Notes on Iterated Function Systems, Math 331, Fall 2002 (Suzanne Hruska).

Section 1:

How can we draw a self-similar fractal using linear algebra and Maple? Take a look or think about the kind of fractal you want to draw. A self-similar fractal is made up of N copies of itself, each scaled, rotated, and shifted around so that they fit together to make up the whole. A particular scaling, rotation, and shift, defines a function A from R^2 to R^2, called an *affine transformation*. It turns out that the information which completely encodes the fractal is just the list of N affine transformations required to turn the fractal into each self-similar piece.

We need formulas for affine transformations. First, let $[x, y]$ be a point in R^2. Then scaling is just multiplying each coordinate by some factor, so Scaling is a map

$\mathrm{S}([x, y]) = [r\,x, s\,y]$.

For most of our fractals, we scale both directions by the same amount, but this isn't required. Shifting a figure/point just involved translation in one or both variable, so Translation is a map

$\mathrm{T}([x, y]) = [x + t, y + u]$.

Rotation is the most complicated. To rotate a point by an angle w, you need to multiply the point $[x, y]$ by the *2x2* matrix

$[[\cos(w), -\sin(w)], [\sin(w), \cos(w)]]$.

Then the formula for rotation is:

$R_w([x, y]) = [x\cos(w) - y\sin(w), x\sin(w) + y\cos(w)]$.

Now an affine transformation is a composition of these three functions, so it is of the form:

$\mathrm{A}([x, y]) = [x\,r\cos(w) - y\,s\sin(w) + t, x\,r\sin(w) + y\,s\cos(w) + u]$.

Thus, an affine transformation is determined by the 5 real numbers r, s, t, u, w. Note if there is no rotation, this simplifies to $[x\,r + t, y\,s + u]$.

Let's examine the example of the Sierpinksi triangle. We can view the Sierpinksi triangle as an equilateral triangle containing three copies of itself. Se we need three affine transformations. Each copy is scaled by *1/2* in both x and y-directions, and none of the copies are rotated, so *w=0*, and *r=s=.5* . To compute translations, let's say the origin *(0,0)* is the lower leftmost corner of the triangle. Scaling the whole picture by *1/2* with origin in LL corner gives exactly the LL triangle, so its affine transformation has *t=u=0*. To get the lower right triangle, shift to the right by *1/2* the LL triangle, so this affine transformation has *t=.5, u=0*. Finally, to get the upper central triangle, shift the LL triangle right by *1/2* and up by *1/4*, so *t=.5, u=.25*.

As practice, you should write out the affine transformations needed for several other fractals (find pictures on the course home page, or do a web search for fractals and see what you can find!)

Section 2:

To draw a fractal in Maple using affine transformations, first we define the function 'transform_point', which takes an affine transformation A as a vector with 5 elements, and a point x as a vector with 2 elements,
then computes the image of x under A:

```
> transform_point := proc(A,x)
     evalf( [x[1]*A[1]*cos(A[5]) - x[2]*A[2]*sin(A[5]) + A[3],
       x[1]*A[1]*sin(A[5]) + x[2]*A[2]*cos(A[5]) + A[4] ])
   end:
```

We want to compute the image of a polygon under *A*. A polygon is encoded as a list of vertices, so the image of the polygon will be the list of images of each vertex. So we define a function to compute the image of a list:

```
> transform_polygon:= proc(A,set)
     local i;
     [ seq(transform_point(A, set[i]), i=1...nops(set)) ]
   end:
```

Finally, we write a procedure to plot an IFS which is defined by applying a list of affine transformations to a starting polygon. We split this into two parts. In the first part, IFS is a function which takes as arguments a depth n, a list of affine transformations, and the list of the vertices of a base polygon which we want to apply the affine transformations to repeatedly to get the fractal. The output of IFS is going to be list of the polygons that compose the level n (Lindemeyer) approximation of the fractal. A polygon is encoded as a list of vertices, so the output is a list of lists of vertices.

How does the computation work? The main body is a loop, repeated n times for depth n. The jth run of the loop will compute the polygons composing the jth level. Start with ‘base_vertices‘ defining a polygon, and set the local variable ‘vertices‘ equal to ‘base_vertices‘. Then run a loop for j from 1 to n. The local variable ‘vertices‘ will always be a list of the polygons at the j-1 level, (so it is a list of lists of vertices). At the start of each loop, set the local variable s:= NULL (the empty list). ’s’ will be the sequence of polygons at level j computed so far--we will be building up s one polygon at a time in an inner loop. So, we run an inner loop, going through each affine transformation. For each transformation, we compute the image of each polygon in the (j-1)-list ‘vertices’, and stick those images onto the end of ’s’. When ’s’ contains all the level-j-polygons, we re-define vertices to be s, so its the level j polygons. So at the end, vertices is the level n polygons.Notice IFS is not a recursive procedure.

```
>
> IFS:= proc( n, list_of_transformations, base_vertices)
     local i,j,k,s, vertices;
     vertices:=base_vertices;
     for j from 1 to n do
       s:= NULL;
       for i from 1 to nops(list_of_transformations) do
         s:=s,
         seq(transform_polygon(list_of_transformations[i],
             [vertices][k]),
                k=1...nops([vertices]))
       od;
       vertices := s
     od;
     vertices;
     end:
```

Finally, to draw the picture, we need a function which calls IFS, and then uses the Maple function ’polygonplot’, which takes a list of polygons (each a list of vertices), and draws each polygon filled in:

```
> IFSplot:= proc( n, list_of_transformations, base_vertices)
     local vertices;
```

```
    vertices := IFS(n, list_of_transformations, base_vertices);
    plots[polygonplot]([vertices], axes=none,
  color=black,scaling=constrained);
    end:
```

Now let's do an example. The Sierpinksi triangle starts with a base equilateral triangle, like:

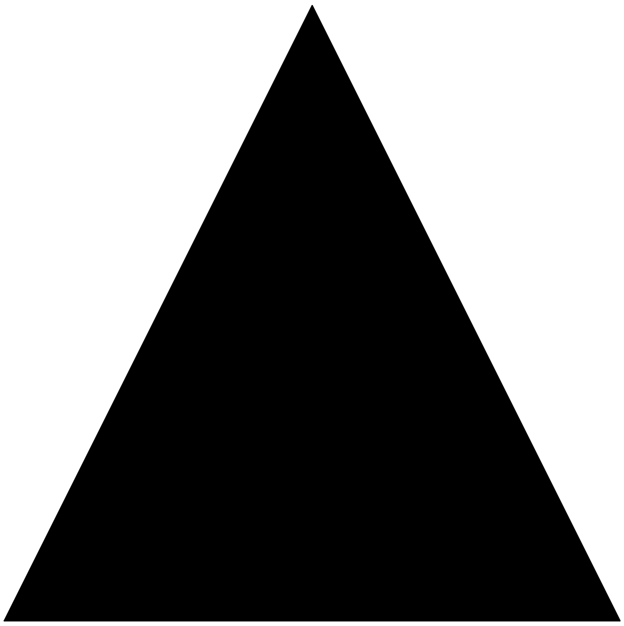
```
> equ_triangle:= [[0,0], [1,0], [.5,1]]:
```

Then we want to apply three affine transformations. Each transformation scales the picture by 1/2, and none of the transformations involve a rotation. So, a[1]=.5, a[2]=.5, a[5]=0. Then, keeping in mind that the triangle starts with the vertices above, one transformation has no translation, so a[3]=a[4]=0, one transformation shifts by .5 to the right, so a[3]=.5, a[4]=0, and the third transformation shifts right by 1/4 and up by 1/2, so a[3]=.25, a[4]=.5:

```
> Sierp_triangle:= [ [.5,.5,0,0,0], [.5,.5,.5,0,0], [.5, .5, .25,
  .5,0] ]:
> IFS( 0, Sierp_triangle, equ_triangle);
```

[[0, 0], [1, 0], [.5, 1]]

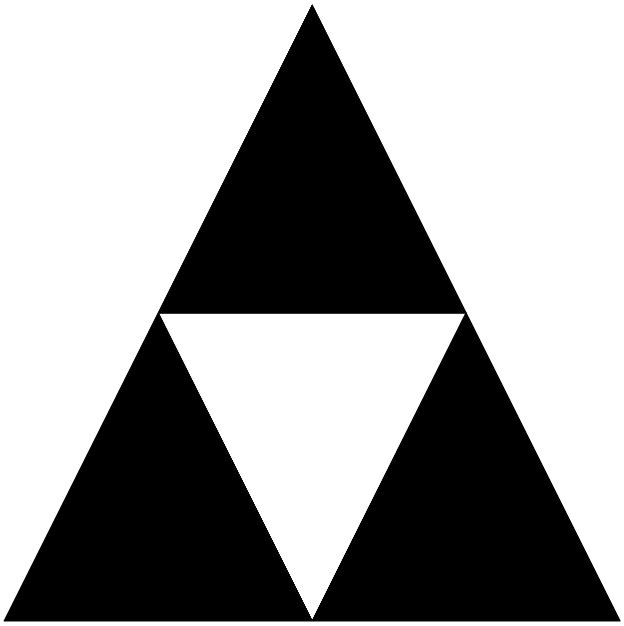
```
> IFSplot(0, Sierp_triangle, equ_triangle);
```



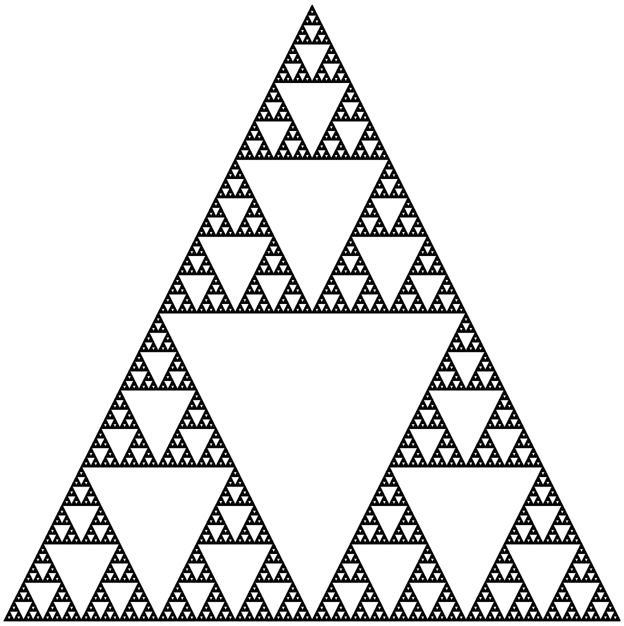
```
> IFS(1,Sierp_triangle, equ_triangle);
```

[[0., 0.], [.5, 0.], [.25, .5]], [[.5, 0.], [1.0, 0.], [.75, .5]], [[.25, .5], [.75, .5], [.50, 1.0]]

```
> IFSplot(1,Sierp_triangle, equ_triangle);
```

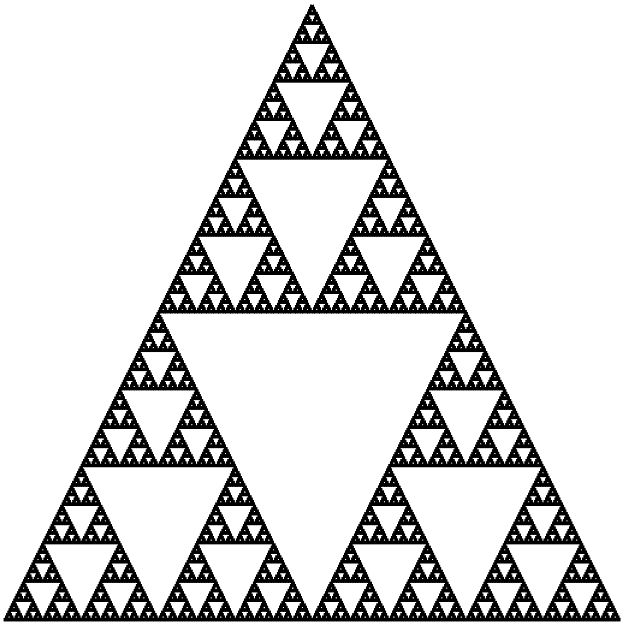


```
> IFSplot(8, Sierp_triangle, equ_triangle);
```

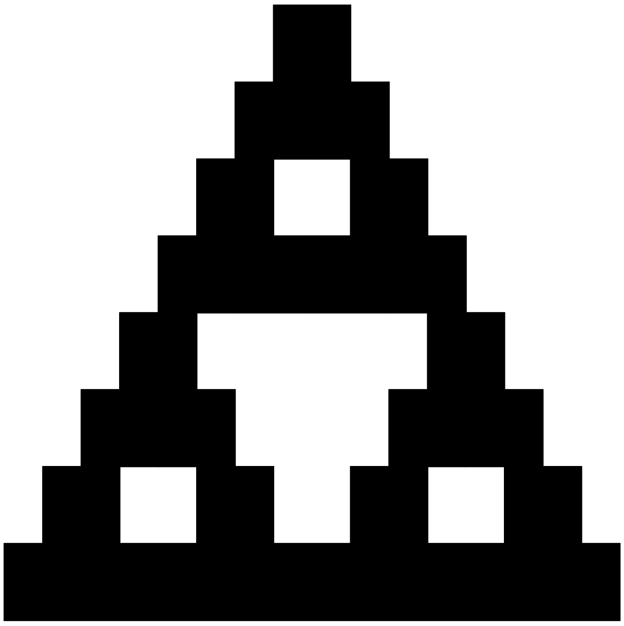


In fact, in the limiting picture, it doesn't matter what base polygon you started with! Any shape will be sucked into the same pattern. Try a square:

```
> base_square:= [[0,0], [0,1], [1,1], [1,0]]:
> IFSplot(8, Sierp_triangle, base_square);
```



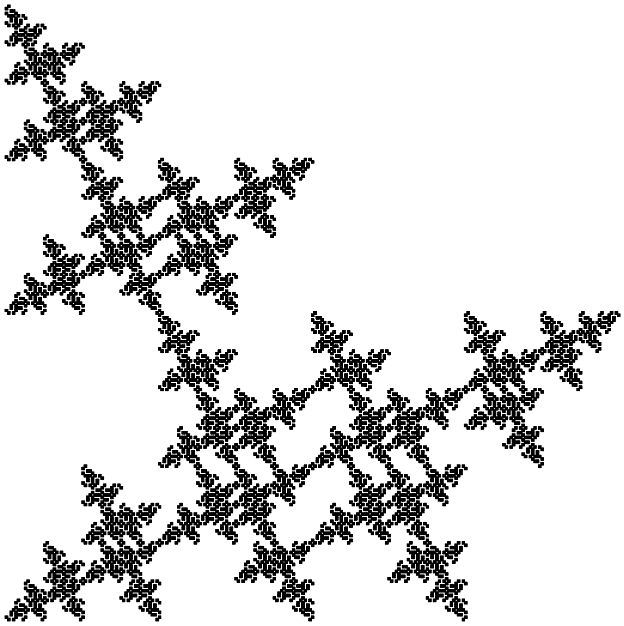
```
> IFSplot(3, Sierp_triangle, base_square);
```



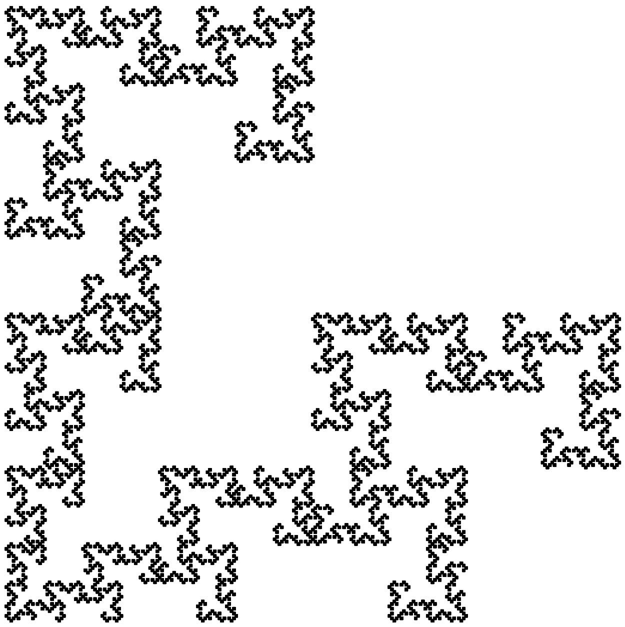
This is why we can say that the fractal is an 'attractor' for the set of affine transformations. We call the list of affine transformations and process of iterating them an 'Iterated Function System', or 'IFS'.

Here are a couple more basic examples (you should come up with more interesting ones, vary some of the parameters and see what happens!):

```
> F1:= [ [.5,.5,-.25,.25,0], [.5,.5,-.25,-.25,Pi/2],
  [.5,.5,.25,-.25,-Pi/2] ]:
> IFSplot(8, F1, base_square);
```



```
> F2:= [ [.5,.5, -.25, .25, -Pi/2],[.5,.5,-.25,-.25,0],[.5,.5,
  .25,-.25, -Pi/2] ]:
> IFSplot(8, F2, base_square);
```



References: Our function for drawing IFS's is a modified version of one found at: http://www.oci.unizh.ch/mirror/maple/frame03.htm*, it was written by Alain Schauber, Forbach, France, 1996.*