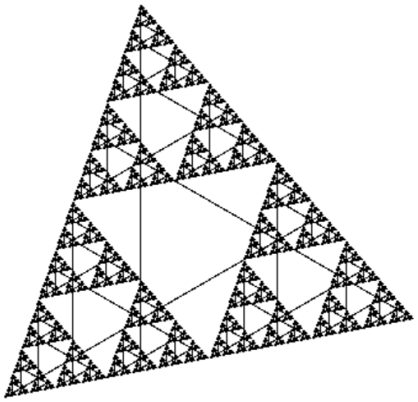
Turtle Python 8 – Iterated Function Systems

In an earlier document "Introducing Recursion", we saw the following program, which includes a "triangle" function that makes a single *recursive* call (i.e. it "calls" itself once), but which is otherwise very similar to the "Recursive triangles" example program (and produces an identical pattern):

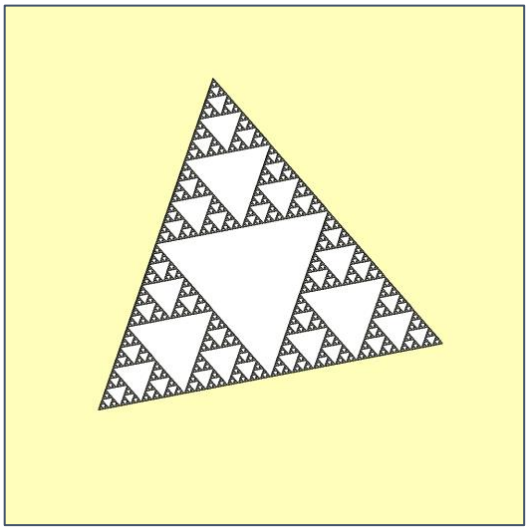
```
def triangle(size):
    if size>1:
        for count in range(3):
            forward(size)
            triangle(size/2) # just one recursive call
            right(120)

movexy(-100,150)
triangle(256)
```



When run, it produces the image pictured here, giving the effect of triangles within triangles within triangles. In brief, this happens because the program first sets out to draw a triangle of *size* 256 (this triangle can be seen in the middle of the pattern, with its corners obscured). But each time the *triangle* function draws any side of a triangle of a given *size*, it will interrupt the drawing of the current triangle and instruct the drawing of a complete new triangle of half that *size*. So around the central triangle of *size* 256, we see 3 triangles of size 128; around those are 9 triangles of size 64; around those are 27 of *size* 32, and so on. Recursion stops when the *size* parameter gets down to 1, and the function then just exits without doing anything.

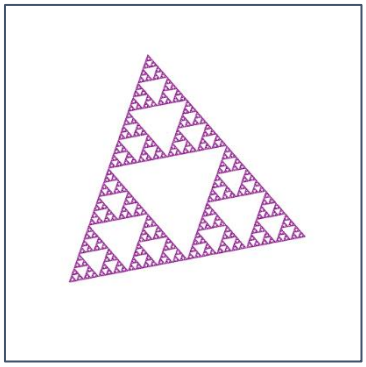
This image exhibits *self-similarity*, in that we see the same sort of pattern repeating at different scales, so that small parts of the image are smaller copies of the entire thing. And it bears a striking resemblance to what is called a *Sierpiński Triangle*, as we shall see.

1. Three Ways of Creating a Sierpiński Triangle



The most standard way of creating a *Sierpiński Triangle* is to take a large equilateral triangle, divide it into four smaller equilateral triangles, "erase" the central one, then repeat these operations on the three other triangles, and continue recursively down the levels. The "Sierpinksi triangle – by deletion" program in the menu "Examples 9 – self-similarity and chaos" works like this, producing the image shown here (which is given a cream background so as to highlight the progressive erasing of the central triangles). As with the previous program, the recursion is made to terminate when the triangle size gets very small. The original triangle is set up with vertices at coordinate points (400, 138), (843,650) and (179,778) – this is done purely to ensure that the pattern matches as closely as possible with the pattern produced by the "Recursive triangles" program above.

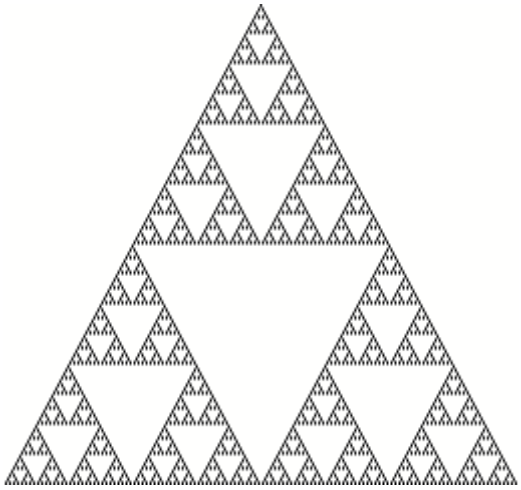
Another more surprising way of generating a Sierpiński triangle is by starting with a random point within the overall triangular area, then repeatedly choosing one of the three main vertices at random, moving halfway towards it, and drawing a dot. Here are the program – called "Sierpinski triangle (by random dots)", again in the "Examples 9" menu – together with the resulting image:



```
x=[400,843,179]
y=[138,650,778]
thisx=randint(400,600)
thisy=randint(400,600)
while 1>0:
    thisc=randrange(3)
    thisx=(thisx+x[thisc])/2
    thisy=(thisy+y[thisc])/2
    pixset(thisx,thisy,purple)
```

This records the coordinates of the triangular area at (400,138), (843,650) and (179,778) – as before set so as to match the three vertices of the output produced by the "Triangles" program. A random initial point is chosen with x- and y-coordinates (*thisx* and *thisy*) between 400 and 600, and then we reach the *while* loop. This first sets *thisc* randomly to 0, 1, or 2, then moves *thisx* and *thisy* (whatever they might currently happen to be) halfway towards the corresponding point of the triangular area – e.g. if *thisc* is equal to 1, then (*thisx*,*thisy*) is moved halfway towards (*x*[1], *y*[1]) which is (843,650). Then a purple pixel is placed at the new point; and the loop repeats. Leave this running, and a Sierpiński triangle gradually emerges!

Another surprising construction uses a one-dimensional cellular automaton of the kind we saw in the previous document on "Cellular Automata", seeding Rule 90 with a single "live" cell at the top:

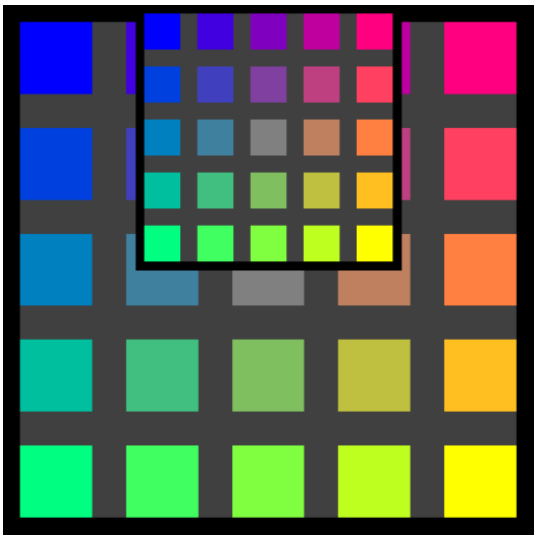


```
width=257
height=128
cellcol=[0xFFFFFE,0x000001]
nextstate=[0]*8

def setup(rulecode):
    for nhood in range(8):
        nextstate[nhood]=rulecode%2
        rulecode=rulecode//2

def nextgen(g):
    for x in range(-1,width+1):
        xmod=(x+width)%width
        thispix=pixcol(xmod,g-1)&1
        n3=n2*2+thispix
        n2=n1*2+thispix
        n1=thispix
        if x>0:
            pixset(x-1,g,cellcol[nextstate[n3]])

canvas(0,0,width,height)
resolution(width,height)
setup(90)
blank(0xFFFFFE)
pixset(width//2,0,cellcol[1])
for generation in range(1,height):
    nextgen(generation)
```

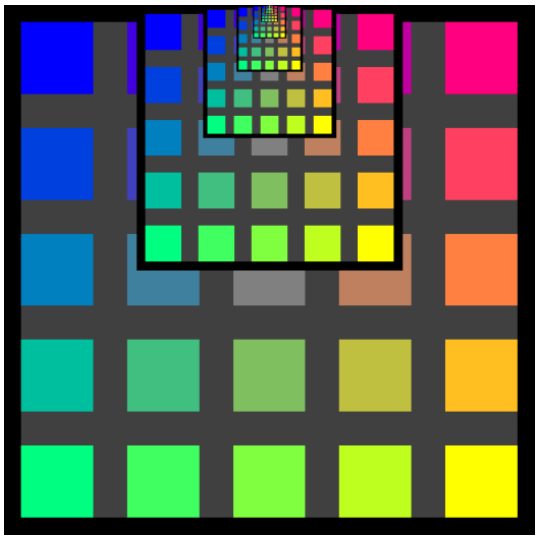
2. From the Sierpiński Triangle to Iterated Function Systems

Yet another way of creating a Sierpiński triangle provides a good way into the theory of *iterated function systems* (IFSs), which, as we shall see, enable intricate and beautifully natural patterns to be generated with surprising ease.

For obvious reasons, we focus here on IFSs that operate on the *Turtle* canvas, and for this purpose, it will help to use a standard background pattern, like that generated by the "Iterated function systems (IFS) background" example program from the "Examples 9" menu (shown here). This includes a grid which is helpful for recognising where the pattern has been rotated or shrunk, and also a spectrum of colours which makes the various parts easy to distinguish.



Now suppose we define a function which "maps" points on this 1000×1000 canvas to other points, for example by mapping (x, y) to $(0.5x + 250, 0.5y)$. Imagine starting at the top of the canvas, scanning down the pixel rows, and copying the colour from each pixel onto the corresponding pixel as determined by this mapping – e.g. the pixel at (100, 500) is copied to (0.5×100+250, 0.5×500) which is (300, 250). The result is shown on the left here. Notice how, in accordance with the specification, the entire original canvas has been copied onto part of itself, but reduced 50% in each direction (as implied by the "0.5*x*" and "0.5*y*"), and shifted right by 250 pixels ("+ 250"), so that the reduced copy is centred horizontally, but still vertically flush with the top of the Canvas.



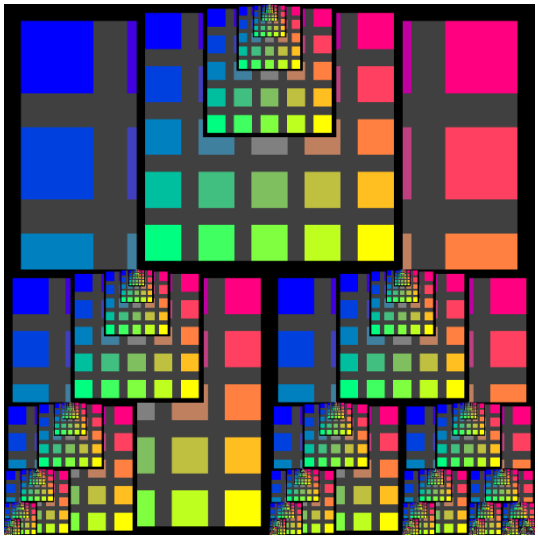
What would happen if, instead of starting from the top in this mapping operation, we started scanning from the bottom and moved upwards? This would mean that by the time we reached halfway up, the bottom of the canvas pattern would *already* have been copied there, so that as we continued scanning upwards, we would be *recopying* that already-copied part. This recopying would be done three-quarters of the way up the canvas, so by the time we reached that point in our sweep up the canvas, we would be *recopying* the *recopy* of a *copy* (to seven-eighths of the way up). Thoughts of the Sierpiński triangle might be rekindled by the resulting image, again shown here. But to follow the Sierpiński path, we need to make *three* reduced copies of our canvas, with the following mappings (the first of which is the same one we just used):

$(x, y) \Rightarrow (0.5x + 250, 0.5y)$ *[top centre]*

$(x, y) \Rightarrow (0.5x, 0.5y + 500)$ *[bottom left]*

$(x, y) \Rightarrow (0.5x + 500, 0.5y + 500)$ *[bottom right]*

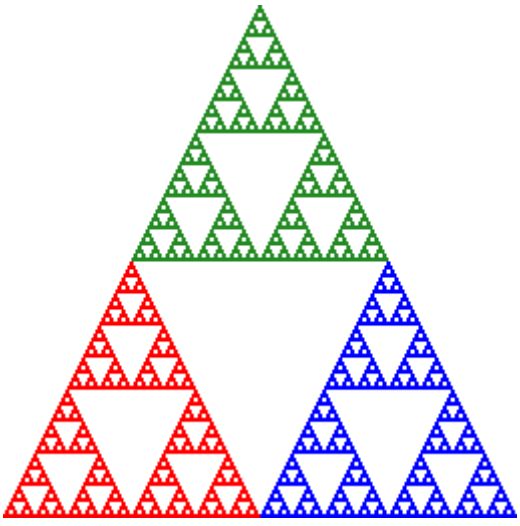
Performing these three mappings in different orders will yield different results, but shown here is what we get if we do them in the order just listed, while scanning the first from bottom to top and the others from top to bottom. Looking at the bottom right of the canvas (most developed because last to be completed), it is easy to imagine that repeating these operations may before long give us something very like a Sierpiński triangle, at least in the repeatedly recopied parts (rather than the backgrounds). If you want to experiment with this sort of thing, try out the Sierpińsky option in the program "IFS mappings on coloured background". This program is quite complicated, so we won't go through it in detail here, but near the end of the program you'll see that the mappings are performed using the *domap* function, whose parameter specifies which mapping is to be done (i.e. 0, 1, or 2), though within this program as it stands they are all scanned from top to bottom, so what you see won't be quite the same.



The coloured background in these programs is helpful for understanding how the mappings work, but otherwise an unwelcome complication – for example, it's obviously impossible to achieve a clean Sierpiński triangle if the background parts are never erased. So to take this further, we shall from now on treat the Canvas – as we did in the Game of Life – as consisting mainly of "live" (black) and "dead" (white) cells, with other colours used only to keep track of intermediate copying (e.g. *darkgrey* if we want minimal contrast*; green*, *red*, or blue for high contrast). To see this in action, load the "Sierpinski triangle (by iterated functions)" program from the "Examples 9" menu, and set it running. We'll be examining the code of this program in a bit more detail in §3 below, but for now let's just focus on its overall operation. It starts with all cells on a 256×256 Canvas set to *black* (i.e. live), and then it repeatedly:

- applies the three Sierpiński mappings to those cells that are not currently dead, colouring the copies in turn *green*, *red* and *blue* according to the mapping (this is done by the *domap* function);
- at the end of each iteration, kills off any cells that are still *black* (and hence have not been copied over), while making the other coloured cells *black* (this is done by the *cleanup* procedure).

As it goes, the *domap* function keeps track of how many new cells have been *born* (i.e. changed from white to some other colour), while the *cleanup* function keeps track of how many have been *killed* (i.e. changed from *black* to *white*). The whole process iterates until the pattern has reached a "fixed point", with no cells being either born or killed during a full iteration. Then the three mappings are performed one last time, so as to show the final pattern with colours indicating the three mapping areas (as pictured here).



With this pattern in view, consider how this program has achieved it, which is conceptually very simple. Starting with a fully "live" canvas, on each iteration it has made three reduced copies of the live-cells pattern (each copy being produced by one of the three defined mappings), and then it has killed off any cells that do not feature in any of those copies. These iterations continued until *every* cell in the pattern was getting recreated on copying, so that the pattern of live cells from one iteration to the next became constant – the "fixed point" of the set of mappings. Now obviously in practice there is a lot of approximation going on here, because our Canvas is only 256x256 (or whatever) and cannot store detail "all the way down". But if we ignore this limitation, and ask what kind of pattern could possibly provide a fixed point of this sort of mapping system, it is clear that *only* a self-similar pattern could do so, for it needs to

have the property that the entire original pattern, shrunk to 50% size, exactly matches a sub-part of that original pattern. So it should be no surprise that the process of iterating our mapping functions – if it reaches a fixed point at all – ends up yielding such a self-similar, or "fractal" pattern (though again, strictly it is a mere approximation to a true fractal, which would have new detail at every scale, all the way down).

What might, however, seem more surprising is that an *iterated function system* of this kind – as long as it is able to get started with some "live" cells – will yield the very same fixed point pattern irrespective of its starting point. We began with a Canvas in which every cell was "live", but in fact the result will be identical (apart from the time taken) if we start instead with a small "live" blot anywhere on the Canvas, for example by replacing the third line of the main program:

```
blank(live)
```

with:

```
setxy(randint(1,xcanvas),randint(1,ycanvas))
colour(live)
blot(10)
```

That blot will be copied (and shrunk) three times on the first iteration, then each of those copies will be copied (and shrunk) on the second iteration, and so on. The copying and continual shrinking of scale means that eventually some part of the copied pattern will get close enough to coincide with pixels in the Sierpiński fixed-point pattern, after which every subsequent threefold copy of those pixels will populate other pixels in that fixed-point pattern (since the fixed-point pattern is by definition the pixel pattern that gets copied into itself). This same logic means that the program could start from one single live pixel (e.g. try replacing "*blank(live)*" with "*pixset(1,1,live)*". And this in turn enables us to explain why the surprisingly simple program "Sierpinski triangle (by random dots)" (in the middle of §1 above) works. Recall that this started with a single random dot, which was moved iteratively halfway towards a (randomly chosen) vertex of the pre-defined large triangle. But such a halfway move is, in effect, exactly the same kind of mapping that we have defined above: if we imagine every point of the large triangle being copied by such a movement, then the result would be a copied triangle of half the size, occupying one corner of the original triangle. So the "Sierpinski triangle (by random dots)" program is, in effect, an implementation of the same kind of iterated function system that we have been exploring here, except that instead of copying the entire pattern on each iteration, it copies only one pixel while retaining all the pixels that it has previously visited, and thus – over time – builds up the same pattern (though probably with small imperfections because it will also, of course, include any initial pixels that it visited before it reached one of the fixed-point pixels).

3. From Iterated Functions to a Dragon Curve

The "Sierpinski triangle (by iterated functions)" program is designed to be very easily modifiable to work with other iterated function systems, simply by changing the relevant constants (i.e. the canvas dimensions and parameters of mappings). This can be seen by comparing it directly with the "Dragon curve (by iterated functions)" program in the same menu, which differs only in the initial settings in its first dozen lines.

Each program starts with the relevant canvas dimensions and "live" colour. Here is how they are set in the Sierpiński program, with *xcanvas* and *ycanvas* both being set to an exact power of 2 to avoid rounding errors when the triangle pattern repeatedly halves its size:

```
xcanvas = 256
ycanvas = 256
live = black
```

Then follow the details of the crucial mappings, which are preceded by two lines which specify the number of mappings, and a *coefficient divisor* – in the Sierpiński program, these are:

```
mappings = 3  # number of mappings
coeffdiv = 2  # dividing factor for all x and y coefficients
```

The latter is needed because the coefficients of *x* and *y* will generally be fractional, whereas *Turtle* only recognises integer numbers. So, for example, we can express an effective *x*-coefficient of 1/2 by specifying the coefficient of *x* as 1 with *coeffdiv* being 2.

When defining the Sierpińsky mappings, instead of using the numbers 128 and 64 to represent half and a quarter of the Canvas width, it is better to refer to these as *xcanvas*/2 and *xcanvas*/4 respectively (likewise *ycanvas*/2 to represent half of the Canvas height). This makes it easy to modify the resolution of the image, by simply changing the initial values of *xcanvas* and *ycanvas* (e.g. to 512 or 1024).[1] Here, then are the mappings as they appear in the "Sierpinski triangle (by iterated functions)" program:

$(x, y) \Rightarrow (\ 1/2\ x + xcanvas/4\ ,\ 1/2\ y)$ *[top centre]*

$(x, y) \Rightarrow (\ 1/2\ x\ ,\ 1/2\ y + ycanvas/2)$ *[bottom left]*

$(x, y) \Rightarrow (\ 1/2\ x + xcanvas/2\ ,\ 1/2\ y + ycanvas/2)$ *[bottom right]*

Rewriting these out in full, with explicit coefficients of *x*, *y*, and a constant term, we have:

$(x, y) \Rightarrow (\ 1/2\ x\ +\ 0\ y\ +\ xcanvas/4\ ,\ 0\ x\ +\ 1/2\ y\ +\ 0\ )$

$(x, y) \Rightarrow (\ 1/2\ x\ +\ 0\ y\ +\ 0\ ,\ 0\ x\ +\ 1/2\ y\ +\ ycanvas/2\ )$

$(x, y) \Rightarrow (\ 1/2\ x\ +\ 0\ y\ +\ xcanvas/2\ ,\ 0\ x\ +\ 1/2\ y\ +\ ycanvas/2\ )$

And now if we look at the corresponding coefficients in all three mappings together, we can see that these have been captured in the lists below, noting again that the *x* and *y* coefficients are understood as being divided by 2 (because this is the specified value of *coeffdiv*):

```
mapxx = [1,         1,         1        ]
mapxy = [0,         0,         0        ]
mapxc = [xcanvas/4, 0,         xcanvas/2]

mapyx = [0,         0,         0        ]
mapyy = [1,         1,         1        ]
mapyc = [0,         ycanvas/2, ycanvas/2]
```

So in general, if the n^{th} mapping is understood as mapping the point (*x*, *y*) to the point (*x*', *y*'), then:

```
x'  =  (mapxx[n]/coeffdiv) x  +  (mapxy[n]/coeffdiv) y  + mapxc[n]
y'  =  (mapyx[n]/coeffdiv) x  +  (mapyy[n]/coeffdiv) y  + mapyc[n]
```

Thus it is possible to express any desired set of mappings straightforwardly within this program structure.

In a parallel way, the "Dragon curve (by iterated functions)" program applies the following three mappings on a 1000×1000 Canvas:

$(x, y) \Rightarrow (0.577y + 95\ ,\ -0.577x + 589)$

$(x, y) \Rightarrow (0.577y + 441\ ,\ -0.577x + 789)$

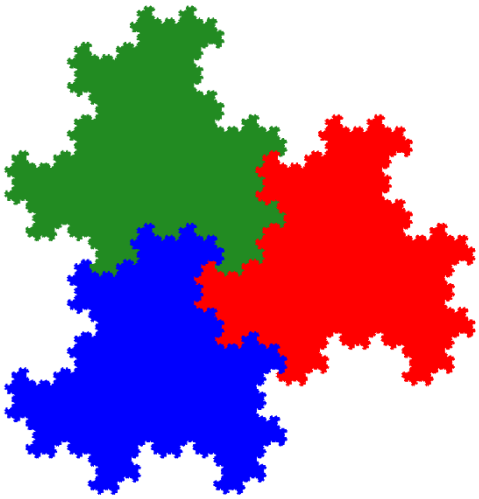
$(x, y) \Rightarrow (0.577y + 95\ ,\ -0.577x + 989)$

To enable the coefficients of *x* and *y* to be specified to three decimal places, this program uses a *coeffdiv* constant of 1000 rather than 2:

```
mappings = 3     # number of mappings
coeffdiv = 1000  # dividing factor for all x and y coefficients

mapxx = [0,    0,    0   ]
mapxy = [577,  577,  577 ]
mapxc = [95,   441,  95  ]
```

[1] In the mappings of the "Iterated function systems (IFS) background" example program of §2 above, *xcanvas*/2 and *ycanvas*/2 were 500, and *xcanvas*/4 was 250. But as already noted, we get a much neater Sierpińsky pattern (all the way down to the pixel level) if the Canvas width and height are an exact power of 2.

```
mapyx = [-577, -577, -577]
mapyy = [0,    0,    0   ]
mapyc = [589,  789,  989 ]
```



All three mappings here have the same coefficients of both *x* and *y*, differing only in their constant terms. They all copy the Canvas, reduced by a linear factor of 0.577, which is approximately $1/\sqrt{3}$ (so the copies will each have an area 1/3 of the original) and turned 90° clockwise (achieved by making the new *x* and *y* coordinates linear functions of the old *y* and negative *x* coordinates). The second mapping is shifted horizontally well to the right of the others, and 200 units lower than the first, while the third mapping is 200 units lower again. When the fixed point pattern has been reached as shown here, the three copies of the pattern will fit together more or less exactly within the original: here they are coloured green, red, and blue in order. Note visually that if you take the whole pattern, shrink and turn it 90° to the right, you do indeed get the correct shape and orientation for all three of the smaller copies.

To keep this program relatively simple, we have used a 1000×1000 Canvas which makes its operation rather prolonged (taking 20 iterations and probably 30 minutes or more). In the following section, we see a program which can generate a (less detailed) Dragon curve more quickly, by using a mapping procedure which allows us to scale any pattern down, and also to locate it more flexibly within the *x*-*y* plane.

4. Introducing the Barnsley Fern and the "IFS Demonstrator" Program

In our Sierpińsky and Dragon curve programs, we used a square Canvas whose origin was (1,1).[2] But many iterated function systems – including Michael Barnsley's famous fern – are better expressed on a Canvas which is placed elsewhere on the plane, and we would prefer to have a program whose generation of these patterns is less dependent on the Canvas dimensions, so as to allow a trade-off between speed and detail (as with the Mandelbrot programs in the document on Chaotic Phenomena). So we now move on to the "IFS demonstrator program", again in the "Examples 9" menu, which includes also the Sierpińsky and Dragon examples (plus a tree). The fern is made by applying the following four mappings to a 500×1000 Canvas stretching from (–236,1) to (263,1000) inclusive, and starting with a single live pixel, e.g. at (1, 1):

$$(x, y) \Rightarrow (0, 0.16y)$$

$$(x, y) \Rightarrow (0.85x + 0.04y, -0.04x + 0.85y + 160)$$

$$(x, y) \Rightarrow (0.2x - 0.26y, 0.23x + 0.22y + 160)$$

$$(x, y) \Rightarrow (-0.15x + 0.28y, 0.26x + 0.24y + 44)$$

To achieve this, while also keeping the program speed relatively manageable – by contrast with the slow Dragon curve program in §3 above – the Barnsley Fern option in the "IFS demonstrator program" uses a

[2] Note that the Sierpińsky program works significantly more neatly with a Canvas extending from 1 to 256 than from 0 to 255, because of how *Turtle* rounds numbers whose decimal part is 0.5. *Turtle* consistently rounds such numbers *upwards* to the next integer (e.g. rounding 11.5 to 12, 12.5 to 13, and –1.5 to –1), unlike standard *Python* 3, which uses what is called "banker's rounding", rounding such numbers to the nearest *even* integer (thus rounding 11.5 to 12, 12.5 to 12, and –1.5 to –2). The latter is a good policy when doing statistics on large sets of numbers (e.g. in financial accounts), because it results in a mixture of rounding up and rounding down, which tend to cancel each other out in the overall totals. But it's obviously not so good when programming visual models, in which we generally want differences between values (e.g. functions of Canvas coordinates) to act in a consistent way. When writing iterated function system programs, it's good to be aware that rounding errors – which are inevitable given the lack of infinite precision – can cause irregularities in the patterns, and experiment might be needed to find the optimal approach.

scaledown variable which reduces the *actual* Canvas size while maintaining the same *virtual* coordinates for the mappings. Thus with a *scaledown* value of 2, the actual Canvas ranges from –118 to 132 horizontally (instead of –236 to 263), and from 1 to 500 vertically (instead of 1 to 1000). This enables the mappings and virtual Canvas dimensions to be specified in a way that's independent of the *scaledown* factor, making it very easy to run the program at different "magnifications" by changing *scaledown*. Here we see *scaledown* being made equal to 2, in the context of the complete list of settings within the *setbarnsley* function:

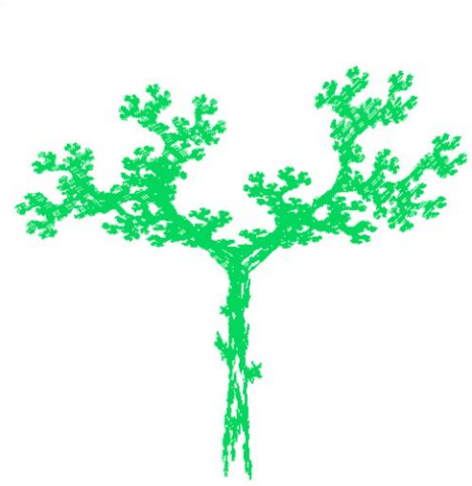
```
xleft = -236
xright = 263
ytop = 1
ybottom = 1000
scaledown = 2
mappings = 4
live = seagreen
copied = emerald
coeffdiv = 100
mapxx=[ 0,   85,  20,  -15 ]
mapxy=[ 0,   4,   -26, 28  ]
mapxc=[ 0,   0,   0,   0   ]
mapyx=[ 0,   -4,  23,  26  ]
mapyy=[ 16,  85,  22,  24  ]
mapyc=[ 0,   160, 160, 44  ]
```

The main program applies this scaling down by calculating some surrogate values for the Canvas dimensions, and the *x*- and *y*-coefficients, as follows:

```
xl = xleft/scaledown
xr = xright/scaledown
yt = ytop/scaledown
yb = ybottom/scaledown
for i in range(mappings):
    mapxc[i] = mapxc[i]/scaledown
    mapyc[i] = mapyc[i]/scaledown
```

Now the Canvas and image resolution are set accordingly, so that the top-left of the Canvas will be at (*xl*,*yt*) and the bottom-right at (*xr*,*yb*), with resolution set to match the coordinate values:

```
canvas(xl,yt,xr-xl+1,yb-yt+1)
resolution(xr-xl+1,yb-yt+1)
```



Then either the Canvas is filled with the *live* colour (e.g. *seagreen* in the list of settings above), or else a single pixel – randomly selected on the Canvas – is made *live* (depending on the *seedpixel* variable as described below), and we enter the same kind of loop as in the Sierpińsky and Dragon IFS programs.

Apart from the Canvas scaling, the other most obvious differences between this program and those two earlier programs arise from the user choices provided by the lengthy (but conceptually straightforward) initial *startprompt* function. First comes a choice of four pattern options (including an inverted version of the Tree curve pictured here) with the following prompt:

```
Select Barnsley fern, Sierpinski, Dragon, or Tree curve? (B/S/D/T)
```

The program waits until one of the four specified keys has been pressed, and then calls the corresponding

function (either *setbarnsley*, *setsierpinsky*, *setdragon*, or *setinvertedtree*) to fix the parameters for the iterated function that is to be generated. (We have already seen – in the previous paragraph – a list of such parameter settings taken from the *setbarnsley* function.) Then follow two more prompts:

```
Start with single Pixel or entire Canvas? (P/C)
Use Uniform or Different colours for mappings? (U/D)
```

Depending on which key is pressed in response to the first of these, the variable *seedpixel* is set either *True* ("P") or *False* ("C"), and likewise with the latest prompt, the variable *uniform* is set either *True* ("U") or *False* ("D"). The *seedpixel* setting determines whether the pattern will start with a single randomly chosen pixel – which sometimes reaches the "fixed point" pattern much more quickly – or with a fully coloured Canvas. The *uniform* setting determines whether the Canvas copying will all be done using the colour specified by the *copied* variable (which is *emerald* in the *setbarnsley* settings above), or alternatively, in a sequence of *Turtle* standard colours, with the first mapping being done in *green*, the second in *red*, the third in *blue*, the fourth in *yellow*, the fifth in *violet*, and so on as necessary. Finally, in the latter (non-uniform) case, a fourth prompt is given, to determine the value of the *multicolour* variable:

```
When completed, leave in Single colour or Multicolour? (S/M)
```

If "S" is pressed, then the eventual "fixed point" pattern at the very end of the program will be left in the *live* colour (e.g. *seagreen*), whereas if "M" is pressed, *multicolour* will be set *True*, and the multicoloured mappings will be performed one last time before the program terminates. (To see the difference vividly, compare the green Tree image just above with the earlier multi-coloured Dragon and Sierpińsky images.)

There is finally one other significant difference between the "IFS demonstrator" program and the previous Sierpińsky and Dragon IFS programs, namely, that as each successive iterated mapping is carried out, the calculation of the new coordinates *x*' and *y*' (as described earlier) is performed using the *divmult* function – as in the Mandelbrot set program in the document on "Chaotic Phenomena" – to avoid potential numerical overload whilst achieving correctly rounded calculations. Thus within the *domap* function, when performing mapping *n*, it replaces:[3]

```
newx = x*mapxx[n]/coeffdiv + y*mapxy[n]/coeffdiv + mapxc[n]
```

with:

```
newx = divmult(x,coeffdiv,mapxx[n]) + divmult(y,coeffdiv,mapxy[n]) + mapxc[n]
```

And likewise it replaces:

```
newy = x*mapyx[n]/coeffdiv + y*mapyy[n]/coeffdiv) + mapyc[n]
```

with:

```
newy = divmult(x,coeffdiv,mapyx[n]) + divmult(y,coeffdiv,mapyy[n]) + mapyc[n]
```

This change is not actually necessary for the four "built-in" options (Barnsley fern, Sierpińsky triangle, Dragon, or Tree), but provides more flexibility for experimentation, in case you wish to try out parameters that are numerically more precise (with larger integer coefficients and also larger values of *coeffdiv*).

5. Understanding the Barnsley Fern

Recall from the beginning of §4 above that the standard Barnsley fern is to be produced using the following four mappings on a Canvas stretching from (–236,1) to (263,1000) inclusive. Our program can scale down both the parameters in the mappings and the Canvas dimensions, proportionately reducing them by an appropriate *scaledown* factor (e.g. 2) in order to achieve more program speed (at the cost of less detail in the image), but we can ignore that complication here. For our aim now is to understand conceptually how these mappings can indeed work to create the beautiful fern image shown below:

[3] *Domap*'s parameter is actually called *mapnum*, but here we use *n* for brevity.

$$(x, y) \Rightarrow (0, 0.16y)$$

$$(x, y) \Rightarrow (0.85x + 0.04y, -0.04x + 0.85y + 160)$$

$$(x, y) \Rightarrow (0.2x - 0.26y, 0.23x + 0.22y + 160)$$

$$(x, y) \Rightarrow (-0.15x + 0.28y, 0.26x + 0.24y + 44)$$

It seems very surprising that such a wonderfully lifelike pattern can be produced in such a mathematically simple way, and also that an otherwise identical program can yield either the Sierpiński triangle or this intricate fern, just by changing the coefficients of a few mappings! But how can the required mappings be worked out? What Barnsley did was to observe that the fern pattern is self-similar with three of its own components, though in a more complex way than the Sierpiński triangle. These three components, coloured red, blue and cyan in the image below, correspond respectively to the second, third, and fourth mappings:

$(x, y) \Rightarrow (0.85x + 0.04y, -0.04x + 0.85y + 160)$
maps the entire fern onto the large red sub-fern;

$(x, y) \Rightarrow (0.2x - 0.26y, 0.23x + 0.22y + 160)$
maps the entire fern onto the blue frond;

$(x, y) \Rightarrow (-0.15x + 0.28y, 0.26x + 0.24y + 44)$
maps the entire fern onto the cyan frond.

(The first mapping – not shown here – simply draws the stalk of the fern, which will then be copied to form the stalk of every frond.)

The "IFS mappings on coloured background" program (which we saw in §2 above) can be used to explore these mappings. On the left here is what we get if we just perform the second mapping *from bottom to top*, which maps the entire fern onto the red sub-fern without recopying bits that have already been copied by the same mapping. As we saw with the Sierpiński mappings, however, things change if we do this in the opposite direction (here, from top to bottom) in such a way that multiple recopying takes place. And below to the right is what we get if all three mappings are performed from top to bottom, in sequence. Notice how the large curved pattern which was generated down the canvas by the multiple recopyings during the second mapping then gets copied by the third and fourth mappings to the places occupied by the blue and cyan fronds in the picture above. Notice also how once these fronds are in place, *subsequent* applications of the first mapping will copy them further down the canvas to produce the smaller fronds on either side of the curving main fern. Again, what is remarkable about Barnsley's discovery is how the definition of appropriate mappings from a planned overall pattern to its intended self-similar components, followed by repeated iteration of those mappings until a fixed point is reached, is sufficient by itself to create whatever pattern has those desired self-similarities. So once we have identified and specified the three self-similarity mappings that characterise our desired fern, no further programming or mental work is required – just press "RUN", and see the pattern emerge!