CSP Create Task

The code is split into 3 files each one is at the listed page number in this document.

Main.py

```
#import other files we programmed and the turtle and tkinter libraries
from turtle import bgcolor
import functionClass as graphCalc
import guiClass as guiElements
import tkinter as tk

##################
#CREATE TK WINDOW#
##################
#create main Tkinter window
mainWindow = tk.Tk()
#make it white and be 1000 wide by 800 tall
mainWindow.config(bg="white")
mainWindow.geometry("1000x800")


#############################
# CREATE COORDINATE PLANE ###
#############################

#Create a coordinate plane on the main tkinter window.
#At coordinates 0,0 on a canvas
mainPlane = graphCalc.coordinatePlane(mainWindow,0,0,500,500,3,1)


######################
# CREATE OUTPUT LABEL#
#####################

#create and place a label that will contain the outputs of some calculations
outPutLabel= tk.Label(mainWindow, text= "Output will appear here")
outPutLabel.pack()
outPutLabel.place(x=50,y=50)


###################
#CREATE FUNCTION BOXES
```

```python
###################

#create a list that will contain the functionEntryBoxes, this will be recalled when an operation chooses "function1" or 2 or 3


#create a function box with a blue color at x=0,y=200
fncBox1= guiElements.functionEntryBox(mainWindow,0,200,1,mainPlane,"dodger blue")


#create a green second function Box just below it at x=0 y=300
fncBox2= guiElements.functionEntryBox(mainWindow,0,300,2,mainPlane,"green")


#create a third red function box just below the other two at x=0 y=400
fncBox3= guiElements.functionEntryBox(mainWindow,0,400,3,mainPlane,"red")


fncList=[fncBox1,fncBox2,fncBox3]


##############################
#CREATE MENUS FOR CALCULATIONS
##############################

#Create derivative menu at x=300, y =0
derivativeMenu= guiElements.derivativeMenu(mainWindow,"Derivative",300,0, outPutLabel, fncList)

#create integral menu at x=500, y=0
integralMenu= guiElements.integralMenu(mainWindow,"Integral",500,0, outPutLabel, fncList)

#create rotation menu at x= 800, y=0
rotationmenu= guiElements.rotationMenu(mainWindow,"Rotate",800,0, outPutLabel, fncList)

#tell the plane to draw itself
mainPlane.createGraph()

#Run mainloop so the code actually starts
mainWindow.mainloop()
```

[redacted]

functionClass.py

```
#import required libraries
import turtle as trtl
import math as math
import tkinter as tk

#Create a coordinate plan class
#this will draw and label the axis's and keep track of the visible x range and y range


class coordinatePlane:


    #range of x values visible
    xRange=(-5,5)
    #range of coordinates on y axis
    yRange=(-5,5)


    #function to initialize every instance of a coordinate plane
    #WHENEVER A COORDINATE PLANE IS CREATED THE __INIT__ FUNCTION WILL RUN TO INITIALIZE IT

    #parameters:
    #self = the current instance of the object, can be ignored in the constructor
    #master = tkinter window to place the coordinate plane on
    #originx= where the origin should be placed, measured in pixels
    #originy where the origin should be placed measured in pixels
    #pixel width = the width of the coordinate plane in pixels
    #pixel height = the height of the coordinate plane in pixels
    #axiswidth =  the thickness of the lines for the axis
    #gridwidth =  the thickness of the grid lines
    def __init__(self, master, originx,originy,pixelwidth,pixelheight,axiswidth,gridwidth):
        ##########################################
        #TURN INPUTS INTO PROPERTIES OF THE OBJECT
        ##########################################
        #coordinates of the origin of the plane
        self.originx=originx
```

```
        self.originy=originy
        #the width and height of the plane measured in pixels
        self.pixelwidth=pixelwidth
        self.pixelheight=pixelheight
        #the thickness of the axis
        self.axissize=axiswidth
        #the thickness of the grid
        self.gridsize=gridwidth
        #calculate the conversion for coordinates to pixels.
        #Take the pixel width of the coordinate plane and divide it by how many tiles there need to be
        self.scaleFactor=self.pixelwidth/(self.xRange[1]-self.xRange[0])


        #############################
        # SETUP CANVAS AND TURTLE
        ############################



        self.master=master

        #create a canvas on the tkinter window
        self.canvas = tk.Canvas(master)
        #make the canvas the size of the coordinate plane + 100 pixels in the horizontal and vertical
direction
        self.canvas.config(width=pixelwidth+100, height=pixelheight+100)
        #put the canvas on the right side of the window
        self.canvas.pack(side=tk.RIGHT)

        #create a turtleScreen on the canvas for the turtle to draw on
        self.wn = trtl.TurtleScreen(self.canvas)
        #create a turtle to draw on the canvas
        self.graphingTurtle=trtl.RawTurtle(self.wn)
        self.graphingTurtle.color("black")
        #hide this turtle from view
        self.graphingTurtle.hideturtle()



    ###############################################
    # METHODS TO CHANGE X and Y RANGES OF THE PLANE
```

```
    ###############################################

    #method to change the x range of the coordinate plane
    #Parametrs:
    # a= lower x range
    # b= upper x range
    def setxRange(self,a,b):
        self.xRange=(a,b)

    #method to change the y range of the coordinate plane
    #Parameters:
    # a= lower y range
    # b= upper y range
    def setyRange(self,a,b):
        self.yRange=(a,b)


    ############################
    #TURTLE MOVEMENT METHODS
    ############################

    #method to move turtle relative to the origin of the coordinate plane, instead of the origin of the
canvas
    #Parameters:
    # desX= desired x on the coordinate plane in pixels
    # desY= desired y on the coordinate plane in pixels
    def moveRelative(self,desX,desY):
        self.graphingTurtle.goto(self.originx+desX,self.originy+desY)


    ##########################
    # PLANE DRAWING METHODS
    ##########################



    #####
    #MAKING THE X AXIS
    ####
    #Method to draw axis's on the coordinate plane
```

```python
    def drawAxis(self):

        #takes the turtle from __init__ and assigns it to be called "grapher"
        grapher=self.graphingTurtle

        #stops tracking turtle's movements so it all appears at once
        self.wn.tracer(False)

        #pen size changes to given axis thickness
        grapher.pensize(self.axissize)

        #pen up so no line is drawn as it moves to the coordinate plane
        grapher.penup()

        #moves to origin of coordinate plane
        self.moveRelative(0,0)

        #prepares to draw
        grapher.pendown()

        #draw x axis
        #go all the way to the right
        self.moveRelative(self.pixelwidth/2,0)
        #go all the way to the left
        self.moveRelative(-self.pixelwidth/2,0)

        #draws y axis
        #Stop drawing as it goes to the top of the Y-Axis
        grapher.penup()
        self.moveRelative(0,self.pixelheight/2)

        #go all the way down drawing the Y-Axis
        grapher.pendown()
        self.moveRelative(0,-self.pixelheight/2)
        #start tracking so it all appears again
        self.wn.tracer(True)
```

```python
    #method to label X axis
    def LabelXAxis(self):



        #make it so when grapher is mentioned it references the turtle made in __init__
        grapher=self.graphingTurtle
        #stop drawin while turtle goes to left x axis
        grapher.penup()
        grapher.pensize(self.axissize)
        #stop tracking what the turtle is doing so it all instantly appears at the end
        self.wn.tracer(False)
        #turtle move to left of x-axis
        self.moveRelative(self.xRange[0]*self.scaleFactor,0)

        #create a variable to iterate over every integer on the x range
        iterator=self.xRange[0]

        #while the iterator hasn't reached the rights side of the x range
        while(iterator<=self.xRange[1]):
            #go to where the iterator is (next tick mark location)
            self.moveRelative(iterator*self.scaleFactor,0)
            #prepare to draw
            grapher.penup()

            #LABEL WITH NUMBER
            #go 0.1 to the right and up
            self.moveRelative( (iterator+0.1)*self.scaleFactor,0.1 *self.scaleFactor)
            #label with number
            grapher.write(iterator, align="left", font=("Arial", 9))

            #DRAW TICK MARK
            #go up 0.2 (on the coordinate plane) to the top of the tick mark
            self.moveRelative(iterator*self.scaleFactor,0.2*self.scaleFactor)
            grapher.pendown()

            #down to -0.2 (on the coordinate plane) to the bottom of the tick mark
            self.moveRelative(iterator*self.scaleFactor,-0.2*self.scaleFactor)
            #stop drawing as it prepares to go to the next tick mark
```

```
        grapher.penup()
        #increment iterator so it goes to the next integer to draw a tick mark
        iterator+=1

    #get ready to draw
    grapher.pendown()
    #go to the right side of the x-axis
    self.moveRelative(self.xRange[1]*self.scaleFactor,0)
    #activate tracer again so it all appears
    self.wn.tracer(True)

#####
#MAKING THE Y AXIS
####

# method to label y-axis
def LabelYAxis(self):
    #stops tracking drawings so it all appears at once
    self.wn.tracer(False)
    #make it so when grapher is mentioned it's the turtle associated with the coordinate plane
    grapher=self.graphingTurtle
    grapher.pensize(self.axissize)
    grapher.penup()
    #move it to the bottom of the y axis
    self.moveRelative(self.yRange[0]*self.scaleFactor,0)
    #set iterator to bottom of y axis
    iterator=self.yRange[0]
    #iterate over every integer from the bottom of the y range to the top of the y range
    while(iterator<=self.yRange[1]):
        # go to new height we iterated to and then 0.1 up and to the right
        self.moveRelative(0.1*self.scaleFactor,(iterator+0.1)*self.scaleFactor)
        #place the number labelling that height
        grapher.write(iterator, align="left", font=("Arial", 9))

        #stop drawing
        grapher.penup()

        #DRAW TICK MARK
        #go 0.2 to the right of the height, which is the right of the tick mark
```

```
            self.moveRelative(0.2*self.scaleFactor,iterator*self.scaleFactor)
            #start drawing
            grapher.pendown()
            #go to the left of the tick mark
            self.moveRelative(-0.2*self.scaleFactor,iterator*self.scaleFactor)
            grapher.penup()
            #update the iterator so we go to the next tick
            iterator+=1

        #move to last tick
        grapher.pendown()
        self.moveRelative(0,self.yRange[1]*self.scaleFactor)
        #reactivate tracer to show all the ticks
        self.wn.tracer(True)

    #method to add grid lines to coordinate plane
    def addGridLines(self):
        #set iterator to left of x axis
        iterator=self.xRange[0]

        #set up turtle
        grapher=self.graphingTurtle
        grapher.pensize(self.gridsize)
        #change pencolor to gray so lines appear less prominently
        grapher.color("Gray")
        #stop tracer so the viewer doesn't see the turtle draw, they only see the result
        self.wn.tracer(False)

        #iterate over every integer in the x range
        while(iterator<=self.xRange[1]):

            grapher.penup()
            #go to the current integer
            self.moveRelative(iterator*self.scaleFactor,0)
            #draw up and down grid lines
            # this is so a grid line isn't drawn at x=0, because the axis is already there
            if (not iterator==0):
                grapher.pendown()
            #go all the down and then all the way up to draw the grid line
```

```
            self.moveRelative(iterator*self.scaleFactor,self.yRange[0]*self.scaleFactor)
            self.moveRelative(iterator*self.scaleFactor,self.yRange[1]*self.scaleFactor)

            #increase the iterator by one so that we go to the next integer on the grid line
            iterator+=1

        #set the iterator to the bottom of the y range
        iterator = self.yRange[0]
        #increment over every integer on the y range
        while(iterator<=self.yRange[1]):
            grapher.penup()
            #go to the current y
            self.moveRelative(0,iterator*self.scaleFactor)

            # don't draw if we're at y=0
            if(not iterator==0):
                grapher.pendown()
            #go all the way left and right to draw the grid line
            self.moveRelative(self.xRange[0]*self.scaleFactor,iterator*self.scaleFactor)
            self.moveRelative(self.xRange[1]*self.scaleFactor,iterator*self.scaleFactor)
            #increment iterator so we go to the next y value
            iterator+=1

        #activate tracer so the drawings all appear again
        self.wn.tracer(True)
        #set graphing color back to black
        grapher.color("black")


    #PUTTING IT ALL TOGETHER
    #method to combine all the previous methods to draw an entire coordinate plane
    def createGraph(self):
        self.drawAxis()
        self.addGridLines()
        self.LabelXAxis()
        self.LabelYAxis()
```

```
################
#EQUATION CLASS#
################

#create an equation class so we can use the same methods to calculate and graph multiple equations
class equation:

    #spacing between points, a smaller number means more points which means a higher resolution for the graph
    #0.01
    deltaX=0.01

    #When an equation instance is created this function will run to initialize it

    #Parameters:
    #self, the current instance of the object
    #equation, a string of the equation that is supposed to be graphed
    #coordPlane, the coordPlane that it is supposed to be graphed on
    #function color, the color that the equation should appear as on the plane
    def __init__(self,equation,coordPlane, functioncolor):
        #the actual equation itself
        self.equation=equation
        #a list of tuples where the first member is the x coordinate and the second is the y coordinate
        self.coordinates=[]
        #the plane that it will be graphed on
        self.coordPlane=coordPlane
        #Get xRange,YRange, and scale factor from the target coordinate plane
        self.xRange=coordPlane.xRange
        self.yRange=coordPlane.yRange
        self.scaleFactor=coordPlane.scaleFactor
        #create a turtle to graph the equation on the target coordinate plane
        self.graphingTrtl=trtl.RawTurtle(self.coordPlane.wn)
        self.graphingTrtl.penup()
        self.graphingTrtl.pencolor(functioncolor)
        self.graphingTrtl.pensize(5)
        self.graphingTrtl.hideturtle()
        self.functioncolor=functioncolor
```

```
    #method to change the equation that is being graphed
    def setEq(self,newEq):
        self.equation=newEq

    #method to move turtle relative to origin of coordinate plane instead of origin of canvas
    def moveRel(self,wantedX,wantedY):
        origX=self.coordPlane.originx
        origY=self.coordPlane.originy
        self.graphingTrtl.goto(origX+wantedX,origY+wantedY)


    #######################
    #CALCULATING COORDINATES
    #######################
    #method to use equation to calculate coordinates
    def calculateCoords(self, shouldExtend=False):

        #a list to hold the coordinates so they can be graphed later
        self.coordinates=[]

        #a variable to set how far beyond the xRange should be calculated so that when the graph is
rotated it stretches to fit
        extendVal = 0

        if(shouldExtend):
            extendVal=3


        #iterate over the visible x range and a bit further so when the function is rotated it can
stretch to fit the graph
        iterator=self.xRange[0]-extendVal
        while(iterator<=self.xRange[1]+extendVal):
            #set x equal to the one we've iterated to
            x=iterator

            try:
                #use eval to run the function string as actual python code, then take the output and
put it into the current y
```

```
                currY= eval(self.equation)
                #check that the y calculated is an integer or float and not a complex
                if( type(currY) == int or type(currY) == float ):
                    #if it's good then add the x, and calculated y to the coordinates list
                    self.coordinates.append((iterator,currY))
            #if when calculating we run into a value error, (like plugging a negative into math.log)
then it prints, "exception handled" and keeps calculating instead of stopping
            except ValueError:
                    print("Exception Handled")
            #go to the x a little bit to the right
            iterator+=self.deltaX

    ####################
    # MENU CALCULATIONS
    ###################
    def calculateDerivative(self, chosenX):
        #take an x that's a little bit less then the desired one
        lowerx=chosenX-0.00000001
        #take an that's a little bit more than the desired one
        upperX=chosenX+0.00000001

        #Use these to calculate their corresponding Y-Values
        x=lowerx
        lowerY=eval(self.equation)

        x=upperX
        upperY=eval(self.equation)

        #print out these values in the terminal for debugging purposes
        print(lowerx,lowerY,upperX,upperY)

        #Calculate the slope using Y1-Y2 divided by X1-X2. This is the limit definition of the
derivative
        return (upperY-lowerY)/(upperX-lowerx)


    def calculateIntegral(self,lowerX,UpperX):
        #set the current x to the lower bound
        currentX=lowerX
```

```
    #set the dx in the integral term
    deltaX=0.00001
    #variable to keep track of sum
    totalIntegral=0
    #keep going until the upper bound is reached
    while(currentX<UpperX):
        #set x = the current one
        x=currentX
        #calculate the y at that x
        y=eval(self.equation)
        #increase the currentX by the change
        currentX+=deltaX
        # find the area of the dX by Y rectangle and add it to the total integral variable
        totalIntegral += deltaX * y
    #return the value of the total integral
    return totalIntegral

#use to rotate a graph
def calculateRotatedPoints(self,angle):
    #convert the desired angle from degrees to radians
    desiredAngle= angle*math.pi/180
    #create a list to keep track of the points post rotation
    newPoints=[]
    #iterate over every point plotted
    for point in self.coordinates:
        #get it's x and y coordinates
        currentX=point[0]
        currentY=point[1]
        #use the rotation matrix formula to rotate it a certain angle
        newX=currentX*math.cos(desiredAngle) - currentY*math.sin(desiredAngle)
        newY=currentX*math.sin(desiredAngle) + currentY*math.cos(desiredAngle)
        #add that to the new list
        newPoints.append((newX,newY))

    #update the equations coordinates with the new points
    self.coordinates=newPoints

####################
#GRAPHING THE POINTS
```

```
    ####################

    def plotCoords(self):
        #clear any previous graph the turtle may have drawn
        self.graphingTrtl.clear()
        self.graphingTrtl.penup()

        #iterate over every coordinate and make the turtle go there
        for point in (self.coordinates):
            #get the coordinates of the curren point
            Xpoint=point[0]
            Ypoint=point[1]
            #if the point is outside the coordinate plane range, penup so the turtle doesn't draw it
            if(Ypoint<self.coordPlane.yRange[0] or Ypoint>self.coordPlane.yRange[1] or
Xpoint<self.coordPlane.xRange[0] or Xpoint>self.coordPlane.xRange[1]):
                self.graphingTrtl.penup()

            #scale the coordinates up to pixels
            Xpoint*=self.scaleFactor
            Ypoint*=self.scaleFactor
            #turn of tracer so the user does not see these being drawn
            self.coordPlane.wn.tracer(False)

            #make the turtle go to the point
            self.moveRel(Xpoint,Ypoint)
            #get ready to draw again
            self.graphingTrtl.pendown()

        #activate the tracer so all the drawings appear at once
        self.coordPlane.wn.tracer(True)
```

guiClass.py

```
import tkinter as tk
import functionClass as graphCalc

#making a class that creates a function box
class functionEntryBox:



    #when a function entry box is made create the things
    #parameters:
    # wn = tk window where the boxes should be
    # place X, the coordinate it should be placed on
    # place Y, the y coordinate it should be placed on
    # function Num, the number associated with the function (Ex: function 1, 2 or 3)
    #target plane, the coordinate plane the function will be drawn on
    #function color, the color the function will be in the menu and when graphed
    def __init__(self, wn, placeX,placeY, functionNum, targetPlane, functionColor):
        #set function num property as the given one
        self.functionNum=functionNum

        #create a frame where the function box elements will be contained
        self.fncFrame = tk.Frame(wn, pady=5,  bg="black")
        self.fncFrame.pack()
        self.fncFrame.place(x = placeX, y = placeY)
        #create a label that will say Function 1: or Function 2:
        self.fncLabel = tk.Label(self.fncFrame, text="Function" + str(functionNum) + ": ",
   compound="center",
   font=("comic sans", 12),
   bd=0,
   relief=tk.FLAT,
   fg=functionColor,
   bg="black")

        self.fncLabel.pack(side=tk.LEFT)

        #Create an entry box so the user can type in their function
        self.fncEntry= tk.Entry(self.fncFrame, font=("comic sans", 10), width = 18)
        self.fncEntry.pack(side=tk.LEFT, padx=(0, 5))
```

```
        #create a button so the user can click on it to submit their equation
        #When clicked it will update the equation
        self.EnterBtn = tk.Button(wn, text="Enter", bd = '2', command= self.updateEq )
        self.EnterBtn.pack()
        self.EnterBtn.place(x = placeX+245, y = placeY+5)

        #Create an equation object for this function box to submit to.
        #It defaults to x**2 but the user will never notice this
        self.targetEq=graphCalc.equation("x**2", targetPlane, functionColor)

    #Method to update the equation on the graph
    def updateEq(self):
        #it gets what the user typed in the box
        newEq = self.fncEntry.get()
        #sets the target equation to the one typed in
        self.targetEq.setEq(newEq)
        #calculates the new coordinates and plots them
        self.targetEq.calculateCoords(True)
        self.targetEq.plotCoords()

######################
# MENU CLASSES
#####################

#Derivative menu

class derivativeMenu:

    #When a derivative menu is created this function will run to initialize it
    #Parameters:
    #self, the current instance of the derivative menu, can be ignored when constructing
    # wn, the tkinter window the derivative menu should be in
    #placeX, the x coordinate it should be placed on
    #placeY, the y coordinate it should be placed on
    #outputLabel, the label where it will print the calculated derivative
    #fncList, a list of all functionboxes so when the user types in function 1 2 or 3, the menu knows
what that means
    def __init__(self,wn, title, placeX,placeY, outputLabel , fncList):
```

```
        self.title=title

        #create a frame the menu will be located in
        self.menuFrame = tk.Frame(wn, pady=5,  bg="cyan")
        self.menuFrame.pack(side=tk.TOP)
        self.menuFrame.place(x = placeX, y = placeY)

        #Create a label that says this is the derivative menu
        self.menuLabel = tk.Label(self.menuFrame, text=title,font=("comic sans", 10), bd=0, bg="white")
        self.menuLabel.grid(row=0,column=1)

        ###############################
        # CHOOSING WHICH FUNCTION TO TAKE THE DERIVATIVE OF
        #############################

        #Label saying, this is the textbox to type in to choose the textbox
        self.fncChooserLabel = tk.Label(self.menuFrame, text="Function",font=("comic sans", 10), bd=0,
bg="white")
        self.fncChooserLabel.grid(row=1,column=0)

        #Box to actually type in to choose the function

        self.fncChooserEntry= tk.Entry(self.menuFrame, font=("comic sans", 15), width = 1)
        self.fncChooserEntry.grid(row=2,column=0)


        ######################################
        # CHOOSING XVal to take derivative at
        ######################################

        #label saying, this is the place to type in the x value
        self.fncxValLabel = tk.Label(self.menuFrame, text="X-Value:",font=("comic sans", 10), bd=0,
bg="white")
        self.fncxValLabel.grid(row=1,column=2)

        #box to type in to choose xValue
        self.fncxValEntry= tk.Entry(self.menuFrame, font=("comic sans", 15), width = 1)
        self.fncxValEntry.grid(row=2,column=2)
```

```
       #button to submit and say "take the derivative"
        self.SubmitBtn = tk.Button(self.menuFrame, text="Submit", bd = '2',
command=self.whenButtonClicked )
        self.SubmitBtn.grid(row=2,column=1)

        #save label to output to
        self.outPutLabel=outputLabel
        #save list of function boxes
        self.fncList=fncList

    ##################################
    #What Happens when Button Clicked#
    ##################################

    def whenButtonClicked(self):
        #the chosen function number, take it from the entry box
        chosenNum = self.fncChooserEntry.get()
        chosenNum=int(chosenNum)
        #the equation we're taking the derivative of, doing -1 because first in list is [0] but we want
1 to be first function to type in
        currentEquation = self.fncList[chosenNum-1]
        #get chosen x coordinate
        xCor= float(self.fncxValEntry.get())
        #get the derivative called
        derivativeValue= currentEquation.targetEq.calculateDerivative(xCor)
        #print for debugging purposes
        print(derivativeValue)
        #round it to 4 decimal places
        derivativeValue=round(derivativeValue,4)
        #create a string to output on the label
        outputString = "The derivative of function " + str(chosenNum)+"\n when x=" +
self.fncxValEntry.get()+"\n is "+ str(derivativeValue)
        #change the text of the label to that string
        self.outPutLabel.config(text=outputString)


###############
#INTEGRAL MENU
###############
```

```
class integralMenu:

    #when the integral menu is created
    def __init__(self,wn, title, placeX,placeY, outputLabel , fncList):

        #save the parameters as properties of the object
        self.title=title

        #create a fram for the menu to sit in
        self.menuFrame = tk.Frame(wn, pady=5,  bg="cyan")
        self.menuFrame.pack(side=tk.TOP)
        self.menuFrame.place(x = placeX, y = placeY)

        #Create label on the menu saying it's the integral menu
        self.menuLabel = tk.Label(self.menuFrame, text=title,font=("comic sans", 10), bd=0, bg="white")
        self.menuLabel.grid(row=0,column=1)

        #########################
        # BOX TO CHOOSE FUNCTION
        #########################

        #Label to say this box is the one to choose functions
        self.fncChooserLabel = tk.Label(self.menuFrame, text="Function",font=("comic sans", 10), bd=0,
bg="white")
        self.fncChooserLabel.grid(row=1,column=0)


        #create box to type in to choose function
        self.fncChooserEntry= tk.Entry(self.menuFrame, font=("comic sans", 15), width = 1)
        self.fncChooserEntry.grid(row=2,column=0)

        #############################
        # BOX TO CHOOSE LOWER X-BOUND
        ##############################

        #label the lower x box
        self.fnclowerxValLabel = tk.Label(self.menuFrame, text="Lower X-Value:",font=("comic sans",
10), bd=0, bg="white")
```

```
        self.fnclowerxValLabel.grid(row=1,column=2)

        # create box to type in the lower x value
        self.fnclowerxValEntry= tk.Entry(self.menuFrame, font=("comic sans", 15), width = 1)
        self.fnclowerxValEntry.grid(row=2,column=2)

        ########################
        #CHOOSE HIGHER X-BOUND
        ########################

        #label upper x box
        self.fncupperxValLabel = tk.Label(self.menuFrame, text="Upper X-Value:",font=("comic sans",
10), bd=0, bg="white")
        self.fncupperxValLabel.grid(row=1,column=3)

        #create box to type in upper x box
        self.fncupperxValEntry= tk.Entry(self.menuFrame, font=("comic sans", 15), width = 1)
        self.fncupperxValEntry.grid(row=2,column=3)

        ######################
        # BUTTON TO SUBMIT
        #####################
        self.SubmitBtn = tk.Button(self.menuFrame, text="Submit", bd = '2',
command=self.whenButtonClicked )
        self.SubmitBtn.grid(row=2,column=1)

        #save label to output to
        self.outPutLabel=outputLabel

        #save list of functions that can be accessed
        self.fncList=fncList

    #FUNCTION to respond to button press
    def whenButtonClicked(self):

        #get the function the user chose
        chosenNum = self.fncChooserEntry.get()
        chosenNum=int(chosenNum)
        currentEquation = self.fncList[chosenNum-1]
```

```
        #get the lower x bound from the box
        lowerxCor= float(self.fnclowerxValEntry.get())
        #get the higher x bound from the box
        upperxCor= float(self.fncupperxValEntry.get())

        #get the value of the integral using the calculateIntegral method
        integralValue= currentEquation.targetEq.calculateIntegral(lowerxCor,upperxCor)

        #print it into the terminal for debugging purposes
        print(integralValue)
        #round it to 4 decimal places
        integralValue=round(integralValue,4)
        #set the label to the desired message
        outputString = "The integral of function " + str(chosenNum)+"\n from x=" + str(lowerxCor)+" to
x="+ str(upperxCor)+"\n is "+ str(integralValue)

        self.outPutLabel.config(text=outputString)
################
# ROTATION MENU
################

class rotationMenu:
    def __init__(self,wn, title, placeX,placeY, outputLabel , fncList):
        self.title=title
        #create and setup a frame for the menu to sit in
        self.menuFrame = tk.Frame(wn, pady=5,  bg="cyan")
        self.menuFrame.pack(side=tk.TOP)
        self.menuFrame.place(x = placeX, y = placeY)

        #create a label to say that this is the rotation menu
        self.menuLabel = tk.Label(self.menuFrame, text=title,font=("comic sans", 10), bd=0, bg="white")
        self.menuLabel.grid(row=0,column=1)

        #FUNCTION CHOOSER BOX

        #label it
        self.fncChooserLabel = tk.Label(self.menuFrame, text="Function",font=("comic sans", 10), bd=0,
bg="white")
```

```
        self.fncChooserLabel.grid(row=1,column=0)

        #create box to type into
        self.fncChooserEntry= tk.Entry(self.menuFrame, font=("comic sans", 15), width = 1)
        self.fncChooserEntry.grid(row=2,column=0)



        #CHOOSE ANGLE TO ROTATE BY

        #label the box as the place to enter an angle
        self.fncangleLabel = tk.Label(self.menuFrame, text="Angle:",font=("comic sans", 10), bd=0,
bg="white")
        self.fncangleLabel.grid(row=1,column=2)

        #create box to type angle into
        self.fncangleEntry= tk.Entry(self.menuFrame, font=("comic sans", 15), width = 3)
        self.fncangleEntry.grid(row=2,column=2)

        #Button to submit angle rotation
        self.SubmitBtn = tk.Button(self.menuFrame, text="Submit", bd = '2',
command=self.whenButtonClicked )
        self.SubmitBtn.grid(row=2,column=1)

        #save label to output to
        self.outPutLabel=outputLabel
        #list of functions so the fncChooser can get the right function
        self.fncList=fncList


    #WHEN BUTTON CLICKED

    def whenButtonClicked(self):
        #get the function the user chose
        chosenNum = self.fncChooserEntry.get()
        chosenNum=int(chosenNum)
        currentEquation = self.fncList[chosenNum-1]

        #get the angle the user chose
```

```
chosenAngle= float(self.fncangleEntry.get())
#calculate the newly rotated points
currentEquation.targetEq.calculateRotatedPoints(chosenAngle)
#plot the new points
currentEquation.targetEq.plotCoords()
#output to the label that the function has been rotated
outputString = "The function has been rotated"
self.outPutLabel.config(text=outputString)
```