Missing Code # calculate f(x) PEDMAS
    return f # return f(x)

def Simpson(fcn, args, a, b, npoints = 21):
    """
    This uses the simpson 1/3 rule for numerical integration, which uses quadratic
    Lagrange polynomials for interpolation.
    See Kryszig pp831-32 for derivation.
    The panel size is:  $dX = \text{abs}(a-b)/(2 * \text{npoints})$ 
    The integral is found by:  $I = dX/3 * (f_{cn}(0) + f_{cn}(\text{npoints}) + 4 * \sum(f_{cn}(j), 1 \text{ to } n-1, \text{odd}) + 2 * (f_{cn}(j), 2 \text{ to } n-2, \text{even}))$ 
    :param fcn: the callback function. the function of x to integrate
    :param args: contains parameters for the Gaussian Normal distrubution (mean, standard
    deviation)
    :param a: left limit of integration (check  $a < b$ )
    :param b: right limit of integration (check  $a < b$ )
    :param npoints: number of points in integration
    :return: the value of the integral of fcn(x) between a,b
    """
    mu, sig = # $JES Missing Code # unpack mu and sig
    area = 0 # initial value for the integral
    m = npoints # staying consistent with book nomenclature
    n = 2*m # ensure an even number of panels
    # if a,b are passed wrong, this checks to put them in correct order
    xL = min(a, b) # lower limit of integration
    xR = max(a, b) # upper limit of integration

    if xL == xR:
        return 0 # nothing to do, so return 0

    h = (xR-xL)/(2*m) # calculate panel width
    x = xL
    area = (fcn(xL, args) + fcn(xR, args)) # first calculate f(x) at xL and xR using the
    callback fcn

```

```

for j in range(1,2*m): # counts from 1 to 2*m-1
    x=j*h+xL # update the x position for evaluating the function
    if not j % 2 == 0: # the odds
        area += # $JES Missing Code
    else: # the evens.
        area += # $JES Missing Code
return (h/3.0)*area # finally, return the value for the integral

def main():
    '''
    This main function calculates probabilities using the probability function I defined.
    Step 1: define mean and standard deviation.
    Step 2: set c=1 for  $P(x<1|N(0,1))$  i.e.,  $c=\mu_1 + 1*\text{stdev}$ 
    Step 3: calculate probability by calling Probability with GNPDF as callback, ( $\mu_1$ ,
stdev1) as tuple argument, c1, and GT=False
    Step 4: repeat steps 1-3 for  $P(x>\mu_2+2*\text{stdev}_2|N(175,3))$ 
    Step 4: calculate  $P(-1<x<1|N(0,1))$ 
    Step 5: output results to the console.
    :return: no return value from this function.
    '''

    #  $P(x<1|N(0,1))$ 
    # step 1
    mu1=0
    stdev1 = 1

    # step 2
    c1 = 1

    # step 3
    #GNPDF is the Gaussian-Normal Probability Density Function (see def GNPDF)
    P1 = # $JES Missing Code #  $P(x<1.00|N(0,1))$ 

    #  $P(x>181.0|N(175,3))$ 
    # step 1 (part 2)
    mu2 = 175
    stdev2 = 3

    # step 2 (part 2)
    c2 = mu2+2*stdev2

    # step 3 (part 3)
    P2 = # $JES Missing Code

    #  $P(-2.00<x<2.00|N(0,1))$ 
    # step 1 (part 3)
    mu3 = 0.0
    stdev3 = 1.0

    # step 2 (part 3)
    c3 = 2.0

    #step 3 (part 3)
    P3 = # $JES Missing Code

    # step 5
    print('P(x<{:0.2f}|N({:0.2f},{:0.2f}))={:0.3f}'.format(# $JES Missing Code))
    print('P(x>{:0.2f}|N({:0.2f},{:0.2f}))={:0.3f}'.format(# $JES Missing Code))
    print('P({:0.2f}<x<{:0.2f}|N({:0.2f},{:0.2f}))={:0.3f}'.format(# $JES Missing Code))

# endregion

```

```
# region function call(s)
if __name__ == "__main__":
    main()
# endregion
```

b) Write a function defined as: `def Secant(fcn, x0, x1, maxiter=10, xtol=1e-5):` that use the Secant Method to find the root of the callback `fcn(x)`, in the neighborhood of `x0` and `x1`.

fcn: the callback function for which we want to find the root (see the specific functions below).

x0 and **x1**: two x values in the neighborhood of the root

xtol: exit if the $|x_{\text{newest}} - x_{\text{previous}}| < \text{xtol}$

maxiter: exit if the number of iterations (*new x values*) equals this number

return value: the final estimate of the root (most recent new x value)

Write and call a `main()` function that uses your `Secant` function to estimate and print the solution of:

$x - 3\cos(x) = 0$; with $x_0=1$, $x_1=2$, $\text{maxiter} = 5$ and $\text{xtol} = 1e-4$

$\cos(2x) \cdot x^3 = 0$; with $x_0=1$, $x_1=2$, $\text{maxiter} = 15$ and $\text{xtol} = 1e-8$

$\cos(2x) \cdot x^3 = 0$ with $x_0=1$, $x_1=2$, $\text{maxiter} = 3$ and $\text{xtol} = 1e-8$

NOTE: you MUST use lambda functions in the call(s) to Secant. e.g.:

`Secant(lambda x: x-3*math.cos(x), x0=1, x1=2, maxiter=5, xtol=1e-4)`

Programs need to be work on 2b.

```
# region imports
import math
from math import cos
# endregion

# region functions
def Secant(fcn, x0, x1, maxiter=10, xtol=1e-5):
    """
    Secant method finds the root of the callback function fcn (i.e., the value of x that
    makes fcn==0)
    :param fcn: callback function to find root of
    :param x0: initial guess for the root
    :param x1: second initial guess for the root. Required: x1 != x0
    :param maxiter: maximum number of iterations
    :param xtol: exit tolerance if better than
    :return: the root of the callback function
    """
    delta = (x1-x0) # seed the value for delta with initial guesses
    x = x1
    fOld = fcn(x0) # value of callback at x0
    fNew = fOld
    Niter = 0 # a counting variable for number of iterations
    while Niter < maxiter and abs(delta) > xtol:
        fNew = fcn(x) # value of callback at x
        delta = # $JES Missing Code # see Secant derivation in textbook
        x += delta # the next value of x
        fOld = fNew
        Niter += 1
    return x

def main():
    """
    this tests the secant method for finding the root of equations
    :return: nothing
    """
    # define our functions using lambda notation (to be used as callbacks in calls to
    Secant)
    fn1 = lambda # $JES Missing Code
```

```

fn2 = lambda # $JES Missing Code

maxiter1 = 5
maxiter2 = 15
maxiter3 = 3

r1 = # $JES Missing Code # call Secant with proper callback and arguments
r2 = # $JES Missing Code
r3 = # $JES Missing Code

#output of root values and test in functions to see if fn(r)=0.00
print("r1={:0.6f}".format(r1))
print("fn1(r1)={:0.6f}".format(fn1(r1)))
print("\nwith maxiter={}: r2={:0.6f}".format(maxiter2, r2))
print("fn2(r2)={:0.6f}".format(fn2(r2)))
print("\nwith maxiter={}: r3={:0.6f}".format(maxiter3, r3))
print("fn2(r3)={:0.6f}".format(fn2(r3)))
print("\nExact answer for fn2={:0.6f}".format(math.pi/4.0))
# endregion

# region function call(s)
if __name__ == "__main__":
    main()
# endregion

```

c) Write a function defined as: `def Cramer(Aaug) :` that uses Cramer's method to find the solution for a set of N linear equations expressed in matrix form as $\mathbf{Ax} = \mathbf{b}$. Both \mathbf{A} and \mathbf{b} are contained in the function argument – **Aaug**.

Aaug: an augmented matrix containing $[\mathbf{A} \mid \mathbf{b}]$ having N rows and $N+1$ columns, where N is the number of equations in the set.
 return **x**: the solution vector.

Write and call a `main()` function that uses your `Cramer` function to solve and print the solutions to the following sets of linear equations:

$$\begin{bmatrix} 3 & 1 & -1 \\ 1 & 4 & 1 \\ 2 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 12 \\ 10 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -10 & 2 & 4 \\ 3 & 1 & 4 & 12 \\ 9 & 2 & 3 & 4 \\ -1 & 2 & 7 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 2 \\ 12 \\ 21 \\ 37 \end{bmatrix}$$

Programs need to be work on 2c.

```

# region imports
import copy
import Determinant as myDet
# endregion

```

```

# region functions
def Cramer(Aaug):
    """
    This function uses the Cramer method to find the solution to the
    matrix equation  $A[x]=b$ 
    The Cramer method finds the elements of  $[x]$  by:
     $x[c] = \det(A_c)/\det(A)$ , where  $A_c$  is  $A$  with column  $c$  replace by  $b$ 
    I will use previous code from Determinant.py to calculate determinants
    I will create another function for replacing the  $c$  column of a matrix with  $b$ 
    :param Aaug: this is the augmented matrix  $A|b$ 
    :return: the transpose of the solution column vector  $[x]$  (i.e., a list)
    """
    # step 1: deep copy Aaug and strip off the b vector from Aaug
    A = # $JES Missing Code
    b = []
    for r in A: # scan through rows of A pop the last element and append it to b
        b.append(# $JES Missing Code)

    # step 2: calculate determinant of A
    D = myDet.Determinant(A) # this determinant gets used over and over, so just
calculate once
    x = [] # create a place to store the solution
    colCntr = 0 # a counter for which column is being replaced
    for r in A:
        AA = # $JES Missing Code # deep copy A
        AA = # $JES Missing Code # replace column colCntr with b
        x.append(myDet.Determinant(AA)/D)
        colCntr+=1
    return x

def getA(Aaug):
    """
    This function removes the last column from an augmented matrix from  $Ax=b$ 
    :param Aaug: The augmented matrix  $A|b$ 
    :return: The A matrix
    """
    A=copy.deepcopy(Aaug)
    for r in A:
        r.pop(len(r)-1)
    return A

def replaceCol(A, b, col):
    """
    This function replaces a column of A with b.
    :param A: a matrix
    :param b: transpose of solution column vector (i.e., a list)
    :param col: the index of column to be replaced
    :return: a new matrix
    """
    AA = # $JES Missing Code # make a deep copy of A
    #check to ensure len(b) == len(AA)
    rows = len(AA)
    if rows == len(b): # compatible sizes
        row = 0 # a counter for which row
        for r in AA: # for each row in AA, replace element col with b[row]
            r[col] = # $JES Missing Code
            row += 1 # increment row counter
    return AA

def checkAns(A,x):
    """
    I want to check to see if answer vector x is correct and matrix multiplication gives

```

```
b
    :param A: A square matrix
    :param x: The solution vector transpose (a row vector)
    :return: The b vector transpose (a row vector)
    """
    b=[]
    for r in A:
        s=0
        cCntr=0
        for c in r:
            s += c*x[cCntr]
            cCntr += 1
        b.append(s)
    return b

def main():
    """
    This function tests the Cramer method to find the solution to a matrix equation
    :return: nothing
    """
    # Step 1: Create the matrix equation(s)
    aug1 = [[3, 1, -1, 2], [1, 4, 1, 12], [2, 1, 2, 10]]
    aug2 = [[1, -10, 2, 4, 2], [3, 1, 4, 12, 12], [9, 2, 3, 4, 21], [-1, 2, 7, 3, 37]]

    # Step 2: Call Cramer to get transpose of [x]
    x1 = Cramer(aug1)
    x2 = Cramer(aug2)

    # Step 3: Print the transpose of [x]
    for r in aug1:
        print(r)
    print("x1^T=", x1)
    print("b=", checkAns(getA(aug1),x1))
    for r in aug2:
        print(r)
    print("x2^T=", x2)
    print("b=", checkAns(getA(aug2),x2))

# endregion

# region function call(s)
if __name__ == "__main__":
    main()
# endregion
```