1 Code journal: Square Spiral

21.07.2020, Mariana Ávalos Arce

1.1 What is constant and what is not?

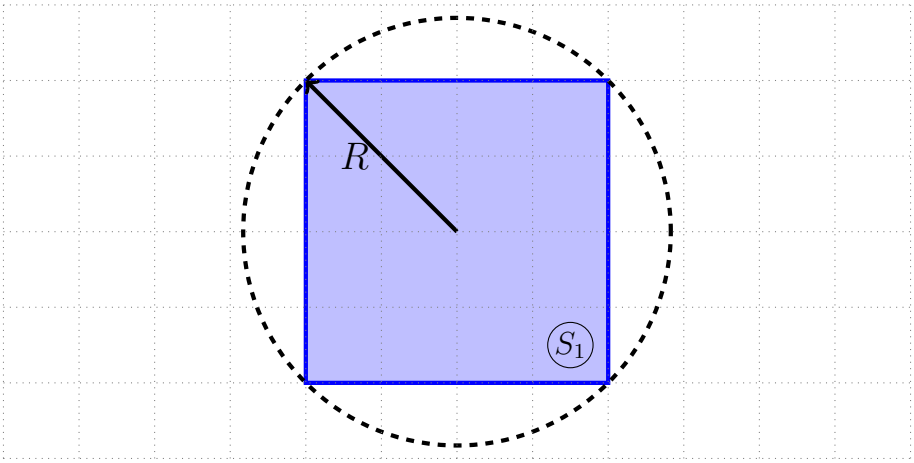
A **Geometric Spiral** is basically the well-known image where a regular polygon rotates and gets bigger and bigger as the image is filled. The image might look deceitful at times, which made me doubt: are both the angle and the size increasing, or just one of them?

1.2 A square, a regular polygon

The first thing to build the image is to create the central **regular polygon**. To build any regular polygon, the whole 360 degrees must be divided by the number of sides in the desired polygon. During this report, the theory will be based only on a **square** spiral. Therefore,

$$\text{step angle} = \frac{360}{sides} = \frac{360}{4} = 90\,\text{deg} \tag{1}$$

Which tells us that every 90 degrees a square vertex is positioned. Then, we just need a custom radius or polygon size to draw square 1 (S_1) inside a circumference of radius R (or size), following the idea below. A simplified code for the mentioned concept is also shown further below.



```
acc_angle = 0.0; acc_size = size
last_point = [0.0, 0.0]; first_point = [0.0, 0.0]
    for i in range(num_sides):
    angle = i * step_angle - (step_angle / 2.0) - acc_angle
    x = fig_center[0] + (acc_size) * math.cos(angle * PI / 180.0)
    y = fig_center[1] + (acc_size) * math.sin(angle * PI / 180.0)
    if i == 0:
            first_point = [x, y]
    if i > 0:
            ln.line([x, y], last_point, thick, color, img)
    if i == num_sides - 1:
            ln.line([x, y], first_point, thick, color, img)
    last_point = [x, y]
```

1.3 One square leads to another

The above mentioned algorithm results in the first square at the center of the image, which will be also the smallest. The following iterations of squares in an external loop answer the first question: what is constant, the angle or the size increase in each square? If we look at the figure below, the blue lines are the Δa or the slight change in the rotation angle of each square, and we see that **each square follows a constant angle increase** when compared to the previous one.

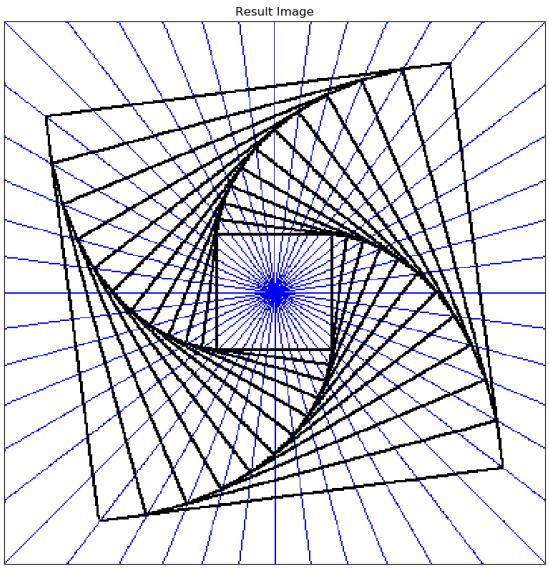
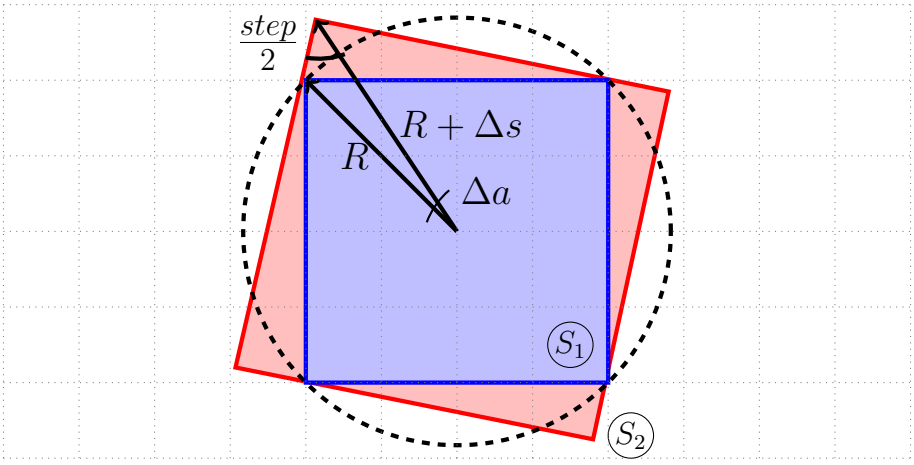


Figure 1: Test showing constant angle

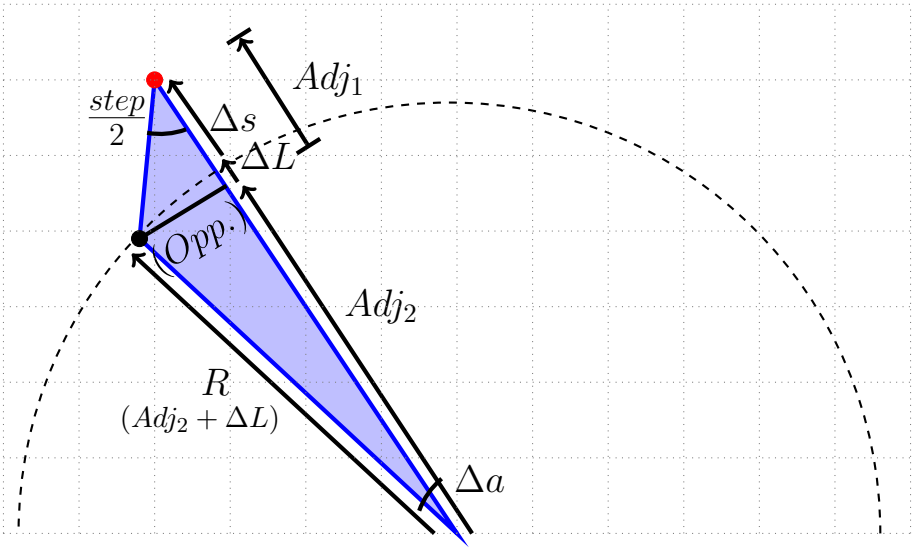
The test image also shows that each of the blue lines slides the `step_angle` in half, which will be crucial to calculate the **increase in size** or Δs of each square so that its sides exactly coincide with the last square's sides. This increase in size (or radius) will then be **larger** in every square drawn. To explain this, let's take an example of a two-square spiral, as in the diagram below.



1.4 Predicting the next size

In order to calculate the exact change in size *for the next square* so that the sides touch each other exactly, we must take on account that we know the value of two crucial angles: the custom Δa

and the angle $\frac{step}{2}$. If we take the triangle and amplify it a bit as in the following diagram, we get a more evident discovery to know the value of Δs, and with this, the red point will be given now that the new accumulated radius or size gets increased with the new value; allowing the size value of the next square to be known at the *end* of each iteration.



Thus, taking into account some trigonometry,

$$\text{Opp.} = \text{size} \times sin(\Delta a), \tag{2}$$

Which leads to

$$\frac{tan(\frac{step}{2})}{1} = \frac{Opp.}{Adj_1}, \tag{3}$$

$$\frac{Adj_1}{Opp.} = \frac{1}{tan(\frac{step}{2})} \tag{4}$$

$$\boxed{Adj_1 = \frac{Opp.}{tan(\frac{step}{2})} = \frac{\text{size} \times sin(\Delta a)}{tan(\frac{step}{2})}} \tag{5}$$

But then, we also need to notice

$$Adj_2 = \text{size} \times cos(\Delta a), \tag{6}$$

$$\Delta L = |\text{size} - Adj_2|, \tag{7}$$

So that we can finally get

$$\boxed{\Delta s = |Adj_1 - \Delta L|} \tag{8}$$

We now can complete the code with Δs (`delta_size`) as the following:

```
for j in range(num_iter):
    last_point = [0.0, 0.0]; first_point = [0.0, 0.0]
    for i in range(num_sides):
        angle = i * step_angle - (step_angle / 2.0) - acc_angle
        x = fig_center[0] + (acc_size) * math.cos(angle * PI / 180.0)
        y = fig_center[1] + (acc_size) * math.sin(angle * PI / 180.0)
        if i == 0:
            first_point = [x, y]
        if i > 0:
            ln.line([x, y], last_point, thick, color, img)
        if i == num_sides - 1:
            ln.line([x, y], first_point, thick, color, img)
        last_point = [x, y]
    adj1 = (acc_size * math.sin(delta_angle * PI / 180.0)) / (math.tan((step_angle / 2.0) * PI / 180.0))
    adj2 = acc_size * math.cos(delta_angle * PI / 180.0)
    delta_L = abs(acc_size - adj2)
    acc_angle += delta_angle
    delta_size = abs(adj1 - delta_L)
    acc_size += delta_size
```

This code creates a number of iterations where each makes a square slightly rotated by a custom `delta angle` that remains **constant during all** the iterations. Nevertheless, each square should **grow its size** or radius, but not by a constant amount. The increase in size of the squares should obey the relation in Eq.(5) and, more precisely, accumulate the size with Eq.(8) at the end of every square drawing. The result is the image shown below.

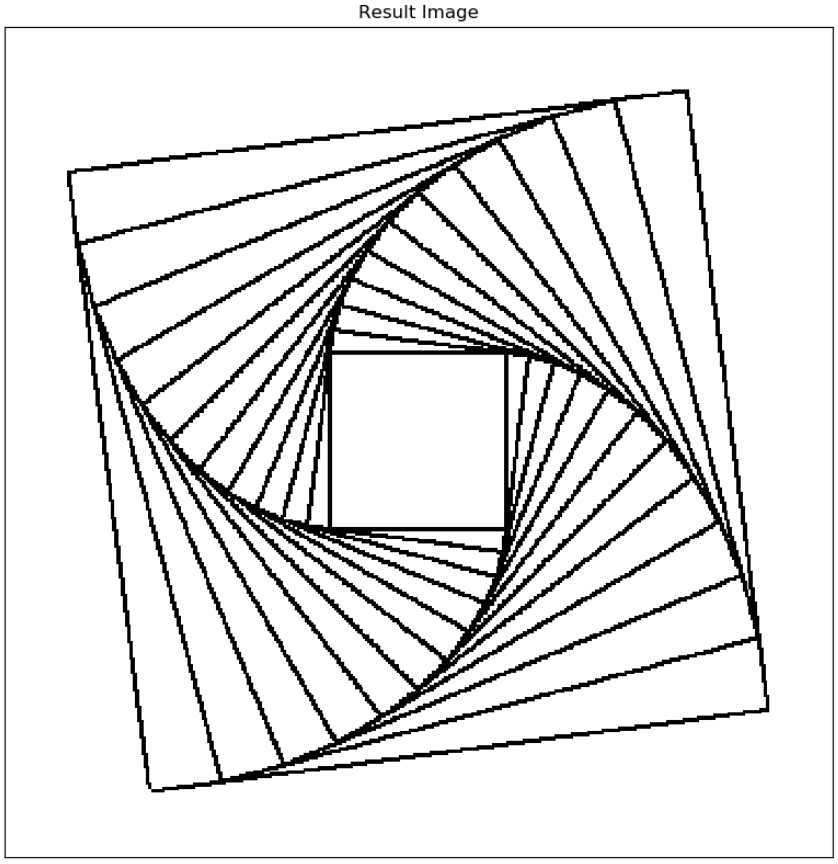


Figure 2: Final Geometric Spiral using Python 3.6.2

1.5 Further results

The program uses `delta_angle` as a variable to store the Δa, but the variable is not fully a custom variable: it depends on the number of sides of the polygon (in this case 4 to form the square) and the number of iterations the user wants. Thus, this variable and others are defined as the following for the program to work in a complete way.

```
img = np.zeros([400, 400,3],dtype=np.uint8); img[:] = (255, 255, 255)
color = [0, 0, 0]; thick = 2
num_sides = 4 # as it needs to be a square
fig_center = [0.0, 0.0]
size = 60 # in pixels
num_iter = 12
angle_dist = 2
delta_angle = (angle_dist * 180.0 / (num_sides)) / num_iter
acc_angle = 0.0
acc_size = size
step_angle = 360.0 / num_sides
```

Further testing of the explained program done with Python 3.6.2 gave the output below. The only parameter changed by the user is the number of iterations.

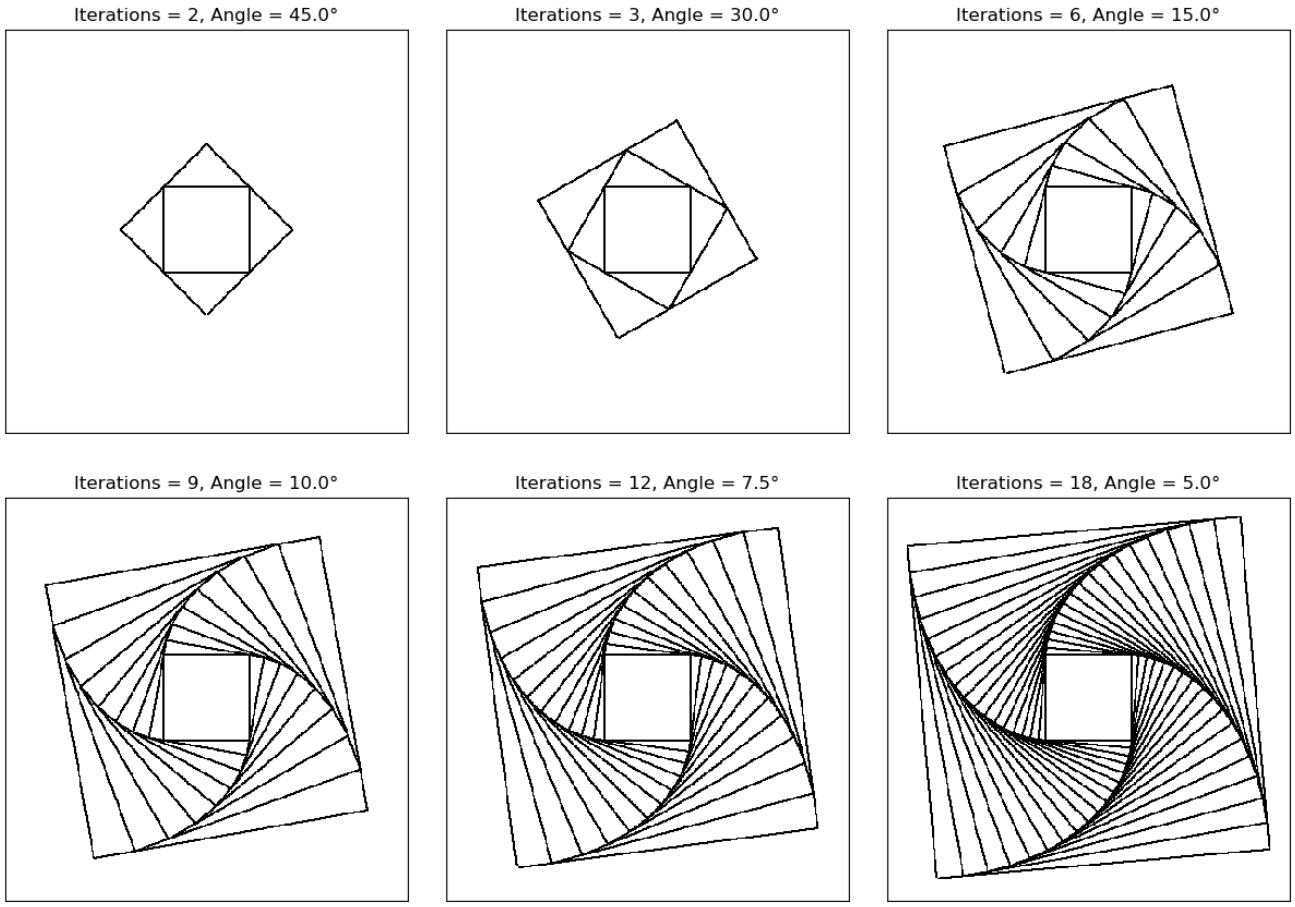


Figure 3: Program tests following Eq.(8)