The Hilbert Curve - Python Representation

Pranav Phadke

March 3, 2022

Abstract

This paper explores the mathematical beauty behind what is known as the Hilbert Curve. Furthermore, we will discuss the logic and math behind a python code that animates a representation of a "Pseudo Hilbert Curve" - more on this later. The Hilbert Curve is a continuous fractal space filling curve of Hausdorff dimension 2. It is formally defined by a function that maps points on a one dimensional line to points in an infinite two-dimensional space. For some clarity, in mathematics a curve represents any line that runs through space, even if it has jagged corners. It is impossible to visually display a concrete Hilbert Curve as the curve itself is infinitely long and essentially has no thickness (it is one dimensional); thus, the python program we will discuss has the ability to display what are known as Pseudo Hilbert Curves. All python code with its documentation can be found here: **https://github.com/pranavphadke1/HilbertCurve**

1 Pseudo Hilbert Curve

Let us first discuss what a Pseudo Hilbert Curve (PHC) is. These can be thought of as curves that fill up a discrete area of space with a known set of points. As seen in Figure 1, we have what is called an order 1 Pseudo Hilbert Curve. In this figure, we essentially have a 2x2 grid of 4 points where each point is connected by a single curve that looks like an upside down U. For simplicity, the points are numbered, and we connect them in order (so start at 0, then draw a line to 1, then to 2, then to 3, etc.). Thus, we have successfully connected all points in a discrete grid of 4 points with a single

curve. Now let us consider Figure 2. In this figure, we have a 4x4 grid of 16 points. We will take the 4x4 grid and split it into 4 quadrants of 2x2 grids. We then connect the 4 points in each of the quadrants in the same manner as done in the order 1 PHC. Then, we will reflect the order 1 PHCs in the bottom left and bottom right quadrants over their bottom left to top right and top left to bottom right diagonals respectfully. Figure 2 has dotted lines in both of those quadrants to help understand the lines of reflection. After the reflections, the result is Figure 3. We can then connect points 3 and 4, 7 and 8, and 11 and 12 to get our resulting single curve that connects all 16 points. A cleaner example of an order 1 and 2 PHC is shown in figures 4 and 5. Through some mental induction, we can realize that for any order n PHC, we can create it by splitting our grid into 4 quadrants, drawing an order $n-1$ PHC in each of the grids, reflecting the bottom left and bottom right order $n-1$ PHCs over their respective diagonals, and then connecting the resulting 4 curves together to form a single curve connecting every point in the grid.

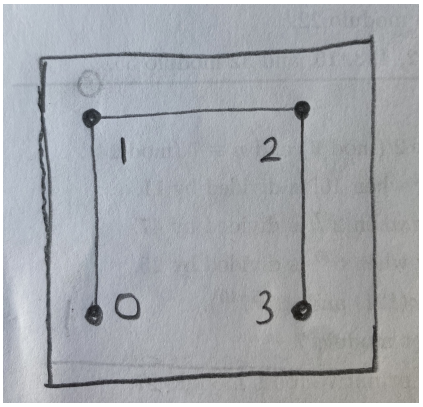

Figure 1: Order 1 Pseudo Hilbert Curve

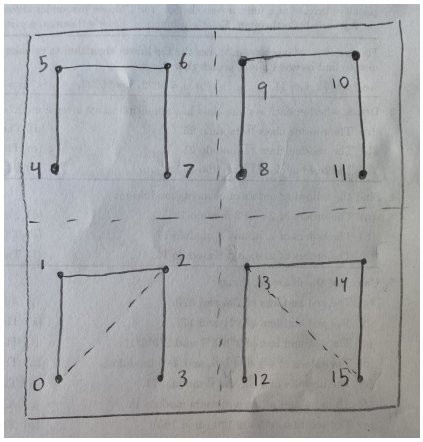

Figure 2: Order 2 PHC (Before Reflection)

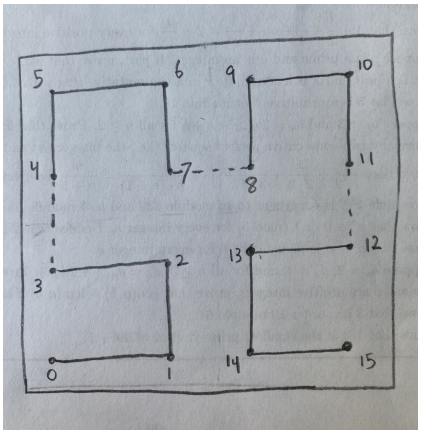

Figure 3: Order 2 PHC (After Reflection)

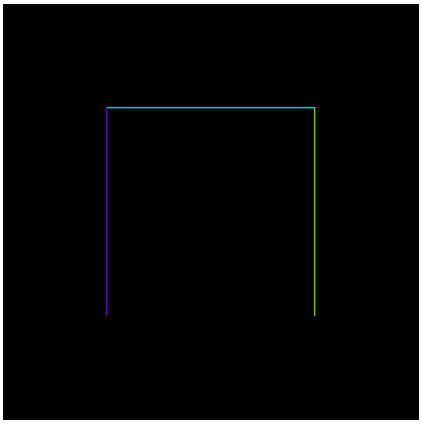

Figure 4: Order 1 PHC

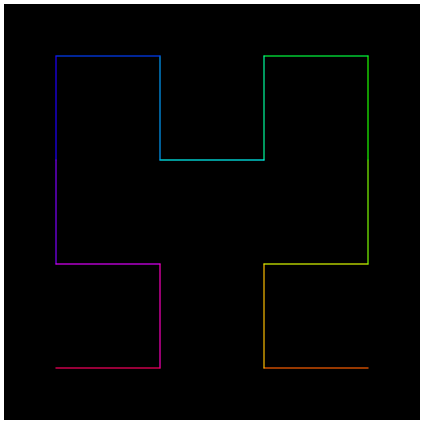

Figure 5: Order 2 PHC

2 History

We just discussed what a Pseudo Hilbert Curve is — a curve that fills a finite space of points. This slightly varies from what a true Hilbert Curve (or any space filling curve) really is. So why did we start thinking about space filling curves? Well the interest in space filling curves began near the end of

the 19th century after the German mathematician Georg Cantor shocked the world with his discovery of set theory and the idea of infinite numbers. After these discoveries, mathematicians became interested in the possibility of a mapping between a one dimensional line and a two-dimensional space in such a way that the line goes through every single point in space. This is different from a line going through a finite grid of pixels, as previously discussed with Pseudo Hilbert Curves. We are talking about an infinite space of points. The goal was to come up with some kind of line (curve) that is as thin as possible yet passes through every single point in an infinite space of points. For a long time, many believed this to not be possible until Italian mathematician Giuseppe Peano discovered the first space filling curve, known as the Peano Curve (Figure 6), in 1890. Shortly after (1891), German mathematician David Hilbert discovered his own space filling curve, the Hilbert Curve. It was a slightly simpler space filling curve than the Peano Curve. Both of these curves were represented by filling a finite set of space; however, as the order of both of these curves reached infinity, the single curve maps to every point in an infinite space of points.

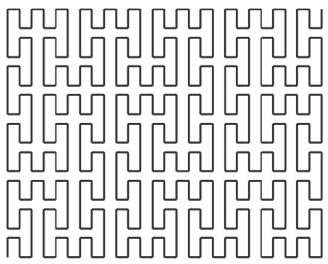

Figure 6: Peano Curve

3 Application - Seeing With Your Ears

The key idea behind the Hilbert Curve, and other space filling curves in general, is that they create a mapping from one dimension to two dimensions. This has some great theoretical application. Let us imagine some software that uses a camera to take in data and somehow transform it into sound in a meaningful way. What does this mean? Let's imagine taking an image of low resolution, say 256x256 pixels, and having every single pixel in the image

be represented by some frequency. Then, brighter pixels in the image would have louder frequencies and darker pixels would have quieter frequencies. If our brains managed to somehow make sense of all these frequencies layered upon one another, we would theoretically be able to process what an image looks like through our ears. Let's set aside if this would actually work and focus on the problem. Pixels live on a two-dimensional space while frequency space is one dimensional. We need to consider some function from pixel space to frequency space that gives our software the best chance at working. One key intuition in such a function is that pixels near each other in the pixel space should correspond to frequencies near each other in the frequency space. Thus, whatever function we use should follow this property.

What if we use a snake like curve, as seen in Figure 7, to connect all the points? It would connect all the points effectively. However, a snake like curve does not follow the property of points near each other on the two-dimensional space being near each other on the one dimensional curve. What's the problem with this? Let's imagine our software does work and there is in fact a way for people to see images through their ears. Let's say our old software worked for low resolution images, but we wanted to push out a software update that allows us to interpret higher resolution images. Those higher resolution images would have more pixels and thus a longer snake curve connecting all the pixels. Particularly, there would be many more points for the curve to go through horizontally. This means that as the resolution increases and more and more points are added, the amount of pixels on a horizontal stretch of the snake curve increases and thus the distance between two points vertically in the pixel space has a drastic increase in distance on the one dimensional frequency space. Essentially, as the resolution improves, pixels on the curve will drastically shift positions. This means that for everyone who had trained to understand frequency sound to pixel location relations for the lower resolution software would have to retrain their brain to understand how the frequencies now work for higher resolution images.

Now what if we used a Hilbert Curve instead. Let's first consider the Pseudo Hilbert Curves again. One key fact about the PHCs is that as their order approaches infinity, each point on the curve maps to a single, exact point on the two-dimensional space. We can summarize this with a function. Let $PHC_n(x) = (q, r)$ represent a mapping from point x on the Pseudo Hilbert Curve (of order n) and pair (q, r) be a point on the two-dimensional space. One proven property of sequences of Pseudo Hilbert Curves is that

$PHC_n(x)$ has a limit point (q, r) as n approaches ∞. Therefore, we know that no matter the order of the Pseudo Hilbert Curve, a point on the one dimensional curve will always refer to the same general region on the two-dimensional space. Furthermore, lets note that a real Hilbert Curve is defined as: $HC(x) =$ the limit as n approaches ∞ of $PHC_n(x)$ for any point x on the HC. Thus, as the order of a PHC reaches infinity, it becomes a real Hilbert Curve. This means that if a real Hilbert Curves is used for the software, any software updates that get pushed out result in users fine-tuning their already built intuitions instead of having to relearn how to see with their ears.

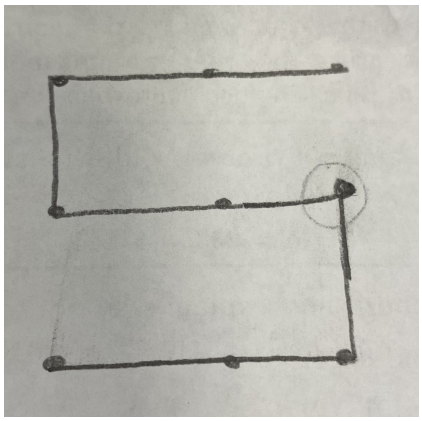

Figure 7: Snake Curve

4 Math behind the Python Code

Now that we have discussed what a Hilbert Curve is, let's talk about some of the math behind the python code. As mentioned in the Abstract section, all the python code is available on my public GitHub repository called "HilbertCurve." The code has plenty of documentation to help one understand how it works. It is written in a IDE/graphical library called Processing. The code is specifically written in Python for Processing, a plugin for Processing that allows you to write in Python while still taking advantage of the predefined functions the Processing language/library provides. As for the math, I took advantage of some basic algebra which is easy to follow through the documentation. I also used the concepts of Bit Masking and Bit Shifting, which are documented in the code as well, but further discussed

below. Additionally, note that the resulting Pseudo Hilbert Curves this program generates is a reflection over the horizontal line of what the typical Pseudo Hilbert Curves look like. This is because the coordinate system I dealt with starts with the point $(0, 0)$ in the top left of the screen. Also, the first pixel has the value of 0. So in a 4x4 grid, the first pixel is 0 and the last is $4 - 1 = 3$.

4.1 Bit Masking

The idea behind bit masking is taking two integers, writing them in their binary representations, and then applying the bitwise "&" operation on each of the individual bits that make up the two binary numbers. In the code, I use this operation to take a pixel value, say 7, and only look at the last two bits of their binary representation to figure out where that pixel would be positioned in an order 1 PHC. This is done because all order PHCs are initially derived from order 1 PHCs. So, by bit masking a pixel's value down to one of the numbers of either 0, 1, 2, or 3, it is easy to figure out where the pixel is relative to the coordinate system. For example, let us apply bit masking to 7 and 3. In the code, this looks like 7&3. Let's rewrite both these numbers in their binary representation: $7 = 0111$ and $3 = 0011$. Then apply the & operation on each of the individual bits: $0\&0 = 0, 1\&0 = 0, 1\&1 = 1, 1\&1 = 1$. Furthermore, since we are using a bit mask with the number 3, we really only have to consider the last two bits of the other number as the resulting number will have a bunch of 0s up until the last two bits.

4.2 Bit Shifting

The idea behind bit shifting is taking an integer, rewriting it in binary, and then either dropping the last n bits or the first n bits. In my code, I did a combination of a forward (rightwards) bit shift of 2 and then a bit masking with 3. This was done to figure out what quadrant a pixel's value belongs to. This is necessary to understand where a pixel is relative to the coordinate system. So if we take the number $4 = 0100$, bit shift it right by 2 to get the binary number 0000, and then bit mask it with 3, we get the binary number 00 which is the same as the integer 0. This means the number 4 belongs to quadrant 0.

5 Resulting Animation

Below are some videos of the animation working on an order 5 PHC and an order 7 PHC:

Order 5: Click Here

Order 7: Click Here

6 Order Images

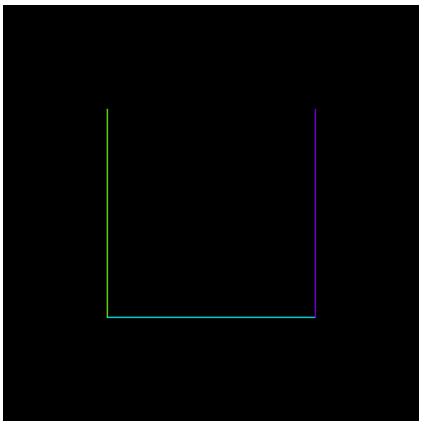

Figure 8: Order 1 PHC

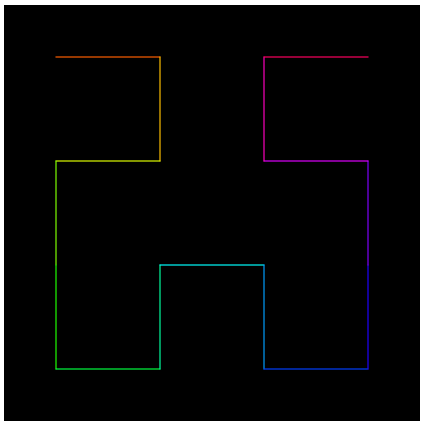

Figure 9: Order 2 PHC

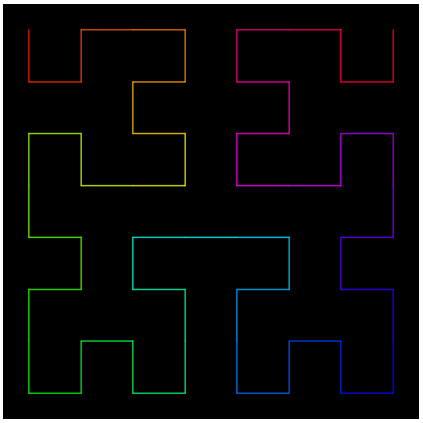

Figure 10: Order 3 PHC

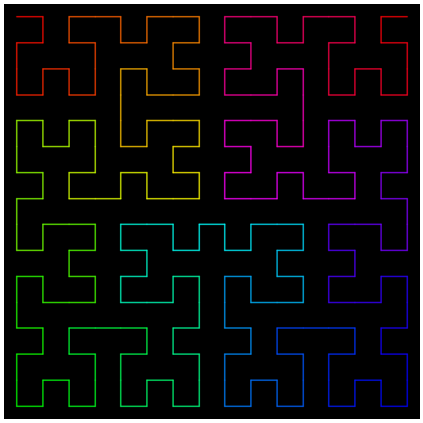

Figure 11: Order 4 PHC

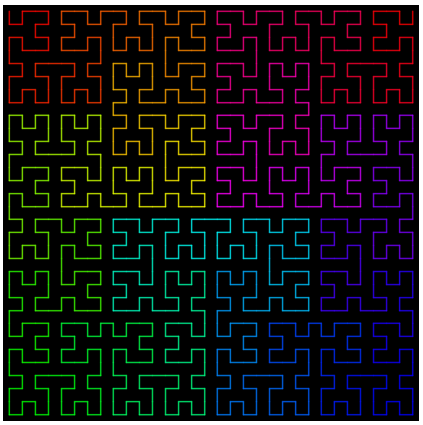

Figure 12: Order 5 PHC

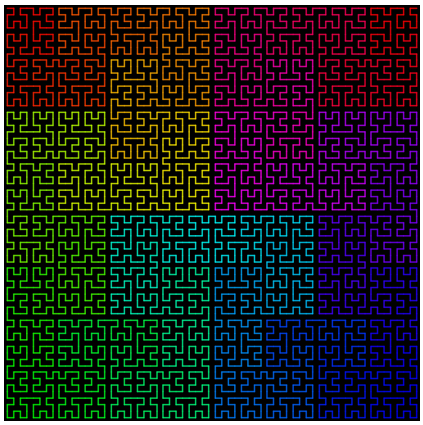

Figure 13: Order 6 PHC

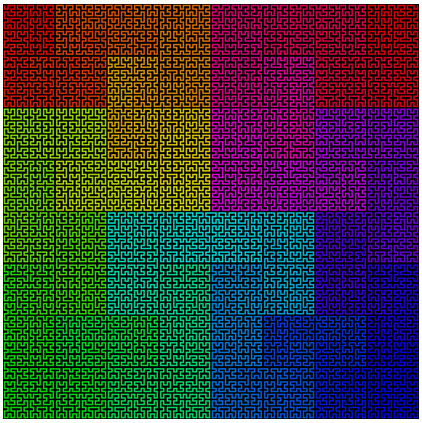

Figure 14: Order 7 PHC

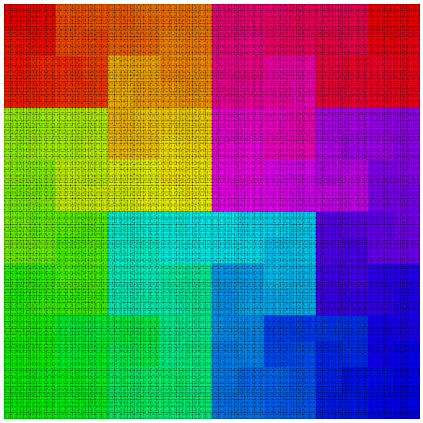

Figure 15: Order 8 PHC

7 References

1. https://github.com/pranavphadke1/HilbertCurve
2. TheCodingTrain YouTube Channel
3. http://blog.marcinchwedczuk.pl/iterative-algorithm-for-drawing-hilbert-curve
4. 3Blue1Brown YouTube Channel