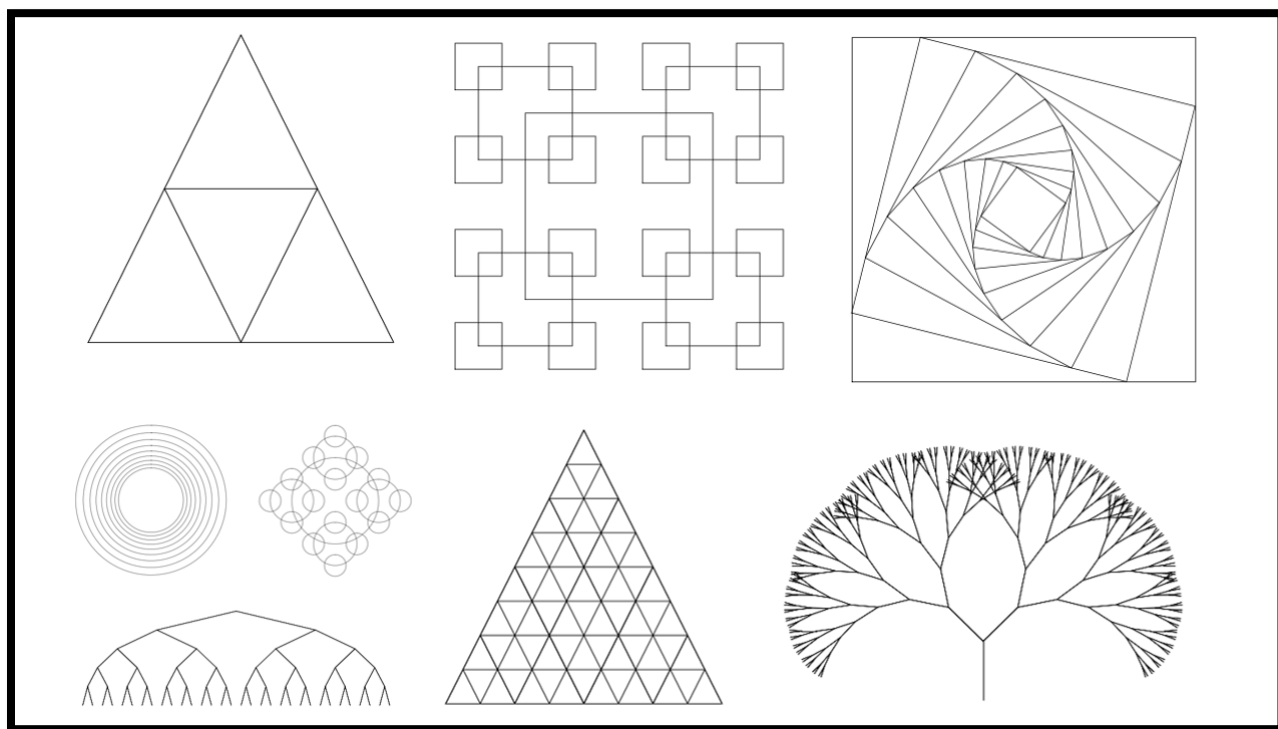


# Lab I

## Recursive Drawing



Course: CS 2302

Section: 12:00 p.m. – 1:20 p.m.

Author: Oswaldo Escobedo

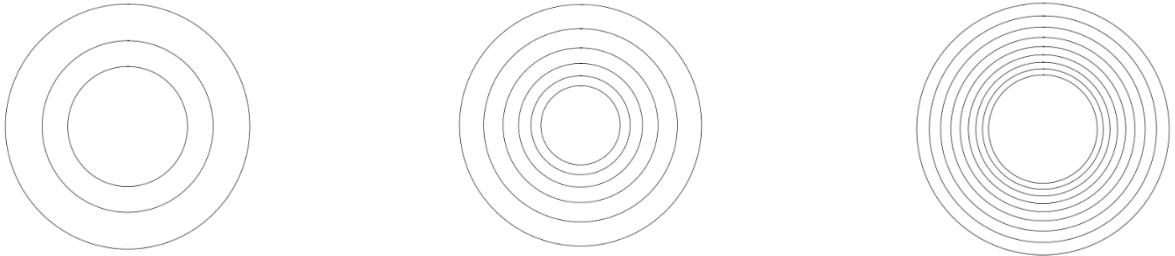
Instructor: Dr. Fuentes

TA: Harshavardhini Bagavathyraj

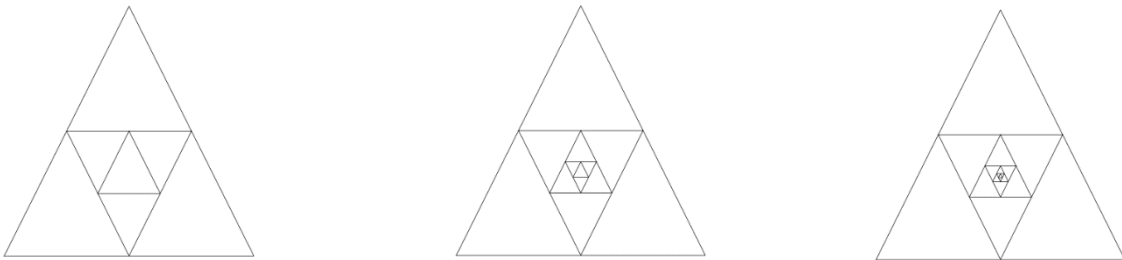
# Introduction

The objective of this lab is to create figures, which are usually called "fractal figures", through the python matplotlib and using recursion. The figures that we are going to try to replicate and make are found on blackboard. There are eighteen in total:

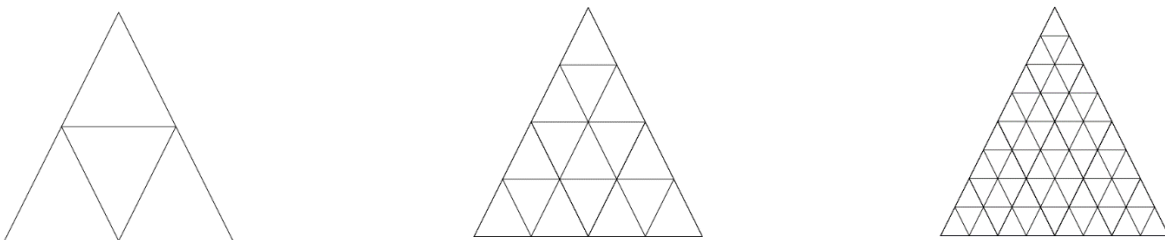
3 Circles (with different radius):



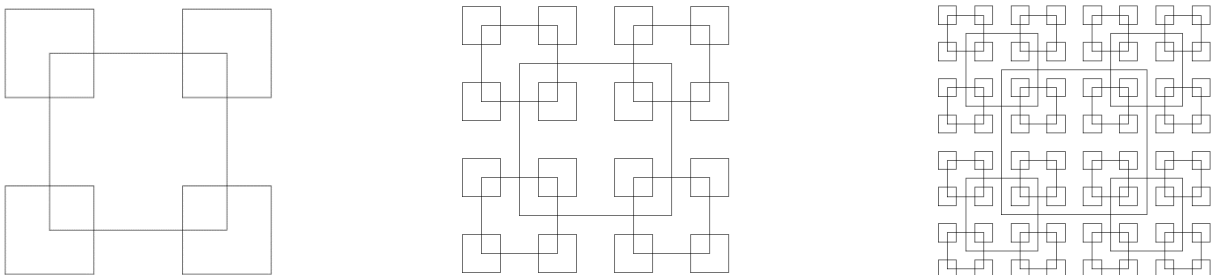
3 Triangles (within a triangle within another and so on):



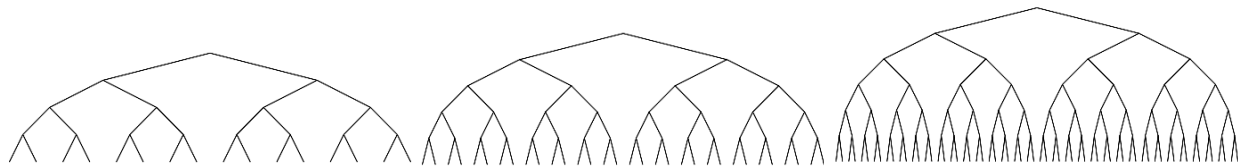
3 triangles (which are made of smaller triangles):



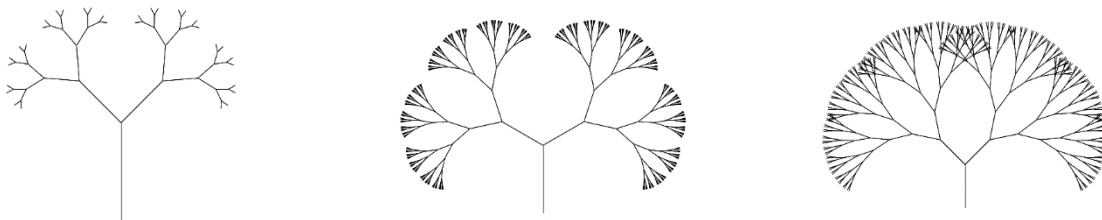
3 squares (in which they have another 4 squares twice as small in its corners):



3 figures similar to tree roots:



3 “Fractal” trees:



The operations that we will use to solve and complete the laboratory are those of manipulating the X and Y coordinates in a Cartesian plane, using the sine and cosine functions, manipulating the radius, length of a figure, degrees, and other mathematical operations.

To obtain these figures, we should have knowledge of the numpy (numerical python) and matplotlib library, which is used to generate graphs using the coordinates of X and Y. Moreover, we need to make use of its methods, such as `ax.plot()`, `plt.subplot()`, `ax.axis()` and above all, we must know how recursive calls work.

```
10 import numpy as np
11 import matplotlib.pyplot as plt
12
115 fig, ax = plt.subplots()
116 draw_squares(ax, 6, p, .1)
117 ax.set_aspect(1.0)
118 ax.axis('off') # Uncomment to see coordinates in drawing
119 plt.show()
120 fig.savefig('squaresa.png')
```

## Proposed Solution Design and Implementation

The first thing I did was to understand a part of the code that Dr. Fuentes gave us. In his code, the teacher left us three methods, one that creates a circle and creates other circles on its right, left, up and down sides, the second method creates a square within another square that becomes smaller and turns and the third one is to create and locate the X and Y coordinates of a circle

```
29 def draw_four_circles(ax,n,center,radius):
30     if n>0:
31         x,y = circle(center,radius)
32         ax.plot(x,y,linewidth=0.5,color='k')
33         draw_four_circles(ax,n-1,[center[0],center[1]+radius],radius/2)
34         draw_four_circles(ax,n-1,[center[0],center[1]-radius],radius/2)
35         draw_four_circles(ax,n-1,[center[0]+radius,center[1]],radius/2)
36         draw_four_circles(ax,n-1,[center[0]-radius,center[1]],radius/2)

22 def draw_squares(ax,n,p,w):
23     if n>0:
24         ax.plot(p[:,0],p[:,1],linewidth=0.5,color='k') # Draw rectangle
25         i1 = [1,2,3,0,1]
26         q = p*(1-w) + p[i1]*w
27         draw_squares(ax,n-1,q,w)

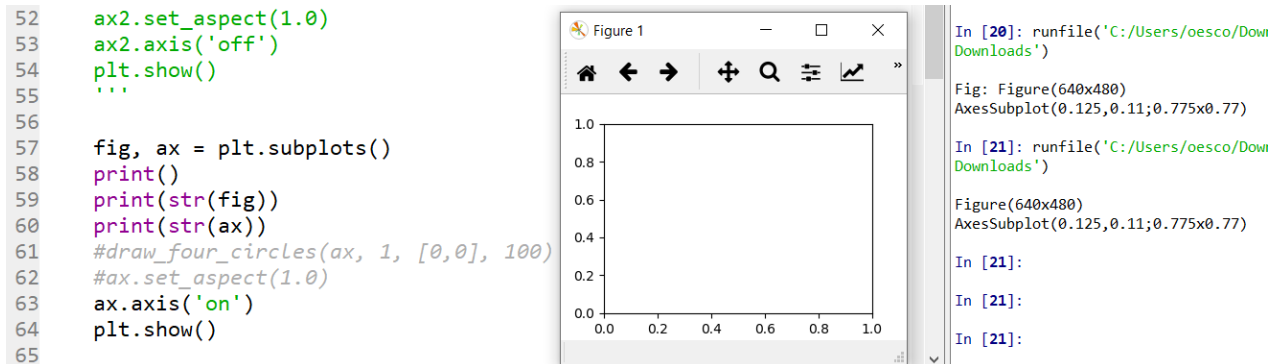
14 def circle(center,rad):
15     # Returns the coordinates of the points in a circle given center and radius
16     n = int(4*rad*math.pi)
17     t = np.linspace(0,6.3,n)
18     x = center[0]+rad*np.sin(t)
19     y = center[1]+rad*np.cos(t)
20     return x,y
```

Therefore, we must analyze and understand why Dr. Fuentes put X variable, why he used Y formula and method and why he put Z quantity in a variable. Otherwise we will not be able to understand the functionality of the library and the methods nor will we be able to solve and complete the laboratory.

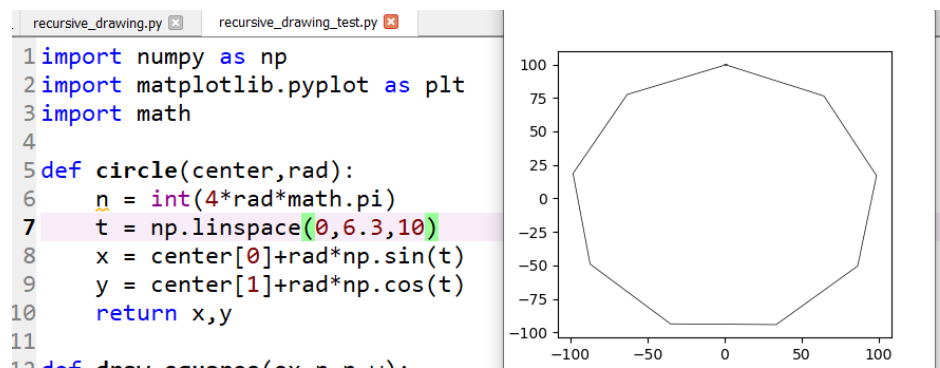


After observing and analyzing the code provided by Fuentes, I decided to create another file (with the same methods) where I would put random inputs and do experiments, this to know how these inputs would affect the figures created by the Fuentes and thus to know how each line of his code works.

One of my first experiments was to comment on the method call and to print the variables fig and ax to understand how it works. As I did this, I understood that these methods created a kind of Cartesian plane and graph, whose default measurement values were 640x480



On this occasion I tried to find out what the function of n and np.linspace() was. After many random inputs I began to notice a pattern and I understood that the purpose of n was that it calculated the amount of points needed to make the circle look like a circle and that np.linspace() returned an array with a sequence of numbers based on n.



After understanding the function of the circle() and the draw\_draw\_circle () method. I started to analyze the draw\_squares() method. Because I did not understand how the internal squares were spinning around like a spiral. And since I already knew the functionality of the other lines of code, I sensed that the answer was in the variables i1 and q.

```

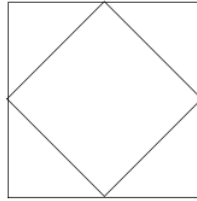
15 i1 = [1,2,3,0,1]
16 q = p*(1-w) + p[i1]*w

```

After many analyzes and experiments of random entries, I still did not understand the purpose of these variables. So I decided to ask Dr. Fuentes and he explained to me that i1 worked to reorder the elements of the array, although in this case it was to reorder the coordinates, and that they

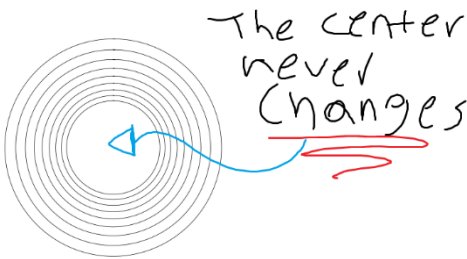
would serve to change the rotation of the figure. For this he used the formula  $q = p * (1-w) + p[i1] * w$ , in which, as he said, he gave more weight to a coordinate and make the figure smaller. For example, if I put a weight of 0.5, on the first recursive call, each point at  $q$  will be at the midpoint of the sides of the original square.

```
35 print(p)
36
37 fig, ax = plt.subplots()
38 draw_squares(ax,2,p,.5)
39 ax.set_aspect(1.0)
40 ax.axis('off') # Uncomment
41 plt.show()
```



Having already understood the code provided by Dr. Fuentes, I decided to start doing the methods to create the figures that we were asked to do.

```
12 def draw_inside_circles(ax,n,center,radius,w):
```



I started analyzing the figure and I saw that the **center** was never going to change. Therefore, I decided that in the recursive call I would leave the center's value intact.

And because the algorithm of the figure is similar to the `draw_four_circles()` method that Dr. Fuentes gave us, I decided to leave the parameters the same as his.

Next, I noticed that the radius of the circle was constantly decreasing, so I assumed that I had to add a parameter which made the value of the radius decrease in each recursive call. Therefore, I added the variable weight ( $w$ ), which would multiply the radius constantly so that the circle becomes the  $w$  of smaller times.

```
16 new_radius = radius * w
17 draw_inside_circles(ax,n-1,[center[0],center[1]],new_radius,w)
```

```
12 def draw_inside_triangle(ax,n,p):
```

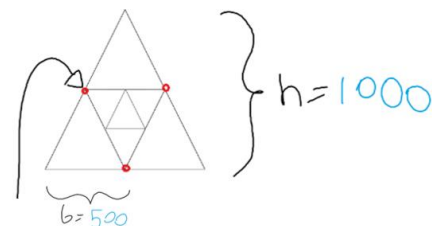
To create the triangle I had to create 4 coordinates for X and Y. In which I decided that the base would be a size of 500, while the height would be of 1000.

I noticed in the figure that the vertices of the inner triangle were located in the middle of the sides of the original triangle. So I decided to have the coordinates divided in half and make use of the variable  $i1$ , which will

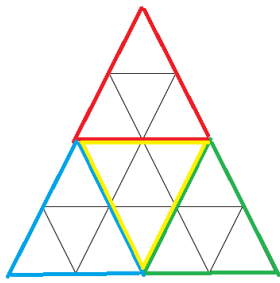
```
i1 = [1,2,3,1]
q = p/2 + p[i1]/2
draw_inside_triangle(ax,n-1,q)
```

help us to add the starting

point to the following point. Assuming this works, it will make the triangle smaller and turn around.



```
5 def draw_triangles(ax,n,p):
```



The algorithm to perform this figure is very similar to the previous triangle. However, the only thing I knew was that to get the yellow triangle I had to copy the previous algorithm's triangle, but for the green, blue and red triangle, I didn't know how to do it.

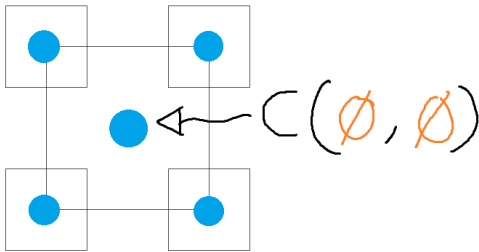
So, to make the task easier, I separated the triangles by colors. Having this done, I saw an interesting pattern, and it was that the algorithm for the blue, green and red triangle was almost the same as the yellow one, only that its internal triangles were turned on the opposite side of the yellow inner triangle.

After I saw that yellow triangle needed a recursive call, I assumed that the other three

were going to need their own recursive call, only that we would have to reorganize their coordinates, so that they are on their corresponding sides.

```
10 draw_triangles(ax,n-1,q) |
11 draw_triangles(ax,n-1,np.array([p[0],q[0],q[2],p[0]]))
12 draw_triangles(ax,n-1,np.array([q[0],p[1],q[1],q[0]]))
13 draw_triangles(ax,n-1,np.array([q[2],q[1],p[2],q[2]]))
14
```

```
5 def draw_four_squares(ax,n,C,L):
```



The algorithm for drawing this figure was one of the easiest to understand, since if you analyze the figure well, you will have noticed that it is very similar to the algorithm that Dr. Fuentes gave us only that instead of circles, it is a square.

However, my first approach was very simple and can be considered that I hard-coded. I mention this because instead of using a specific formula, I made use of numbers and operations in the recursive calls to change the position of the squares: -750, -250, +250, +750. Nevertheless, after thinking for a few days I came to the conclusion that there was a better way to solve the problem and make use of a formula.

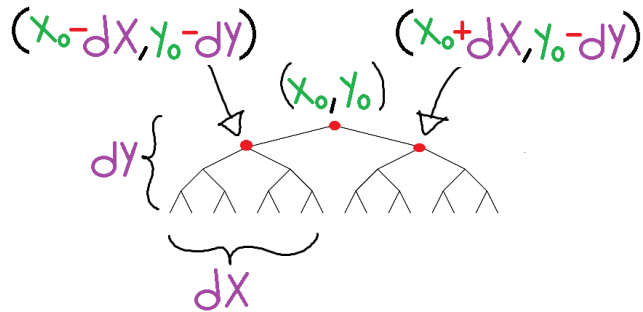
Therefore, I decided to put a center to the square and give it an initial length. Next I would call the method with an initial center of [0,0] and 1000 of length. So I created 4 variables which would represent the vertices of the square. I would subtract or add to these points the length of the square, depending on which vertex is, in order to form a square.

```
p0 = [C[0]-L,C[1]-L]
p1 = [C[0]-L,C[1]+L]
p2 = [C[0]+L,C[1]+L]
p3 = [C[0]+L,C[1]-L]
p = np.array([p0,p1,p2,p3,p0])
```

```
draw_four_squares(ax,n-1,p0,L/2)
draw_four_squares(ax,n-1,p1,L/2)
draw_four_squares(ax,n-1,p2,L/2)
draw_four_squares(ax,n-1,p3,L/2)
```

Once the initial square was plotted, I gave the four recursive calls their respective centers, which would be the ones that were used to create the original square, in addition, I divided the length of the square by two so that the new squares were twice more small than the original square.

```
85 def draw_root(ax,n,x,y,dx,dy):|
```



Despite how complicated the figure looks, the algorithm for making it was one of the easiest and simplest. It should only have the following factors: an initial center of  $[0,0]$ , a constant for its height ( $dy$ ) and a value that will decrease its width over time ( $dx$ ).

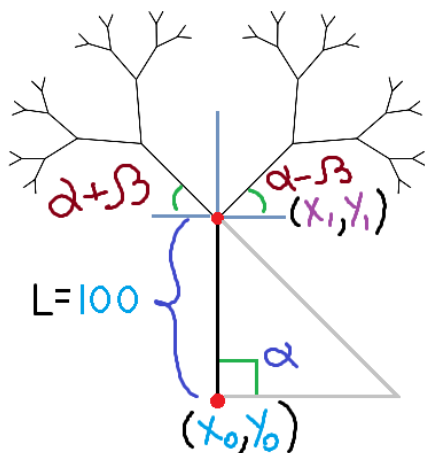
```
delta_x = 800
delta_y = 200
center = np.array([[0,0],[0,0],[0,0]])
```

Having this ready we will call the method, we

will plot for the first time, but since the coordinates are  $[0,0]$  it will not draw anything. So then a recursive call will be made, one in which  $X$  will be subtracted with  $DX$  and another in which it will be added, while  $Y$  will be subtracted in both,  $DX$  will be divided between two and  $DY$  will remain constant. This will make the first recursive call draw on the left, while the second will

```
draw_root(ax,n-1,[x[0]-dx,x[0],x[2]-dx],[y[0]-dy,y[0],y[2]-dy],dx/2,dy) make the one on the
draw_root(ax,n-1,[x[0]+dx,x[0],x[2]+dx],[y[0]-dy,y[0],y[2]-dy],dx/2,dy) right.
```

```
5 def draw_tree(ax,n,p,L,m1,alpha,beta,m2):
```



This figure was definitely the most complicated to do. My first approach was to do the same procedure that I did with the previous figure, but there was a problem that I did not take into account and it was that this figure began with a stick and that according to the inputs the sticks were going to gradually tilt.

So I went to ask Dr. Fuentes to explain better the algorithm of the figure, he answered that we had to use mathematical formulas, specifically trigonometry, to obtain  $x_1$  and  $y_1$  to then draw the first stick.

For example, he told me and explained that to obtain the value of the width ( $x_1$ ) and height ( $y_1$ ) we had to make use of the trigonometric ratios of sine and cosine:

$$\begin{aligned} \frac{x_1}{L} &= \sin \alpha \\ \frac{y_1}{L} &= \cos \alpha \end{aligned} \quad \Rightarrow \quad \begin{aligned} x_1 &= x_0 + L * \cos \alpha \\ y_1 &= y_0 + L * \sin \alpha \end{aligned}$$

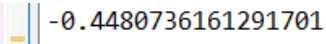
But before using this formula, we must solve for  $x_1$  and  $y_1$ . Moreover, since we want to move continuously in the Cartesian plane, we must add the value of  $x_0$  and  $y_0$  to the equation so that these are added so that we can move constantly.



When I implemented the formula in the code, I realized that the output was not what I expected. I experienced many difficulties, due to the following problems:

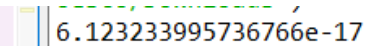
- When I printed the value of  $\cos(90)$  I got a negative value and far from the real result, when the correct result was 0.

```
print(math.cos(alpha))
```




- After researching through the internet I realized that when you want to put any trigonometric function, we must first convert the degrees to radians. So I decided to use the `math.radians()` function. However, after reprinting `math.cos(math.radians(90))`, the output returned a wrong result again.

```
print(math.cos(math.radians(alpha)))
```



- I went back to research on the internet and found many solutions to the problem, but it was very difficult, since I tried to apply them and none worked, until finally I decided to go to the one that had more precision. I had to use the `round()` method, which has the function of rounding the decimals of a number. This solution finally worked.

```
print(round(math.cos(math.radians(alpha)),2))
```



After solving the problem, I decided to rewrite the code in a cleaner way:

```
x0 = p[0]
y0 = p[1]
x1 = x0 + L*round(math.cos(math.radians(alpha)),2)
y1 = y0 + L*round(math.sin(math.radians(alpha)),2)
p = np.array([[x0,y0],[x1,y1],[x0,y0]])
```

But the most important part of the code was still missing: the recursive call. In it we were going to change the starting point from `[x0, y0]` to `[x1, y1]`, we would reduce the size of the branch, we would add or remove degrees to Alpha so that the branches are tilting more in each recursion, and finally we would multiply beta by a constant so that it becomes smaller over time

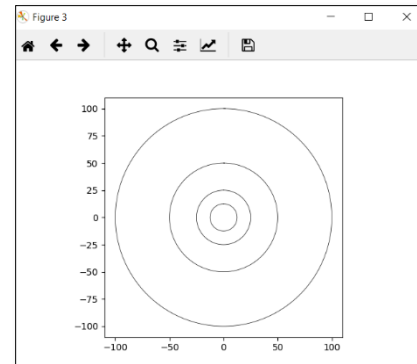
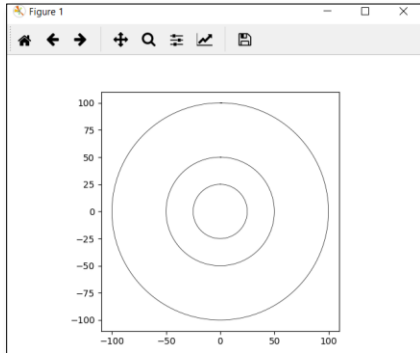
```
draw_tree(ax,n-1,[x1,y1],L*m1,m1,alpha+beta,beta*m2,m2)
draw_tree(ax,n-1,[x1,y1],L*m1,m1,alpha-beta,beta*m2,m2)
```

# Experimental Results

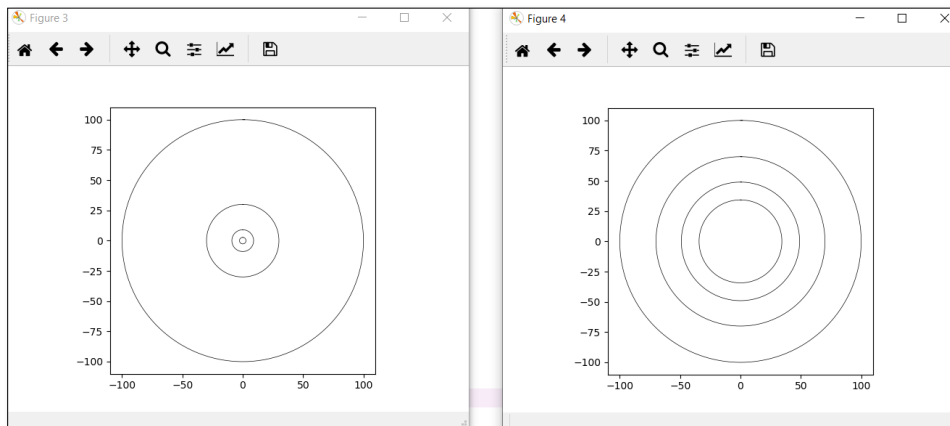
Now that we have finished writing our code we must test its outputs and make it look similar to the requested figures.

**We will start with the `draw_inside_circles()` method:**

I started by putting a 0.5 radius weight input and I wanted 2 and 4 circles. Seeing the result I was stunned, because the circle that contained 2 circles looked a lot like the teacher's, while the one with 4 circles was very different.



After seeing this I decided to make two circles of 4 circles with a radius weight of 0.3 and another of 0.7.

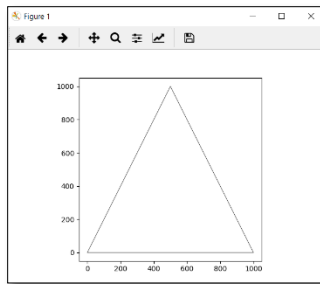


I observed that from this experiment I obtained a very important fact: the smaller the value of the weight of the radius, the smaller the next circles would be.

And after putting many combinations of inputs I reached the following conclusions:

- The first circle of the lab required that  $n = 3$  and that the weight of the radius be  $w = 0.7$
- The second required that  $n = 6$  and that the weight of the radius be  $w = 0.8$
- The third required that  $n = 9$  and that the weight of the radius be  $w = 0.9$

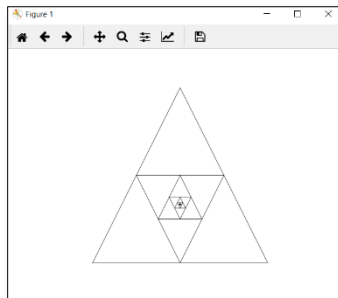
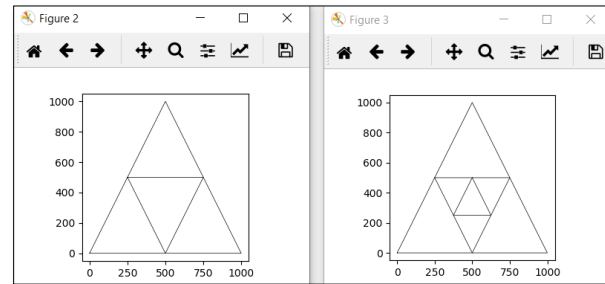
### **def draw\_inside\_triangle() method:**



In this figure I started with  $n = 1$ , to check if the triangle was formed correctly.

And seeing that my method was working I decided to put an input of  $n = 2$  and  $n = 3$  to see

the behavior of the method.

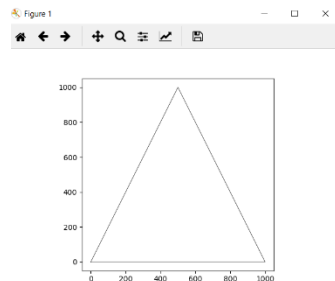


When I saw the

output of the method, I checked that one of the figures coincided with what the laboratory was asking for, while the other did not, this might be due to the amount I gave  $n$ . After putting more values on  $n$ , I realized something the more you increase the values of  $n$ , the triangles would be twice smaller than their predecessor, the left example has an input of  $n = 9$ , taking this into account I obtained the figures I wanted with the following inputs:

- For the first triangle I needed that  $n = 3$
- For the second one I needed that  $n = 5$
- For the third part that  $n = 7$

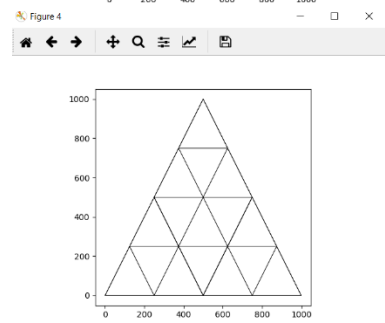
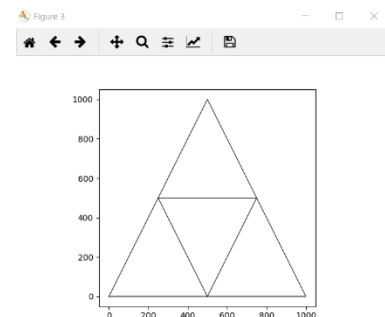
### **def draw\_triangles() method:**



The inputs to call this method were the same as the one in the previous triangle, so the only option the user will have will be to change  $n$  to his liking. Therefore, I started by checking if the triangle was formed correctly by entering an input of  $n = 1$ .

After verifying if the triangle was drawn, I continued to enter inputs of  $n = 2$  and  $n = 3$ . (Outputs are on the right side)

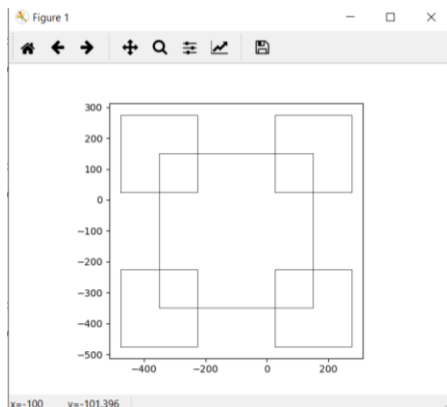
For a moment I thought that with the triangle of  $n = 2$  the method would not work, but after seeing the output of the triangle  $n = 3$  I saw that it was not so. The explanation is that, the triangle  $n = 2$  was overlapping in the original triangle and therefore it could not be noticed if the algorithm had worked or not.



When I checked that it worked I decided to put many input combinations to obtain the figures requested in the laboratory and I came to the conclusion that to create the figures:

- For the first triangle  $n$  should be  $n = 2$
- For the second  $n = 3$
- For the third  $n = 4$

### def draw\_four\_squares() method:



In this method the user can choose between how much will be  $n$ , where the center will be and how much the sides of the square will measure. The following example shows an input of  $n = 2$ , the center will be at  $[-100, -100]$  and the square will measure 250 per side.

```
246 fig, ax = plt.subplots()
247 draw_four_squares(ax, 2, [-100, -100], 250.0)
248 ax.set_aspect(1.0)
249 ax.axis('on')
250 plt.show()
```

So, to be more symmetrical, I made the center of the square go at the coordinates  $[0,0]$  and the square measured 1000. Finally, to verify that it worked, I made  $n$  equal to be 1.

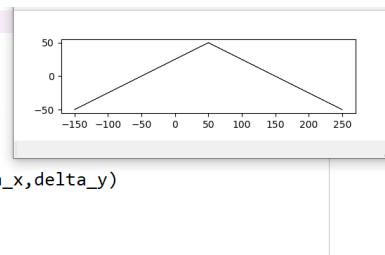
When checking that it worked, I decided to start putting different inputs to obtain the figures requested in the laboratory and came to the conclusion that to create the figures:

- For the first square we needed  $n$  to be equal to 2
- For the second one that  $n = 3$
- For the third,  $n = 4$

### def draw\_root() method:

I started by putting an input of  $[50,50]$  as the point of origin and I put  $n$  to be equal to 1, in addition, I decided that the horizontal side ( $dx$ ) would

```
258 ''' Figure No. 5 '''
259
260 delta_x = 200
261 delta_y = 100
262 center = np.array([[50,50],[50,50],[50,50]])
263
264 fig, ax = plt.subplots()
265 draw_root(ax, 2, center[:,0], center[:,1], delta_x, delta_y)
266 ax.set_aspect(1.0)
267 ax.axis('on')
268 plt.show()
```



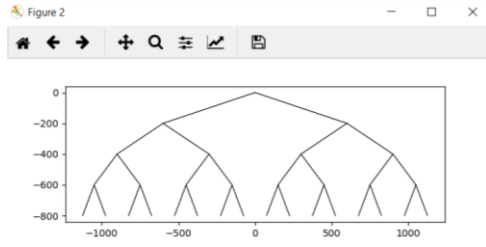
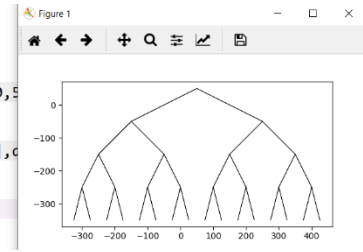
advance 200 and the vertical side ( $dy$ ) would advance 100. This to check if the parameters worked properly. The output surprised me, since I did not know if my approach would work.

Next, I decided to increase the value of  $n$  to 5, this to know what the output would be. Seeing that the output turned out to be similar to the figure that the laboratory asked us to do, I decided to change the values of  $dx$  and  $dy$ , only this time  $dx$  would be 600 and  $dy$  would be 200.

```

258 ''' Figure No. 5 '''
259
260 delta_x = 200
261 delta_y = 100
262 center = np.array([[50,50],[50,50],[50,50]])
263
264 fig, ax = plt.subplots()
265 draw_root(ax,5,center[:,0],center[:,1],center[:,2])
266 ax.set_aspect(1.0)
267 ax.axis('on')
268 plt.show()
269 fig.savefig('roota.png')
270

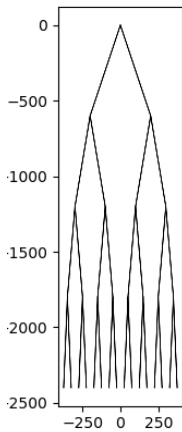
```



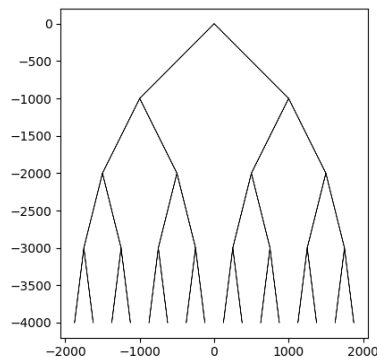
This was the result. If we analyze the figure in detail, you will have noticed that it is almost similar to the figure requested by the laboratory.

After placing several random entries I came to the conclusion that:

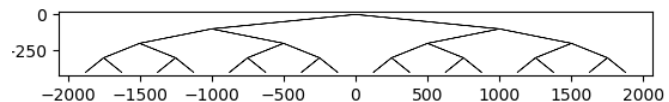
- If  $dy$  had a higher value than  $dx$  the figure would narrow but at the same time its height will be greater. ( $dx = 200$ ,  $dy = 600$ ). [Image 1](#).
- On the other hand, if  $dx$  was greater than  $dy$ , the figure would be wider, but at the same time its height will be smaller. ( $dx = 1000$ ,  $dy = 100$ ). [Image 2](#).
- Also, if you put the same value for  $dx$  and  $dy$ , the figure would look similar to the one requested in the laboratory. ( $dx = 1000$ ,  $dy = 1000$ ). [Image 3](#).



[Image 1](#)



[Image 3](#)



[Image 2](#)

And after trying many inputs to obtain a figure similar to those required by the laboratory, I got them and they were the following:

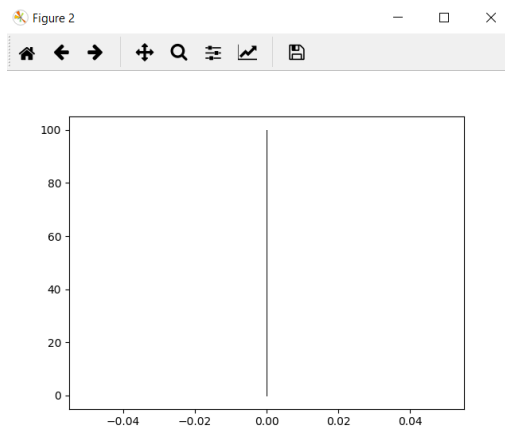
- For all three roots, the value of  $dx$  and  $dy$  would be the same,  $dx$  would be 800 and  $dy$  would be 200.
- To get the first root,  $n$  would be 5.
- For the second root,  $n$  would be 6.
- And for the third,  $n$  would be 7.

### def draw\_tree() method:

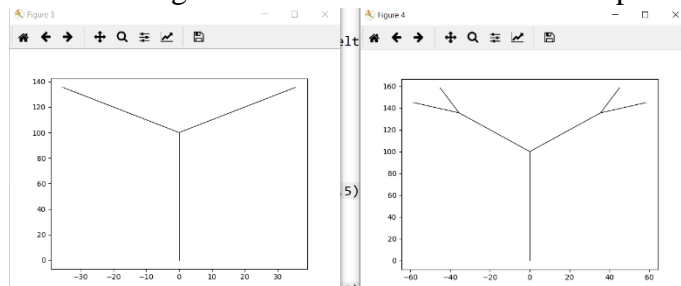
I must admit that to test this method, it was the most complex and complicated of all. But I started by putting the following default values in the parameters of the method:

- $n = 1$
- $X_0 = 0$
- $Y_0 = 0$
- $large = 100$
- $\alpha = 90.0$
- $\beta = 45.0$
- weight of length = 0.5
- weight of beta = 0.5

Due to these parameters I was able to obtain the following result:



The trunk of the tree came out as it appears in the image of the laboratory, now it was only necessary to try more entries, in which it would add  $n$  to values of 2 and 3, to check if the algorithm worked. This were the outputs:



The method worked well, but the output was not what I expected. Therefore, I decided to put random inputs in the large and beta weights.

- In the first test I put the weight of beta and length to be 0.7 and for  $n$  to be 3. [Image 1](#)
- In the second test, I put the weight of beta to be 0.6, for the weight of the length I put 0.8 and for  $n$  I put 4. [Image 2](#)

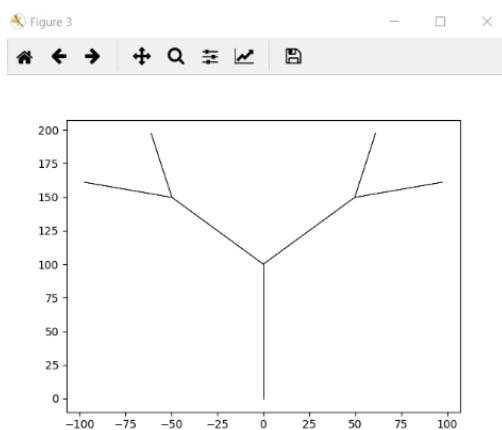


Image 1

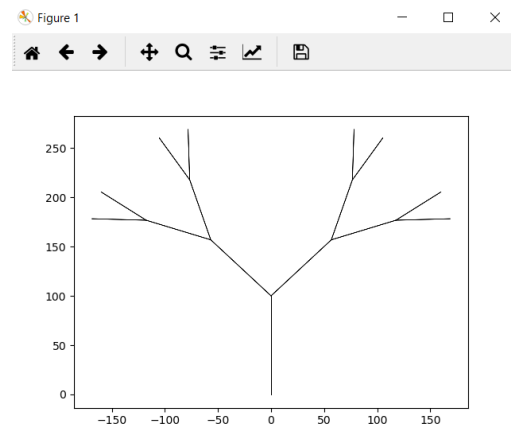


Image 2

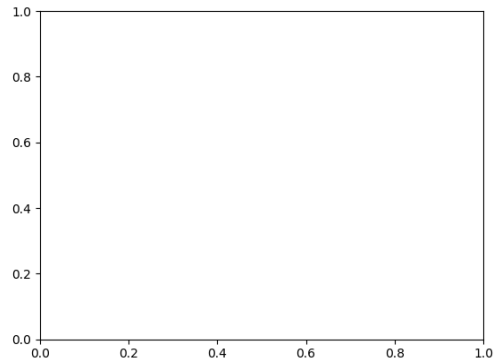
After seeing the two tests, I understood that the larger the weight value was, the branches would become smaller sooner. In addition, I also noticed that the larger the beta weight value was, the branches would have a greater opening.

And after several entries, I obtained the values to create the 3 figures that were requested in the laboratory and were the following:

- For the first tree, the inputs will be the following:  $n = 6$ , origin = [0,0], length = 100, weight of length = 0.6,  $\alpha = 90.0$ ,  $\beta = 45.0$ , weight of  $\beta = 0.9$
- For the second tree, the inputs will be the following:  $n = 10$ , origin = [0,0], length = 100, weight of length = 0.7,  $\alpha = 90.0$ ,  $\beta = 60.0$ , weight of  $\beta = 0.7$
- Finally, for the third tree, the inputs will be the following:  $n = 9$ , origin = [0,0], length = 100, weight of length = 0.82,  $\alpha = 90.0$ ,  $\beta = 45.0$ , weight of  $\beta = 0.75$

## Conclusions

In conclusion, this project helped me a lot to get acquainted with Python and reinforce my recursion skills. Although, not only did I learn that, but this project also helped me to know the matplotlib library, which is a python library that contains methods that help us create graphic content using arrays. Some creations that we can make by using this library are figures, histogram, bar chart, line chart, pie chart and area charts, as well we can generate 3d graphics and image plot. This library has many applications, which I would like to apply in the future for a personal project.



I learned something very important and it was something I always complained about and that is that I didn't understand why we, the computer scientists, needed to take Math and Calculus courses as a requisite to take CS courses. I didn't think they had an important application in our career. However, because of this project I began to realize that even to create such a simple figure you need to have knowledge of mathematics and I say it because in this project we had to use trigonometry.

One of the things that I loved to learn in this laboratory is that we were able to use the coordinates in a Cartesian plane and that we had to draw lines, and that, in addition, we had to manipulate this to create a figure.



## Appendix

```
# Course: CS 2302
# Assignment: Lab I
# Author: Oswaldo Escobedo
# Instructor: Dr. Fuentes
# TA: Harshavardhini Bagavathyraj
# Date of Last Modification: 01/31/2020
# Purpose of the Program: to practice and
# use recursion to draw fractal figures
```

```
import numpy as np
import matplotlib.pyplot as plt
import math
```

```
def circle(center,rad):
    # Returns the coordinates of the points in a circle given center and radius
    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
```

```
def draw_squares(ax,n,p,w):
    if n>0:
        ax.plot(p[:,0],p[:,1],linewidth=0.5,color='k') # Draw rectangle
```

```

i1 = [1,2,3,0,1]
q = p*(1-w) + p[i1]*w
draw_squares(ax,n-1,q,w)

```

```

def draw_four_circles(ax,n,center,radius):

```

```

    if n>0:
        x,y = circle(center,radius)
        ax.plot(x,y,linewidth=0.5,color='k')
        draw_four_circles(ax,n-1,[center[0],center[1]+radius],radius/2)
        draw_four_circles(ax,n-1,[center[0],center[1]-radius],radius/2)
        draw_four_circles(ax,n-1,[center[0]+radius,center[1]],radius/2)
        draw_four_circles(ax,n-1,[center[0]-radius,center[1]],radius/2)

```

''' Figure No. 1 '''

```

def draw_inside_circles(ax,n,center,radius,w):

```

```

    if n > 0:
        x,y = circle(center,radius)
        ax.plot(x,y,linewidth=0.5,color='k')
        new_radius = radius * w # We decrement the length of the radius
        draw_inside_circles(ax,n-1,[center[0],center[1]],new_radius,w)

```

''' Figure No. 2 '''

```

def draw_inside_triangle(ax,n,p):

```

```

    if n > 0:
        ax.plot(p[:,0],p[:,1],linewidth=0.5,color='k')
        i1 = [1,2,3,1] # We swap the elements to the left, to flip the figure

```

```

q = p/2 + p[i1]/2 # We divide the coordinates by two so that the figure
draw_inside_triangle(ax,n-1,q) # is twice as small as the previous one

```

''' Figure No. 3 '''

```

def draw_triangles(ax,n,p):
    if n>0:
        ax.plot(p[:,0],p[:,1],linewidth=0.5,color='k')
        i1 = [1,2,3,1]
        q = p/2 + p[i1]/2
        draw_triangles(ax,n-1,q) # Center Triangle
        draw_triangles(ax,n-1,np.array([p[0],q[0],q[2],p[0]])) # Left Triangle
        draw_triangles(ax,n-1,np.array([q[0],p[1],q[1],q[0]])) # Top Triangle
        draw_triangles(ax,n-1,np.array([q[2],q[1],p[2],q[2]])) # Right Triangle

```

''' Figure No. 4 '''

```

def draw_four_squares(ax,n,C,L):
    if n>0:
        p0 = [C[0]-L,C[1]-L] # Bottom left corner point
        p1 = [C[0]-L,C[1]+L] # Upper left corner point
        p2 = [C[0]+L,C[1]+L] # Upper right corner point
        p3 = [C[0]+L,C[1]-L] # Bottom right corner point
        p = np.array([p0,p1,p2,p3,p0]) # Coordinates to create the square
        ax.plot(p[:,0],p[:,1],linewidth=0.5,color='k')
        draw_four_squares(ax,n-1,p0,L/2) # Bottom left corner square
        draw_four_squares(ax,n-1,p1,L/2) # Upper left corner square
        draw_four_squares(ax,n-1,p2,L/2) # Upper right corner square

```

```
draw_four_squares(ax,n-1,p3,L/2) # Upper right corner square
```

```
''' Figure No. 5 '''
```

```
def draw_root(ax,n,x,y,dx,dy):
```

```
    if n>0:
```

```
        ax.plot(x,y,linewidth=0.5,color='k')
```

```
        draw_root(ax,n-1,[x[0]-dx,x[0],x[2]-dx],[y[0]-dy,y[0],y[2]-dy],dx/2,dy) # Left line
```

```
        draw_root(ax,n-1,[x[0]+dx,x[0],x[2]+dx],[y[0]-dy,y[0],y[2]-dy],dx/2,dy) # Right line
```

```
''' Figure No. 6 '''
```

```
def draw_tree(ax,n,p,L,m1,alpha,beta,m2):
```

```
    if n > 0:
```

```
        x0 = p[0]
```

```
        y0 = p[1]
```

```
        x1 = x0 + L*round(math.cos(math.radians(alpha)),2) # Formula to find x1 (ending point).
```

```
        y1 = y0 + L*round(math.sin(math.radians(alpha)),2) # Formula to find y1 (ending point).
```

```
        p = np.array([[x0,y0],[x1,y1],[x0,y0]]) # We create an array with the coordinates.
```

```
        ax.plot(p[:,0],p[:,1],linewidth=0.5,color='k')
```

```
        draw_tree(ax,n-1,[x1,y1],L*m1,m1,alpha+beta,beta*m2,m2)
```

```
        draw_tree(ax,n-1,[x1,y1],L*m1,m1,alpha-beta,beta*m2,m2)
```

```
if __name__ == "__main__":
```

```
    plt.close("all") # Close all figures
```

```
orig_size = 1000.0  
p = np.array([[0,0],[0,orig_size],[orig_size,orig_size],[orig_size,0],[0,0]])  
print('Points in original square:')  
print(p)
```

```
fig, ax = plt.subplots()  
draw_squares(ax,6,p,.1)  
ax.set_aspect(1.0)  
ax.axis('off') # Uncomment to see coordinates in drawing  
plt.show()  
fig.savefig('squaresa.png')
```

```
fig, ax = plt.subplots()  
draw_squares(ax,10,p,.2)  
ax.set_aspect(1.0)  
ax.axis('off')  
plt.show()  
fig.savefig('squaresb.png')
```

```
fig, ax2 = plt.subplots()  
draw_squares(ax2,5,p,0.3)  
ax2.set_aspect(1.0)  
ax2.axis('off')  
plt.show()  
fig.savefig('squaresc.png')
```

```
fig, ax = plt.subplots()
```

```
draw_four_circles(ax, 2, [0,0], 100)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('four_circlesa.png')
```

```
fig, ax = plt.subplots()
draw_four_circles(ax, 3, [0,0], 100)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('four_circlesb.png')
```

```
fig, ax = plt.subplots()
draw_four_circles(ax, 4, [0,0], 100)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('four_circlesc.png')
```

''' Figure No. 1 '''

```
fig, ax = plt.subplots()
draw_inside_circles(ax, 3, [0,0], 100, 0.7)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
```

```
fig.savefig('inside_circlea.png')
```

```
fig, ax = plt.subplots()
draw_inside_circles(ax, 6, [0,0], 100, 0.8)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('inside_circleb.png')
```

```
fig, ax = plt.subplots()
draw_inside_circles(ax, 9, [0,0], 100, 0.9)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('inside_circlec.png')
```

```
''' Figure No. 2 '''
```

```
triangle_height = 1000.0
triangle_base = 500.0
p = np.array([[0,0],[triangle_base,triangle_height],[triangle_height,0],[0,0]])
```

```
fig, ax = plt.subplots()
draw_inside_triangle(ax,3,p)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('inside_trianglea.png')
```

```

fig, ax = plt.subplots()
draw_inside_triangle(ax,5,p)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('inside_triangleb.png')

```

```

fig, ax = plt.subplots()
draw_inside_triangle(ax,7,p)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('inside_trianglec.png')

```

''' Figure No. 3 '''

```

triangle_height = 1000.0
triangle_base = 500.0
p = np.array([[0,0],[triangle_base,triangle_height],[triangle_height,0],[0,0]])

```

```

fig, ax = plt.subplots()
draw_triangles(ax,2,p)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('triangle_of_trianglea.png')

```



```
fig, ax = plt.subplots()
draw_triangles(ax,3,p)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('triangle_of_trianglesb.png')
```

```
fig, ax = plt.subplots()
draw_triangles(ax,4,p)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('triangle_of_trianglec.png')
```

''' Figure No. 4 '''

```
fig, ax = plt.subplots()
draw_four_squares(ax,2,[0,0],1000.0)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('four_squaresa.png')
```

```
fig, ax = plt.subplots()
draw_four_squares(ax,3,[0,0],1000.0)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
```

```
fig.savefig('four_squaresb.png')
```

```
fig, ax = plt.subplots()
draw_four_squares(ax,4,[0,0],1000.0)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('four_squaresc.png')
```

```
''' Figure No. 5 '''
```

```
delta_x = 800
delta_y = 200
center = np.array([[0,0],[0,0],[0,0]])
```

```
fig, ax = plt.subplots()
draw_root(ax,5,center[:,0],center[:,1],delta_x,delta_y)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('roota.png')
```

```
fig, ax = plt.subplots()
draw_root(ax,6,center[:,0],center[:,1],delta_x,delta_y)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('rootb.png')
```

```
fig, ax = plt.subplots()
draw_root(ax,7,center[:,0],center[:,1],delta_x,delta_y)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('rootc.png')
```

''' Figure No. 6 '''

```
fig, ax = plt.subplots()
draw_tree(ax,6,[0,0],100,0.6,90.0,45.0,0.9)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('treea.png')
```

```
fig, ax = plt.subplots()
draw_tree(ax,10,[0,0],100,0.7,90.0,60.0,0.7)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('treeb.png')
```

```
fig, ax = plt.subplots()
draw_tree(ax,9,[0,0],100,0.82,90.0,45.0,0.75)
ax.set_aspect(1.0)
ax.axis('off')
```

```
plt.show()
```

```
fig.savefig('treec.png')
```