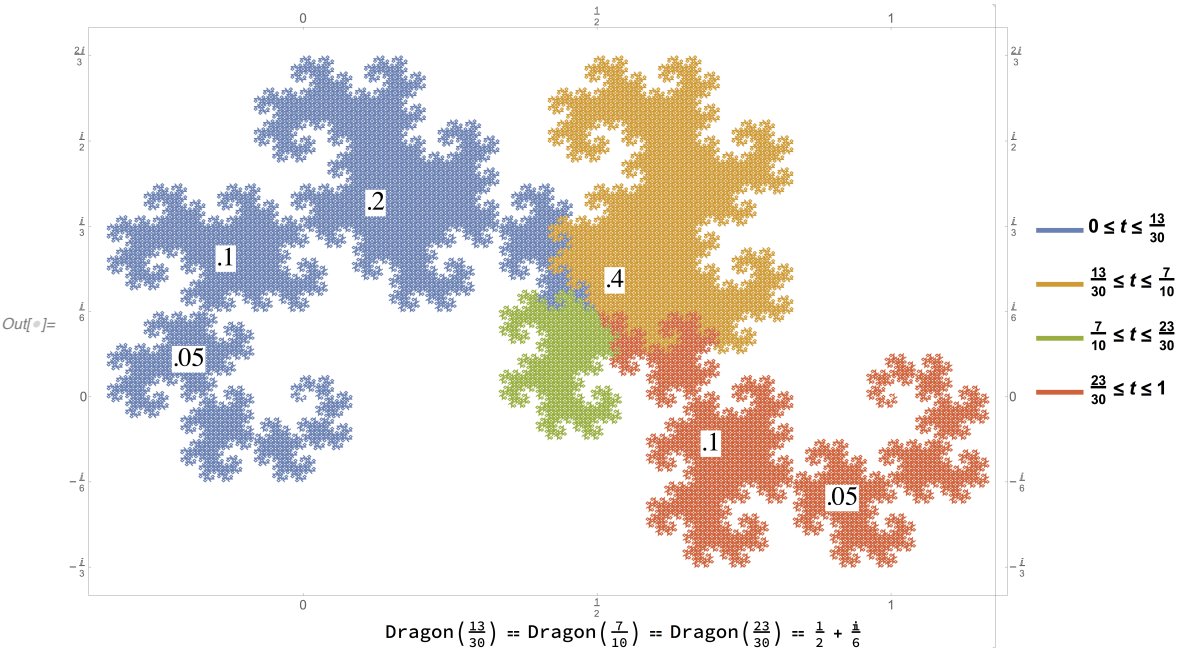
The Dragon Function is way cooler than the “Dragon Curve”

The graph of the truly spacefilling Dragon Function is the Dragon “Curve”:

In[]:=
```
Import@"/Users/billgosper/Math/dragtrip!.png"
```

Out[]=



As 0 ≤ t ≤ 1, Dragon Function(t) varies from 0 + 0*i* to 1 + 0*i* in the Complex Plane.

The texture is an artifact of finite sampling. But notice that it’s not the usual square grid texture you get from the traditional binary recursion!
Julian’s miraculous piecewiserecursivefractal completely defines the Dragon Function (named “dragun”) in one line:

```
dragun[t_] := piecewiserecursivefractal[t, Identity,
  Piecewise[{{{1}, 0 ≤ # ≤ 1/2}, {{2}, 1/2 ≤ # ≤ 1}}, {}] &,
  {2 * # &, 2 * (1 - #) &}, {(1 + I) * #/2 &, (I - 1) * #/2 + 1 &}]
```

E.g.,

In[6]:=
```
dragun[13/30]   (*Not a dyadic rational!*)
```

Out[6]=
$$\left\{\frac{1}{2}+\frac{i}{6}\right\}$$

It works for any rational in [0,1], not just dyadic rationals. And since spacefilling functions are continuous, it works for any real. In principal , traditional implementations of the dragon function, which are restricted to dyadic rational preimages, suffice to define `dragun[t]` for real 0≤t≤1. But painfully. E.g., to get `dragun[13/30]`, you’d need to take a limit n ■∞ of `dragun[ ⌊13 2^n/30⌋ / 2^n]`.

But piecewiserecursivefractal can also define the inverse of dragun!

```
undrag[z_] := piecewiserecursivefractal[z, Identity,
  If[-(1/3) ≤ Re[#] ≤ 7/6 && -(1/3) ≤ Im[#] ≤ 2/3, {1, 2}, {}] &,
  {# * (1 - I) &, (1 - #) * (1 + I) &}, {#/2 &, 1 - #/2 &}]
```

Thus the preimage of $\frac{1}{2} + \frac{i}{6}$ (near the label ".4") is

In[8]:=
```
undrag[1/2 + i/6]
```

Out[8]=
$$\left\{\frac{13}{30}, \frac{7}{10}, \frac{23}{30}\right\}$$

so $\frac{1}{2} + \frac{i}{6}$ is a triple point! Actually, triple points are no big deal. Any patch (with positive area) of a "spacefilling curve" contains infinitely many triple points. I.e., it is dense with them.
This is because spacefilling functions map closed intervals onto closed sets.

The graph is `dragun` applied to the four closed subintervals

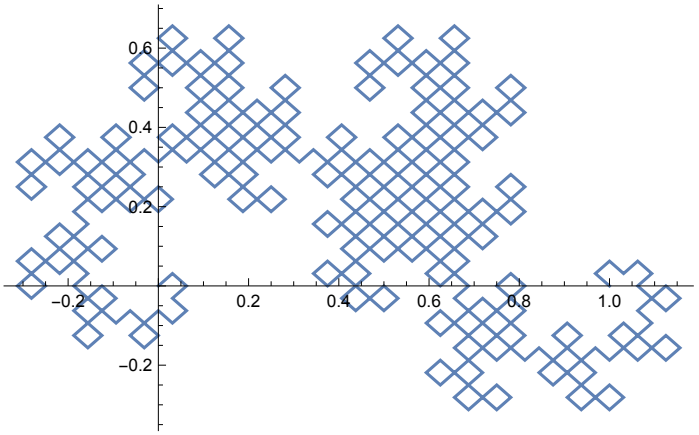
In[12]:=
```
"" @@ # & /@ {{0, 13/30}, {13/30, 7/10}, {7/10, 23/30}, {23/30, 1}}
```

Out[12]=
$$\left\{\left[0, \frac{13}{30}\right], \left[\frac{13}{30}, \frac{7}{10}\right], \left[\frac{7}{10}, \frac{23}{30}\right], \left[\frac{23}{30}, 1\right]\right\}$$

comprising the unit interval [0,1], the Dragon Function's entire preimage. Each colored patch is the image under Dragon of a closed interval, and is thus a closed set. Thus the (interior) boundaries are all double points (at least), belonging to both of the two patches that share it. In fact, a **dense** subset of any such boundary is necessarily **triple** points like $\frac{1}{2} + \frac{i}{6}$! "Dense" means that between any two (running along the boundary), there's another. The entire image is dense with triple points because you can apply the Dragon Function to the unit interval divided into arbitrarily many closed subintervals whose images will tile the whole figure. There are infinitely many triple points in any patch with positive area. Spacefilling functions can't help but to "overspacefill".
This makes particularly ironic the recreational (and oxymoronic) pursuit of "self-avoiding spacefilling curves". What they probably mean are self-avoiding polygonal ("connect the dots") samplings of spacefilling functions. These polygons are helpful in showing where the function is actually going. You can't see this by connecting consecutive images of dyadic rationals (which are the only exact values available without Julian's magic). You just get the familiar square grid texture:

In[15]:=
```
ListLinePlot[ReIm[First /@ dragun /@ Range[0, 1, 1/2^9]]]
```

Out[15]=

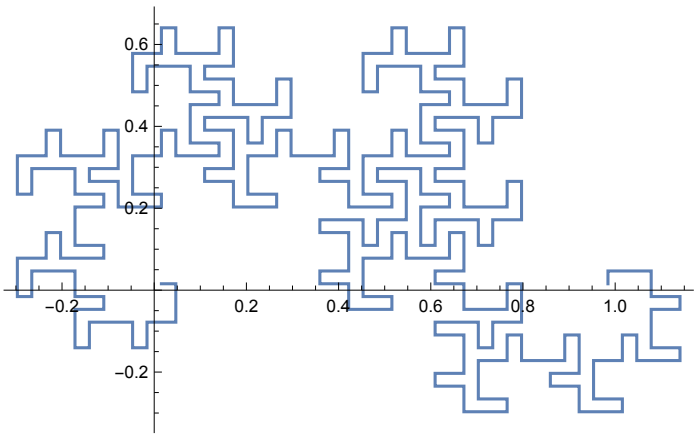


The vertices are legitimate values of the function, but the line segments are meaningless because we can't see their sequence.

Mandelbrot's answer to this was the "median curve" formed by joining the midpoints of consecutive segments:

In[16]:=
```
ListLinePlot[
 Mean[{Drop[#, 1], Drop[#, -1]}] &@ReIm[First /@ dragun /@ Range[0, 1, 1/2^9]]]
```

Out[16]=



But are these artificial points in proper sequence?

Julian confirms that they are, with the remarkable identity

In[17]:=
```
Inactive[dragun][(m + (-1)^m/4 + 1/2)/2^n] ==
 Mean[{Inactive[dragun][m/2^n] + Inactive[dragun][(m + 1)/2^n]}]
```

Out[17]=

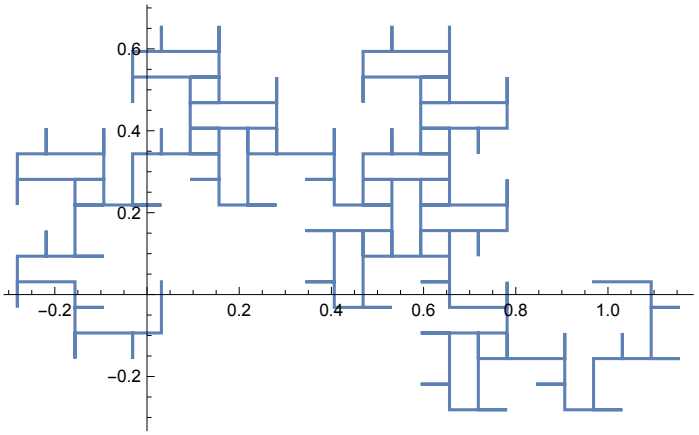
$$\text{dragun}\left[\left(m+\frac{(-1)^m}{4}+\frac{1}{2}\right)2^{-n}\right]=\text{dragun}[(m+1)\,2^{-n}]+\text{dragun}[m\,2^{-n}]$$

In other words, the midpoints exactly coincide with a slightly dithered regular sampling.

It does not work to simply connect every other point:

In[18]:=
```
ListLinePlot[ReIm[First /@ dragun /@ Range[2^-9, 1, 2^-8]]]
```

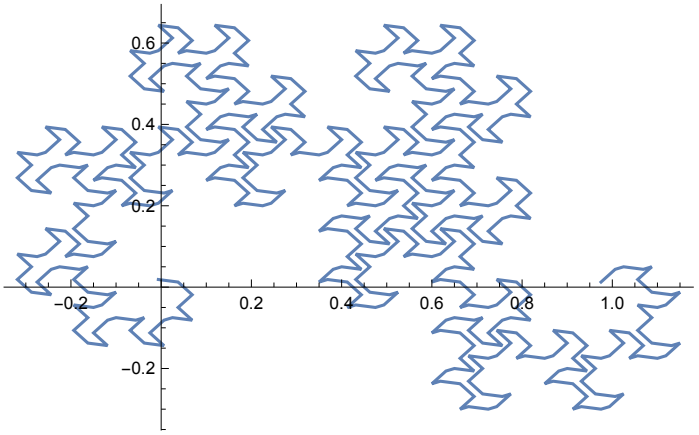
Out[18]=



But we can self-avoid by connecting every third point:

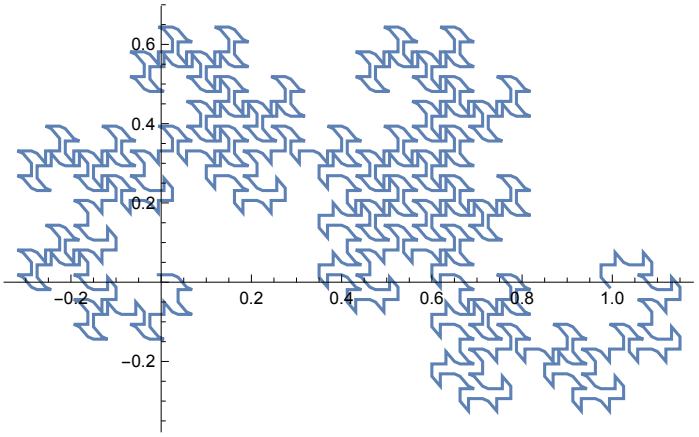
In[19]:=
```
ListLinePlot[ReIm[First /@ dragun /@ Range[2^-9 / 3, 1, 2^-9]]]
```

Out[19]=



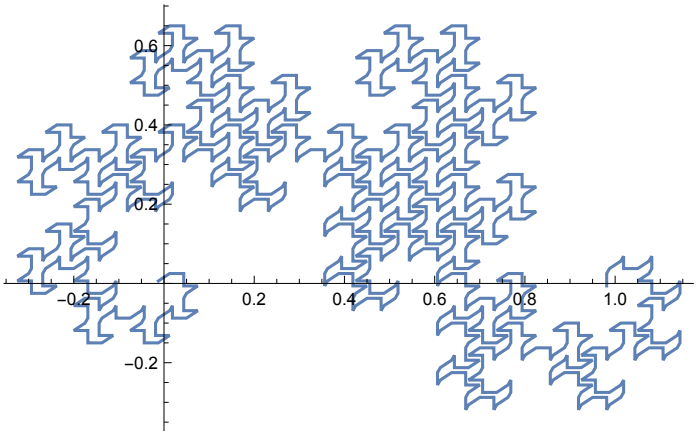
In[20]:=
```
ListLinePlot[ReIm[First /@ dragun /@ Range[2^-9 / 3, 1, 2^-10]]]
```

Out[20]=



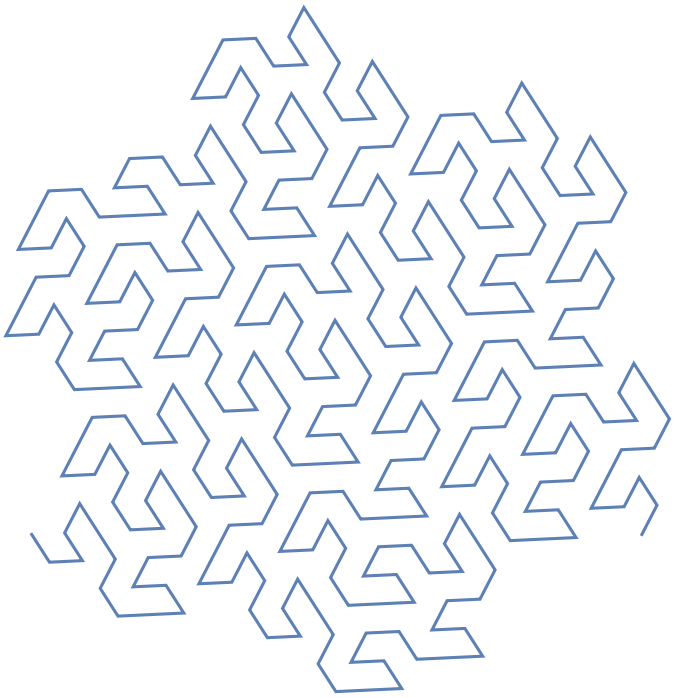
In[22]:=
```
ListLinePlot[ReIm[First /@ dragun /@ Range[2^-9 / 6, 1, 2^-10]]]
```

Out[22]=



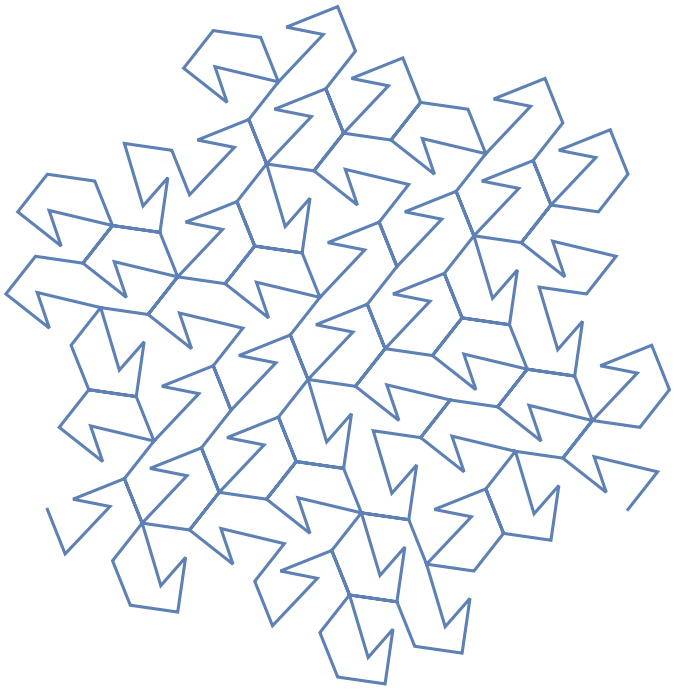
It was only recently that I understood the surprising popularity of the six-around-one spacefill that I regrettably named Flowsnake.

```
ListLinePlot[{Re[#], Im[#]} & /@ (FlowS /@ Range[0, 1, 1 / 49 / 7]),
 Axes → False, AspectRatio → Automatic]
```



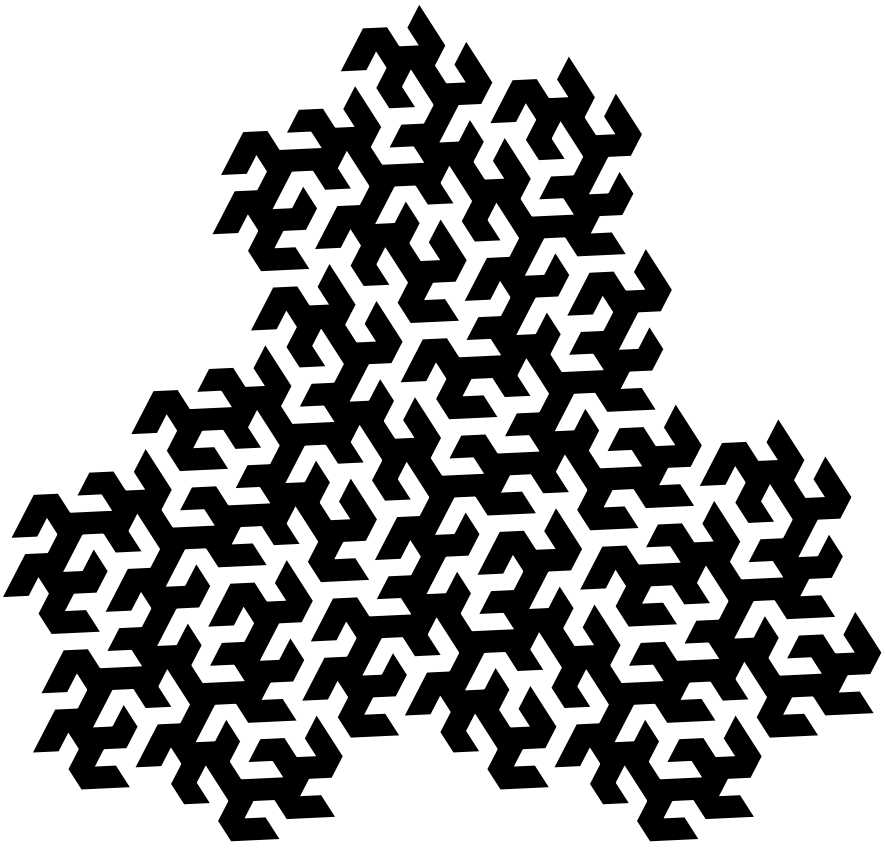
People like it simply because the canonical sampling of it self-avoids! Merely changing a 7 to a 6 in the sampling spec:

```
ListLinePlot[{Re[#], Im[#]} & /@ (FlowS /@ Range[0, 1, 1 / 49 / 6]),
 Axes → False, AspectRatio → Automatic]
```

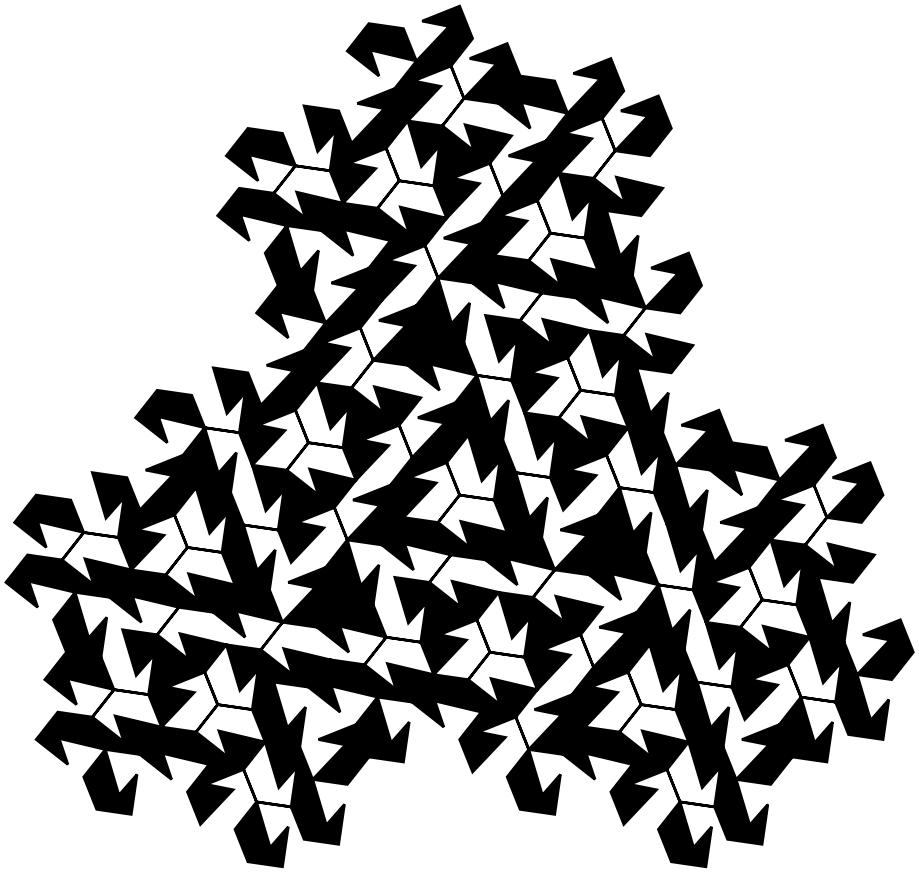


These are both the same “curve”! Connecting three in a loop closes a polygon:

```
trifl[#, #, #] &@Range[0, 1, 1/49/7]
```

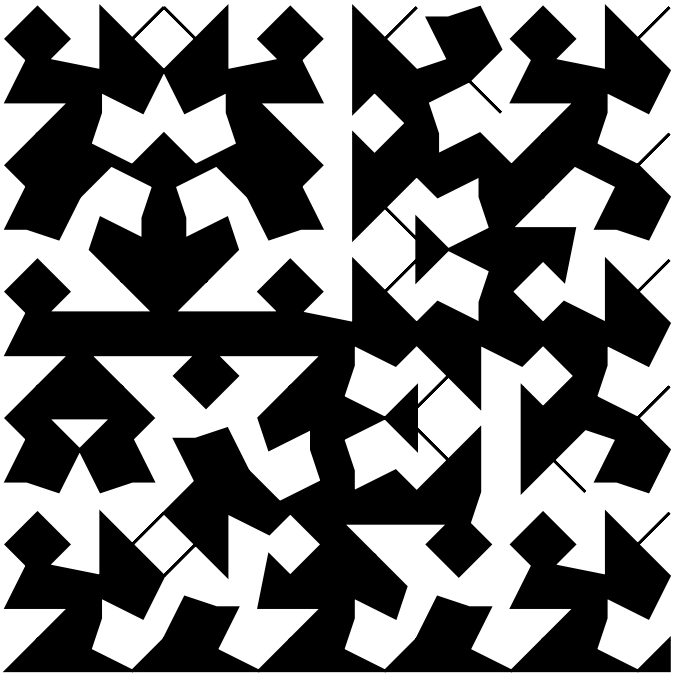


```
trifl[#, #, #] &@Range[0, 1, 1/49/6]
```

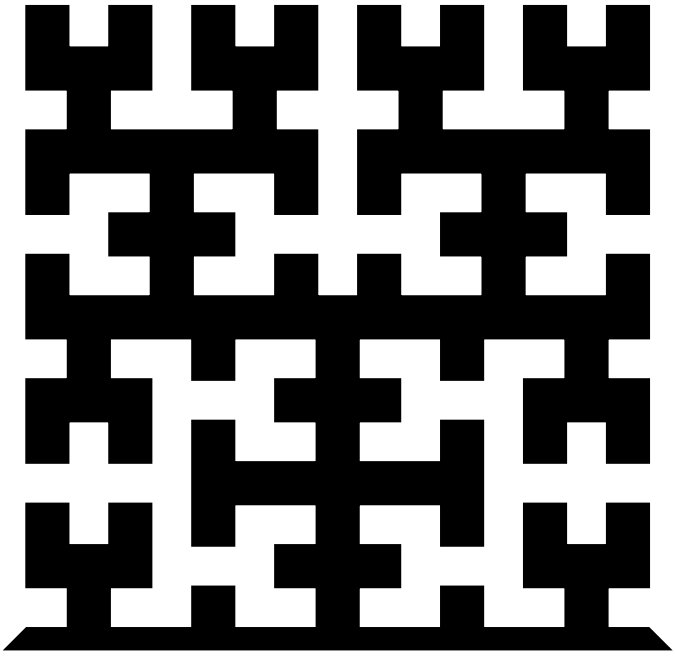


To belabor the point, here are two samplings of the traditional Hilbert curve:

```
Graphics[Polygon[
  ReIm@Append[Table[First@hilbert[5 k / 2048 + 1 / 4096], {k, 0, Floor[2048 / 5]}], 1]]]
```



```
Graphics[
 Polygon[ReIm@Join[{0}, Table[First@hilbert[(k + 1/6) / 256], {k, 0, 255}], {1}]]]
```



I am fond of telling the (very) uninitiated that the upper image is actually the Foo Dynasty ideograph for "My hovercraft is full of eels."

It is finally time to work up the courage to confront the magic function `piecewiserecursivefractal`, which created dragun and undrag (and hilbert and countless other fractals). We might expect it to be big and hairy. Big? No. Hairy? You bet! This is the damnedest, most miraculous piece of code I've ever seen:

```
piecewiserecursivefractal[x_, f_, which_, iters_, fns_] :=
  piecewiserecursivefractal[x, g_, which, iters, fns] =
   ((piecewiserecursivefractal[x, h_, which, iters, fns] :=
      Block[{y}, y /. Solve[f[y] == h[y], y]]);
    Union @@ ((fns[[#]] /@ piecewiserecursivefractal[iters[[#]][x],
          Composition[f, fns[[#]]], which, iters, fns]) & /@ which[x]));
```

It mentions itself in three places, so it's recursive, as you'd expect. But wouldn't you expect a termination condition? It terminates by redefining itself to be nonrecursive when it finds itself in a non-terminating recursion! Whereupon it solves for the fixed point(s) of that infinite recursion.

Julian presenting Minsky recurrence results at G4G9

In[•]:=
```
Import@"/Users/billgosper/Math/JulianG9.jpg"
```

Out[•]=

promptly computes the three preimages 13/30, 7/10, and 23/30. Julian's function converts the "wiggly lines" into an exact definition of the Dragon Function! —Bill Gosper