Chapter 4

A turtle in a fractal garden

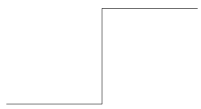
4.1 Turtle Graphics

Imagine you have a small turtle who responds to certain commands like "move forward a step", "move back a step", "turn right", and "turn left". Imagine also that this turtle carries a pen (or just leaves a trail of green slime wherever he crawls— the pond he came out of was none too clean). As long as our turtle follows our commands without tiring, we can get him to make rather intricate patterns easily.[1] The turtle graphics commands described here are not a standard part of maple. Rather, they are in a file "turtle.txt" which we will examine in more detail later in this chapter. Maple has a somewhat different implementation included in the share library. In order to get our turtle commands known to maple, we must first load them with a command such as

```
> read(`turtle.txt`);
```

Let us suppose our turtle is given a string such as "FLFRF" and interprets it as the sequence of instructions "Move Forward. Turn Left. Move Forward. Turn Right. Move Forward." Then we should see the turtle move in a zigzag pattern:

```
> TurtleCmd(`FLFRF`);
```



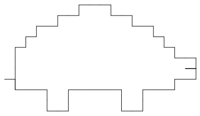
We shall begin assuming the turtle knows the following commands:

F — move forward one step	R — turn right (but don't move)
B — move backward one step	L — turn left

[1]This is the premise of the computer language LOGO, invented in the late 1960s and often used to teach children the basic ideas of programming. The "turtle graphics" we do here has very little to do with real LOGO, which is a much more powerful system.

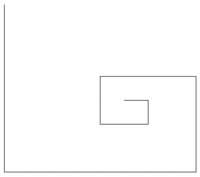
We will point out some more commands later, although this set is sufficient to do quite a lot.[2] For example, we can ask our turtle to make a self portrait:

```
> TurtleCmd(`FRFLFFF RFFLFFLFFR FFFFF RFFLFFLFFR FFFLFR FFLF RBFL
             FLFF RFLFRFLFRFLF FRFLFFRFLFFF LFRFFLF RFFLF RFLFRFLF FFF`);
```



Or, we can make a spiral:

```
> TurtleCmd(`FRFR FFRFFR FFFFRFFFFR FFFFFFFFRFFFFFFFR`);
```

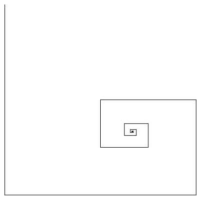


This last can be represented more succinctly if we add a pair of new commands, namely

G — "Grow", i.e. double the unit of length   S — "Shrink" (halve the unit of length)

Then the last picture can be produced by the string "FRFRG FRFRG FRFRG FRFR", or, even better, we can make it spiral much more by repeating "FRFRG" twenty times:

```
> TurtleCmd(cat(seq(`FRFRG`,i=1..20)));
```

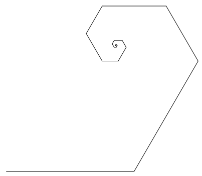


You may have noticed that the command `TurtleCmd` always resizes the resulting graphic to be of constant width. While this may seem a minor point, it is not. In any single string of turtle commands, "FF" will always be twice as long as "F"; however, `TurtleCmd(`F`)` and `TurtleCmd(`FFFFFFFFFF`)` produce identical results. This is because the a single "F" in the second case is 1/10 the width of the figure, while it is the full width in the first case.

We can also adjust some of the turtle's attributes. For example, the turtle need not make only 90 degree turns. We use the command `ResetTurtle()` to set the turtle back to its initial state, then `SetTurtleAngle` to specify the angle (in degrees) that the turtle will turn left or right when it encounters an L or R command.

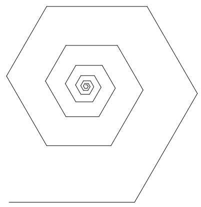
[2] In fact, this set of commands is already redundant. We could replace L with RRR and B with RRFLL.

```
> ResetTurtle(); SetTurtleAngle(60);
  TurtleCmd(cat(seq(`FRFRG`,i=1..20)));
```



`SetTurtleScale` allows us to modify the behavior of the S and G commands. Its argument is the factor the unit of length changes when an S is encountered.

```
> ResetTurtle(); SetTurtleAngle(60); SetTurtleScale(.8);
  TurtleCmd(cat(seq(`FRFRG`,i=1..20)));
```



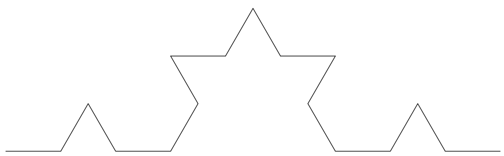
4.2 A fractal

Now let's try to do something more interesting. Begin with a straight segment (command "F"). Now remove the middle third of the segment, and replace it with the top two sides of an equilateral triangle. Since the overall figure produced by the turtle is always rescaled to unit size, we can represent the latter with the command "F LFRRFL F", where the part LFRRFL describes the bump we added. Of course, we have to set the angle the turtle turns to be 60°, and we need the RR in the middle to make a 120° turn.

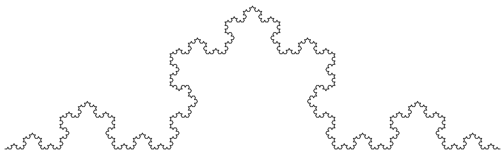
```
> SetTurtleAngle(60);
  plots[display](array([TurtleCmd(`F`), TurtleCmd(`FLFRRFLF`)]));
```

Now let's put a bump on each segment in the second figure. This is the same as replacing each F in the second command with "FLFRRFLF", giving us

```
> TurtleCmd(`FLFRRFLF L FLFRRFLF RR FLFRRFLF L FLFRRFLF`);
```



Putting bumps on each segment of that figure, and then adding smaller bumps to each segment of *that* figure, and then doing it yet again, gives us a very wiggly curve:



Notice that the width of each bump added is 1/3 of the one added previously, so by the time we have done this five times (as in the previous figure), the bumps are $1/3^5$ the size of the whole figure. If we continue much more, the changes become smaller than we can discern. It should be clear that there is a well defined "curve" which corresponds to the limit of doing this process infinitely often. We shall try to make this statement a little more precise now.

Let $\mathcal{K}_0$ be the figure at the 0-th stage, that is, the straight segment, and let $\mathcal{K}_1$ be the curve after we have added one bump. In general, let $\mathcal{K}_n$ be the curve after the n-th step. We claim that there is a well-defined limit curve $\mathcal{K}_\infty = \lim_{n\to\infty} \mathcal{K}_n$.

To make sense of this statement, we need a way to measure how far apart two sets in the plane are. That is, if S_1 and S_2 are two sets of points, we shall define the "Hausdorff distance" between S_1 and S_2 as follows. First, define the distance from a point x to a set S as the distance between x and the closest point in S, that is

$$d(x, S) = \inf_{y\in S} d(x, y).$$

Then let the Hausdorff distance between S_1 and S_2 be the greatest of all such distances, as x moves through each set.

$$d(S_1, S_2) = \max\left\{\sup_{x\in S_1} d(x, S_2), \quad \sup_{x\in S_2} d(x, S_1)\right\}$$

Once you have absorbed that, it should be simple to see that that we now need only show that for any $\epsilon > 0$, we can always find N sufficiently large so that $d(\mathcal{K}_N, \mathcal{K}_n) < \epsilon$ for all $n > N$. This follows from the fact that the size of the bumps goes down by a factor of 1/3 at each stage. Thus, $d(\mathcal{K}_n, \mathcal{K}_{n+1}) < 1/3^{n+1}$, and

$$d(\mathcal{K}_N, \mathcal{K}_n) < \sum_{j=N}^{n} 1/3^j < \frac{1}{2\cdot 3^{N-1}}.$$

So there certainly is a well-defined limit curve $\mathcal{K}_\infty$.

Now, this "curve" $\mathcal{K}_\infty$ has some interesting properties. First, notice that the distance in the plane from one end to the other is 1, but the "curve" itself is infinitely long. In fact, the length of any small part of the curve is infinite, as well. We can see that by examining how the lengths of each $\mathcal{K}_n$ varies with n:

curve	# segments	segment length	total length
$\mathcal{K}_0$	1	1	1
$\mathcal{K}_1$	4	1/3	4/3
$\mathcal{K}_2$	16	1/9	16/9
$\mathcal{K}_3$	64	1/27	64/27
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathcal{K}_n$	4^n	$1/3^n$	$4^n/3^n$

At each stage, every segment is replaced by four segments 1/3 as long. Thus, the overall length grows by a factor of 4/3 at each step. For any finite length ℓ we choose, we can find an n so that the length of $\mathcal{K}_n$ is greater than ℓ.

Note also that $\mathcal{K}_\infty$ is nowhere differentiable: it has corners densely throughout it. Finally, note that $\mathcal{K}_\infty$ is "self-similar". By this we mean that if you take any small piece of it, no matter how small, there is a small copy of the whole $\mathcal{K}_\infty$ within it.

The curve $\mathcal{K}_\infty$ is an example of a fractal.[3] This particular example is called the von Koch curve, and was discovered by H. von Koch in the late nineteenth century. This curve, and others like it, caused quite a stir in the mathematics community at the time because of its peculiar properties. We will explore additional fractals and some of their properties in the remainder of this chapter.

The particular way we generated the Koch curve is called an "L-system" or a "Lindenmeyer system". In an L-system, one begins with an initial figure (called the initiator), and a set of rules for modifying any figure to obtain the next in the sequence. In our example above, the initiatior was a straight segment (F) and the recursion rule was F $\mapsto$ FLFRRFLF.

4.3 Recursion and making a Koch Snowflake with Maple

In the previous section, we didn't give the specifics of exactly how we generated the Koch curve of level 5 which we showed. The general procedure was pointed out ("Begin with F. Replace each F with FLFRRFLF. Repeat 5 times."), but if you actually tried to do it yourself, you might have found it a bit tricky. Since the turtle command for the curve has 2388 letters, certainly it isn't something you would expect someone to type in directly.

Instead, a small maple procedure was written to generate it. Before giving the details, let's analyze the process a bit. We shall rely heavily on the self-similar nature of the Koch curve $\mathcal{K}_\infty$.

[3]The term "fractal" was coined by Benoit Mandelbrot in the late 1970s. Unfortunately, there is no universally accepted definition for this term. Some definitions require that a fractal be a self-similar object, but this excludes the most well-known example of a fractal, the Mandelbrot set, which is only approximately self-similar. We shall take it to mean any set whose Hausdorff dimension exceeds its topological dimension (we will see what this means in section 4.5).

Notice that the curve $\mathcal{K}_\infty$ is made up of four small copies of itself, arranged as

$$\mathcal{K}_\infty \mathrm{L}\, \mathcal{K}_\infty \mathrm{RR}\, \mathcal{K}_\infty \mathrm{L}\, \mathcal{K}_\infty.$$

If we use K_n to denote the set of turtle commands used to produce $\mathcal{K}_n$, then a similar statement is true for each K_n, namely

$$K_n = \begin{cases} K_{n-1}\mathrm{L}K_{n-1}\mathrm{RR}K_{n-1}\mathrm{L}K_{n-1} & \text{if } n > 0 \\ \mathrm{F} & \text{if } n = 0 \end{cases}.$$

4.3.1 Recursive functions.

This is an example of a recursive definition. You've probably encountered such definitions previously in math classes. For example, the factorial function can be defined either as

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

or

$$n! = \begin{cases} n \cdot (n-1)! & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}.$$

The second definition is much simpler to implement in a programming language which allows recursive functions, such as maple.[4] To see this, let's implement the factorial function in both ways:

```
> fact1 := proc(n::posint)
    local i,ans;
    ans := 1;
    for i from 1 to n do
      ans := ans*i;
    od;
    ans;
  end:
> fact2 := proc(n::posint)
    if (n=1) then
        1;
    else
       n*fact2(n-1);
    fi;
  end:
```

Notice how much shorter and clearer the second version is.[5] However, there is a price to pay: a recursively defined function has a bit more overhead than a non-recursive one. However,

[4] Most programming languages these days do allow recursive function calls, although this was not the case until 1985 or so.

[5] We've stretched things a bit here. We could have implemented the factorial even more simply as `fact3:=n->product(i,i=1..n);`. However, this relies on the fact that the factiorial is a special product.

the savings in simplicity often overwhelms the minor loss in speed.[6]

To test your understanding, you might try to write a procedure to generate the Fibonacci numbers, which are defined by

$$F(n) = \begin{cases} F(n-1) + F(n-2) & \text{if } n > 2 \\ 1 & \text{if } n = 1 \text{ or } n = 2 \end{cases}.$$

Try to implement this with both a recursive and non-recursive procedure. The non-recursive one is much more complicated!

4.3.2 A recursive procedure to generate K_n

With the idea of recursion in our bag of tools, we can now easily write our procedure to generate the Koch curve, using the observation we made in §4.3, namely that

$$K_n = \begin{cases} K_{n-1}\mathrm{L}K_{n-1}\mathrm{RR}K_{n-1}\mathrm{L}K_{n-1} & \text{if } n > 0 \\ \mathrm{F} & \text{if } n = 0 \end{cases}.$$

```
> Koch:= proc (n::nonnegint)
    if (n=0) then
       `F`;
    else
        cat( Koch(n-1), `L`, Koch(n-1), `RR`, Koch(n-1), `L`, Koch(n-1));
    fi;
  end:
> Koch(2);
```

$$FLFRRFLFLFLFRRFLFRRFLFRRFLFLFLFRRFLF$$

If you think about it for a second, you might notice that what we have just done is to write a little program (`Koch`) whose output is a program in another language (the string of turtle commands which describe K_n). While this may seem odd at first, doing this sort of thing is not at all uncommon.

4.3.3 The Koch Snowflake

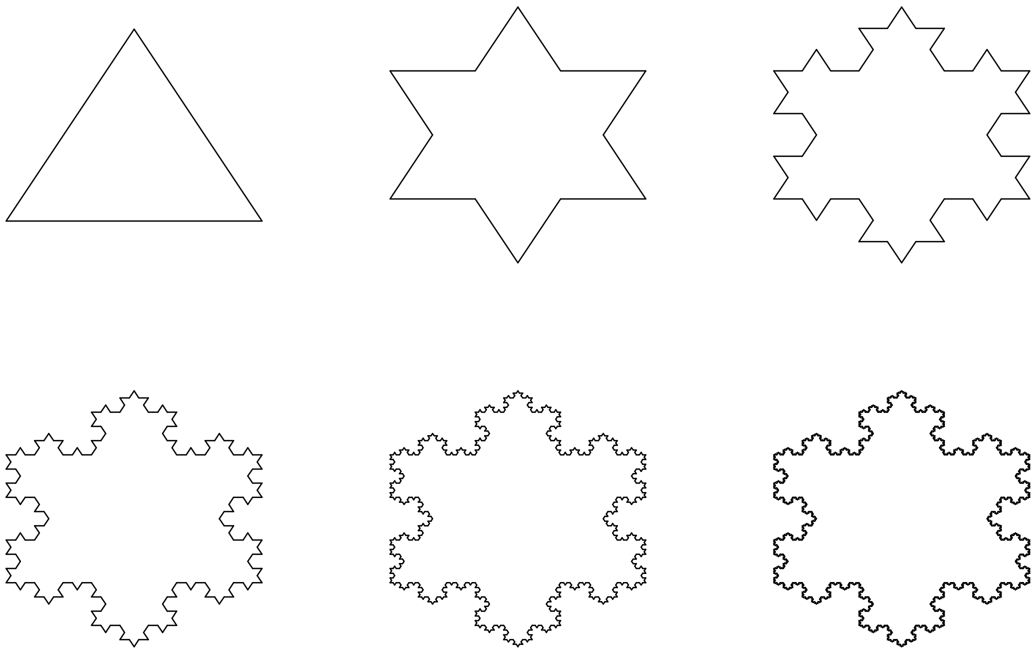
Often, one sees the curve $\mathcal{K}_\infty$ as one side of a closed plane figure, the Koch Snowflake (sometimes also called the "Koch Island"). This is an infinite length "curve" which bounds a finite area, and resembles a snowflake. It is made by sticking together three copies of $\mathcal{K}_\infty$ meeting each other at 60° angles so that they close up. We do this now:

---

[6] Also, symbolic systems such as maple often have additional means to speed up recursive procedures. If you expect your function to be called several times with the same argument, you can add `options remember` to the definition. This instructs maple to save the result of previous function calls. When the function is called again, maple just gives the same result as before, rather than computing it again. But be careful: you get the same answer, even if you change the function later.

```
> KochFlake := proc(n::nonnegint)
   ResetTurtle();
   SetTurtleAngle(60);
   cat(`L`,Koch(n),`RR`,Koch(n),`RR`,Koch(n));
  end:
```

```
> plots[display](array([[seq(TurtleCmd(KochFlake(i)),i=0..2)],
                         [seq(TurtleCmd(KochFlake(i)),i=3..5)]]));
```



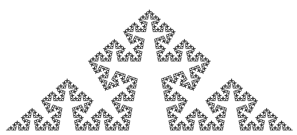
4.3.4 Some variations on the Koch curve

What happens to the curve $\mathcal{K}_\infty$ if we change the height of the bumps we add at each stage? We can do this easily by changing the angle the turtle turns when it encounters an R or an L. For example, an angle of $10°$ gives a curve which is nearly smooth.

```
> ResetTurtle(); SetTurtleAngle(10);
  TurtleCmd(Koch(6));
```

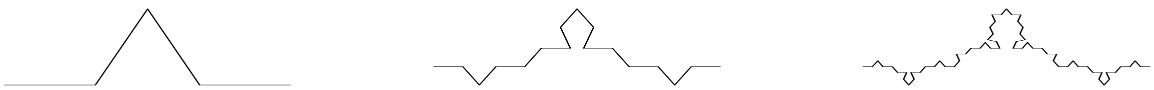
But an angle of $80°$ gives a curve which wiggles wildly and almost seems to take up area.

```
> ResetTurtle(); SetTurtleAngle(80);
  TurtleCmd(Koch(6));
```



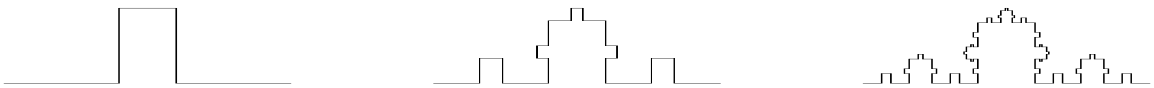
We shall make this observation more precise in section 4.5. You might try some additional experiments on your own, to gain intuition about how the limit curve depends on the parameters. For example, what would happen if you make the angle be 90°? Do you find the result surprising?

Some other variations you might want to try on your own might be to modify the `Koch` procedure so that the bumps are added alternately above and below the curve, as shown in the following figure:



Here the angle used was less than 60°. What happens for an angle of exactly 60°? How about a larger angle? Can you adjust things so that using 75° angle does not cause self-intersections?

Or, you might try using a different shape of a bump. For example, below we used squares, shrinking the scale first so the bumps don't run into each other. What happens if you vary the scale factor with `SetTurtleScale`?



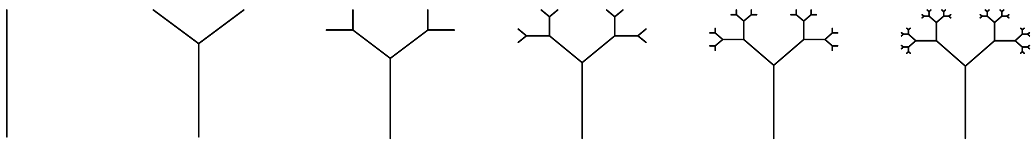
4.4 Making a tree

We now try to get our turtle to trace out a tree. In our tree, we start with a trunk, and then add two branches at the top. Then, at the end of each branch, we will add more branches.

First, we prefer our trees to grow up, so we change the turtle's initial heading. We also set our angle to be 45°.

```
> read(‘turtle.txt‘);
  SetInitialTurtleHeading(90);
  SetTurtleAngle(45);
```

The trunk of our tree will have no branching, and will just be the single command F. We will call this $\mathcal{T}_0$. For $\mathcal{T}_1$, we add a pair of branches to the top, to make a Y shape.[7] At each subsequent stage, we add another level of branches, as below:



How do we go about implementing this? As in the case of the Koch curve, we can use recursion, by noticing that $\mathcal{T}_n$ contains two small copies of $\mathcal{T}_{n-1}$ forming the left and right limbs

[7] As in §4.3, we will use a calligraphic font ($\mathcal{T}_n$) for the curve and ordinary roman (T_n) to denote the commands to produce the curve.

of the tree. Thus, we can define T_n recursively. Before proceeding, you might want to take a minute to think about how to do this on your own.

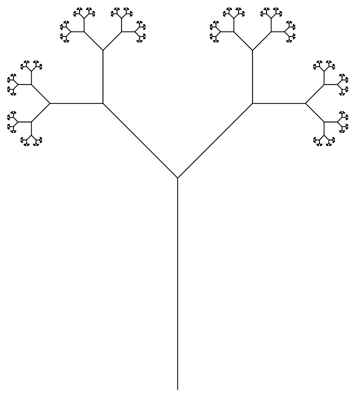
Notice that the letter Y cannot be drawn without either retracing part of the curve or lifting your pen. Rather than using a `PenUp` and `PenDown` command to tell the turtle to stop drawing,[8]we will have our turtle back up over a branch after we draw it. We also have to make sure our turtle is pointing in the right direction.

Thus, T_0 = FB, and we can take T_1 as "F S LFBR RFBL G B". The "LFBR" part goes out the left branch and returns, and "RFBL" goes out the right branch and returns. To get T_2, we replace each copy of "FB" in T_1 with a scaled version of all of T_1, after turning a bit to the left or right. In general, we have

$$T_n = \begin{cases} \mathrm{FSL}T_{n-1}\mathrm{RR}K_{n-1}\mathrm{LGB} & \text{if } n > 0 \\ \mathrm{FB} & \text{if } n = 0 \end{cases} .$$

Using this, it is now easy to write the generating procedure.

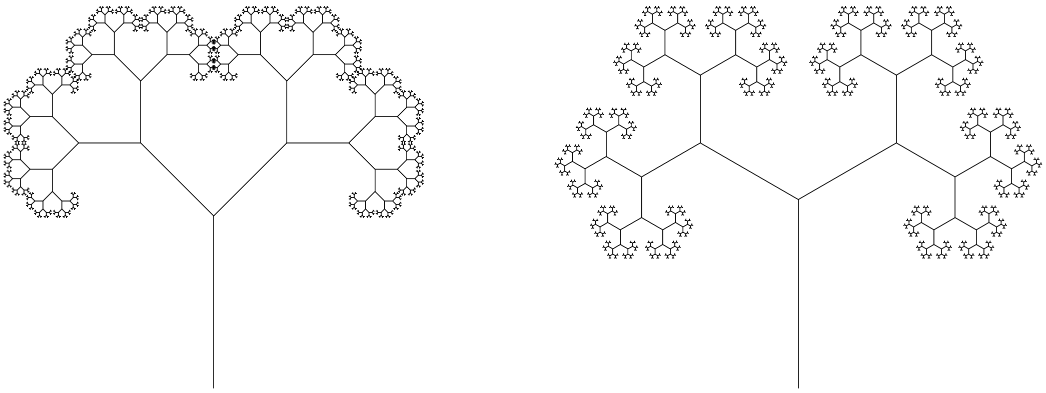
```
> Tree:= proc(n::nonneg)
    options remember;
    if (n=0) then
       `FB`;
    else
       cat(`FSL`, Tree(n-1), `RR`, Tree(n-1), `LGB`);
    fi;
  end:

> TurtleCmd(Tree(10));
```



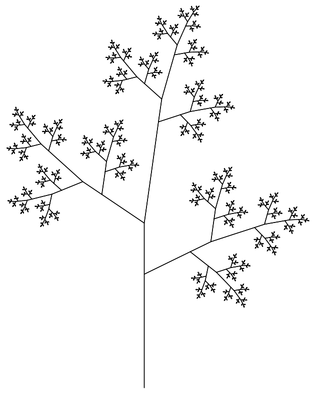
You might want to experiment with varying the angle and the scale factor, and see how the tree changes. For example, if we use `SetTurtleScale(.6)`, the branches of $\mathcal{T}_8$ just touch, and they overlap in $\mathcal{T}_n$ if $n > 8$. However, if we change the angle to 60°, the branches stay away from each other.

[8]The file turtle.txt does, in fact, implement these commands as D and U— see §4.9 for more details.

```
> ResetTurtle(); SetInitialTurtleHeading(90);
  SetTurtleAngle(45); SetTurtleScale(.6);
  t10a:=TurtleCmd(Tree(10)):
> SetTurtleAngle(60);
  t10b:=TurtleCmd(Tree(10)):
> plots[display](array([t10a, t10b]));
```



As an extra challenge, try your hand at producing a fern like the one below.



4.5 Fractal Dimension

From an early age, we that learn lines and curves are one-dimensional, planes and surfaces are two-dimensional, solids such as a cube are three dimensional, and so on. More formally, we say a set is n-dimensional if we need n independent variables to describe a neighborhood of any point. This notion of dimension is called the *topological dimension* of a set.[9] The dimension of the union of finitely many sets is the largest dimension of any one of them, so if we "grow hair" on a plane, the result is still a two-dimensional set. We should note here that if we take

[9] In fact, there is a much more precise definition of topological dimension, but to give it requires more background material than we are prepared to give here.

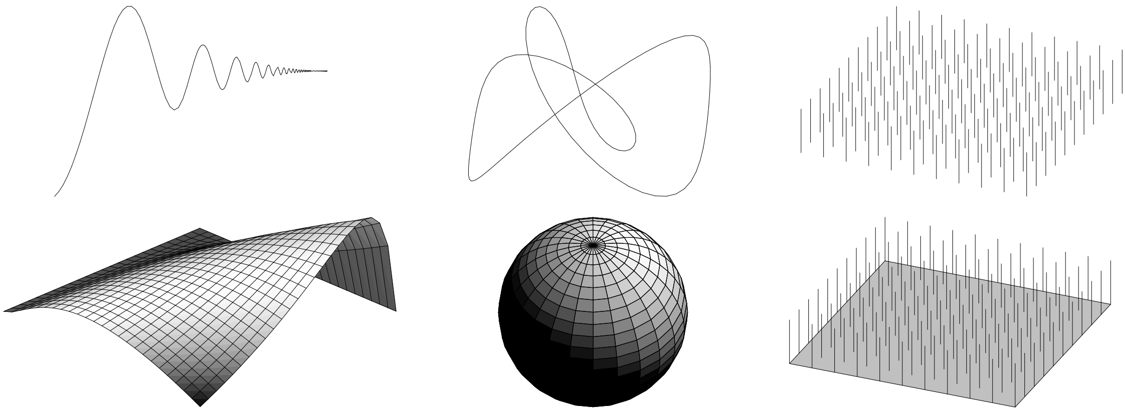


Figure 4.5.1: Some one- and two-dimensional sets (the sphere is hollow, not solid).

the union of an infinite collection of sets, the dimension can grow. For example, a line, which is one-dimensional, is the union of an infinite number of points, each of which is a zero-dimensional object.

There can occaisionally be a little confusion about the dimension of an object: sometimes people call a sphere a three-dimensional object, because it can only exist in space, not in the plane. However, a sphere is two-dimensional: any little piece of it looks like a piece of the plane $\mathbb{R}^2$, and in such a small piece, you only need two coordinates to describe the location of a point.[10]

But, what about the fractals we have been considering? For example, what is the dimension of the Koch snowflake? It has topological dimension one, but it is by no means a curve— the length of the "curve" between any two points on it is infinite. No small piece of it is line-like, but neither is it like a piece of the plane or any other $\mathbb{R}^n$. In some sense, we could say that it is too big to be thought of as a one-dimensional object, but too thin to be a two-dimensional object. Maybe its dimension should be a number *between* one and two. In order to make this kind of thinking more precise, let's look at the dimension of familiar objects another way.

4.5.1 Box counting dimension

What is the relationship between an object's length (or area or volume) and its diameter? The answer to this question leads to another way to think about dimension. Let us consider a few examples.

[10] In fact, this is just a different measure of dimension, called the *embedding dimension*: a set has embedding dimension n if n is the smallest integer for which it can be embedded into $\mathbb{R}^n$ without intersecting itself. Thus, the embedding dimension of a plane is 2, the embedding dimension of a sphere is 3, and the embedding dimension of a klein bottle is 4, even though they all have (topological) dimension two. A famous theorem (the Whitney embedding theorem) says that if a manifold has topological dimension n, its embedding dimension is at most $2n$.

If we try to cover the unit square with little squares of side length ϵ, how many will we need? Obviously, the answer is $1/\epsilon^2$. How about to cover a segment of length 1? Here we need only $1/\epsilon$ little squares. If we think of the square and segment as sitting in space and try to cover them with little cubes ϵ on a side, we get the same answer. And if we use the little cubes to cover a $1 \times 1 \times 1$ cube, how many will we need? Exactly $1/\epsilon^3$. Note that the exponent here is the same as the dimension of the thing we are trying to cover. This is no coincidence.

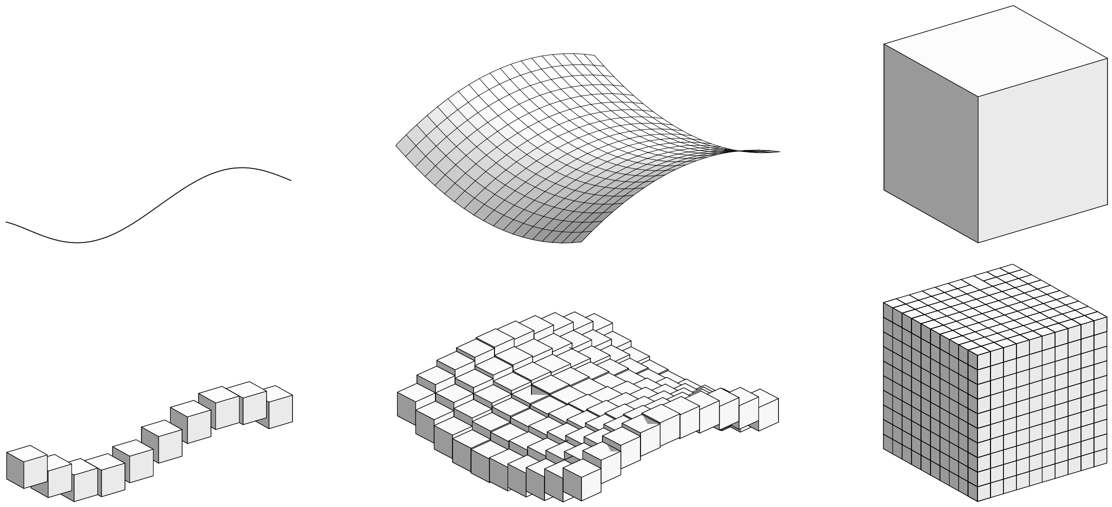


Figure 4.5.2: Covering a curve, a surface, and a solid cube with cubes of diameter ϵ.

We define the *box-counting dimension* (or just "box dimension") of a set $\mathcal{S}$ contained in $\mathbb{R}^n$ as follows: For any $\epsilon > 0$, let $N_\epsilon(\mathcal{S})$ be the minimum number of n-dimensional cubes of side-length ϵ needed to cover $\mathcal{S}$. If there is a number d so that

$$N_\epsilon(\mathcal{S}) \sim 1/\epsilon^d \qquad \text{as} \qquad \epsilon \to 0,$$

we say that the box-counting dimension of $\mathcal{S}$ is d. We will denote this by $\dim_{\square}(\mathcal{S}) = d$.

Note that the box-counting dimension is d if and only if there is some positive constant k so that

$$\lim_{\epsilon \to 0} \frac{N_\epsilon(\mathcal{S})}{1/\epsilon^d} = k.$$

Since both sides of the equation above are positive, it will still hold if we take the logarithm of both sides to obtain

$$\lim_{\epsilon \to 0} (\ln N_\epsilon(\mathcal{S}) + d \ln \epsilon) = \ln k.$$

Solving for d gives

$$d = \lim_{\epsilon \to 0} \frac{\ln k - N_\epsilon(\mathcal{S})}{\ln \epsilon} = -\lim_{\epsilon \to 0} \frac{N_\epsilon(\mathcal{S})}{\ln \epsilon}.$$

Note that the $\ln k$ term drops out, because it is constant while the denominator becomes infinite as $\epsilon \to 0$. Also, since $0 < \epsilon < 1$, $\ln \epsilon$ is negative, so d is positive as we would expect.

We should remark that there are some sets $\mathcal{S}$ for which $\dim_{\Box}(\mathcal{S})$ cannot be defined because there is no d for which the limit converges.[11] We will not encounter such examples here, however. Since the box-counting dimension is so often used to calculate the dimensions of fractal sets, it is sometimes referred to as "fractal dimension". We prefer the term box dimension, however, because sometimes the term "fractal dimension" might refer to box dimension, Hausdorff dimension, or even other measures of dimension such as the information dimension or capacity dimension.

When computing box dimension, several simplifications can be made.

- We need not use boxes if some other shape is more convenient: if we cover our set with disks of diameter ϵ, or even stars or mickey-mouses of diameter ϵ, we will get the same answer.

- Not every possible ϵ need be considered. It is enough to consider the limit of $\ln N_{\epsilon_i}/\ln \epsilon_i$ where ϵ_i is a sequence converging to zero. Choosing a convenient sequence often makes the calculations much easier.

Sometimes box counting dimension is referred to as "similarity dimension" in the context of self-similar sets. If a set is self-similar, there is an expansion factor r by which one can blow up a small copy to get the whole set. If there are exactly N such small copies that make up the entire set, the box dimension is easily seen to be $\ln N/\ln r$.

4.5.2 Computing the box dimension of some examples

Not surprisingly, the box dimensions of ordinary Euclidean objects such as points, curves, surfaces, and solids coincide with their topological dimensions of 0, 1, 2, and 3— this is, of course, what we would want to happen, and follows from our the discussion at the beginning of §4.5.1. But what about other, more complicated sets? Let's try a few simple examples for some subsets of the unit interval $[0, 1]$. In these cases, our ϵ-cubes can be closed intervals of length ϵ.

Consider the set of points $\mathcal{A} = \left\{0, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots\right\}$. For any $n \geq 0$, we can cover $\mathcal{A}$ with n intervals of length $1/2^n$: we need one for the elements between 0 and $\frac{1}{2^n}$, and another interval for each of the remaining $n-1$ elements of $\mathcal{A}$. This means that $\dim_{\Box}(\mathcal{A}) = \lim_{n\to\infty} n/2^n = 0$. The box dimension and the topological dimension of $\mathcal{A}$ are the same.

Let $\mathcal{Q}$ be the set of rational numbers in the interval $[0, 1]$, that is,

$$\mathcal{Q} = \left\{ \frac{p}{q} \,\middle|\, p, q \text{ are relatively prime integers with } p \leq q \right\}.$$

[11] Mathematicians often use a more general measure called *Hausdorff dimension*, which is defined for every such set. One difficulty with the Hausdorff dimension is that it is often very hard to compute. The Hausdorff and box dimensions coincide for compact, self-similar fractals, so we will not concern ourselves with the distinction.

What is $\dim_\square(\mathcal{Q})$? Since the rationals are dense in $[0,1]$, any interval we choose contains some. This means for every $\epsilon > 0$, we will need $1/\epsilon$ intervals to cover $\mathcal{Q}$. Thus, $N_\epsilon(\mathcal{Q}) = 1/\epsilon$. Consequently, $\dim_\square(\mathcal{Q}) = \lim_{\epsilon\to 0} \frac{1/\epsilon}{1/\epsilon} = 1$.

Recall that any real number x can be represented as a (possibly infinitely long) decimal. For example, $1/4 = .25$, $1/11 = .0101\overline{01}\ldots$, and $\pi = 3.14159265\ldots$. The decimal expansion is not quite unique— for example, $1/2$ can be written as either $.4999999\overline{9}\ldots$ or $.5000000\overline{0}\ldots$. This is the only possible point of confusion, however: any real number x ending in all zeros has another representation ending in all nines.

Let us determine the box-counting dimension of the set

$$\mathcal{D} = \{x \in [0,1] \,|\, x \text{ has a decimal expansion containing no 4s or 5s}\}.$$

First, let's note a few things about $\mathcal{D}$:

- It is *totally disconnected*: that is, between any two points of $\mathcal{D}$, there is another point which is not in $\mathcal{D}$.
- It is *closed*: the limit of any convergent sequence of points in $\mathcal{D}$ is still in $\mathcal{D}$. Note that this is not the case for the set $\mathcal{Q}$ above— a sequence of rational numbers can converge to an irrational.
- It has uncountably many points. Much like the irrational numbers, there are far too many elements of $\mathcal{D}$ to enumerate them. Also like the irrational numbers, we *can* enumerate what is not in $\mathcal{D}$.
- We shall see that $\mathcal{D}$ is also self-similar: any small piece of it can be scaled up to look like the whole thing just by multiplying by an appropriate power of 10.

So, what is $\dim_\square(\mathcal{D})$? Let's construct a sequence of covers of $\mathcal{D}$ whose diameter tends to zero, and count the number of pieces we need. Note that while $\mathcal{D}$ contains the points[12] .4 and .6, it does not contain the open interval $(.4, .6)$, so we can cover it by two intervals of length .4, namely $[0, .4]$ and $[.6, 1]$. This means $N_{.4} = 2$.

At the next finer level, we can see that $\mathcal{D}$ can be covered with the intervals

$$\begin{array}{cccccccc} [0,.04], & [.06,.10], & [.10,.14], & [.16,.20], & [.20,.24], & [.26,.30], & [.30,.34], & [.36,.40], \\ [.60,.64], & [.66,.70], & [.70,.74], & [.76,.80], & [.80,.84], & [.86,.90], & [.90,.94], & [.96,1] \end{array}$$

That is, we need 16 intervals of length .04. This means that $N_{.04} = 16$.

For $\epsilon = .004$, we will need 8 times as many intervals to cover $\mathcal{D}$. This pattern continues: each time we shrink ϵ by another factor of 10, we need 8 times as many intervals.

[12] $\mathcal{D}$ contains .4 because it can be written as $0.399\overline{9}\ldots$, which contains no 4s or 5s.

This means that

$$\dim_{\square}(\mathcal{D}) = -\lim_{n\to\infty}\frac{\ln 8^{n-1}}{\ln(4\cdot 10^{-n})} = -\lim_{n\to\infty}\frac{(n-1)\ln 8}{\ln 4 - n\ln 10} = \frac{\ln 8}{\ln 10} = \frac{3\ln 2}{\ln 10} \approx .903089987$$

The box-counting dimension of $\mathcal{D}$ is not an integer. Note that the topological dimension of $\mathcal{D}$ is zero.

4.6 Cantor sets

The set $\mathcal{D}$ of the previous section is an example of a Cantor set. Often, a Cantor set is described as the result of an iterative process, much in the same way we described the Koch curve in §4.2. We'll describe the most common Cantor set, the "middle-thirds Cantor set", this way now.

Begin with the interval $[0,1]$ as the set $\mathcal{C}_0$. Now remove the open interval $(\frac{1}{3},\frac{2}{3})$ from $\mathcal{C}_0$ to obtain

$$\mathcal{C}_1 = \left\{\left[0,\frac{1}{3}\right], \left[\frac{2}{3},1\right]\right\}.$$

To get the second stage, we remove the middle third of each interval in $\mathcal{C}_1$. This gives

$$\mathcal{C}_2 = \left\{\left[0,\frac{1}{9}\right], \left[\frac{2}{9},\frac{1}{3}\right], \left[\frac{2}{3},\frac{7}{9}\right], \left[\frac{8}{9},1\right]\right\}.$$

We continue in this way, removing the middle third of each interval in $\mathcal{C}_n$ to obtain $\mathcal{C}_{n+1}$ (See Fig. 4.6.3). Note that $\mathcal{C}_n \subset \mathcal{C}_m$ for all $n > m$. What is left as we let $n \to \infty$ is the Cantor set. That is,

$$\mathcal{C} = \bigcap_{n=0}^{\infty} \mathcal{C}_n.$$

Figure 4.6.3: The first few stages $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_5$ in the construction of the middle-thirds Cantor set $\mathcal{C}$.

It may not be clear at first that $\mathcal{C}$ even exists. It seems we are taking just about everything out when we remove the intervals. Indeed, the total length of intervals we remove is

$$\frac{1}{3} + \frac{2}{9} + \frac{4}{27} + \frac{8}{81} + \ldots = 1.$$

That is, the total length of the segments removed is equal to the segment we started with! For this reason, a set like $\mathcal{C}$ is sometimes called a "Cantor dust". But notice that the endpoints of each interval of $\mathcal{C}_n$ must be in $\mathcal{C}$, because they never get removed at any level. Thus, $\mathcal{C}$ is nonempty, since it contains at least the points $\left\{0, 1, \frac{1}{3}, \frac{2}{3}, \frac{1}{9}, \frac{2}{9}, \frac{7}{9}, \frac{8}{9}, \frac{1}{27}, \frac{2}{27}, \ldots\right\}$. And it contains quite a few more points, as well. To see that, we can view $\mathcal{C}$ in a manner similar to the description of the set $\mathcal{D}$ given in §4.5.2. However, $\mathcal{C}$ is more naturally described in base 3, rather than base 10. Let's briefly review what we mean by this.

When we express a number x in base 10, we are expressing it as sums of powers of ten. Thus, when we write $e^5 \approx 148.413$, we are actually saying that

$$e^5 \approx 1 \cdot 10^2 + 4 \cdot 10^1 + 8 \cdot 10^0 + 4 \cdot 10^{-1} + 1 \cdot 10^{-2} + 3 \cdot 10^{-3}.$$

There is nothing magic about powers of 10, of course (except that it is the base we learned as children). For example, we could express this[13] in ternary (that is, base 3), where we would have

$$e^5 \approx 12111.102011_3 = 1{\cdot}3^4{+}2{\cdot}3^3{+}1{\cdot}3^2{+}1{\cdot}3^1{+}1{\cdot}3^0{+}1{\cdot}3^{-1}{+}0{\cdot}3^{-2}{+}2{\cdot}3^{-3}{+}0{\cdot}3^{-4}{+}1{\cdot}3^{-5}{+}1{\cdot}3^{-6}.$$

In computer applications, the bases 16 (hexadecimal) and 2 (binary) are very common, because the computer represents everything internally in binary.[14]

Returning to the Cantor set $\mathcal{C}$, note that when we remove the interval $(1/3,\, 2/3)$, we are removing all the numbers whose ternary expansion has a 1 immediately after the decimal place.[15] At the next stage, we remove those points which have a 1 in the second place after the decimal, and so on. Thus, in the limit, we obtain the following alternate description of $\mathcal{C}$:

$$\mathcal{C} = \{x \in [0,\, 1] \mid x \text{ has a ternary expansion containing no 1s}\}.$$

You may remember that rational numbers correspond to numbers with a decimal expansion that eventually repeats; the same holds true of the expansion in any base (how might you prove this if you assume it for decimals?). It is easy to see using this description that $\mathcal{C}$ is closed (that is, it contains all its limit points), totally disconnected, and contains an uncountable number of elements (there are a *lot* of sequences of 0s and 2s), just like the set $\mathcal{D}$ of the previous section. It is also self-similar, because if we take any small section of it and expand it by an appropriate power of 3, it looks like the entire set.

[13] Maple will convert integers to other bases, using a command such as `convert(148, base, 3)`. One has to work a little bit to convert fractions— can you think of a way to do this?

[14] Hexadecimal is convenient for use with a binary system, because we can group blocks of 4 binary digits to obtain one hexadecimal digit (by convention, the letters A–F are used to represent the integers 10–15). This is quite handy in computer applications, because a single byte is represented by 8 binary digits, or two hex digits. Before 8-bit bytes became standard, octal (base 8) was very common as well. Octal is much less convenient for 8-bit quantities, however, because 8 bits doesn't break up nicely into groups of 3 bits.

[15] As in decimal expansions, some points have two representations. Thus, the point 1/3 can be written either as $.1_3$ or as $.0222\overline{2}\ldots$. As in the construction of $\mathcal{D}$ in §4.5.2, we keep such points in our set.

What is its box dimension? The description by the sets $\mathcal{C}_n$ give us an effective way to calculate the limit: at the n^{th} stage, we have 2^n intervals of length 3^{-n}. Using this, we can immediately calculate that

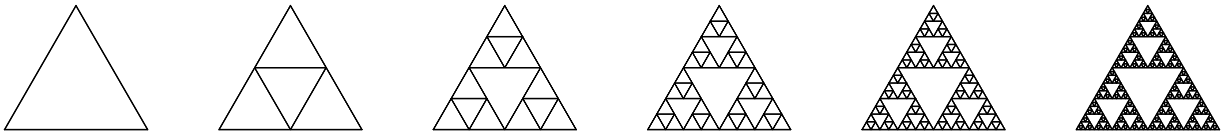
$$\dim_{\Box}(\mathcal{C}) = \lim_{n\to\infty} \frac{\ln 2^n}{\ln 3^n} = \frac{\ln 2}{\ln 3} \approx 0.63093.$$

You may have noticed the strong similarity in the construction of the Koch curve $\mathcal{K}$ of §4.2 and the middle-thirds Cantor set $\mathcal{C}$. In fact, notice that the intersection of $\mathcal{K}$ with its base (the initial segment $\mathcal{K}_0$) is exactly the middle-thirds Cantor set.

4.7 The Sierpinski gasket

Another very commonly encountered fractal is the Sierpinski gasket, which can be described as follows:

We start with an equilateral triangle $\mathcal{S}_0$, and replace it by three equilateral triangles with a base half the size of the original, stacked so the original perimiter is kept, but leaving a hole in the center. We then replace each of those triangles by three more triangles, to obtain $\mathcal{S}_2$ as nine triangles, each with a base of length 1/4. Continuing in this fashion infinitely many times yields the Sierpinski gasket $\mathcal{S}$. The sets $\mathcal{S}_0$ through $\mathcal{S}_6$ are shown below.



It is easy to see that $\dim_{\Box}(\mathcal{S})$ is $\ln 3/\ln 2$, or about 1.58. You should try to confirm this fact for yourself. The self-similarity of the Sierpinski gasket should help this calculation considerably.

How might we coerce our turtle into making a Sierpinski gasket? While the procedure is simple enough to describe, explicitly writing it down in symbols requires a bit of thought. Give it a try before continuing.

One way to accomplish this[16] involves taking a slightly different viewpoint. Rather than making the gasket by replacing triangles with smaller triangles, we construct a fractal "curve" that traverses the base and the perimeters of the "holes". This traces out the same sets $\mathcal{S}_n$, except for two segments which we can easily add later.

We view S_0 as the straight horizontal segment F, along with two other segments which are not horizontal. To emphasize the difference between the horizontal and the non-horizontal segments, we will traverse the non-horizontal segments backwards, and use the command B. Thus, our first triangle is $S_0 =$ FRBLLB.

[16]We will give another solution which is closer to our original description following §4.9.

Now, to obtain S_1 from S_0, we replace the horizontal segment F by two smaller ones with an inverted triangle between them. As with S_0, we use F to denote the horizontal parts and B for the non-horizontal portions. Thus, we apply the transformation F $\mapsto$ SF RBLFLBR FG. This gives us S_1 from S_0 as the command "SF RBLFLBR FG RBLLB". If we replace all three Fs in S_1 using this transformation, we will obtain S_2.

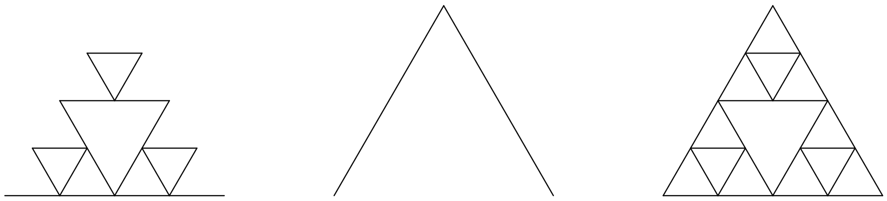


Figure 4.7.4: The sets produced by the turtle commands for s_2, h, and S_2 in the construction of the Sierpinski gasket.

Before we try to describe any of the S_n recursively, note that in this description, each one really consists of two pieces: there is the complicated, recursive portion which describes most of the gasket, and then there is the final RBLLB which makes the outer "hat" (see figure 4.7.4). Let's call the commands to produce this first part s_n and call commands for the hat h, that is, $S_n = s_n\, h$. Then we can describe S_n as follows

$$S_n = s_n\, h \qquad s_n = \begin{cases} \mathrm{S}s_{n-1}\mathrm{RBL}s_{n-1}\mathrm{LBR}s_{n-1}\mathrm{G} & \text{if } n > 0 \\ \mathrm{F} & \text{if } n = 0 \end{cases} \qquad h = \mathrm{RBLLB}.$$

As in the previous sections, once we have worked out the algorithm, writing the maple code to implement it is quite straightforward. We we will use the name `Sierp` for the procedure that produces s_n, and `Sierpinski` for the procedure to produce S_n.

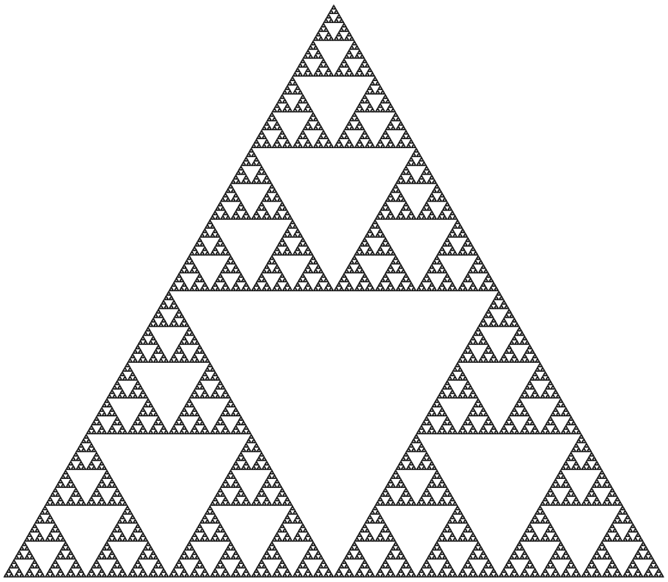
```
> Sierp:=proc(n::nonnegint)
    if (n=0) then
      `F`;
    else
      cat(`S`,Sierp(n-1), `RBL`, Sierp(n-1), `LBR`, Sierp(n-1), `G`);
    fi;
  end:
```

```
> Sierpinski:= n -> cat( Sierp(n), `RBLLB`):
```

Since we also need to set the angle and scale appropriately, we can combine them all into a single command `DrawSierp`:

```
> DrawSierp:= proc(n::nonnegint)
     SetTurtleAngle(60);
     SetInitTurtleHeading(0);
     SetTurtleScale(.5);
     TurtleCmd(Sierpinski(n));
  end:
```

```
> DrawSierp(9);
```



Warning: Drawing even $\mathcal{S}_8$ takes a significant amount of time and is almost indistinguishable from $\mathcal{S}_7$. If you try to draw $\mathcal{S}_9$, be prepared to wait quite a while, and make sure your computer has plenty of memory.

4.8 Inside the turtle's shell

We have how to use the turtle package in several ways; now it is time to look at how the package itself is implemented. This was done in several layers

4.9 Extending the turtle's commands