MATH1220 Project 2: Fractal Properties and Generation of Fractals Using Code

Hogan Choi

4/8/2024

Contents

1 Introduction

Fractals are geometric shapes that exhibits the property self-similarity at different scales. In other words, when we zoom into a fractal, we'll be able to see infinite smaller copies of the overall shape. Due to its recursive properties, calculating their size is near impossible. We can determine complexity of a fractal structure by using the self-similarity definition,

$$D = \frac{Log(N)}{Log\left(\frac{1}{r}\right)}$$

where N is the number of copies of the original shape and r is the dilation factor. Hence, the higher the value of D is, the more complex the overall structure is.

Fractal-like structures can be found in nature as well. For example, the spiral shells of ammonites (an extinct cephalopod) were made up of complex fractal curves. Certain mountains, such as the Himalayas, would have branches resembling the shape of tree fractals. Other natural objects, such as leaves and lightning, would also inherit similar traits.

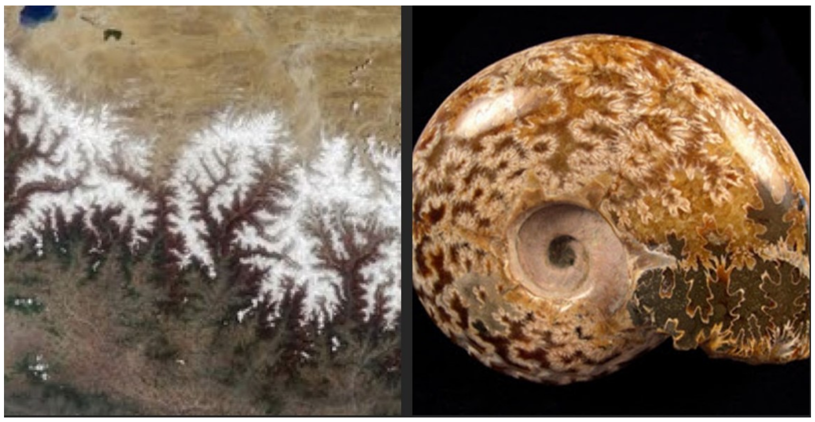

Figure 1: Fractals exhibited by Ammonite Shell and Himalayan Mountains.

In this project, I will go over some of the four well-known fractal structures: Tree Fractals, Sierpinski's Triangle, The Koch Curve, and Mandelbrot's Fractal. I will discuss their mathematical properties and provide code in Python for generating these structures.

2 Tree (Canopy) Fractals

Tree Fractals, also known as Fractal Canopies, are one of the most basic fractals and easiest to create. These fractals are lines that branch out into other, similar-like, trees. These fractals are exhibited in nature through canopy trees and lightning.

Figure 2: Canopy Fractals in Trees and Lightning.

The algorithm needed to create Tree Fractals must exhibit the following three properties:

1. Establish an angle that will be used throughout the entire graph.
2. Ratio of the lengths of two consecutive branches are equivalent.
3. The graph must be connected, meaning the edges (branches) must go through every node.[3]

Hence, let's assume we have a branch that goes from coordinate A to coordinate B. We have a cutoff point at P, which is determined by the ratio C. From there, we can diverge into our two branches, which will have endpoints B' and B'' and a separation angle of α. Below is a graphic demonstaration of how one branch can be separated into two.

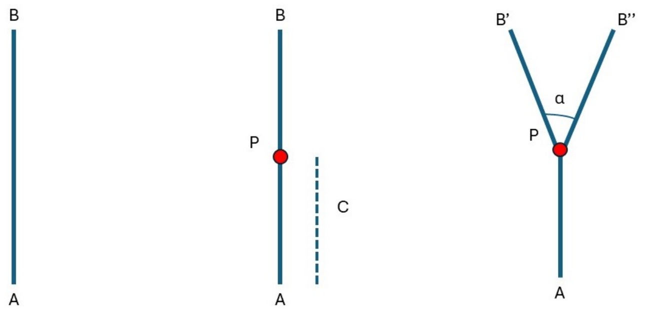

Figure 3: Creating Branches in Tree Fractals.

We can mathematically calculate the coordinates for B' by using trigonometry.

$$\text{Length of Branch} = (1 - C) \times (\text{Length of Tree})$$

$$B'_x = B_x + ((\text{Length of Branch}) \times cos\left(\frac{\alpha}{2}\right))$$

$$B'_y = B_y + ((\text{Length of Branch}) \times sin\left(\frac{\alpha}{2}\right))$$

To find B'', we would repeat the same equation, except use $-\frac{\alpha}{2}$.

$$B''_x = B_x + ((\text{Length of Branch}) \times cos\left(-\frac{\alpha}{2}\right))$$

$$B'_y = B_y + ((\text{Length of Branch}) \times sin\left(-\frac{\alpha}{2}\right))$$

We can recreate this mathematical algorithm using code in Python. We can plot lines from point A to point B and use sin and cos to calculate B' and B''. After, we can implement a for loop that will iterate and create branches until it reaches a certain limit addressed by the user.

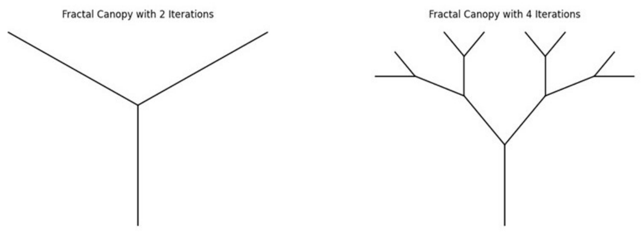
Fractal Canopy with 2 Iterations
Fractal Canopy with 4 Iterations
Fractal Canopy with 7 Iterations

Figure 4: Different Iterations of Tree Fractals from Code.

We can create more diverse structures by changing the angle of seperation, such as shown below.

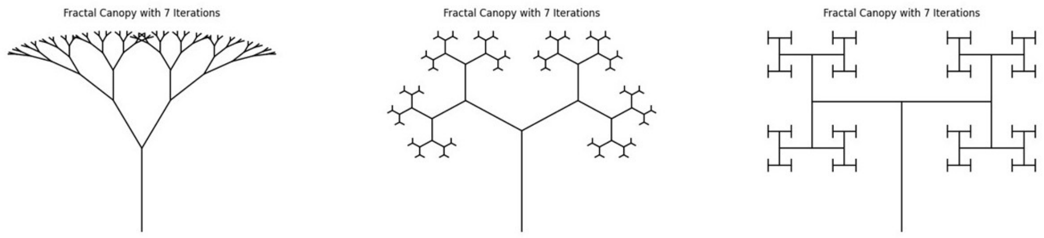
Fractal Canopy with 7 Iterations
Fractal Canopy with 7 Iterations
Fractal Canopy with 7 Iterations

Figure 5: Canopy Structures with Different Angles.

3 Sierpinski's Triangle

The Sierpinski's Triangle was invented by Polish mathematician, Waclaw Sierpinski, in 1915. In order to create a Sierpinski's Triangle, we first need to identify the midpoints of a given triangle.[1]

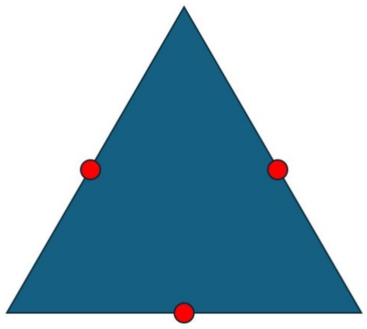

Figure 6: Identify Triangle Midpoints.

We would then connect the midpoints to create another triangle within the original triangle structure.

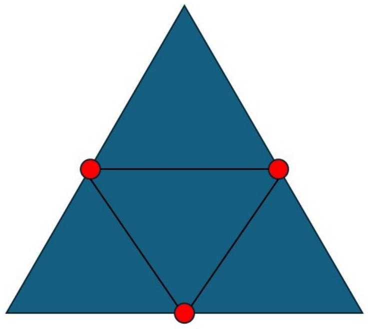

Figure 7: Connect Midpoints to Create New Triangle.

Finally, we can take out the inner triangle, giving us a figure of three triangles stacked up. This would be the first iteration in a Sierpinski's Triangle.

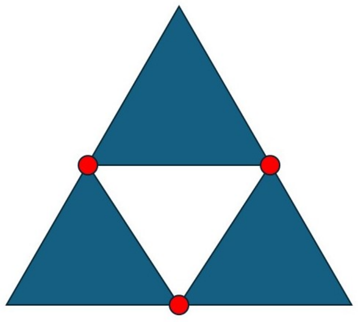

Figure 8: Remove Inner Triangle to Create Sierpinski's Triangle Step One.

To replicate these steps in code, we first need to locate the midpoints of all three sides. This can be done by using the equation below.

For arbitrary coordinates A and B

$$M_x = \frac{A_x + B_x}{2}$$

$$M_y = \frac{A_y + B_y}{2}$$

After finding the midpoints, we can connect them using `plt.plot`. This would create a new triangle variable that we can edit independantly to the original structure. Hence, we can color this inner triangle with a different color from the original triangle, giving us a Sierpinski Triangle. By applying this algorithm through a for loop, we can create a much more complex Sierpinski's Triangle, as shown below.

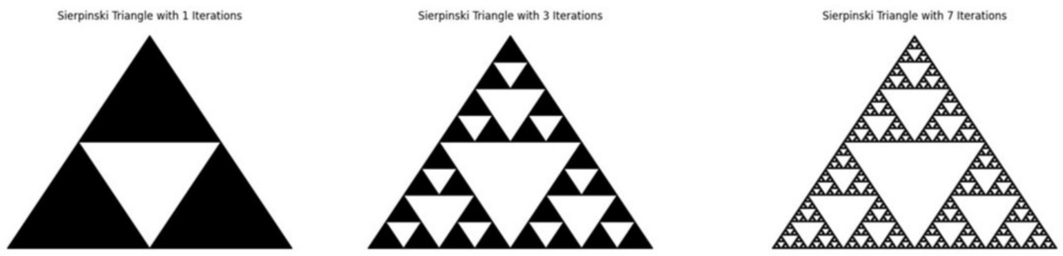
Sierpinski Triangle with 1 Iterations
Sierpinski Triangle with 3 Iterations
Sierpinski Triangle with 7 Iterations

Figure 9: Sierpinski's Triangle Strucutres from Code.

4 The Koch Curve

The Koch Curve was introduced by Swedish mathematician Helge von Koch. This fractal is also known as the Koch Snowflake, due to its similar structure when compared to a snowflake (as shown below).[5]

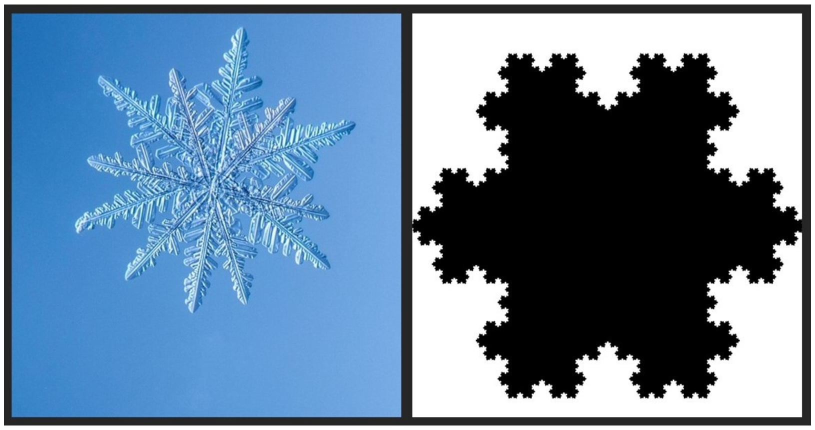

Figure 10: Snowflake and a Koch Snowflake Fractal.

For this project, we'll only look at the Koch Curve, which is exhibits fractal structures on a single line.

The Koch curve is created by first dividing a line into three equal segments. Then, remove the middle semgent and create a triangle in place of the removed middle segment. We can continue this process for all the remaining line segments. By doing this recursive process, we would be able to create a fractal structure. Below is a graphical representation of creating the first step in a Koch Curve.[5]

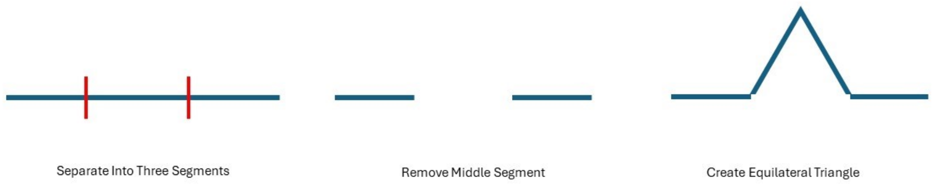

Figure 11: Creation of Koch Curve.

Locating and removing the middle segment of a line is quite straightforward. This can be done by finding $\frac{1}{3}$ of the length of the line and adding it to the starting point and subtracting it from the ending point. However, finding the coordinate of the created triangle requires more steps, as it involves angles.

From Euclidean geometry, we know that rotating coordinate x by θ counterclockwise is that same as multiplying x by the matrix below.

$$x' = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Since we're creating an equilateral triangle when generating a Koch Curve, we could define $\theta = \frac{\pi}{3}$. Hence, if coordinate C is our endpoint of the middle line segment, we can rotate by $\frac{\pi}{3}$ radians to get our triangle coordinate.

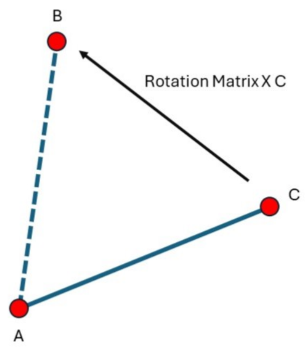

Figure 12: Rotating Coordinate for Koch Curve

The equation for the following rotational transformation can be seen below.

$$B = \begin{bmatrix} cos(\frac{\pi}{3}) & -sin(\frac{\pi}{3}) \\ sin(\frac{\pi}{3}) & cos(\frac{\pi}{3}) \end{bmatrix} \begin{bmatrix} C_x \\ C_y \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} C_x \\ C_y \end{bmatrix}$$

Using this property, we can now create our code in Python. We would use a for loop to call the above equation for a certain amount of times. The greater the number of iterataions, the more complex our Koch Curve would look like.

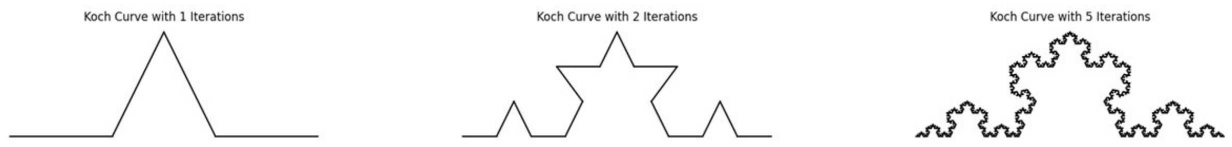
Koch Curve with 1 Iterations
Koch Curve with 2 Iterations
Koch Curve with 5 Iterations

Figure 13: Iterations of the Koch Curve Generated by Code

5 Mandelbrot Fractal

The Mandelbrot Set was first defined by mathematicians Robert W. Brooks and Peter Matelski in 1978.[2] It was later refined by Benoit Mandelbrot in 1980. The Mandelbrot Set contains complex numbers that do not diverge into infinity when sequenced through the following equation.[4]

$$Z_{n+1} = Z_n^2 + C$$

For example, the complex number 1 doesn't belong in the Mandelbrot Set because Z explodes to infinity as the equation is called multiple times. [6]

$$\begin{aligned} Z_0 &= 1 \\ Z_1 &= 2 \\ Z_2 &= 5 \\ Z_3 &= 26 \\ &\vdots \\ Z_9 &= 1.95 \times 10^{45} \end{aligned}$$

However, complex numbers, such as -1, would belong in the Mandelbrot Set since -1 would repeat itself after two iterations of the function. [6]

$$\begin{aligned} Z_0 &= -1 \\ Z_1 &= 0 \\ Z_2 &= -1 \\ Z_3 &= 0 \end{aligned}$$

To demonstrate this in code, we would first create a imaginary plane where the real numbers would be plotted on the x-axis and the imaginary numbers would be plotted on the y-axis. We would then give a range for the x-axis and the y-axis (e.g. $real \in [-2, 2]$ and $imaginary \in -1, 1)$. From this range, we would generate a uniformly distributed number of samples. Based on the number of samples generated, the resolution for our fractal will differ greatly (more samples means higher resolution).

We would then need to set a thresholed value and the number of times the equation is called. For example, if we set the threshold to 2 and the number of iterations to 30, this would mean we would calculate Z_{30} for a given number and see if its abste value is less than or equal to 2. If it's less than or equal to 2, we can assume that the particular complex number doesn't diverge into infinity. Therefore, the greater the threshold value, the more samples we would most likely accept.

On the contrary, the greater the iteration number, the more samples we'll most likely deny.[6]

By recreating this algorithm in code, we can now generate a Mandelbrot fractal structure by giving it the parameters previously discussed. Below are two Mandelbrot fractal shapes with varying number of pixels.

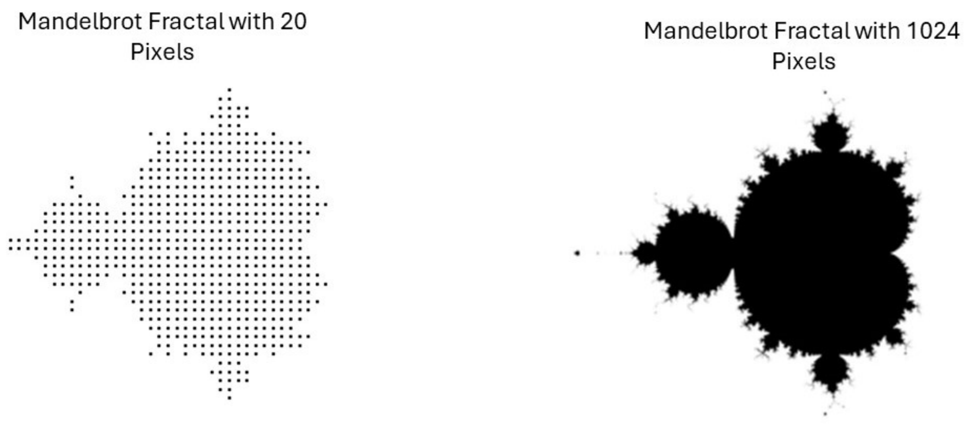
Mandelbrot Fractal with 20 Pixels
Mandelbrot Fractal with 1024 Pixels

Figure 14: Mandelbrot Fractal with Different Resolutions

As you can see, the structure generated on the left is much simpler than the structure generated on the right. The left structure is the one that was first introduced by Brooks and Matelski. The right structure was the shape that was revised by Mandelbrot in 1980. We can clearly see the fractalization formed on the right figure is much more defined and complex compared to the left.

6 Conclusion

From this project, I was able to understand more about the fundamental properties of certain fractals. Before this project, I thought that there was one universal formula for creating fractals, yet now I understand that each structure has their own unique algorithm. I also believe that by creating code that generated fractals, I was able to more clearly understand how each structure is developed. Instead of seeing and understanding their mathematical properties, by personally creating the algorithm helped me visualize each step of generating a new self-similar shape.

To access the code used in this project, please use this github repo link: `https://github.com/hoganchoi1/Fractal_Drawings.git`.

References

[1] Cynthia Goforth. Sierpinski triangle — definition, pattern and formula.

[2] Irwin Kra and Bernard Maskit. *The Dynamics of 2-Generator Subgroups of PSL(2, C).*, page 1–2. University of Tokyo Press, 1981.

[3] Magazine. Fractal tree, Aug 2022.

[4] Benoit B. Mandelbrot. Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, Dec 1980.

[5] Larry Riddle. Koch curve.

[6] Bartosz Zaczyński. Draw the mandelbrot set in python, Oct 2023.