Fractal Geometry

Mark McClure

Contents

Illustrations

1

Introduction

1.1 The idea behind fractal geometry

The term *fractal* was coined around 1975 by Benoit Mandelbrot to intuitively describe an object as complicated, rough, or fractured. Mandelbrot was inspired by objects in nature like the shape of a cloud or the path of a river, but his ideas had many precursors in pure mathematics. In this book, we study fractals as they arise in pure mathematics, focusing on the computational aspects of the subject.

Figure 1.1 suggests the flavor of the subject.

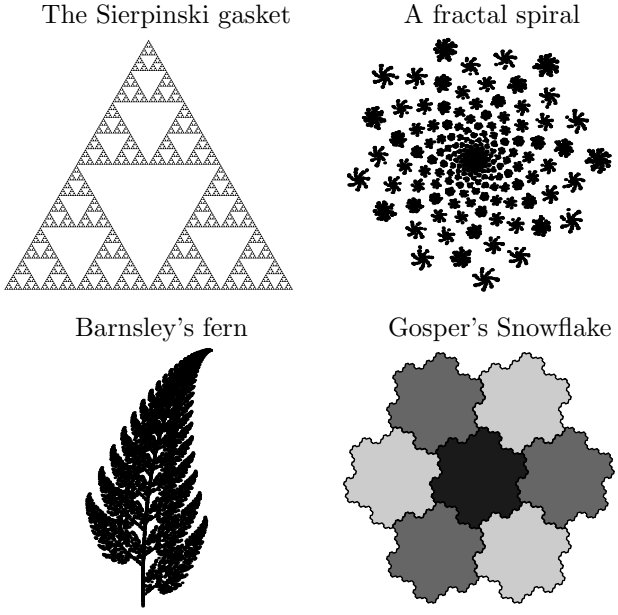


Figure 1.1 Some fractal sets

While different in appearance, these objects all have features in common with most of the objects considered in this book. First of all, they are all sets of points in the plane generated by some mathematical process. More importantly, they all have detailed structure at infinitely many levels of magnification. This is the central idea in fractal geometry from Mandelbrot's perspective. The basic objects of Euclidean geometry don't have this property. When modelling a natural object (say the coastline of an island), with Euclidean objects (say a collection of line segments), the model loses much of the structure when examined on a fine scale. That is, if we zoom in too much, we see the straight line segments and the physical nature of the picture is lost. While the pictures in figure 1.2 are only finite approximations to fractal sets, they are generated by algorithms that can, in principle, be used to generate detail on as fine a level as desired. This idea is illustrated using the Barnsley fern in figure 1.2. The outlined region has been magnified to reveal a branch with a similar level of detail as the whole. Off of this branch are other branches, again with a similar level of detail.

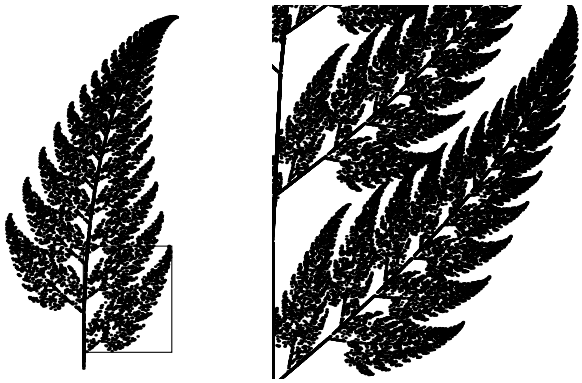


Figure 1.2 The Barnsley Fern with detail

Our zoom into Barnsley's fern reveals another important aspect of fractal geometry - the sets all display some degree of self-similarity. This self-similarity is most clearly seen in the Sierpinski gasket, which is composed of three smaller copies of itself. In fact, this strict self-similarity of the Sierpinski gasket implies the fact that it has structure on all levels of magnification, since each of the three smaller copies of the set are in turn composed of three smaller copies each, etc. Gosper's snowflake is also strictly self-similar, as it is composed of seven smaller copies of itself. It is the boundary of that set which is truly of interest and its analysis is a bit more involved.

This book is divided roughly into two parts - chapters 2 through 4 on general theory and subsequent chapters on applications to other parts of pure mathematics. In chapter 2, we examine the idea of self-similarity from an algorithmic perspective, describing how to generate self-similar sets using iterated function systems. Chapter 3 takes a closer look at the mathematics lurking in the background of chapter 2 and introduces the concept of fractal dimension, which is a quantitative measure of the size of a set. When the fractal dimension is not an integer, this might indicate some degree of complexity in the set. In chapter 4, we generalize the notion of self-similarity to include somewhat more complicated objects. Later chapters examine how fractal geometry can be applied to graphs of functions, tilings of the plane, and physical problems. There are also several appendices which cover background material.

1.2 Notes

The most famous book on fractal geometry is undoubtedly Mandelbrot (1982). In this beautifully illustrated book, he lays down the argument that his techniques are the most natural ones to describe a large variety of natural objects. The foundations of fractal geometry were laid down in pure mathematics long before Mandelbrot, however. The prototypical fractal is certainly the Cantor set, which dates back to the 1880's and is described in the next chapter. Most mathematicians of Cantor's time found his set to "pathological"; it seemed to be the exception rather than the rule. The fractal perspective has made Cantor's set seem quite natural, however. This was made possible by the work of many mathematicians throughout the early twentieth century, most notably Hausdorff, Besicovitch and his students. Some of the seminal works on fractal geometry are collected in Edgar (1993).

There are now several excellent books on fractal geometry from the mathematical perspective. For the collegiate math major who would like to gain a deep understanding of fractal dimension, I would suggest Edgar (2009). For another excellent and broader view of fractal geometry there is Falconer (2003). Both these authors have more advanced texts, namely Edgar (1998) and Falconer (1997). Another text focusing on iterated function systems is Barnsley (1993).

2

Self-similarity

In this chapter, we describe the simplest types of fractal objects - self-similar sets. An intuitive definition of a self-similar set is one that is composed of smaller copies of itself. The prototypical example of such a set is the Cantor set.

2.1 The Cantor set

Cantor constructed his set in the 1880's to help him understand a problem in Fourier series. While the set seemed unnatural to mathematicians of the time, it has become a central example in real analysis. Cantor's construction is as follows. Start with the unit interval $I = [0, 1]$, the set of all real numbers between 0 and 1 inclusive. Remove the open middle third $\left(\frac{1}{3}, \frac{2}{3}\right)$ of the interval I to obtain the two intervals $I_1 = \left[0, \frac{1}{3}\right]$ and $I_2 = \left[\frac{2}{3}, 1\right]$. Then remove the open middle thirds of the intervals I_1 and I_2 to obtain the intervals $I_{1,1} = \left[0, \frac{1}{9}\right]$, $I_{1,2} = \left[\frac{2}{9}, \frac{1}{3}\right]$, $I_{2,1} = \left[\frac{2}{3}, \frac{7}{9}\right]$, and $I_{2,2} = \left[\frac{8}{9}, 1\right]$. Repeating this process inductively, we obtain 2^n intervals of length $1/3^n$ at the n^{th} stage. The cantor set C consists of all those points in I which are never removed at any stage. More precisely, if C_n denotes the union of all of the intervals left after the n^{th} stage of the construction, then

$$C = \bigcap_{n=1}^{\infty} C_n.$$

This process is illustrated in figure 2.1.

It's clear that C should be self-similar, since the effect of the construction on the intervals I_1 and I_2 is the same as the effect on the whole interval I, but on a smaller scale. Thus C consists of two copies of itself scaled by the factor 1/3.

Figure 2.1 Construction of the Cantor set

The Cantor set has many non-intuitive properties. In some sense, it seems very small; if we were to assign a "length" to it, that length would have to be zero. Indeed, by it's very construction it is contained in 2^n intervals of length $1/3^n$. Thus the length of C_n is $2^n/3^n$ which tends to zero as $n \to \infty$. Since C is contained in C_n for all n, the length of C must be zero. It might even appear that there is nothing left in C after tossing so much out of the original interval I. In reality, the Cantor set is a very rich set with infinitely many points. Recall that only open intervals are removed during the construction. Thus all of the infinitely many endpoints remain. For example, 1/3, 2/3, and 80/81 are all in C. There are still many more points in C, however.

There is a general technique for finding points of the Cantor set. The first stage in the construction consists of the two intervals I_1 and I_2. Choose one and discard the other. Now the interval we chose, say I_1 for concreteness, contains two disjoint intervals, $I_{1,1}$ and $I_{1,2}$, in the next stage of the construction. Choose one of those and discard the other. If we continue this process inductively, we obtain a nested sequence of closed intervals which will collapse down to a point in the Cantor set. For example, we might have chosen the interval I_1 at the first stage. Then we could have chosen the interval $I_{1,2}$ at the next stage. We might then choose to alternate between the first or second sub-interval at any point generating intervals of the form $I_{1,2,1,2,\ldots 1,2}$. These intervals collapse down to a single point which is not the endpoint of any removed interval.

The process for finding points in C constructs a one to one correspondence with the set of infinite sequences of 1s and 2s. The sequence corresponding to a particular point in C might be called the *address* of that point. As we will see, this addressing scheme can be generalized to other situations and pro-

vides a powerful tool for understanding self-similar sets. Note, for example, that the addressing scheme implies that the Cantor set is uncountable.

A major question that we will address later in the book asks, "What is the dimension of the Cantor set?" Certainly, it is too small to be considered a one dimensional set; it is just a scattering of points along the unit interval with length zero. It is uncountable, however; perhaps it is too large to be considered as zero dimensional. We will develop a notion of "fractal dimension" that quantitatively captures this in-betweeness.

2.2 The Sierpinski gasket

A similar process can be used to construct the Sierpinski gasket, also called the Sierpinski triangle. We start with a closed, filled in equilateral triangle. The line segments joining the midpoints of the sides of this triangle divide it into four equilateral sub-triangles. We can discard the one in the center, keep the others and then repeat the process on the remaining triangles. This process is illustrated in figure 2.2.

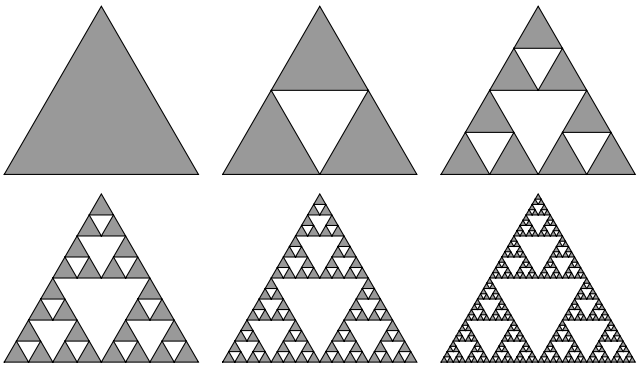


Figure 2.2 Construction of the Sierpinski gasket

Many of the comments regarding the Cantor set are applicable to the Sierpinski gasket as well. It is a self-similar set consisting of 3 copies of itself, each scaled by the factor 1/2. Its dimensional properties are, in a sense, between dimension one and dimension two.

2.3 Iterated function systems

We now enter the careful, mathematical portion of the text by defining one of the central tools of fractal geometry - the *iterated function system* or

IFS. The idea behind the IFS technique is to focus on how the parts of a set might fit together to create the whole. Thus, we focus on what makes two sets similar; we do this with the notion of a similarity. An IFS is a collection of similarities and a self-similar set is the attractor of an IFS. Thus, we actually need to make several definitions, which we will illustrate with examples afterwards.

We assume we are working in $\mathbb{R}^n$, n-dimensional Euclidean space. The dimension n will almost always be 2 in this book, although $n = 1$ might be the natural setting for the Cantor set. Our sets will generally be subsets of the plane. Given x in $\mathbb{R}^n$, $|x|$ will refer to the distance from x to the origin. Thus, $|x - y|$ represents the distance from x to y. A function $f : \mathbb{R}^n \to \mathbb{R}^n$ will be called a *contraction* if there is a real number r such that $0 < r < 1$ and $|f(x) - f(y)| \leq r|x - y|$ for all x, y in $\mathbb{R}^n$. If $|f(x) - f(y)| = r|x - y|$ for all x, y in $\mathbb{R}^n$, then f is called a *similarity* and the number r is called its *contraction ratio*. An iterated function system on $\mathbb{R}^n$ is simply a non-empty, finite collection of contractions of $\mathbb{R}^n$. We will usually express an IFS in the form $\{f_i\}_{i=1}^m$ where m is the number of functions in the IFS. If $\{f_i\}_{i=1}^m$ is an IFS of similarities, the list $\{r_i\}_{i=1}^m$ of corresponding contraction ratios is called the *contraction ratio list* of the IFS.

As we will see, associated with any IFS there is always a unique non-empty, closed, bounded subset E of $\mathbb{R}^n$ satisfying

$$E = \bigcup_{i=1}^{m} f_i(E) \tag{2.1}$$

The set E defined in equation 2.1 is called the *invariant set* or *attractor* of the IFS. If the IFS consists entirely of contractive similarities, then E is called *self-similar*.

We will make frequent use of equation 2.1, so let's make sure we understand what it is saying. First, the notation $f_i(E)$ refers to the image of the set E under the action of the function f_i. For the general function $f : \mathbb{R}^n \to \mathbb{R}^n$ and set $E \subset \mathbb{R}^n$,

$$f(E) = \{f(x) : x \in E\}.$$

If the function f happens to be a similarity, then application of f transforms E into a set which is geometrically similar to E. This idea is illustrated in figure 2.3.

Let us examine several examples of iterated function systems. Given a self-similar set, there are always many iterated function systems which generate

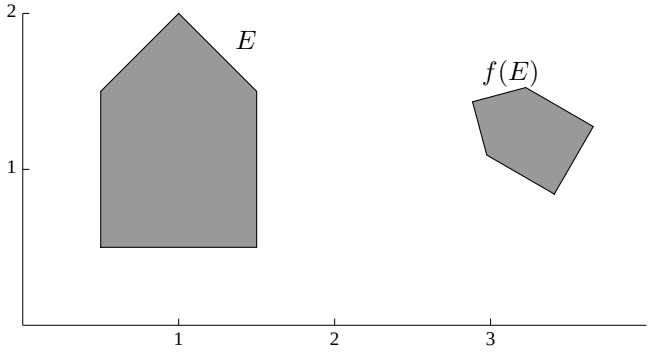


Figure 2.3 The action of a contractive similarity on a set E in the plane.

that set. We would usually like to find a particularly simple IFS to generate a given set. The Cantor set C is a subset of the real line which consists of two copies of itself each scaled by the factor $\frac{1}{3}$. Therefore, it is most easily described using an iterated function system consisting of two similarities $\{f_1, f_2\}$ mapping $\mathbb{R}$ to $\mathbb{R}$. One way to choose the functions is $f_1(x) = \frac{1}{3}x$ and $f_2(x) = \frac{1}{3}x + \frac{2}{3}$. Then $f_1(C) = C \cap I_1$ and $f_2(C) = C \cap I_2$, where I_1 and I_2 are the intervals from the first stage of construction of C.

As most of our examples will be subsets of the plane, we need a convenient notation for describing similarity transformations of $\mathbb{R}^2$. Linear algebra provides such notation. The general similarity transformation $f : \mathbb{R}^2 \to \mathbb{R}^2$ can be expressed in the form

$$f(x) = Mx + b,$$

where M is a 2-dimensional matrix, b is a translation vector, and M acts on x by matrix multiplication. Not all matrices lead to similarity transformations, but any similarity transformation can be expressed this way. For example, we can express a contraction about the origin with contraction factor $1/2$ using the matrix

$$M = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}. \tag{2.2}$$

Armed with this notation, we can attempt to find an IFS for the Sierpinski gasket. Recall that this set consists of three copies of itself scaled without rotation by the factor $1/2$. Thus we should be able to express an IFS for the gasket using the matrix M above together with shift vectors. Here is an IFS for the gasket using the matrix M defined by formula 2.2:

$$f_1(x) = Mx$$
$$f_2(x) = Mx + \begin{pmatrix} 1/2 \\ 0 \end{pmatrix}$$
$$f_3(x) = Mx + \begin{pmatrix} 1/4 \\ \sqrt{3}/4 \end{pmatrix}$$

The function f_1 maps the gasket onto the lower left copy of itself, f_2 maps it onto the lower right, and f_3 maps it onto the upper copy.

Our next example is called the Koch curve. Figure 2.4 shows how the Koch curve, denoted K, is composed of 4 copies of itself scaled by the factor 1/3.

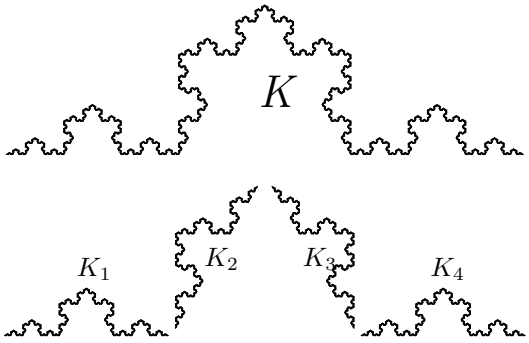


Figure 2.4 The Koch curve

The IFS for K is more complicated than the IFS for the Sierpinski gasket, since it involves rotation. Rotation through the angle θ about the origin can be represented by the matrix

$$R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \tag{2.3}$$

Thus an IFS for K is

$$\begin{aligned} f_1(x) &= \tfrac{1}{3}x \\ f_2(x) &= \tfrac{1}{3}R\left(\tfrac{\pi}{3}\right)x + \begin{pmatrix} 1/3 \\ 0 \end{pmatrix} \\ f_3(x) &= \tfrac{1}{3}R\left(-\tfrac{\pi}{3}\right)x + \begin{pmatrix} 1/2 \\ \sqrt{3}/6 \end{pmatrix} \\ f_4(x) &= \tfrac{1}{3}x + \begin{pmatrix} 2/3 \\ 0 \end{pmatrix} \end{aligned} \tag{2.4}$$

Each function f_i maps K onto the sub-portion labeled K_i in figure 2.4.

We can add reflection to the Koch curve example to generate the image in figure 2.5.

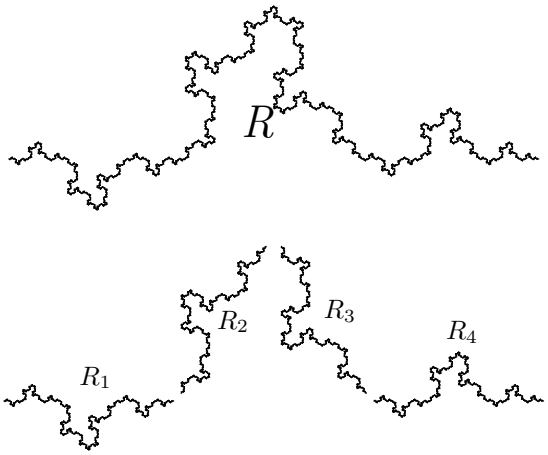


Figure 2.5 A Koch type curve with reflection

In this example, the similarities mapping the whole curve R to the portions labeled R_1 and R_3 both involve reflection. Reflection about the x or y axes can be represented using the matrices

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \text{ or } \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

respectively. Reflection about an arbitrary line through the origin can be achieved by first rotating, then reflecting, and rotating back. Thus an IFS for R is

$$\begin{aligned} f_1(x) &= \tfrac{1}{3}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} x \\ f_2(x) &= \tfrac{1}{3} R\left(\tfrac{\pi}{3}\right) x + \begin{pmatrix} 1/3 \\ 0 \end{pmatrix} \\ f_3(x) &= \tfrac{1}{3} R\left(-\tfrac{\pi}{3}\right) \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} x + \begin{pmatrix} 1/2 \\ \sqrt{3}/6 \end{pmatrix} \\ f_4(x) &= \tfrac{1}{3} x + \begin{pmatrix} 2/3 \\ 0 \end{pmatrix} \end{aligned} \tag{2.5}$$

2.4 Applying iterated function systems

We turn now to the question of how to generate self-similar images. Ultimately, we will consider several algorithms for this purpose. It turns out they are all related to theoretical questions presented in the next chapter. For example, we will present a constructive proof that any IFS yields a unique closed, bounded invariant set. This construction is essentially our first algorithm, which we call the basic deterministic algorithm.

Given an iterated function system $\{f_i\}_{i=1}^m$, we can define a function T which maps the collection of closed, bounded subsets of $\mathbb{R}^n$ to itself by

$$T(F) = \bigcup_{i=1}^{m} f_i(F). \tag{2.6}$$

We can use the function T to generate the invariant set using *iteration*, an important theme in fractal geometry. To iterate a function f which maps a set X into itself, start with a some point x_0 in X, set $x_1 = f(x_0)$, and for larger natural numbers n set $x_n = f(x_{n-1})$. Equivalently, we can write $x_n = f^n(x_0)$, the result of n-fold composition of f applied to x_0. In the current situation, the set X is the set of all non-empty, closed, bounded subsets of $\mathbb{R}^n$ and the function is the transformation T defined in equation 2.6. It turns out that if we start with an arbitrary non-empty, closed, bounded subset of $\mathbb{R}^n$ and iterate the function T, then the generated sequence of sets converges in a natural sense to the invariant set of the IFS. We'll examine this statement more carefully in the next chapter, but in this chapter we'll demonstrate the statement visually through experimentation.

Figure 2.2, for example, was generated in exactly the manner described above. The initial set F was taken to be the solid equilateral triangle with vertices $(0,0), (1,0),$ and $\left(1/2, \sqrt{3}/2\right)$. The transformation T is obtained

by applying equation 2.6 to the Sierpinski IFS. The subsequent images correspond to application of T up to 5 times.

The previous example might be misleading, since the relationship between the initial approximation and the fractal attractor is so close. Indeed, the choice of initial approximation has no effect on the final attractor, although the images illustrating convergence can be affected. Figure 2.6, for example, illustrates the sequence of sets generated by applying this same IFS to the initial approximation consisting of the single point at the origin. Successive approximations fill the set out more and more completely. Figure 2.7 illustrates the algorithm taking the initial set to be the shape from figure 2.3.

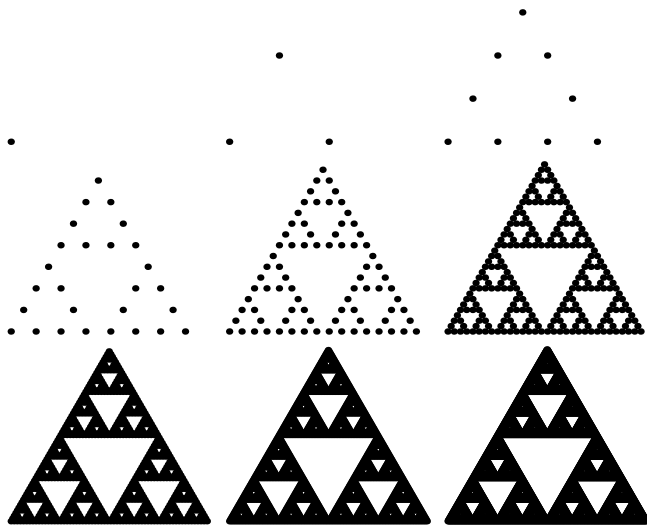


Figure 2.6 Approximating the Sierpinski gasket with finite sets of points.

The point of this demonstration bears repeating: the iterated function system determines the self-similar set and the initial approximation has no effect on the final outcome.

2.5 The ShowIFS command

Like most of the algorithms in this book, the IFS scheme has been implemented in *Mathematica* and is included in one of the `FractalGeometry` packages. Review of the appendix on *Mathematica* might be a good idea prior to starting this section. In particular, we'll make heavy use of Graphics primi-

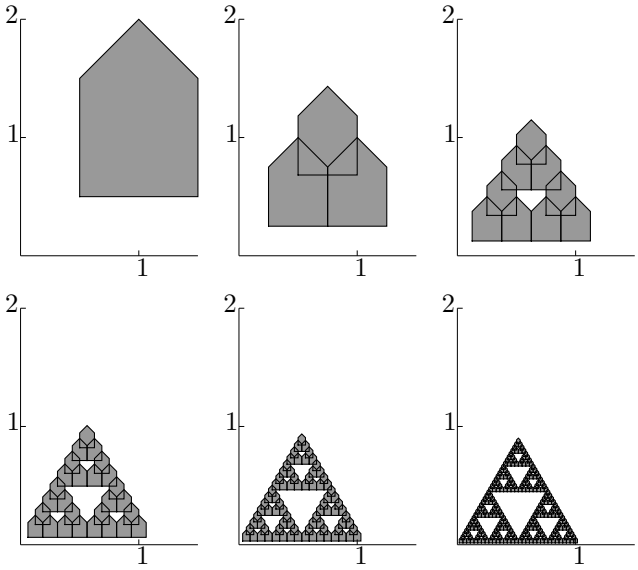


Figure 2.7 Approximating the Sierpinski gasket with an arbitrary set

tives to describe basic shapes. First, let's load the `IteratedFunctionSystems` package.

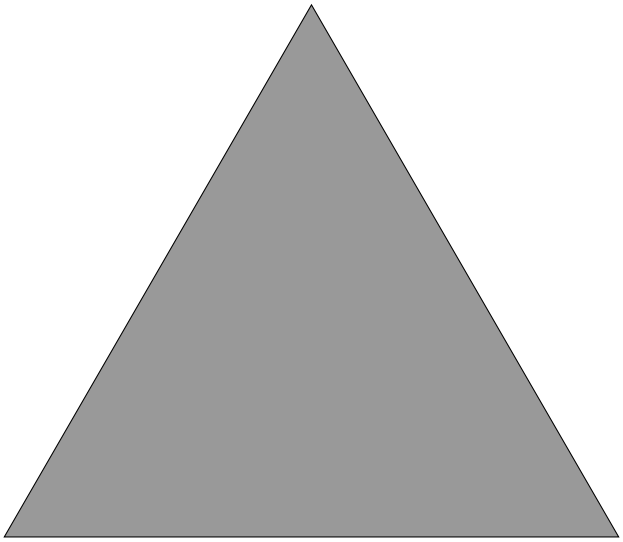
```
Needs["FractalGeometry`IteratedFunctionSystems`"];
```

To use the package, we need a way of representing the IFS. The natural way to do this is as a list of pairs of the form `{M,b}`, where `M` is a matrix and `b` is a shift vector. Thus the IFS for the Sierpinski gasket can be represented as:

```
M = {{1 / 2, 0}, {0, 1 / 2}};
gasketIFS = {
    {M, {0, 0}},
    {M, {1 / 2, 0}},
    {M, {1 / 4, √3 / 4}}
};
```

Next, we need some initial approximation to the invariant set. In principle, this could be any list of `Graphics` primitives, but some initial approximations lead to more natural pictures than others. An equilateral triangle whose base is the unit interval makes a nice initial approximation to the Sierpinski gasket.

```
vertices = {{0, 0}, {1, 0}, {1 / 2, √3 / 2}};
gasketInit = {GrayLevel[.6], EdgeForm[Black], Polygon[vertices]};
Graphics[gasketInit]
```

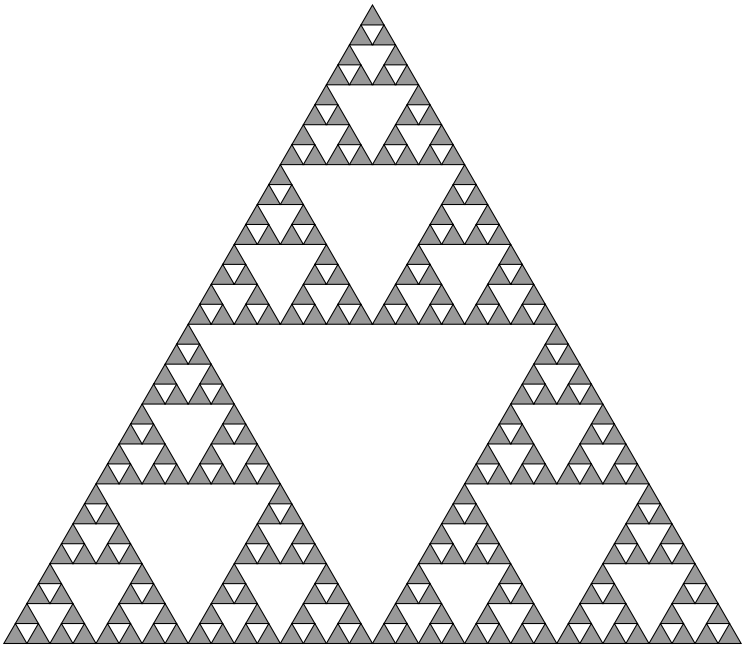


Now we can use the package function `ShowIFS`. The syntax is as follows.

```
ShowIFS[IFS, depth, Initiator → initiator];
```

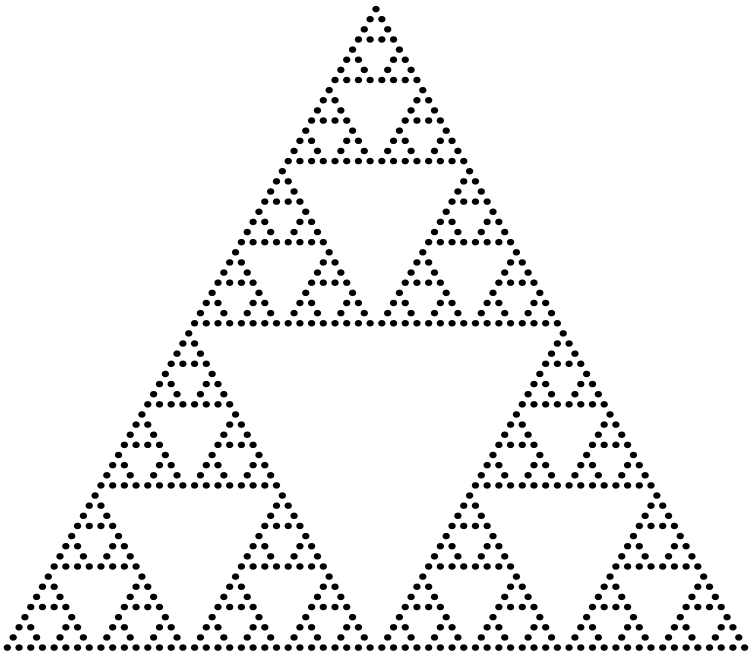
In this command, `IFS` is the IFS defining the set using the representation we have described and `depth` is a non-negative integer representing the number of times the IFS is iterated. The option `Initiator` should be set to a `Graphics` primitive or a list of `Graphics` primitives describing the initial approximation. Thus the following command generates a level 5 approximation to the Sierpinski gasket.

```
ShowIFS[gasketIFS, 5, Initiator → gasketInit]
```



If we don't specify the `Initiator`, then the single point at the origin is used as the default.

```
ShowIFS[gasketIFS, 6]
```



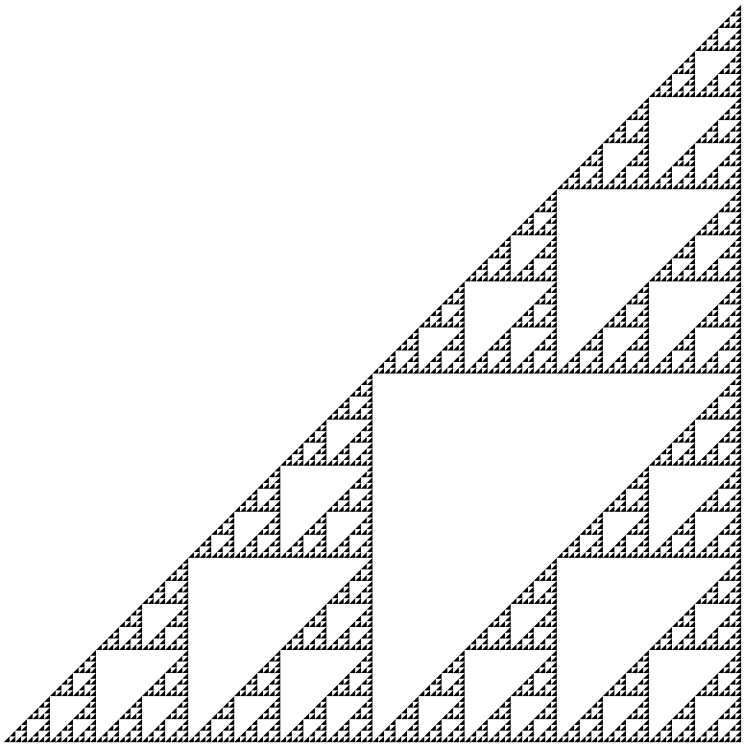
2.6 Examples

The wide variety of beautiful forms which can be generated by fractal algorithms is part of the lure of the subject. We've now developed enough theory and code to look at a few new examples. The first example is a natural generalization of the Sierpinski gasket. That construction can be easily based on any (not necessarily equilateral) triangle. Simply construct the IFS which contracts by the factor $\frac{1}{2}$ about each vertex. Note that if the matrix M defines a contraction about the origin, then the affine function defined by $\{M, x_0 - Mx_0\}$ defines a contraction about the point x_0. Thus here is *Mathematica* code which generates the IFS for the Sierpinski type triangle with vertices at the points A, B, and C.

```
SierpIFS[{A_, B_, C_}] := With[{M = {{1, 0}, {0, 1}} / 2},
   {{M, A - M.A}, {M, B - M.B}, {M, C - M.C}}];
```

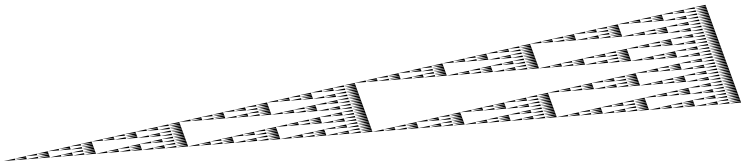
Here's `SierpIFS` in action.

```
vertices = {{0, 0}, {1, 0}, {1, 1}};
skewedGasketIFS = SierpIFS[vertices];
ShowIFS[skewedGasketIFS, 7,
 Initiator → Polygon[vertices]]
```



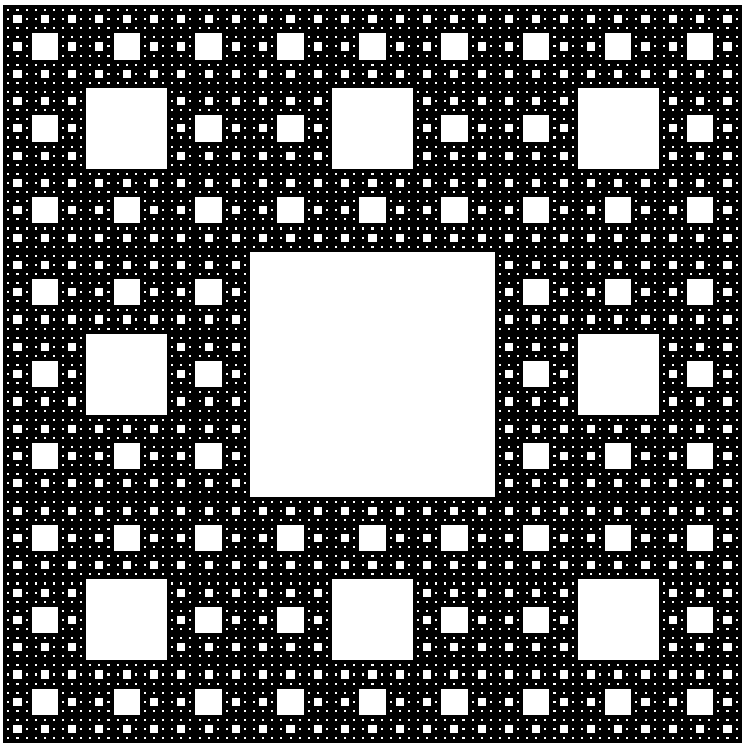
We can do the same thing with randomly chosen vertices.

```
SeedRandom[1];
vertices = RandomReal[{0, 1}, {3, 2}];
randomGasketIFS = SierpIFS[vertices];
ShowIFS[randomGasketIFS, 6,
 Initiator → Polygon[vertices]]
```



Our next example, called the Sierpinski carpet, is similar to the Sierpinski gasket but based on a square rather than a triangle.

```
M = {{1 / 3, 0}, {0, 1 / 3}};
carpetIFS = {
   {M, {0, 0}}, {M, {1 / 3, 0}}, {M, {2 / 3, 0}},
   {M, {0, 1 / 3}}, {M, {2 / 3, 1 / 3}},
   {M, {0, 2 / 3}}, {M, {1 / 3, 2 / 3}}, {M, {2 / 3, 2 / 3}}
  };
unitSquare = Polygon[{{0, 0}, {1, 0}, {1, 1}, {0, 1}}];
ShowIFS[carpetIFS, 5, Initiator → unitSquare]
```



To generate the Koch curve, we need to use rotation and the unit inter-

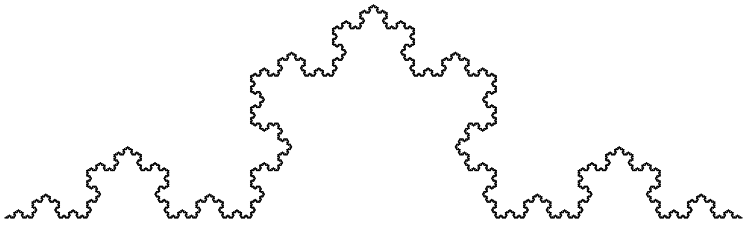
val makes a more natural initiator. Rotation can be added using the rotation matrix defined in equation 2.3. This is implemented in *Mathematica* as `RotationMatrix`.

```
RotationMatrix[θ] // MatrixForm
```

$$\begin{pmatrix} \text{Cos}[\theta] & -\text{Sin}[\theta] \\ \text{Sin}[\theta] & \text{Cos}[\theta] \end{pmatrix}$$

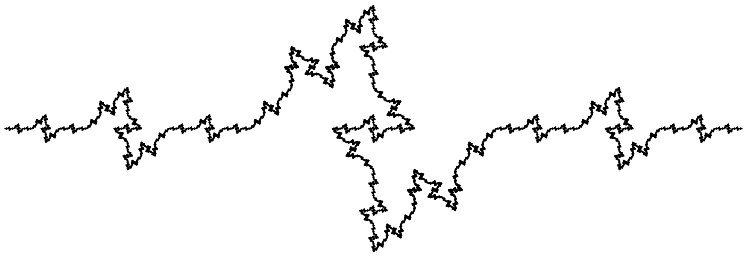
Using this rotation matrix, we can implement the Koch curve IFS from equations 2.4 in *Mathematica* as follows.

```
M = {{1 / 3, 0}, {0, 1 / 3}};
KochCurveIFS = {
   {M, {0, 0}},
   {RotationMatrix[π / 3] / 3, {1 / 3, 0}},
   {RotationMatrix[-π / 3] / 3, {1 / 2, √3 / 6}},
   {M, {2 / 3, 0}}
  };
KochCurveInit = Line[{{0, 0}, {1, 0}}];
ShowIFS[KochCurveIFS, 6, Initiator → KochCurveInit]
```



Here is another IFS where the unit interval is a natural initiator.

```
M = {{1 / 3, 0}, {0, 1 / 3}};
zCurveIFS = {
   {M, {0, 0}},
   {RotationMatrix[π / 4] √2 / 6, {1 / 3, 0}},
   {RotationMatrix[-π / 2] / 3, {1 / 2, 1 / 6}},
   {RotationMatrix[π / 4] √2 / 6, {1 / 2, -1 / 6}},
   {M, {2 / 3, 0}}
  };
zCurveInit = Line[{{0, 0}, {1, 0}}];
ShowIFS[zCurveIFS, 6, Initiator → zCurveInit]
```



There is a major difference between the z-curve in the last example and our other examples. The contraction ratio list for the z-curve is $\left\{\frac{1}{3}, \frac{\sqrt{2}}{6}, \frac{1}{3}, \frac{\sqrt{2}}{6}\frac{1}{3}\right\}$; this is the first example where the contraction ratios are not all the same. This can be a problem for our algorithm. A nice example illustrating this is called the *Sierpinski pedal triangle*.

The Sierpinski pedal triangle is related to a standard construction in classical geometry - the construction of an altitude of a given triangle T. This is a line segment perpendicular to a given side of T and passing through the opposite vertex. Assuming T is acute, the points of intersection of the sides of T with the three altitudes of T determine the vertices of an inscribed triangle called the *pedal triangle* of T. This is not an arbitrary construction; the pedal triangle is the triangle of smallest perimeter which can be inscribed in T.

Figure 2.8 illustrates the pedal construction and the corresponding decomposition of the triangle. In Sierpinski like fashion, we can discard the light pedal triangle in the interior, perform the same procedure on the darker remaining triangles, and iterate. This leads to a fractal figure called the Sierpinski pedal triangle.

It is not hard to show that the darker gray triangles in figure 2.8 are similar to the whole. Segment $\overline{\mathrm{aB}}$, for example, can be considered as a leg of the right triangle ΔAaB. The definition of the cosine function then yields the fact that $\overline{\mathrm{aB}}$ is $\overline{\mathrm{AB}}$ scaled by the factor $\cos(\theta)$, where θ is the measure of $\angle$ABC. A similar argument shows that segment $\overline{\mathrm{Bc}}$ is $\overline{\mathrm{BC}}$ scaled by $\cos(\theta)$. Thus ΔaBc is similar to ΔABC by the side-angle-side criterion and the scaling factor is $\cos(\theta)$. The exact same argument applies to the other dark triangles in figure 2.8.

Since all the non-discarded triangles are similar to the original, the Sierpinski pedal triangle is a self-similar set. We would like to set up the IFS to generate it. Thus we look for three similarities that map the whole triangle onto the darker sub-triangles. Perhaps the easiest way to do this is to assume that each transformation is given in the form $\{M, v\}$, where M is a matrix and v is a shift vector, and write down equations to determine the entries

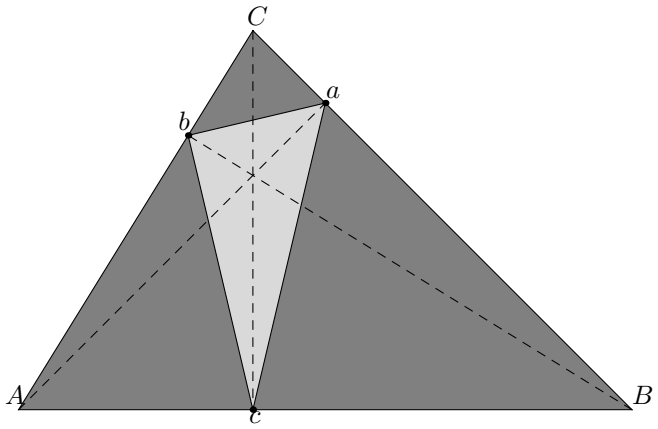


Figure 2.8 The altitudes of a triangle and the associated pedal triangle

of M and v. For example, if we are a trying to map $\triangle ABC$ to $\triangle Abc$ then we must have

$$MA + v = A, MB + v = b, MC + v = c. \tag{2.7}$$

Of course, there are standard projection formulae for b and c in terms of A, B, and C. To obtain b, for example, set $u = B - A$, $v = C - A$ and compute

$$A + \text{proj}_v u = A + \frac{u \cdot v}{v \cdot v} v.$$

Expanding equations 2.7 out yields a linear system of six equations in the unknown entries of M and v. Solving this system yields the similarity. While tedious by hand, *Mathematica* makes this sort of work easy. Here is code which accepts the vertices $A, B, \text{and} C$ of a triangle and returns the IFS defining the Sierpinski pedal triangle determined by A, B, and C. (This function is also encoded as `SierpinskiPedalTriangleIFS` in the `IteratedFunctionSystems` package. Note the difference in case to distinguish between the two functions.)

```
sierpinskiPedalTriangleIFS[{A_, B_, C_}] := Module[
   {a, b, c, d, x, y, v, eqs, M, fA, fB, fC,
    A1 = C + ((B - C).(A - C)) / ((B - C).(B - C)) (B - C),
    B1 = C + ((A - C).(B - C)) / ((A - C).(A - C)) (A - C),
    C1 = A + ((B - A).(C - A)) / ((B - A).(B - A)) (B - A)},

   M = {{a, b}, {c, d}};
   v = {x, y};
   eqs = {M.A + v == A, M.B + v == B1, M.C + v == C1};
   {fA} = {{{a, b}, {c, d}}, {x, y}} /.
     Solve[eqs, {a, b, c, d, x, y}];
   eqs = {M.A + v == A1, M.B + v == B, M.C + v == C1};
   {fB} = {{{a, b}, {c, d}}, {x, y}} /.
     Solve[eqs, {a, b, c, d, x, y}];
   eqs = {M.A + v == A1, M.B + v == B1, M.C + v == C};
   {fC} = {{{a, b}, {c, d}}, {x, y}} /.
     Solve[eqs, {a, b, c, d, x, y}];
   {fA, fB, fC}
  ];
```

For example, the triangle we have been using for illustration has the following vertices.

```
φ = GoldenRatio;
vertices = {{0, 0}, {1, 0}, {1 / φ^2, 1 / φ}};
```

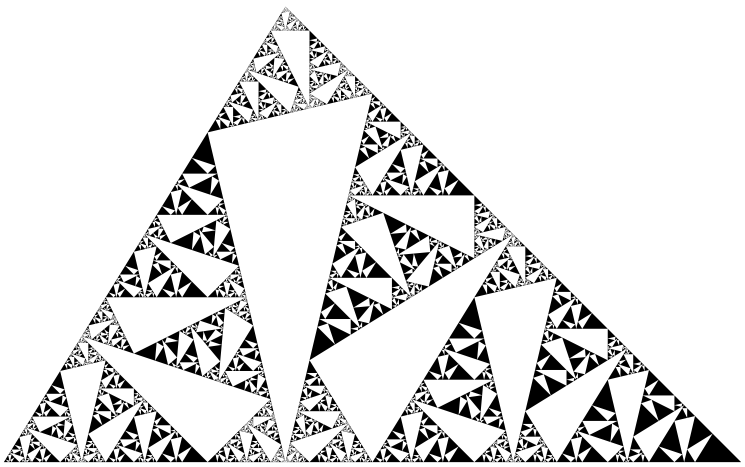
We can generate the IFS for the corresponding Sierpinski pedal triangle as follows.

```
pedalIFS = N[sierpinskiPedalTriangleIFS[vertices]]
```

```
{{{{0.276393, 0.447214}, {0.447214, -0.276393}}, {0., 0.}},
 {{{0.5, -0.5}, {-0.5, -0.5}}, {0.5, 0.5}},
 {{{-0.223607, -0.0527864}, {-0.0527864, 0.223607}}, {0.5, 0.5}}}
```

This can now be passed to the `ShowIFS` function.

```
ShowIFS[pedalIFS, 8, Initiator → Polygon[vertices]]
```



This is an interesting image and it points towards a challenge for our algorithm. Some parts of the fractal are being approximated faster than other parts. This is due to the fact that different functions in the IFS have different contraction ratios. In the next section we look at an algorithm which frequently works better for iterated function systems with varying contraction ratios.

2.7 A modified algorithm

We now describe a simple modification of the basic deterministic algorithm which yields a much more uniform approximation to the invariant set when the IFS has different contraction ratios. As it turns out, these same ideas will help us analyze the dimension of this type of set in the next chapter.

We begin by illustrating the idea behind the modification for the particular case of the Sierpinski pedal triangle. The input is a positive parameter r and a level zero approximation consisting of a single acute triangle. The algorithm will yield a finite sequence of improving approximations to the attractor; the n^{th} level approximation consists of a collection of triangles and we consider each of these separately to get to level $n+1$. Let T be a triangle in the n^{th} level approximation. T will be subdivided according to the pedal scheme only if the length of its longest side exceeds r. Otherwise, T appears unchanged at level $n+1$. This recursive process stops when the maximum side length of all triangles in the approximation is less than r.

This process is illustrated in figure 2.9 using the parameter $r = 0.4$. The longest side of each dark triangle is labeled by its length. The process terminates after three steps since all labels are less than r. Note that once the top triangle (labeled 0.23) appears, it never subdivides further since $0.23 < 0.4$. The triangle on the bottom right (labeled 0.707) subdivides once, and one of its sub-triangles subdivides further. As a result, the ratio of the largest triangle to the smallest is not too large and the approximation is relatively uniform.

In order to generalize this scheme to an arbitrary IFS $\{f_i\}_{i=1}^m$, we need to develop a bit of notation. A string with symbols chosen from $\{1,\ldots,m\}$ is simply a finite sequence with values in $\{1,\ldots,m\}$. A string α of length k will be denoted $\alpha = i_1\cdots i_k$ and J_k will denote the set of all strings of length k. Strings can be concatenated to form longer strings. Thus if $\alpha = i_1\cdots i_k$ and $\beta = j_1\cdots j_l$, then $\alpha\beta = i_1\cdots i_k j_1\cdots j_l$. Given a string $\alpha = i_1\cdots i_k \in J_k$, let $f_\alpha = f_{i_1}\circ\cdots\circ f_{i_k}$ and $r_\alpha = r_{i_1}\cdots \mathrm{r}_{i_k}$. Note that the invariant set E of the IFS satisfies

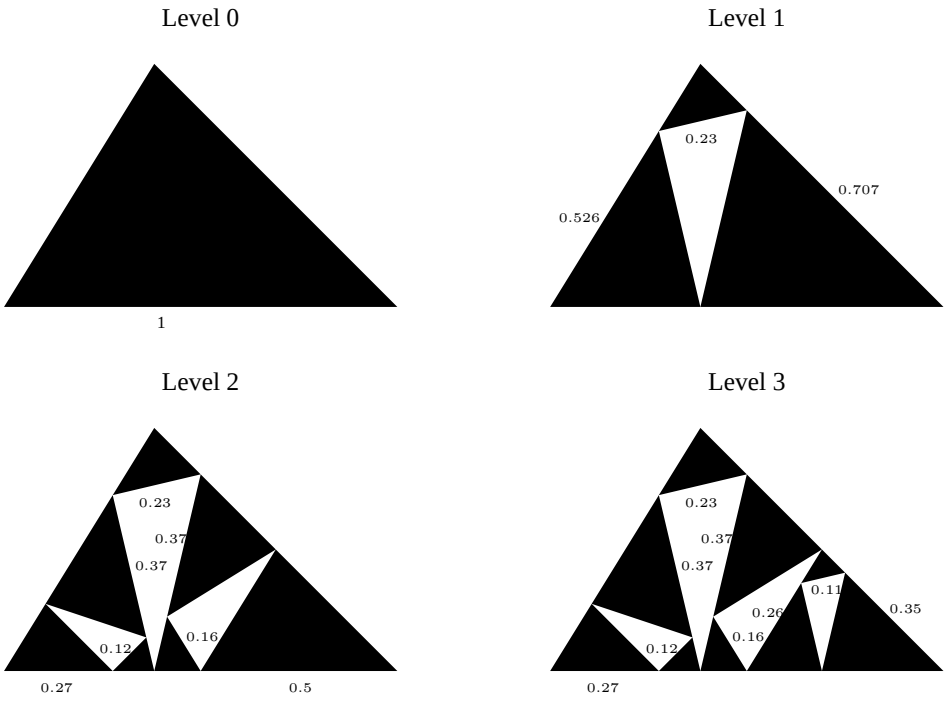


Figure 2.9 The modified algorithm for the Sierpinski pedal triangle

$$E = \bigcup_{\alpha \in J_k} f_\alpha(E).$$

Thus, J_k induces a k^{th} decomposition of E. This decomposition is not as useful as it could be since the sizes of the sets $f_\alpha(E)$ can vary greatly. Our objective is to use this notation to develop an alternative decomposition.

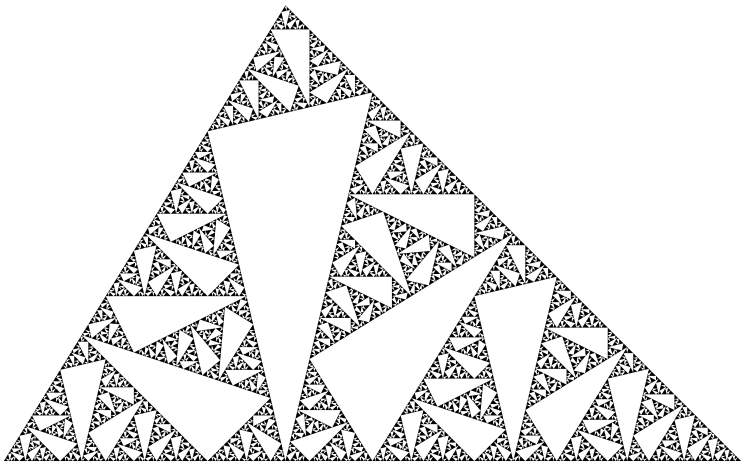
We start now with a positive parameter r and the set $J[0]$ containing just the empty string e. The function f_e is, by convention, the identity function and the corresponding ratio r_e is 1. We get from $J[n]$ to $J[n+1]$ as follows by considering each $\alpha \in J[n]$ in turn. The string α will be replaced with the m strings $\alpha 1$, $\alpha 2$, ..., αm formed by concatenation, only if $r_\alpha \geq r$. Otherwise, α is left unchanged at level $n+1$. This recursive process stops with the set of strings J' when $r_\alpha < r$ for all $\alpha \in J'$.

The set J' can now be used to generate a fractal picture. Given an initial approximation E_0 to the invariant set E, an approximation to E can be generated by applying all function from J' to E_0. In symbols,

$$E \approx \bigcup_{\alpha \in J'} f_\alpha (E_0) .$$

This process is implemented by the `ShowIFS` function. Simply call the function using a real number as the second parameter. Of course, the parameter should be less than 1 and smaller values lead to better approximations. Here is an approximation to our Sierpinski pedal triangle using this algorithm.

```
ShowIFS[pedalIFS, 0.01,
 Initiator → Polygon[vertices]]
```



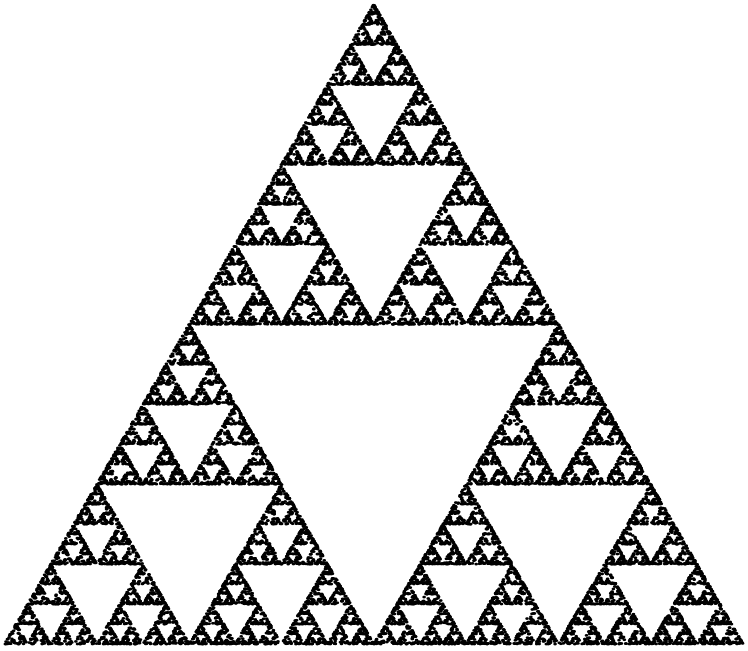
2.8 A stochastic algorithm

There is a stochastic algorithm for generating invariant sets of iterated function systems. Rather than applying every function in the IFS to a given initiator and iterating, we randomly apply the functions one at a time and iterate. With each application of a function from the IFS, the generated point should get closer to the attractor and the randomness of the algorithm will (hopefully) generate a somewhat uniform distribution of points over the attractor. This algorithm is encapsulated in the `ShowIFSStochastic` command. The general syntax is as follows:

```
ShowIFSStochastic[IFS, numPoints];
```

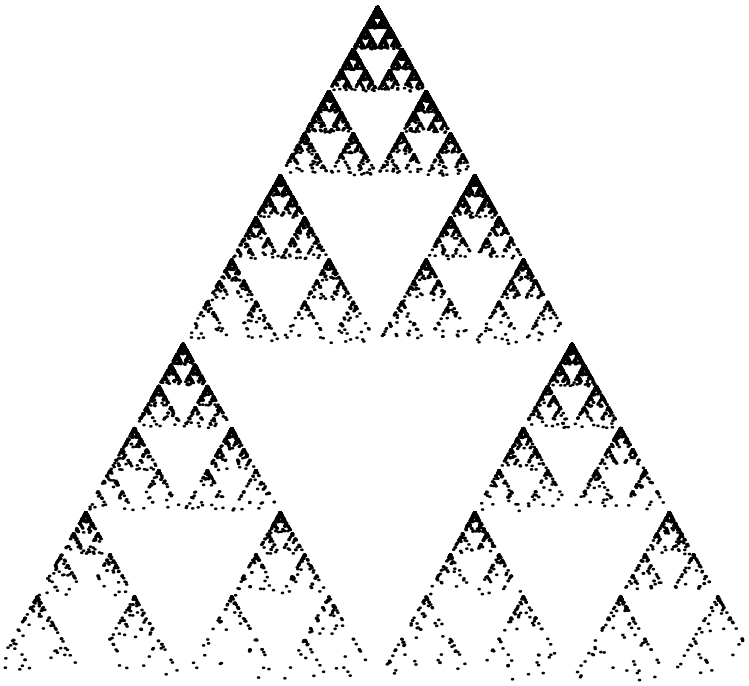
Thus, assuming that `gasketIFS` is defined as before and the `IteratedFunctionSystems` package has been loaded, we can generate a 20000 point approximation to the Sierpinski gasket as follows.

```
ShowIFSStochastic[gasketIFS, 20 000]
```



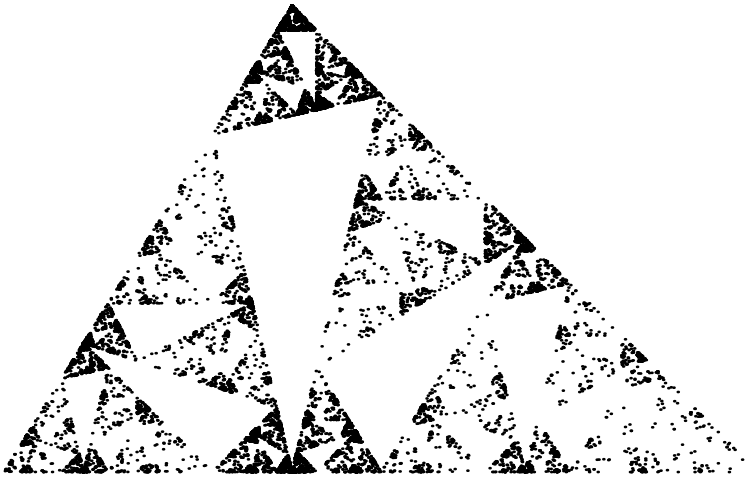
The functions of the IFS need not be chosen with equal probabilities. The `Probabilities` option can be used to skew the choices. Here is the effect of choosing the contraction about the top vertex with greater probability than the other two.

```
ShowIFSStochastic[gasketIFS, 20 000,
 Probabilities → {1 / 6, 1 / 6, 2 / 3}]
```



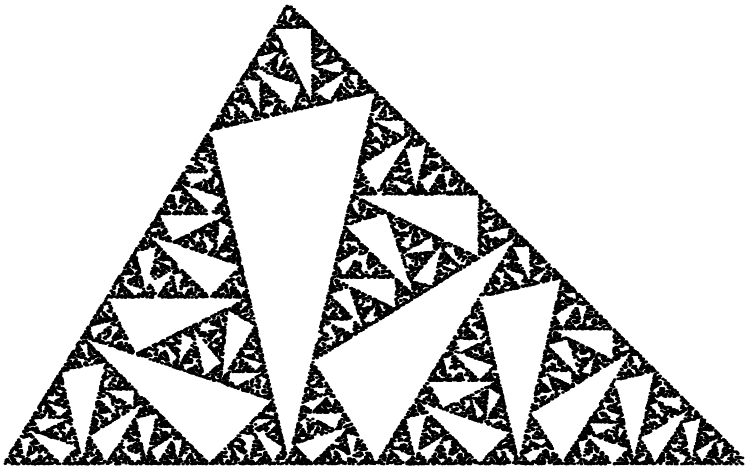
In some situations, the probabilities for choosing the similarities from an IFS should be skewed in order to get a uniform distribution of points. For example, consider the following image of a Sierpinski pedal triangle.

```
φ = GoldenRatio;
vertices = {{0, 0}, {1, 0}, {1 / φ^2, 1 / φ}};
pedalIFS = SierpinskiPedalTriangleIFS[vertices];
ShowIFSStochastic[pedalIFS, 20 000,
 Probabilities → {1 / 3, 1 / 3, 1 / 3}]
```



The similarities were all chosen with equal probability 1/3, yet the distribution of points is non-uniform. The basic problem is that the variation in the ratio list leads to a non-uniformity in the points approximating the image. However, the stochastic algorithm allows for a way to compensate for this by varying the probabilities by which we choose the functions. In general, the probability of choosing a similarity from an IFS should depend upon the contraction ratio of the similarity; larger contraction ratios should correspond to larger probabilities. A precise statement of this idea will have to wait until we discuss dimension, but we can experiment with the function in the `IFS` package which implement the probability adjustment. For example, the default choice of probabilities made by `ShowIFSSTochastic` gives a more uniform distribution of points.

```
ShowIFSStochastic[pedalIFS, 20000]
```



The command `FindProbabilities` returns the probabilities used by `ShowIFSStochastic`.

```
FindProbabilities[pedalIFS]
```

```
{0.346572, 0.564856, 0.0885727}
```

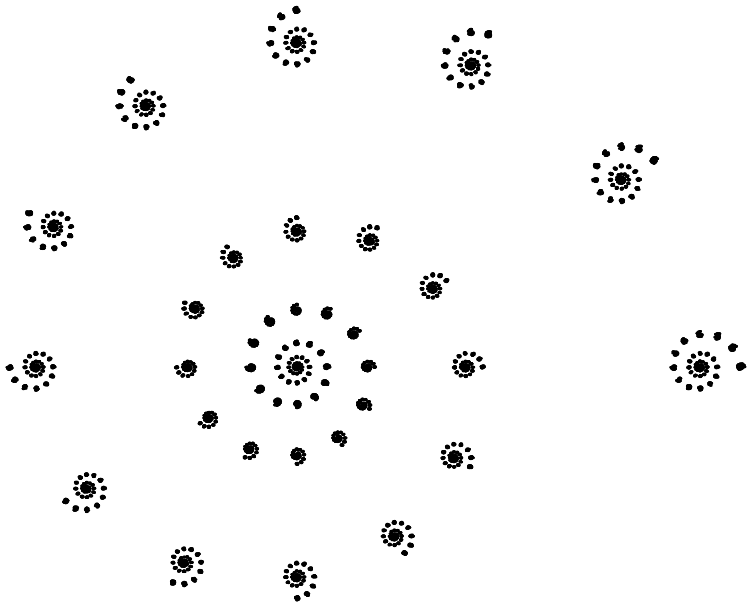
```
Total[%]
```

```
1.
```

2.9 Fractal Spirals

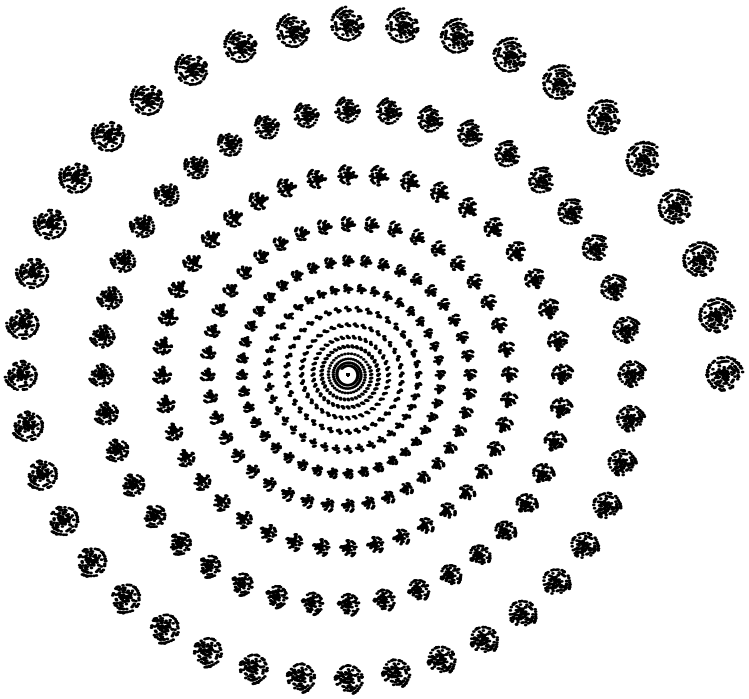
Beautiful spiral images can be generated using iterated function systems with only two transformations. The basic idea is to combine a rotation and mild contraction at the origin with a shift and stronger contraction. The rotation induces the spiral effect. Here is a simple example of this idea.

```
simpleSpiralIFS = {
   {.93 RotationMatrix[Pi / 6], {0, 0}},
   {.1 IdentityMatrix[2], {1, 0}}
  };
ShowIFSStochastic[simpleSpiralIFS, 10 000]
```



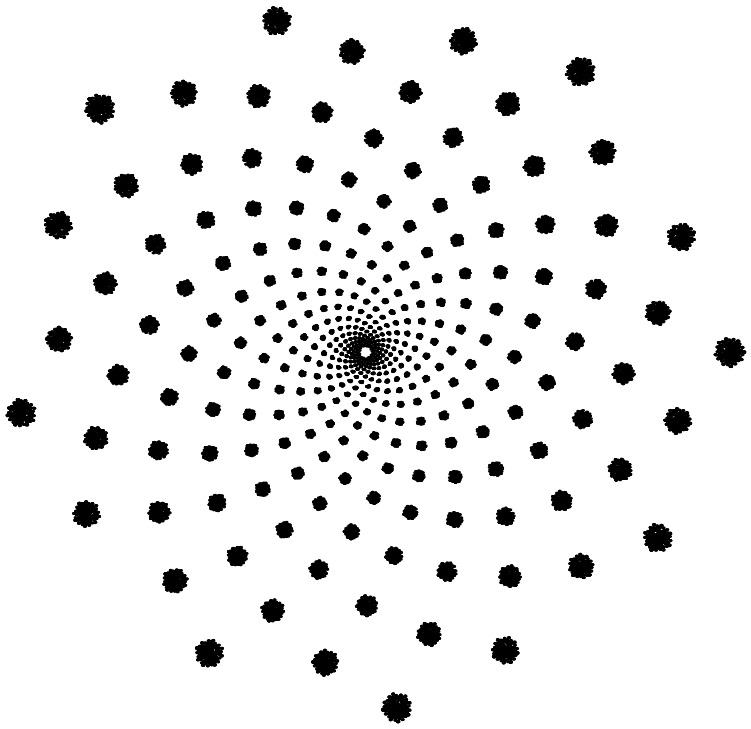
Experimentation with the contraction and rotation values can lead to dramatic effects.

```
anotherSpiralIFS = {
   {.993 RotationMatrix[Pi / 20], {0, 0}},
   {.05 IdentityMatrix[2], {1, 0}}
  };
ShowIFSStochastic[anotherSpiralIFS, 15 000]
```



A particularly beautiful class of fractal spirals is motivated by phyllotaxis - the study of circular arrangements of lateral components in plants. For example, the seeds in a sunflower or the scales on a pine cone tend to arrange themselves in intriguing spiral patterns involving Fibonacci numbers and the golden ratio. A careful analysis of this fact suggests that these patterns allow the components to be packed together as tightly as possible, a prime example of economy in nature. If we look closely at the seeds in a sunflower first at the seed farthest away from the center, then at the seed next farthest away, and so on, we notice that successive seeds are separated by an angle of approximately 137.5°. Further analysis suggests that the exact value of the angle should be $2\pi/\varphi^2$, where $\varphi = \left(1+\sqrt{5}\right)/2$ is the golden ratio. This discussion motivates the following IFS.

```
phyllotacticSpiralIFS = {
   {.99 RotationMatrix[2 Pi / GoldenRatio^2], {0, 0}},
   {.04 IdentityMatrix[2], {1, 0}}
  };
ShowIFSStochastic[phyllotacticSpiralIFS, 30 000]
```



In this figure, we see two predominant families of intertwined spirals; one family spirals clockwise while the other counter-clockwise. Furthermore, if we count the clockwise spirals we find 13 and if we count the counter-clockwise spirals we find 21. The numbers 13 and 21 are successive Fibonacci numbers! Experimentation with the contraction ratios can lead to varying numbers of clockwise and counter-clockwise spirals, but those numbers will always be successive Fibonacci numbers.

2.10 *Mathematica* implementation

One goal of this book is to demonstrate how *Mathematica* can be used to generate fractal objects. In this section, we describe some of the details of the `ShowIFS` and `ShowIFSStochastic` commands.

2.10.1 Details for the ShowIFS command

The first step for the deterministic version is to transform the given IFS into a list of pure functions which can act on an object consisting of `Graphics` primitives. Here is a function which accepts our representation of an affine

function and returns the corresponding pure function which acts on graphics primitives.

```
toFunc[{M_, b_}] := GeometricTransformation[#,
    AffineTransform[N[{M, b}]]] &;
```

You can think of `GeometricTransformation` as a high level graphics primitive. The corresponding function is automatically applied to the graphic, when this construct appears inside `Graphics`. We can now map `toFunc` onto the IFS.

```
pureFuncs = toFunc /@ gasketIFS;
```

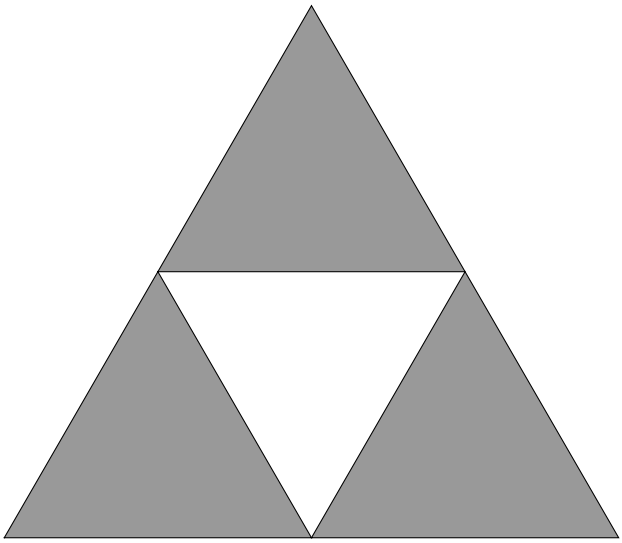
Each one of these can now act on our initiator. For example:

```
pureFuncs[[3]][gasketInit]
```

```
GeometricTransformation[
 {GrayLevel[0.6], EdgeForm[GrayLevel[0]], Polygon[{{0, 0}, {1, 0}, {1/2, Sqrt[3]/2}}]},
 {{{0.5, 0.}, {0., 0.5}}, {0.25, 0.433013}}]
```

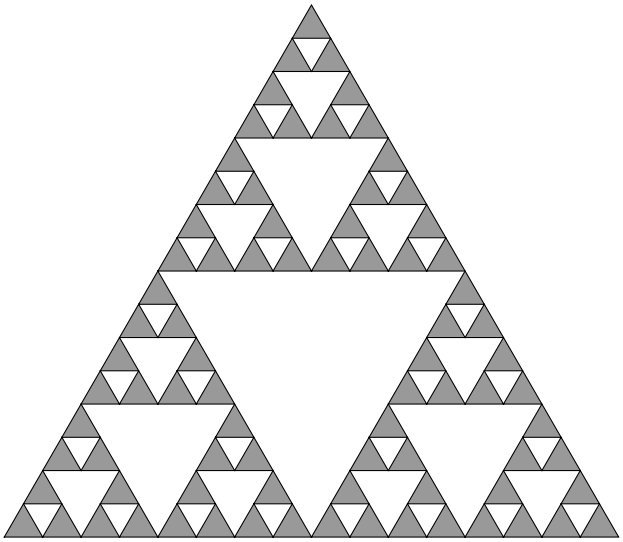
Let's see what it looks like if we apply each function in `pureFuncs` to the initiator (which can be accomplished using *Mathematica*'s `Through` function) and pass the result to graphics.

```
Graphics[Through[pureFuncs[gasketInit]]]
```



Not bad. Now we need to iterate the procedure. For this purpose, we use *Mathematica*'s `Nest` function.

```
graphic = Nest[Through[pureFuncs[#]] &, gasketInit, 4];
Graphics[graphic]
```



That is really the essentials. Of course, there is a fair amount of code that deals with options and encapsulates the command into a package. The code for the modified algorithm is a bit trickier as well, but the primary mathematical portion of the algorithm is contained in the few lines of code above.

2.10.2 Details for the ShowIFSStochastic command

We again transform the given IFS into a list of pure functions, but this time they only need act on points.

```
toFunc[{M_, b_}] := N[M.# + b] &;
funcs = toFunc /@ gasketIFS;
```

We next generate a long list of these functions chosen randomly.

```
numPoints = 10;
functionList = RandomChoice[funcs, numPoints];
```

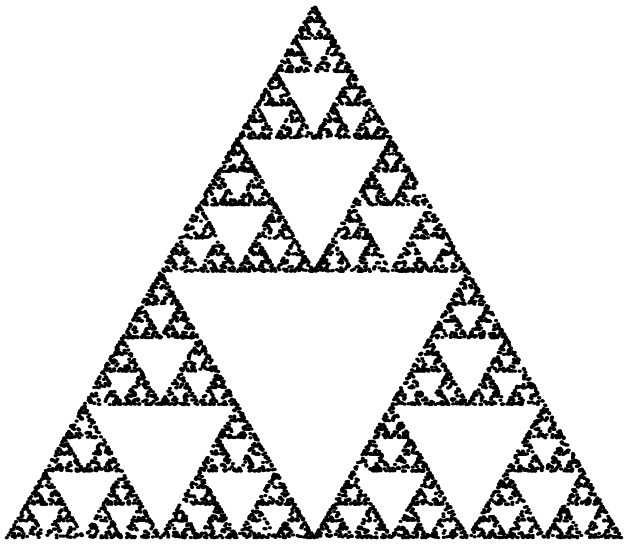
Finally, we generate a list of points applying the first function in the `functionList` to the origin, applying the second function in the `functionList` to the first result, and iterating. This is easily accomplished using *Mathematica*'s `ComposeList` function.

```
ComposeList[functionList, {0, 0}]
```

```
{{0, 0}, {0.5, 0.}, {0.5, 0.433013}, {0.5, 0.649519}, {0.75, 0.32476},
 {0.875, 0.16238}, {0.9375, 0.0811899}, {0.46875, 0.0405949},
 {0.484375, 0.45331}, {0.742188, 0.226655}, {0.621094, 0.54634}}
```

These points can be passed to the graphics routines, but we should generate more points to get a good image.

```
numPoints = 8000;
points = ComposeList[RandomChoice[funcs, numPoints], {0, 0}];
Graphics[{PointSize[0.005], Point[points]}]
```



Again, there are a few other issues that the package must deal with - particularly, the computation of the scaling ratios and determination of the correct probabilities to yield a uniform distribution in terms of those ratios.

2.11 Notes

The definitive paper on self-similarity is Hutchinson (1981), although similar ideas certainly appeared much earlier. Barnsley (1993) is generally credited for popularizing the technique. Many of the examples in this chapter are well known. The Sierpinski pedal triangle construction is relatively new and first appears in Zhang et al. (2008). While there are a number of freely available *Mathematica* implementations of the IFS technique, the treatment here owes a bit to Gutierrez et al. (1997)

Exercises

2.1 Show that a number $x \in [0, 1]$ is in the Cantor set if and only if x can be written in base 3 using only 0s and 2s. Why does this imply that the Cantor set is uncountable?

2.2 Use the previous exercise and a geometric series to show that $\frac{1}{4}$ is in the Cantor set.

2.3 Use a geometric series to compute the total length of the intervals removed from the unit interval during the construction of the Cantor set.

2.4 Write three different iterated function systems to describe the Cantor set - one using two similarties, one using three similarities, and one using four similarities.

2.5 Let D be the set of all numbers in the unit interval which can be written in base 5 using only 0s, 2s, and 4s. Write down an iterated function system to describe D.

2.6 Describe a natural addressing scheme for the Sierpinski gasket using three symbols.

2.7 Show the area of the Sierpinski gasket is zero.

2.8 Find an IFS for the Koch curve consisting of only two similarities. Use the `ShowIFS` command to generate the Koch curve using your IFS.

2.9 Find an IFS to render each of the images in figure 2.10.

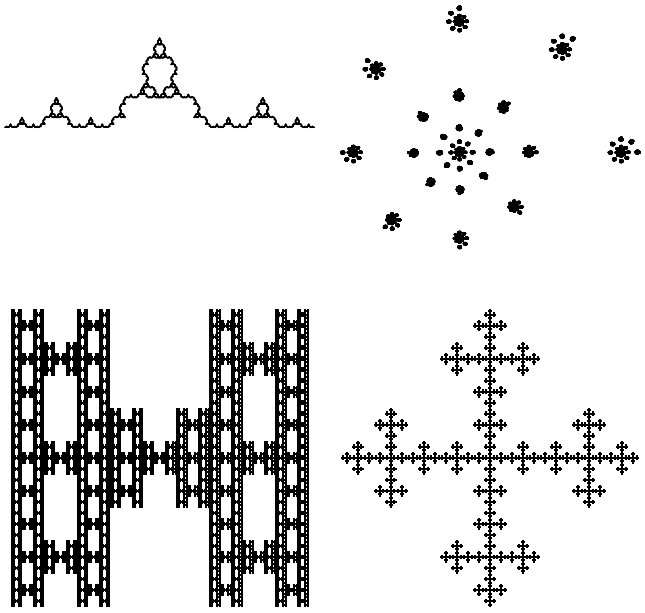


Figure 2.10 Several self-similar sets

2.10 Find an IFS that might generate the sequence of images shown in figure

2.11. Use Graphics primitives together with the `ShowIFS` command to generate the sequence of images.

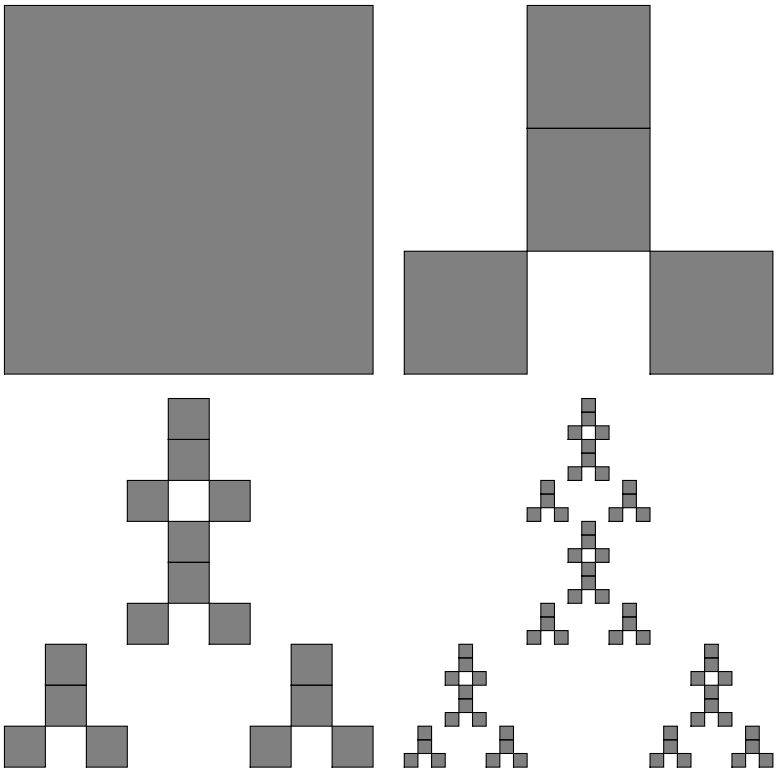


Figure 2.11 Approximations to a self-similar set

2.11 The images in figure 2.12 depict the action of two iterated function systems on a pentagon; the original pentagon is the bold boundary. For each image, determine the iterated function system which generated it and use it to generate a better approximation to the corresponding fractal.

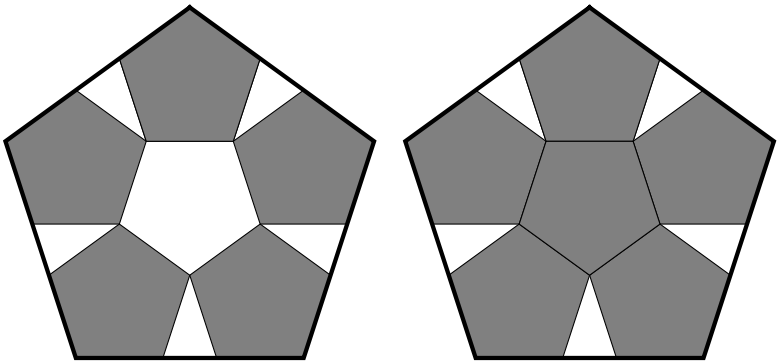


Figure 2.12 Skeletons of pentagonal self-similar sets

3

Some mathematical details

In this chapter, we will be address questions of a purely theoretical nature. How do we know that the IFS scheme always yields a unique self-similar set? What is the fractal dimension of a set and what does it tell us? It is not uncommon for these kinds of questions lead back to applications. For example, the fractal dimension of a set turns out to be the proper tool to help us determine the proper probabilities when implementing the stochastic algorithm for generating self-similar sets.

This is the most theoretical chapter of the book. Most results will be proven carefully using techniques of real analysis which are briefly outlined in the Appendix A.3.

3.1 Invariant sets

The existence and uniqueness of invariant sets of iterated function systems can be established by an elegant application of the contraction mapping theorem from real analysis. The contraction mapping theorem states that any contractive function mapping $\mathbb{R}^n$ to $\mathbb{R}^n$ has a unique fixed point. But as we will see, the statement is true in a more general setting. We will be able define a notion of distance between two subsets of $\mathbb{R}^n$, consider our space to be the set of all appropriately chosen subsets, and use the IFS scheme to define a contraction on this space. The contraction mapping theorem then yields a unique invariant set for the IFS.

Theorem 3.1 *If $f : \mathbb{R}^n \to \mathbb{R}^n$ is a contraction, then there is a unique x' in $\mathbb{R}^n$ such that $f(x') = x'$.*

Proof Suppose that f is a contraction of $\mathbb{R}^n$ with contractivity factor r and let x_0 be an arbitrary element of $\mathbb{R}^n$. We will define a sequence starting at x_0 by iterating the function f. That is, define $x_1 = f(x_0), x_2 = f(x_1)$, and

for larger integers n, $x_n = f(x_{n-1}) = f^n(x_0)$. We'll show that the sequence (x_n) thus defined is Cauchy and therefore convergent. It turns out that the limit of this sequence is the unique fixed point of the function.

To show the sequence is Cauchy, let $\varepsilon > 0$and choose $N \in \mathbb{N}$ such that

$$\sum_{n=N}^{\infty} r^n |f(x_0) - x_0| < \varepsilon.$$

This is possible since the series is the tail of a convergent geometric series. Note that for any $n \geq N$,

$$|x_{n+1} - x_n| = \left|f^{n+1}(x_0) - f^n(x_0)\right| \leq r^n |f(x_0) - x_0|.$$

Thus, by the triangle inequality, for $n > m \geq N$,

$$\begin{aligned} |x_n - x_m| &= |x_n - x_{n-1} + x_{n-1} - x_{n-2} + x_{n-2} + ... + x_{m+1} - x_{m+1} - x_m| \\ &\leq |x_n - x_{n-1}| + |x_{n-1} - x_{n-2}| + ... + |x_{m+1} - x_m| \\ &\leq \left(r^{n-1} + r^{n-2} + ...r^m\right) |f(x_0) - x_0| < \varepsilon. \end{aligned}$$

This shows that the recursively defined sequence (x_n) is Cauchy. Therefore, the sequence converges to some value x'.

We now show that $f(x') = x'$. Let $\varepsilon > 0$. Since $x_n \to x'$, we can choose some $N \in \mathbb{N}$ such that $|x_n - x'| < \varepsilon/2$ whenever $n \geq N$. Then

$$\begin{aligned} |f(x') - x'| &= \left|f(x') - x_{N+1} + x_{N+1} - x'\right| \\ &\leq \left|f(x') - x_{N+1}\right| + \left|x_{N+1} - x'\right| \\ &= \left|f(x') - f(x_N)\right| + \left|x_{N+1} - x'\right| \\ &\leq r\left|x' - x_N\right| + \left|x_{N+1} - x'\right| \\ &< r\frac{\varepsilon}{2} + \frac{\varepsilon}{2} < \varepsilon. \end{aligned}$$

Thus $f(x') = x'$ as $\varepsilon > 0$ is arbitrary.

Finally, we show that the fixed point x' is unique. Consider two distinct points, say x_1 and x_2, Then

$$|f(x_1) - f(x_2)| \leq r|x_1 - x_2| < |x_1 - x_2|.$$

Thus not both can be fixed.

□

The proof of the contraction mapping theorem is constructive, i.e. the proof outlines a technique for finding the fixed point. Furthermore, the basic technique of iteration plays an important role in this text. As an example, consider the function $f(x) = \cos(x)$ which maps the unit interval $I = [0, 1]$ to itself. We can show that f is a contraction on I. In fact, $|f(x) - f(y)| \leq r|x-y|$, where $r = \sin(1)$. To see this we apply basic trigonometric identities to simplify $\cos(x) - \cos(y)$. *Mathematica* will do this for us.

```
TrigFactor[Cos[x] - Cos[y]]
```

$$-2\,\mathtt{Sin}\left[\frac{x}{2} - \frac{y}{2}\right]\,\mathtt{Sin}\left[\frac{x}{2} + \frac{y}{2}\right]$$

Recall also that $\sin(\theta) \leq \theta$ for all $\theta \in \mathbb{R}$ and $\frac{x+y}{2} \leq 1$, since x and y are both in I. Thus

$$|\cos(x)-\cos(y)| \leq 2|\sin\left(\frac{x-y}{2}\right)\sin\left(\frac{x+y}{2}\right)| \leq 2\frac{|x-y|}{2}\sin(1) = r|x-y|.$$

Now, $\cos(x)$ clearly has precisely one fixed point in the unit interval as a simple graph shows. According to the proof of the contraction mapping theorem, we should be able to find this fixed point by iterating the function from an arbitrary starting value. We can accomplish this via *Mathematica*'s `NestList` command.

```
NestList[Cos, .5, 20]
```

```
{0.5, 0.877583, 0.639012, 0.802685, 0.694778, 0.768196, 0.719165,
 0.752356, 0.730081, 0.74512, 0.735006, 0.741827, 0.737236, 0.74033,
 0.738246, 0.73965, 0.738705, 0.739341, 0.738912, 0.739201, 0.739007}
```

It appears that to three significant digits, the fixed point is 0.739. Note that this iterative procedure is exactly how *Mathematica*'s `FixedPoint` command works.

```
FixedPoint[Cos, .5]
```

```
0.739085
```

3.1.1 The Hausdorff metric

Although we proved the contraction mapping theorem for $\mathbb{R}^n$, it is valid in the more general setting of complete metric spaces described in the appendix on real analysis. In particular, it applies to the Hausdorff metric space, which defines a notion of distance between compact sets of $\mathbb{R}^n$. Two sets, A and B, will be considered close to one another if every point in A is close to some point of B and vice-versa. To make this precise, we make the following

definitions. First define $\mathcal{H}$ to the collection of all non-empty, closed, bounded subsets of $\mathbb{R}^n$. For a point x in $\mathbb{R}^n$ and a set $B \in \mathcal{H}$, define the distance from x to B by

$$\text{dist}(x, B) = \min\{|x - y| : y \in B\}.$$

Thus $\text{dist}(x, B)$ is simply the Euclidean distance from x to the closest point of B. For two compact sets A and B contained in $\mathbb{R}^n$, define

$$\text{dist}(A, B) = \max\{\text{dist}(x, B) : x \in A\}.$$

Thus $\text{dist}(A, B)$ represents the largest possible distance of a point chosen from A to the set B. Note that dist does not form a metric on $\mathcal{H}$, since it is not necessarily symmetric. The Hausdorff metric d is defined to be a symmetric version of dist by

$$d(A, B) = \max\{\text{dist}(A, B), \text{dist}(B, A)\}.$$

For example, if A and B are the two overlapping closed disks shown in figure 3.1, then $d(A, B)$ is the maximum of the two labeled distances.

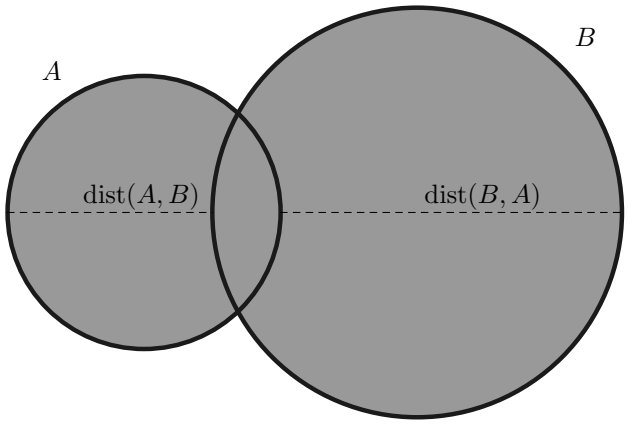


Figure 3.1 Definition of the Hausdorff distance

It can be shown that d is a complete metric on $\mathcal{H}$; see for example Edgar (2009). We can use this fact in conjunction with the contraction mapping theorem to establish the existence and uniqueness of invariant sets for iterated function systems. Given an IFS $\{f_i\}_{i=1}^m$ and a set $A \in \mathcal{H}$, define $T(A)$ by

$$T(A) = \bigcup_{i=1}^{m} f_i(A).$$

Since the continuous image of a compact set is compact and the finite union of compact sets is compact, $T(A)$ will also be in $\mathcal{H}$; i.e. $T : \mathcal{H} \to \mathcal{H}$. Furthermore, if r satisfies $|f_i(x) - f_i(y)| \leq r|x - y|$, for all $x, y \in \mathbb{R}^n$ and for all $i = 1, \ldots, m$, then

$$d(T(A), T(B)) \leq r d(A, B).$$

(The details of this will be left to the exercises.) Thus T is a contraction on $\mathcal{H}$ and, therefore, has a unique fixed point E. By the definition of T, the set E is the invariant set of the IFS. This result is worth summarizing as a theorem.

Theorem 3.2 *Let $\{f_i\}_{i=1}^m$ be an IFS of contractions defined on $\mathbb{R}^n$. Then there is a unique non-empty, compact set E in $\mathbb{R}^n$ such that*

$$E = \bigcup_{i=1}^{m} f_i(E).$$

By the proof of the contraction mapping theorem, we have a technique constructing approximations to invariant sets. Start with an initial approximation A_0 and construct a sequence recursively by $A_{n+1} = T(A_n)$. This is exactly the technique implemented by the code described in chapter 2.

3.2 Fractal Dimension

Fractal geometry is concerned with the study of geometrically complicated objects. The concept of "fractal dimension"' is a quantitative measure of this complexity. The Cantor set, for example, is a set between dimensions. On one hand it seems small enough to be of dimension zero, but on the other hand it is a much richer set than what one might think of as a zero dimensional set. Fractal dimension quantifies its place in this spectrum.

There are many notions of fractal dimension - Hausdorff, similarity, box-counting, and packing dimensions are just a few. In the serious study of fractal geometry it is important to understand the relationships between these ideas. In particular, we would like to know conditions guaranteeing equality of two or more definitions. Perhaps one definition is of theoretical importance, while the other is easier to calculate.

We will focus on the similarity dimension and the box-counting dimension. The box-counting dimension is broadly applicable and widely used, but can be difficult to calculate. The similarity dimension is of much more restricted applicability, but easy to calculate when appropriate. Fortunately, it turns out that these concepts are equivalent on suitably chosen sets.

3.2.1 Quantifying dimension

Both definitions of dimension under consideration are generalizations of a very simple idea. We attempt to quantify the dimension of some very simple sets in a way that generalizes to more complicated sets. The simple sets we consider are the unit interval $[0,1]$, the unit square $[0,1]^2$, and the unit cube $[0,1]^3$, which should clearly have dimensions 1, 2, and 3 respectively. Each of these can be decomposed into some number N_r of copies of itself when scaled by certain factors r. The following table shows the values of N_r for various choices of r and for each of our simple objects.

	$[0,1]$	$[0,1]^2$	$[0,1]^3$
$r = 1/2$	$N_r = 2$	$4 = 2^2$	$8 = 2^3$
$r = 1/3$	$N_r = 3$	$9 = 3^2$	$27 = 3^3$
$r = 1/5$	$N_r = 5$	$25 = 5^2$	$125 = 5^3$

Note that no matter the scaling factor or set, the number of pieces N_r, the scaling factor r, and the dimension d are related by $N_r = (1/r)^d$. This motivates the following definition: If $E \subset \mathbb{R}^n$ can be decomposed into N_r copies of itself scaled by the factor r, then

$$\dim(E) = \frac{\log N_r}{\log 1/r} \tag{3.1}$$

Many sets constructed via iterated function systems can be analyzed via equation 3.1. For example, the Cantor set is composed of two copies of itself scaled by the factor $1/3$. Thus it's fractal dimension is $\frac{\log 2}{\log 3}$. The fractal dimension of the Sierpinski gasket is $\frac{\log 3}{\log 2}$ and the fractal dimension of the Koch curve is $\frac{\log 4}{\log 3}$.

3.2.2 Similarity dimension of an IFS

Equation 3.1 is not quite general enough to compute the fractal dimension of all self-similar sets, since iterated function systems need not have all contraction ratios equal to one another. For example, equation 3.1 cannot

compute the dimension of the z-curve. There is an important generalization of equation 3.1 which defines the dimension associated with any IFS. We will assume that all iterated function systems consist of pure similarities for the remainder of this chapter.

Definition 3.3 Let $\{f_i\}_{i=1}^m$ be a fixed IFS of similarities and let $\{r_i\}_{i=1}^m$ be the list of associated similarity ratios. Define a function $\Phi : [0, \infty) \to \mathbb{R}$ by

$$\Phi(s) = r_1^s + \cdots + r_n^s.$$

Note that Φ is continuous, strictly decreasing, $\Phi(0) = m$, and $\lim_{s\to\infty} \Phi(s) = 0$. Thus there is a unique positive number s such that $\Phi(s) = 1$. This unique value of s is defined to be the similarity dimension of the IFS.

We can see that this definition agrees with that given by equation 3.1, when applicable. If $r_i = r$ for each $i = 1, \ldots, m$, then the similarity dimension is the unique s such that

$$\sum_{i=1}^{m} r^s = mr^s = 1,$$

which has solution $\frac{\log m}{\log 1/r}$.

Consider, as an example, the following generalization of the Cantor set. Suppose that r_1 and r_2 are positive numbers satisfying $r_1 + r_2 \leq 1$. We define an iterated function system $\{f_1, f_2\}$ on $\mathbb{R}$ by setting $f_1(x) = r_1 x$ and $f_2(x) = r_2 x + (1 - r_2)$. Note that f_1 contracts the unit interval by the factor $r_1$towards 0, while f_2 contracts the unit interval by the factor r_2 towards 1. If $r_1 + r_2 = 1$, then the invariant set is the unit interval and the dimension is 1. If $r_1 + r_2 < 1$, then this IFS generalizes the Cantor construction. The dimension of this IFS is the unique solution to the equation $r_1^s + r_2^s = 1$. For example if $r_1 = 1/2$ and $r_2 = 1/4$, we obtain

$$\frac{1}{2^s} + \frac{1}{4^s} = 1 \text{or} s = \frac{\log \frac{1+\sqrt{5}}{2}}{\log 2}.$$

This set is shown in figure 3.2.

Not all equations of this form can be solved explicitly. For example, if $r_1 = 1/2$ and $r_2 = 1/3$ then the similarity dimension is the unique solution to $1/2^s + 1/3^s = 1$. While this equation cannot be explicitly solved, it does uniquely characterize the dimension. Furthermore, numerical algorithms like

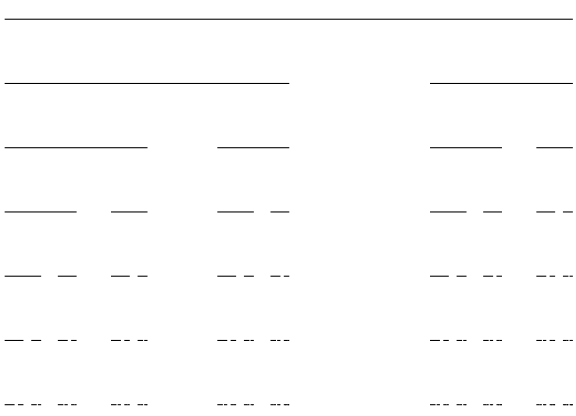


Figure 3.2 The invariant set corresponding to $r_1 = 1/2$ and $r_2 = 1/4$.

the `FindRoot` command can be used to approximate the dimension to a high degree of accuracy. In this case, the dimension can be estimated by the following command.

```
FindRoot[1 / 2^s + 1 / 3^s == 1, {s, 1}]
{s → 0.787885}
```

Another example is provided by the z-curve, which has similarity ratio list $\left\{1/3, \sqrt{2}/6, 1/3, \sqrt{2}/6, 1/3\right\}$. It's dimension is given by the equation

$$\frac{3}{3^s} + 2\left(\frac{\sqrt{2}}{6}\right)^s = 1.$$

The solution is approximately 1.32038.

If $r_1 + r_2 > 1$, then the solution to $r_1^s + r_2^s = 1$ will satisfy $s > 1$. This indicates a potential problem with similarity dimension since we don't want to assign number larger than one to be the dimension of a subset of $\mathbb{R}$.

3.2.3 Box-counting dimension

As mentioned earlier, there are many definitions of fractal dimension, all with their own strengths and weaknesses. The major strength of the similarity dimension is that it is very easily computed. The major weakness of the similarity dimension is that it is very restrictive; there are many sets (even very simple sets) which are simply not self-similar. Another weakness is that many people find it to be non-intuitive when compared to the simple

definition of equation 3.1. The box-counting dimension is a definition whose properties are somewhat complementary to those of the similarity dimension. In particular, it is much more broadly applicable, although harder to compute. Also, it is a more intuitively direct generalization of equation 3.1.

There are several equivalent formulations of box-counting dimension, but all rely on the same idea. Throughout this section, E will denote a non-empty, compact subset of $\mathbb{R}^n$. The *diameter* of such a general set E is defined to be $\max_{x,y\in E} |x-y|$. For $\varepsilon > 0$, an *ε-cover* of E is simply a collection of compact sets of diameter at most ε whose union contains E. Given $\varepsilon > 0$, let $C_\varepsilon(E)$ denote the minimum possible number of sets in an ε-cover of E. While the sets defining $C_\varepsilon(E)$ can be quite arbitrary, they give a measure of how E decomposes into smaller sets not necessarily similar to E. Thus we can think of $C_\varepsilon(E)$ as analogous to N_r in equation 3.1. Of course, the expression

$$\frac{\log C_\varepsilon(E)}{\log 1/\varepsilon}$$

will in general vary with ε. Since we want our definition of dimension to depend upon the finest details of E, it is natural to consider the limit as $\varepsilon \to 0^+$.

Definition 3.4 The box-counting dimension $\dim(E)$ of a non-empty, bounded subset E of $\mathbb{R}^n$ is defined by

$$\dim(E) = \lim_{\varepsilon\to 0^+} \frac{\log C_\varepsilon(E)}{\log 1/\varepsilon},$$

provided this limit exists.

The name box-counting dimension is due to another common formulation. The expression $C_\varepsilon(E)$ is not the only measure of how E decomposes into smaller sets. For $\varepsilon > 0$, the ε-mesh for $\mathbb{R}^n$ is the grid of closed cubes of side length ε with one corner at the origin and sides parallel to the coordinate axes. For $n = 2$, this can be visualized as fine graph paper. Define $N_\varepsilon(E)$ to be the smallest number of ε-mesh cubes whose union contains E. We can define a notion of dimension completely analogous to box-counting dimension using $N_\varepsilon(E)$ in place of $C_\varepsilon(E)$. We will (temporarily) denote the dimension of a set computed in this way by $\dim_b(E)$.

Another useful variation is to consider packings of the set E rather than coverings. Given $\varepsilon > 0$, an ε-packing of E is a collection of closed, disjoint balls of radius ε with centers in E. Define $P_\varepsilon(E)$ to be the maximum possible

number of balls in an ε-packing of E. We can again define a notion of dimension using $P_\varepsilon(E)$ and we (temporarily) denote the dimension of a set computed this way by $\dim_p(E)$.

These ideas are illustrated in figure 3.3. In figure 3.3(a), we see a finite collection of points. Figure 3.3(b) illustrates a covering of those points, figure 3.3(c) illustrates a packing of the points, and figure 3.3(d) illustrates a box-covering of the points.

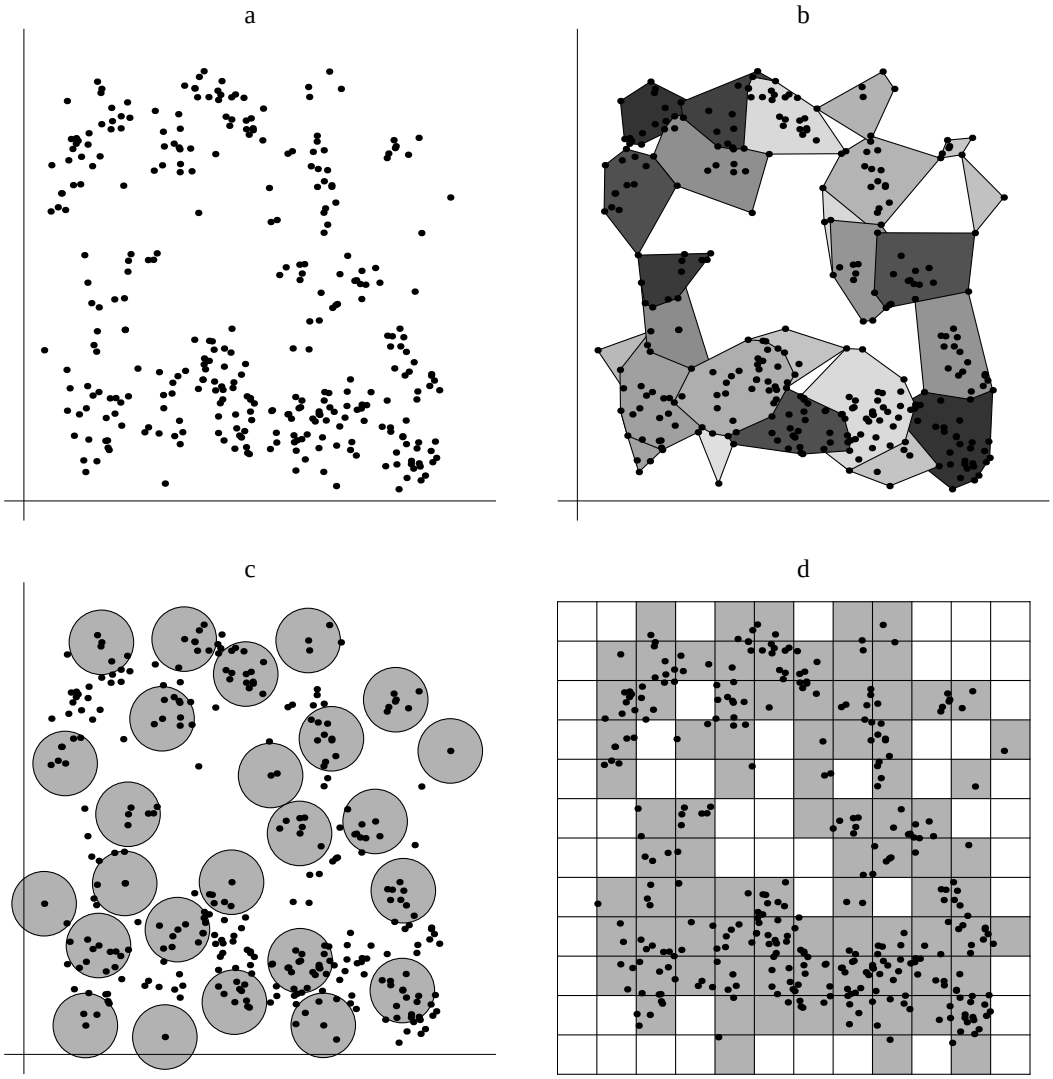


Figure 3.3 A covering, packing, and box covering of a finite set.

The need to understand relationships between dimensions should be clear, given the wide assortment of possible definitions. As we prove in the following theorem, all three definitions of dimension given in this section are equivalent. This is a good situation because it gives us flexibility in computing box-counting dimension. For many sets, $N_\varepsilon(E)$ is the easiest definition to work with. In other situations, $C_\varepsilon(E)$ or $P_\varepsilon(E)$ might be more natural.

Lemma 3.5 *Let E be a non-empty, bounded subset of $\mathbb{R}^n$ and let $\dim_b(E)$, $\dim_c(E)$, and $\dim_p(E)$ denote the dimensions of E defined using $N_\varepsilon(E)$, $C_\varepsilon(E)$, and $P_\varepsilon(E)$ respectively. Then $\dim_b(E) = \dim_c(E) = \dim_p(E)$.*

Proof Assume first that the packing dimension is well defined. In other words

$$\lim_{\varepsilon \to 0^+} \frac{\log P_\varepsilon(E)}{\log 1/\varepsilon}$$

exists. We will show that the covering dimension is also well defined and that $\dim_c(E) = \dim_p(E)$. Suppose we have an ε-packing of E that is maximal in the sense that no more closed ε-balls centered in E can be added without intersecting one of the balls in the packing. Then any point of E is within a distance at most 2ε from some center in the packing; otherwise we could add another ε-ball to the packing. Thus this ε-packing induces a 4ε-cover of E obtained by doubling the radius of any ball in the packing as illustrated in figure 3.4. This says that $C_{4\varepsilon}(E) \leq P_\varepsilon(E)$. It follows that

$$\frac{\log C_{4\varepsilon}(E)}{\log \frac{1}{4\varepsilon}} \leq \frac{\log P_\varepsilon(E)}{\log \frac{1}{4\varepsilon}} = \frac{\log P_\varepsilon(E)}{\log \frac{1}{4} + \log \frac{1}{\varepsilon}}.$$

Next, the centers of the balls of any ε-packing of E are separated by more than 2ε. Thus for any ε-cover of E, different sets are needed for each center of any ε-packing. It follows that $P_\varepsilon(E) \leq C_\varepsilon(E)$ for every ε. Thus we now have

$$\frac{\log P_{4\varepsilon}(E)}{\log \frac{1}{4\varepsilon}} \leq \frac{\log C_{4\varepsilon}(E)}{\log \frac{1}{4\varepsilon}} \leq \frac{\log P_\varepsilon(E)}{\log \frac{1}{4} + \log \frac{1}{\varepsilon}}.$$

The expressions on the left and the right both approach $\dim_p(E)$ as $\varepsilon \to 0^+$. Thus, by the squeeze theorem, the expression in the middle approaches this same value as $\varepsilon \to 0^+$. Of course, this middle limit defines the covering dimension. Thus, the covering dimension exists and $\dim_c(E) = \dim_p(E)$.

If we assume that the covering dimension exists, then a similar argument shows that the packing dimension exists using the inequality

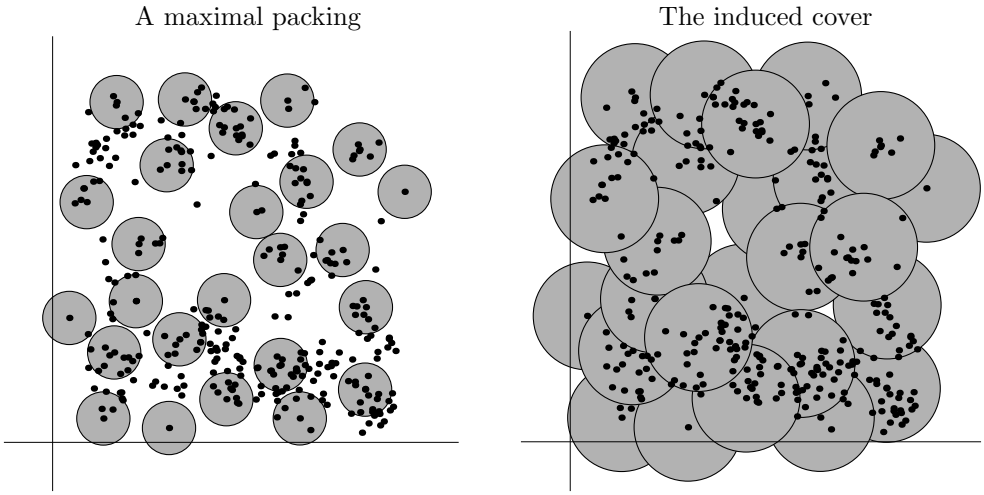


Figure 3.4 A maximal packing and the induced cover.

$$C_{4\varepsilon}(E) \leq P_{\varepsilon}(E) \leq C_{\varepsilon}(E).$$

One can also show that $\dim_c(E) = \dim_b(E)$ using the inequality

$$C_{\sqrt{n}\varepsilon}(E) \leq N_{\varepsilon}(E) \leq 2^n C_{\varepsilon}(E).$$

The details are left as an exercise.

□

In light of lemma 3.5, we will drop the subscripts and refer to the dimension computed using any one of $N_\varepsilon(E)$, $C_\varepsilon(E)$, or $P_\varepsilon(E)$ as the box-counting dimension.

Before looking at an example, we prove another lemma, which simplifies computation considerably.

Lemma 3.6 *Let $(\varepsilon_k)_k$ be a sequence which strictly decreases to zero and for every k satisfies $\varepsilon_{k+1} \geq c\varepsilon_k$, where $c \in (0,1)$ is fixed, and suppose that*

$$\lim_{k\to\infty} \frac{\log C_{\varepsilon_k}(E)}{\log 1/\varepsilon_k} = d.$$

Then $\dim(E)$ is well defined and $\dim(E) = d$.

Proof Given any $\varepsilon > 0$, there is a unique value of k such that $\varepsilon_{k+1} \leq \varepsilon < \varepsilon_k$. Assuming this relationship between ε and k, we have

$$\frac{\log C_\varepsilon(E)}{\log 1/\varepsilon} \leq \frac{\log C_{\varepsilon_{k+1}}(E)}{\log 1/\varepsilon_k} = \frac{\log C_{\varepsilon_{k+1}}(E)}{\log \frac{1}{\varepsilon_{k+1}} + \log \frac{\varepsilon_{k+1}}{\varepsilon_k}} \leq \frac{\log C_{\varepsilon_{k+1}}(E)}{\log \frac{1}{\varepsilon_{k+1}} + \log c}$$

and

$$\frac{\log C_\varepsilon(E)}{\log 1/\varepsilon} \geq \frac{\log C_{\varepsilon_k}(E)}{\log 1/\varepsilon_{k+1}} = \frac{\log C_{\varepsilon_k}(E)}{\log \frac{1}{\varepsilon_k} + \log \frac{\varepsilon_k}{\varepsilon_{k+1}}} \geq \frac{\log C_{\varepsilon_k}(E)}{\log \frac{1}{\varepsilon_k} + \log \frac{1}{c}}.$$

Taking these two inequalities together we have

$$\frac{\log C_{\varepsilon_k}(E)}{\log \frac{1}{\varepsilon_k} + \log \frac{1}{c}} \leq \frac{\log C_\varepsilon(E)}{\log 1/\varepsilon} \leq \frac{\log C_{\varepsilon_{k+1}}(E)}{\log \frac{1}{\varepsilon_{k+1}} + \log c}.$$

Now as $k \to \infty$, the expressions on either side of the inequality both approach d. Furthermore, $\varepsilon \to 0^+$ as $k \to \infty$. Thus, by the squeeze theorem,

$$\lim_{\varepsilon \to 0^+} \frac{\log C_\varepsilon(E)}{\log 1/\varepsilon} = d$$

and this value is $\dim(E)$ by definition.

□

Of particular interest are the geometric sequences $\varepsilon_k = c^k$ where $c \in (0, 1)$. Note that similar results hold for $N_\varepsilon(E)$ and $P_\varepsilon(E)$.

Lemma 3.6 makes it easy to compute the box-counting dimension of the Cantor set. If E is the Cantor set, then $N_{3^{-k}}(E) = 2^k$. Thus,

$$\dim(E) = \lim_{k \to \infty} \frac{\log 2^k}{\log 3^k} = \frac{\log 2}{\log 3}.$$

It has turned out that the box-counting dimension of the Cantor set is the same as its similarity dimension. As we will see in the next section, this is not by chance. This example should not leave the impression that box-counting dimension is easy to compute. An attempt to compute the box-counting dimension of the generalized Cantor sets should make this clear.

3.2.4 Comparing fractal dimensions

We have now defined two notions of dimension. Box-counting dimension is broadly defined and well motivated. Similarity dimension is much more re-

strictive, but very easy to compute when applicable. The main goal in this section is to show that these definitions agree under a reasonable assumption. The advantage of this is two-fold. Similarity dimension can be used to compute the (usually more difficult) box-counting dimension of a self-similar set. Furthermore, we can assume that similarity dimension is a reasonable definition of dimension as it agrees with the box-counting dimension.

We first illustrate the need for an additional assumption concerning the similarity dimension Recall the definition of the generalized Cantor set, where $f_1(x) = r_1 x$ and $f_2(x) = r_2 x + (1 - r_2)$ define an IFS on $\mathbb{R}$. If $r_1 = r_2 = 3/4$, then the similarity dimension of the IFS is $\log(2)/\log(4/3) \approx 2.4 > 1$. This is clearly nonsense since the invariant set is precisely the unit interval which has dimension 1. The problem is that the two images of $[0, 1]$ under the action of the IFS have a significant amount of overlap, thus the IFS does not generate efficient covers of the invariant set. There is an assumption we can place on an IFS which limits this type of overlap.

An IFS $\{f_i\}_{i=1}^m$ on $\mathbb{R}^n$ with invariant set E is said to satisfy the *strong open set condition* if there is an open set U in $\mathbb{R}^n$ such that $U \cap E \neq \emptyset$ and

$$U \supset \bigcup_{i=1}^{m} f_i(U)$$

with this union disjoint. Note that if a generalized Cantor set satisfies $r_1 + r_2 \leq 1$, then the strong open set condition is satisfied by taking U to be the open unit interval. If $r_1 + r_2 > 1$, then the strong open set condition is no longer satisfied.

Theorem 3.7 *Let E be the invariant set of an IFS with similarity dimension s. If the IFS satisfies the strong open set condition, then* $\dim(E) = s$.

Before proving the theorem, we need to develop some useful notation associated with an IFS $\{f_i\}_{i=1}^m$ with ratio list $\{r_i\}_{i=1}^m$, invariant set E, and similarity dimension s. A string with symbols chosen from $\{1, \ldots, m\}$ is simply a finite or infinite sequence with values in $\{1, \ldots, m\}$. A finite string α will be denoted $\alpha = i_1 \cdots i_k$. Finite strings can be concatenated to form longer strings. Thus if $\alpha = i_1 \cdots i_k$ and $\beta = j_1 \cdots j_l$, then $\alpha\beta = i_1 \cdots i_k j_1 \cdots j_l$. Every string $\alpha = i_1 \cdots i_k$ has one parent $\alpha^- = i_1 \cdots i_{k-1}$. Given a positive integer k, let J_k denote the set of all strings of length k with values chosen from the set $\{1, \ldots, m\}$. Let $J_* = \bigcup_{k=1}^{\infty} J_k$ denote the set of all such finite strings. Given $\alpha = i_1 \cdots i_k \in J_k$, let $f_\alpha = f_{i_1} \circ \cdots \circ f_{i_k}$and $r_\alpha = r_{i_1} \cdots \mathrm{r}_{i_k}$. Then J_k induces a k^{th} level decomposition of E given by

$$E = \bigcup_{\alpha \in J_k} f_\alpha(E).$$

The sets J_k are examples of *cross-sections* of J_*, but J_k does not necessarily induce a useful decomposition of J_* with respect to box-counting dimension as the sizes of the sets $f_\alpha(E)$ may vary greatly. A more general type of cross-section may be defined as follows. Let J_ω denote the set of all infinite strings with symbols chosen from $\{1, \ldots, m\}$. Given $\sigma = i_1 i_2 \cdots \in J_\omega$, let $\omega|_k$ denote the element of J_k whose symbols are the first k symbols of σ. Given $\alpha \in J_k$, let $[\alpha] = \{\sigma \in J_\omega : \sigma|_k = \alpha\}$ denote the set of all infinite strings whose first k symbols agree with α. A cross-section of J_* is a finite set $J \subset J_*$ such that $\cup_{\alpha \in J}[\alpha] = J_*$ with this union disjoint. Note that a cross-section J of J_* defines a decomposition of E given by

$$E = \bigcup_{\alpha \in J} f_\alpha(E).$$

Furthermore, if $\alpha \in J_*$, then

$$\sum_{j=1}^{m} r_{\alpha j}^s = r_\alpha^s \sum_{j=1}^{m} r_j^s = r_\alpha^s.$$

It follows by induction that $\sum_{\alpha \in J} r_\alpha^s = 1$ for any cross-section J of J_*.

Now for $\varepsilon > 0$, define J_ε by

$$J_\varepsilon = \{\alpha \in J_* : r_\alpha \mathrm{diam}(E) \leq \varepsilon < r_{\alpha^-} \mathrm{diam}(E)\}.$$

Note that if $\sigma \in J_\omega$, then $\left(r_{\sigma|_k}\right)_k$ is a sequence of positive numbers which is strictly decreasing to zero. Thus there is a unique value of k with $\sigma|_k \in J_\varepsilon$ so J_ε defines a cross-section of J_*. Furthermore, the decomposition of E induced by J_ε has diameters comparable to ε. In particular, if $r = \min\{r_i\}_{i=1}^m$, then

$$r\varepsilon < \mathrm{diam}\left(f_\alpha(E)\right) \leq \varepsilon$$

for all $\alpha \in J_\varepsilon$.

Finally, $\#(J)$ will denote the number of elements in a cross-section. We are now ready to prove our theorem on the comparison of dimensions.

Proof Our strategy is to find constants M_1 and M_2 so that

$$s + \frac{\log M_1}{\log 1/\varepsilon} \leq \frac{\log P_\varepsilon(E)}{\log 1/\varepsilon} \leq \frac{\log C_\varepsilon(E)}{\log 1/\varepsilon} \leq s + \frac{\log M_2}{\log 1/\varepsilon}, \tag{3.2}$$

for then the squeeze theorem applies. Note that the second inequality follows from the fact that $P_\varepsilon(E) \leq C_\varepsilon(E)$ which was established in the proof of our lemma comparing packings and coverings.

The last inequality in equation 3.2 is equivalent to $\varepsilon^s C_\varepsilon(E) \leq M_2$. We will show that this is true for $M_2 = r^{-s}\text{diam}(E)^s$, where $r = \min\{r_i\}_{i=1}^m$ as above. Recall that $C_\varepsilon(E) \leq \#(J_\varepsilon)$ and $r\varepsilon < r_\alpha \text{diam}(E)$. Thus

$$(r\varepsilon)^s C_\varepsilon(E) \leq \sum_{\alpha \in J_\varepsilon} (r_\alpha \text{diam}(E))^{\,s} = \text{diam}(E)^s$$

and $\varepsilon^s C_\varepsilon(E) \leq r^{-s}\text{diam}(E)^s$.

Proof of the first inequality of equation 3.2 is somewhat more difficult and will require the use of the strong open set condition. We will show that there is a constant $M_1 > 0$ such that $(r\varepsilon)^s P_{r\varepsilon}(E) \geq M_1$. A simple change of variables shows that this is equivalent to $\varepsilon^s P_\varepsilon(E) \geq M_1$, which is in turn equivalent to the first inequality of equation 3.2. Let U be the open set specified by the strong open set condition and let $x \in U \cap E$. Choose $\delta > 0$ such that $\overline{B_\delta(x)} \subset U$. Then any cross-section J of J_* induces a packing $\left\{ f_\alpha\left(\overline{B_\delta(x)}\right) : \alpha \in J \right\}$ of E. Now let

$$J_\varepsilon = \{\alpha \in J_* : r_\alpha \delta \leq \varepsilon < r_{\alpha^-}\delta\}.$$

Then $r\varepsilon < r_\alpha \delta \leq \varepsilon$ for all $\alpha \in J_\varepsilon$. Thus $P_{r\varepsilon}(E) \geq \#(J_\varepsilon)$ and

$$\varepsilon^s P_{\text{r}\varepsilon}(E) \geq \sum_{\alpha \in J_\varepsilon} r_\alpha^s \delta^s = \delta^s.$$

Multiplying through by r^s we obtain $(r\varepsilon)^s P_{r\varepsilon}(E) \geq (r\delta)^s \equiv M_1$.

□

3.2.5 Choosing probabilities for the stochastic algorithm

With a firm understanding of fractal dimension, we are now ready to describe a way to choose probabilities for `ShowIFSStochastic` to generate a uniform approximation to invariant set of an IFS. Since a probability list must sum to one, it is easy to guess that $\{{r_1}^s, \ldots, {r_m}^s\}$ is the appropriate choice. Why?

At an intuitive level, the proof of our theorem on the comparison of dimensions states that we need approximately $\#(J_\varepsilon)$ of size ε to cover E.

For the approximation to appear uniform, "pieces" of the set of comparable sizes should have comparable numbers of points. By a piece, we mean $f_\alpha(E)$

3.2.6 Notes

Much of the this chapter has been influenced by Edgar (2009) and Falconer (2003). One notable difference is the use of the strong open set condition. Typically, this is presented in a weaker formulation. An iterated function system is said to satisfy the *open set condition* if there is a non-empty, open set U in $\mathbb{R}^n$ such that

$$U \supset \bigcup_{i=1}^{m} f_i(U)$$

with this union disjoint. Note that we have dropped the requirement that U and the invariant set E of the IFS have non-empty intersection. The strong open set condition was introduced by Lalley (1988) and it can be used to obtain results in more general metric spaces Schief (1996). In $\mathbb{R}^n$, it can be proved that if an IFS satisfies the open set condition, then it automatically satisfies the strong open set condition We have assumed the strong open set condition here since it simplifies our exposition.

Exercises

3.1 Show that $\dim_c(E) = \dim_b(E)$.

3.2 Compute the box dimension of the modified cantor set directly from the definition and the lemma. Hint: Using the sequence $\varepsilon_k = 1/2^k$, compute the first few values of $N_{\varepsilon_k}(F)$. Can you make a conjecture as to what sequence this is? Can you prove your conjecture? There is a well known approximation concerning the growth of this sequence.

4

Generalizing self-similarity

The self-similar sets developed in chapter 2 form an important class of sets for two main reasons; they are complicated enough to display the fine level of detail we expect of fractal sets, yet they are regular enough to allow some level of analysis. Self-similarity is a very restrictive condition, however. In this chapter, we'll explore two natural generalizations of self-similarity - self-affinity and digraph self-similarity.

4.1 Self-affinity

Theorem 3.7, which guarantees the existence of an invariant set for any iterated function system, immediately suggests a natural generalization of self-similarity; we simply consider iterated function systems whose functions are more general types of contractions than pure similarities. For example, an *affine function* or *affinity* is a function which consists of a linear portion together with a translation. A *self-affine set* is simply a set generated by an IFS whose functions are affinities. Although the class of affine functions is still more restrictive that the class of all contractions, there are good reasons to carefully investigate self-affinity. On one hand, self-affinity is such an immediate generalization of self-similarity that we will be able to use the *exact* same *Mathematica* notation to represent an IFS, since all we need to know the matrix and the shift vector. We will see, however, that dimensional analysis of such sets is already significantly more complicated.

Figure 4.1 illustrates an example of a self-affine set. Figure 4.1 (a) portrays the action of an IFS of three similarities on the unit square. The dashed outline of the unit square contains the three shaded images of the square under the action of the IFS. Figures 4.1 (b) and 4.1 (c) show the second and third level approximations and figure 4.1 (d) is a higher level approximation. Note that the key difference between a similarity and an affine function is that

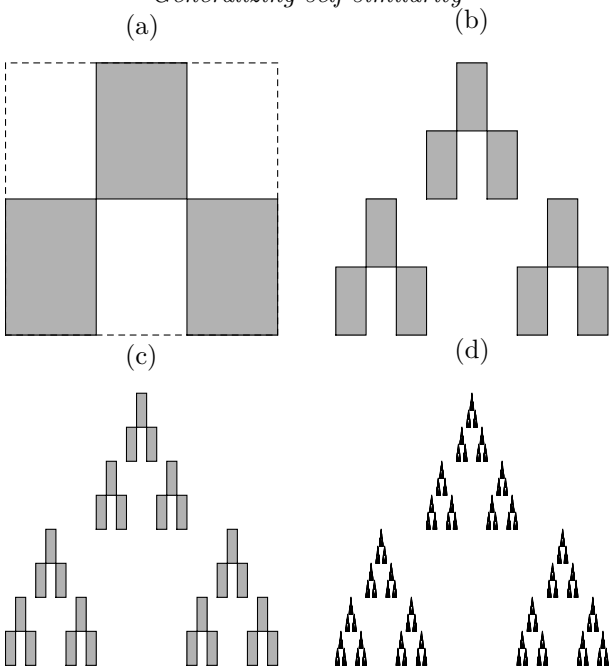


Figure 4.1 Approximations to a self-affine set.

an affine functions may contract by different amounts in different directions. Thus the images of the square may be rectangles, or even parallelograms. In this particular example, the set is generated by the IFS

$$\begin{aligned} f_1(x) &= Mx \\ f_2(x) &= Mx + \begin{pmatrix} 1/3 \\ 1/2 \end{pmatrix} \\ f_3(x) &= Mx + \begin{pmatrix} 2/3 \\ 0 \end{pmatrix} \end{aligned}$$

where

$$M = \begin{pmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}.$$

Of course, affine transformations can rotate and/or reflect an object as well. Figure 4.2 illustrates a self-affine set where reflection is used for one of the transformations. As before, figure 4.2 (a) portrays the action of an IFS

of three similarities on the unit square; this time, however, the initial square has an upward orientation. The image shows how one of the rectangles has been flipped using the matrix

$$\begin{pmatrix} \frac{1}{3} & 0 \\ 0 & -\frac{1}{2} \end{pmatrix}.$$

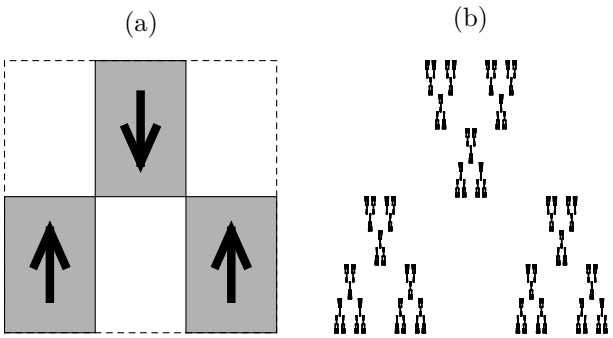


Figure 4.2 Approximations to a self-affine set with reflection

Affine functions can also have a shear effect, which a similarity cannot. Figure 4.3 shows such a set constructed with four affine transformations, two of which involve a shear. The two matrices used in the generation of this set are

$$M_1 = \begin{pmatrix} 1/3 & 0 \\ 0 & 1/4 \end{pmatrix} \text{and} M_2 = \begin{pmatrix} 0.7\cos(\pi/6) & 0.3\cos(\pi/3) \\ 0.7\sin(\pi/6) & 0.7\sin(\pi/3) \end{pmatrix}$$

Our last example, Barnsley's Fern, is shown in figure 4.4 and is truly incredible. The obvious difference between Barnsley's fern and our other examples is its natural appearance. The amazing thing is that it is generated using an IFS with only four affine functions. Since an affine function is specified by six numbers, this means that only 24 numbers are required to encode this very natural looking object.

Barnsley's fern is the invariant set of the following IFS.

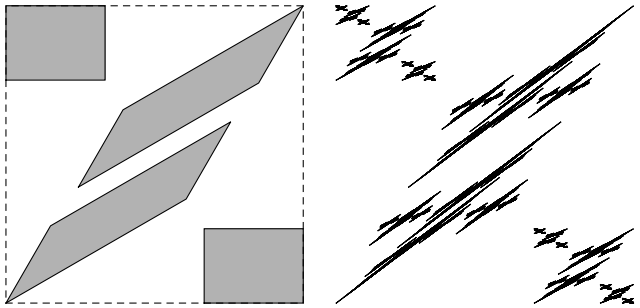


Figure 4.3 Approximations to a self-affine set with shears

Figure 4.4 Barnsley's Fern

$$f_1(x) = \begin{pmatrix} .85 & .04 \\ -.04 & .85 \end{pmatrix} x + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix}$$
$$f_2(x) = \begin{pmatrix} -.15 & .28 \\ .26 & .24 \end{pmatrix} x + \begin{pmatrix} 0 \\ .44 \end{pmatrix}$$
$$f_3(x) = \begin{pmatrix} .2 & -.26 \\ .23 & .22 \end{pmatrix} x + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix}$$
$$f_4(x) = \begin{pmatrix} 0 & 0 \\ 0 & .16 \end{pmatrix} x$$

Figure 4.5 shows the action of the IFS on an outline of the fern. Note that the function f_4 maps all of $\mathbb{R}^2$ onto the y-axis. In particular, f_4 maps the entire fern onto the stem at the bottom of figure 4.5 (b).

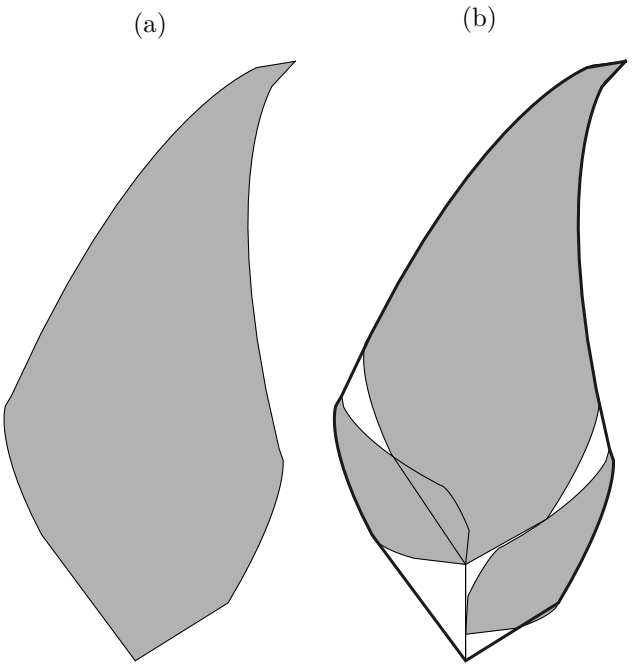


Figure 4.5 Construction of Barnsley's fern

4.1.1 Generating images with the IteratedFunctionSystems package

As mentioned in the introduction, one reason to study self-affinity is that the *exact* same *Mathematica* commands and notation introduced in chapter 2 can be used to generate images. The only difference is that we use more general matrices but the notation for representing these matrices remains unchanged. Since the commands are so similar, we only indicate very briefly how a couple of the images in the previous section were generated.

The fractal in figure 4.1 can be generated using the deterministic `ShowIFS` command as follows.

```
Needs["FractalGeometry`IteratedFunctionSystems`"];
M = {{1 / 3, 0}, {0, 1 / 2}};
IFS = {{M, {0, 0}},
   {M, {1 / 3, 1 / 2}}, {M, {2 / 3, 0}}};
vertices = {{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}};
square = Polygon[{{0, 0}, {1, 0}, {1, 1}, {0, 1}}];
ShowIFS[IFS, 6,
 Initiator → Polygon[{{0, 0}, {1, 0}, {1, 1}, {0, 1}}]]
```

The Barnsley fern shown in figure 4.4 was generated with the stochastic algorithm with probabilities specified.

```
ShowIFSStochastic[{
  {{{.85, .04}, {-.04, .85}}, {0, 1.6}},
  {{{-.15, .28}, {.26, .24}}, {0, .44}},
  {{{.2, -.26}, {.23, .22}}, {0, 1.6}},
  {{{0, 0}, {0, .16}}, {0, 0}}}, 20000,
 Probabilities → {.8, .09, .09, .02}]
```

4.1.2 The Collage Theorem

Barnsley's fern suggests that we might be able to use iterated function systems to approximate natural images. To do so, we would like a method to find an iterated function system whose invariant set is close to a given set. The theoretical tool for doing this is called *the collage theorem.*

Theorem 4.1 *Let $\{f_1, ..., f_m\}$ be an IFS satisfying $|f_i(x) - f_i(y)| \leq r|x - y|$, where $r \in (0, 1)$ is fixed. For an arbitrary compact set F, let*

$$T(F) = \bigcup_{i=1}^{m} f_i(F).$$

Also, let E be the invariant set of the IFS and let d denote the Hausdorff metric. Then for any compact set F,

$$d(E, F) \leq \frac{1}{1-r} d(F, T(F)).$$

In other words, if an IFS doesn't move a particular compact set F too much and the IFS has small contraction ratios, then the set F must be close to the attractor of the IFS. Now suppose we have a compact set A that we would like to approximate with an IFS. We should attempt to find a compact set F which is close to A and an IFS with small contraction ratios which moves F very little in the Hausdorff metric. Then the collage theorem guarantees that the invariant set E of the IFS will be close to F. Since F is close to A, E will be close to A.

As an example, suppose we would like an IFS whose invariant set is approximately the closed unit disk. Although this is not a very natural looking target set, the example is interesting since the disk is manifestly *not* self-similar. We first find a collection of small disks whose union is close to the unit disk in the Hausdorff metric. Such a collection of disks is shown in figure 4.6 (a) which also shows the boundary of the unit disk.

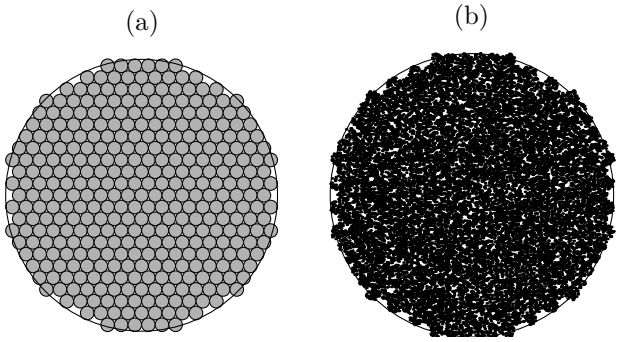


Figure 4.6 An IFS approximating a disk

Note that each small disk in figure 4.6 (a) determines a similarity transformation which maps the whole unit disk onto the small disk. If we assume that the similarity neither rotates nor reflects the image, then the similarity is uniquely determined. Thus figure 4.6 (a) describes an iterated function system. The invariant set of this IFS is rendered with a stochastic algorithm in figure 4.6 (b). As we can see, it is close to the whole disk.

A major problem with the above technique is that it typically yields a large number of transformations. The IFS determined by figure 4.6, for example, has 365 functions. A practical approach which avoids this problem is to try

to approximate the image you want with as small a number of affine copies of itself as possible. Such an approach is illustrated for a disk in figure 4.7.

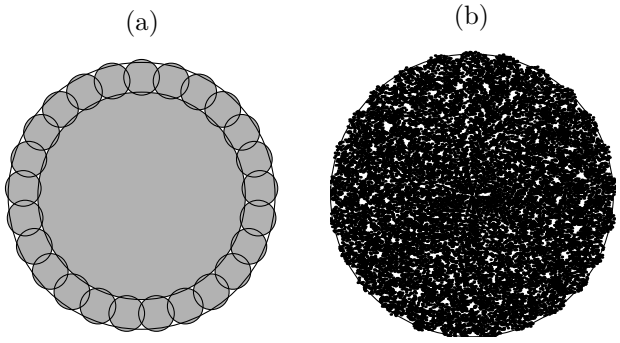


Figure 4.7 An IFS with fewer functions approximating a disk

4.1.3 Dimension of self-affine sets

Computing the fractal dimension of self-affine sets in general is much harder than for self-similar sets, even assuming the open set condition. We'll begin by looking at several examples. Our first example is the set E illustrated in figure 4.8. It consists of four parts scaled by the factor 1/4 in the horizontal direction and 1/2 in the vertical direction.

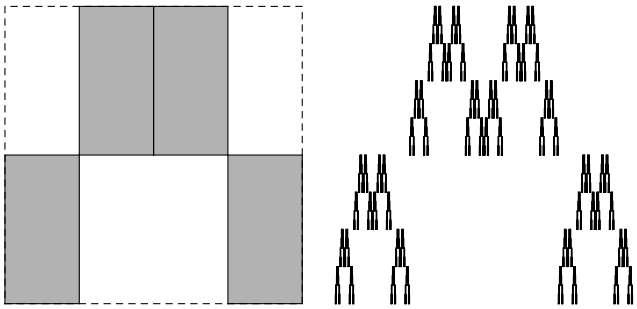


Figure 4.8 A self-affine set to analyze

We will compute the box-counting dimension of E. To do so, we first cover it with rectangles. By applying the IFS to the unit square n times, we see that 4^n rectangles of width $1/4^n$ and height $1/2^n$ can cover E. This is illustrated for $n = 2$ in figure 4.9.

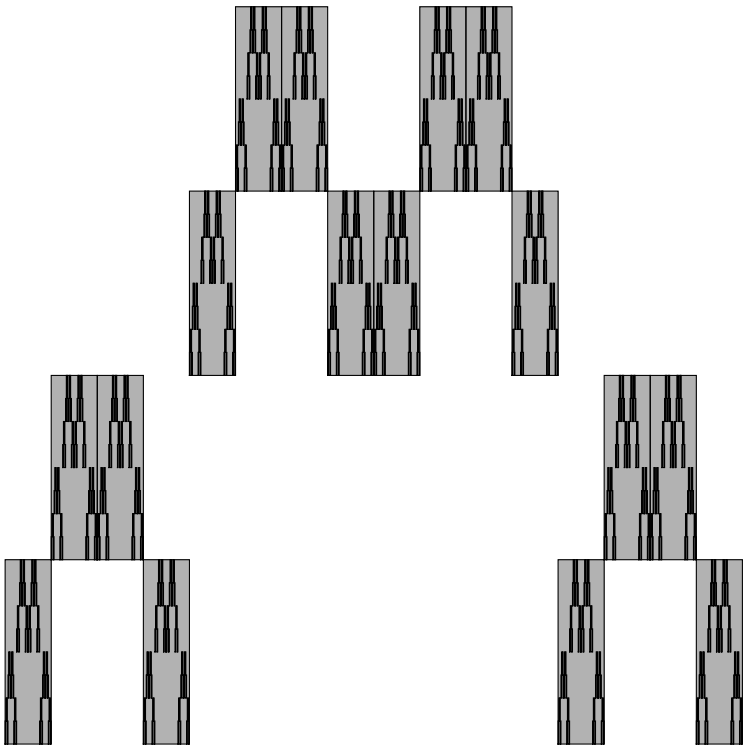


Figure 4.9 Covering the self-affine set with boxes

We need to translate this cover by rectangles into a cover by ε mesh squares. A rectangle of width $1/4^n$ and height $1/2^n$ can be decomposed into a stack of 2^n squares of side length $1/4^n$, for then the total height of the stack is $2^n/4^n = 1/2^n$. We now take $N_\varepsilon(E)$ to denote the number of open ε mesh squares that intersect E. Since the n^{th} level approximation generated by the IFS yields 4^n rectangles that can each be decomposed into 2^n squares of side length $1/4^n$, we compute $N_{4^{-n}}(E) = 2^n 4^n = 8^n$.

$$\dim(E) = \lim_{n\to\infty} \frac{\log\left(N_{4^{-n}}(E)\right)}{\log\left(4^n\right)} = \lim_{n\to\infty} \frac{\log\left(8^n\right)}{\log\left(4^n\right)} = \frac{3}{2}.$$

There are a couple of technical points that have been glossed over in this exposition. First, we explicitly stated that $N_\varepsilon(E)$ denotes the number of *open* ε mesh squares that intersect E. This allows us to ignore those squares that intersect the set E at their boundary. As mentioned in chapter 3, this definition is within a constant multiple of the result obtained by using *closed*

ε mesh squares and, therefore, yields the correct dimension computation. The other point is more subtle. Clearly, each rectangle generated by the IFS intersects the invariant set E but how do we know that each of the 2^n sub-squares into which these rectangles were decomposed intersect the set E? For this particular example, we can prove that the projection of E onto the y-axis is the unit interval, since this is clearly true for each approximation to the set. Using the self-affine structure of the set it can be shown that each of those 2^n sub-squares indeed intersects the set.

To outline an approach for a slightly harder example, consider the self-affine set illustrated back in figure 4.1. We now denote this set by E, which consists of 3^n pieces scaled by the factor 3^{-n} in the horizontal direction and 2^{-n} in the vertical direction. Now, there is no single natural choice of a sequence ε_n to use in the computation of N_ε. Nonetheless, we can estimate $C_{\sqrt{2}3^{-n}}(E)$, the number of sets of diameter $\sqrt{2}3^{-n}$ required to cover E, by using squares of side length 3^{-n}. Now, the IFS generates 3^n rectangles of width 3^{-n} and height 2^{-n} that cover E. In turn, these can be covered with at most $\lceil 3^n/2^n \rceil + 1$ squares of side length 3^n. Thus, $C_{\sqrt{2}3^{-n}}(E) \leq 3^n\left(\lceil 3^n/2^n \rceil + 1\right)$ so

$$\dim(E) = \lim_{n\to\infty} \frac{\log\left(C_{\sqrt{2}3^{-n}}(E)\right)}{\log\left(3^n/\sqrt{2}\right)} \leq \lim_{n\to\infty} \frac{\log\left(3^n\left(\lceil 3^n/2^n \rceil + 1\right)\right)}{\log\left(3^n/\sqrt{2}\right)} = \frac{\log(9/2)}{\log(3)}.$$

To generate (the same) lower bound for the dimension, we consider packings by balls of radius $3^{-n}/2$ with centers in E. Each rectangle generated by the IFS as shown in figure 4.1 stretches vertically along a y interval of the form $\left[\frac{i}{2^n}, \frac{i+1}{2^n}\right]$. As in the previous example, the projection of the portion of E contained in this rectangle onto the y-axis is precisely this interval. Thus, we can find points in the intersection of E and this rectangle with y-coordinates of the form $\frac{i}{2^n} + \frac{j}{3^n}$ for $j = 1, 2, \ldots, \lfloor 3^n/2^n \rfloor$. Thus, we can pack at least $\lfloor 3^n/2^n \rfloor$ open balls of radius $3^n/2$ into this portion of E. A glance at figure 4.1 indicates that we need not worry about lateral intersections between balls coming from different rectangles. Thus, $P_{3^{-n}/2}(E) \geq \lfloor 3^n/2^n \rfloor$ so

$$\dim(E) = \lim_{n\to\infty} \frac{\log\left(P_{3^{-n}/2}(E)\right)}{\log\left(23^n\right)} \geq \lim_{n\to\infty} \frac{\log\left(3^n\left(\lfloor 3^n/2^n \rfloor\right)\right)}{\log\left(23^n\right)} = \frac{\log(9/2)}{\log(3)}.$$

Taking these two results together, we see that $\dim(E) = \log(9/2)/\log(3) \approx 1.369$.

Clearly, it would be nice to have theorem analogous to theorem 3.7 that

would allow us to compute the dimension of a self-affine set using a simple formula. Ideally, the formula should hold under reasonably simple hypotheses. While we'll meet such a formula in the next section, it is not nearly as broadly applicable as theorem 3.7. In our next example, we'll see that, even if we assume the strong open set condition, that no simple formula can hold.

Let M be the matrix

$$M = \begin{pmatrix} 1/3 & 0 \\ 0 & 1/2 \end{pmatrix}$$

and consider the IFS with two affine functions

$$f_1(x) = Mx + \begin{pmatrix} 0 \\ r \end{pmatrix}$$
$$f_2(x) = Mx + \begin{pmatrix} 2/3 \\ 0 \end{pmatrix}$$

where r is a real parameter. The invariant set of this IFS for several choices of r is illustrated in figure 4.10. The dark vertical line segment is the projection of the set onto the y-axis. There are two key observations to make here. First, for any $r > 0$, this projection is a line segment; thus, the dimension of the set must be at least one. Second, for $r = 0$, the invariant set is the Cantor set with dimension $\log(2)/\log(3)$, which is strictly less than one. Thus the dimension does not vary continuously with the parameter r so we can't expect the dimension to be given by a simple, continuous function of r.

4.1.4 Falconer dimension

A technique to estimate the box-counting dimension of self-affine sets was discovered by Kenneth Falconer. It's complicated and it doesn't always yield the exact value. It does yield an upper bound though and, in certain special cases, it yields the exact value.

Falconer's technique is expressed in terms of the *singular values* of the matrices used in the IFS. We can provide a geometric description of the singular values as follows. Let $B = \{x \in \mathbb{R}^n : |x| \leq 1\}$ denote the closed unit ball in $\mathbb{R}^n$. If we apply the linear transformation induced by the matrix A to B we generate an ellipsoid. The singular values of A are the lengths of the semi-major axes of this ellipsoid. Thus, they are a good measure of how A distorts space in different directions and, as we saw in the previous sub-section, this is exactly the information we need to compute the box

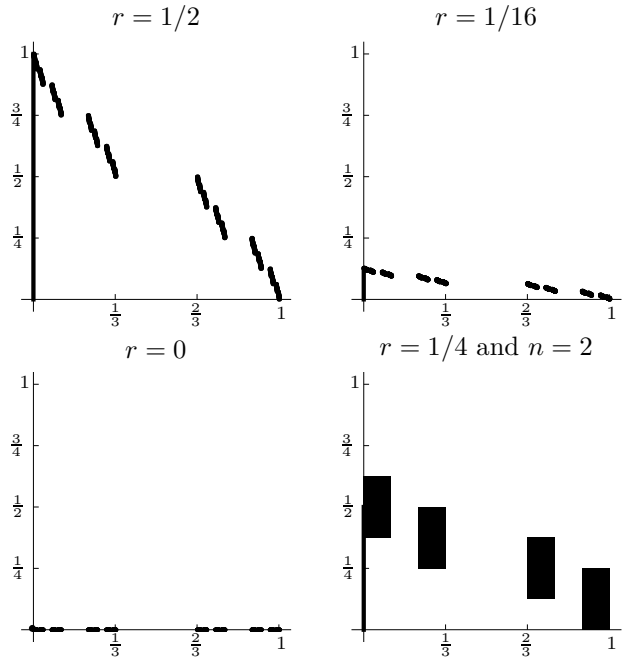


Figure 4.10 Family of self-affine sets dependent upon a parameter

counting dimension of the invariant set. Algebraically, the singular values may be computed as the square roots of the eigenvalues of the matrix $A^T A$. The singular values of an affine function are simply the singular values of the linear part of that function. It is customary to write the singular values in a list $\alpha_1, \alpha_2, \ldots, \alpha_n$ where $\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_n \geq 0$. In the context of iterated function systems, we also have $\alpha_1 < 1$ since all our functions are contractions.

Using the singular values, we can define the *singular value function*. Let f be an affine function mapping $\mathbb{R}^n$ to $\mathbb{R}^n$, let s satisfy $0 \leq s \leq n$ and let r be the integer such that $r - 1 < s \leq r$. Define $\phi^s(f)$ by

$$\phi^s(f) = \alpha_1 \alpha_2 \cdots \alpha_{r-1} \alpha_r^{s-r+1}. \tag{4.1}$$

Our examples will be in the plane with $n = 2$. In this context, we will write the singular values as α and β with $\alpha > \beta$. We will also frequently be interested in the situation where $s > 1$. Thus, equation 4.1 will often simplify to

$$\phi^s(f) = \alpha \beta^{s-1}. \tag{4.2}$$

Now we have an iterated function system $\{f_1, \ldots, f_m\}$, where $f_i(x) = A_i x + b_i$. As in chapter 3, let J_k denote the set of all sequences of integers chosen from $\{1, \ldots, m\}$. Falconer proved that there is a unique number s so that

$$\lim_{k\to\infty} \left(\sum_{(i_1,\ldots,i_k)\in J_k} \phi^s \left(f_{i_1} \circ \cdots \circ f_{i_k} \right) \right)^{1/k} = 1 \tag{4.3}$$

and, furthermore, that this number s is an upper bound for the box dimension of the set E. This number is now called the Falconer dimension of the set. While Falconer dimension is not necessarily equal to the box-counting dimension, we will discuss hypotheses that force equality after looking at some examples.

Finding Falconer dimension in the plane

Unfortunately, Falconer dimension is theoretically complicated and practically hard to compute. While we won't prove Falconer's results, we will look at several example that, hopefully, provide some level of understanding. These examples exploit some of the following simplifications that can happen when dealing with self affine sets in the plane.

- Since the examples live in $\mathbb{R}^2$ these sets have dimension at most 2. Furthermore, using projection techniques, we can frequently show that the dimension of the attractor is at least 1. These properties imply that ϕ^s will have the specific form mentioned in formula 4.2.
- The matrices in the associated iterated function systems are diagonal. Of course, their products will also be diagonal; thus, it is easy to compute the singular values.
- In the computation of the singular value function using equation 4.2, we need to know be able to systematically tell which singular value is larger. This will be easy in all but the last example, since the matrices will have the specific form

  $$A_i = \begin{pmatrix} a_i & 0 \\ 0 & d_i \end{pmatrix},$$

  with $d_i < a_j$ for each i and j. In particular, the singular values of A_j and A_j will be $a_i a_j$ and $d_i d_j$, with $a_i a_j < d_i d_j$.

For our first example, let's revisit the self-affine set shown in figure 4.1, whose fractal dimension we've already computed to be $\log(9/2)/\log(3)$. All three of the function in the IFS have linear part

$$\begin{pmatrix} 1/3 & 0 \\ 0 & 1/2 \end{pmatrix}.$$

Thus, it is not hard to show that for any sequence $(i_1, i_2, \ldots, i_k) \in J_k$, $\phi^s\left(f_{i_1} \circ \cdots \circ f_{i_k}\right) = 1 \Big/ \left(2^k 3^{k(s-1)}\right)$.
Furthermore, $|J_k| = 3^k$. Thus

$$\left(\sum_{(i_1,\ldots,i_k)\in J_k} \phi^s\left(f_{i_1} \circ \cdots \circ f_{i_k}\right)\right)^{1/k} = \left(3^k \Big/ \left(2^k 3^{k(s-1)}\right)\right) = \frac{1}{2}\frac{1}{3^{s-2}}.$$

Setting this equal to one and solving for s, we get $s = \log(9/2)/\log(3)$.

Next, let's look at an example whose features make it just a bit harder but is still approachable using Falconer's technique. We'll consider the IFS

$$\begin{aligned} f_1(x) &= \begin{pmatrix} 1/3 & 0 \\ 0 & 3/4 \end{pmatrix} x \\ f_2(x) &= \begin{pmatrix} 1/3 & 0 \\ 0 & 1/2 \end{pmatrix} x + \begin{pmatrix} 1/3 \\ 1/2 \end{pmatrix} \\ f_3(x) &= \begin{pmatrix} 1/3 & 0 \\ 0 & 3/4 \end{pmatrix} x + \begin{pmatrix} 2/3 \\ 0 \end{pmatrix}. \end{aligned}$$

The invariant set for this IFS is shown in figure 4.11.

Denote the singular values of each function f_i in the IFS by α_i and β_i, where $1 > \alpha_i \geq \beta_i > 0$. Thus,

$$\alpha_1 = \alpha_3 = \frac{3}{4}, \alpha_2 = \frac{1}{2}, \text{and} \beta_1 = \beta_2 = \beta_3 = \frac{1}{3}.$$

Since $\alpha_i \geq \beta_j$ for every $i, j = 1, \ldots, 3$, it's a *relatively* simple matter to compute ϕ^s:

$$\phi^s\left(f_{i_1} \circ \cdots \circ f_{i_k}\right) = \left(\frac{1}{2}\right)^{n_2} \left(\frac{3}{4}\right)^{n_1+n_3} \left(\frac{1}{3}\right)^{k(s-1)},$$

where n_i is the number of occurrences of i in the sequence $(i_1, \ldots, i_k)$. Thus

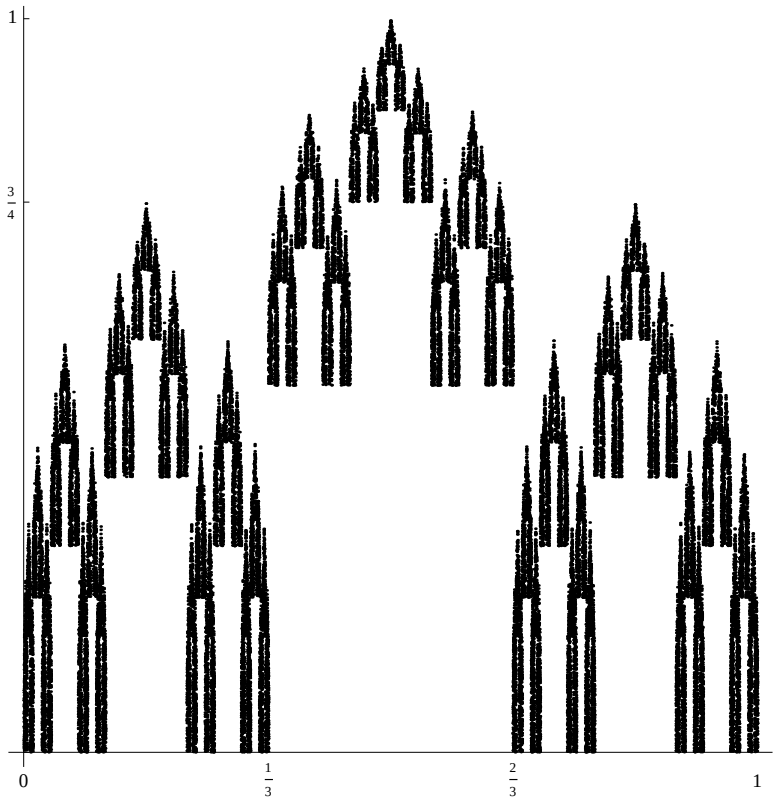


Figure 4.11 A self-affine set for Falconer's formula

$$\begin{aligned}\sum_{(i_1,\ldots,i_k)\in J_k} \phi^s\left(f_{i_1}\circ\cdots\circ f_{i_m}\right) &= \sum_{(i_1,\ldots,i_k)\in J_k} \left(\frac{1}{2}\right)^{n_2}\left(\frac{3}{4}\right)^{n_1+n_3}\left(\frac{1}{3}\right)^{k(s-1)}\\ &= \left(\frac{1}{3}\right)^{k(s-1)} \sum_{(i_1,\ldots,i_k)\in J_k} \left(\frac{1}{2}\right)^{n_2}\left(\frac{3}{4}\right)^{n_1+n_3}\\ &= \left(\frac{1}{3}\right)^{k(s-1)}\left(\frac{1}{2}+\frac{3}{4}+\frac{3}{4}\right)^{k}.\end{aligned}$$

This last computation follows by simply expanding $(1/2 + 3/4 + 3/4)^k$. We now have

$$\left(\sum_{(i_1,\ldots,i_k)\in J_k} \phi^s\left(f_{i_1}\circ\cdots\circ f_{i_m}\right)\right)^{1/k} = \left(\frac{1}{2}+\frac{3}{4}+\frac{3}{4}\right)\left(\frac{1}{3}\right)^{s-1} = 2\left(\frac{1}{3}\right)^{s-1}.$$

Since this expression is constant, it's pretty easy to find the exact s so that $2/3^{s-1} = 1$. In fact, $s = 1 + \frac{\log 2}{\log 3} \approx 1.63$.

Finally, we look at a somewhat harder example - namely

$$\begin{aligned} f_1(x) &= \begin{pmatrix} 1/3 & 0 \\ 0 & 3/4 \end{pmatrix} x \\ f_2(x) &= \begin{pmatrix} 1/3 & 0 \\ 0 & 1/2 \end{pmatrix} x + \begin{pmatrix} 1/3 \\ 1/2 \end{pmatrix} \\ f_3(x) &= \begin{pmatrix} 1/3 & 0 \\ 0 & 3/4 \end{pmatrix} x + \begin{pmatrix} 2/3 \\ 0 \end{pmatrix}. \end{aligned}$$

The attractor is shown in figure 4.12.

The essential difference here is that f_2 compresses more in the vertical direction than in the horizontal, while the reverse is true of f_1 and f_3. Algebraically, we again have matrices of the form

$$A_i = \begin{pmatrix} a_i & 0 \\ 0 & d_i \end{pmatrix},$$

but now $a_1 \geq d_1$ while $a_2 \leq d_2$. Thus the value of $\phi^s(A_1A_2)$ is not immediately clear. The determination of an expression of the form $\phi^s\left(f_{i_1}\circ\cdots\circ f_{i_k}\right)$ can be quite difficult. In this particular example, the singular values α_i and β_i, satisfying $1 > \alpha_i \geq \beta_i > 0$ are

$$\alpha_1 = \alpha_3 = \frac{3}{4}, \alpha_2 = \frac{1}{3}, \text{and} \beta_1 = \beta_3 = \frac{1}{3} \text{and} \beta_2 = \frac{1}{4}.$$

While we will not be able to find a closed form expression for ϕ^s in this case, a good upper bound for the dimension can still be obtained. Define s_k to be the unique value of s such that

$$\sum_{(i_1,\ldots,i_k)\in J_k} \phi^s\left(f_{i_1}\circ\cdots\circ f_{i_m}\right) = 1.$$

It turns out that the Falconer dimension is the infimum of all the s_k [REF NEEDED]. We can find s_1 as follows.

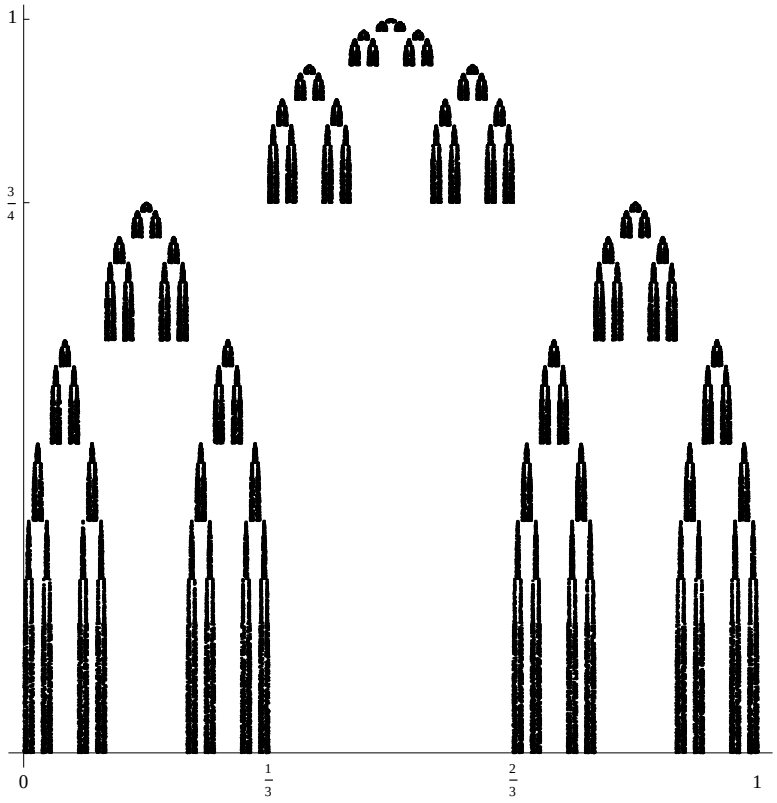


Figure 4.12 A harder self-affine set for Falconer's formula

```
FindRoot[3/4 1/3^(s-1) + 1/3 1/4^(s-1) + 3/4 1/3^(s-1) == 1, {s, 1}]
{s → 1.5281}
```

Thus, $s = 1.53$ is an upper bound for the Falconer dimension and, therefore, the box-counting dimension. Finding values for higher level s_ks is harder; it seems to prudent to use some computational tool. Here's how we can do so for this example. First we define the list of matrices from the IFS. Note that we have one matrix for each function, repeating the matrices as necessary.

```
matrices = {
   {{1 / 3, 0}, {0, 3 / 4}},
   {{1 / 3, 0}, {0, 1 / 4}},
   {{1 / 3, 0}, {0, 3 / 4}}};
```

Then, the following code computes the first 8 s_ks.

```
Table[
 expandedMatrices = Dot @@@ Tuples[matrices, k];
 svs = SingularValueList /@ expandedMatrices;
 dimensionFormula[s_] := Total[Max[#]^s & /@ svs] /; s ≤ 1;
 dimensionFormula[s_] := Total[Max[#] Min[#]^(s - 1) & /@ svs] /; s > 1;
 s /. FindRoot[dimensionFormula[s] == 1, {s, 1}],
 {k, 1, 8}]
{1.5281, 1.51238, 1.50985, 1.50951, 1.50945, 1.5094, 1.50939, 1.50939}
```

It appears that the Falconer dimension is about 1.50939.

4.2 Digraph Iterated Function Systems

Self-affinity generalizes self-similarity by expanding the class of functions in the iterated function system. In this section, we'll generalize the concept of an iterated function system itself. Rather than considering a single set composed of copies of itself, we consider a collection of sets each composed of copies of sets chosen from the collection. As an example, consider the two curves K_1 and K_2 shown in figure 4.13. The curve K_1 is composed of 1 copy of itself, scaled by the factor $1/2$, and 2 copies of K_2, rotated by $\pm\pi/3$ and scaled by the factor $1/2$. The curve K_2 is composed of 1 copy of itself, scaled by the factor $1/2$, and 1 copy of K_1, reflected and scaled by the factor $1/2$. The curves K_1 and K_2 are called *digraph self-similar sets* and may be described using a *digraph iterated function system* or *digraph IFS*.

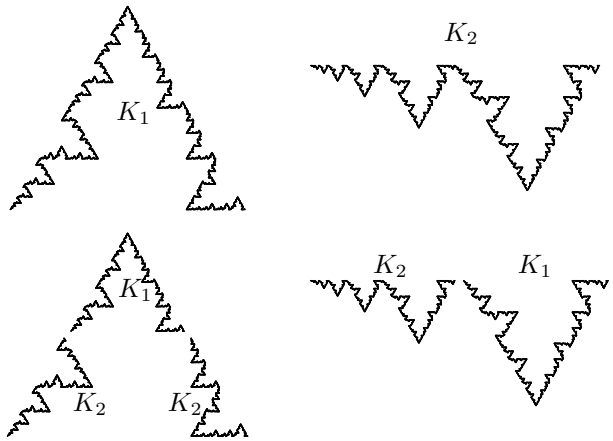


Figure 4.13 Digraph self-similar curves

The first ingredient to define a digraph IFS is a *directed multi-graph*. A directed multi-graph G consists of a finite set V of vertices and a finite set E of directed edges between vertices. We call G a multi-graph because we

allow more than one edge between any two vertices. Figure 4.14 shows the directed multi-graph for the curves K_1 and K_2. There are two edges from vertex 1 to vertex 2 and one edge from vertex 1 to itself since K_1 consists of two copies of K_2 together with one copy of itself. Similarly, there is one edge from vertex 2 to vertex 1 and one edge from vertex 2 to itself since K_2 consists of one copy of K_1 together with one copy of itself.

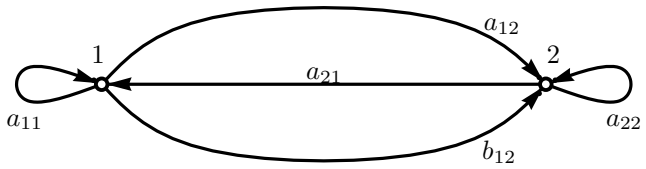


Figure 4.14 The digraph for the curves

Note that the edges in figure 4.14 are labeled. The subscript of the label indicates the initial and terminal vertices. We obtain a *digraph IFS* from a directed multi-graph by associating an affine function f_e with each edge e of the digraph. The digraph IFS for the digraph curves K_1 and K_2 is generated from figure 4.13 by associating the following affine functions with the edges of the digraph.

$$f_{a_{11}}(x) = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} x + \begin{pmatrix} 1/4 \\ \sqrt{3}/4 \end{pmatrix}$$

$$f_{a_{12}}(x) = \begin{pmatrix} 1/4 & -\sqrt{3}/4 \\ \sqrt{3}/4 & 1/4 \end{pmatrix} x$$

$$f_{b_{12}}(x) = \begin{pmatrix} 1/4 & \sqrt{3}/4 \\ -\sqrt{3}/4 & 1/4 \end{pmatrix} x + \begin{pmatrix} 3/4 \\ \sqrt{3}/4 \end{pmatrix}$$

$$f_{a_{21}}(x) = \begin{pmatrix} 1/2 & 0 \\ 0 & -1/2 \end{pmatrix} x + \begin{pmatrix} 1/2 \\ 0 \end{pmatrix}$$

$$f_{a_{22}}(x) = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} x$$

The functions associated with the edges indicate how the large sets map onto the constituent parts. For example, the function $f_{a_{12}}$ maps the set K_2 onto the lower left portion of the set K_1. To make this type of statement for the general digraph IFS, we need to develop some notation associated with digraph iterated function systems. Given two vertices, u and v, we denote the set of all edges from u to v by E_{uv}. A *path* through G is a finite sequence

of edges so that the terminal vertex of any edge is the initial vertex of the subsequent edge. A *loop* through G is a path which starts and ends at the same vertex. G is called *strongly connected* if for every u and v in V, there is a path from u to v. We denote the set of all paths of length n with initial vertex u by E_u^n. A digraph IFS is called *contractive* if the product of the scaling factors in any closed loop is less than 1. Theorem 4.3.5 of [Edgar 1990] states that for any contractive digraph IFS, there is a unique set of non-empty, compact sets K_v, one for every v in V, such that for every u in V,

$$K_u = \bigcup_{\mathrm{v}\epsilon\mathrm{V},\mathrm{e}\epsilon\mathrm{E}_{\mathrm{uv}}} f_e\left(K_v\right). \tag{4.4}$$

Such a collection of sets is called the *invariant list* of the digraph IFS and we will refer to its members as *digraph fractals*. In general, if $e \in E_{\mathrm{uv}}$, then f_e maps K_v into K_u. Thus the functions of the digraph IFS map against the orientation of the edges. Also note that an IFS is the special case of a digraph IFS with one vertex and theorem 4.3.5 of [Edgar 1990] is a generalization of the statement that a unique invariant set exists for a contractive IFS.

To clarify equation 4.4, let's write it down for the specific case of the digraph curves K_1 and K_2. There is one equation for each vertex, thus in this case, equation 4.4 denotes a pair of equations. Substituting $u = 1$ then $u = 2$ we obtain the two equations

$$K_1 = f_{a_{11}}\left(K_1\right) \bigcup f_{a_{12}}\left(K_2\right) \bigcup f_{b_{12}}\left(K_2\right)$$
$$K_2 = f_{a_{21}}\left(K_1\right) \bigcup f_{a_{22}}\left(K_2\right)$$

4.2.1 The Digraph Fractals package

Digraph fractal images may be generated using the `DigraphFractals` package. Of course, we first need to load the package.

```
Needs["FractalGeometry`DigraphFractals`"]
```

We will need an appropriate data structure to describe a digraph IFS. For this purpose, we will use the concept of an *adjacency matrix*. The adjacency matrix representation of a digraph G is a matrix whose rows and columns are indexed by the vertices of G. The entry in row u and column v is a number indicating the number of edges of G from u to v. For example, the digraph in figure 4.14 would have adjacency matrix

$$\begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}.$$

To represent a digraph IFS we need to consider how the pieces of the invariant sets fit together. So the entry in row u and column v of our adjacency matrix representation will be a list of the functions mapping E_v into E_u. For our digraph curves K_1 and K_2, the digraph IFS has matrix representation

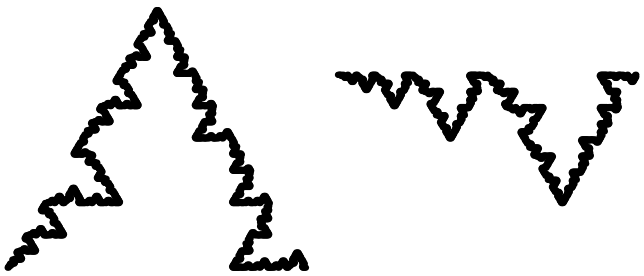
$$\begin{pmatrix} \{f_{a_{11}}\} & \{f_{a_{12}}, f_{b_{12}}\} \\ \{f_{a_{21}}\} & \{f_{a_{22}}\} \end{pmatrix}.$$

To write this in *Mathematica* we use the usual representation of a matrix, which is most easily entered using the BasicMathInput palette. Thus here is the code describing the digraph IFS for the curves K_1 and K_2.

```
a11 = {{{1 / 2, 0}, {0, 1 / 2}}, {1 / 4, Sqrt[3] / 4}};
a12 = {1 / 2 RotationMatrix[π / 3], {0, 0}};
b12 = {1 / 2 RotationMatrix[-π / 3], {3 / 4, Sqrt[3] / 4}};
a21 = {{{1 / 2, 0}, {0, -1 / 2}}, {1 / 2, 0}};
a22 = {{{1 / 2, 0}, {0, 1 / 2}}, {0, 0}};
curvesDigraph = ({a11}  {a12, b12}
                 {a21}     {a22} );
```

We can now use the `ShowDigraphFractals` command.

```
GraphicsRow[ShowDigraphFractals[curvesDigraph, 8]]
```



The algorithm which generates these pictures is very similar to the one which works for a standard IFS. We start with a list of initial approximations to the invariant sets, $\{K_{v,0} : v \in V\}$. By default, each $K_{v,0}$ is taken to be the single point at the origin. We then recursively define a sequence of lists of approximations $\{K_{v,n} : v \in V\}_{n=1}^{\infty}$by

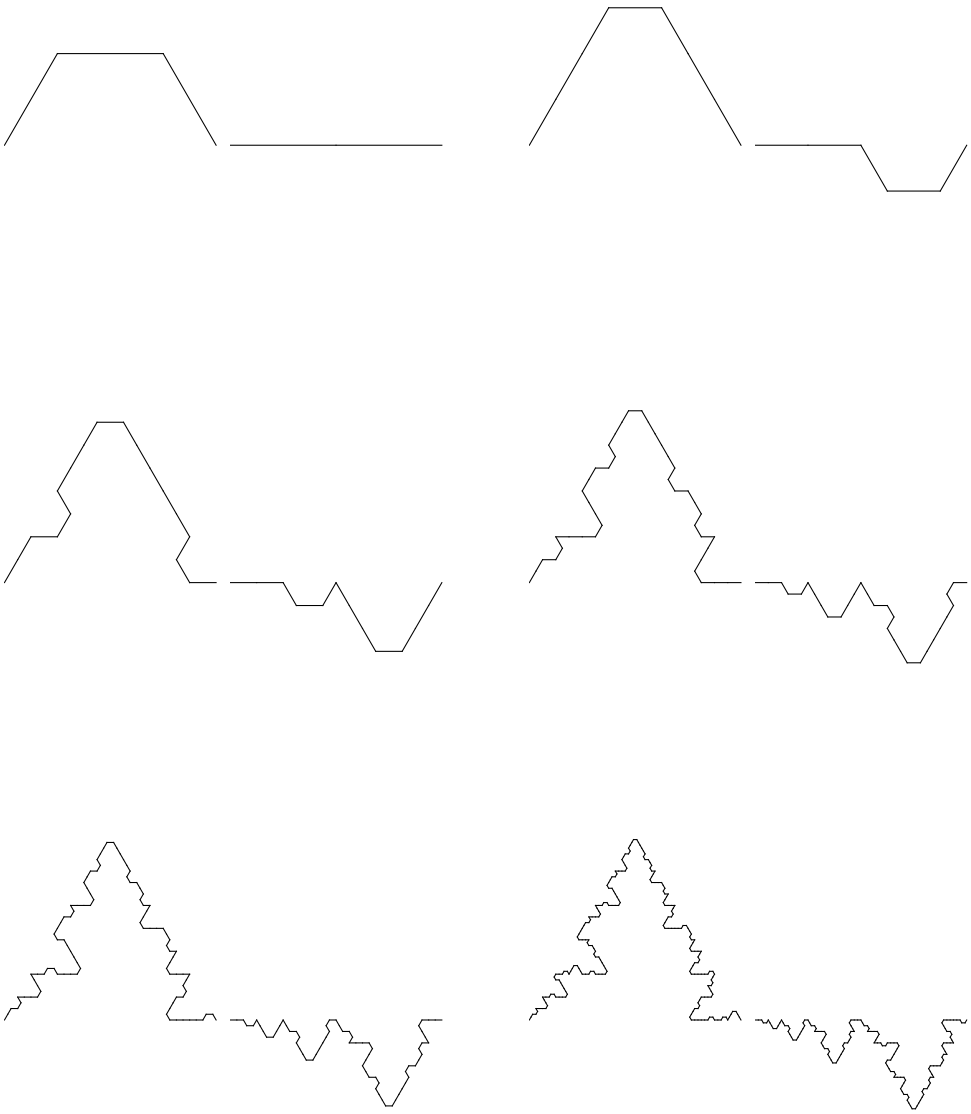
$$K_{u,n+1} = \bigcup_{\mathrm{v}\epsilon\mathrm{V},\mathrm{e}\epsilon\mathrm{E}_{\mathrm{uv}}} f_e\left(K_{v,n}\right).$$

We can see this process in action by plotting a list of the first few approximations to the digraph curves. In order to make the approximations look like curves, we choose the unit interval to be the initiator for both $K_{1,0}$ and $K_{2,0}$. We do this by using the `Initiators` option, which should be a list of lists of `Graphics` primitives.

```
lineSegment = Line[{{0, 0}, {1, 0}}];
initiators = {{lineSegment}, {lineSegment}};
```

We now use the `Table` command to generate a sequence of approximations and display these in a `GraphicsGrid`.

```
GraphicsGrid[Partition[GraphicsRow /@ Table[
    ShowDigraphFractals[curvesDigraph, n,
     Initiators → initiators,
     PlotRange → {{0, 1}, {-.5, .9}}], {n, 0, 5}], 2],
 Spacings → {Scaled[0.3], Scaled[0]}]
```



4.2.2 Examples

As a second example, we'll generate the so called golden rectangle fractals. This interesting pair shows that not all functions in the digraph IFS need be contractions. It also turns out to be a natural pair to generate with a stochastic algorithm. The golden rectangle fractals are based on the observation that a golden rectangle and a square form a digraph pair as indicated in figure 4.15.

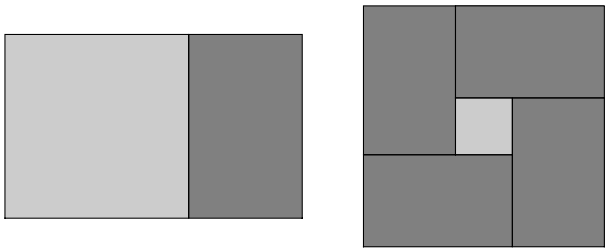


Figure 4.15 Decomposition of a golden rectangle and square into a digraph pair.

Now suppose we modify the decomposition shown in figure 4.15 by deleting one of the rectangles as shown in figure 4.16.

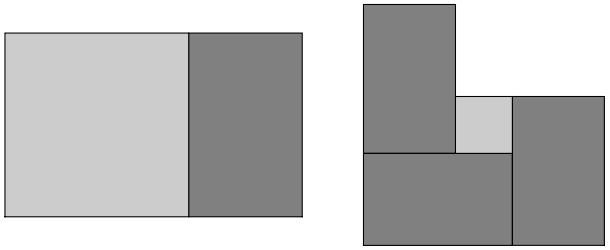


Figure 4.16 A modification of the square

The digraph IFS suggested by figure 4.16 may be encoded in *Mathematica* as follows.

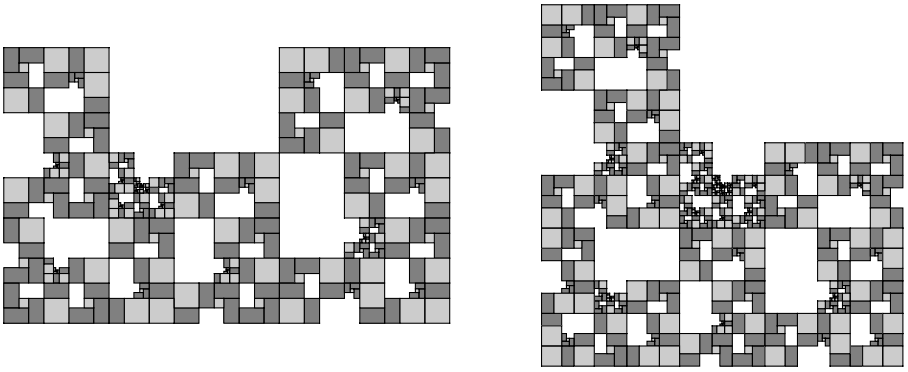
```
φ = GoldenRatio;
identity = IdentityMatrix[2];
rotation = RotationMatrix[Pi / 2];
a11 = {rotation / φ, {φ, 0}};
a12 = {identity, {0, 0}};
a21 = {identity / φ^2, {0, 0}};
b21 = {rotation / φ^2, {1 / φ + 1 / φ^2, 0}};
c21 = {rotation / φ^2, {1 / φ^2, 1 / φ^2}};
a22 = {identity / φ^3, {1 / φ^2, 1 / φ^2}};
goldenRectangleDigraph = ( {a11}        {a12}
                           {a21, b21, c21} {a22} );
```

We would also like to use the following initiators.

```
squareVertices = {{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}};
rectangleVertices = {{0, 0}, {φ, 0}, {φ, 1}, {0, 1}, {0, 0}};
square = {{GrayLevel[.8], Polygon[Drop[squareVertices]]},
   Line[squareVertices]};
rectangle = {{GrayLevel[.5], Polygon[Drop[rectangleVertices]]},
   Line[rectangleVertices]};
goldenInits = {rectangle, square};
```

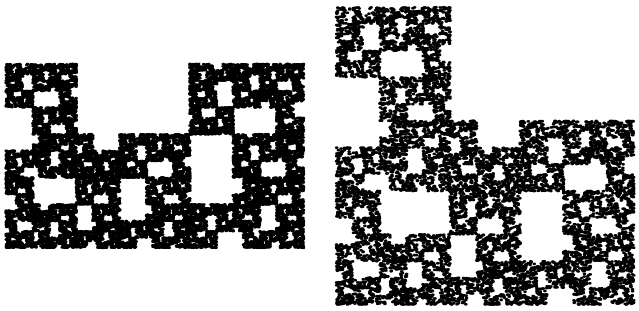
We can then use the `ShowDigraphFractals` command.

```
GraphicsRow[ShowDigraphFractals[
  goldenRectangleDigraph, 6,
  Initiators → goldenInits]]
```



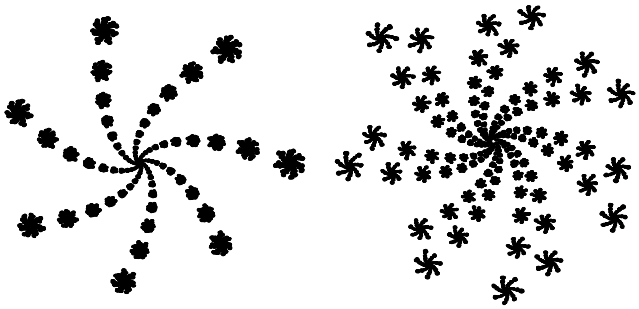
Note that just as with our basic IFS algorithm, some parts of the fractal approximations shown in this image are more finely detailed that other parts. This is exactly the situation where we might consider using the `ShowDigraphFractalsStochastic` command as demonstrated next. The second argument refers to the total number of points generated and there is no `Initiators` option.

```
GraphicsRow[ShowDigraphFractalsStochastic[
  goldenRectangleDigraph, 30 000]]
```



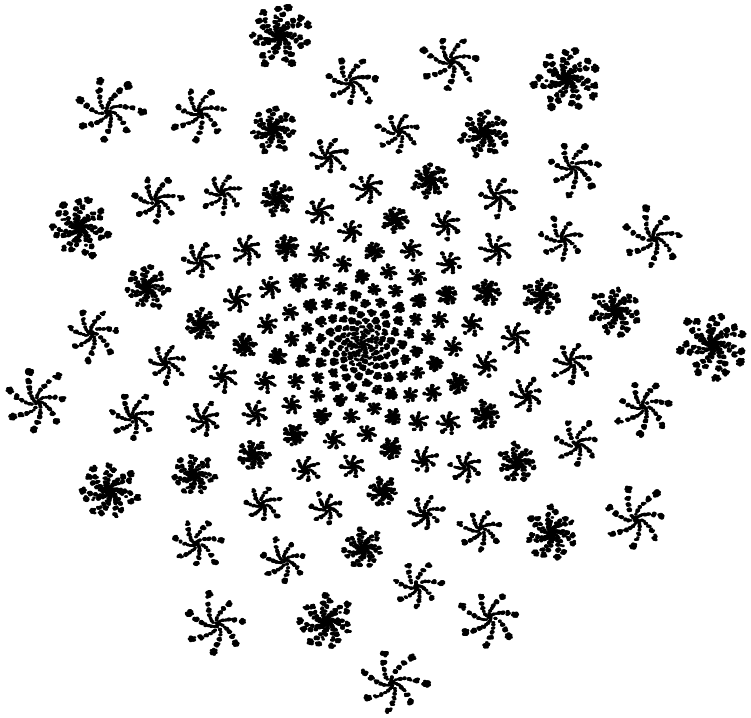
Our final example is a pair of spirals based on phyllotaxis which intertwine together beautifully.

```
goldenAngle = 2 Pi (2 - GoldenRatio);
s1 = {.985^3 RotationMatrix[3 goldenAngle], {0, 0}};
s2 = {{{.1, 0}, {0, .1}}, {1, 0}};
d1 = {.985 {{.1, 0}, {0, .1}}, .985 RotationMatrix[goldenAngle].{1, 0}};
d2 = {.985^2 {{.1, 0}, {0, .1}}, .985^2 RotationMatrix[2 goldenAngle].{1, 0}};
d3 = s1;
intertwinedSpirals = {{{s1}, {s2}}, {{d1, d2}, {d3}}};
both = ShowDigraphFractalsStochastic[intertwinedSpirals, 30 000];
GraphicsRow[both]
```



These spirals look very nice when displayed together.

```
Show[both]
```



Note that probabilities to generate a uniform distribution are automatically estimated. We may assign other probabilities using the option `PMatrix`. A basic understanding of the algorithm is necessary to use this option. We begin by assigning a positive probability p_e to every $e \epsilon E$ such that for every $v \epsilon V$,

$$\sum_{e \epsilon E_{\mathrm{uv}}, u \epsilon V} p_e = 1.$$

This simply means that the sum of all of the probabilities of all of the edges coming in to any particular vertex should be 1. For our matrix representation of the digraph, this means that the sum of all of the probabilities in any column should be 1. We then choose an arbitrary point $x \epsilon \mathbb{R}^2$, an arbitrary vertex $v \epsilon V$, and some $e \epsilon E_{uv}$ according to our probability list $\{p_e : e \in E_{uv}, v \in V\}$ and apply f_e to $\boldsymbol{x}$. If $e \in E_{uv}$, this gives us a point $f_e(\boldsymbol{x})$ in our approximation to K_u, which then becomes our new input and the process continues. We are essentially performing a random walk along the digraph and picking up points as we go along. The edges are traversed in reverse order since $f_e : K_v \to K_u$ when $e \in E_{uv}$. Thus the option `PMatrix` will be a matrix of lists of numbers, with the same shape as the digraph, such

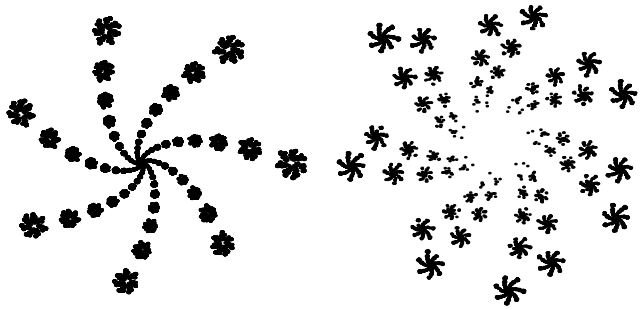
that the sum of all of the probabilities in any column is 1. We may compute the `PMatrix` used by the package using the `ComputePMatrix` function. Here is the `PMatrix` for the previous example.

```
ComputePMatrix[intertwinedSpirals]
```

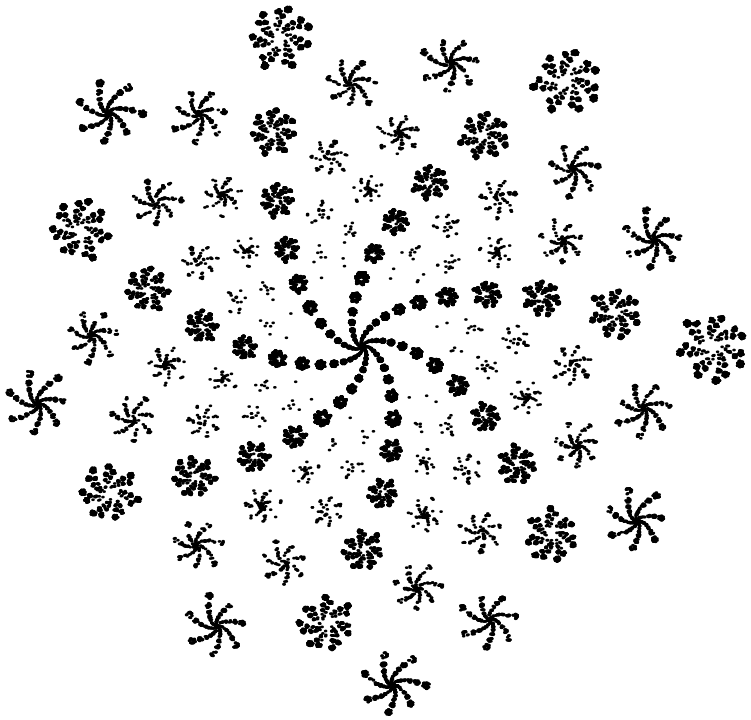
```
{{{0.939968}, {0.060032}}, {{0.0303257, 0.0297063}, {0.939968}}}
```

Let's see what happens if we fiddle with this a little.

```
both = ShowDigraphFractalsStochastic[intertwinedSpirals, 30 000,
   PMatrix → {{{0.95}, {0.15}}, {{0.025, 0.025}, {0.85}}}];
GraphicsRow[both]
```



```
Show[both]
```



The distribution has changed considerably.

4.2.3 Computing dimension

Our objective in this section is to develop an intuitive understanding of the dimension computation for digraph fractals. As with self-similar sets, the digraph IFS generates efficient covers of the digraph fractals and the rate of growth of the number of sets in these covers indicates the dimension of the set.

Consider, for example, our digraph self-similar curves and the simple coverings by triangles as shown below. Note that each of these triangles has diameter 1. Now, if we apply the digraph IFS to the pair of triangles, we obtain the finer covering shown in figure 4.18. The sets in these covers all have diameter 1/2. Iterating again, we obtain covers by sets of diameter 1/4, as shown in figure 4.19.

At the n^{th} iteration, we obtain covers of K_1 and K_2 by sets of diameter 2^{-n}; we can count the number of sets in these to estimate $N_{2^{-n}}(K_1)$ and $N_{2^{-n}}(K_2)$. The key to doing so efficiently is to realize that each set in either cover at the n^{th} level corresponds to a path of length n through the digraph for the sets. Furthermore, the number of such paths may be computed by

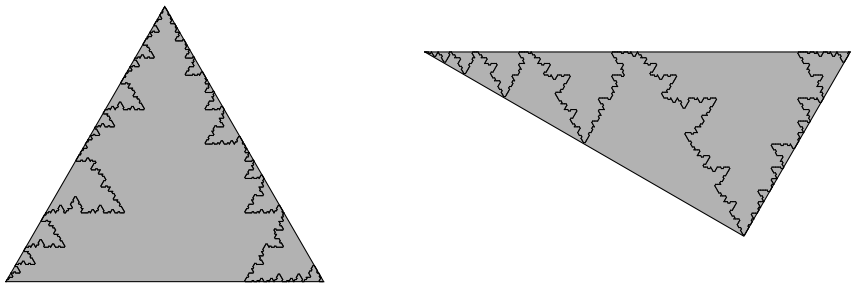


Figure 4.17 A level zero cover of the digraph curves

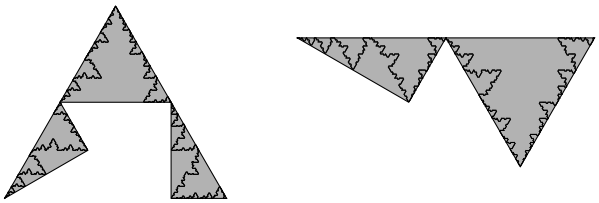


Figure 4.18 A level one cover of the digraph curves

summing the terms in the n^{th} power of the adjacency matrix of the digraph. To see this, let A denote the adjacency matrix for the digraph. Thus, for our example,

$$A = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}.$$

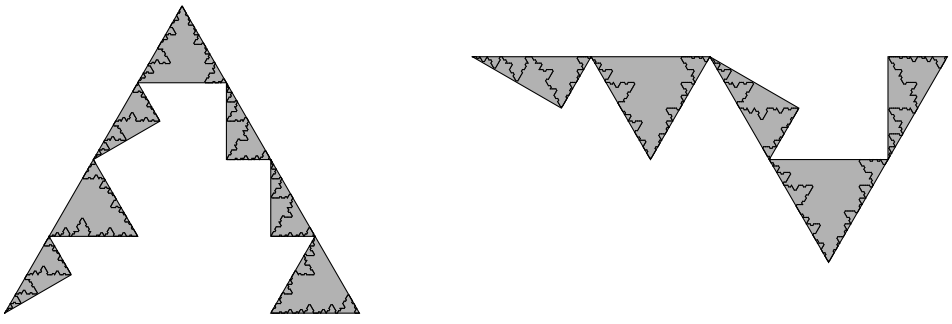


Figure 4.19 A level two cover of the digraph curves.

Recall that the entry in row i and column j denotes the number of edges (or paths of length one) from the i^{th} vertex to the j^{th}. Now consider

$$A^2 = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1+2 & 2+2 \\ 1+1 & 2+1 \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ 2 & 3 \end{pmatrix}.$$

The entry in row i and column j of A^2 is the dot product of the i^{th} row of A with the j^{th} column and represents the number of paths of length 2 from the i^{th} vertex to the j^{th}; the k^{th} term in that sum represents the number of paths through vertex k. It follows from induction that the entry in row i and column j of A^n represents the number of paths of length n from vertex i to vertex j.

Now consider the vector $v_n = A^n 1$, where 1 represents the vector of ones. In our example,

$$A^n 1 = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The i^{th} element of this vector is the sum of the i^{th} row of A^n and represents the number of sets in the n^{th} level cover of the i^{th} digraph fractal. It turns out that the entries of v_n grow exponentially at the rate λ^n, where λ is the largest eigenvalue of A. We can demonstrate this for our example matrix as follows.

```
A = {{1, 2}, {1, 1}};
sequence = NestList[A.# &, {1, 1}, 9]
{{1, 1}, {3, 2}, {7, 5}, {17, 12}, {41, 29},
 {99, 70}, {239, 169}, {577, 408}, {1393, 985}, {3363, 2378}}
```

The exponential growth rate can be checked by examining successive ratios from this list.

```
#[[2]] / #[[1]] & /@ N[Partition[sequence, 2, 1]]
{{3., 2.}, {2.33333, 2.5}, {2.42857, 2.4}, {2.41176, 2.41667}, {2.41463, 2.41379},
 {2.41414, 2.41429}, {2.41423, 2.4142}, {2.41421, 2.41422}, {2.41421, 2.41421}}
```

This common ratio is precisely the largest eigenvalue of the matrix.

```
Eigenvalues[N[A]]
{2.41421, -0.414214}
```

These computations suggest that the common dimension of the digraph fractal curves is $\log(\lambda)/\log(2)$, where $\lambda = 1 + \sqrt{2}$ is the largest eigenvalue of A.

4.2.4 Dimensional theorems

We now state some definitions and theorems that carefully confirm the observations of the last section. First, recall that a digraph is strongly connected if, given vertices u and v, there is a path from u to v. For a collection of digraph fractals, this implies that each fractal contains a similar copy of every other fractal. Therefore, there is a common dimension for all the digraph fractals.

When a directed graph is strongly connected, the corresponding adjacency matrix must have a property called irreducibility; indeed, this is taken as the definition of irreducible matrix in many texts on graph theory. In our context, this implies that we can apply a powerful theorem from linear algebra, called the Perron-Frobenius theorem. We state here a part of this theorem, without proof.

Theorem 4.2 *Let A be a non-negative, irreducible square matrix and let $\rho = \max(|\lambda|)$, where the maximum is taken over all eigenvalues of A. Let $\lambda_1 = \rho$. Then λ_1 is a simple (non-repeated) eigenvalue of A and the corresponding eigenvector may be taken to have all positive terms.*

The value λ_1 is called the spectral radius of the matrix. In the case where all similarities in the digraph IFS have the same contraction ratio, we can write down a simple upper bound for the dimension of the digraph fractals in terms of the spectral radius.

Just as in the IFS case, an assumption must be made to limit overlap between the pieces in the decomposition of the digraph fractals. Thus, we say that a digraph IFS satisfies the open set condition if there is a collection of open sets $\{U_v : v \in V\}$ such that for every $u \in V$,

$$U_u \supset \bigcup_{\mathrm{v}\epsilon\mathrm{V},\mathrm{e}\epsilon\mathrm{E_{uv}}} f_e\left(U_v\right),$$

with this union disjoint. In our digraph curves example, the open sets may be taken to be the interiors of the triangles used to generate the covers. Using the open set condition, we may now state a major theorem on the dimension of digraph self-similar sets.

Theorem 4.3 *Let $G = (V, E)$ be a strongly connected, directed multigraph. For each $e \in E$, associate a similarity f_e with common contraction ratio $r < 1$ so that the resulting digraph IFS satisfies the open set condition. Then the corresponding digraph fractals all have dimension $\log(\lambda_1)/\log(1/r)$, where λ_1 is the spectral radius of the adjacency matrix of G.*

Note that this theorem immediately verifies our dimension computation for the digraph fractals. On the other hand, it says nothing about the dimension of the golden rectangle fractals, or any example where the contraction ratios are not all the same.

Theorem 4.4 *Let $G = (V, E)$ be a strongly connected, directed multigraph. For each $e \in E$, associate a similarity f_e with contraction ratio r_e so that the resulting digraph IFS is contractive and satisfies the open set condition. For $s > 0$, define a matrix $A(s)$, with rows and columns indexed by V, such that the entry in row u and column v is*

$$A_{uv}(s) = \sum_{e \epsilon E_{uv}} r_e^s.$$

Then the box counting dimension of the digraph IFS is the unique value of s such that the matrix $A(s)$ has spectral radius 1.

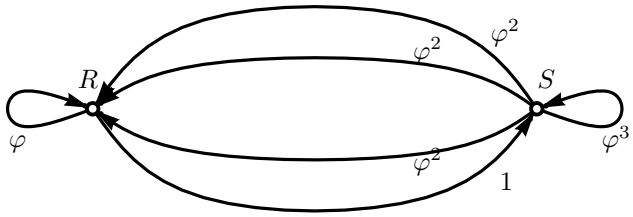


Figure 4.20 The digraph for the golden rectangle fractals

Example 4.5 We compute the fractal dimension of the golden rectangle fractals. The digraph for these fractals is shown in figure 4.20; the edge labels indicate the corresponding contraction ratios.

$$A(s) = \begin{pmatrix} 1/\varphi^s & 1 \\ 3/\varphi^{2s} & 1/\varphi^{3s} \end{pmatrix}.$$

Let us define the auxilliary function $\Phi(s)$ to be the largest eigenvalue of $A(s)$. We must find the value of s so that $\Phi(s) = 1$. Generally, this equation cannot be solved in closed form. A good numerical approximation can be found with the `FindRoot` command, however.

```
φ = GoldenRatio;
Φ[s_?NumericQ] := Max[Eigenvalues[{{1 / φ^s, 1}, {3 / φ^(2 s), 1 / φ^(3 s)}}]];
FindRoot[Φ[s] == 1, {s, 1}]
```

```
{s → 1.79246}
```

This procedure is encapsulated in the `FindDigraphDimension` command, which works directly on the digraph IFS.

```
FindDigraphDimension[goldenRectangleDigraph]
1.79246
```

Exercises

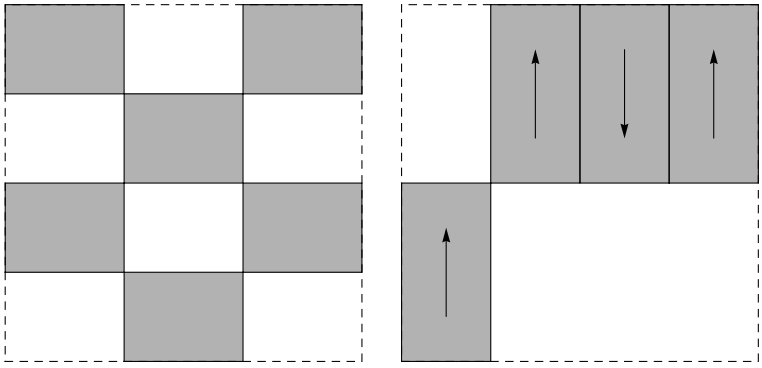


Figure 4.21 Figures for exercise 1.

4.1 Generate figures 4.2 and 4.3 from the text using the `IteratedFunctionSystems` package.

4.2 Figure 4.21 illustrates the effect of two IFSs of affine transformations on the unit square. Find each IFS and generate a high level approximation to the invariant set.

4.3 The IFS for figure 4.6 was generated with the following *Mathematica* code.

```
r = .05;
u = {1, 0};
v = {Cos[Pi / 3], Sin[Pi / 3]};
hexaLattice = Partition[Flatten[Table[2 r (m u + n v),
      {m, -12, 12}, {n, -12, 12}]], 2];
hexaLattice = Select[hexaLattice, #.# < 1 &];
M = {{r, 0}, {0, r}};
diskIFS = Prepend[#, M] & /@ hexaLattice;
```

1 How exactly does the code work? What is the effect of the parameter r?

2 Use the collage theorem 4.1 to find an upper bound for the distance between the invariant set of the IFS and the unit disk.

3 Use a similar technique to find an IFS that yields an invariant set that is close to the unit circle. *Note*: this problem deals with the circle, *not* the solid unit disk as in the example.

4.4 Use the box-counting technique developed in sub-section **??** to compute the fractal dimension of the sets in figure 4.21.

4.5 Suppose that f is a similarity transformation with contraction ratio α. Show that in this case, the singular value function defined in equation 4.1 simplifies to $\phi^s(f) = \alpha^s$. Conclude that Falconer dimension yields the similarity dimension in the strictly self-similar case.

4.6 Find a Falconer dimension.

5

Fractals and tiling

In tiling theory, we study ways to cover the plane without gaps or overlap. In classical tiling theory, we typically use a small number of relatively simple sets in the cover; any intricacies in the tiling arise from the overall pattern, rather than from the individual sets. Figure 5.1, for example, shows three simple tilings - one using squares, one using triangles, and one using both triangles and squares.

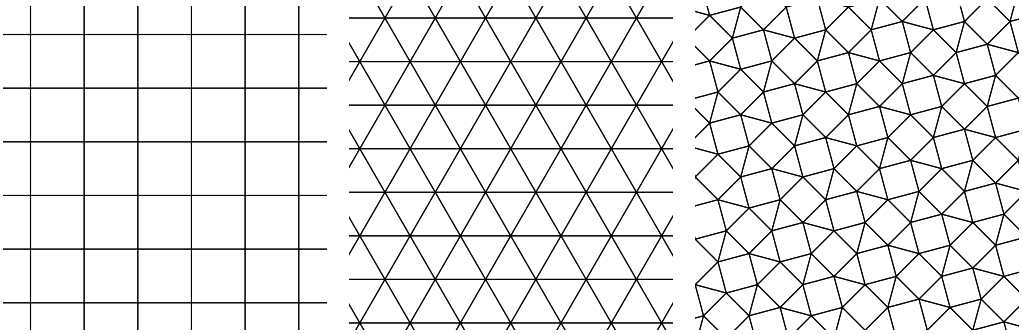


Figure 5.1 Three simple tilings

If we are to cover the plane with countably many sets, then these individual sets will need to be two dimensional. However, the *boundaries* of these sets could be fractal. Figure 5.2, for example, illustrates a tiling with a single set called the terdragon that appears in three different orientations. The boundary of the set is a self-similar fractal curve.

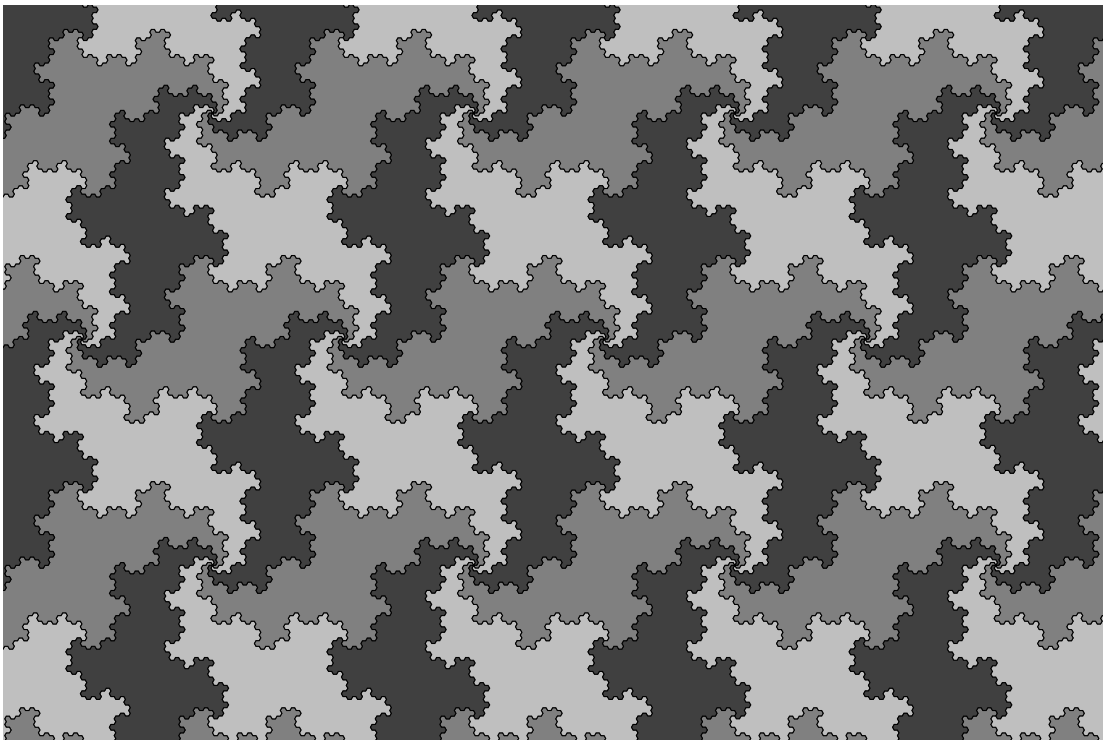


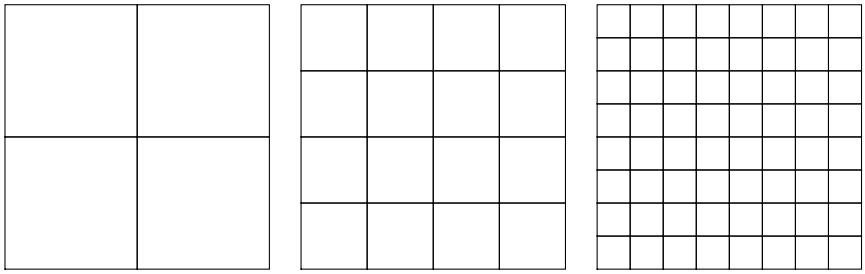
Figure 5.2 A fractal tiling

5.1 Tiling and self-similarity

5.1.1 Basics

Perhaps the simplest example of a tiling is the basic square tiling shown in figure 5.1. This tiling can be constructed using the self-similar structure of the square. In fact, we can do this with the `IteratedFunctionSystems` package as follows.

```
Needs["FractalGeometry`IteratedFunctionSystems`"];
A = {{1, 0}, {0, 1}} / 2;
IFS = {{A, {0, 0}}, {A, {1 / 2, 0}},
    {A, {0, 1 / 2}}, {A, {1 / 2, 1 / 2}}};
GraphicsRow[Table[ShowIFS[IFS, k, Initiator →
     Line[{{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}}]],
  {k, 1, 3}]]
```



To generate a tiling, of course, we would scale the images up so that the smallest squares in each image have side length 1. Nonetheless, it should be clear that any two dimensional set satisfying an open set condition yields a tiling in a natural way. This suggests the possibility of introducing fractacality into the picture. Figure 5.2 is based the self similar set shown in figure 3, for example. Figure 3 (a) illustrates the decomposition of the terdragon into 3 copies of itself. Figure 5.3 (b) shows how a higher level decomposition can induce a tiling of the plane.

We should point out that self-similarity can be used to generate all kinds of interesting tilings, even from the perspective of tiling theory. Figure 5.4, for example, illustrates a nice tiling called the chair tiling. Of course, we are primarily interested in the fractal possibilities.

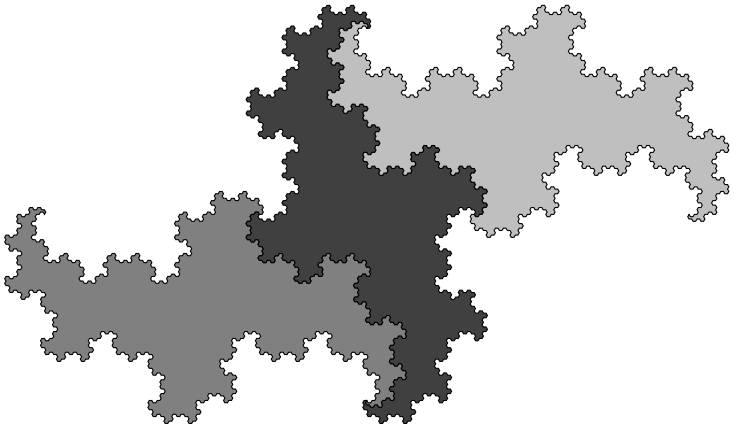
5.1.2 Self-affine tiles

While it is easy to see to how squares, triangles, and a few other two dimensional sets decompose to produce tilings, it's harder to find fractal tilings whose boundaries need to fit together just right. The earliest such tilings were found by ad hoc methods - starting with a simple tiling and iteratively modifying the boundary to obtain a fractal tiling. Figure 5.5, for example, illustrates a tiling using two, different sized Sierpinski snowflakes. We start with a simple tiling by triangles and hexagons and modify the tiling as shown. In the limit, both regions converge to Sierpinski snowflakes.

As pretty as these ad hoc methods can be, it would be nice to have a systematic way to generate fractal tilings. Furthermore, if we are to prove general theorems concerning the properties that a broad class of tilings have, then we need a precise definition of the class of sets we are studying.

A set T is called a *self-affine tile* if there is an expanding matrix A and a collection of vectors $\mathcal{D}$ (called the *digit set*) such that

(a)



(b)

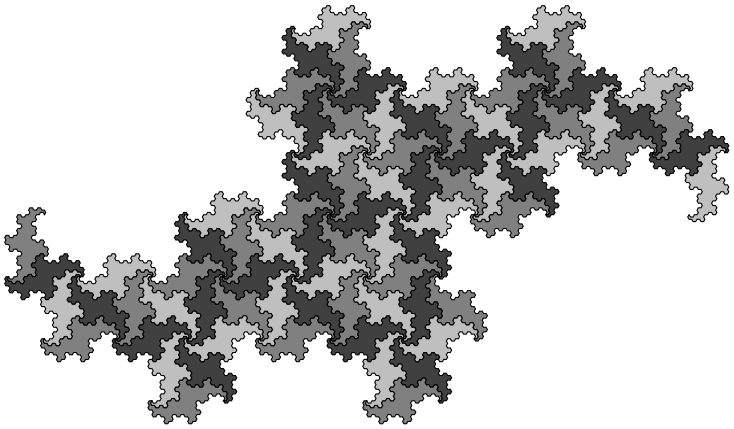


Figure 5.3 The terdragon

$$AT = T + \mathcal{D} \equiv \bigcup_{d \in \mathcal{D}} (T + d), \tag{5.1}$$

where the pieces in the union are assumed to intersect only in their boundaries. Note that AT is the image of T under multiplication by the matrix A. Also, if T is a self-affine tile with respect to A and $\mathcal{D}$, then AT is a self-affine tile with respect to A and $A\mathcal{D}$. Thus, iteration of equation 5.1 yields arbitrarily large tiling images. The unit square is an example of a self-affine tile where

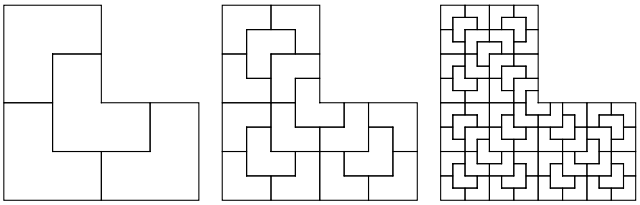


Figure 5.4 The chair tiling

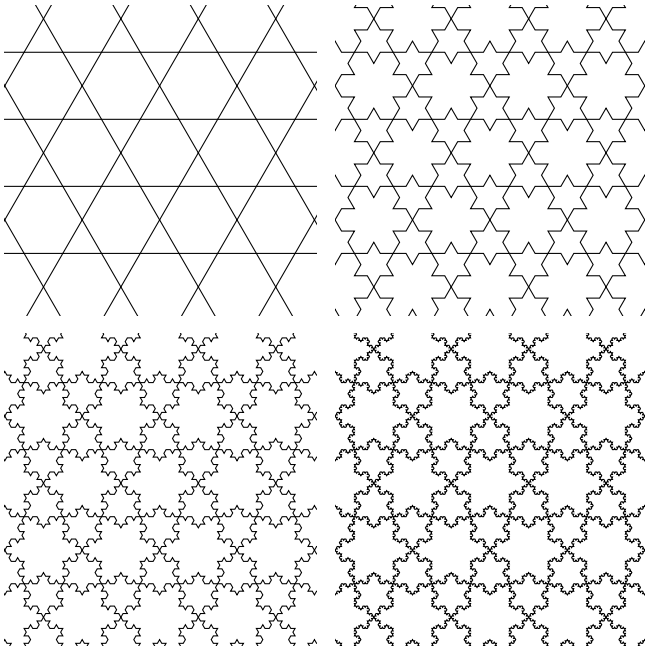


Figure 5.5 A tiling by Sierpinski snowflakes.

$$A = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \text{and} \mathcal{D} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}. \tag{5.2}$$

Iteration of equation 5.2 yields the checkerboard pattern.

As it turns out, the expansion identity 5.1 is inverse to the notion of an iterated function system. If a self-affine tile T satisfies equation 5.1, then after applying A^{-1} to both sides we see that

$$T = \bigcup_{d \in \mathcal{D}} \left(A^{-1}T + A^{-1}d \right).$$

Thus $\left\{A^{-1}x + d : d \in \mathcal{D}\right\}$ forms an IFS that generates the tile. The major question now is how to choose a matrix A and digit set $\mathcal{D}$ to generate interesting images. A beautiful theorem, published by Christoph Bandt [], provides an answer. This theorem is also described in [] at a more elementary level.

Theorem 5.1 *Let A be a two-dimensional expansive matrix with integer entries and let $\mathcal{D}$ form a residue system for A. Then, there is a unique self-affine tile T with matrix A and digit set $\mathcal{D}$. In fact, T is the invariant set of the IFS defined by $\left\{A^{-1}x + A^{-1}d : d \in \mathcal{D}\right\}$*

An *expansive* matrix is simply a matrix whose eigenvalues are all larger than one in absolute value. The terminology residue system and digit set originates from work by Gilbert [] describing certain self-similar sets in terms of number representation in the complex plane. By definition, a *residue system* for A is a complete set of coset representatives for the quotient group $\mathbb{Z}^2 / A\mathbb{Z}^2$. This means quite simply that the integer lattice $\mathbb{Z}^2$ can be decomposed as

$$\mathbb{Z}^2 = \bigcup_{d \in \mathcal{D}} \left(A\mathbb{Z}^2 + d\right).$$

This decomposition of $\mathbb{Z}^2$ plays a key role in Bandt's proof of theorem 5.1. Of course, we need to be able to generate appropriate digits sets for expansive, integer matrices. Given the matrix A with column vectors v_1 and v_2, the simplest residue system for A consists of those points with integer coordinates lying inside the parallelogram determined by v_1 and v_2 and including only the two sides containing the origin. For example, figure 5.6 illustrates this digit set for the matrix

$$\begin{pmatrix} 2 & 2 \\ -1 & 2 \end{pmatrix}.$$

We can construct other residue systems from this simple one as follows: two integer points are said to be equivalent if their difference is a linear combination of v_1 and v_2. Any vector from our simple residue system may be replaced by another from its equivalence class; that is, starting from our simplest residue system, we can simply shift some of its members by some linear combination of v_1 and v_2 to obtain another residue system. A digit set which forms a residue system for A is called a *standard digit set*. Note that the shift of a standard digit set by an integer vector is again a standard digit set; thus, we may suppose that the zero vector is one of the digits.

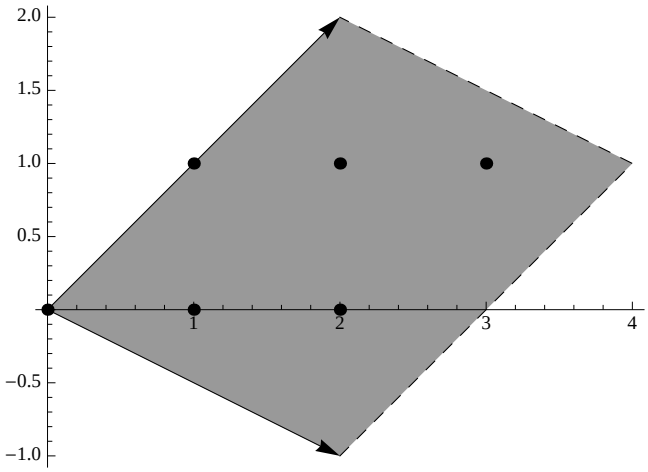


Figure 5.6 A complete residue system as a digit set

5.1.3 Examples

Example 5.2 The simplest example of a self-affine tile is the unit square, which has matrix and digit set given by equation 5.2. For our first example of a self-affine tile with fractal boundary, we modify that example by shifting the digit $(1,1)$ to the digit $(1,1)-(2,0)-(0,2)=(-1,-1)$. (We've expressed the digit this way to emphasize that it has been shifted by a linear combination of the column vectors of A.). The result is shown in figure 5.7. It is hard to imagine how such an exotic set could possibly tile the plane. The tiling is possible since the red portion of figure 5.7 complement the gray portion perfectly and this behavior is replicated when the tile is shifted, as shown in figure 5.8.

Example 5.3 The twin-dragon is the self-affine tile with matrix A and digit set $\mathcal{D}$:

$$A=\begin{pmatrix}1 & 1\\ -1 & 1\end{pmatrix};\mathcal{D}=\left\{\begin{pmatrix}0\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix}\right\}.$$

The twin-dragon is the red portion shown in figure 5.9. The gray portion is a copy of the twin-dragon shifted to the right one unit. Multiplication by the matrix A expands by the factor $\sqrt{2}$ and rotates clockwise through the angle 45°; this would map the twin-dragon onto the union of the red and gray twin-dragons.

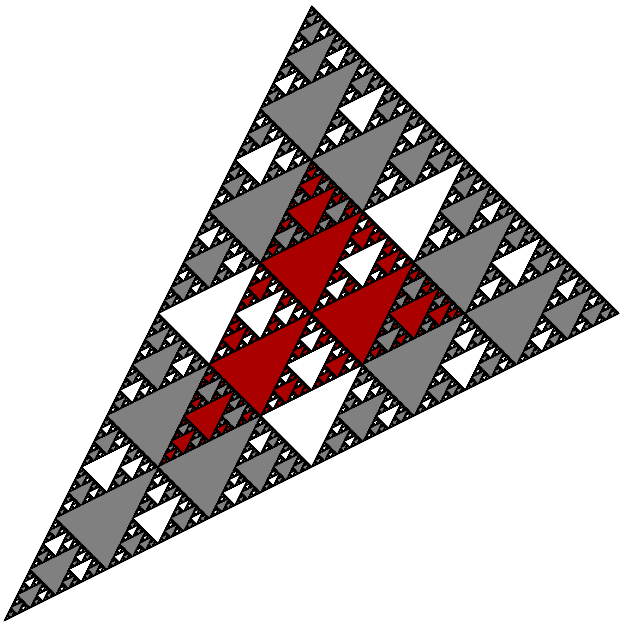


Figure 5.7 Modification of the square's digit set

Example 5.4 The type 1 terdragon is the self-affine tile with matrix A and digit set $\mathcal{D}$:

$$A = \begin{pmatrix} 2 & -1 \\ 1 & 1 \end{pmatrix}; \mathcal{D} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}.$$

The type 1 terdragon is the red portion shown in figure 5.10. The yellow and blue portions in figure 10 are shifts of the red portion under the last two vectors in the digit set $\mathcal{D}$. The union of these three portions form the union of the image of the red portion under multiplication by the matrix A.

The type 1 terdragon in example 5.4 is self-affine but not self-similar. Sometimes, such a set is affinely equivalent to a self-similar set. In this case, the self-similar set will correspond to the same matrix and digit set expressed in another basis. As explained in [], if A has a pair of complex conjugate eigenvalues, then A is similar (i.e., conjugate) to a similarity matrix. In this case, we may find the change of basis matrix B as follows. Suppose that the vector

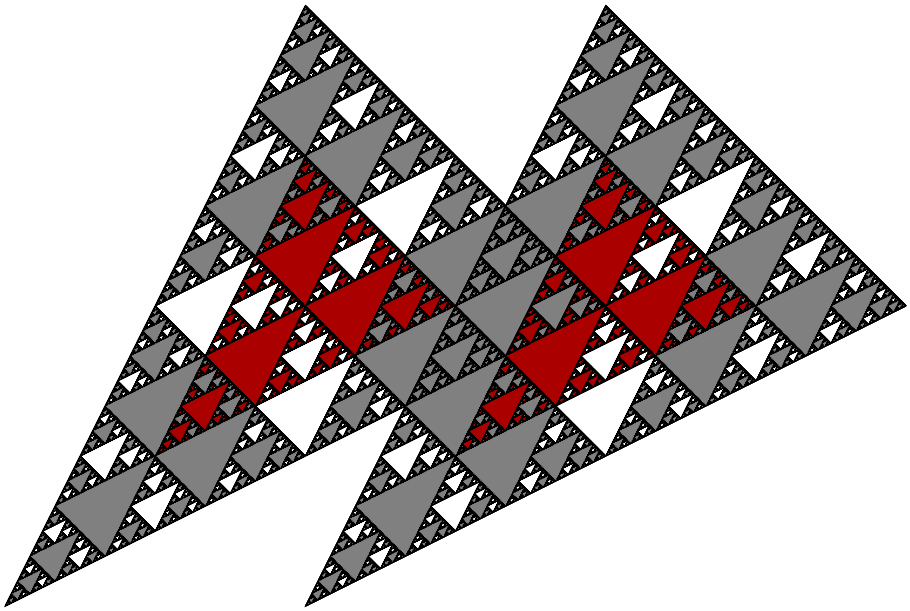


Figure 5.8 Shifting the modified square

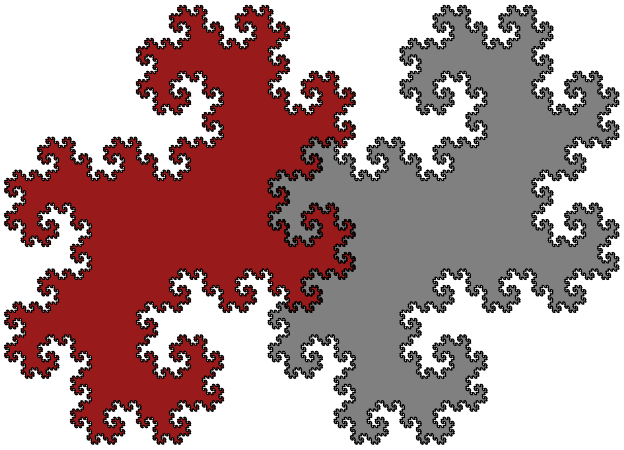


Figure 5.9 The twin-dragon

$$\begin{pmatrix} v_{11} + iv_{12} \\ v_{21} + iv_{22} \end{pmatrix}$$

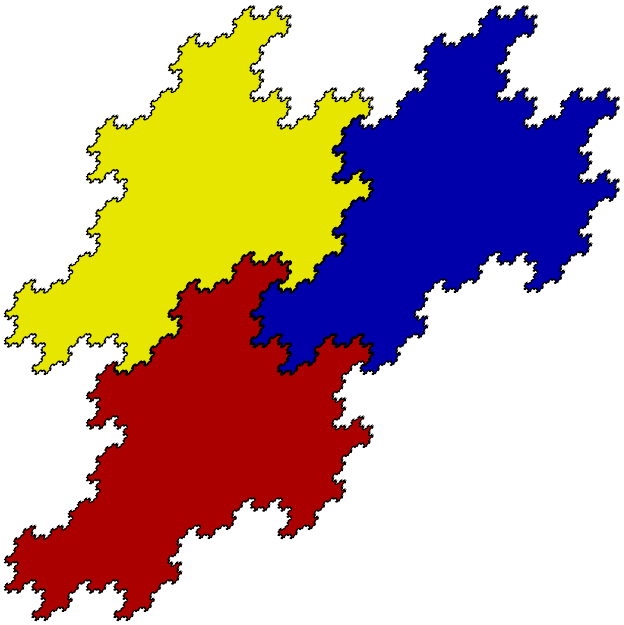


Figure 5.10 Type 1 terdragon

is an eigenvector for A, and let B be the inverse of the matrix

$$\begin{pmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{pmatrix}.$$

Then, BAB^{-1} will be a similarity transformation. The type 1 terdragon falls into this case, as the following computation shows.

```
A = {{2, -1}, {1, 1}};
Eigenvalues[A]
```

$$\left\{\frac{1}{2}\left(3+i\sqrt{3}\right), \frac{1}{2}\left(3-i\sqrt{3}\right)\right\}$$

We can now find one of the corresponding eigenvectors.

```
eigenvec = Eigenvectors[A][[1]] // Simplify
```

$$\left\{\frac{1}{2}\left(1+i\sqrt{3}\right), 1\right\}$$

We can now use this to find the change of basis matrix B.

```
B = Inverse[{{Re[eigenvec[[1]]], Im[eigenvec[[1]]]},
    {Re[eigenvec[[2]]], Im[eigenvec[[2]]]}}];
B // MatrixForm
```

$$\begin{pmatrix} 0 & 1 \\ \frac{2}{\sqrt{3}} & -\frac{1}{\sqrt{3}} \end{pmatrix}$$

The matrix B should conjugate A to a similarity matrix.

```
B.A.Inverse[B] // MatrixForm
```

$$\begin{pmatrix} \frac{3}{2} & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{3}{2} \end{pmatrix}$$

We can see that BAB^{-1} does indeed induce a similarity transformation. In fact, it is simply a clockwise rotation throughout the angle $\pi/6$ together with an expansion of $\sqrt{3}$.

```
√3 RotationMatrix[-π / 6] // MatrixForm
```

$$\begin{pmatrix} \frac{3}{2} & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{3}{2} \end{pmatrix}$$

Now the point is that, while this last matrix does not have integer entries, so it does not seem to fall into the scheme outlined by Bandt's theorem, it may be expressed as a matrix with integer entries with respect to the correct choice of basis. In fact, if we choose our basis to be the column vectors of B, then this similarity is expressed as the matrix A. The statement and proof of Bandt's theorem are essentially algebraic, so the choice of basis does not affect the result. In order to display the self-similar, type 1 terdragon in this basis, we can simply multiply the tile by B. The result is shown in figure 5.11.

5.1.4 Multiple orientations

A major weakness in Bandt's theorem is that all the functions in an IFS generated by the theorem have the same linear part. Thus we can never generate pieces in multiple orientations, like the terdragon in figure 5.3. Bandt has a second theorem that is harder to apply but more general. This theorem uses the notion of a symmetry group, which is simply a finite group G of integer matrices with determinant $+$ or $-$ one such that $MG = GM$.

Theorem 5.5 *Let M be an expansive, integer matrix, let $\{s_1, \ldots, s_m\}$ contained in a symmetry group of M, and suppose that the integer lattice can be decomposed as*

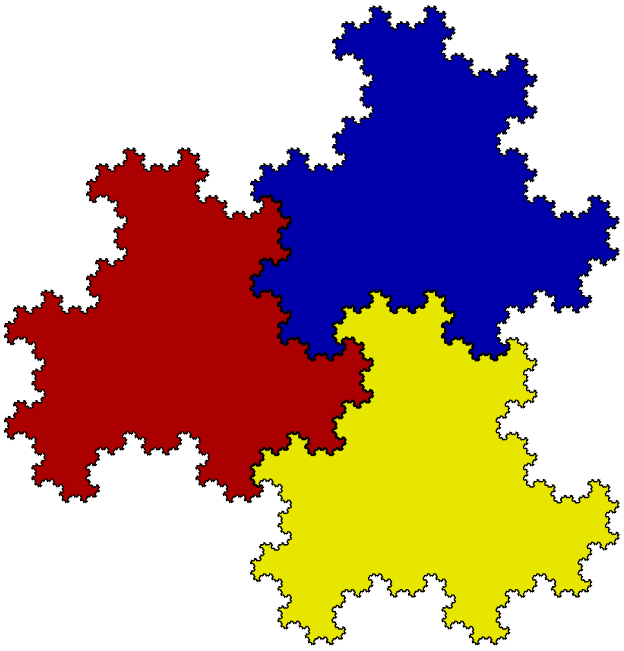


Figure 5.11 A symmetric type 1 terdragon

$$L = \bigcup_{i=1}^{m} s_i^{-1} (y_i + ML) .$$

Then, the invariant set of the IFS with $f_i(x) = s_i M^{-1} x + y_i$ *is a self-affine tile.*

This theorem is much more tedious to apply than Bandt's theorem 5.1. In that theorem, the symmetry group G is the trivial group consisting of just the identity matrix and the lattice decomposition is automatic. Both of these must be checked to apply theorem 5.5. To illustrate the process, we consider the matrix

$$M = \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix},$$

which generated the twin-dragon in example 5.3. An example of a symmetry group for this matrix is

$$G = \left\{ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \right\},$$

which is just the cyclic group of order 4 generated by

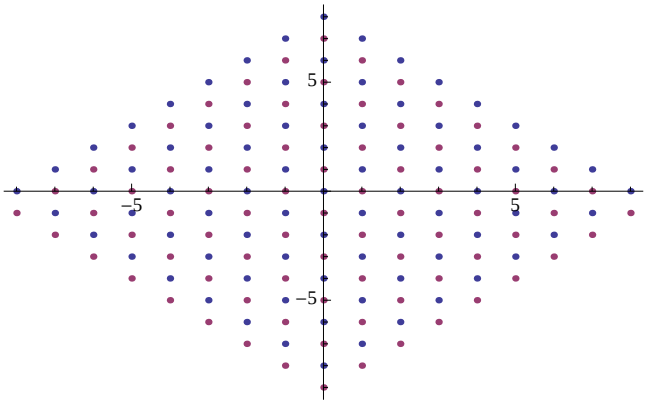
$$s_2 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

We can verify all this by simply checking that s_2 generates G and that $GM = MG$. While tedious by hand, this is easy on the computer.

```
M = {{1, 1}, {-1, 1}};
s2 = {{0, -1}, {1, 0}};
G = Table[MatrixPower[s2, k], {k, 4}];
Last[G] == {{1, 0}, {0, 1}} &&
 Union[Table[M.s, {s, G}]] ==
  Union[Table[s.M, {s, G}]]
True
```

We can check the lattice decomposition condition by generating the lattice of `points`, breaking this into two sets `points1` and `points2` that are the image of `points` under $x \to Mx$ and $x \to s_2^{-1}(Mx + (1,0))$ and checking the union of `points1` and `points2` form the whole lattice. We can visualize this as follows.

```
points = Flatten[Table[{i, j},
    {i, -4, 4}, {j, -4, 4}], 1];
points1 = M.# & /@ points;
points2 = Inverse[s2].(M.# + {1, 0}) & /@ points;
ListPlot[{points1, points2}]
```



Example 5.6 The observations above, together with theorem 5.5, imply that the invariant set of the IFS with function $f_1(x) = M^{-1}x$, $f_2(x) =$

$s_2 M^{-1} x + \langle 1, 0 \rangle$ is a self-similar tile. The result, called the Heighway dragon, is shown if figure 5.12.

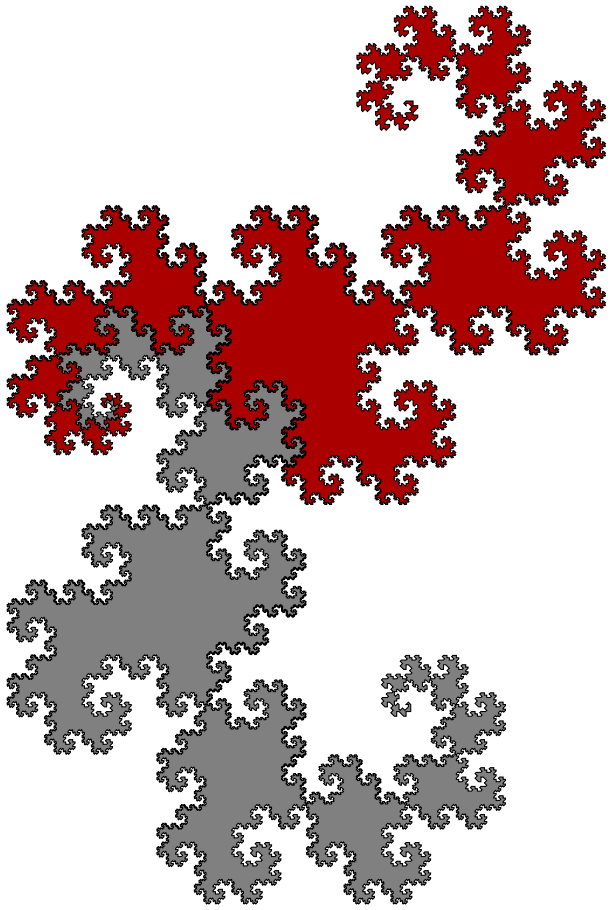


Figure 5.12 The Heighway dragon

As another example, consider the following integer matrix.

```
M = ( 2  1
     -1  1 );
```

For this example, we need the M to be an integer matrix with respect to the hexagonal basis. This means that, with respect to the standard basis, we can express M as follows.

```
S = {{1, 1 / 2}, {0, Sqrt[3] / 2}};
M = S.M.Inverse[S];
M // MatrixForm
```

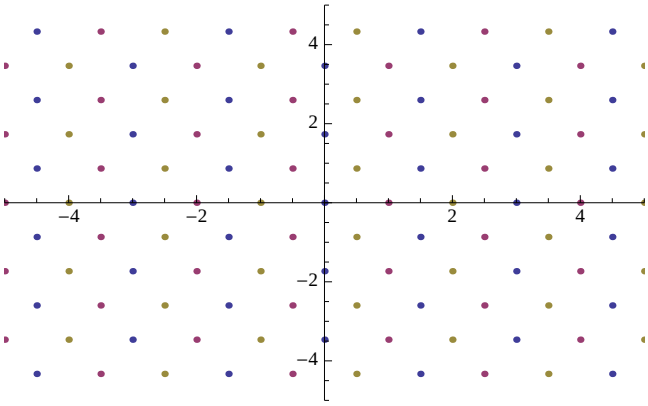
$$\begin{pmatrix} \frac{3}{2} & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{3}{2} \end{pmatrix}$$

We claim that the group of order 6 generated by the rotation through 60° is a symmetry group for M. We can check this just as we did in the previous example.

```
G = Table[MatrixPower[RotationMatrix[Pi / 3], k],
   {k, 1, 6}];
Union[Table[s.M, {s, G}]] ==
 Union[Table[M.s, {s, G}]]
True
```

Next, we claim that the lattice decomposition is satisfied for the hexagonal lattice by taking s_1 and s_2 to be the identity matrix, s_3 to be the rotation through 60°, $y_1 = (0, 0)$, and $y_2 = y_3 = (1, 0)$. We again demonstrate this by simply generating the lattice.

```
points = Flatten[Table[{i, 0} + j {1 / 2, Sqrt[3] / 2},
    {i, -5, 5}, {j, -5, 5}], 1];
points1 = M.# & /@ points;
points2 = M.# + {1, 0} & /@ points;
points3 = RotationMatrix[-Pi / 3].(M.# + {1, 0}) & /@ points;
ListPlot[{points1, points2, points3},
 PlotRange → {{-5, 5}, {-5, 5}}]
```



Example 5.7 The observations above, together with theorem 5.5, imply that the invariant set of the IFS with function $f_1(x) = M^{-1}x$, $f_2(x) = M^{-1}x + \langle 1, 0\rangle$, and $f_3(x) = s_3M^{-1}x + \langle 1, 0\rangle$ is a self-similar tile. The result is the type 2 terdragon that we first met in figure 5.3.

5.2 Fractal boundaries

While we have used iterated function systems to describe the tilings in this chapter, it is clearly the boundaries of the tiles that are somehow fractal. We now try to analyze these boundaries. The key observation is illustrated in figure 5.13, where we see a twin dragon and its six adjacent neigbors in the tiling induced by the twindragon in part (a). The collection of intersections between these neighbors and the original tile form a collection of six sets whose union is in the whole boundary, as shown in part (b). It turns out that this collection of sets forms the invariant list of a digraph IFS and this can be used analyze the boundary.

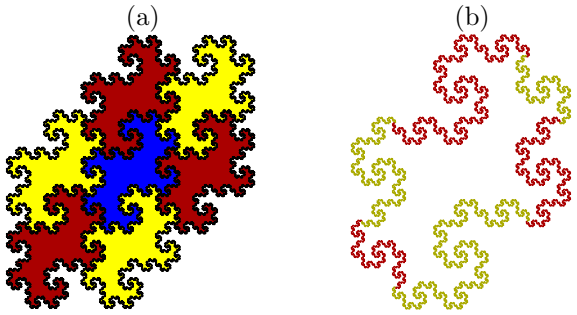


Figure 5.13 The adjacent neighbors of the twindragon and intersections that form the boundary

5.2.1 The boundary digraph IFS

To make all this more precise, suppose that T is a self-affine tile and there is a lattice Γ of points in the plane so that the translates of T by the points of Γ form a tiling of the plane. The lattice should be invariant under the action of A in the sense that $A(\Gamma) \subset \Gamma$. (Note that the lattice condition is frequently, but not always satisfied.) Given $\alpha \in \Gamma$, define $T_\alpha = T \cap (T + \alpha)$. The boundary of T is formed by the collection of sets T_α which are non-empty, excluding the case $\alpha = 0$. Let $\mathcal{F} = \{\alpha \in \Gamma : T_\alpha \neq \emptyset \text{and} \alpha \neq 0\}$. We hope to find a digraph IFS such that $\{T_\alpha : \alpha \in \mathcal{F}\}$ is the invariant list of that digraph IFS. We do so by examining how the expansion matrix A affects each set T_α and then translating to a digraph IFS by applying A^{-1}.

$$\begin{aligned} A\left(T_\alpha\right) &= A(T) \bigcap A(T+\alpha) \\ &= \left(\bigcup_{d\in\mathcal{D}} (T+d)\right) \bigcap \left(\bigcup_{d'\in\mathcal{D}} (T+d'+A\alpha)\right) \\ &= \bigcup_{d,d'\in\mathcal{D}} ((T+d) \bigcap (T+d'+A\alpha)) \\ &= \bigcup_{d,d'\in\mathcal{D}} [(T \bigcap (T-d+d'+A\alpha))+d] \\ &= \bigcup_{d,d'\in\mathcal{D}} ((T_{A\alpha-d+d'})+d)\,. \end{aligned} \tag{5.3}$$

We are only interested in the non-empty intersections so, given α and β in $\mathcal{F}$, let $M(\alpha,\beta)$ denote the set of pairs of digits (d,d') so that $\beta = A\alpha - d + d'$. Then applying A^{-1} to both sides of equation 5.3 we see that

$$T_\alpha = \bigcup_{\beta\in\mathcal{F}} \bigcup_{(d,d')\in M(\alpha,\beta)} \left(A^{-1}T_\beta + A^{-1}d\right). \tag{5.4}$$

Equation 5.4 defines a digraph IFS to generate the sets T_α. Given α and β in $\mathcal{F}$, the functions mapping T_β into T_α are precisely those affine functions defined by $\left\{A^{-1}, A^{-1}d\right\}$ for all digits d so that there is a digit d' satisfying $\beta = A\alpha - d + d'$.

We now implement the above ideas, to generate the boundary of the twin dragon. We first define A and $\mathcal{D}$.

```
A = ( 1  1
     -1  1 );
𝒟 = {{0, 0}, {1, 0}};
```

We also need to know the set $\mathcal{F}$. In general, it can be difficult to determine $\mathcal{F}$. Fortunately, [10] describes an algorithm to automate the procedure. The algorithm is fairly difficult, however, and the technique seems rather far removed from the other techniques described here. Thus we refer the interested reader to [10] and the code defining the NonEmptyShifts function in the SelfAffineTiles package. Examining figure 5.13, it is not to difficult to see that the correct set of vectors $\mathcal{F}$ for the twindragon is defined as follows.

```
ℱ = {{-1, -1}, {0, -1}, {1, 0},
   {1, 1}, {0, 1}, {-1, 0}};
```

Now for each pair (α,β) where α and β are chosen from $\mathcal{F}$, we want $M(\alpha,\beta)$ to denote the set of pairs of digits (d,d') so that $\beta = A\alpha - d + d'$. This can be accomplished as follows.

```
M[α_, β_] := Select[Tuples[𝒟, 2],
   #[[1]] - #[[2]] == β - A.α &];
digitPairsMatrix = Outer[M, ℱ, ℱ, 1];
```

In order to make sense of this, let's look at the length of each element of the matrix.

```
S = Map[Length, digitPairsMatrix, {2}];
S // MatrixForm
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

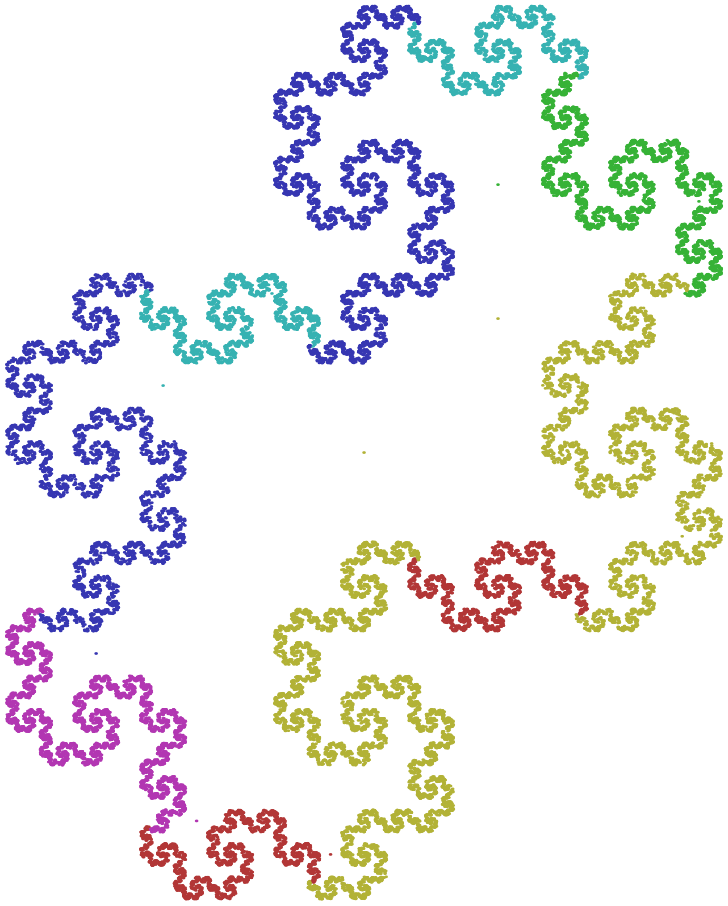
This matrix is called the *substitution matrix* of the tile and tells us simply the combinatorial information of how the pieces of the boundary fit together. Reading the rows, for example, we see that the first piece is composed of one copy of the last piece, the second piece is composed of one copy of itself and two copies of the first, etc. Note also that the order of the rows and columns is dictated by the order of the set $\mathcal{F}$. Thus the first piece refers to the boundary along the maroon image in the lower left of figure 5.13, since $(-1,-1)$ is the first shift vector in the set $\mathcal{F}$. The subsequent pieces are numbered counterclockwise around the central tile, since that is the way that $\mathcal{F}$ is set up.

We can transform the `digitPairsMatrix` into a digraph IFS defining the boundary by simply replacing each pair (d, d') with the affine function $\left(A^{-1}, A^{-1}d\right)$.

```
boundaryDigraphIFS = digitPairsMatrix /.
   {_, {x_?NumericQ, y_}} → {Inverse[A], Inverse[A].{x, y}};
```

Let's see how it worked. We'll use the function `ShowDigraphFractalsStochastic` defined in the `DigraphFractals` package and then `Show` all the parts together. We'll use color to distinguish the constituent parts.

```
Needs["FractalGeometry`DigraphFractals`"];
boundaryParts = ShowDigraphFractalsStochastic[
   boundaryDigraphIFS, 40 000, Colors → True];
Show[boundaryParts]
```



5.2.2 The dimension of the boundary

The preceeding section provides a complete description of the digraph IFS that generates the boundary of a self-affine tile. Assuming that the matrix A defines a similarity, theresulting digraph is strongly connected, and a separation condition like the digraph OSC is satisfied, we should be able to use theorem 4.3 to compute the dimension of the boundary. Unfortunately, these conditions don't necessarily hold in general. Remarkably, Strichartz and Wang have shown that the conclusion of theorem 4.3 holds for these digraph IFSs anyway. Note that the substitution matrix, as defined by Strichartz and Wang, is exactly the adjacency matrix for the digraph IFS. Thus we have the following theorem.

Theorem 5.8 *Let T be a self-affine tile for the matrix A and with substi-*

tution matrix S. Let $\rho(M)$ denote the spectral radius of a matrix M. Then, the fractal dimension of the boundary of the tile A is $\log(\rho(S))/\log(\rho(A))$.

For example, we can compute the dimension of the boundary of the twin-dragon using the matrices `S` and `A` as defined at the end of the previous sub-section. We simply compute the spectral radii as follows.

```
{ρ1, ρ2} = {Max[Abs[Eigenvalues[S]]], Max[Abs[Eigenvalues[A]]]}
```

$$\left\{\texttt{Root}\left[-2-\#1^2+\#1^3\ \&,\ 1\right],\ \sqrt{2}\right\}$$

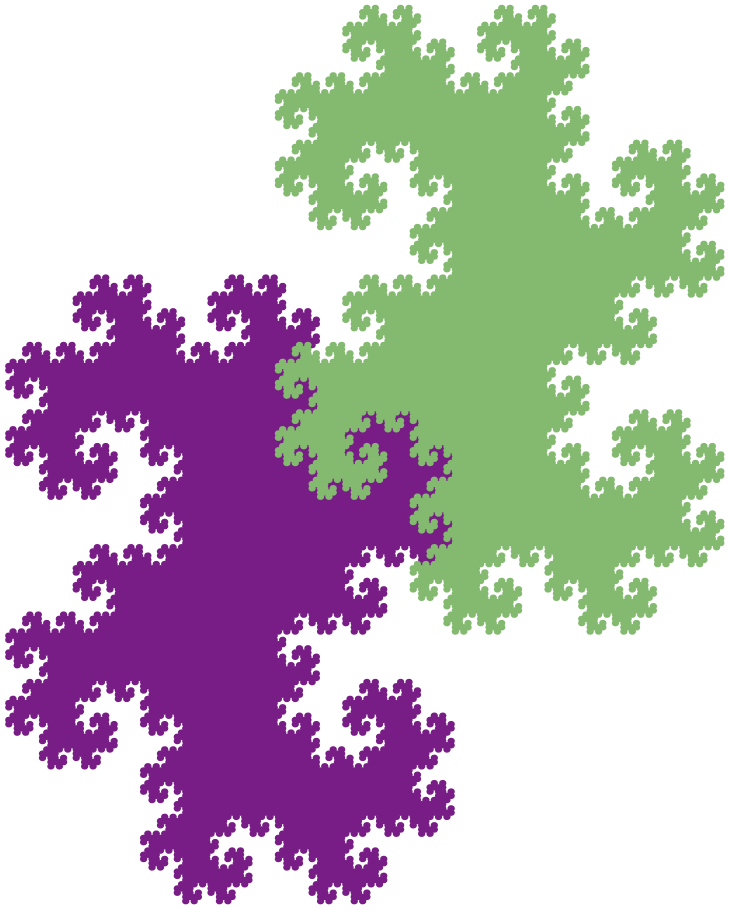
The spectral radius of `A` is clearly no surprise but the spectral radius of `S` requires a bit of interpretation. This `Root` result simply means that the spectral radius is the largest root of the cubic polynomial $x^3 - x^2 - 2$, which happens to be a factor of the characteristic polynomial of `A`. If we donote this number by ρ_1, then the dimension is $\log(\rho_1)/\log\left(\sqrt{2}\right)$. We can find a numerical estimate for this as follows.

```
Log[ρ1] / Log[ρ2] // N
1.52363
```

5.3 The SelfAffineTiles package

Given a matrix and digit set, it is amazingly easy to generate a basic image of the corresponding self-affine tile. Let's illustrate the process to generate the twin-dragon.

```
Needs["FractalGeometry`IteratedFunctionSystems`"];
A = {{1, 1}, {-1, 1}};
𝒟 = {{0, 0}, {1, 0}};
IFS = {Inverse[A], Inverse[A].#} & /@ 𝒟;
ShowIFS[IFS, 14, Colors → True]
```



5.4 Aperiodic tiling

Appendix A
A brief introduction to *Mathematica*

Mathematica is an immense computer software package for doing mathematical computation and exploration. It contains hundreds of mathematical functions, commands for producing graphics, and a complete programming language. This appendix is a brief introduction to *Mathematica*, focusing on the tools used in this book. The serious user of *Mathematica* will need more comprehensive reference and tutorial materials. While there are a not many resources available specifically for the latest version of *Mathematica* (V6, which this text uses), there are several excellent references which adequately cover the core programming tools...

A.1 The very basics

At the most basic level, *Mathematica* can be used interactively, like a calculator. Commands to be evaluated are entered into *input cells*, which are displayed `inBoldInput` in this text. They are evaluated by pressing the ENTER key. (Note that ENTER is distinct from RETURN which is used to start a new line.) Results are displayed in *output cells*. Here is a simple example:

```
2 + 2
```
```
4
```

You can enter longer expressions over several lines. You can also include comments in input cells by enclosing them in `(* These *)`

```
(5 * 7 - 32 / 6) / 4

(* Here is a comment *)
```
$$\frac{89}{12}$$

Notice that we get a fraction. *Mathematica* is capable of exact computations, rather than decimal approximations. We can always get numerical approximations using the `N` function (for numerical). In the following line, `%` refers to the previous result and `// N` passes that result to the `N` function

```
% // N
```

```
7.41667
```

Mathematica's ability to do exact computations is truly impressive.

```
2^1000
```

```
10 715 086 071 862 673 209 484 250 490 600 018 105 614 048 117 055 336 074 ↘
  437 503 883 703 510 511 249 361 224 931 983 788 156 958 581 275 946 729 ↘
  175 531 468 251 871 452 856 923 140 435 984 577 574 698 574 803 934 567 ↘
  774 824 230 985 421 074 605 062 371 141 877 954 182 153 046 474 983 581 ↘
  941 267 398 767 559 165 543 946 077 062 914 571 196 477 686 542 167 660 ↘
  429 831 652 624 386 837 205 668 069 376
```

Approximations can be computed to any desired degree of accuracy.. Here we use the `N` function again along with an optional second argument to obtain 100 decimal digits of π.

```
N[Pi, 100]
```

```
3.14159265358979323846264338327950288419716939937510582097494↘
  459230781640628620899862803482534211706 8
```

Mathematica has extensive knowledge about mathematical functions.

```
Sin[Pi / 3]
```

$$\frac{\sqrt{3}}{2}$$

Note: *built in function names are always capitalized and the arguments are enclosed in square brackets.*

Mathematica can do algebra.

```
Expand[(x + 1)^10]
```

$$1 + 10\,x + 45\,x^2 + 120\,x^3 + 210\,x^4 + 252\,x^5 + 210\,x^6 + 120\,x^7 + 45\,x^8 + 10\,x^9 + x^{10}$$

```
Factor[x^4 - x^3 - 2 x - 4]
```

$$(-2 + x)\,(1 + x)\,\left(2 + x^2\right)$$

Mathematica can do calculus.

```
D[x^2, x]
```

$$2\,x$$

```
Integrate[Sin[x], {x, 0, Pi}]
```

$$2$$

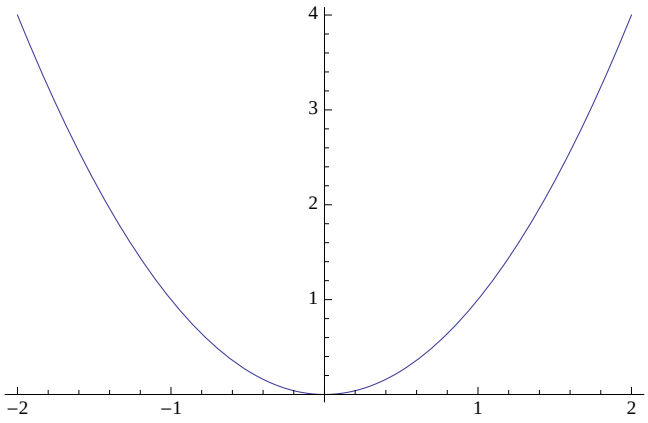
Mathematica can compute sums.

```
Sum[2 / 9^n, {n, 1, Infinity}]
```

$$\frac{1}{4}$$

Mathematica can plot functions.

```
Plot[x^2, {x, -2, 2}]
```



A.2 Brackets [], braces , and parentheses ()

Brackets, braces, and parentheses all have distinct purposes in *Mathematica*. Brackets `[]` are used to enclose the arguments of a function such as:

```
N[Tan[1], 20]
1.5574077246549022305
```

Parentheses `()` are used for grouping in mathematical expressions such as:

```
(2 * 5 - 1) / 3
3
```

Braces `{}` are used to form lists. Lists play a very important role in *Mathematica* as they are a fundamental data structure. Many functions automatically act on the individual elements of a list.

```
N[{Sqrt[2], E}, 20]
{1.4142135623730950488, 2.7182818284590452354}
```

Some commands return lists. The `Table` and `Range` commands are two of the most important such commands. The `Table` command has the syntax `Table[expr,{x,xMin,xMax}]`. This evaluates `expr` at the values `xMin`,

`xMin+1`, ..., `xMin+n`, where `n` is the largest integer so that `xMin+n` $\leq$ `xMax`. For example, here are the first 10 natural numbers squared.

```
Table[x^2, {x, 1, 10}]
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

It is sometimes convenient to use a slightly more general version of `Table` which specifies the step size: `Table[expr, {x,xMin,xMax,step}]`, where `step` represents the step size.

```
Table[x^2, {x, 1, 10, .5}]
{1., 2.25, 4., 6.25, 9., 12.25, 16., 20.25, 25.,
 30.25, 36., 42.25, 49., 56.25, 64., 72.25, 81., 90.25, 100.}
```

The more specific version `Table[expr,{n}]` simply produces `n` copies of `expr`. We will see several instances when this is useful. `Table` accepts multiple variables to create nested lists. In the following example, x is fixed for each sublist and ranges from 0 to 2 as we move from list to list; y ranges from 1 to 4 inside each sublist.

```
Table[x + y, {x, 0, 2}, {y, 1, 4}]
{{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}}
```

The `Range` command simply produces a list of numbers given a certain `min`, `max`, and `step`. This is not as limited as it might seem, since functions can act on lists.

```
Range[10]^2
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

```
Range[1, 10, .5]^2
{1., 2.25, 4., 6.25, 9., 12.25, 16., 20.25, 25.,
 30.25, 36., 42.25, 49., 56.25, 64., 72.25, 81., 90.25, 100.}
```

A.3 Entering typeset expressions

Up until this point, it has been fairly clear how to enter all of the example input cells. *Mathematica* allows input to resemble mathematical notation much more closely, however. For example, the following are equivalent.

```
N[Pi^2 / 6]
1.64493
```

$$\mathtt{N}\left[\frac{\pi^2}{6}\right]$$

```
1.64493
```

There are two basic ways of entering typeset expressions - using palettes or using keyboard shortcuts. Palettes are more intuitive, but typically take longer. The best approach is usually to know a few important keyboard shortcuts and to use the palettes when necessary. Let's try both techniques to enter $\frac{\pi^2}{6}$ into the notebook.

First, we'll use the Basic Math Input palette. This is the second palette under the Palettes menu in V6 and is also the second palette under Palettes▷Other in V7. Note that V7 has much more extensive palettes than V6 but the basic usage is the same. To enter $\frac{\pi^2}{6}$ into the notebook using the Basic Math Input palette, proceed as follows.

1 Click on the $\frac{\square}{\square}$ button in the first row
2 Click on the $\square^{\square}$ button near the bottom row.
3 Click on the π button near the middle of the palette.
4 Press the TAB key.
5 Type "2".
6 Press the TAB key.
7 Type "6".
8 Press the right arrow twice key to exit the typeset expression.

Alternatively, we can enter $\frac{\pi^2}{6}$ using just the keyboard. Here's how:

1 Press the ESC key.
2 Type "pi".
3 Press the ESC key.
4 Press the CTRL and ^ keys simultaneously.
5 Type "2".
6 Press the right arrow key.
7 Press the CTRL and / keys simultaneously.
8 Type "6".
9 Press the right arrow twice key to exit the typeset expression.

A.4 Defining constants and functions

Constants can be defined in the natural way.

```
c = 7
7
```

The symbol `c` will now be treated as 7.

```
c + 3
10
```

Symbol names can be longer alphanumeric strings and may refer to more complicated objects

```
theList = Range[10] + c
{8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

We can still act on this object.

```
theList^2
{64, 81, 100, 121, 144, 169, 196, 225, 256, 289}
```

We can clear the contents of the symbols.

```
Clear[c, theList]
```

```
c + 3
3 + c
```

```
theList^2
theList^2
```

Here is how to define the function $f(x) = x + 2$.

```
f[x_] := x + 2
```

We can now plug in any value we want for `x`.

```
f[3]
5
```

```
f[y^2]
2 + y^2
```

The important things to remember when defining a function are 1) (almost) always use a `:=` sign and 2) use underscores after variable names in the function declaration to the left of the `:=` sign. The underscore is used because this is how *Mathematica* recognizes a *pattern*. Pattern recognition is a central part of *Mathematica* and functions are defined in terms of patterns. The `:=` sign is the abbreviated form of `SetDelayed`, as opposed to `=` which is the abbreviated form of `Set`. The difference is that using `=` sets the definition immediately while using `:=` waits until the expression is actually evaluated, which is usually what you want when declaring a function. Here is an example where the difference is important. Suppose we want to define a function `diff` that automatically differentiates with respect to `x`. The correct way to do this is as follows. (Note that the semi-colon at the end of the first line suppresses the output of that command. The semi-colon is frequently used when entering multiple lines of code.)

```
Clear[f];
diff[f_] := D[f, x];
diff[x^2]
2 x
```

If we use = instead of :=, the function does not work properly.

```
Clear[diff];
diff[f_] = D[f, x];
diff[x^2]
0
```

The problem is that D[f,x] is evaluated immediately and returns 0.

```
D[f, x]
0
```

Using := tells *Mathematica* to wait until some expression is substituted for f to take the derivative.

Sometimes it is convenient to use a *pure function*. A pure function has the syntax expr &, where expr is a *Mathematica* expression involving the # symbol and the & operator tells *Mathematica* to treat expr as a function with argument #. For example, #^#& is a pure function which raises a number (or possibly another *Mathematica* object) to itself. We use square brackets [] to plug in an argument just as with any other function.

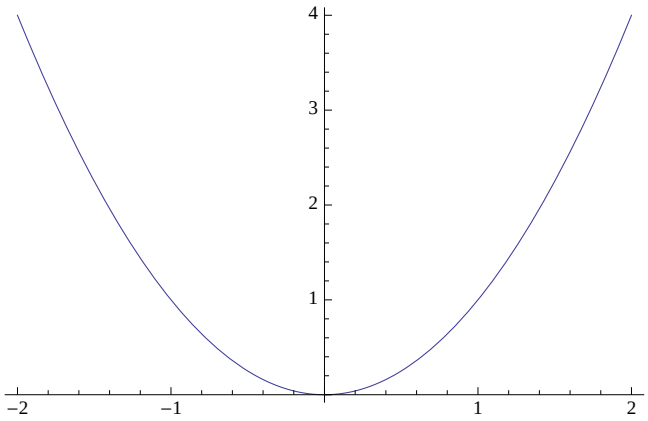
```
#^# &[3]
27
```

As we will see shortly, pure functions are particularly useful in the context of list manipulation.

A.5 Basic graphics

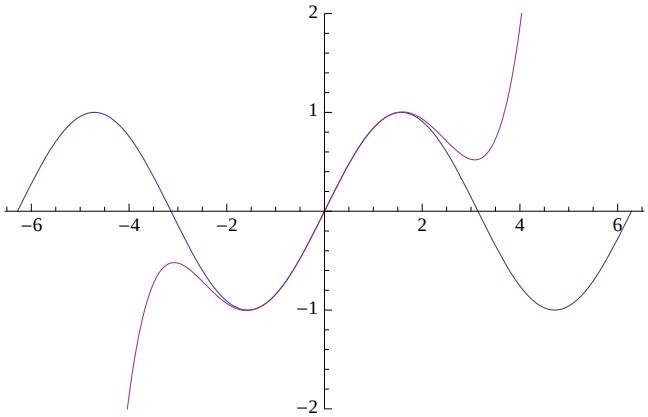
There are several basic commands for plotting functions or data. The most basic is the Plot command. For example, we can plot the function x^2 over the range -2 $\leq x \leq$ 2 (denoted {x,-2,2}).

```
Plot[x^2, {x, -2, 2}]
```



We can plot more than one function on the same graph by using a list of functions. Here's the plot of $\sin(x)$ together with a polynomial approximation.

```
Plot[{Sin[x], x - x^3/6 + x^5/120}, {x, -2 π, 2 π},
 PlotRange → {-2, 2}]
```

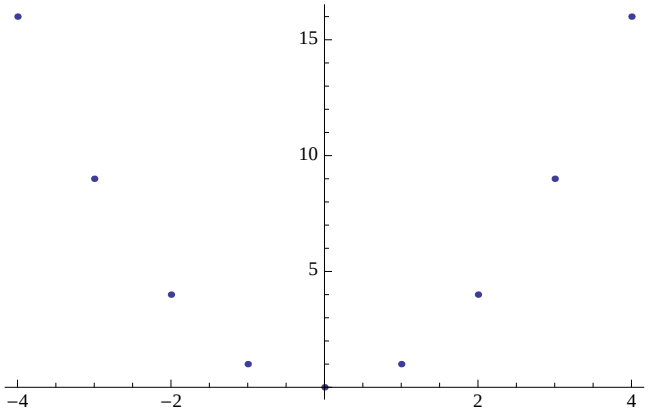


Note that we have also included the option `PlotRange->{-2,2}` which controls the vertical range in the plot. The `Plot` command has many such options which you can examine through the Documentation Center.

The `ListPlot` command plots a list of points in the plane.

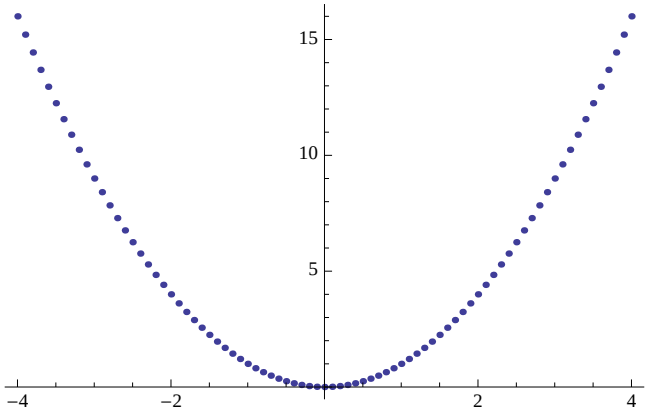
```
Table[{x, x^2}, {x, -4, 4}]
{{-4, 16}, {-3, 9}, {-2, 4}, {-1, 1}, {0, 0}, {1, 1}, {2, 4}, {3, 9}, {4, 16}}

ListPlot[%]
```



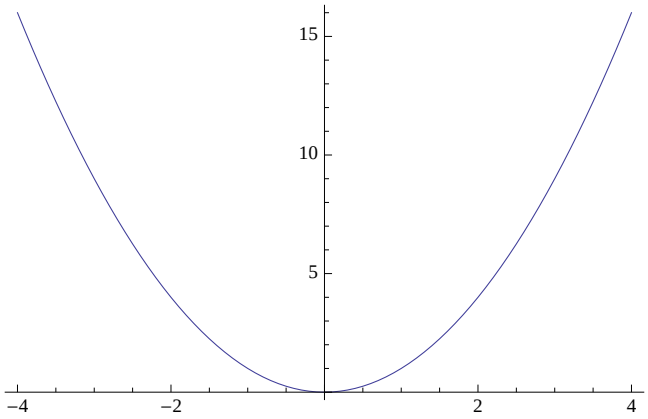
When plotting many points, it makes sense to assign the output of the `Table` command to a variable and to suppress the output.

```
data = Table[{x, x^2}, {x, -4, 4, .1}];
ListPlot[data]
```



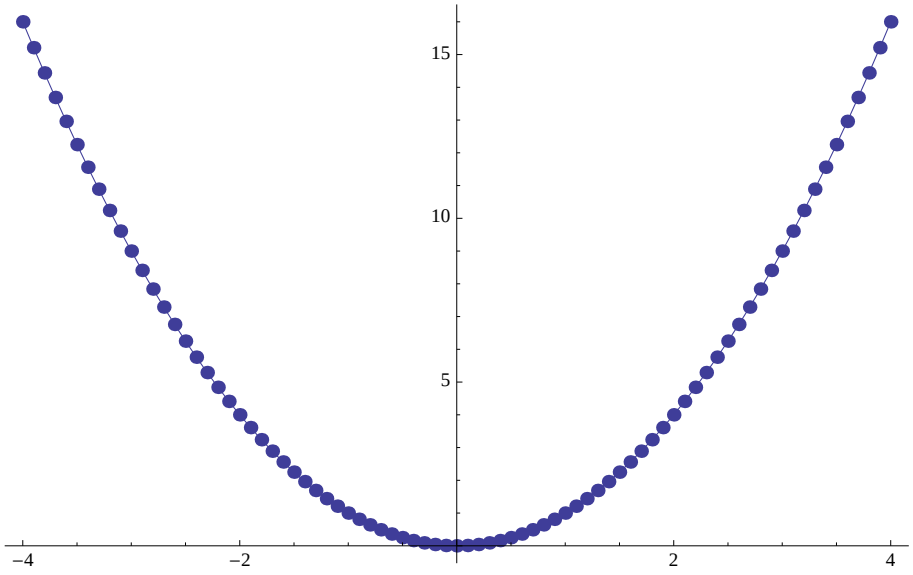
If you want the dots connected, you can use `ListLinePlot`.

```
ListLinePlot[data]
```



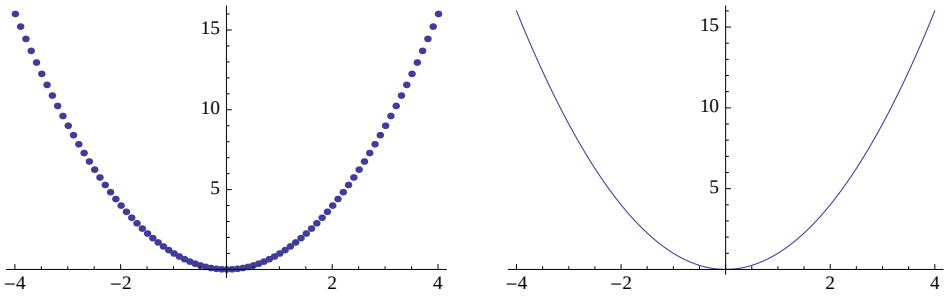
Again, a semi-colon suppresses the output. This is convenient when you don't want to see the output, but want to combine it with subsequently generated graphics. For example, to plot the parabola above together with the sample points chosen to generate it, we could generate both plots separately, store those results using variables, and use the `Show` command to combine those plots.

```
plot1 = ListPlot[data,
   PlotStyle → {PointSize[.015]}];
plot2 = ListLinePlot[data];
Show[plot1, plot2]
```



We can also use a `GraphicsGrid` to display both plots separately.

```
GraphicsGrid[{{plot1, plot2}}]
```



A.6 Solving equations

Mathematica has powerful built in techniques for solving equations algebraically and numerically. The basic command is `Solve`. Here is how to use `Solve` to find the roots of a polynomial.

```
Solve[x^3 - 2 x - 4 == 0, x]
{{x → -1 - i}, {x → -1 + i}, {x → 2}}
```

The general quintic cannot be exactly solved in terms of roots, so *Mathematica* has it's own representation of such roots.

```
Solve[x^5 - 2 x - 3 == 0, x]
{{x → Root[-3 - 2 #1 + #1^5 &, 1]}, {x → Root[-3 - 2 #1 + #1^5 &, 2]},
  {x → Root[-3 - 2 #1 + #1^5 &, 3]}, {x → Root[-3 - 2 #1 + #1^5 &, 4]}, {x → Root[-3 - 2 #1 + #1^5 &, 5]}}
```

Note that the `NSolve` command is similar, but returns numerical approximations.

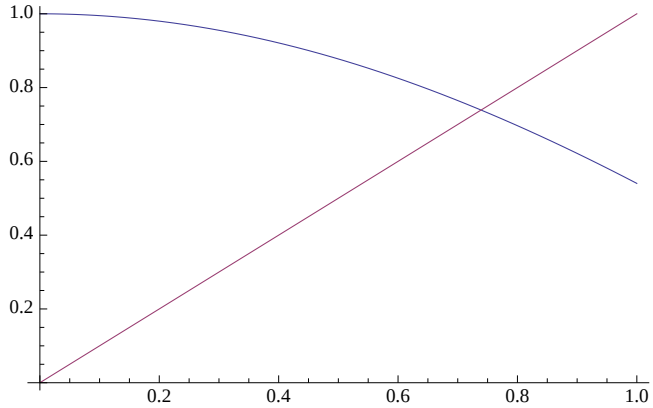
```
NSolve[x^5 - 2 x - 3 == 0, x]
{{x → -0.958532 - 0.498428 i}, {x → -0.958532 + 0.498428 i},
  {x → 0.246729 - 1.32082 i}, {x → 0.246729 + 1.32082 i}, {x → 1.42361}}
```

The `Solve` and `NSolve` commands work well with polynomials and systems of polynomials. However, many equations involving transcendental functions are beyond their capabilities. Consider, for example, the simple equation $\cos(x) = x$. `NSolve` will not solve the equation, but returns a statement to let you know this.

```
NSolve[Cos[x] == x, x]
Solve::tdep : The equations appear to involve the
   variables to be solved for in an essentially non-algebraic way. >>
NSolve[Cos[x] == x, x]
```

A simple plot shows that there is a solution in the unit interval.

```
Plot[{Cos[x], x}, {x, 0, 1}]
```



The `FindRoot` command uses Newton's method to find this solution. Since this is an iterative method, an initial guess is required. The graph indicates that 0.8 would be a reasonable initial guess.

```
FindRoot[Cos[x] == x, {x, 0.8}]
{x → 0.739085}
```

We will use `FindRoot` to solve certain equations for the fractal dimension of a set.

A.7 Random sequences

Many fractal algorithms have a random element so it is important to be able to generate random sequences. *Mathematica* has several commands for this purpose; `RandomInteger` is, perhaps, the most fundamental. `RandomInteger[{iMin,iMax},n]` generates a list of `n` random integer between `iMin` and `iMax`.

```
RandomInteger[{1, 4}, 10]
{4, 2, 4, 2, 4, 3, 4, 1, 2, 4}
```

In applications, the terms in a randomly generated sequence need not be numbers and they might not be uniformly weighted. Suppose, for example, that `f1`, `f2`, and `f3` are functions and must be chosen randomly with the weights 1/2, 1/3, and 1/6. This is easily accomplished using the `RandomChoice` command.

Of course, these are not genuinely random sequences but pseudo-random; they are deterministic procedures which appear random. The output of such a pseudo-random number generator depends upon an initial input called a *seed*. We can set the seed using the `SeedRandom` command. `SeedRandom` accepts an integer input.

```
SeedRandom[1];
RandomReal[{1, 4}, 5]
{3.45217, 1.33426, 3.36858, 1.56341, 1.72408}
```

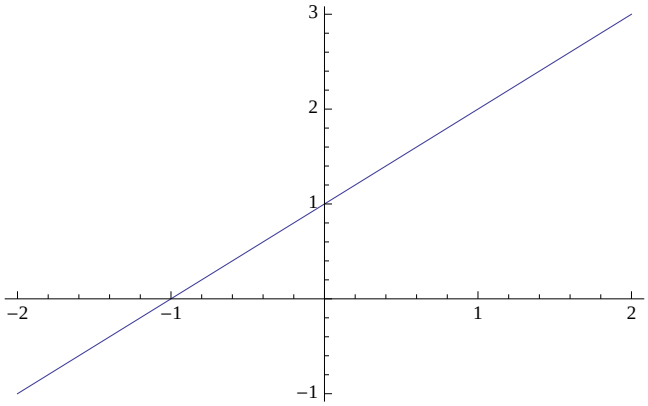
If the table is generated again after another call to `SeedRandom[1]`, `Random` will generate the same sequence of numbers.

```
SeedRandom[1];
RandomReal[{1, 4}, 5]
{3.45217, 1.33426, 3.36858, 1.56341, 1.72408}
```

A.8 Graphics primitives

Graphics functions like `Plot` generate graphical output.

```
Plot[x + 1, {x, -2, 2}, PlotPoints → 2,
 MaxRecursion → 0]
```



The graph is just one way to format the result. The `InputForm` is much closer to *Mathematica*'s internal representation

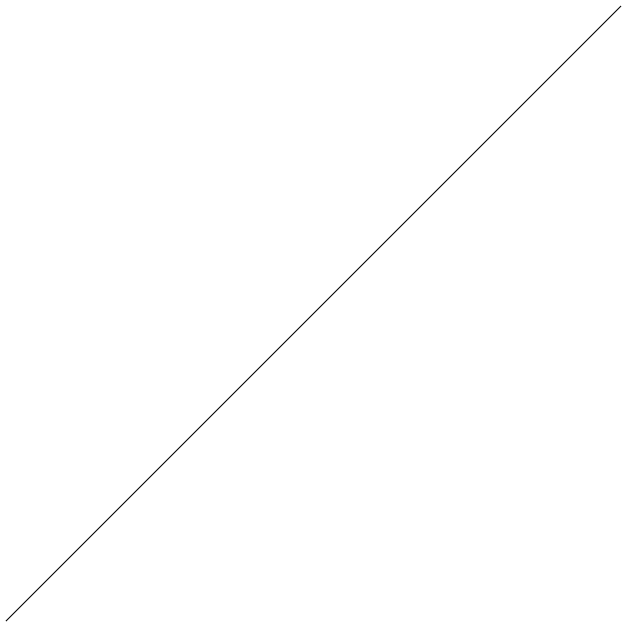
```
% // InputForm
Graphics[{{{}, {}, {Hue[0.67, 0.6, 0.6],
 Line[{{-1.999996, -0.9999960000000001},
 {1.8472175954999577, 2.8472175954999575},
 {1.999996, 2.9999960000000003}}]}}},
 {AspectRatio -> GoldenRatio^(-1),
 Axes -> True, AxesOrigin -> {0, 0},
 PlotRange -> {{-2, 2}, {-0.9999960000000001,
 2.9999960000000003}},
 PlotRangeClipping -> True,
 PlotRangePadding -> {Scaled[0.02],
 Scaled[0.02]}}]
```

This output is in the form `Graphics[primitives_,options_]`, where `primitives` is a list of graphics primitives and `options` is a list of options. *Graphics primitives* are the building blocks of graphics objects; you can build your own list of graphics primitives and display them using the `Graphics` command.

Graphics primitives are important for this book since initial approximations to fractal sets will frequently be expressed in terms of graphics primitives.

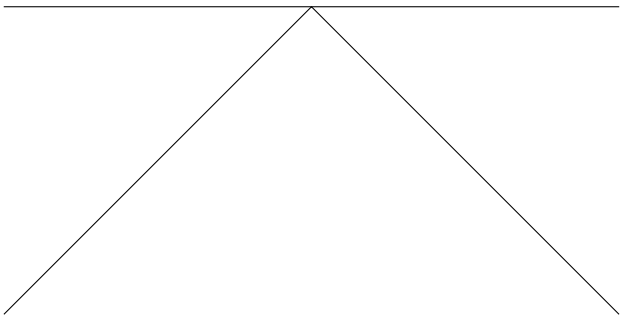
There are many two-dimensional graphics primitives; perhaps the simplest are `Line` and `Polygon`. `Line` accepts a single list of the form `{{x1,y1},...,{xn,yn}}` to define the line through the points with given coordinates. If the points are not collinear, then this defines a polygonal path. If we encase such a line in the `Graphics` command, *Mathematica* will display the line. For example, here is the line from the origin to $\{1, 1\}$.

```
Graphics[Line[{{0, 0}, {1, 1}}]]
```



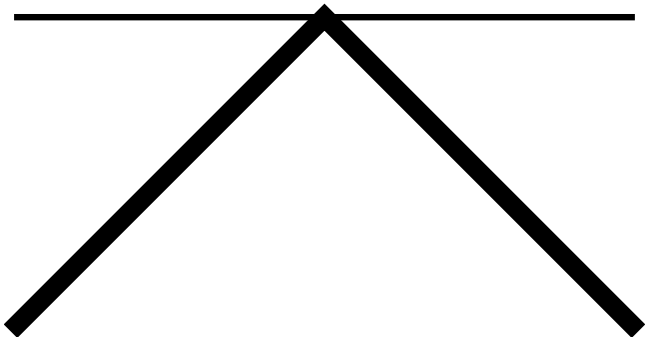
The argument of `Line` can take the more complicated form `{{{x1,x2}...}...}` to yield multiple lines.

```
Graphics[Line[{{{0, 1}, {2, 1}},
   {{0, 0}, {1, 1}, {2, 0}}}]]
```



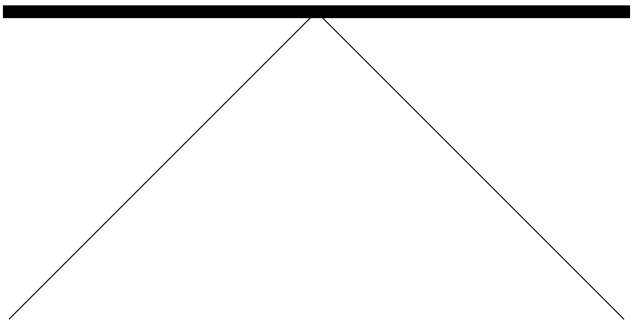
The appearance of a graphics primitive can be affected using a *graphics directive*. For example, the `Thickness` directive affects the thickness of a `Line`. A graphics directive should be placed in the list of graphics primitives and will affect any applicable graphics primitives which follow it until changed by another graphics directive. If we want to use conflicting graphics directives, we need distinct graphics primitives. For example, we can distinguish the two lines above as follows.

```
Graphics[{
  Thickness[.01], Line[{{0, 1}, {2, 1}}],
  Thickness[.03], Line[{{0, 0}, {1, 1}, {2, 0}}]
 }]
```



Nested lists can be used inside the `Graphics` command. In this case, a graphics directive does not have any effect outside the list it is in. For example, only the top line is affected by the `Thickness` directive the next example.

```
Graphics[{{Thickness[.02], Line[{{0, 1}, {2, 1}}]},
  Line[{{0, 0}, {1, 1}, {2, 0}}]
 }]
```



The `Polygon` primitive also accepts a list of points, but represents a filled polygon.

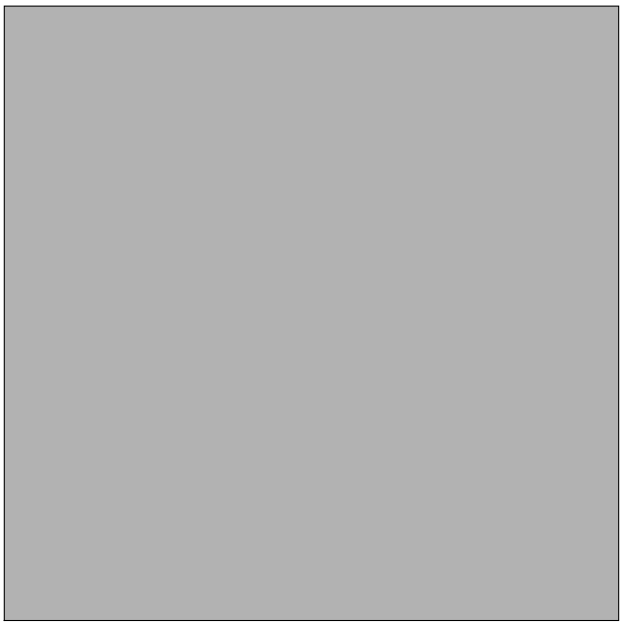
```
Graphics[
 Polygon[{{0, 0}, {1, 0}, {1, 1}, {0, 1}}]]
```

Color directives can be used to change the color.

```
Graphics[{GrayLevel[.7],
  Polygon[{{0, 0}, {1, 0}, {1, 1}, {0, 1}}]}]
```

We can combine `Line` and `Polygon` primitives to include a border, although the `EdgeForm` directive provides a nicer way to do this.

```
vertices = {{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}};
Graphics[{
  {GrayLevel[.7], Polygon[vertices]},
  Line[vertices]},
 AspectRatio → Automatic]
```



A.9 Manipulating lists

Mathematica can act on lists in many ways. Here is a list of randomly chosen integers.

```
SeedRandom[1];
theList = RandomInteger[{1, 10}, {10}]
{2, 5, 1, 8, 1, 1, 9, 7, 1, 5}
```

We can sort the list

```
sortedList = Sort[theList]
{1, 1, 1, 1, 2, 5, 5, 7, 8, 9}
```

We can reverse the order.

```
Reverse[sortedList]
{9, 8, 7, 5, 5, 2, 1, 1, 1, 1}
```

We can shift the list.

```
RotateLeft[sortedList, 3]
{1, 2, 5, 5, 7, 8, 9, 1, 1, 1}
```

We can drop some elements from the list

```
Drop[sortedList, 3]
{1, 2, 5, 5, 7, 8, 9}
```

We can access elements from the list

```
sortedList[[3]]
1
```

We can partition the list into subparts.

```
Partition[sortedList, 2]
{{1, 1}, {1, 1}, {2, 5}, {5, 7}, {8, 9}}
```

Nested lists can be flattened to remove the nested structure.

```
Flatten[%]
{1, 1, 1, 1, 2, 5, 5, 7, 8, 9}
```

We can select data from the list that satisfies some particular criteria. For example, we can select the odd numbers from the list.

```
Select[theList, OddQ]
{5, 1, 1, 1, 9, 7, 1, 5}
```

The second argument to `Select` should be a function returning either True or False. For example, `OddQ` returns True if called with an odd integer argument or False otherwise.

```
{OddQ[2], OddQ[3]}
{False, True}
```

Pure functions are convenient in this context. For example, `(#>5)&` represents a function which returns True if called with a numeric argument which is bigger than 5 or False if called with a numeric argument which is less or equal to 5. Thus the following returns those elements of the list which are greater than 5.

```
Select[theList, # > 5 &]
{8, 9, 7}
```

As we've seen, many functions act on the individual elements of a list.

```
sortedList^2
{1, 1, 1, 1, 4, 25, 25, 49, 64, 81}
```

In this example, we say that the square function has *mapped* onto the list. A function which automatically maps onto lists is said to be *listable*. The arbitrary function is not listable.

```
Clear[f];
f[theList]
f[{2, 5, 1, 8, 1, 1, 9, 7, 1, 5}]
```

Any function can be mapped onto a list using the `Map` command.

```
Map[f, theList]
{f[2], f[5], f[1], f[8], f[1], f[1], f[9], f[7], f[1], f[5]}
```

`Map` can be abbreviated using `/@`.

```
f /@ theList
{f[2], f[5], f[1], f[8], f[1], f[1], f[9], f[7], f[1], f[5]}
```

We can map a pure function onto a list. For example, we can use a function which accepts a number and returns a list whose elements are the original number and the original number squared.

```
{#, #^2} & /@ theList
{{2, 4}, {5, 25}, {1, 1}, {8, 64}, {1, 1}, {1, 1}, {9, 81}, {7, 49}, {1, 1}, {5, 25}}
```

We can view the internal representation of a list using the `FullForm` command.

```
theList // FullForm
List[2, 5, 1, 8, 1, 1, 9, 7, 1, 5]
```

All non-atomic expressions in *Mathematica* are represented this way. That is all expressions other than numbers or symbols (called atoms) are represented `head[args___]`. We can change the head of an expression using the `Apply` command. For example,

```
Apply[f, theList]
f[2, 5, 1, 8, 1, 1, 9, 7, 1, 5]
```

As with `Map`, there is an abbreviated form, namely `@@`.

```
f @@ theList
f[2, 5, 1, 8, 1, 1, 9, 7, 1, 5]
```

We can use this in conjunction with the `Plus` command to add the elements of a list.

```
Plus @@ theList
40
```

Here is a function which finds the average of a list.

```
average[l_] := Total[l] / Length[l];
average[theList]
4
```

Some more advanced techniques to apply functions to lists include the commands `Through`, `Inner`, and `Outer`. These commands are used in some of the programs defined in the `FractalGeometry` packages. `Through` is similar

to Map, but applies each of a list of functions to a single argument, rather than a single function to a list of arguments.

```
Through[{f1, f2, f3}[x]]
{f1[x], f2[x], f3[x]}
```

Inner and Outer are a bit more complicated and will be discussed in appendix A2 on linear algebra.

A.10 Iteration

Iteration is fundamental to both fractal geometry and computer science. If a function f maps a set to itself, then an initial value x_0 can be plugged into the function to obtain the value x_1. This value can then be plugged back into f continuing the process inductively. This yields the sequence $\{x_0, x_1, x_2, ...\}$ where $x_n = f(x_{n-1})$ for every integer $n > 0$. Put another way, $x_n = f^n(x_0)$ where f^n represents the n-fold composition of f with itself. The basic commands which perform this operation in *Mathematica* are Nest and NestList. Nest[f,x0,n] returns $f^n(x_0)$.

```
Clear[f];
Nest[f, x0, 5]
f[f[f[f[f[x0]]]]]
```

NestList[f,x,n] returns the list $\{x_0, f(x_0), f(f(x_0)), \ldots\}$.

```
NestList[f, x0, 5]
{x0, f[x0], f[f[x0]], f[f[f[x0]]], f[f[f[f[x0]]]], f[f[f[f[f[x0]]]]]}
```

Of course, this will be more interesting if we use a specific function and a specific starting point. For example, iteration of Cos starting at 0.8, should yield a sequence which converges to the fixed point of Cos. Note that we type Cos instead of Cos[x] since NestList anticipates a function.

```
NestList[Cos, .8, 30]
{0.8, 0.696707, 0.76696, 0.720024, 0.75179, 0.730468, 0.744863,
 0.735181, 0.741709, 0.737315, 0.740276, 0.738282, 0.739626, 0.738721, 0.73933,
 0.73892, 0.739196, 0.73901, 0.739136, 0.739051, 0.739108, 0.73907, 0.739096,
 0.739078, 0.73909, 0.739082, 0.739087, 0.739084, 0.739086, 0.739084, 0.739086}
```

The command FixedPoint[f,x0] automatically iterates f starting at x0 until the result no longer changes, according to *Mathematica*'s internal representation.

```
FixedPoint[Cos, .8]
0.739085
```

A.11 Pattern matching

When a function is declared, internally *Mathematica* adds a *pattern* to its global rule base. As we have seen, here is how to define $f(x) = x^2$.

```
f[x_] := x^2

f[2]
4
```

If we have an algebraic expression, rather than a function, we can do this manually using a *rule*.

```
x^2 /. x → 2
4
```

In this example, the expression is x^2, the rule is $x \to 2$, and `/.` is the replacement operator which tells *Mathematica* to implement the rule. The arrow $\to$ can also be typed `->` which *Mathematica* automatically converts to $\to$ in an input cell.

Commands for solving algebraic formulae return lists of rules. This makes it easy to plug solutions back into expressions. For example, suppose we use `Solve` to find the critical points of a polynomial.

```
poly = x^3 + 4 x^2 - 3 x + 1;
cps = Solve[D[poly, x] == 0, x]
```

$$\left\{\{x \to -3\}, \left\{x \to \frac{1}{3}\right\}\right\}$$

We can plug the critical points back into `poly`.

```
poly /. cps
```

$$\left\{19, \frac{13}{27}\right\}$$

An underscore is used to denote a more general pattern. For example, suppose we want to square the *integers* in a list.

```
{1, 2, a, b, π} /. {1 → 1^2, 2 → 2^2}
{1, 4, a, b, π}
```

We have used a list of rules - one for each integer in the list. We would prefer a pattern which matches each integer. One way to do this is as follows.

```
{1, 2, a, b, π} /. n_Integer → n^2
{1, 4, a, b, π}
```

Any functions to the right of the `Rule` operator `->` are evaluated immediately. There is a delayed version of `Rule` called `RuleDelayed` which waits

until the rule is applied to evaluate any functions. `RuleDelayed` can be abbreviated by :→ which can be typed as `:>`. The difference between → and :→ is similar to the difference between `=` and `:=`. Suppose, for example, we want to expand any powers in an expressions. We might try the following.

```
(a + b)^1/2 (c + d)^3 /. x_^n_ → Expand[x^n]
```

$$\sqrt{a + b}\ (c + d)^3$$

This doesn't work since the `Expand` is evaluated immediately to yield x^n. Thus this rule is equivalent to $x^n \to x^n$ which does nothing. By contrast, the rule delayed version waits until any substitutions occur to perform the expansion.

```
(a + b)^1/2 (c + d)^3 /. x_^n_ :→ Expand[x^n]
```

$$\sqrt{a + b}\ \left(c^3 + 3\,c^2\,d + 3\,c\,d^2 + d^3\right)$$

Pattern matching is a powerful and important yet subtle aspect of *Mathematica*.

A.12 Programming

Mathematica has a full featured, high-level programming language including loop constructs such as `Do` and `While`, flow control devices such as `If` and `Switch`, and scoping constructs which allow you to declare local variables. Programming with *Mathematica* is a deep subject and entire books, such as [Mae], have been written about it. In this appendix, we'll develop one short program which illustrates some important themes for this book. Of course, the book contains many more such examples.

A word of warning is in order to experienced programmers. The procedural programming paradigm used for many compiled languages, such as C, C++, or Java, is emphatically not appropriate for a high-level language such as *Mathematica*. This book primarily follows a paradigm known as functional programming and, to a lesser extent, rule based programming. The idea is to use *Mathematica*'s vast set of tools to define functions which act naturally on *Mathematica*'s rich set of data structures. The "natural" way to do something in any language is determined by the internal structure of the language; it takes experience to learn what the most appropriate technique for a given situation might be. Consider, for example, the following procedural approach to generating a list of squares of the first 10 squares.

```
squareList = {};
For[i = 1, i ≤ 10, i++,
 squareList = AppendTo[squareList, i^2]]
squareList
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

Most people with procedural programming experience should be comfortable with this example. If you are not, don't worry because this is not the way to do it anyway. It is much more natural in *Mathematica* to do the following.

```
Range[10]^2
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

The functional approach is much shorter and more readable. More importantly, the functional approach generally executes faster. Suppose we want to generate a list of 20,000 squares and measure how long it takes. Here is a procedural approach.

```
squareList = {};
For[i = 0, i ≤ 20 000, i++,
   squareList = AppendTo[squareList, i^2]]; // Timing
{2.15142, Null}
```

That's slow! A big part of the problem is that `AppendTo` is a very expensive operation, particularly when applied to long lists. This can be sped up considerably by generating an initial list and modifying it in place as follows.

```
squareList = Range[20 000];
For[i = 0, i ≤ 20 000, i++,
   squareList[[i]] = i^2]; // Timing
{0.049875, Null}
```

The functional approach is still considerably faster.

```
Range[20 000]^2; // Timing
{0.000691, Null}
```

For a more serious example, consider the first picture describing the construction of the Cantor set which appears in chapter 2. To generate this picture, we first define a function `CantorStep` which will be defined to act on `Line` primitives and nested lists of `Line` primitives. If `CantorStep` encounters an object of the form `Line[x_]`, it should return a list of two lines, one contracted towards the origin by a factor `r1` and the other contracted towards 1 by a factor `r2`. For the classical Cantor set, `r1` and `r2` are both 1/3; we would like to include more general contraction ratios so we define `CantorStep` in terms of parameters `r1` and `r2`.

We first define the action of `CantorStep` on `Line` primitives.

```
r1 = r2 = 1 / 3;
CantorStep[Line[x_]] :=
  N[{Line[r1 x], Line[r2 x + {{1 - r2, 0}, {1 - r2, 0}}]}];
```

Let's check to see how `CantorStep` works on the unit interval.

```
firstStep = CantorStep[Line[{{0, 0}, {1, 0}}]]
```

```
{Line[{{0., 0.}, {0.333333, 0.}}], Line[{{0.666667, 0.}, {1., 0.}}]}
```

Note that we force numerical approximation using the `N` function. This is important in iterative fractal programming, because computations with decimal approximations is much faster than computations with exact rational numbers.

Next, if `CantorStep` encounters a list, it should map over that list.

```
CantorStep[l_List] := CantorStep /@ l;
```

This version should now be applicable to `firstStep`.

```
CantorStep[firstStep]
```

```
{{Line[{{0., 0.}, {0.111111, 0.}}], Line[{{0.666667, 0.}, {0.777778, 0.}}]},
 {Line[{{0.222222, 0.}, {0.333333, 0.}}], Line[{{0.888889, 0.}, {1., 0.}}]}}
```

We should be able to display this approximation.

```
Graphics[%, AspectRatio → 1 / 10]
```


We can define a function `CantorSet` which uses the `Nest` command to generate a higher level approximation.

```
CantorSet[n_Integer] :=
  Nest[CantorStep, Line[{{0, 0}, {1, 0}}], n];
Graphics[CantorSet[5],
 AspectRatio → 1 / 10]
```


We can now generate a list of approximations and display them using `GraphicsColumn`.

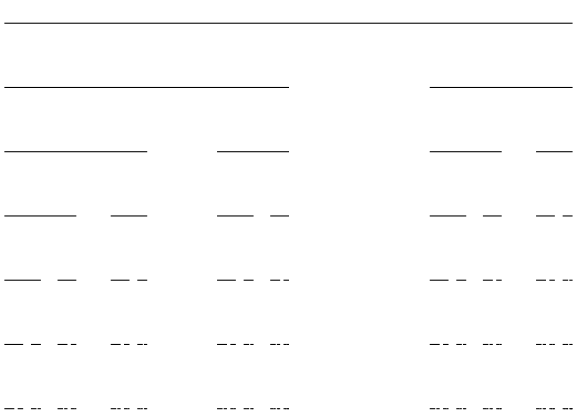
```
approximations = Table[Graphics[CantorSet[n],
    AspectRatio → 1 / 10], {n, 0, 5}];
GraphicsColumn[approximations]
```

When you have developed many lines of code that will be implemented many times with small modifications, you might consider encapsulating that code in a `Module`, which allows you to declare local variables. The `Module` command has the syntax `Module[localVars_, body_]`, where `localVars` is a list of the local variables and `body` is the sequence of *Mathematica* commands to be executed. Here is how to define a function `ShowCantorApproximations` which encapsulates the code we have just discussed.

```
ShowCantorApproximations[depth_Integer, r1_, r2_] :=
  Module[{CantorStep, CantorSet, approximations},
   CantorStep[Line[x_]] :=
    N[{Line[r1 x], Line[r2 x + {{1 - r2, 0}, {1 - r2, 0}}]}];
   CantorStep[l_List] := CantorStep /@ l;
   CantorSet[n_Integer] :=
    Nest[CantorStep, Line[{{0, 0}, {1, 0}}], n];
   approximations = Table[{Graphics[CantorSet[n],
       AspectRatio → 1 / 10]}, {n, 0, depth}];
   GraphicsGrid[approximations]
  ];
```

`ShowCantorApproximations` takes three arguments which appear in the body of the module. We can use this function to generate the figure in chapter 2, which shows the construction of a Cantor type set with contraction ratios 1/2 and 1/4.

```
ShowCantorApproximations[6, 1 / 2, 1 / 4]
```



A.13 Notes

Exercises

A.1 Enter and evaluate the following input cells.

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

```
Sum[1 / n^2, {n, 1, Infinity}]
```

$$\int_0^{\infty} e^{\frac{-x^2}{2}}\, dx$$

```
Integrate[Exp[-x^2 / 2], {x, 0, Infinity}]
```

A.2 For both of the following functions, use the commands `D`, `Solve`, `NSolve`, and/or `FindRoot` to find the critical points of the function. Then use `Plot` to graph the function.

1 $f(x) = x^3 - 6x^2 + 10x - 2$
2 $g(x) = x \sin\left(x^2\right)$

A.3 Use the `Table` command and the trigonometric functions to generate a list of the vertices of a regular hexagon of side length 1.

A.4 Use graphics primitives to create a picture of a regular hexagon and a picture of a square inscribed inside an equilateral triangle as in figure A.1.

A.5 Write a function `nGon` which accepts a positive integer n and returns a list of graphics primitives describing a regular n-sided polygon. Thus your function should be able to generate figure A.2.

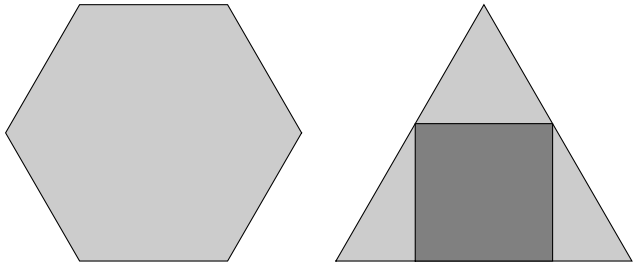


Figure A.1 Graphics exercises

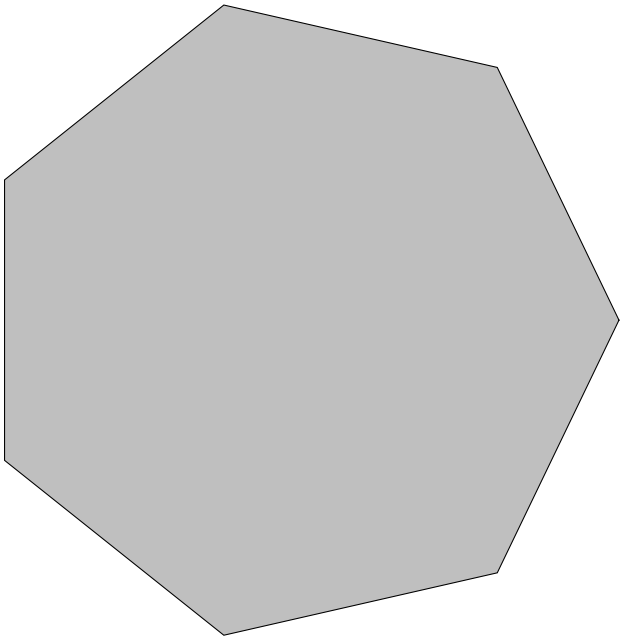


Figure A.2 A heptagon

Appendix B
Linear Algebra

Linear algebra is important for fractal geometry since it provides a convenient notation for describing iterated function systems. Linear algebra is the study of linear transformations of vector spaces, which are represented by matrices. The most important part of linear algebra to understand for this book is how the basic two dimensional vector space $\mathbb{R}^2$ is transformed under multiplication by specific types of matrices. We will also need to compute eigenvalues and eigenvectors to determine the fractal dimension of digraph self-similar sets. Finally, certain programming constructs are best understood in the context of matrices.

B.1 Vectors and matrices

Vectors are represented as lists of the appropriate length in *Mathematica*. Thus a two dimensional vector is simply a list of length two. Matrices are represented as lists of lists. A two dimensional matrix should be a list of length two and each of its elements should be a list of length two; the first element should represent the first row and the second element should represent the second row. You can pass a matrix to the `MatrixForm` command to see it typeset as a matrix.

```
{{a, b}, {c, d}} // MatrixForm
```

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

You can also enter matrices directly using the $\begin{pmatrix} \square & \square \\ \square & \square \end{pmatrix}$ button from the BasicInputs palette or the using **Table/Matrix** command from the **Insert** menu.

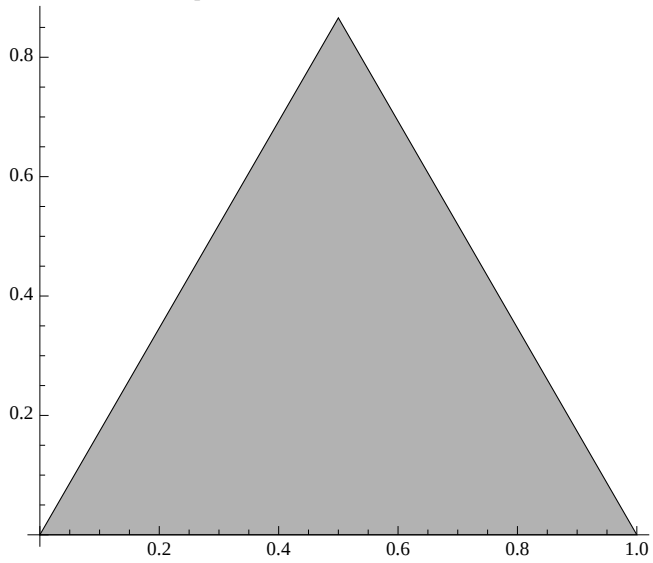
Matrix multiplication is represented using a `.` between two matrices or between a matrix and a vector.

```
{{a, b}, {c, d}}.{x₁, x₂}
```

$$\{a\,x_1 + b\,x_2,\ c\,x_1 + d\,x_2\}$$

We can now explore the effect of multiplication by certain types of matrices on the general two dimensional vector. First we create a simple picture to act on.

```
triangle = Graphics[{{{GrayLevel[.7],
    Polygon[{{0, 0}, {1, 0},
      {1 / 2, √3 / 2}}]},
   {Line[{{0, 0}, {1, 0},
      {1 / 2, √3 / 2}, {0, 0}}]}},
  Axes → True]
```

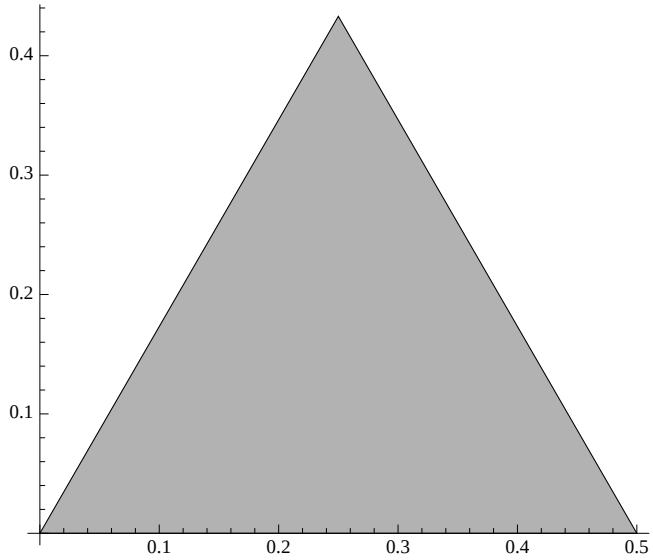


The following simple function accepts a `Graphics` object and a matrix and applies the linear transformation defined by the matrix to all two-dimensional numerical vectors in the `Graphics` object.

```
transform[Graphics[g_, opts___], M_?MatrixQ] :=
  Graphics[GeometricTransformation[g,
    AffineTransform[M]], opts];
```

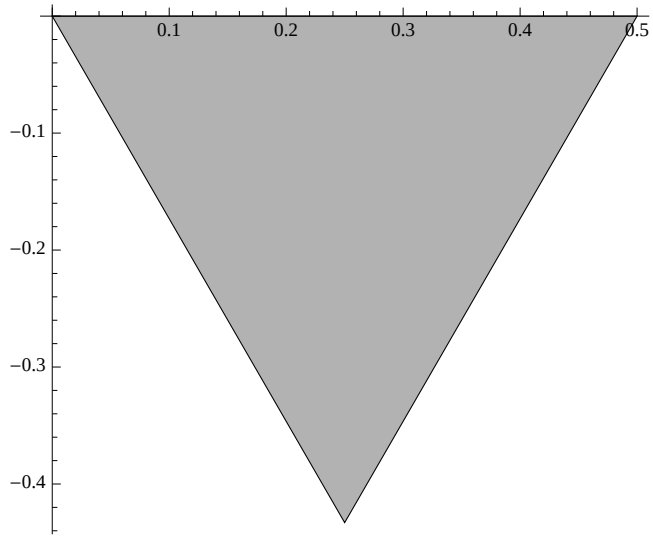
Now, a diagonal matrix $\begin{pmatrix} r & 0 \\ 0 & r \end{pmatrix}$ should stretch or compress the graphic by the factor r. (Note the axes in the next picture.)

```
M = {{1 / 2, 0}, {0, 1 / 2}};
transform[triangle, M]
```



We can introduce a reflection by placing a minus sign in the correct place.

```
M = {{1 / 2, 0}, {0, -1 / 2}};
transform[triangle, M]
```



Rotation is a bit trickier. Rotation about the origin through the angle θ can be represented by the matrix

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}.$$

This is exactly the matrix encoded in the *Mathematica* function `RotationMatrix`.

```
RotationMatrix[θ] // MatrixForm
```

$$\begin{pmatrix} \text{Cos}[\theta] & -\text{Sin}[\theta] \\ \text{Sin}[\theta] & \text{Cos}[\theta] \end{pmatrix}$$

In fact, there is a `RotationTransform` command analogous to the `AffineTransform` command we used above, but it defeats the purpose of our linear algebra review.

Now, positive θ implies counter-clockwise rotation, while negative θ implies clockwise rotation. To see why this matrix works, first apply it to the standard basis vectors $\vec{\imath} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\vec{\jmath} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. This should simply extract the two columns of the rotation matrix. We first consider the matrix multiplied by $\vec{\imath}$.

```
{{Cos[θ], -Sin[θ]}, {Sin[θ], Cos[θ]}}.{1, 0}
{Cos[θ], Sin[θ]}
```

By basic trigonometry, this is the basis vector $\vec{\imath}$ rotated through the angle θ. We next look at the matrix multiplied by $\vec{\jmath}$.

```
{{Cos[θ], -Sin[θ]}, {Sin[θ], Cos[θ]}}.{0, 1}
{-Sin[θ], Cos[θ]}
```

Now the basis vector $\vec{\jmath}$ can be written

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \pi/2 \\ \sin \pi/2 \end{pmatrix}.$$

Thus $\vec{\jmath}$ rotated by the angle theta can be written

$$\begin{pmatrix} \cos(\theta + \pi/2) \\ \sin(\theta + \pi/2) \end{pmatrix} = \begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix},$$

which exactly the rotation matrix multiplied by $\vec{\jmath}$. This last equality can be seen by applying basic trigonometric identities using `TrigExpand`.

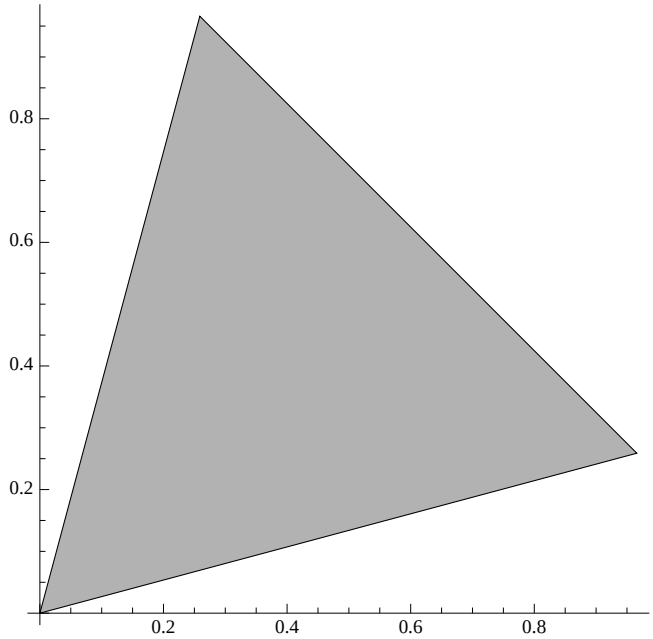
```
{Cos[θ + π / 2], Sin[θ + π / 2]} // TrigExpand
{-Sin[θ], Cos[θ]}
```

Now since M rotates both $\vec{\imath}$ and $\vec{\jmath}$ by the angle θ, it rotates any vector through that angle by linearity since

$$M \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = M \left(x_1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + x_2 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) = x_1 M \begin{pmatrix} 1 \\ 0 \end{pmatrix} + x_2 M \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

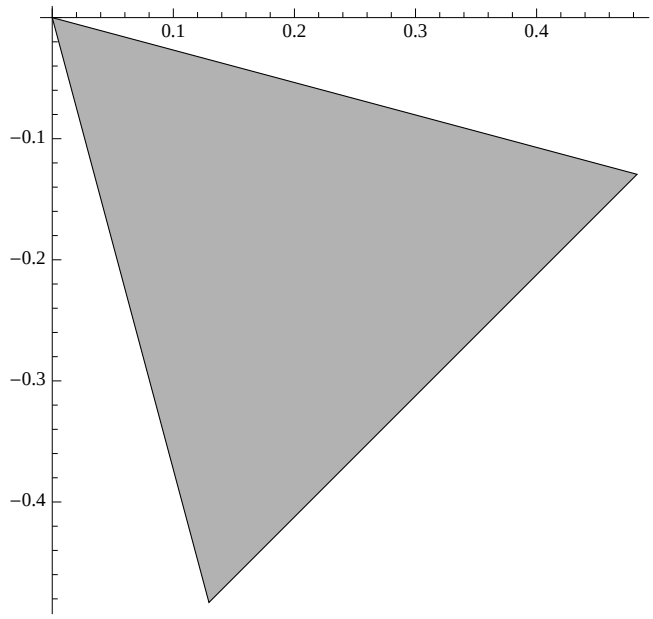
For example, we can rotate our triangle rotated through the angle $\pi/12$.

```
M = {{Cos[π / 12], -Sin[π / 12]},
    {Sin[π / 12], Cos[π / 12]}};
Show[transform[triangle, M],
 Axes → True, AspectRatio → Automatic]
```



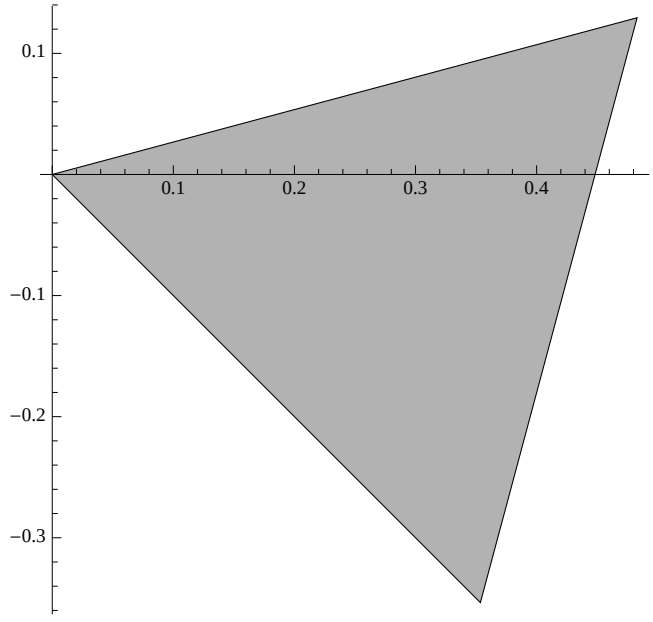
We can combine these geometric transformations by multiplying the appropriate matrices.

```
compression = {{1 / 2, 0}, {0, 1 / 2}};
reflection = {{1, 0}, {0, -1}};
rotation = {{Cos[π / 12], -Sin[π / 12]},
    {Sin[π / 12], Cos[π / 12]}};
M = reflection.rotation.compression;
transform[triangle, M]
```



These operations are not generally commutative.

```
M = rotation.reflection.compression;
transform[triangle, M]
```

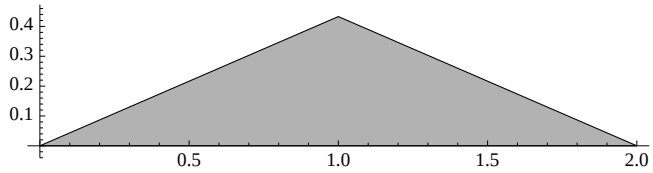


When we enter the broader world of *affine* functions, rather than just similarities, there are other possibilities. A diagonal matrix

$$\begin{pmatrix} a & 0 \\ 0 & d \end{pmatrix}$$

should stretch or compress the graphic by the factor a in the horizontal direction and d in the vertical direction.

```
M = {{2, 0}, {0, 1 / 2}};
transform[triangle, M]
```

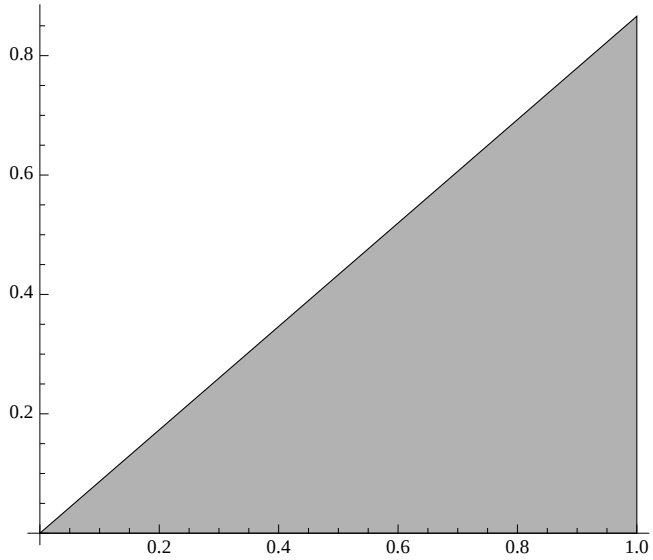


There are also *shear transformations*, which have the form

$$\begin{pmatrix} 1 & s \\ 0 & 1 \end{pmatrix} \text{ or } \begin{pmatrix} 1 & 0 \\ s & 1 \end{pmatrix}.$$

You can see why these are called shear transformations by observing their geometric effect.

```
M = {{1, 1 / Sqrt[3]}, {0, 1}};
transform[triangle, M]
```



You can see why this works by observing

$$\begin{pmatrix} 1 & s \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x+sy \\ y \end{pmatrix}.$$

B.2 Eigenvalues and eigenvectors

The set of eigenvectors and eigenvalues forms a more advanced way to understand the geometric behavior of certain matrices. We will use eigenvalues to compute the fractal dimension of digraph self-similar sets and related fractal sets. A vector v is an *eigenvector* of a matrix M if there is a number λ so that $Mv = \lambda v$. In this case, λ is the corresponding *eigenvalue*. The collection of eigenvalues and eigenvectors is sometimes called the *eigensystem* of the matrix. The eigenvector determines a one dimensional subspace which is invariant under M and the eigenvalue determines the factor by which M expands or compresses this subspace. *Mathematica* has built in commands `Eigenvalues`, `Eigenvectors`, and `Eigensystem` to compute these.

```
M = {{1 / 2, 1}, {0, 2}};
Eigenvalues[M]
```

$$\left\{2, \frac{1}{2}\right\}$$

```
Eigenvectors[M]
```

$$\left\{\left\{\frac{2}{3}, 1\right\}, \{1, 0\}\right\}$$

Thus this matrix should compress the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ by the factor 1/2 and it should expand the vector $\begin{pmatrix} 2/3 \\ 1 \end{pmatrix}$ by the factor 2.

```
M.{1, 0}
```

$$\left\{\frac{1}{2}, 0\right\}$$

```
M.{2 / 3, 1}
```

$$\left\{\frac{4}{3}, 2\right\}$$

The command `Eigensystem` returns a list of two lists; the first list contains the eigenvalues and the second list contains the corresponding eigenvectors.

```
Eigensystem[M]
```

$$\left\{\left\{2, \frac{1}{2}\right\}, \left\{\left\{\frac{2}{3}, 1\right\}, \{1, 0\}\right\}\right\}$$

Finally, the commands `Inner` and `Outer` generalize the dot and matrix products of vectors. These commands will be useful when implementing the digraph iterated function system scheme. The dot product of two vectors simply multiplies termwise and adds the results. The dot product of w and v is denoted `w.v`. Vectors are represented as lists. For example,

```
w = {1, 2, 3};
v = {a, b, c};
w.v
```
```
a + 2 b + 3 c
```

The inner product generalizes this by allowing operations other than addition or multiplication to be used. In general, operations can be defined by multivariate functions `f` and `g` where `f` plays the role of multiplication and `g` plays the role of addition. `Inner` implements this generalized inner product as follows.

```
Clear[f, g];
Inner[f, {1, 2, 3}, {a, b, c}, g]
```
```
g[f[1, a], f[2, b], f[3, c]]
```

Note that if `f` is `Times` denoting multiplication and `g` is `Plus` denoting addition, then we recover the basic dot product.

```
Inner[Times, {1, 2, 3}, {a, b, c}, Plus]
```
```
a + 2 b + 3 c
```

One way to think of the inner product is as matrix multiplication of a row vector times a column vector.

```
(1  2  3).{{a}, {b}, {c}} // MatrixForm
```
```
( a + 2 b + 3 c )
```

If we multiply a column vector by a row vector, we obtain the outer product. Note that the lengths of the vectors need not be the same.

```
{{a}, {b}, {c}}.(1  2  3  4) // MatrixForm
```
$$\begin{pmatrix} a & 2a & 3a & 4a \\ b & 2b & 3b & 4b \\ c & 2c & 3c & 4c \end{pmatrix}$$

This can be generalized using the command `Outer`

```
Outer[f, {a, b, c}, {1, 2, 3, 4}] // MatrixForm
```

$$\begin{pmatrix} f[a, 1] & f[a, 2] & f[a, 3] & f[a, 4] \\ f[b, 1] & f[b, 2] & f[b, 3] & f[b, 4] \\ f[c, 1] & f[c, 2] & f[c, 3] & f[c, 4] \end{pmatrix}$$

If we use `Times` in place of `f`, we recover the usual outer product.

```
Outer[Times, {a, b, c}, {1, 2, 3, 4}] // MatrixForm
```

$$\begin{pmatrix} a & 2\,a & 3\,a & 4\,a \\ b & 2\,b & 3\,b & 4\,b \\ c & 2\,c & 3\,c & 4\,c \end{pmatrix}$$

Exercises

B.1 Create a graphic representing a square with center at the origin and sides parallel to the coordinate axes. This graphic will called The Graphic throughout these exercises.

B.2 Write down a matrix which transforms The Graphic into a square with width 1/2 and height 4. Use *Mathematica* to visualize this transformation.

B.3 Write down a matrix which rotates The Graphic through the angle $\pi/4$.

B.4 Using `Do` or `Table` and our rotation techniques, generate the list of images shown in figure B.1, where each square is a rotation of the previous square through the angle $\pi/12$.

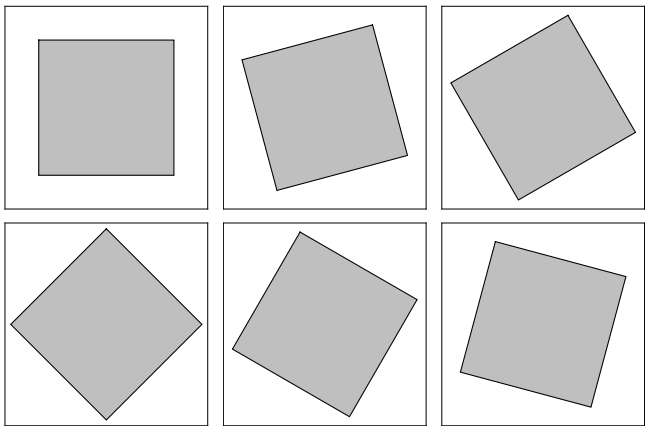


Figure B.1 A rotating square

Appendix C
Real Analysis

Real analysis is the branch of mathematics that grew out of an effort to understand the theoretical aspects of calculus. The main tools of calculus, the derivative and the integral, depend on the difficult notion of limit, which in turn depends on the basic structure of the real numbers. This theoretical framework describing $\mathbb{R}$ and $\mathbb{R}^n$ is the natural setting to prove the major theorems of fractal geometry.

In this appendix, we outline just that part of real analysis relevant to this book. One proof is presented to introduce the flavor of real analysis, but most results are stated without proof. If you have had an undergraduate course in real analysis, then most of this material should be familiar. If this is your first exposure to the subject, this material should seem very complicated or confusing. No pretense is made that this material can be fully absorbed on a first reading and no effort is made to fully elucidate the material. A deep understanding of real analysis can only be acquired through long and difficult study. Much of this book can be appreciated without the use of real analysis, but an understanding of the theoretical framework certainly enhances the beauty of the subject. We look most closely at this theoretical framework in chapter 3, which will be particularly dependent on real analysis.

C.1 Sequences

A sequence can be thought of as an infinite list of real numbers. More precisely, a sequence is a function mapping $\mathbb{N} \to \mathbb{R}$. We frequently represent a sequence using the notation $\{a_n\}_{n=1}^{\infty}$. The most important issue concerning a sequence is that of *convergence*, i.e. does the sequence have a limit. Intuitively, a sequence $\{a_n\}_{n=1}^{\infty}$ is said to converge to the limit L if a_n can be made arbitrarily close to L by taking n to be sufficiently large. This statement is made precise by the following definition

Definition C.1 The sequence $\{a_n\}_{n=1}^{\infty}$ is said to *converge* to the limit L if for every $\varepsilon > 0$, there is an $N \in \mathbb{N}$ so that $|a_n - L| < \varepsilon$ whenever $n > N$. In this case, we write $\lim a_n = L$ or $a_n \to L$.

In this definition, ε may be thought of as an arbitrary error tolerance; it describes how close a_n is supposed to be to L. The N describes how large n should be to guarantee that $|a_n - L| < \varepsilon$. For example,

$$\lim \frac{2n}{3n+1} = \frac{2}{3}, \text{ because} \left| \frac{2n}{3n+1} - \frac{2}{3} \right| < \varepsilon$$

whenever $n > 1/(9\varepsilon)$.

In the definition of limit, we see that $a_n \to L$ if the value of a_n is close to L for large n, but we can't use this definition to prove that a sequence converges without knowing the limit first. Our first theorem, called the *Cauchy criterion*, provides a way to test whether a sequence converges without knowing the limit. Intuitively, a sequence is said to be Cauchy if the values of a_n are close to one another for large n. As we will see, this is equivalent to convergence of the sequence.

Definition C.2 The sequence $\{a_n\}_{n=1}^{\infty}$ is called *Cauchy* if for every $\varepsilon > 0$, there is an $N \in \mathbb{N}$ so that $|a_n - a_m| < \varepsilon$ whenever $m, n > N$.

Theorem C.3 *A sequence converges if and only if it is Cauchy.*

The Cauchy criterion is an "if and only if" statement so it provides a characterization of convergence without reference to the limit of the sequence. We will use it to prove the contraction mapping theorem. An application of the contraction mapping theorem allows us to establish the most fundamental result of this book: the existence and uniqueness of invariant sets of iterated function systems.

Since this theorem is so fundamental, let's prove one direction of the Cauchy criterion. This also provides us with a first look at a genuine proof in real analysis. The proof depends on the *triangle inequality* which says that $|a + b| \leq |a| + |b|$ for all real numbers a and b.

Lemma C.4 *If a sequence converges, then it is Cauchy.*

Proof Suppose that the sequence $\{a_n\}_{n=1}^{\infty}$ converges to L and let $\varepsilon > 0$. We want some $N \in \mathbb{N}$ which is large enough so that $|a_n - a_m| < \varepsilon$ whenever $m, n > N$. Since $\{a_n\}_{n=1}^{\infty}$ converges to L, there is an $N \in \mathbb{N}$ which is large enough so that $|a_n - L| < \varepsilon/2$ whenever $n > N$. Thus if $m, n > N$,

$$|a_n - a_m| = |a_n - L + L - a_m| \leq |a_n - L| + |L - a_m| < \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon.$$

□

Given a sequence $\{a_n\}_{n=1}^{\infty}$, we can form the related *series* $\sum_{n=1}^{\infty} a_n$, which should be familiar to most students of calculus. We will occasionally refer to the *geometric series formula* namely

$$\sum_{n=m}^{\infty} ar^n = \frac{ar^m}{1-r},$$

provided $|r| < 1$.

C.2 The basic structure of $\mathbb{R}$ and $\mathbb{R}^n$

Most students of calculus will have familiarity with the basic structure of $\mathbb{R}$, although they might have taken it for granted. For example, $\mathbb{R}$ has an *algebraic* structure; i.e. it is a set with algebraic operations addition and multiplication which combine real numbers to form new real numbers.

The notions of an open interval versus a closed interval should also be familiar to calculus students. These ideas generalize to *open* and *closed sets* of $\mathbb{R}$ and of $\mathbb{R}^n$. The basic properties of open and closed sets define the *topological* structure of $\mathbb{R}$ or of $\mathbb{R}^n$. All of the fractals defined in this book are closed (in fact compact) sets because these are the natural invariant sets of iterated function systems. Let's carefully define the basic concepts of topology. These definitions are stated for $\mathbb{R}^n$. Of course, they apply to $\mathbb{R}$ as well by taking $n = 1$.

Definition C.5

1 Let $x \in \mathbb{R}^n$ and let $r > 0$. The set $B_r(x) = \{y : |y - x| < r\}$ is called the *open ball of radius r about x*. The set $\overline{B_r(x)} = \{y : |y - x| \leq r\}$ is called the *closed ball of radius r about x*.

$$B_r(x) = \{y : |y - x| < r\}$$

is called the *open ball of radius r about x*. The set

$$\overline{B_r(x)} = \{y : |y - x| \leq r\}$$

is called the *closed ball of radius r about x*.

2 A set $U \subset \mathbb{R}^n$ is called *open* if for every $x \in U$, there is an $r > 0$ such that $B_r(x) \subset U$.
3 A point $x \in \mathbb{R}^n$ is called a *boundary point* of a set $U \subset \mathbb{R}^n$ if for every $r > 0$, $B_r(x)$ contains points in U and in U^c, the complement of U.
4 A set is called *closed* if it contains all of its boundary points.
5 A point $x \in \mathbb{R}^n$ is called a *cluster point* of a set U if for every $r > 0$, $B_r(x)$ contains points in U.
6 A set U is called *bounded* if there is an $r > 0$ big enough so that $U \subset B_r(0)$.
7 A set E is called *compact* if every infinite subset of E has a cluster point in E.

Several comments are in order concerning these definitions. Compactness is an extremely important idea in topology. There are several alternative characterizations of compactness. Perhaps the most important theorem concerning compactness for this book states that a set $E \subset \mathbb{R}^n$ is compact if and only if it is closed and bounded. This is a useful and intuitive way for many readers to think of compactness. However, it should be emphasized that this characterization is not valid in the more general setting of metric spaces described later. The notions of open and closed sets are clearly complementary. In fact, a fundamental theorem states that a set is closed if and only if its complement is open.

The fundamental characteristic of the set of real numbers $\mathbb{R}$ which distinguishes it from the set of rational numbers $\mathbb{Q}$ is completeness. The set of real numbers is called *complete* because every non-empty set in $\mathbb{R}$ which is bounded above has a least upper bound in $\mathbb{R}$. The least upper bound of set $A \subset \mathbb{R}$ is also called the *supremum* of A and is denoted $\sup A$. The greatest lower bound of a set A is also called the *infimum* of A and denoted $\inf A$. For example, $\inf\{1/n : n \in \mathbb{N}\} = 0$.

C.3 Metric spaces

The absolute value function defines a notion of distance d between two points x and y in $\mathbb{R}^n$ by $d(x, y) = |x - y|$. The key algebraic properties of this notion of distance are reflexivity, symmetry, and the triangle inequality. That is, for all x, y, and z in $\mathbb{R}^n$,

$$d(x,x) = 0$$
$$d(x,y) = d(y,x)$$
$$d(x,z) \leq d(x,y) + d(y,z)$$

Conversely, any function satisfying these three criteria where x, y, and z belong to some set S defines a reasonable notion of distance on that set. Functions satisfying these properties are called *metrics* and a set S equipped with such a metric is called a *metric space*. If the metric space happens to have the property that any Cauchy sequence automatically converges, then the metric space is called complete.

In chapter 3, we will define a complete metric on the set of all non-empty compact subsets of $\mathbb{R}^n$ called the Hausdorff metric. The proof of the existence of invariant sets of iterated function systems involves the iteration of a function defined on this set.

C.4 Functions

Functions mapping $\mathbb{R} \to \mathbb{R}$ are the central objects of study in real analysis. We assume that the reader is familiar with the basic properties of limits and continuity of a function from calculus. An important fact that we will use is that the continuous image of a compact set is compact.

Later for the chapter on graphs of functions: Precise definition of continuity / Holder conditions

Appendix D
The Golden Ratio

The golden ratio $\varphi \approx 1.618$ is a beautiful and important irrational number whose status ranks right behind π and e. Many sources indicate that φ appears throughout ancient architecture, medieval art, and even the human body. Much of this is false, having been debunked in Markowsky (1992). There is no doubt, however, that the algebraic properties of the golden ratio lead to geometric constructions involving self-reproduction. In this short appendix, we focus on these mathematical properties of the golden ratio.

D.1 Definition

Suppose an interval of length a is divided into sub-intervals of length b and c. If the lengths of the sub-intervals are chosen so that $a/b = b/c$, then the division is called a *golden cut*. In words, the ratio of the whole to the longer part equals the ratio of the longer part to the shorter. This ratio is called the *golden ratio* and is denoted φ. We can compute a numerical value of φ as follows. Suppose we cut an interval of length φ with a golden cut. Then, as shown in figure D.1, the length of the longer sub-interval must be 1 so the length of the shorter must be $\varphi - 1$. The defining equation then yields $\varphi = 1/(\varphi - 1)$. Thus φ satisfies the quadratic equation $\varphi^2 - \varphi - 1 = 0$ so $\varphi = \left(1 + \sqrt{5}\right)/2 \approx 1.618$.

Figure D.1 A golden cut

D.2 Geometric properties

D.2.1 The golden rectangle

A rectangle whose sides are in the golden proportion is called a golden rectangle. The simplest golden rectangle has side lengths φ and 1. If we cut the longer side of such a rectangle in a golden cut with a line perpendicular to that side, we generate a square and a smaller rectangle as shown in figure D.2. The ratio of the longer side of the smaller triangle to the shorter is then $1/(\varphi - 1) = \varphi$. Thus, the smaller rectangle is another golden rectangle. We can use this geometric decomposition to generate a spiral of squares that fill a golden rectangle, as shown in figure D.3.

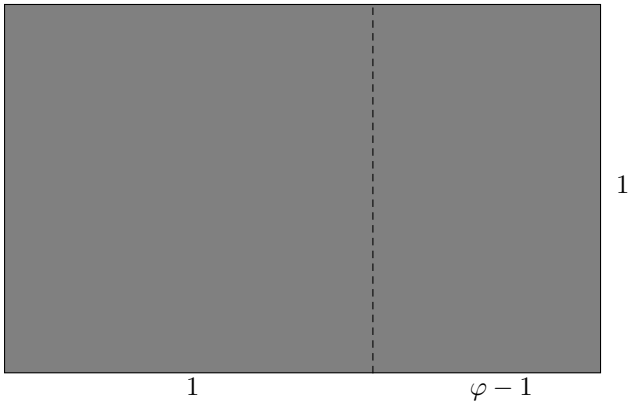


Figure D.2 A golden rectangle

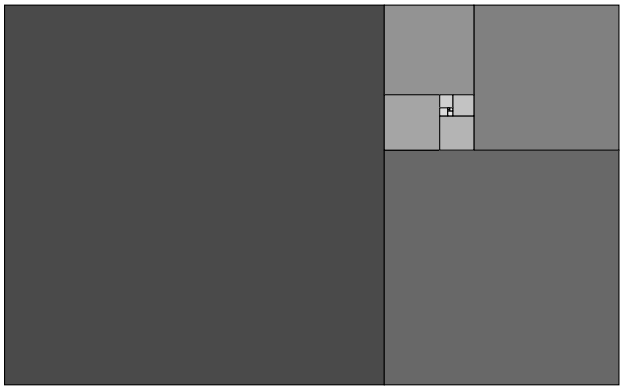


Figure D.3 A spiral of squares filling a golden rectangle

D.2.2 The golden triangles

Consider an isosceles triangle whose base angles are twice the remaining angle. The angle measures must then be $72 - 36 - 72$. Suppose we bisect one of the base angles with a line segment that joins that base angle to the opposite side, as shown in figure D.4 (a). This divides the triangle into two isosceles triangles, one with angles $36 - 108 - 36$ and another with angles $72 - 36 - 72$. Thus, this second triangle is similar to the original and we can use this information to determine the scaling factor between the two. Suppose the length of the longer side of the original triangle is φ while the length of the base is 1. (The symbol φ may be treated as an unknown here but it will turn out to be the golden ratio anyway.) Comparing the ratio of the length of a leg to the base in the larger triangle to the smaller, we see that $\varphi/1 = 1/(\varphi - 1)$. Thus this ratio is indeed the golden ratio.

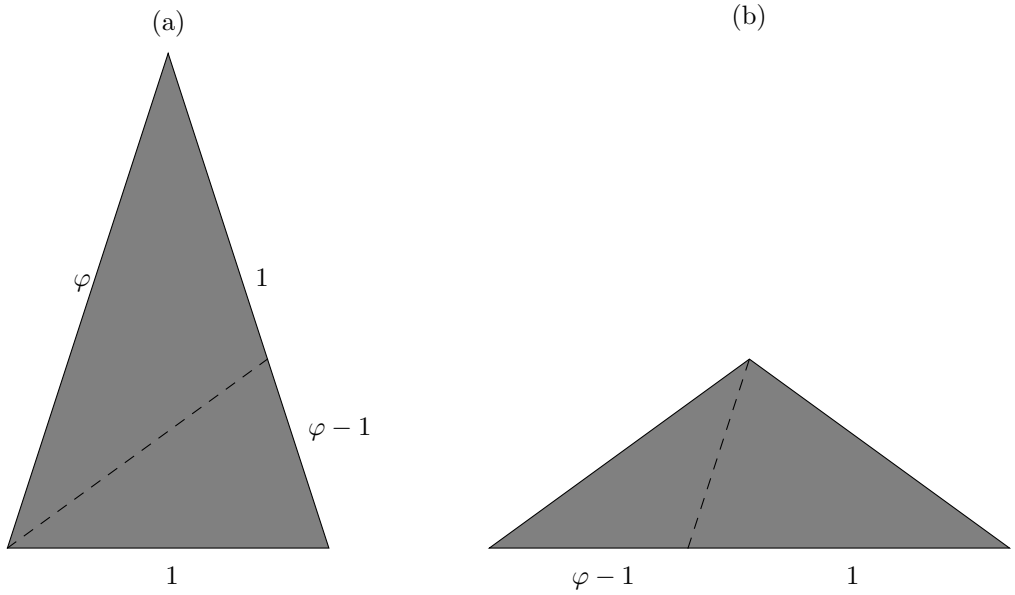


Figure D.4 The golden triangles

A similar argument shows that the $36 - 108 - 36$ can be dissected into a $72 - 36 - 72$ triangle and another $36 - 108 - 36$ triangle. In the language of chapter four, these form a digraph fractal pair.

D.2.3 The pentagon

References

Barnsley, M. 1993. *Fractals Everywhere*. Second edn. Kaufman.

Edgar, G. A. 1993. *Classics on Fractals*. Perseus Books.

Edgar, G. A. 1998. *Integral, Probability, and Fractal Measures*. Springer-Verlag.

Edgar, G. A. 2009. *Measure Topology and Fractal Geometry*. Second edn. Springer-Verlag.

Falconer, K. 1997. *Techiques in Fractal Geometry*. Wiley.

Falconer, K. 2003. *Fractal Geometry*. Second edn. Wiley.

Gutierrez, J., Iglesias, A., Rodriguez, M., and Rodriguez, V. 1997. Generating and rendering fractals images. *Fractals*, **7**.

Hutchinson, J. 1981. Fractals and self similarity. *Indiana University Mathematics Journal*, **30**, 713–747.

Lalley, S. P. 1988. The packing and covering functions of some self-similar fractals. *Indiana Univ. Math. J.*, **37**, 699–710.

Mandelbrot, M. 1982. *The Fractal Geometry of Nature*. Freeman.

Markowsky, G. 1992. Misconceptions about the Golden Ratio. *The College Mathematics Journal*, **23**, 2–19.

Schief, A. 1996. Self-similar sets in complete metric spaces. *Proc. Amer. Math. Soc.*, **124**, 481–490.

Zhang, X., Hitt, R., Wang, B., , and J., Ding. 2008. Sierpinski Pedal Triangles. *Fractals*, **30**, 141–150.