Code journal: Koch Curve in Python

09.01.2021, Mariana Ávalos Arce

The Koch Curve

The **Koch Curve**, also known as Koch Snowflake or Koch Star, is a **fractal curve** developed by mathematician Helge von Koch, which involves basic elementary geometry concepts for its construction. The algorithm that follows is **recursive**.

It is **recursive** mainly because it is a fractal curve, which means that the overall curve is constructed by smaller-scale of its identical basic figure. In the case of the Koch Curve, the idea is to take a line from point p_0 to p_1, and take into account the number of sides we want (S), in this case S = 3 for a triangle, and do the following: divide the line in S parts where the resulting length will be called a, then after the `int(S/2.0)` part (1 for a triangle), start by putting the S - 1 sides left to form the figure desired, and repeat this **for every edge (line)** that the overall curve has, until the total number of iterations (N) is completed. During each iteration, the length of the regular polygon's sides (in this case an equilateral triangle) will always be current a value. For a curve that starts as an equilateral triangle (iteration 0), the first, second and third iterations would look as below.

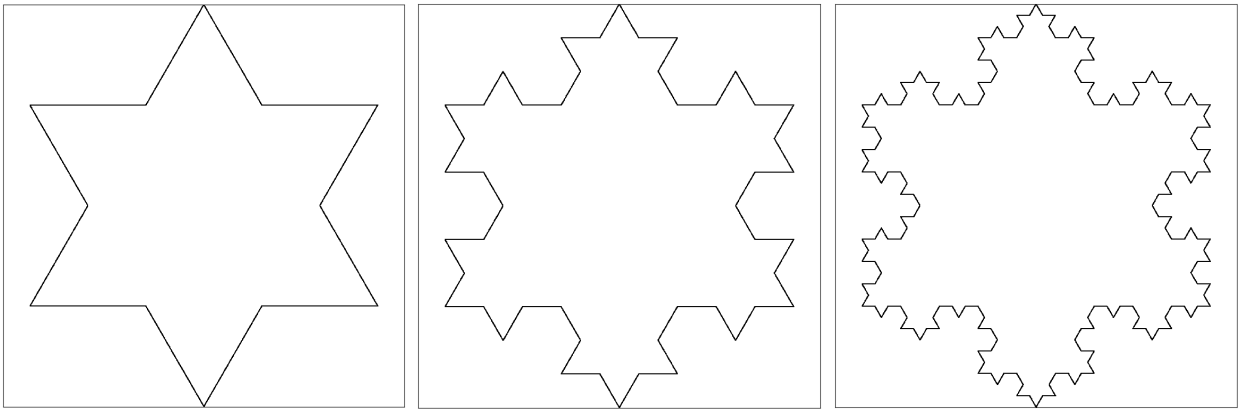

Figure 1: Koch Curve until 3 iterations

Building An Algorithm

The process described above needs to use two mathematical concepts: **vectors** and **rotation matrices** for two dimensions, since we want to paint it over a 2D image.

First thing is to calculate the vector from p_0 to p_1 and call it $\boldsymbol{v}$,

$$\boldsymbol{v} = \langle x_1 - x_0,\, y_1 - y_0 \rangle \tag{1}$$

Then, calculate the distance from these points using the Pythagorean Theorem, or in other words, calculate the length of $\boldsymbol{v}$ and call it d,

$$d = \sqrt{(x_v)^2 + (y_v)^2} = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} \tag{2}$$

And now we can calculate the value of our variable a,

$$a = \frac{d}{\text{Sides}} \tag{3}$$

Which will be the length of current iteration's polygon sides. Now, a useful tool is to normalize the vector $\boldsymbol{v}$ to get the pure direction of the line ready to be multiplied by some factor of a as we move on in the algorithm,

$$\hat{\boldsymbol{v}} = \frac{1}{d} \cdot \boldsymbol{v} = \frac{1}{d} \langle x_1 - x_0,\, y_1 - y_0 \rangle \tag{4}$$

Now, we are ready to continue the iteration. the point where we will start to paint our new polygon will be described as

$$p_{start} = \texttt{int(S/2.0)} \cdot a \cdot \hat{\boldsymbol{v}} \tag{5}$$

Thus, the next step is to take $\hat{\boldsymbol{v}} \cdot a$ and rotate it using a **rotation matrix** defined as follows:

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \cdot \hat{\boldsymbol{v}}_x \\ a \cdot \hat{\boldsymbol{v}}_y \\ 0 \end{pmatrix} = \begin{pmatrix} \boldsymbol{pt}_x \\ \boldsymbol{pt}_y \\ \boldsymbol{pt}_z \end{pmatrix} = \boldsymbol{pt}, \tag{6}$$

Where θ is the **inner angle** of the said regular polygon defined as

$$\theta = -1 \cdot \left| \frac{360}{\text{Sides}} - 180 \right| \tag{7}$$

The -1 is due to the fact that the way I coded the algorithm starts drawing all triangles with a **counter-clockwise** orientation, and therefore, their inner angle is using the **clock-wise** orientation. θ normally goes counter-clockwise in the rotation matrix.

Anyway, going back to the next step: make $\boldsymbol{pt}$ a 2D point by simply eliminating the z value, or making the matrix-vector a 2D multiplication. At that point, the first new point of the added polygon is given by

$$p_i = \boldsymbol{pt} + p_{start} \tag{8}$$

Fortunately, for the next sides of the polygon we just need to rotate another vector resulting from the substraction of the new p_i point and the last iteration's point for each of the remaining sides. Then, the function must call itself for each of this new edges, with the stop condition being when N iterations are reached.

Pentagonal Variant

The mentioned algorithm works for any odd-sided regular polygon, such as triangles, pentagons, heptagons, etc. The triangle variant creates the well known Koch Snowflake (shown in first section), but another famous version is the pentagonal variant. The resulting Pentagonal Variant with this algorithm looks as follows.

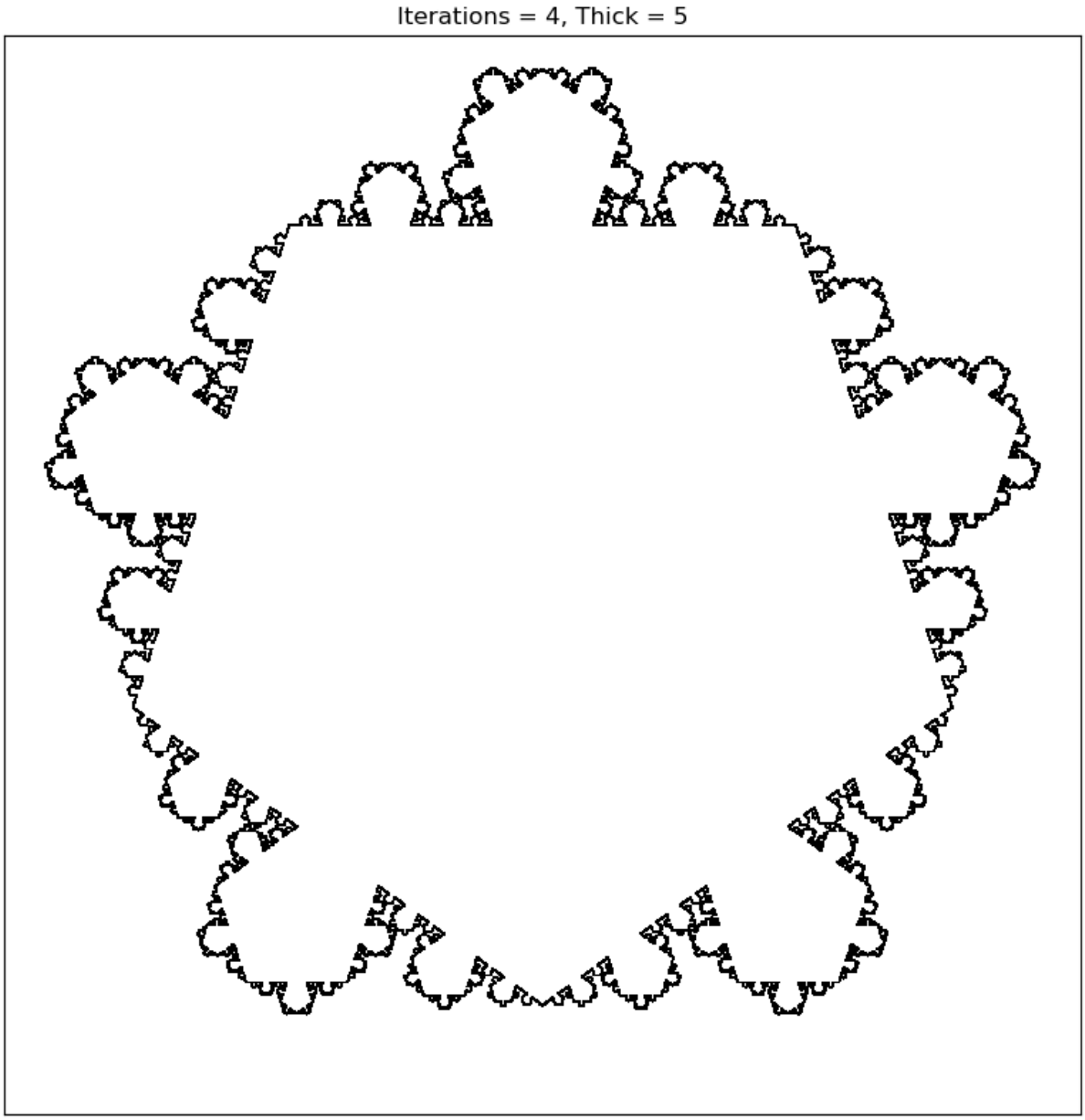

Figure 2: Pentagonal Koch Curve Variant implementation in Python 3