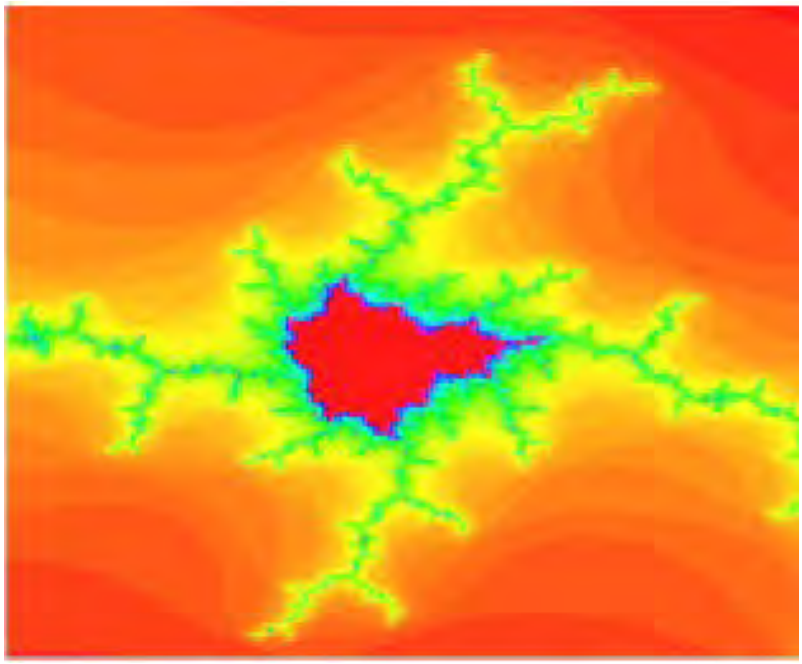
Chaotic Rendezvous:

Introductory Chaos Experimentation Using Maple Software and a Desktop Computer

by Thomas E. Oberst



Chaotic Rendezvous: Introductory Chaos Experimentation Using Maple Software and a Desktop Computer

Thomas E. Oberst

Duquesne University
Department of Physics
Pittsburgh, PA 15282

Chapter 9

Fractals

Loosely speaking, a *fractal* is a geometrical object which shows self-similarity at every level of magnification.[1] A common example of a fractal is the edge of a maple leaf. When you view a maple leaf at arm's length, you can see the several pointed peninsulas on the sides of the maple leaf that give it its characteristic shape (think of the Canadian flag). If you hold the leaf closer to your eyes—perhaps a foot or so away—you will see that in addition to the main large peninsulas there are many smaller pointed peninsulas projecting from the sides of the main peninsulas and also from the clefts between the main peninsulas. The smaller peninsulas resemble the larger peninsulas in shape, spacing, and orientation. If you hold the leaf very close to your eyes and focus on a smaller region along the edge of the leaf, you may be able to make out a third generation of pointed peninsulas similar in shape, spacing, and orientation to the first two generations. Further magnifications can be seen under a microscope. But no matter how closely you look or how small a region of the original maple leaf's edge you are able to magnify, the edge of the leaf will never become more simplified in structure or appearance. With each successive magnification, fine structure will appear which resembles the less fine larger-scale structure of weaker magnifications and, in some cases, which resembles the entire original leaf itself. Some other examples of fractal-like objects which appear in nature are snowflakes, seashells, coastlines, mountains, clouds, and networks of blood vessels. Fractals are all around us.

Until the field of fractal geometry was pioneered by Benoit B. Mandelbrot (1924-) in the late 1970s, scientists have been forced to use the simplified shapes of Euclidean geometry to describe nature: circles, squares, triangles, etc. But often nature cannot be confined to these simplified shapes. Fractal geometry provides an entirely new and perhaps more fitting language for use to describe the natural world.

Furthermore, one of the main assumptions of calculus is that infinitesimal subdivisions of a curve are more simple than the curve as a whole (this is the main concept behind the integral). But this is not true with the many objects in nature which resemble fractals. For

[1] A more formal definition of a *fractal* is a subset of $\Re^n$ which shows self-similarity at every level of magnification and whose fractal dimension exceeds its topological definition. For more information see Devaney 1992, pages 178-190.

this reason chaos has been called the "anti-calculus revolution."[2]

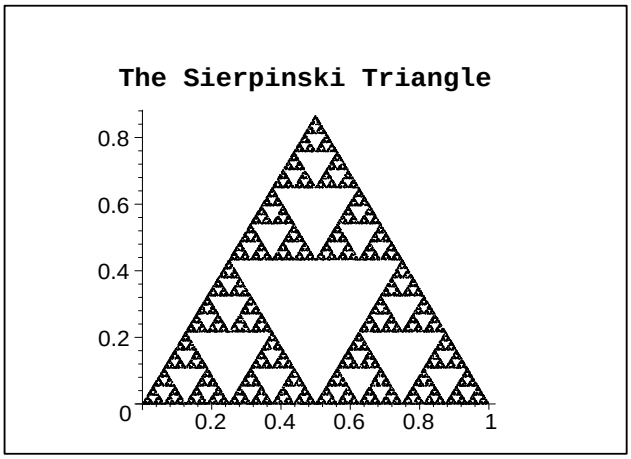
There is no technical connection between dynamical systems and fractal geometry. Dynamical systems are constantly in motion while a fractal is a static image. However, it has been observed that plots of the chaotic regions of most dynamical systems are fractals. There is obviously a deep connection between dynamical systems and fractals. Mandelbrot began asking questions about the relation of fractal objects and the mathematics of dynamical systems in 1975. Despite many advances in the fields of both fractal geometry and dynamical systems since that time, we are still far from understanding the full extent of their connection. But the fact that they are related has given us reason to believe that the natural world is ruled by dynamical systems: tiny and simple mathematical laws repeated over and over and over have the ability to create such tremendous complexity as we see around us.

One of the simplest fractals is the *Sierpinski triangle*. Like most fractals, the Sierpinski triangle is created through an iterative process. We first choose three points forming the vertices of an equilateral triangle and call these points A, B, and C. Next, we choose an arbitrary initial seed (x_0, y_0) from the same plane as A, B, and C (which we assume here to be the xy-plane). The first step in the iterative procedure consists of randomly selecting one of the vertices A, B, or C and computing the point midway between (x_0, y_0) and our selected vertex. This point becomes (x_1, y_1). For the second iteration we again randomly choose one of A, B, or C and call the midpoint between (x_1, y_1) and this vertex (x_2, y_2). The process is repeated again and again, and each time we select one of the vertices A, B, or C randomly. (One way to do this might be to use a die and let 1 and 2 represent A, 3 and 4 represent B, and 5 and 6 represent C). After a large number of iterations are performed we plot the set of points (x_n, y_n). One might expect the entire triangular region within the vertices to become evenly "washed out" with points, and perhaps even appear completely filled in if enough points are plotted. After all, the vertices are being randomly selected, so why should the points (x_n, y_n) show any preference as to where they land? But it turns out that they do. The resulting plot—which is called the Sierpinski triangle—shows a remarkable intricately structured pattern of triangles. It is pictured in the Maple plot below. An explanation of the Maple commands used to create the Sierpinski triangle follows.

```
> X_vertex:=array(1..3):  Y_vertex:=array(1..3):
> X_vertex[1]:=0:  Y_vertex[1]:=0:
> X_vertex[2]:=1:  Y_vertex[2]:=0:
> X_vertex[3]:=1/2:  Y_vertex[3]:=sqrt(3)/2:
> X:=array(0..20000):  Y:=array(0..20000):
> X[0]:=.4:  Y[0]:=.2:
> f:=rand(1..3):
> for i from 1 to 20000 do
```

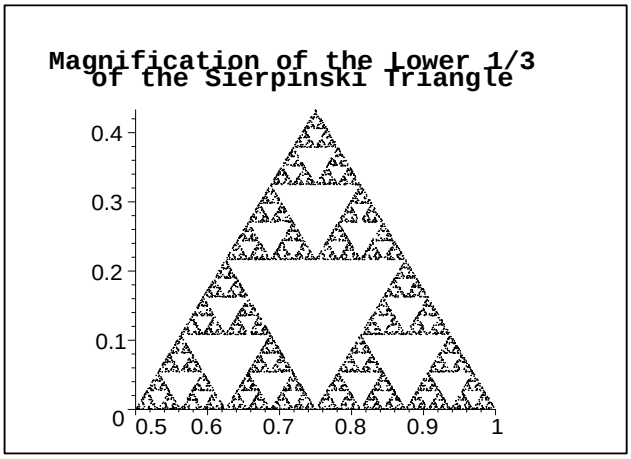
[2]Baranger, Michel, The Meaning of the Chaos Revolution. Center for Theoretical Physics, Laboratory for Nuclear Science and Department of Physics, MIT, Cambridge, MA 02139.

```
> a:=f():
> X[i]:=evalf((X[i-1]+X_vertex[a])/2):
> Y[i]:=evalf((Y[i-1]+Y_vertex[a])/2):
> od:
> with(plots):
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained, symbol=point,
> title=‘The Sierpinski triangle‘, titlefont=[COURIER,BOLD,12]);
```

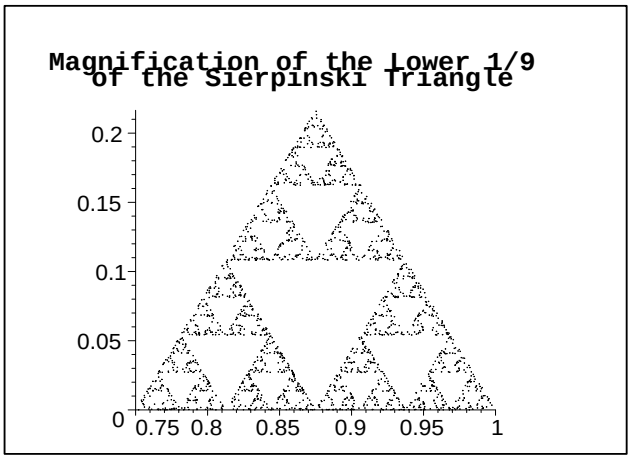


The Sierpinski triangle is quite an amazing figure (as are all fractals). I have zoomed in on the lower right 1/3 and lower right 1/9 of the Sierpinski triangle and displayed the results below. Note the amazing self-similarity! I don't think I need to convince you further that the Sierpinski triangle shows self-similarity at every level of magnification and is indeed a fractal. The lower 1/9 appears less well defined than the lower 1/3 and the original. This is because only 20,000 points were plotted. As we zoom in closer and closer we must exclude more and more points. Plotting 180,000 points would make the lower 1/9 as sharp as the original Sierpinski triangle plotted with 20,000 points. Plotting 180,000 points would take quite some time, however—if your computer could do it at all.

```
> pointplot({seq([X[n],Y[n]], n=10..20000)}, view=[.5..1,0..sqrt(3)/4],
> scaling=constrained, symbol=point, title=‘Magnification of the Lower 1/3
\n of the Sierpinski triangle‘,
> titlefont=[COURIER, BOLD, 12]);
```




```
> pointplot({seq([X[n],Y[n]], n=10..20000)}, view=[.75..1,0..sqrt(3)/8],
> scaling=constrained, symbol=point, title=‘Magnification of the Lower 1/9
\n of the Sierpinski triangle‘,
> titlefont=[COURIER, BOLD, 12]);
```



To create the Sierpinski triangle using Maple, we first must define the three vertices of an equilateral triangle. In the example above, for simplicity I chose two of these to be the points (0,0) and (1,0) in the xy-plane. The third vertex can therefore be chosen to be $(1/2,\sqrt{3}/2)$, also in the xy-plane. I defined the x- and y-coordinates of these three vertices separately in the arrays "X_vertex" and "Y_vertex," respectively. Since there are only three vertices, the arrays X_vertex and Y_vertex need only three components. Here is a list of the vertices used in the example above:

```
> for i from 1 to 3 do [X_vertex[i], Y_vertex[i]] od;
```

$$[0,\ 0]$$

$$[1,\ 0]$$

$$[\frac{1}{2}, \frac{1}{2}\sqrt{3}]$$

Next, we must create two arrays to hold the x- and y-components of the points which will eventually belong to our fractal triangle. These arrays should have a large number of points, since the shape of a fractal often becomes clear only after a large number of iterations. In the example above I call these arrays x and y and give them 20,000 members. (All of the fractals shown in this chapter have 20,000 points). We must also define an initial seed for our iterative process. I arbitrarily chose mine to be (0.4, 0.2).

As part of the iterative process which creates the Sierpinski triangle we must randomly select one of the three vertices. Maple has a built-in procedure called `rand` which randomly selects an integer from a given range of integers—precisely what we need. To effectively employ the `rand` procedure we assign it an arbitrary function name, such as f:

```
> f:=rand(1..3):
```

To produce a random integer between one and three we simply call f:

```
> f();
                                  1
> f();
                                  2
```

Note the empty argumentative parentheses that must follow f. Also note that f produces a new random integer (from the specified range) each time it is called.

```
> for i from 1 to 15 do f() od;
                                  1
                                  1
                                  3
                                  2
                                  3
                                  3
                                  3
                                  1
                                  2
                                  1
                                  2
                                  1
                                  3
                                  3
                                  1
```

During a given iteration we wish to randomly select a vertex number, but we wish to

hold the selected number until that given iteration is complete. Therefore, we must assign a name to that number so that it remains fixed. Here is a short example:

```
> a:=f();
```

$$a := 1$$

a has been assigned the value 1, and it will not change until a is redefined. We can thus use a as the index for the arrays X_vertex and Y_vertex to produce the first vertex:

```
> [X_vertex[a], Y_vertex[a]];
```

$$[0,\ 0]$$

The vertex produced was the origin, (0, 0), as expected.

Now take a look at what would happen if we reused the random generating function f each time (a common mistake):

```
> f();
```

$$2$$

```
> [X_vertex[f()], Y_vertex[f()]];
```

$$[1,\ \frac{1}{2}\sqrt{3}]$$

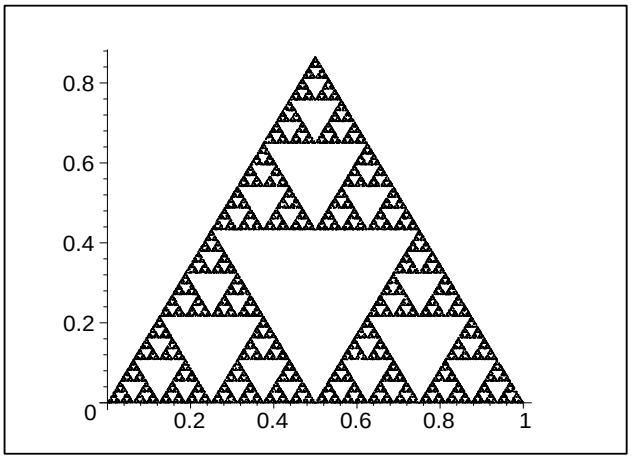
The point $(1, \sqrt{3}/2)$ is returned, which isn't even one of our vertices!

The rest of the iterative process is straightforward. A `for` loop is used to compute the next point in the triangle, (x_n, y_n), to be the midpoint of the previous point, (x_{n-1}, y_{n-1}) with the randomly chosen vertex, (X_vertex_a, Y_vertex_a). During each round of the loop, a is redefined as a new randomly chosen integer between one and three. Finally, `pointplot` is used to plot the set of points $\{(x_n, y_n)\}$. Be sure to throw out the first few points (10 or so) so the orbit is allowed to reach its asymptotic behavior.

Now that you know how to create the Sierpinski triangle using Maple, let's play around with it. The first thing I would like to point out is that the Sierpinski triangle is independent of the choice of the initial seed. In fact, you don't even have to choose the initial seed to be located within the triangle itself. Below I have recreated the triangle using the initial seed (10, 10), which is certainly "far" from the vertices relative to the size of the triangle itself. The Sierpinski triangle looks identical as long as you remember to throw out the transient iterations!

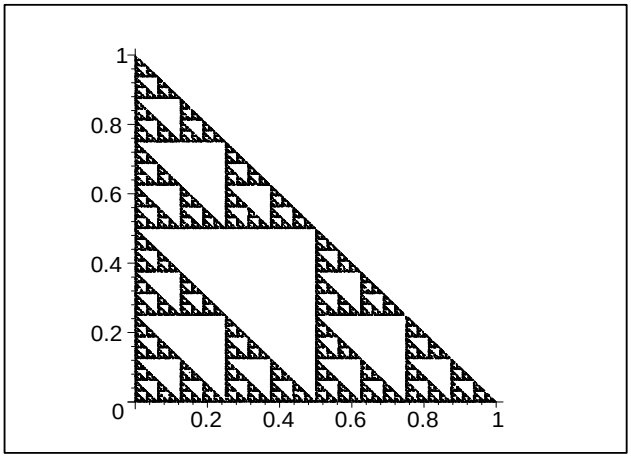
```
> X[0]:=10:  Y[0]:=10:
> for i from 1 to 20000 do
> a:=f():
> X[i]:=evalf((X[i-1]+X_vertex[a])/2):
> Y[i]:=evalf((Y[i-1]+Y_vertex[a])/2):
```

```
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



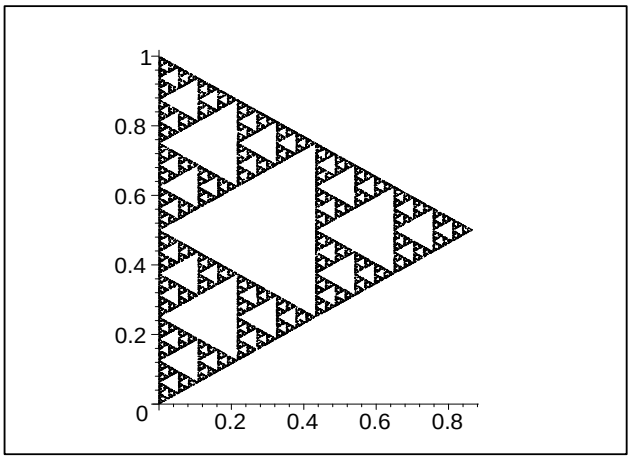
Many other fractals objects can be created using the general procedure for the Sierpinski triangle. For instance, we can create a "Sierpinski Right triangle" by changing our vertices:

```
> X_vertex[1]:=0:  Y_vertex[1]:=0:
> X_vertex[2]:=1:  Y_vertex[2]:=0:
> X_vertex[3]:=0:  Y_vertex[3]:=1:
> for i from 1 to 20000 do
> a:=f():
> X[i]:=evalf((X[i-1]+X_vertex[a])/2):
> Y[i]:=evalf((Y[i-1]+Y_vertex[a])/2):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



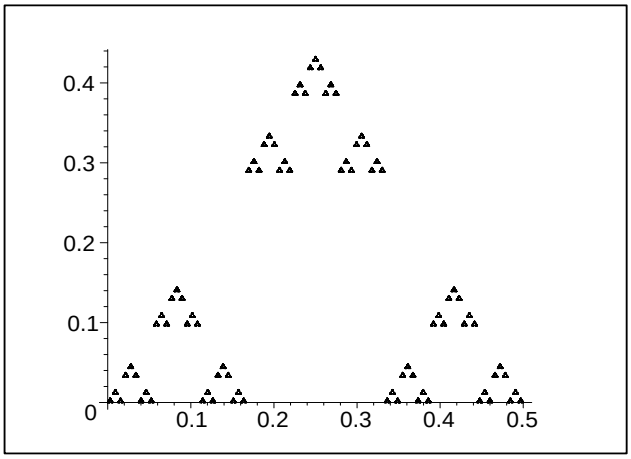
We can also try switching the coordinates of the vertex which has been randomly selected. It is not terribly surprising to find that doing this causes the Sierpinski triangle to flip sideways:

```
> X_vertex[1]:=0:  Y_vertex[1]:=0:
> X_vertex[2]:=1:  Y_vertex[2]:=0:
> X_vertex[3]:=1/2:  Y_vertex[3]:=sqrt(3)/2:
> X[0]:=.4:  Y[0]:=.2:
> for i from 1 to 20000 do
> a:=f():
> X[i]:=evalf((X[i-1]+Y_vertex[a])/2):
> Y[i]:=evalf((Y[i-1]+X_vertex[a])/2):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



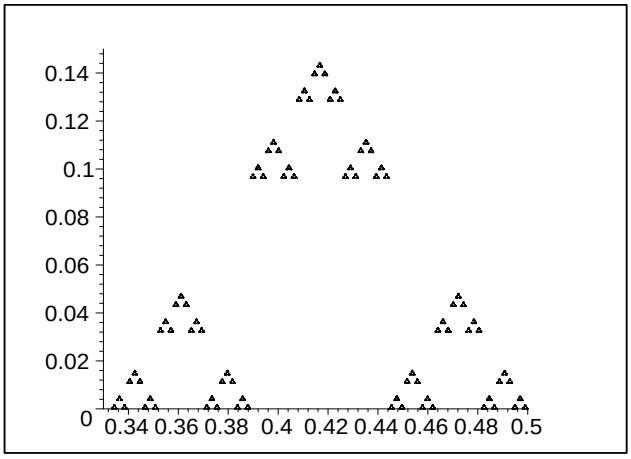
Now, what if, instead of taking the *midpoint* of the last iteration with a randomly selected vertex, we take only one-third of the distance between the last iteration and the vertex?

```
> for i from 1 to 20000 do
> a:=f():
> X[i]:=evalf((X[i-1]+X_vertex[a])/3):
> Y[i]:=evalf((Y[i-1]+Y_vertex[a])/3):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



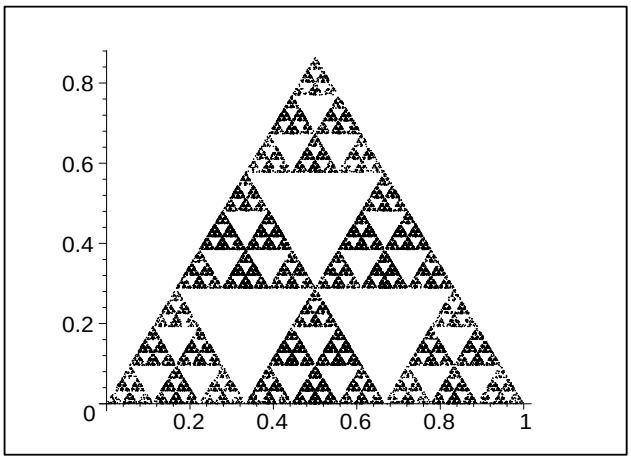
This interesting fractal resembles the general pattern of the Sierpinski triangle, but is missing a lot more points. Also very interesting is the fact that this fractal is contained within the same space as the lower left 1/3 of the Sierpinski triangle. We note the self-similarity of this fractal by zooming in on its lower right 1/3 in the plot pictured below:

```
> pointplot({seq([X[n],Y[n]], n=10..20000)},
> view=[.33..(.5),0..(.15)], scaling=constrained, symbol=point);
```



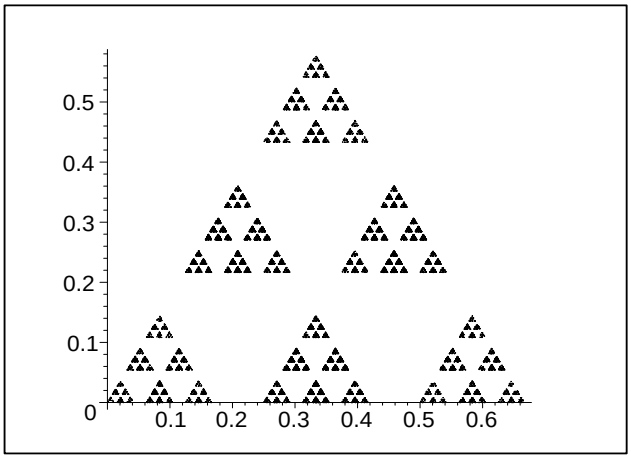
We can make things even more interesting by adding *two* randomly chosen vertices to the mix each time, taking the average of these two points with our previous iterative value to be the next iterative value. The resulting fractal is remarkably similar to the original Sierpinski triangle. It even occupies the same amount of space. The only difference is that the first level of magnification in this fractal reveals nine congruent triangles similar to the original fractal, whereas the first level of magnification in the Sierpinski triangle has only three congruent triangles similar to the original.

```
> for i from 1 to 20000 do
> a:=f():
> b:=f():
> X[i]:=evalf((X[i-1]+X_vertex[a]+X_vertex[b])/3):
> Y[i]:=evalf((Y[i-1]+Y_vertex[a]+Y_vertex[b])/3):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



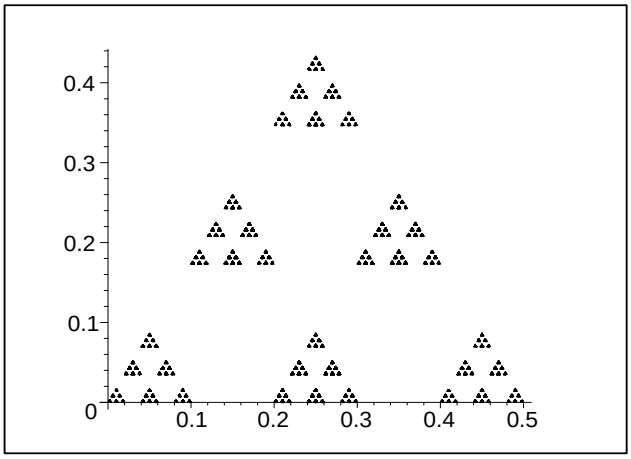
We can put a spin on this variation by dividing by four rather than three. As was the case above, dividing our distances by larger integers not only creates more "white space" in the fractal but also reduces its overall size:

```
> for i from 1 to 20000 do
> a:=f():
> b:=f():
> X[i]:=evalf((X[i-1]+X_vertex[a]+X_vertex[b])/4):
> Y[i]:=evalf((Y[i-1]+Y_vertex[a]+Y_vertex[b])/4):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



Next, we take it one step further and divide by five:

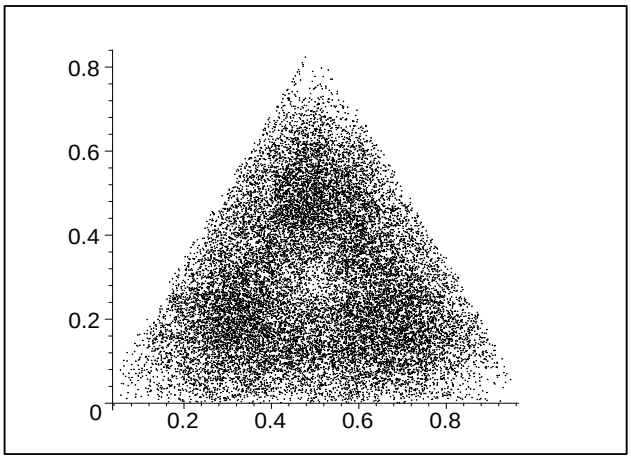
```
> for i from 1 to 20000 do
> a:=f():
> b:=f():
> X[i]:=evalf((X[i-1]+X_vertex[a]+X_vertex[b])/5):
> Y[i]:=evalf((Y[i-1]+Y_vertex[a]+Y_vertex[b])/5):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



Yet another possibility is to take the average of one randomly selected vertex and the *previous two iterates* to compute our next iterative value. This variation produces perhaps the most interesting result so far. The points fall within the boundaries of the original Sierpinski triangle, but do not organize into a definite structure. Instead they cluster into

a formation strikingly similar to probability clouds. Three definitively darker regions are discernable. Whether this object is a fractal or not, I am not sure. If it is a fractal, it may be of the Cantor set type.[3]

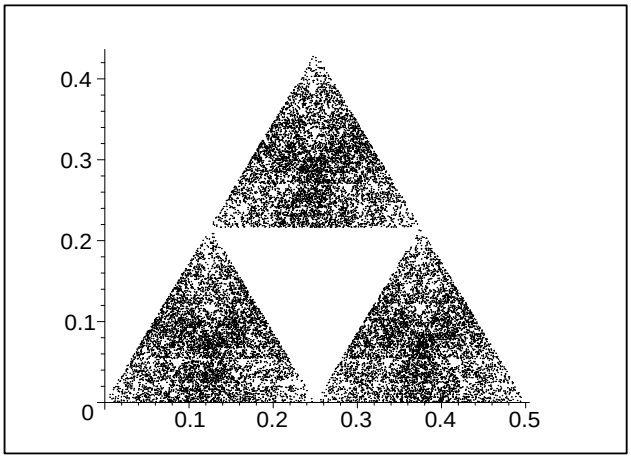
```
> X[1]:=.3:  Y[1]:=.7:
> for i from 2 to 20000 do
> a:=f():
> X[i]:=evalf((X[i-1]+X[i-2]+X_vertex[a])/3):
> Y[i]:=evalf((Y[i-1]+Y[i-2]+Y_vertex[a])/3):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



If we divide by four instead of three, again we see a decrease in overall size as in the other cases when we increased the integer denominator. We also find that a large triangular region has been removed from the center of the figure, leaving three triangles which appear to have the probability-cloud format which we saw in the last graph. Again, I am not entirely sure that this is a fractal. My hunch is that it is a fractal of the Cantor set type.

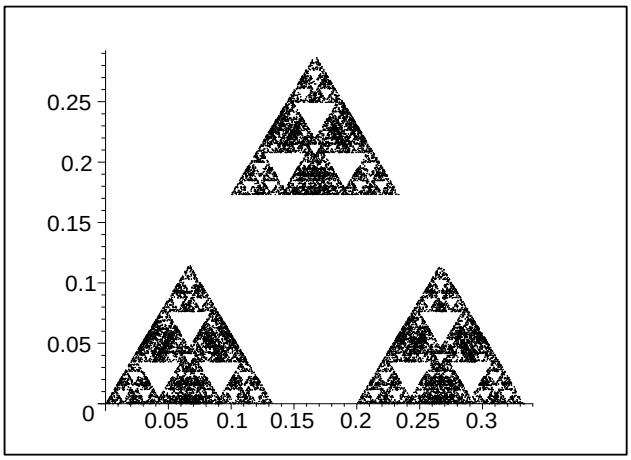
```
> for i from 2 to 20000 do
> a:=f():
> X[i]:=evalf((X[i-1]+X[i-2]+X_vertex[a])/4):
> Y[i]:=evalf((Y[i-1]+Y[i-2]+Y_vertex[a])/4):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```

[3]See Devaney 1992, pages 178-180.



Using the same iteration procedure as in the last plot, but dividing by five instead of four, creates an even more intricate and interesting pattern with a smaller overall area and smaller probability-cloud-like triangles. After seeing this, I am quite convinced that this object and the objects in the previous two graphs are indeed fractals.

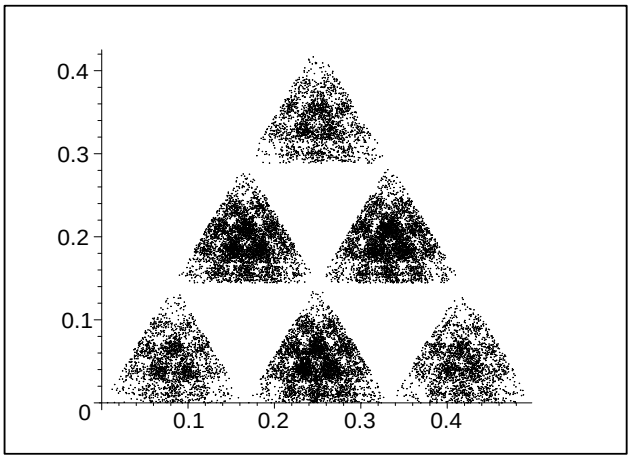
```
> for i from 2 to 20000 do
> a:=f():
> X[i]:=evalf((X[i-1]+X[i-2]+X_vertex[a])/5):
> Y[i]:=evalf((Y[i-1]+Y[i-2]+Y_vertex[a])/5):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



Another weird probability-cloud-like fractal can be created by adding a second randomly selected vertex to the process used for the previous three images and dividing (quite

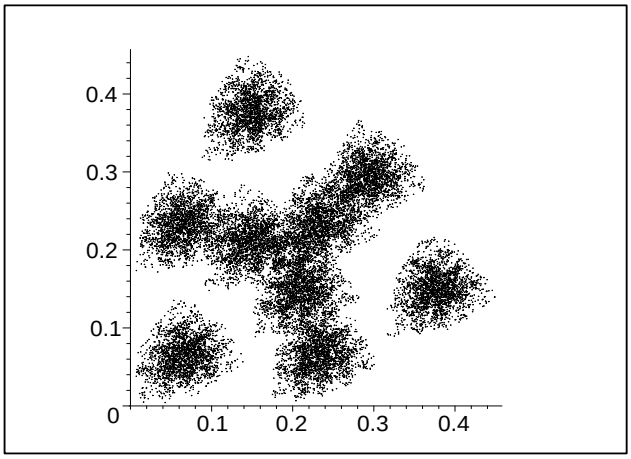
arbitrarily) by six:

```
> for i from 2 to 20000 do
> a:=f():
> b:=f():
> X[i]:=evalf((X[i-1]+X[i-2]+X_vertex[a]+X_vertex[b])/6):
> Y[i]:=evalf((Y[i-1]+Y[i-2]+Y_vertex[a]+Y_vertex[b])/6):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



It turns out that it is quite difficult to "destroy" the fractal. By "destroy," I mean to mess up the iteration process so badly that you no longer get a fractal. Try it for yourself. I will give it one last attempt by not only adding two vertices and the previous two iterative values, but also swapping four of the x- and y-coordinates:

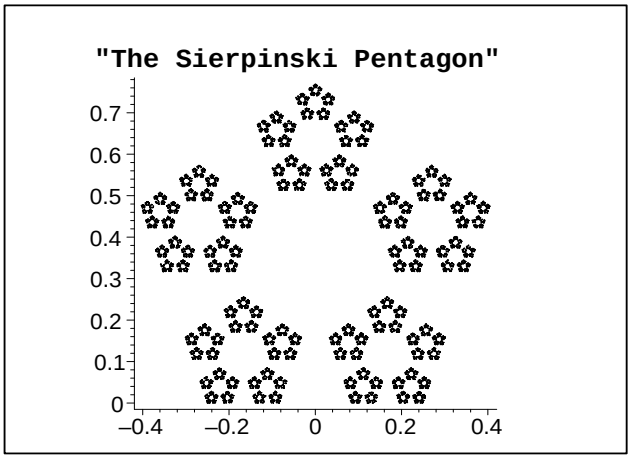
```
> for i from 2 to 20000 do
> a:=f():
> b:=f():
> X[i]:=evalf((X[i-1]+Y[i-2]+X_vertex[a]+Y_vertex[b])/6):
> Y[i]:=evalf((Y[i-1]+X[i-2]+Y_vertex[a]+X_vertex[b])/6):
> od:
> pointplot({seq([X[n],Y[n]], n=10..20000)}, scaling=constrained,
> symbol=point);
```



Now that is just downright crazy. But it's still a fractal!

Finally, realize that this process is not restricted to three vertices. Below I have created a fractal similar to the Sierpinski triangle, but with five vertices. I proudly name it the "Sierpinski pentagon."

```
> f2:=rand(1..5):
> X2:=array(1..5):  Y2:=array(1..5):
> X2[1]:=-.5:  Y2[1]:=0:
> X2[2]:=.5:  Y2[2]:=0:
> X2[3]:=-.5-cos(2*Pi/5):  Y2[3]:=sin(2*Pi/5):
> X2[4]:=.5+cos(2*Pi/5):  Y2[4]:=sin(2*Pi/5):
> X2[5]:=0:  Y2[5]:=.5*tan(3*Pi/10)+.5/cos(3*Pi/10):
> x2:=array(0..20000):  y2:=array(0..20000):
> x2[0]:=.5:  y2[0]:=.5:
> x2[1]:=.3:  y2[1]:=.9:
> for i from 2 to 20000 do
> a:=f2():
> x2[i]:=evalf((x2[i-1]+X2[a])/3):
> y2[i]:=evalf((y2[i-1]+Y2[a])/3):
> od:
> pointplot({seq([x2[n],y2[n]], n=10..20000)}, symbol=point, axes=frame,
> scaling=constrained, title=`"The Sierpinski Pentagon"`,
> titlefont=[COURIER,BOLD,12]);
```



I would like to introduce a new type of fractal called the *fractal fern*. I begin by using the `restart` command to clear all previous definitions used in the fractals above.

```
> restart:
> maxpoints:=20000;
```

$$maxpoints := 20000$$

```
> throwout:=100;
```

$$throwout := 100$$

```
> p1:=77; p2:=89; p3:=99;
```

$$p1 := 77$$
$$p2 := 89$$
$$p3 := 99$$

```
> X:=array(0..maxpoints); Y:=array(0..maxpoints);
```

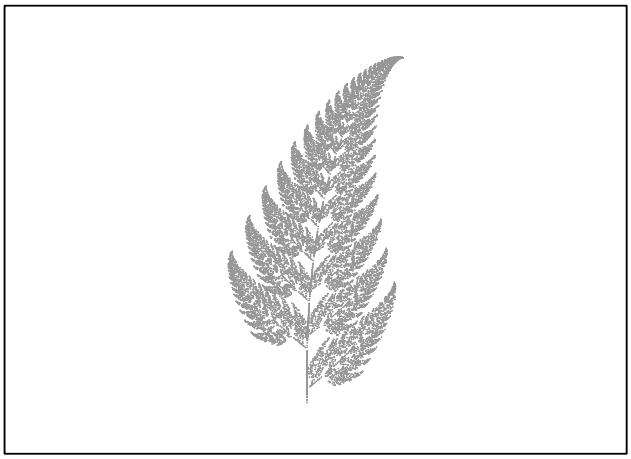
$$X := \text{array}(0..20000, [])$$
$$Y := \text{array}(0..20000, [])$$

```
> X[0]:=0; Y[0]:=0;
```

$$X_0 := 0$$
$$Y_0 := 0$$

```
> for i from 1 to maxpoints do
> f:=rand(0..99):
> a:=f();
> if a<p1 then
> X[i]:=.85*X[i-1]+.04*Y[i-1]+.075:
> Y[i]:=-.04*X[i-1]+.85*Y[i-1]+.18:
> else if a<p2 then
> X[i]:=.20*X[i-1]-.26*Y[i-1]+.4:
```

```
> Y[i]:=.23*X[i-1]+.22*Y[i-1]+.045:
> else if a<p3 then
> X[i]:=-.15*X[i-1]+.28*Y[i-1]+.575:
> Y[i]:=.26*X[i-1]+.24*Y[i-1]-.086:
> else
> X[i]:=.5:
> Y[i]:=.16*Y[i-1]:
> end if end if end if od:
> with(plots):
> pointplot([seq([X[n],Y[n]], n=throwout..maxpoints)], symbol=point,
> axes=none, scaling=constrained, color=khaki);
```



This amazing fractal really looks like a fern! And because it is a fractal, we know that it has all of the infinite complexity that a real fern (or any other biological organism) would have. But what is most amazing is that the formulae which are used to produce this image are very simple in form. In fact, they are linear. After viewing fractals like this one it is difficult not to become convinced that mathematics rules the natural world.

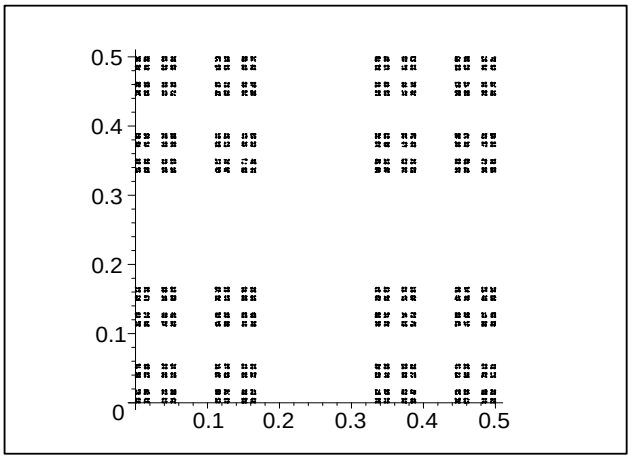
To create the fractal fern we used a Maple structure that you may not be familiar with, but that we will be using throughout the rest of this manual: the `if then` statement. It is simple in both appearance and use. We tell Maple, `if` a certain condition exists or is met, `then` execute the following commands. The statement must end with the words `end if`. If you wish to execute different commands when the specified condition does not exist or is not met, then you can use the keyword `else`. Several `else if` statements may be nested (as in the example above). The last condition does not use `if` but only `else`. At the end of the nested statements, each use of the keyword `if` must conclude with the keywords `end if`. Carefully examine the commands used to generate the fractal fern above as an example of `if then else` statements in action. I believe you will find that they are not difficult to

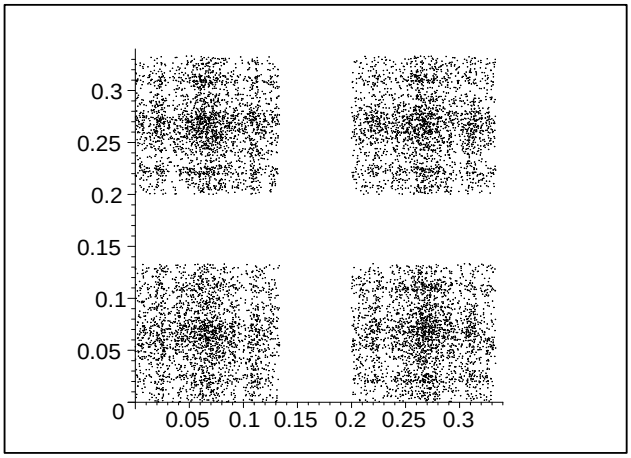
understand or use.

9.1 Experiment: Sierpinski Squares

1. Plot the Sierpinski triangle using Maple.

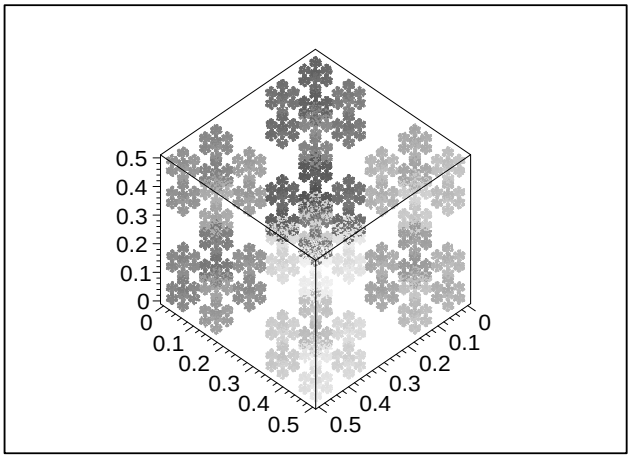
2. Create three different Sierpinski-like fractals using four vertices instead of three. Call them "Sierpinski squares" if you like. Use the general method used to create the Sierpinski triangle and the variations of the Sierpinski triangle shown earlier in this chapter. It is much easier to destroy the Sierpinski square fractal than it is to destroy the triangle. Therefore you may have to play around a bit using several combinations of vertices, previous iterates, and integer denominators in your iteration process. However, several of the variations used earlier in this chapter to create different Sierpinski-like triangular fractals should also produce good Sierpinski-like square fractals. Two possibilities are pictured below. For simplicity, I recommend that you work with the unit square (vertices: (0, 0), (0, 1), (1, 0), (1, 1)).





3. Choose one of your three Sierpinski-like square fractals and perform three successive zoom-ins (using the `view` option in the argument of `pointplot`) to show its self-similarity. Note that this would only be possible on the first of the two fractals I have shown above.

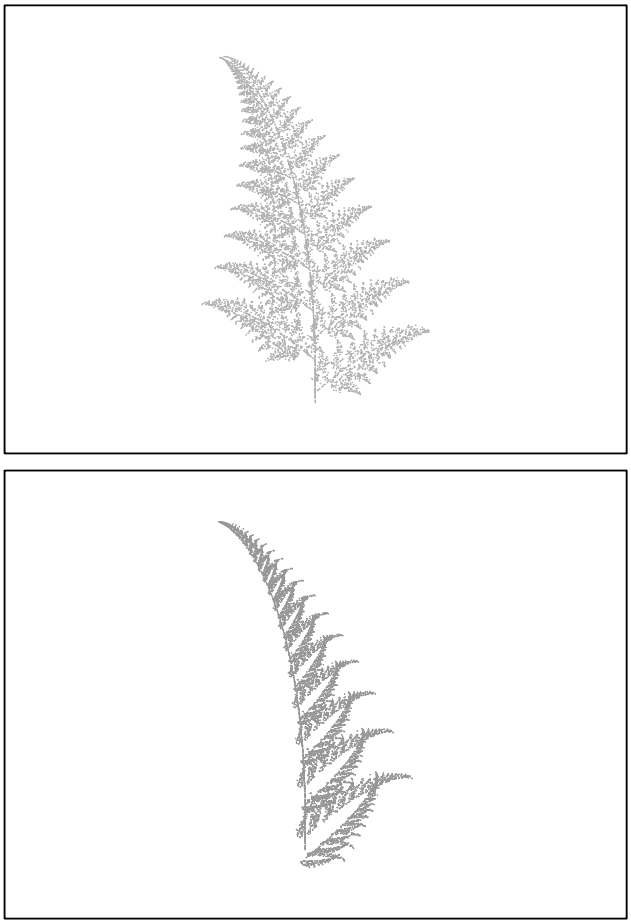
4. Create a three-dimensional "Sierpinski Cube" fractal using similar methods as we have been using to create Sierpinski-like Triangles and Sierpinski Squares. You will have to specify eight vertices and extend all of your arguments to three dimensions. Again, you may have to play around with various combinations of vertices, former iterates, and integer denominators to get a good-looking fractal. I have created one possible Sierpinski Cube and pictured it below.



9.2 Experiment: The Fractal Fern

1. Plot the fractal fern using Maple.

2. Create three fractal variations of the fractal fern by tweaking the numerical values in the equations located within the nested `if then` statements of the commands used to produce the fern. Be careful, however, since changing these values too much may destroy the fern. Your resulting fractals should still look like ferns but should be different from the fern shown earlier in this chapter. You can also try changing the signs of the numerical values. See if you can get the fern to bend to the left instead of to the right. Also see if you can produce a fern with less leaves or more leaves. A couple possibilities are shown below.

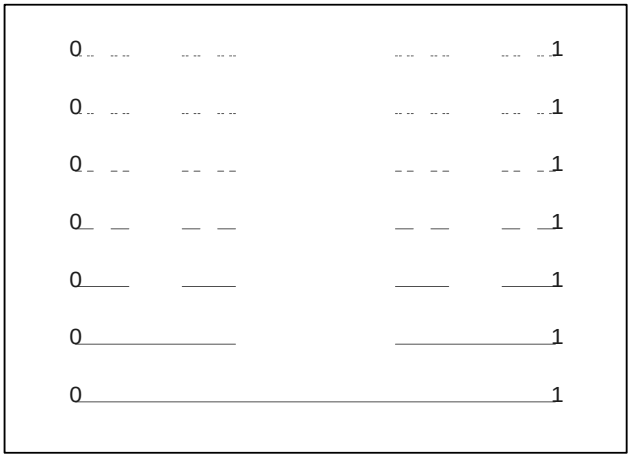


By the way, the best Maple colors to plot your ferns in are `khaki` and `aquamarine`.

9.3 Experiment: The Cantor Set

Write a sequence of commands in Maple (or create a procedure if you like) to produce the *Cantor Middle-Thirds set*, commonly called simply the *Cantor set*. The zeroth iterate of the Cantor set consists of the unit line segment from 0 to 1. The first iterate is created by removing the middle third of the first iterate, leaving two line segments that extend from 0 to 1/3 and from 2/3 to 1. The middle third, which is removed, is an open interval, such that the remaining line segments include the points 1/3 and 2/3. The second iterate is then created by removing the open middle thirds of these two line segments, leaving four line segments that extend from 0 to 1/9, 2/9 to 1/3, 2/3 to 7/9, and 8/9 to 1 (inclusive of all these points). The actual Cantor set is the set of points which remains after the iteration process is carried out infinitely many times. Note that the total length of the set decreases by a third with each iteration, and the number of line segments increases by a factor of 2 with each iteration. It can thus be proven that the number of points in the final Cantor set is infinite and uncountable, and also that the final total length of the Cantor set is zero. Do

not attempt to produce more than five or six iterations or you may freeze up your computer. Your results should be similar to the Maple plot shown below which I created of the first seven iterations from the Cantor set.



9.4 Experiment: The Koch Curve

Write a sequence of commands in Maple (or create a procedure if you like) to produce the *Koch curve*. The zeroth iterate of the Koch curve consists of the unit line segment from 0 to 1. The first iterate is created by removing the middle third of this line segment, just as was done with the Cantor Set, but then replacing the middle third with two line segments each equal to the length of the middle third which was removed. These two segments are added in such a way so as to form an equilateral triangular spike protruding from the original line segment minus the middle third. We are thus left with four connected line segments of equal length. To create the second iterate we remove the middle thirds of each of these four segments and replace them with spikes. The number of line segments increases by a factor of four with each iteration and the length of the curve increases by a factor of 1/3 with each iteration. It can thus be proven that in the limit of infinitely many iterations the Koch curve has an infinite length and an infinite number of sides. Do not attempt to produce more than five or six iterations or you may freeze up your computer. Your results should be similar to the Maple plots shown below which I created of iterations one through five of the Koch curve.