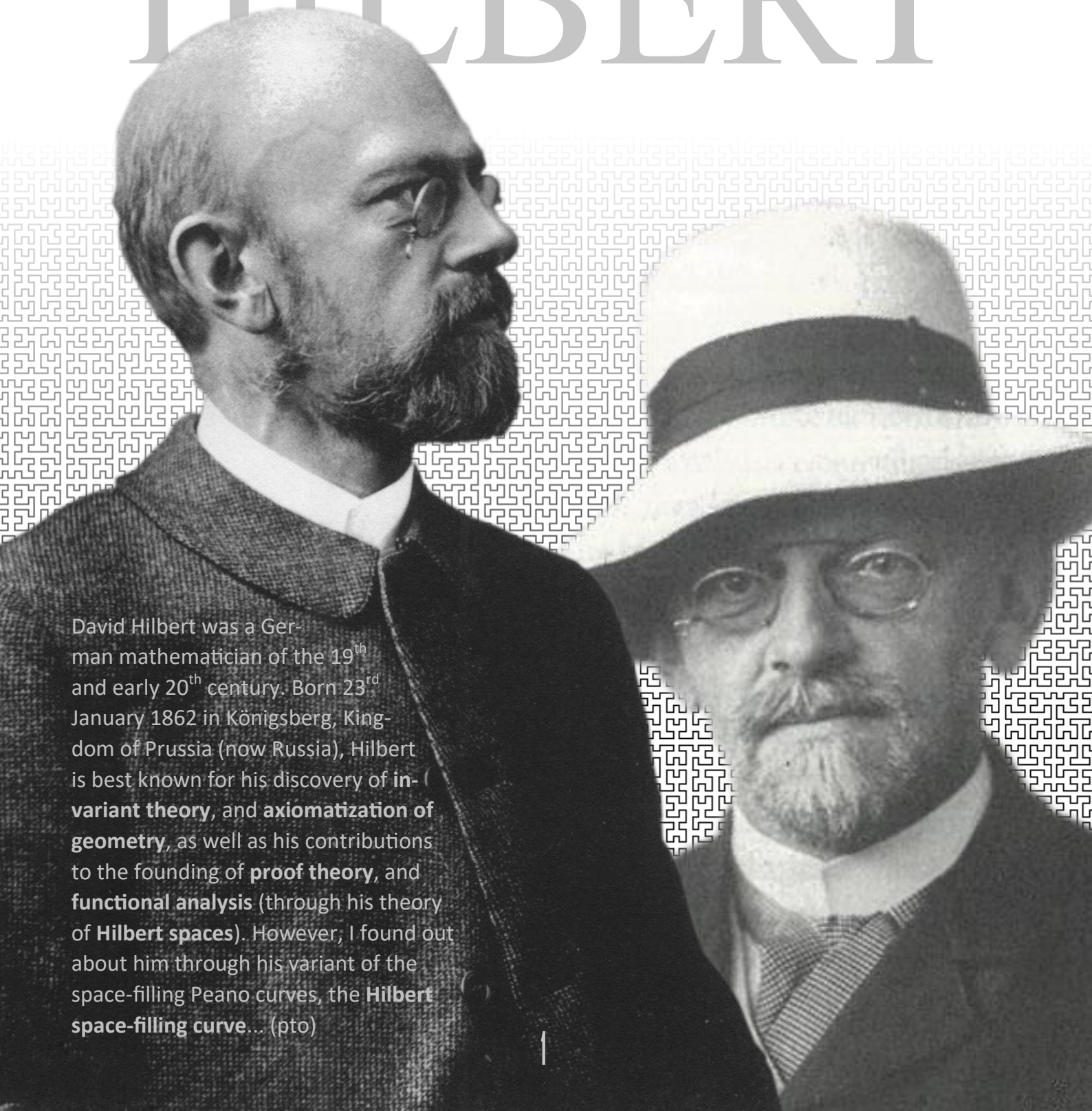
Tom Wright

DAVID HILBERT



David Hilbert was a German mathematician of the 19th and early 20th century. Born 23rd January 1862 in Königsberg, Kingdom of Prussia (now Russia), Hilbert is best known for his discovery of **invariant theory**, and **axiomatization of geometry**, as well as his contributions to the founding of **proof theory**, and **functional analysis** (through his theory of **Hilbert spaces**). However, I found out about him through his variant of the space-filling Peano curves, the **Hilbert space-filling curve**... (pto)

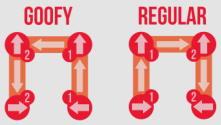
Hilbert curve iteration algorithm

Source: WIKIPEDIA The Free Encyclopedia

A space-filling curve is a curve which has a range containing the entirety of the n-dimensional unit hypercube (eg. Square in 2d, cube in 3d). Most well-known space-filling curves are iterative, meaning they near being space-filling as their iterations (and therefore length) extend to infinity. Hilbert first described his variant of Giuseppe Peano's **Peano curves** (discovered a year earlier in 1890) in 1891. Its length in 2 dimensions can be described as $H_n = 2^n\text{-}2^{-n}$ (where H_n=*the Euclidean length of the curve to the nth approximation i.e. the number of iterations*), meaning the length extends to infinity within a square of finite area.

The way I think about the Hilbert curve, is by starting with one key form, represented by an ordered array of vertices and sides, each with a rotation: the sides and their rotations are used mostly for visualisation, as they instruct the operator on where the curve should go, whereas the vertices are used entirely for the generation of the next instance of the curve- as during each iteration every vertex is replaced by the original key form. Sides however, are never removed from the array.

The algorithm works in the same way; however, the computer requires two key forms, each on the vertical reflection of the other (I call them goofy and regular):

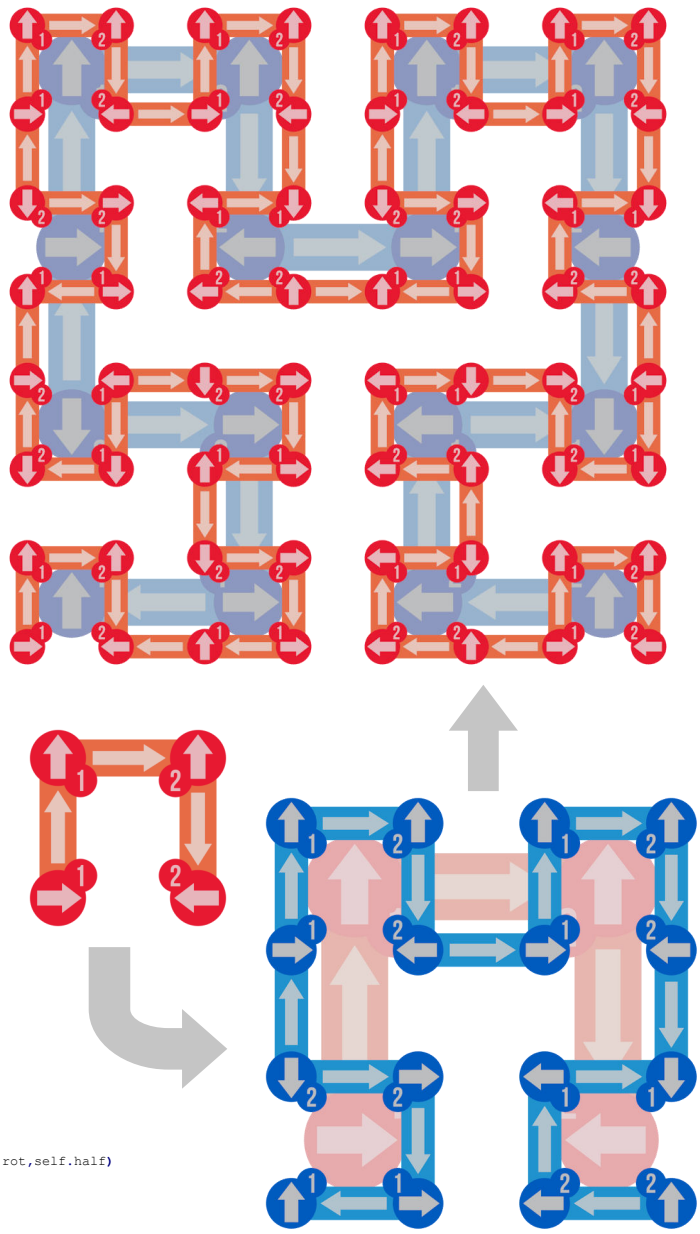


To a human they appear the same because humans are able to adapt when drawing but to the computer, the direction in which key component is drawn matters. After some experimentation, I discovered that to work out whether a newly inserted key form should be goofy or regular, you have to compare the rotations of the vertex being replaced and the adjacent side, as this will show the direction in which the key form will be drawn out. If $\mathbf{R_{side}\text{-}R_{vertex}=90}$ then the key form will be drawn left to right and will therefore be regular. However, if not, and $\mathbf{R_{side}\text{-}R_{vertex}=\text{-}90}$ the key form will be drawn right to left and will therefore be goofy. However, because each key form has 4 vertices by only 3 sides, you can't simply go to the next one along to find the adjacent side. The fact is, half the vertices should look forward along the drawing direction (*+d*), and half should look backwards (*-b*). To implement this in the algorithm, I labelled each vertex as 1 or 2 respectively. This also makes the difference between goofy and regular key forms clearer (see above).

It's worth noting that each side will always have the same length, length *x*, as fraction of the encompassing square. Through experimenting with curves when the algorithm began to function properly, I found that for a curve of order *n*, the curve will have $\mathbf{2^n\text{-}1}$ sides- and will therefore all be Mersenne numbers (1,3,7,15…). Therefore:

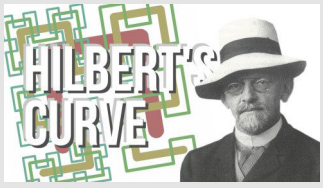
x = k/(2^n-1) **, (where *k=square side length*)**

Ultimately, the algorithm worked, and I used it to generate the pattern on page 1. I have included the main algorithm in python code below, but I have also put the full source code on GitHub, as well as uploading a video visualising each iteration (see links below).



GET THE

https://github.com/tomnwright/HilbertsSpaceFillingCurveAlgorithm



Hilbert's Space Filling Curve Iterations 1-8

https://youtu.be/LCmP_Qqgeow

```
class vert:
    "Hilbert curve vertex"
    def __init__(self,rotation,half):
        self.rot = rotation%360
        self.half=half
    def __str__(self):
        return "v.{}.{}".format(self.rot,self.half)

class side:
    "Hilbert curve side"
    def __init__(self,rotation):
        self.rot=rotation%360
    def __str__(self):
        return "s.{}".format(self.rot)

class constant:
    "Hilbert curve constant form"
    #universal constant:
    class regular:
        u_constant_reg = [vert(90,1),side(0),vert
(0,1),side(90),vert(0,2),side(180),vert(270,2)]
        def __init__(self,rotation=0):
            self.rot=rotation
            self.form = [vert(90+rotation,1),side
(rotation),vert(rotation,1),side(90+rotation),vert
(rotation,2),side(180+rotation),vert(270+rotation,2)]
        def __str__(self):
            return [str(x) for x in self.form]
    class goofy:
        u_constant_reg = [vert(270,1),side(0),vert
(0,1),side(270),vert(0,2),side(180),vert(90,2)]
        def __init__(self,rotation=0):
            self.rot=rotation
            self.form = [vert(270+rotation,1),side
(rotation),vert(rotation,1),side(270+rotation),vert
(rotation,2),side(180+rotation),vert(90+rotation,2)]
        def __str__(self):
            return [str(x) for x in self.form]


def hilbertGen(n):
    curve=constant.regular(0).form
    for q in range(n-1):
        vertsFound = []
        for i,j in enumerate(curve):
            if type(j) == vert:
                if j.half == 1:
                    vertsFound.append([i,j.rot,(curve
[i+1].rot-j.rot)==90])
                else:
                    vertsFound.append([i,j.rot,(curve[i-
1].rot-j.rot)==90])

        vertsFound.reverse()
        for m in vertsFound:
            if m[2]:
                curve[m[0]+1:m[0]+1] = constant.regular(m
[1]).form
                del curve[m[0]]
            else:
                curve[m[0]+1:m[0]+1] = constant.goofy(m
[1]).form
                del curve[m[0]]
    return curve
```

USES OF THE HILBERT SPACE-

Hilbert curves have a useful property- locality preservation, meaning points on the curve with similar *d* values (*d* being the distance along the curve) have similar (x,y) values (although not necessarily the other way round). For this reason, Hilbert curves can be used to map IP addresses, as addresses which are near to each other in value can be represented as being near to each other in position on the curve. Also, because the curve includes all values of the unit square, it can be used to apply threshold filters to images (whereby data from each pixel is transferred to the next along the curve for use by the effect algorithm) to ensure that the effect does not create a noticeable pattern, as might be the case if pixels are selected by rows.