

# 1 Code journal: Sierpinski Triangle in Python

20.12.2020, Mariana Ávalos Arce

## 1.1 The Sierpinski Triangle

The **Sierpinski Triangle** is basically a fractal set with the general shape of a triangle. Each triangle is composed of three smaller triangles, and the more iterations, the more sub triangles there will be.

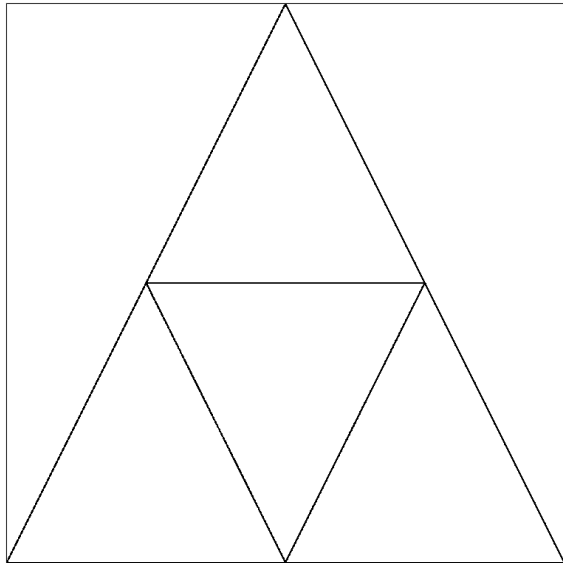


Figure 1: Sierpinski Triangle, 1 iteration

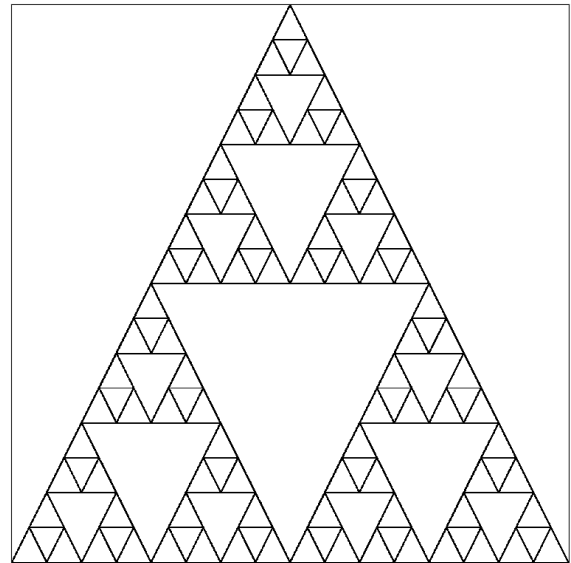
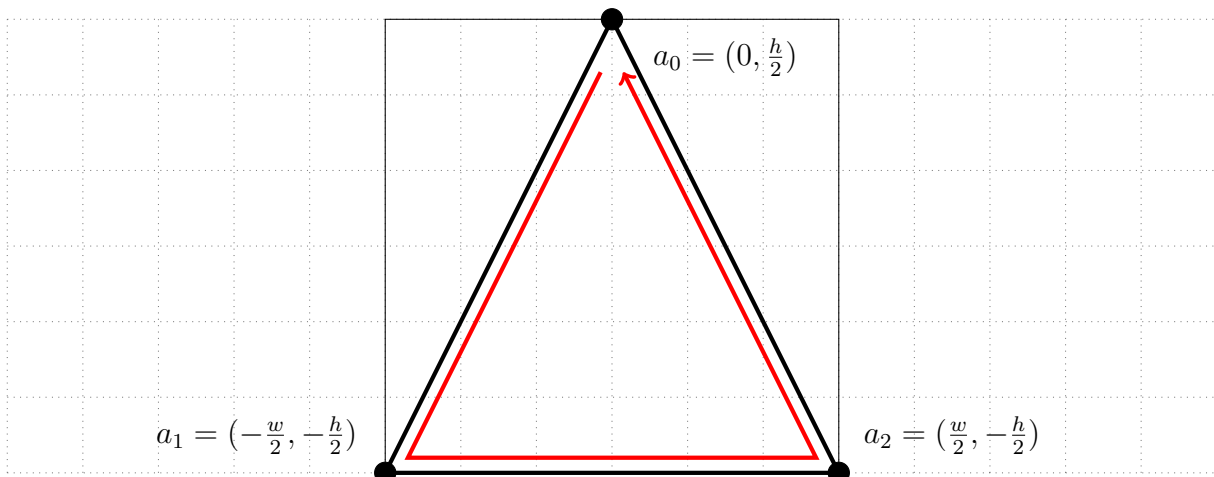


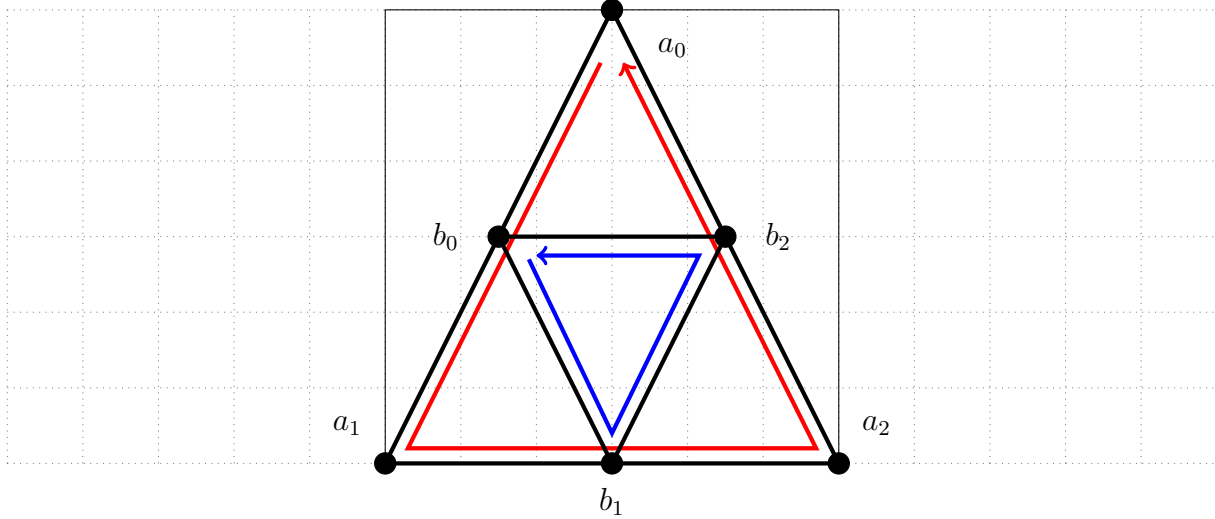
Figure 2: Sierpinski Triangle, 4 iterations

## 1.2 Setting the Base Triangle

If the fractal is going to be painted on an image, the first thing is to define the base triangle (biggest triangle) points. Let's take a 12x12 pixel image, **centered** in a cartesian plane.



Then, the approach that will be **recursively** used is the following idea: for each triangle  $a$ , a sub-triangle  $b$  must be painted using the **3 middle points** that exist in each of the 3 edges of said main triangle. The new sub-triangle vertices will form the triangle following also the **counter-clockwise** orientation shown in the previous diagram. Such ideas will look as follows, if only the first iteration is done.



### 1.3 Generating the Recursive Sequence

Thus, following both arrows of each triangle and its child triangle, the points are stored in a **vertex list**, the indexed vertex sequence would go:  $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0$ , which in a loop that draws lines from  $p_0$  to  $p_1$  with iterator  $i$  would be better expressed as  $\text{vertex}[i] \rightarrow \text{vertex}[(i + 1) \% 3]$ . This would draw both the triangle and its sub-triangle lines. Then, we just need to order the logic as a **recursive** function where its stop condition would be whenever the **N iterations** are completed. Each iteration is understood as the drawing of a base triangle  $a$  and a sub-triangle  $b$ . The algorithm went as below.

---

```
def iterate(tri_pts, curr_thick, curr_iter):
    curr_iter += 1
    if curr_iter <= num_iter:
        for i in range(len(tri_pts)):
            ln.line(tri_pts[i], tri_pts[(i + 1) % len(tri_pts)], curr_thick, color, img)
        new_tri = []
        for i in range(len(tri_pts)):
            mid = [int((tri_pts[i][0] + tri_pts[(i + 1) % len(tri_pts)][0]) / 2.0),
                    int((tri_pts[i][1] + tri_pts[(i + 1) % len(tri_pts)][1]) / 2.0)]
            new_tri.append(mid)
        for i in range(len(new_tri)):
            ln.line(new_tri[i], new_tri[(i + 1) % len(new_tri)], curr_thick, color, img)
        curr_thick *= dt
        for i in range(len(tri_pts)):
            sub_tri = [tri_pts[i], new_tri[i], new_tri[(i + 2) % len(tri_pts)]]
            iterate(sub_tri, curr_thick, curr_iter)
    else:
        return
```

---

## 1.4 Program Output

After defining the function `iterate()`, what is left is just to call it in a structure similar to the one below.

---

```
thick = 5; dt = 1
num_iter = 4; curr_iter = 0

base_triangle = [[0, int(h/2.0 - thick/2.0)], [int(w/2.0 - thick/2.0),
    -1*int(h/2.0 - thick/2.0)], [-1*int(w/2.0 - thick/2.0), -1*int(h/2.0 - thick/2.0)]]
iterate(base_triangle, thick, curr_iter)
```

---

This generated a Sierpinski Triangle with custom thickness, iterations and thickness reduction (dt). A sample output was the following.

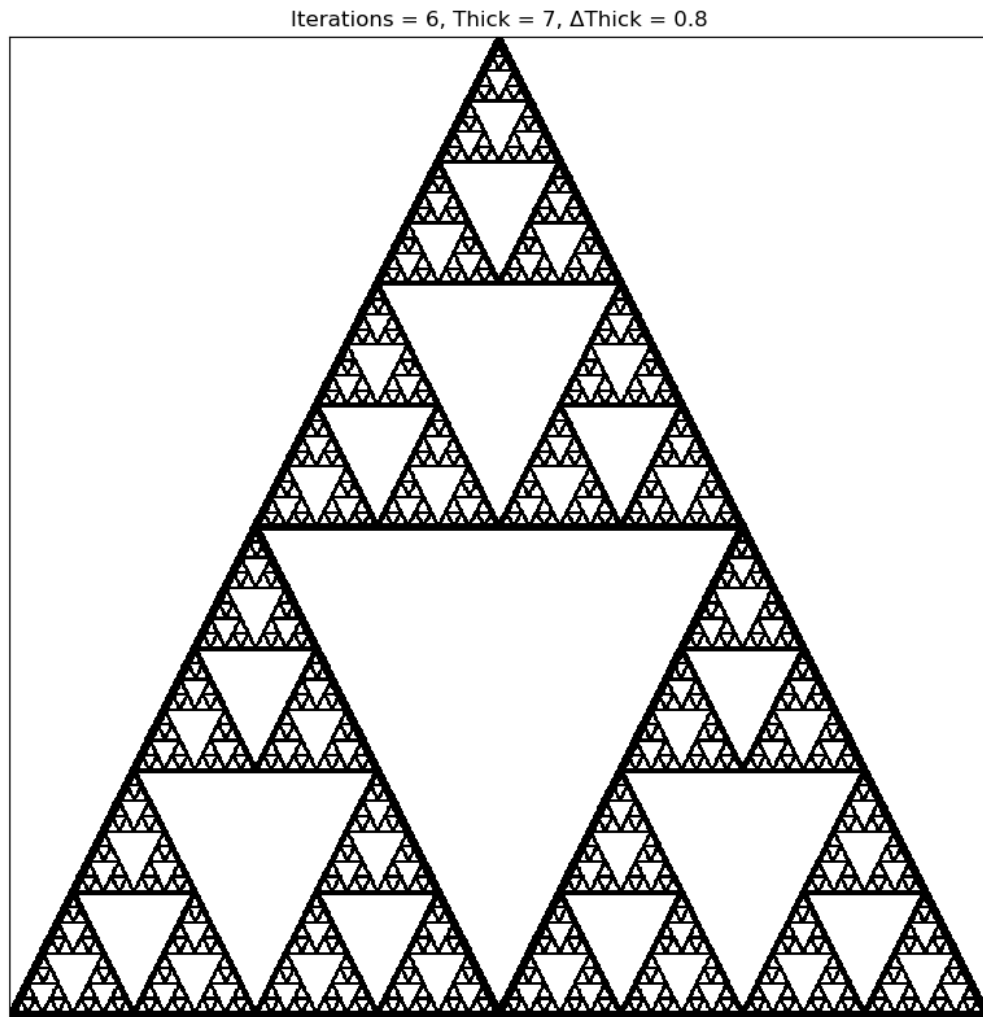


Figure 3: Sierpinski Triangle implementation in Python 3