CS 2302

Nicole Favela

Lab 1 report

February 12, 2019

# Introduction

The problem I am faced with is to draw several shapes (or fractals) using recursion. Part 1 involves drawing squares using ﻿matplotlib and recursion to make smaller and smaller squares by manipulating the array inputs with every call. Part 2 involves drawing concentric circles with smaller radii and off-center origins. Part 3 involves drawing binary trees with the recursive calls corresponding to the height of the tree. Part 4 involves drawing nested circles with where each recursive call corresponds to new circles with different sizes.

**Solution and implementation**

I approached this problem by drawing out the coordinate axes and shapes on paper and determining the coordinates I would need to pass as parameters into the function. I also thought about the base cases required to make the recursive calls terminate. For the squares, I used the array provided and modified the coordinates in each recursive call. In order to make squares in all 4 corners, I copied the original array into temporaries, so I could move the coordinates without altering the original shapes. For the circles, my approach was similar. I changed the number of circles I wanted and the recursive calls so it would decrease the radius of the circles each time. My approach to making the trees was to subdivide the grid horizontally depending on the height of the tree, then to plot points equidistant from the midpoint of y = x, then with each recursive call subdivide again by ½ and changing delta x and delta y. My approach to the circle fractals was similar to the squares. Instead I modified the circles to make the 5 smaller ones fit inside, then based on the diameter of those, I made smaller circles with every recursive call. The next radius would be one third the original then the recursive calls would be moved by double that. For the trees, I made use of the absolute value function to get the correct distance for x and y. The recursive part of this took into account the height of the tree as a parameter in which was divided by the levels needed to create the different trees in part 3. I also chose to leave the graph function as a separate function for simplification and readability.

**Experimental results**

Part 1 recursive squares required a lot of trial and error to obtain the correct output image. An earlier test run of this program led me to obtain the following output:

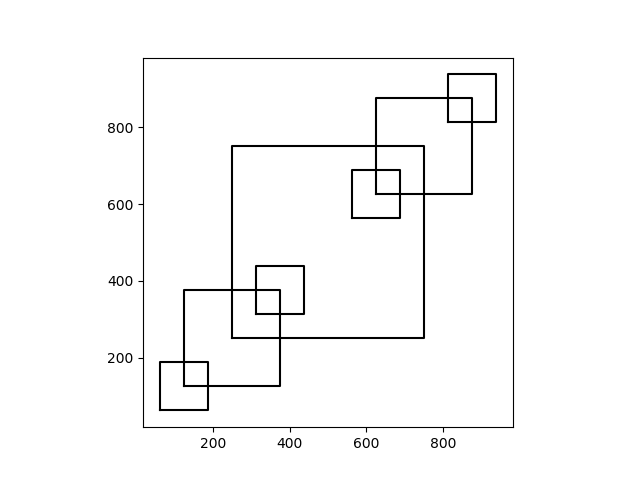


Figure 1

This result was obtained by using the parameters ﻿ax, n =3, p (an array), and w. My initial approach did not modify the code vastly from the given code. The array p did change to start at [250,250] and go to 750. I used the recursive call q+500 to move the squares up 500. After several different versions of figure1, I decided to create temporary copies of the array in order to get to the final result output (figure 2).

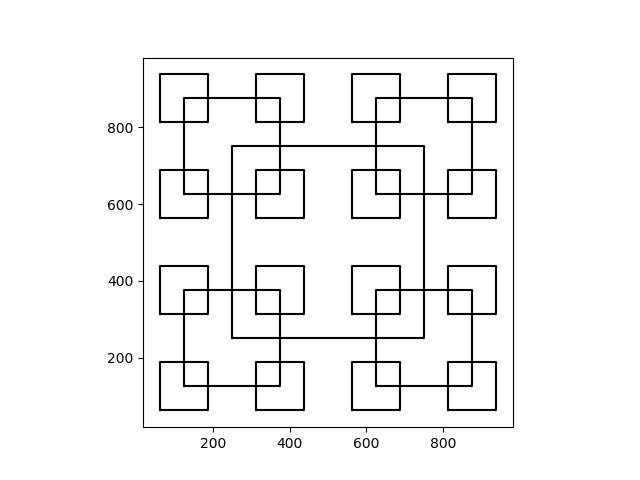


Figure 2 (1.b)

From here I simply modified the parameters to take different n as inputs to produce the other 2 figures defined in lab1.

My approach to draw circles (2a-2c) was to modify the existing code provided in draw\_circles.py and just alter the radius which I found to be unsuccessful as seen in figure 3. In this figure, I divided the radius by 3.

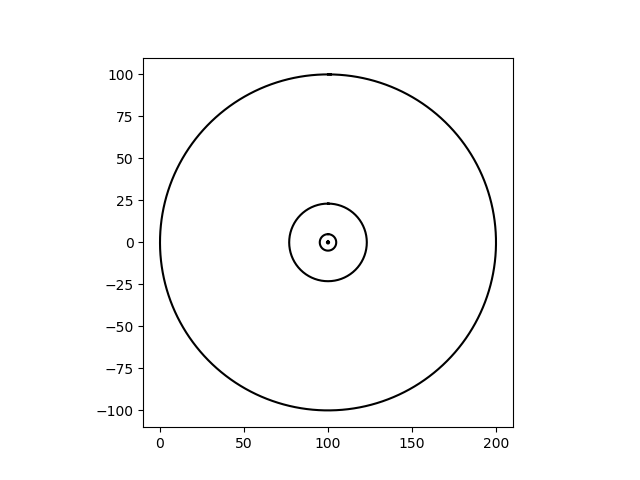
****

Figure 3 (part 2)

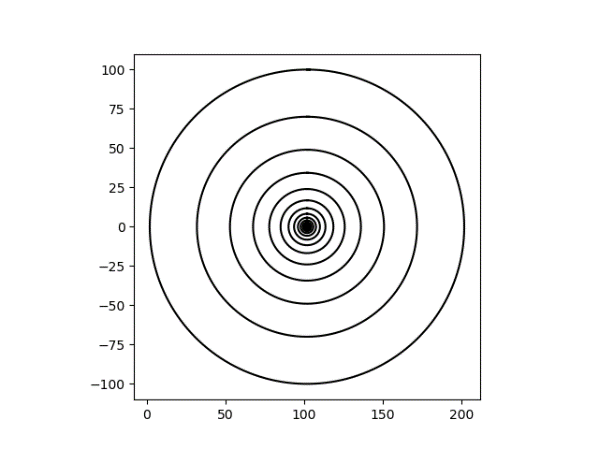
From there I tried to shift the origin the circles which I accomplished by moving the radius + - an integer. Ax.plot(x+2,y,color=’k’) produced the following in figure 4. 

Figure 4

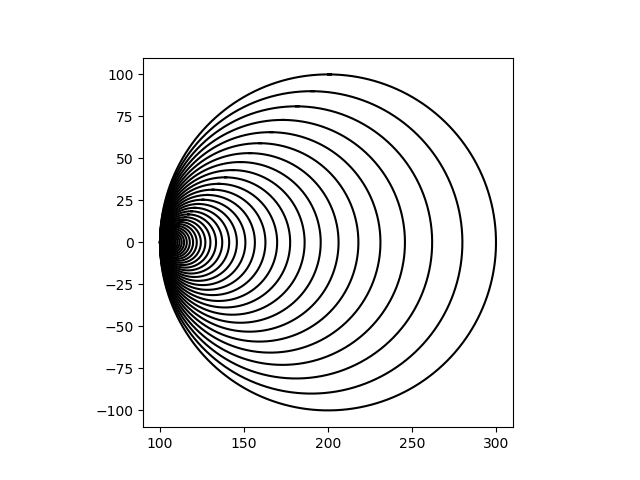
Finally I tried just adding the radius to the x value when calling the graph function. Which produced the following after changing the number of n circles needed and w to .7 and then to .9.

Figure 5 (part 2b)

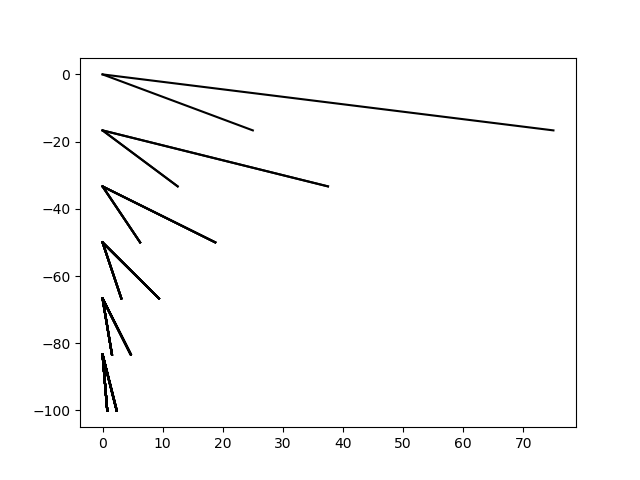
For part 3(trees) I used 3 functions. One function provides the basic line drawing in draw\_line, another changes the height of the tree or levels, and another recursively moves the lines in successive iterations to mimic the nodes and children of the tree. In figure 6, I attempted to create the tree structure with the lines, however, this attempt was unsuccessful because the parts of the graph are not disjoint.

Figure 6

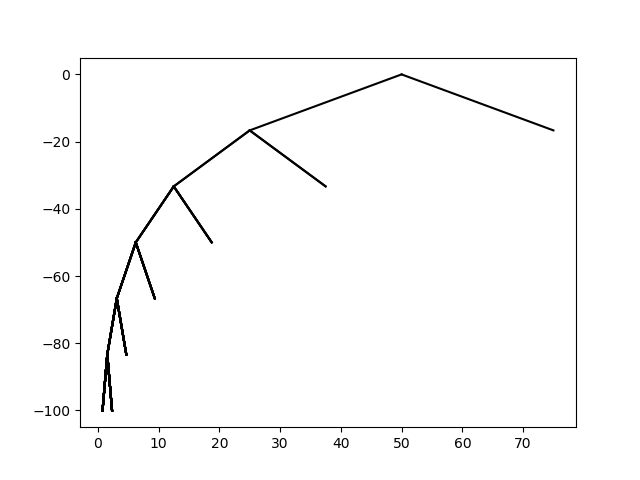
Next, I tried to alter x by incrementing it: ﻿x+w/2 when I called draw\_line. Which produced the following as seen in figure 7. My next approach involved changing the parameters using geometry. I altered the shift between x and y when I called the recursive graph function the second time I called the function recursively only to be x+w/2 as well. This produced the following output as seen in figure 8.

Figure 7

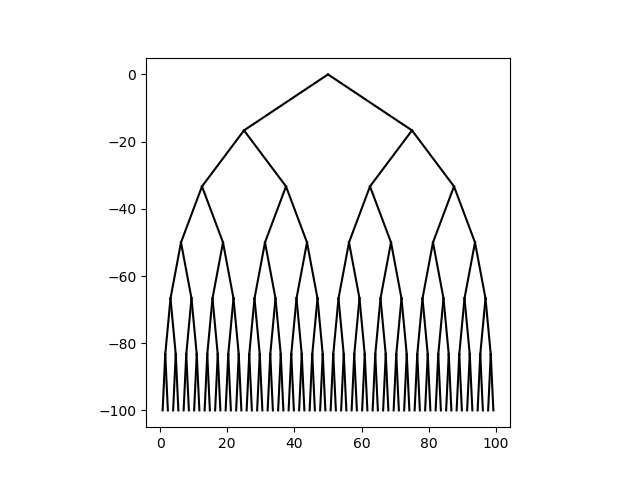


Figure 8 (part 3c)

In the fractal circles images, I started with basic circle method and modified it to have 5 recursive calls after creating a new radius that is 1/3 the original and then storing 2 times that value in a temporary called moved. This was the original radius would not be altered when the new smaller circles were created. Since there are 3 smaller circles within the larger one, I knew the smaller circles need to have 1/3(2r) the larger circle as diameters. Early test runs produced the following output as seen in figure 9. Figure 9 was produced with the following parameters: ﻿ax10, 4, [0,0], 30. Figure 10 as seen on the right was produced by the following parameters ﻿(ax11, 5, [0,0], 30).

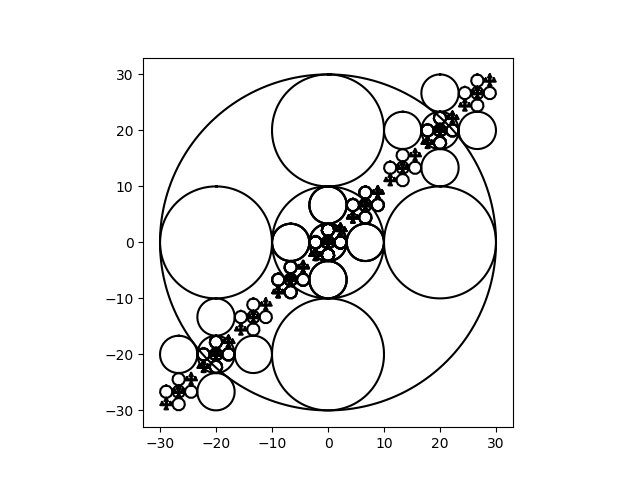
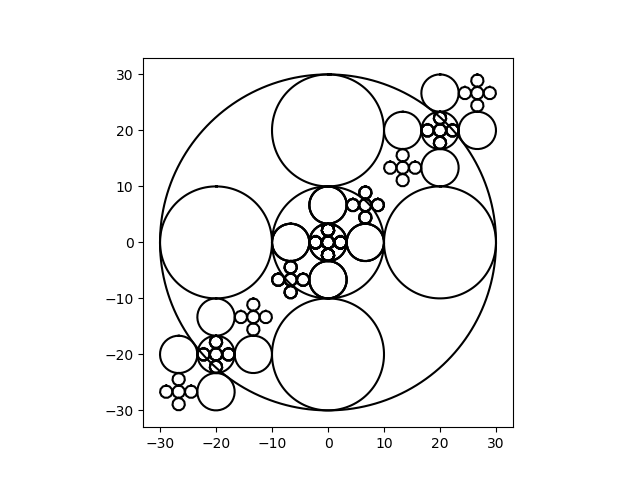


Figure 9

Figure 10

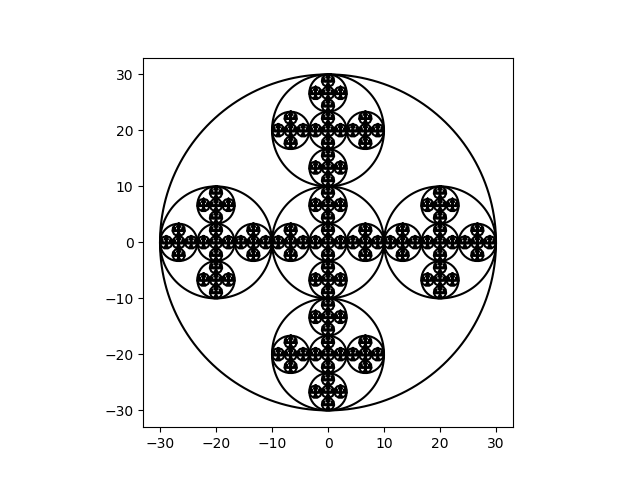
Finally, I arrived at the correct circle pattern with all circles within the larger one by changing each one the appropriate distance I called “moved” as shown in figure 11. 

Figure 11 (part 4c)

|  |  |
| --- | --- |
| runtimes in seconds | figure section |
| .0913 | **2a** |
| .1031 | **2b** |
| .0840 | **1b** |
| .2398 | **3c** |
| .1228 | **4a** |
| .2511 | **4c** |

As you can see from the result in the table, the runtimes were the highest with part 3c and 4c as expected. This is due to the 5 recursive calls made by draw\_circles2 when completing part 2c of the circle fractals. Of the sections tested, 1b took the least amount of time. This may be due in part to the fact that the function required less computationally heavy aspects, however, to accomplish the correct output, there are a total of 4 recursive calls made. Of course, runtimes are also affected by the type of computer and its specifications as well as the amount of RAM that is being consumed while the program is running.

**Conclusion**

This project taught me to plan out my code and logic well before actually coding. This project forced me to draw out the task I was faced with in order to implement it in code. I ended up going through many iterations of code and altering variables to see how they behaved. This also provided me with instant feedback on what I needed to change and how I should proceed in the next trail run. Since I am new to programming in python, it was interesting to try to translate my ideas of how the code should work in java to python. Some images were omitted since changing them is a matter of altering the number of images as parameters to save space. Also, when computing runtimes some images were omitted to show the difference between the least and most complex.

**Appendix (source code)**

﻿#CS2302

#Nicole Favela

#Lab1

#instructor: Olac Fuentes

#TAs: Anindita Nath and Maliheh Zargaran

import matplotlib.pyplot as plt

import numpy as np

import math

#draws basic circles

def circle(center,rad):

n = int(4\*rad\*math.pi)

t = np.linspace(0,6.3,n)

x = center[0]+rad\*np.sin(t)

y = center[1]+rad\*np.cos(t)

return x,y

#draws concentric circles

def draw\_circles(ax,n,center,radius,w):

if n>0:

x,y = circle(center,radius)

ax.plot(x+radius,y,color='k')

draw\_circles(ax,n-1,center,radius\*w,w)

#draws the basic circle

def circle2(center,rad):

n = int(2\*rad\*math.pi)\*2

t = np.linspace(0,6.3,n)

x = center[0]+rad\*np.sin(t) #center + y val

y = center[1]+rad\*np.cos(t) #center + x val

return x,y

#draws fractal circles

def draw\_circles2(ax,num\_circles,center,rad):

if num\_circles>0:

x,y = circle2([center[0],center[1]],rad)

ax.plot(x,y,color='k')

num\_circles-=1

if num\_circles>0:

rad2=rad/3

moved=rad2\*2

draw\_circles2(ax,num\_circles,[center[0],center[1]],rad2)

draw\_circles2(ax,num\_circles,[center[0]-moved,center[1]],rad2)

draw\_circles2(ax,num\_circles,[center[0]+moved,center[1]],rad2)

draw\_circles2(ax,num\_circles,[center[0],center[1]-moved],rad2)

draw\_circles2(ax,num\_circles,[center[0],center[1]+moved],rad2)

#part 1

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

if n>0:

q = p\*w #array elements mulitplied by scalar w

temparray =np.copy(p) #creates copy of array

temp1 =np.copy(p) #creates copy 2

temparray[:,1]+=1000 #stores copy of column 1

temp1[:,0]+=1000 #stores copy of column 0

q1 = temparray\*w # modifies temp and stores it

q2 = temp1\*w

ax.plot(p[:,0],p[:,1],color='k') #plots and colors graph

draw\_squares(ax,n-1,q,w)

draw\_squares(ax,n-1,q+500,w) #recursive call up 500

draw\_squares(ax,n-1,q1,w)

draw\_squares(ax,n-1,q2,w)

#draws basic lines

def draw\_line(ax, x1,y1,x2,y2):

n = int(max(abs(x1-x2), abs(y1-y2)))

x = np.linspace(x1,x2,n)

y = np.linspace(y1,y2,n)

ax.plot(x,y,color='k')

#draws binary tree

def recursive\_graph(ax, x, y, w, h, height ):

if height > 0:

draw\_line(ax, x+w/2,y, x+w/4,y+h)

draw\_line(ax, x+w/2,y, x+w\*3/4,y+h)

height -= 1

recursive\_graph(ax, x, y+h, w/2, h, height)

recursive\_graph(ax, x+w/2, y+h, w/2, h, height)

def graph(ax, x, y, w, h, height):

level\_height = h/height

recursive\_graph(ax, x, y, w, level\_height, height)

plt.close("all")

#part a

fig, ax0 = plt.subplots()

draw\_circles(ax0, 80, [100,0], 100,.7)

ax0.set\_aspect(1.0)

plt.show()

fig.savefig('concentric\_circles\_b.png')

fig, ax1 = plt.subplots()

draw\_circles(ax1, 100, [100,0], 100,.9)

ax1.set\_aspect(1.0)

ax1.axis('on')

plt.show()

fig.savefig('concentric\_circles\_c.png')

#creates array to plot coorinates

p = np.array([[250,250],[250,750],[750,750],[750,250],[250,250]])

fig, ax = plt.subplots()

draw\_squares(ax,3,p,.5)

ax.set\_aspect(1.0)

ax.axis('on')

plt.show() #shows plot

fig.savefig('squares.png') #saves image

#part 3

fig, ax6 = plt.subplots()

graph(plt,0,0,100,-100,6)

ax6.set\_aspect(1.0)

plt.show()

fig.savefig('triangle\_a.png')

fig, ax7 = plt.subplots()

ax7.set\_aspect(1.0)

fig, ax6 = plt.subplots()

graph(plt,0,0,100,-100,6)

plt.show()

fig.savefig('triangle\_b.png')

#part 4

fig, ax9 = plt.subplots()

draw\_circles2(ax9, 3, [0,0], 30)

ax9.set\_aspect(1.0)

plt.show()

fig.savefig('circles0.png')

fig, ax10 = plt.subplots()

draw\_circles2(ax10, 4, [0,0], 30)

ax10.set\_aspect(1.0)

plt.show()

fig.savefig('circles1.png')

fig, ax11 = plt.subplots()

draw\_circles2(ax11, 5, [0,0], 30)

ax11.set\_aspect(1.0)

plt.show()

fig.savefig('circles2.png')