Part 1

The first fractal is the Chaos game. It starts off by defining the vertices of a triangle and one random point.

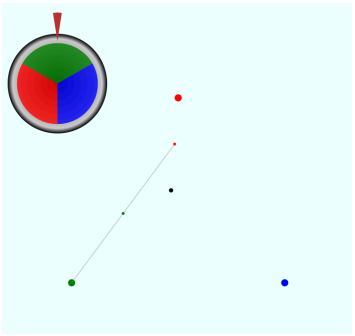
```
vertices = [(0, 0), (2, 2*np.sqrt(3)), (4, 0)]
```

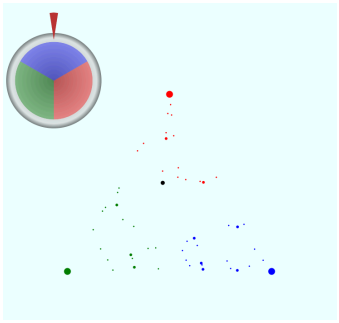
```
x, y = [0]*iterates, [0]*iterates
x[0], y[0] = random(), random()
```

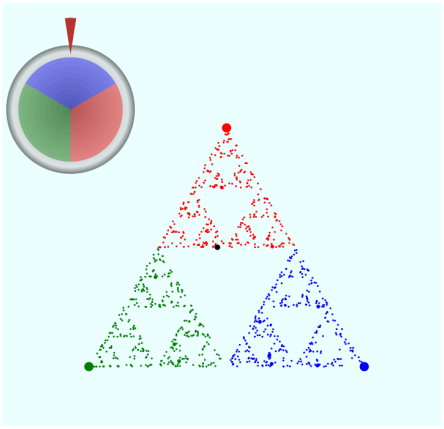
The fractal is similar to the Sierpinski triangle that we worked on earlier in the semester, with the difference that points are plotted based on a random number selection below, where a 0, 1, or 2 is chosen.

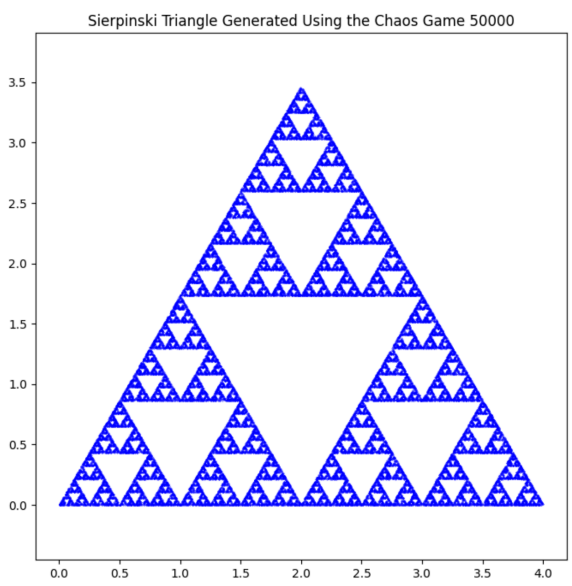
```
for i in range(1, iterates):

    k = randint(0, 2)

    x[i], y[i] = midpoint( vertices[k], (x[i-1], y[i-1]) )
```

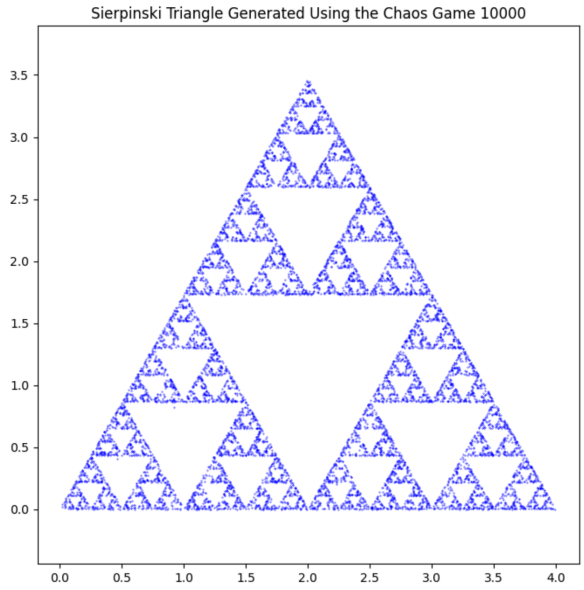
The number selected will determine which of the three vertices is impacted. A point is plotted halfway between the initial random point and the vertex selected. A new random number is chosen and a point is plotted halfway between the previous point and the selected vertex. This is repeated for the number of iterations that was defined, initially 50000. I found a great aid at https://thewessens.net/ClassroomApps/Main/chaosgame.html where you spin a wheel and it plots each of the following points, after 2, 50, and 1000 spins:

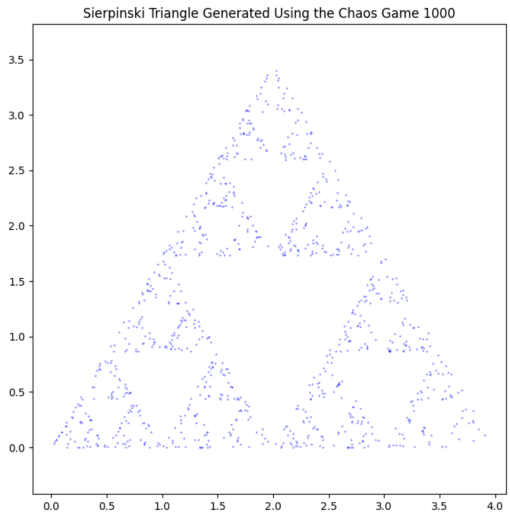






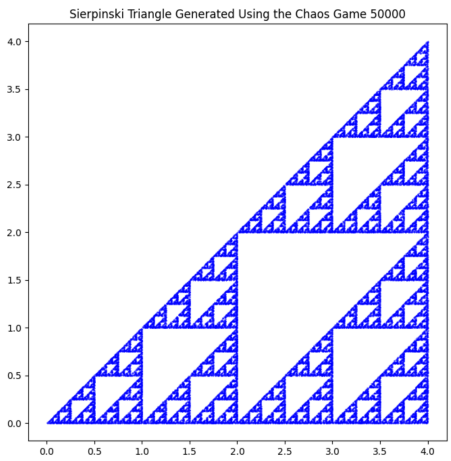


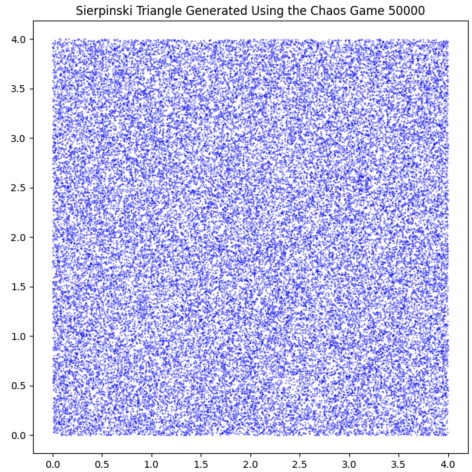




It is hard to see on the triangle on the left, but the points are not evenly distributed in each triangle being formed, and as such, none of the smallest triangles are exactly the same, the results over the long haul are close enough to the human eye.

Going for creativity points, I next created a square but continued (accidentally) selecting a number from 0 through 3 and got the following below on the left. Setting it 0 to 4 creates what looks like Starry Night, but makes sense that it would be scattered as seen below on the right.

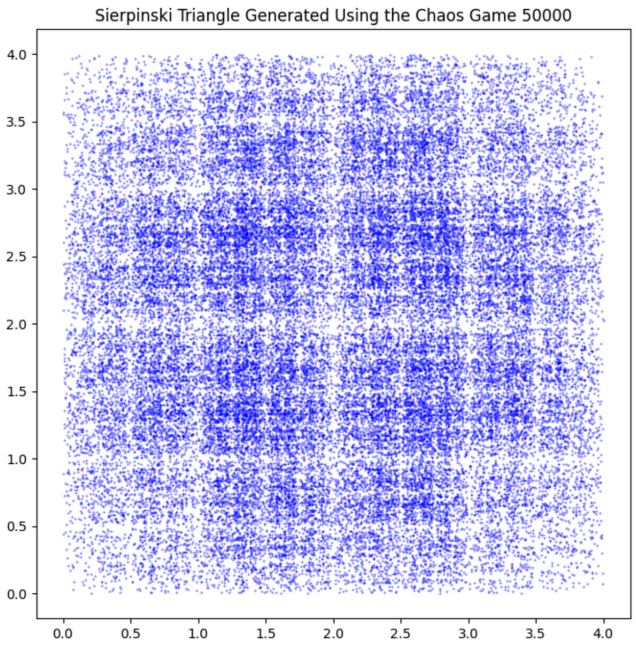


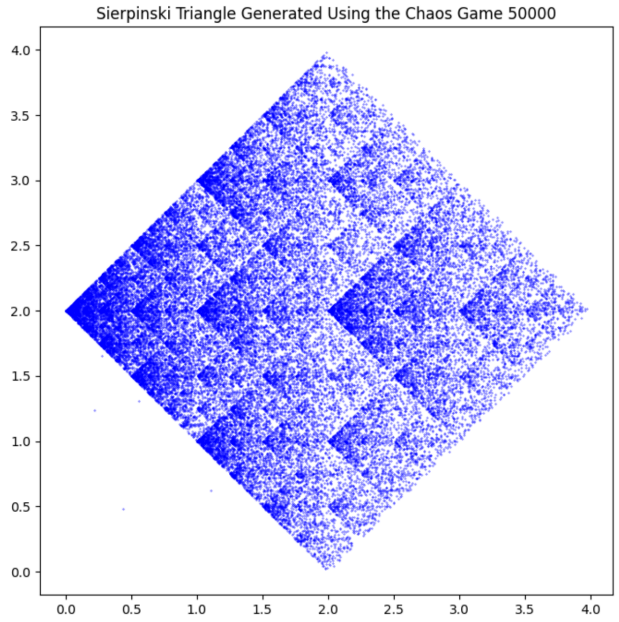


I went a little further with it, keeping the square, but adding midpoints along the way (the original which I forgot to screenshot only had the four corners).

```
vertices = [(0, 0),(2, 0),(4, 0),(4, 2),(4, 4),(2, 4),(0, 4),(0, 2)]
```

Setting the random selection from 0 - 7 created the image on the left, while setting the random selection to 0-4 and then selecting the vertex by multiplying it by 2 and subtracting 1 made the right.





Next is the Barnsley Fern. Like the Chaos Game, it is created by several points. The first step is defining four functions to describe what can happen to the point. Only

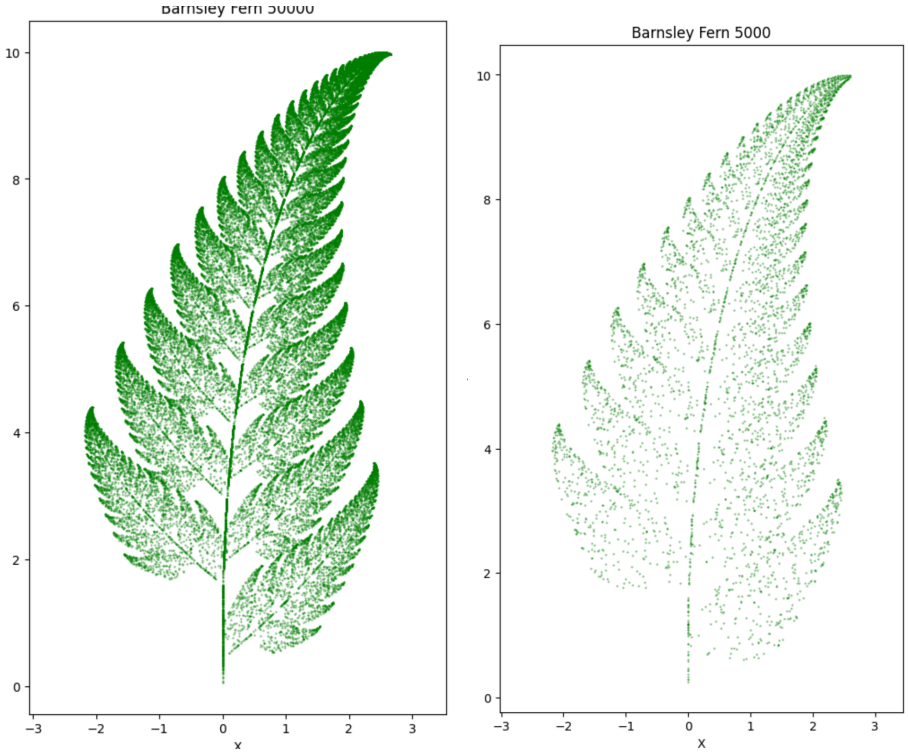
```
# Define the transformation functions
def f1(x, y):
    x = 0
    y = 0.16 * y
    return x, y

def f2(x, y):
    x_new = 0.85 * x + 0.04 * y
    y_new = -0.04 * x + 0.85 * y + 1.6
    return x_new, y_new

def f3(x, y):
    x_new = 0.2 * x - 0.26 * y
    y_new = 0.23 * x + 0.22 * y + 1.6
    return x_new, y_new

def f4(x, y):
    x_new = -0.15 * x + 0.28 * y
    y_new = 0.26 * x + 0.24 * y + 0.44
    return x_new, y_new
```

The first function moves the x-value back to the center and moves up the image slightly. This is the stem piece. The second function creates a new coordinate by moving the x-value a good distance of the current x-value and adds a small amount of the y-value, while the y-coordinate is affected minorly by the previous x-coordinate and then added to a good amount of the y-coordinate, plus 1.6, which would move it up significant and to the right. This is used to create successfully smaller leaflets, and the final two functions define the points for successively large left and right leaves

respectively. Which function is chosen is determined randomly by the defined probabilities:

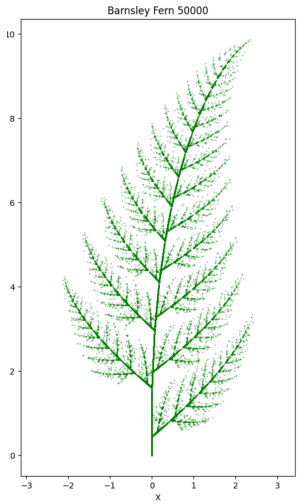
```
# Probabilities for each function
probabilities = [0.01, 0.85, 0.07, 0.07]
```

This allows the majority of the iterations to focus on making smaller and smaller leaves while less frequently moving up along the stem. Like the previous fractal, I have added to the title the number of iterations run to compare depth of field. The only change here is 50000 versus 5000:

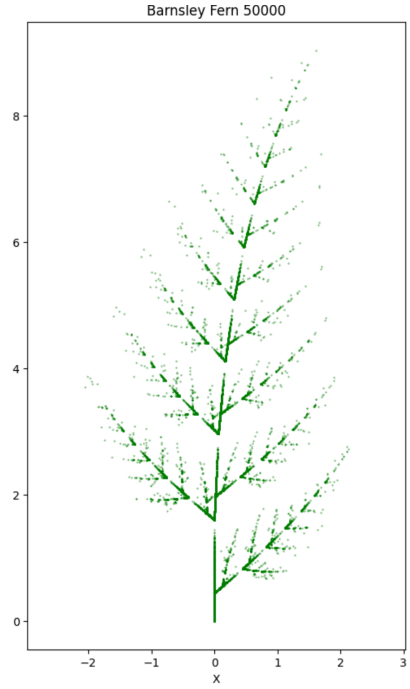


I think the most obvious change is to first play with the probabilities. I would wager increasing the probability of the stem at the cost of the smaller leaves looks like the plants I grow at home, barely hanging on to life.

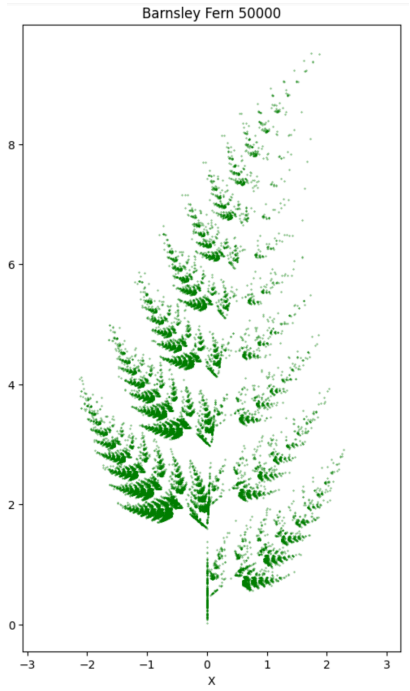
```
probabilities = [0.21, 0.65, 0.07, 0.07]
```



```
# Probabilities for each function
probabilities = [0.41, 0.45, 0.07, 0.07]
```

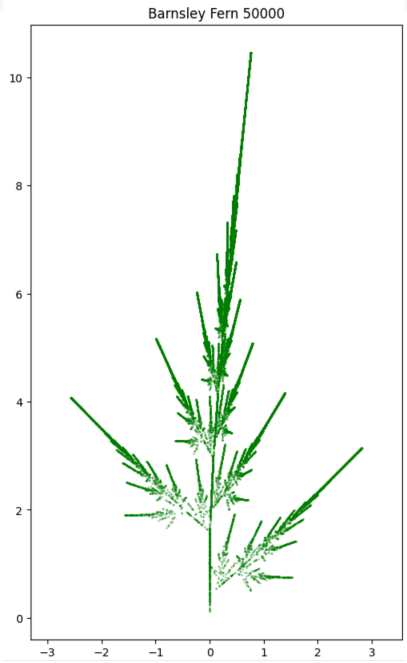



```
# Probabilities for each function
probabilities = [0.01, 0.45, 0.47, 0.07]
```

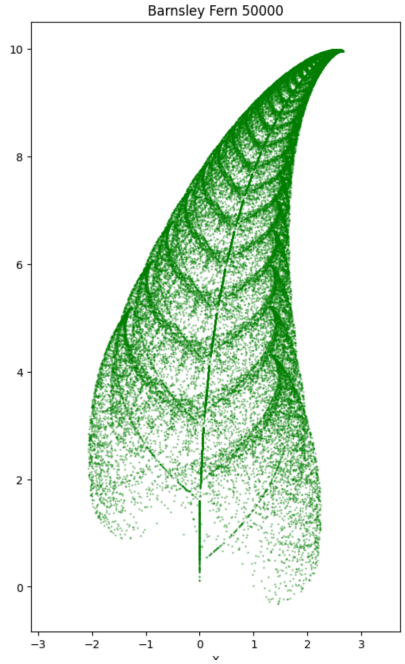


Playing with the leaf values briefly gave me a couple of novel ones, such as the one on the left looking more like a Douglas Fir and something from an episode of Time Tunnel on the right

```
def f2(x, y):
    x_new = 0.45 * x + 0.04 * y
    y_new = -0.04 * x + 0.85 * y + 1.6
    return x_new, y_new
```




```
def f3(x, y):
    x_new = 0.5 * x - 0.26 * y
    y_new = 0.53 * x + 0.22 * y + 1.6
    return x_new, y_new

def f4(x, y):
    x_new = -0.55 * x + 0.28 * y
    y_new = 0.56 * x + 0.24 * y + 0.44
    return x_new, y_new
```



Part 2

We are now moving away from plotting points and bringing Turtle back to do the drawing based on the code. In addition, there is no more random plotting so each run will give identical results. Start off by defining several commands that can be given to the turtle.

```
def draw_lsystem(turtle_string, length, angle):
    stack = []
    for command in turtle_string:
        if command == 'F':
            forward(length)
        elif command == '+':
            left(angle)
        elif command == '-':
            right(angle)
        elif command == '[':
            position = (getx(), gety())
            heading_angle = heading()
            stack.append((position, heading_angle))
        elif command == ']':
            position, heading_angle = stack.pop()
            jump(position[0], position[1])
            face(heading_angle)
```

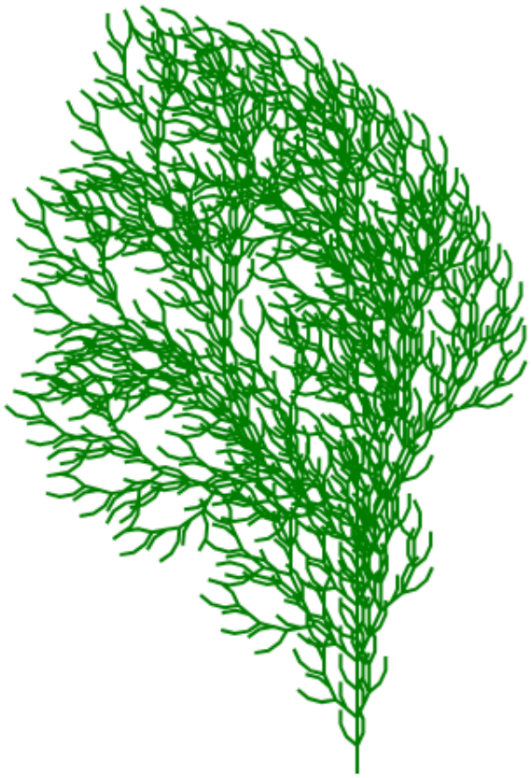
This is so that calling the code as a string will be easy. I do not have it in me, but I like the idea of creating an input command for the user to set up their own L-system. For now I will just copy the rules line and modify it.

```
# Define the L-system rules
rules = {'F': 'FF+[+F-F-F]-[-F+F+F]'}
```

```
iterations = 4
```

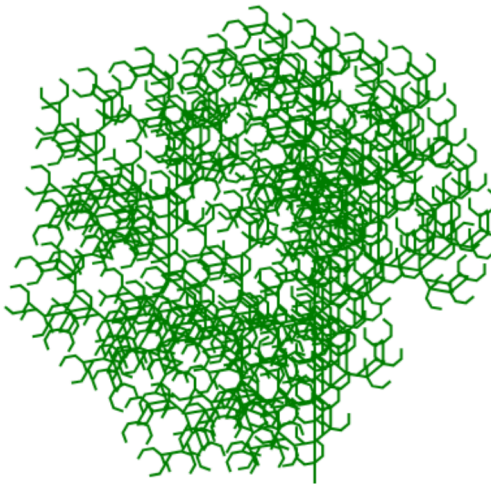
```
length = 8
angle = 25
draw_lsystem(turtle_string, length, angle)
```

The original code creates the following:



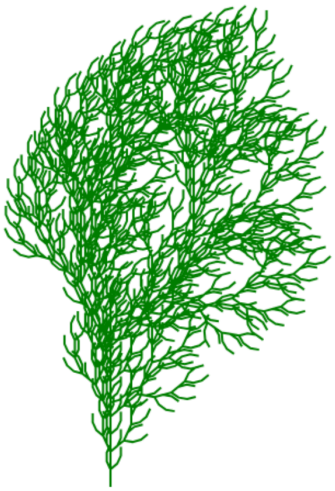
The angle and length give it the tilted ‘moss in the wind’ look. I played with the length and while it makes the fractal taller and slightly more spread out, it is not particularly interesting.

Setting the angle to 50, while also novel, leads to an image that looked like my afro in high school.



This leads back to exploring the L pattern itself and seeing what can be done. Again, because the basic commands within the function, changing those does not do anything of interest, so I will focus on the rules.

Simply reversing the signs of everything so that the lefts become right and vice versa, you get the same image but reflected vertically, like on the left, while the right is when you change only the signs outside the brackets, so to speak:



The rules changes otherwise did not provide me with much of interest, most come out looking like a green egg dropped from a roof, with this being my final attempt:

Playing briefly with the iteration count, while theoretically there is no upper bound, setting it to 7 had the code run for 54 seconds before completing. I quickly reverted it back to 4!

```
rules = {'F': 'FF+[+F-F-F]-[-F+F]+FF+[-F+F+F]-[+F-F]'}
```

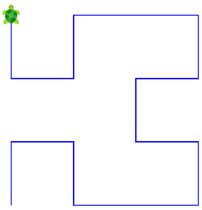
Part 3

Similar to the Hilbert curves, this code is defined by conditionals placed within a subroutine and then the rules defined outside of it, calling the routine one character in the string at a time. One minor note is that you set it so that invalid characters are ignored and I assume are in place to keep track of where the pointer is moving as it travels around the screen. The only commands obeyed are forward, left and right by the given lengths and angle respectively.

Looking first at the iterations to see how it expands each time, when the count is 1, and the axiom is ‘L’, the turtle turns to the right (it was originally facing up), goes forward the length, then turns left and forward, left again and forward before finally facing right, all relative to his prior facing positions.

Setting the iteration at 2 throws me for a loop. It appears that the turtle executes the ‘right’ set of rules first, but at some point, twice specifically, it goes forward twice. I do not see in the code how this can happen.

```
rules = {
    'L': '-RF+LFL+FR-',
    'R': '+LF-RFR-FL+'
```

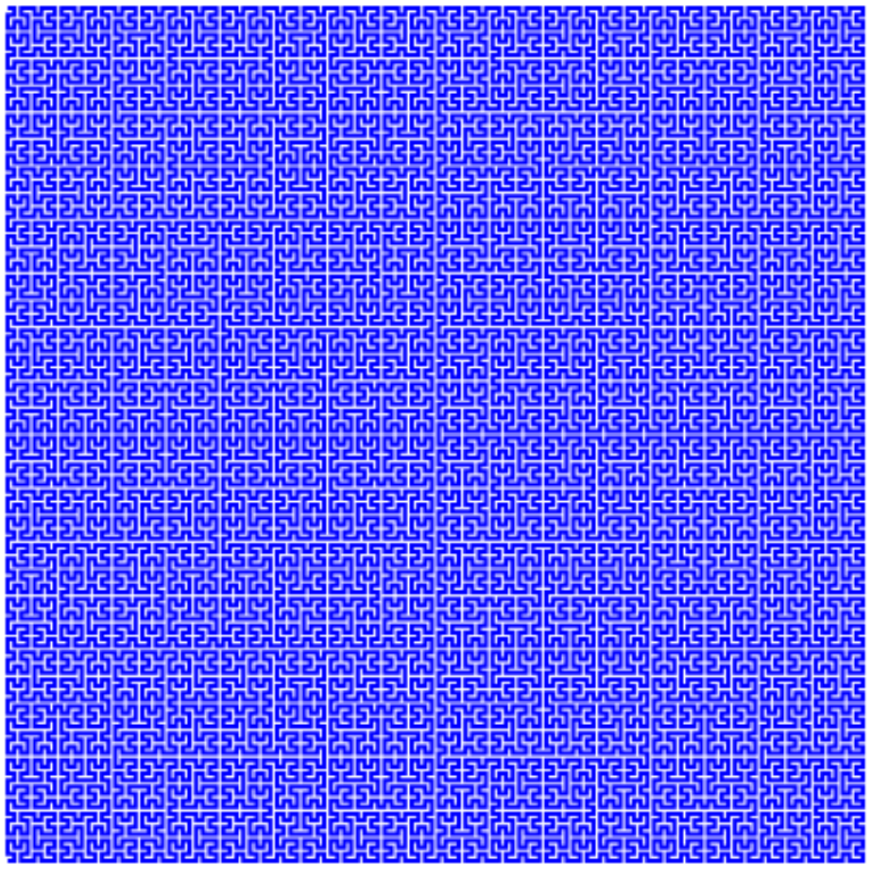


With the final turn at the end, we again turn in the same direction, basically turning 180 degrees before moving forward on top of itself as seen above.

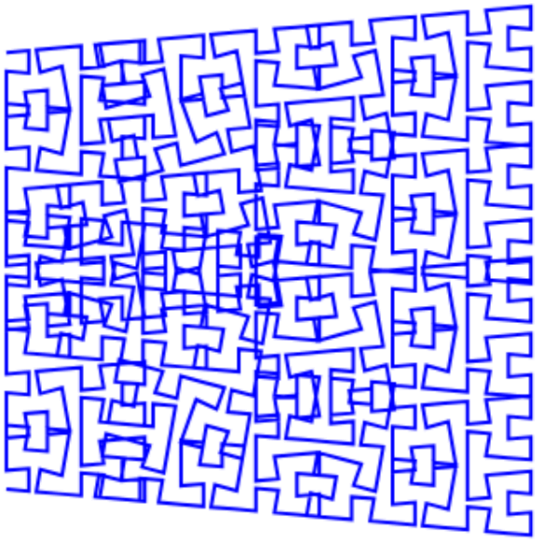
The size of the movements makes sense and does not seem worth adjusting by itself. It simply means that as the iterations increase, the amount of movement must decrease exponentially so that the image does not get too large for the screen.

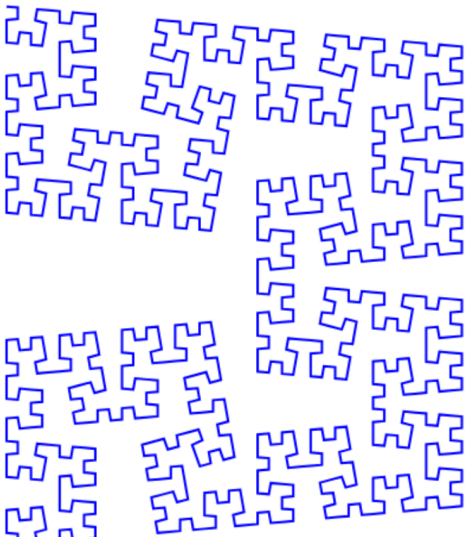
```
size = 400  # Total size of the Hilbert curve (adjust as needed)
n = 2 ** iterations - 1
length = size / n
```

7 iterations is the upper maximum at least on my monitor to get anything recognizable. Another iteration creates a solid blue block.



Resetting the iterations back to 5, but changing the angle slightly to 95 degrees causes the image to fall into itself. Changing the angle to 85 degrees (and having to adjust the length calculations slightly) has it going the opposite way, slowly bursting away as on the right.





Further changing the angle (and size) gives images not as interesting as the 60 degree angles on the left, the 120 degree angle in the middle and the 131 degree angle on the right.