**Lab #1 Report**

**Introduction:**

For this first lab assignment we were asked to complete a set of figures that would be changing their forms by using recursive methods. The first one was to create n numbers of circles and create the next circle smaller than the first one without losing the center. The second one was to create n numbers of squares in which each square had other 4 squares (in the bottom-right corner, bottom left corner, upper-right corner, and in the upper-left corner). The third one was a circle that had another 5 circles in it (center, left, right, top, and bottom). And the last one was to create a tree in which the next node will be moved half of the original coordinate to the left and the right to each side. The task here was to use recursive methods and with the matplotlib library to draw the lines.

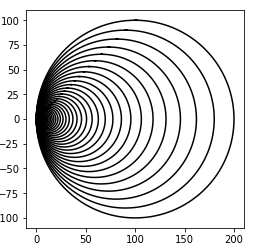
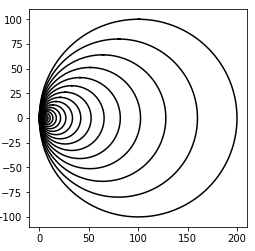
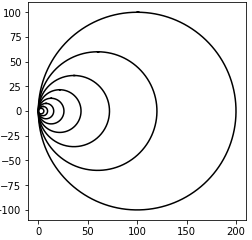
**Task #1: First Circle Challenge**

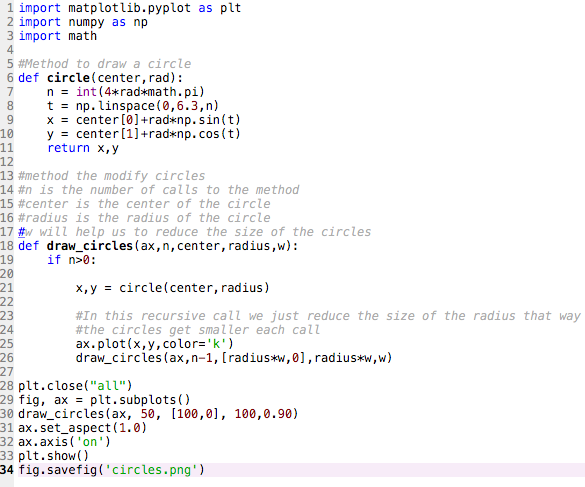
In this first part of the lab we were assigned to create a circle with a certain center and a certain radius, and be reducing the radius that way we keep the same center. In this case, first I used as parameters *ax* which is the variable I am using to create lines, *n* which is the number the program calls the method, *center* which is the center of the circle, *radius* which is the radius of the circle, and *w* which is the value we will be using to reduce the radius.

What I did first was that I used the circle method to create a circle by using the “circle” method. Then I created another method called *draw\_circles* with the parameters explained above and start with the base case, so if n > 0 then will run the method. Secondly, I called the circle method with the parameters *center* and *radius*. As the final step I just opened the *ax.plot* that draws the lines and make the recursive call after it. To get the correct output, in the recursive call I modified the center by changing the x-axis value. I changed *center* to *[radius\*w,0]* that way the center moves proportionally to the left and also change *radius* to *radius\*w* to reduce the length of the radius.

I tested the code with the next values: ﻿*draw\_circles(ax, n, [100,0], 100,w)*

When n = 6 and w = 0.6 when n = 50 and w = 0.8 when n = 50 and w = 0.9

**

﻿

This code has a running time of θ(n)

**Task #2: Drawing squares**

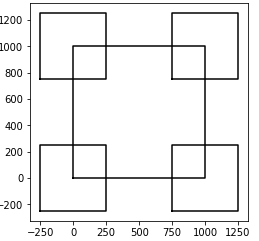
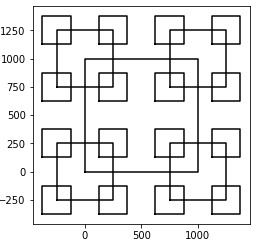
In this second task, the challenge was to create a method the makes a square that has another square, which is reduced, in the middle of each vertex. For this part of the lab I had the next parameters: *ax* which draws the lines, *n* which is the number the method will be called, *p* which has the coordinates for all vertices, and *w* which is the length between each vertex. To solve this problem, I found it easier to be creating square by square so in order to accomplish this I copies the values of *p* to 4 different new variables to be creating squares to each corner. I called the first variable UpRight which creates the upper-right square. What it does is that it moves each vertex from the original square to create a smaller circle in the upper-right corner. For example, when *UpRight[0] = Upright[0] + w\*.75* will move the vertex at p[0][0] and p[0][1] w\*75 to the right and to the top, and will continue moving other vertices to create that square, and this same process will be repeated but with different operations to create the other squares.

I will be having 4 recursive calls. In each I will be using one of the new variables to create circles instead of using *p* and will be reducing the size of *w/2*

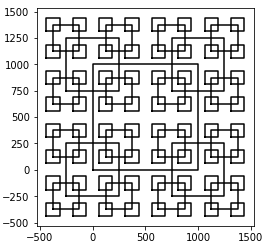
I tested my code with this with the original value of 1000 and *p* as: ﻿ *p =* *np.array([[0,0],[0,orig\_size],[orig\_size,orig\_size],[orig\_size,0],[0,0]])*

So when:

﻿draw\_squares(ax,1,p,orig\_size) ﻿draw\_squares(ax,2,p,orig\_size)

draw\_squares(ax,2,p,orig\_size)



﻿

import numpy as np

import matplotlib.pyplot as plt

#n is the number of recursive calls

#p are the coordinates of eache vertice

#w is the distance between each plot

def draw\_squares(ax,n,p,w):

#copying the arrays to each of the four squares

UpLeft = np.copy(p)

UpRight = np.copy(p)

DownLeft = np.copy(p)

DownRight = np.copy(p)

if n > 0:

#Moves vertices in different ways to create upper-right square

UpRight[0] = UpRight[0] + w\*.75

UpRight[1][0] = UpRight[1][0] + w\*.75

UpRight[1][1] = UpRight[1][1] + w\*.25

UpRight[2] = UpRight[2] + w\*.25

UpRight[3][0] = UpRight[3][0] + w\*.25

UpRight[3][1] = UpRight[3][1] + w\*.75

UpRight[4] = UpRight[4] + w\*.75

#Moves vertices in different ways to create bottom-right square

DownRight[0][0] = DownRight[0][0] + w\*.75

DownRight[0][1] = DownRight[0][1] - w\*.25

DownRight[1][0] = DownRight[1][0] + w\*.75

DownRight[1][1] = DownRight[1][1] - w\*.75

DownRight[2][0] = DownRight[2][0] + w\*.25

DownRight[2][1] = DownRight[2][1] - w\*.75

DownRight[3][0] = DownRight[3][0] + w\*.25

DownRight[3][1] = DownRight[3][1] - w\*.25

DownRight[4][0] = DownRight[4][0] + w\*.75

DownRight[4][1] = DownRight[4][1] - w\*.25

#Moves vertices in different ways to create bottom-left square

DownLeft[0] = DownLeft[0] - w\*.25

DownLeft[1][0] = DownLeft[1][0] - w\*.25

DownLeft[1][1] = DownLeft[1][1] - w\*.75

DownLeft[2] = DownLeft[2] - w\*.75

DownLeft[3][0] = DownLeft[3][0] - w\*.75

DownLeft[3][1] = DownLeft[3][1] - w\*.25

DownLeft[4] = DownLeft[4] - w\*.25

#Moves vertices in different ways to create upper-left square

UpLeft[0][0] = UpLeft[0][0] - w\*.25

UpLeft[0][1] = UpLeft[0][1] + w\*.75

UpLeft[1][0] = UpLeft[1][0] - w\*.25

UpLeft[1][1] = UpLeft[1][1] + w\*.25

UpLeft[2][0] = UpLeft[2][0] - w\*.75

UpLeft[2][1] = UpLeft[2][1] + w\*.25

UpLeft[3][0] = UpLeft[3][0] - w\*.75

UpLeft[3][1] = UpLeft[3][1] + w\*.75

UpLeft[4][0] = UpLeft[4][0] - w\*.25

UpLeft[4][1] = UpLeft[4][1] + w\*.75

#Recursive call for upper-right square.

ax.plot(UpRight[:,0],UpRight[:,1],color='k')

draw\_squares(ax,n-1,UpRight,w/2)

#Recursive call for bottom-right square

ax.plot(DownRight[:,0],DownRight[:,1],color='k')

draw\_squares(ax,n-1,DownRight,w/2)

#Recursive call for bottom-left square.

ax.plot(DownLeft[:,0],DownLeft[:,1],color='k')

draw\_squares(ax,n-1,DownLeft,w/2)

#Recursive call for upper-left square.

ax.plot(UpLeft[:,0],UpLeft[:,1],color='k')

draw\_squares(ax,n-1,UpLeft,w/2)

plt.close("all")

orig\_size = 1000

p = np.array([[0,0],[0,orig\_size],[orig\_size,orig\_size],[orig\_size,0],[0,0]])

fig, ax = plt.subplots()

ax.plot(p[:,0],p[:,1],color='k')

draw\_squares(ax,3,p,orig\_size) #we take the original size since in this case it means the length of the square

ax.set\_aspect(1.0)

ax.axis('on')

plt.show()

fig.savefig('squares.png')

this method has a running time of θ(n^4)

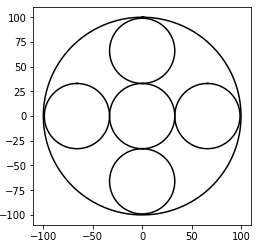
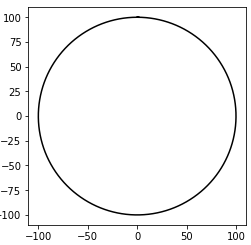
**Task #3: second circle challenge**

This, for me, it was the hardest one because I tried different ways to solve the problem but it took longer for me to get it. In this one, as it was explained in the beginning of this report, we needed to find a way that each circle had other 5 circles in it (one in the middle, left, right, top, and bottom). To complete this task, I reused the method to create circles to use it in the draw\_circles method, which is in charge of completing the task.

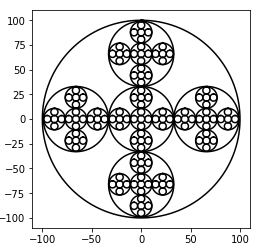
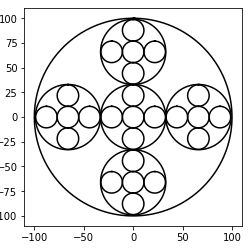
I used the same parameters as in task #1 but the use of *w* is different, here it help us the reduce the size of the radius and move the center to the sides, to the top or bottom.

I began with the base case so if n > 0 then the code will run. Inside I called the circle method to draw the initial circle and after that I started with the recursive calls. I used 5 with the purpose of having 5 circles (center, sides, top, and bottom). In all recursive calls we will be reducing the *radius\*0.33.* the challenge was that we had to modify the center depending on which side the circle was moving. For the middle one the center stays the same. For the sides ones we keep the same coordinate for the y-axis but we have to change the x-axis coordinate by adding (if it goes to the right) or subtracting (if it goes to the left) *center[0]+radius\*w\*2.* For the top and bottom one was the opposite, so we kept the same value for the x-axis but we modify the y-axis values so depending if the circle goes up or down, we add or subtract *center[1]*+ or *center[1] – radius\*w\*2.*

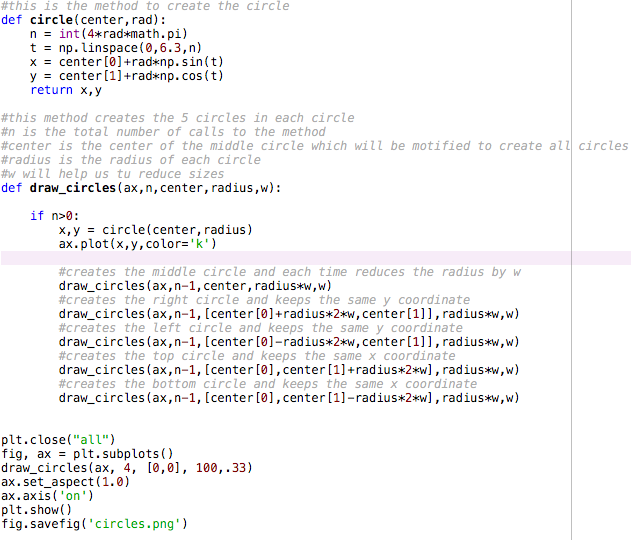
I tested my code with the next calls:

﻿draw\_circles(ax, 1, [0,0], 100,.33) ﻿draw\_circles(ax, 2, [0,0],100,0.33)

﻿draw\_circles(ax, 3, [0,0], 100,.33) ﻿draw\_circles(ax, 4, [0,0], 100,.33)



My code:



This code has a running time of θ(n^5)

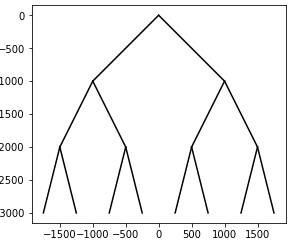
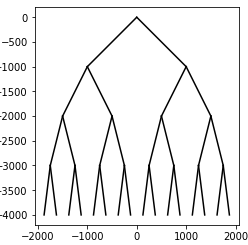
**Task #4: drawing the tree**

For this final task I first wanted to create 3 vertices and modify the first one and the last one to be the second vertex each time that the method got called, but my idea didn’t work at all. To get the correct figure I only created one point and the keep drawing lines to left and right. The issue here was that the second left and right lines had to be reduced the half of the first line on the x-axis. To accomplish this task I used these parameters: *ax* which draws the lines, *n* which is the number of calls to the method, *p* which is the coordinate of the point, xAxis and *yAxis* to be moving the point over the x and y axis, and *w* which will help us to reduce the *xAxis* of the left and right lines.

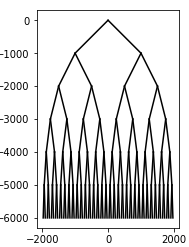
My first step was to give the same values of *p* to 2 new variables that will move the coordinates to the left and right. In this case they are named *RightLine* and *LeftLine*. Both of them will be subtracting (going down) the value of yAxis, that way all lines have the same height, but *RightLine* goes *xAxis* to the right and *LeftLine* goes *xAxis* to the left. The trick here is in the recursive call. There will be two recursive calls, one for *RightLine* and one for *LeftLine.* In both recursive calls we will divide *xAxis*/2 that way the lines are the half of the *xAxis* of the previous line.

I tested my cide with:

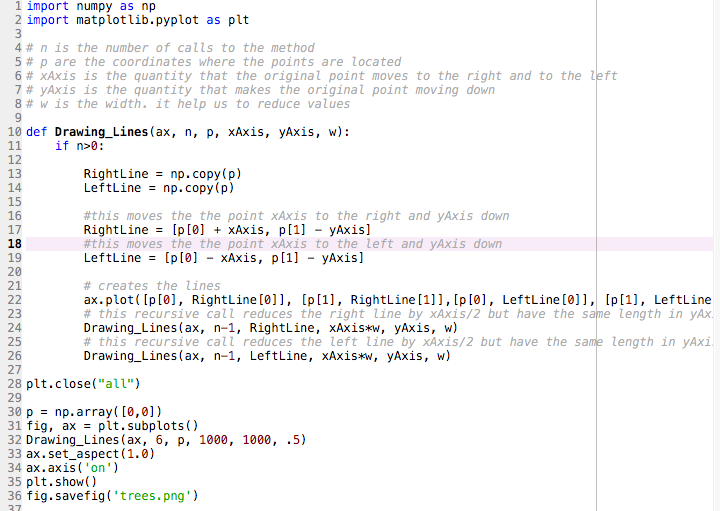
﻿Drawing\_Lines(ax, 3, p, 1000, 1000, .5). ﻿Drawing\_Lines(ax, 4, p, 1000, 1000, .5).

﻿Drawing\_Lines(ax, 6, p, 1000, 1000, .5).



My code is:



The running time of this code is θ(n^2).

**Conclusion**

I think that this lab was very helpful for us because:

* It encouraged us to learn a new language.
* It was a refresh form what we saw in last semester’s class.
* It opened us the panorama by not using only numbers or words but also drawing.

I can strongly say that I feel more confortable while working with recursive methods because now that I worked with them, I noticed that they can save us lot of time in coding while working out a problem.

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.