$$C\epsilon^{-D} = C\left(\frac{\epsilon}{r_1}\right)^{-D} + \cdots + C\left(\frac{\epsilon}{r_m}\right)^{-D}.$$

We can simplify $C\epsilon^{-D}$, leaving us with

$$1 = \frac{1}{r_1^{-D}} + \cdots + \frac{1}{r_m^{-D}} = r_1^D + \cdots + r_m^D.$$

□

Example 11.24 *For the Sierpiński triangle we have that $r = 1/2$ and $m = 3$. Thus the theorem gives us a direct way to calculate its dimension as $\frac{\ln 3}{\ln 2} \approx 1.58496$, the same value obtained by directly counting covering squares as shown in Example 11.22.*

Calculating $D(A)$ when the r_i are not all equal and satisfy equation (11.11). Even if it is not simple to give a completely rigorous proof, an inspection of several examples convinces us that the condition of equation (11.11) is often satisfied by totally disconnected iterated function systems. Equation (11.12) cannot be solved exactly, but we can use numerical methods. To begin with, we know that the dimension lies in the range $[0, 2]$. The function

$$f(D) = r_1^D + \cdots + r_m^D - 1$$

is strictly decreasing on $[0, 2]$, since

$$f'(D) = r_1^D \ln r_1 + \cdots + r_m^D \ln r_m < 0.$$

Indeed, the condition $r_i < 1$ implies that $\ln r_i < 0$. Moreover, $f(0) = m - 1 > 1$ and $f(2) = r_1^2 + \cdots + r_m^2 - 1 < 0$ by (11.11). Thus by the intermediate value theorem the function $f(D)$ must have a unique root in $[0, 2]$. We may graph this function or use any numerical root-finding procedure (such as Newton's method) to find the solution to the desired accuracy.

Example 11.25 *Consider a totally disconnected iterated function system $\{T_1, T_2, T_3\}$ with contraction factors $r_1 = 0.5$, $r_2 = 0.4$, and $r_3 = 0.7$. Figure 11.8(a) shows the graph of the function*

$$f(D) = 0.5^D + 0.4^D + 0.7^D - 1$$

for $D \in [0, 2]$. Figure 11.8(b) shows the same function for $D \in [1.75, 1.85]$, allowing us to evaluate the root with higher precision. Inspection shows that $D(A) \approx 1.81$.

11.7 Photographs as Attractors to Iterated Function Systems?

Everything we have seen up until now is elegant from a theoretical point of view, but it does not really help us compress images. We have seen that iterated function systems

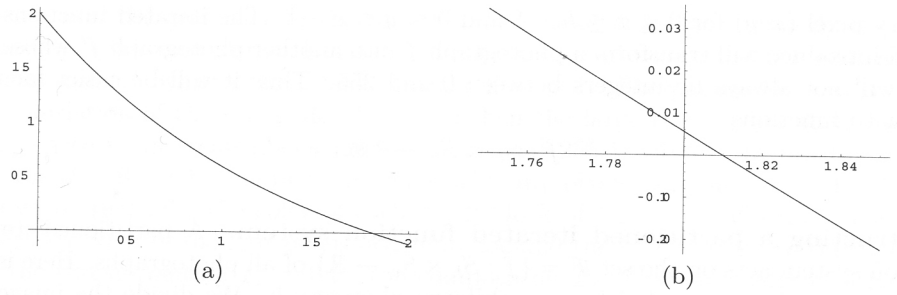


Fig. 11.8. The graph of $f(D)$ for $D \in [0, 2]$ and $D \in [1.75, 1.85]$ for Example 11.25.

allow us to store in memory a fractal image with a very short program. However, to take advantage of this powerful compression we must be able to recognize portions of an image that exhibit strong self-similarity and write short programs constructing them. Are all the parts of an image describable in such a fractal manner? Probably not! Even if a human is able to approximate certain photographs using carefully crafted iterated function system (there are some nice examples in [1]), this is far from providing a systematic algorithm that can operate on hundreds of photographs. If we wish to apply iterated function systems to image compression, we must broaden the ideas we have developed in this chapter.

The concepts of this chapter will thus be applied slightly differently. The common point is that we will still be using a specific type of iterated function system (called *partitioned iterated function system*) whose attractor will approximate the image we wish to compress. The following discussion was inspired by [2]. Research is ongoing to find better-performing alternative methods.

Representing an image as the graph of a function. We discretize a photograph by considering it as a finite set of squares with varying intensity, called *pixels* (for *picture elements*). We associate each pixel in the photo with a number representing its color. To simplify our discussion we will limit ourselves to grayscale images. Thus each point (x, y) of a rectangular photo is associated with a value z that represents its gray tone. Most digital photographs assign integer values in the range $\{0, \ldots, 255\}$ corresponding to black through white, with 0 representing black and 255 representing white. Thus, a photograph may be viewed as a two-dimensional function. If a photograph contains h pixels horizontally and v vertically and we denote by S_N the set $\{0, 1, 2, \ldots, N-1\}$, then a photograph is a function

$$f : S_h \times S_v \longrightarrow S_{255}.$$

In other words, it is a function that associates a gray tone

$$z = f(x, y) \in \{0, \ldots, 255\}$$

to every pixel (x, y) for $0 \leq x \leq h-1$ and $0 \leq y \leq v-1$. The iterated functions that we will introduce will transform a photograph f into another photograph f' whose gray tones will not always be integers between 0 and 255. Thus it will be easier for us to work with functions

$$f : S_h \times S_v \longrightarrow \mathbb{R}.$$

Constructing a partitioned iterated function system. A partitioned iterated function system acts on the set $\mathcal{F} = \{f : S_h \times S_v \to \mathbb{R}\}$ of all photographs. Here is how such a system is constructed for an arbitrary photograph. We divide the image into disjoint neighboring tiles of 4×4 pixels. Each such tile C_i is called a *small tile*, and I is the set of all small tiles. We also consider the set of all possible 8×8 tiles, called big tiles. Each small tile C_i is associated with the big tile G_i that "resembles" it the most (see Figure 11.9). (We will precisely define what we mean by "resemble" a little later.)

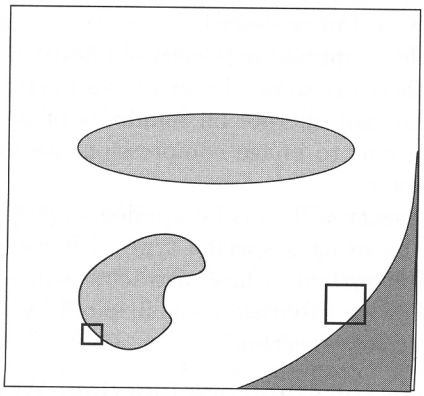


Fig. 11.9. Choosing a big tile that resembles a small tile.

Each point in the image is represented by its coordinates (x, y, z), where z is the gray tone of the pixel at (x, y). An affine transformation T_i will be chosen that maps a big tile G_i onto a small tile C_i, where T_i has the form

$$T_i \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} \alpha_i \\ \beta_i \\ g_i \end{pmatrix}. \tag{11.15}$$

Restricting ourselves to the integer coordinates (x, y), this transformation is a simple affine contraction

$$t_i(x, y) = (a_i x + b_i y + \alpha_i, c_i x + d_i y + \beta_i). \tag{11.16}$$

Consider now the gray tone of the tile. The parameter s_i serves to modify the spread of the gray tones used in the tile: if $s_i < 1$ then the small tile C_i has less contrast than the large tile G_i, while it has more contrast if $s_i > 1$. The parameter g_i corresponds to a translation of the grayscale. If $g_i < 0$ then the large tile is paler than the small tile and vice versa (remember that 0 is black and 255 is white). In practice, since a large tile ($8 \times 8 = 64$) contains four times as many pixels as a small tile ($4 \times 4 = 16$), we start by replacing the color of each 2×2 block of G_i by a uniform color given by the average color of the four pixels originally located there. We compose this operation with the transformation T_i, calling the composition $\overline{T}_i$. Since the sides of a large tile are mapped to those of a small tile, the parameters of the linear part $\left(\begin{smallmatrix} a_i & b_i \\ c_i & d_i \end{smallmatrix}\right)$ of the transformation T_i are greatly limited. In fact, the linear portion of the transformation will be the composition of the homothety of scale $1/2$,

$$(x, y) \mapsto (x/2, y/2),$$

and one of the eight following transformations:

1. the identity transform with matrix $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$;
2. rotation by $\pi/2$ with matrix $\left(\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right)$;
3. rotation by π with matrix $\left(\begin{smallmatrix} -1 & 0 \\ 0 & -1 \end{smallmatrix}\right)$, also called symmetry with respect to the origin;
4. rotation by $-\pi/2$ with matrix $\left(\begin{smallmatrix} 0 & 1 \\ -1 & 0 \end{smallmatrix}\right)$;
5. reflection about the horizontal axis with matrix $\left(\begin{smallmatrix} 1 & 0 \\ 0 & -1 \end{smallmatrix}\right)$;
6. reflection about the vertical axis with matrix $\left(\begin{smallmatrix} -1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$;
7. reflection about the first diagonal axis with matrix $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$;
8. reflection about the second diagonal axis with matrix $\left(\begin{smallmatrix} 0 & -1 \\ -1 & 0 \end{smallmatrix}\right)$.

Note that all of the matrices associated with these linear transformations are orthogonal. (Exercise: which of the above transformations will be used in mapping the big tile to the small tile in Figure 11.9?)

To decide whether two tiles resemble each other we will define a distance function d. The partitioned iterated function system we construct will produce iterates approaching a limit with respect to this same distance as applied to the set $\mathcal{F}$ of all photographs. If $f, f' \in \mathcal{F}$, that is, both f and f' are digitized images of the same size, then the distance between them is defined as

$$d_{h\times v}(f, f') = \sqrt{\sum_{x=0}^{h-1} \sum_{y=0}^{v-1} (f(x, y) - f'(x, y))^2},$$

corresponding to the distance d_3 given in (11.8) of Example 11.8. This distance may seem somewhat intimidating when written out, but it is simply the Euclidean distance on the vector space $\mathbb{R}^{h\times v}$. To decide whether a small tile C_i resembles a large tile G_i we define a similar distance between G_i and C_i. In fact, we calculate the distance between f_{C_i} (the image function restricted to the small tile C_i) and $\overline{f}_{C_i} = \overline{T}_i(f_{G_i})$, that is, the image by $\overline{T}_i$ of the photograph f restricted to the large tile G_i. Recall that the

transformation $\overline{T}_i$ is the composition of replacing the gray tones in each 2×2 block by their average, and then applying T_i to map G_i onto C_i. Let H_i be the set of horizontal indices of the pixels in the C_i, and let V_i be the corresponding set of vertical indices. Then

$$d_4(f_{C_i}, \overline{f}_{C_i}) = \sqrt{\sum_{x \in H_i} \sum_{y \in V_i} \left(f_{C_i}(x,y) - \overline{f}_{C_i}(x,y)\right)^2}. \tag{11.17}$$

It is by carefully choosing s_i and g_i that we obtain a partitioned iterated function system that converges with respect to this distance. Let C_i be a small tile. We discuss how to choose the best large tile G_i and the transform T_i between the two. For a given C_i, we repeat the following steps for each potential large square G_j and each of the possible linear transformations L above:

- apply the smoothing transformation replacing 2×2 blocks of G_j by their average;
- apply the transformation L to the 8×8 square, resulting in a 4×4 square whose pixels are functions in the variables s_i and g_i;
- choose s_i and g_i to minimize the distance d_4 between the original and transformed tiles;
- calculate the minimized distance for the chosen s_i and g_i.

We do the above for each G_j and L and keep track of which G_j, L, s_i, and g_i resulted in the smallest distance between C_i and the resulting transformed tile. This will be one of the transformations in the partitioned iterated function system. We then repeat the above steps for each C_i, for each one determining the optimal associated G_i and T_i. If the image contains $h \times v$ pixels, there are $(h \times v)/16$ small tiles. For each of these, the number of large tiles that it must be compared against is enormous! In fact, a large tile is uniquely specified by its upper left corner, for which there are $(h-7) \times (v-7)$ choices. Since this is too large and would result in too slow an algorithm, we artificially limit ourselves to nonoverlapping large tiles, of which there are $(h \times v)/64$. It is thus with this "alphabet" of tiles that we attempt to accurately reconstruct the original image by associating to each small tile C_i a large tile G_i and a transform $\overline{T}_i$. If $h \times v = 640 \times 640$ then we will have to inspect $(\frac{1}{64} h \times v) \times 8 \times (\frac{1}{16} h \times v) \approx 1.3 \times 10^9$ potential transforms. This is still quite a lot! There are other tricks that may be employed to reduce the search space, but despite these optimizations, this method still has a high compression cost.

Method of least squares. This is the method that is employed in the second-to-last step of the above algorithm, which searches for the best values for s_i and g_i. It is likely that you have already seen this technique in a multivariable calculus, linear algebra, or statistics course. We wish to minimize

$$d_4(f_{C_i}, \overline{f}_{C_i}) = \sqrt{\sum_{x \in H_i} \sum_{y \in V_i} \left(f_{C_i}(x,y) - \overline{f}_{C_i}(x,y)\right)^2}. \tag{11.18}$$

Minimizing d_4 is equivalent to minimizing its square d_4^2, which frees us of the square root. So we must derive the expression of $\overline{f}_{C_i}$ as a function of s_i and g_i. Let us look at how we get $\overline{f}_{C_i}$:

- we start by replacing each 2×2 large square of G_i by a uniform square with the mean color;
- we apply the transformation (11.16), which amounts to sending G_i to C_i without any color adjustment;
- we compose with the mapping $(x, y, z) \mapsto (x, y, s_i z + g_i)$, which is just the color adjustment.

The composition of the first two transformations produces an image on C_i that is described by a function $\tilde{f}_{C_i}$, and we have

$$\overline{f}_{C_i} = s_i \tilde{f}_{C_i} + g_i. \tag{11.19}$$

To minimize d_4^2 in (11.18) we replace $\overline{f}_{C_i}$ by its expression in (11.19) and we require that the partial derivatives with respect to both s_i and g_i be equal to zero. The vanishing of the derivative with respect to g_i yields

$$\sum_{x \in H_i} \sum_{y \in V_i} f_{C_i}(x, y) = s_i \sum_{x \in H_i} \sum_{y \in V_i} \tilde{f}_{C_i}(x, y) + 16 g_i,$$

which implies that f_{C_i} and $\overline{f}_{C_i}$ have the same average gray tone. Requiring that the partial derivative with respect to s_i also vanish implies (after a few simplifications) that

$$s_i = \frac{\mathrm{Cov}(f_{C_i}, \tilde{f}_{C_i})}{\mathrm{var}(\tilde{f}_{C_i})},$$

where the covariance, $\mathrm{Cov}(f_{C_i}, \tilde{f}_{C_i})$, of f_{C_i} and $\tilde{f}_{C_i}$ is defined as follows:

$$\begin{aligned} \mathrm{Cov}(f_{C_i}, \tilde{f}_{C_i}) &= \frac{1}{16} \sum_{x \in H_i} \sum_{y \in V_i} f_{C_i}(x, y) \tilde{f}_{C_i}(x, y) \\ &\quad - \frac{1}{16^2} \left(\sum_{x \in H_i} \sum_{y \in V_i} f_{C_i}(x, y) \right) \left(\sum_{x \in H_i} \sum_{y \in V_i} \tilde{f}_{C_i}(x, y) \right), \end{aligned}$$

and the variance $\mathrm{var}(\tilde{f}_{C_i})$ is defined as

$$\mathrm{var}(\tilde{f}_{C_i}) = \mathrm{Cov}(\tilde{f}_{C_i}, \tilde{f}_{C_i}).$$

The operator W associated with a partitioned iterated function system $\{T_i\}_{i \in I}$. Given a gray tone image $f \in \mathcal{F}$, $W(f)$ is the image obtained by replacing

the image f_{C_i} of the tile C_i by the transformed image $\overline{f}_{C_i}$ of the associated big tile G_i. This gives us a transformed image $\overline{f} \in \mathcal{F}$ defined by

$$\overline{f}(x,y) = \overline{f}_{C_i}(x,y) \qquad \text{if} \quad (x,y) \in C_i.$$

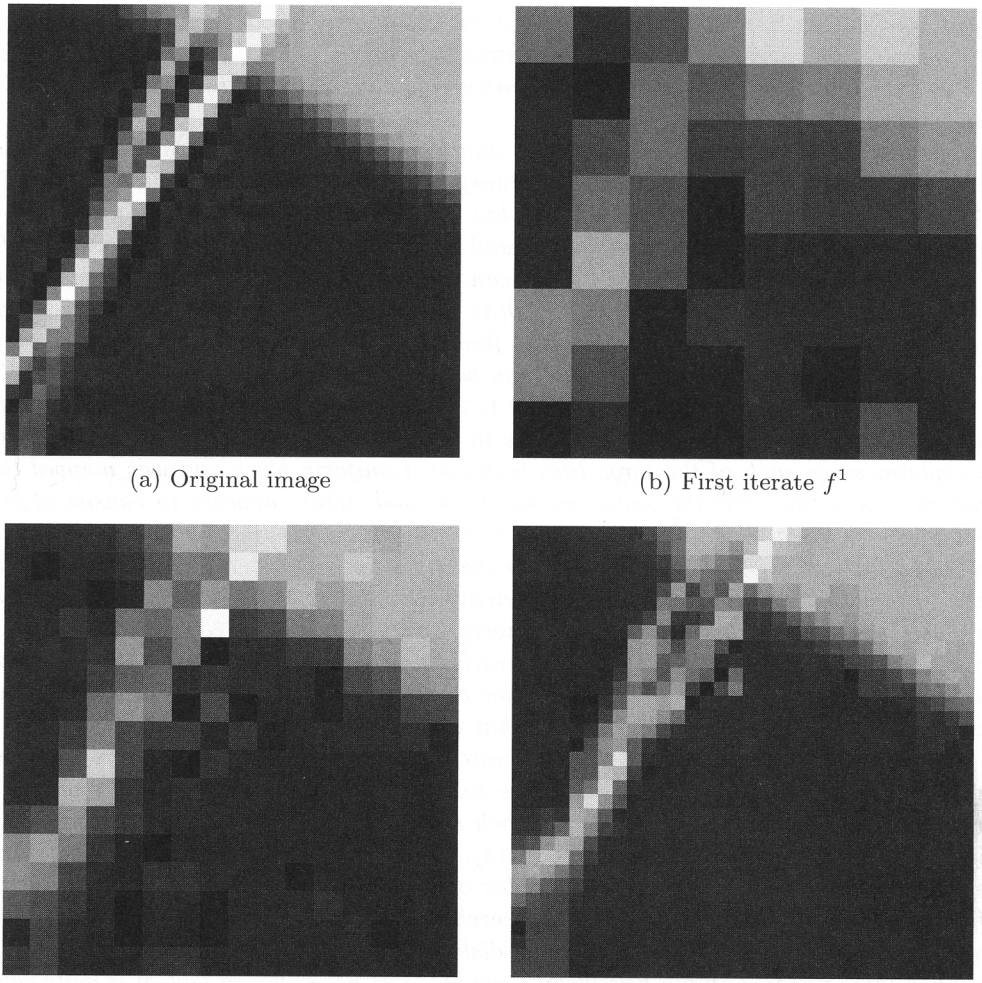
The attractor of this iterated function system should hopefully be something very close to the original image we wished to compress. Thus $W : \mathcal{F} \to \mathcal{F}$ is an operator on the set of all photographs. This technique replaces the *alphabet of geometric objects* we used in our first example with an *alphabet of gray tone tiles*, more specifically the large 8×8 tiles of the photograph to be compressed.

Reconstructing the image. The image can be reconstructed using the following procedure.

- Choose an arbitrary initial function $f^0 \in \mathcal{F}$. A natural choice is the function $f^0(x,y) = 128$ for all x and y, corresponding to a uniformly gray initial image.
- Calculate the iterates $f^j = W(f^{j-1})$. At step $j-1$ the image on each small tile C_i is given by the restriction of f^{j-1} to it. At step j here is how we calculate f^j restricted to C_i: we apply $\overline{T}_i$ to the image given by f^{j-1} on the associated large tile G_i. In practice, we keep track of the distance between successive iterates by calculating $d_{h\times v}(f^j, f^{j-1})$. Once this distance is below a given threshold (the image has largely stabilized), we stop the iteration.
- Replace the real-valued gray tone associated with each pixel by its closest integer value in the range $[0, 255]$.

As it will be shown in the following example, even the iterates f^1 and f^2 give quite good approximations to the original photograph. Furthermore, the distance between successive iterations quickly becomes small, and f^5 is already an excellent approximation to the attractor of the system (and, we hope, of the original image).

Remark: When considered as affine transformations on $\mathbb{R}^3$, the T_i are not always contractions; in fact, T_i is never a contraction if $s_i > 1$! However, most T_i will be contractions, since it is natural to have more contrast across a large tile than across a small one. As far as we know, there is no theorem guaranteeing the convergence of this algorithm for all images. However, in practice we generally see convergence, as if the system $\{T_i\}_{i\in I}$ were in fact a contraction. Benoît Mandelbrot introduced fractal geometry as a way to describe naturally occurring forms, that proved too complicated to be described with traditional geometry. Besides fern leaves and other plants there are many self-similar shapes occurring in nature: rocky coastlines, mountains, river networks, the human capillary system, etc. The technique of compressing images using iterated function systems is particularly well adapted to images having a strong fractal nature, that is, having a strong self-similarity across many scales. For such photos we can generally hope not only for convergence of the resulting system, but for an accurate reproduction of the original image.



(a) Original image

(b) First iterate f^1

(c) Second iterate f^2

(d) Sixth iterate f^6

Fig. 11.10. Reconstructing a 32×32 image (see Example 11.26).

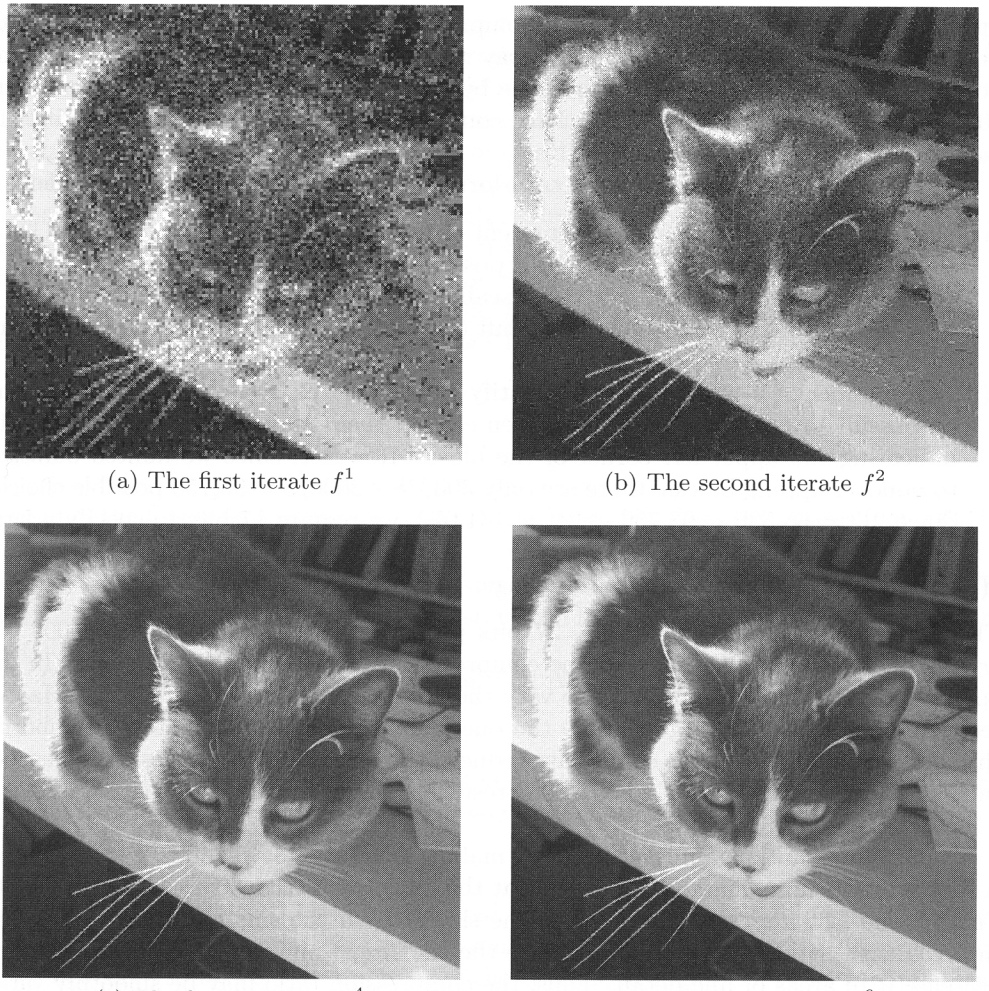
Example 11.26 *An example at last! The above comments may lead one to wonder whether this approach has any chance of accurately reproducing a real photograph. The following example should answer that question. We will use the same photograph used in the discussion of the JPEG image compression standard of Chapter 12, that of Figure 12.1. This photograph contains* $h \times v = 640 \times 640$ *pixels. We will produce two partitioned iterated function systems: the first for reconstructing the* 32×32 *pixel block where two of the cat's whiskers cross (see the zoomed portion of Figure 12.1), and another for the*

entire image. The 32×32 *pixel image is a demanding test of the algorithm. In fact, there are only 16 large tiles to choose from, restricting our chances of finding a good match. We will see, however, that despite this limited "alphabet" the resulting reconstruction is quite accurate!*

For the 32×32 *block there are only* 16 *nonoverlapping* 8×8 *tiles, each of which may be transformed by one of the* 8 *allowed transformations. This creates an "alphabet" of* $16 \times 8 = 128$ *tiles. This is quite limited, but at least it allows the best transformations to be quickly determined. After having found the best tile* G_i *and transformation* T_i *for each of the* $8 \times 8 = 64$ *small tiles* C_i*, we can proceed to the reconstruction. The results are shown in Figure 11.10. Figure 11.10(a) shows the original image to be displayed. For the reconstruction we began with the function* f^0 *associating a constant gray tone of value* 128 *to each of the pixels, halfway between black and white. Figures 11.10(b) through (d) show the reconstruction after* 1, 2, *and* 6 *iterations, respectively. The first surprise is that the first iteration appears to consist of only* 8×8 *pixels. This is easy to explain, since each of the large tiles began as a uniform block and was mapped to a uniform* 4×4 *tile. For the same reason the second iterate appears to consist of only* 16×16 *pixels of width* 2 *each. However, even after only two iterations, the edge of the table and the rough form of the whiskers is clearly visible. The iterates* f^4 *through* f^6 *are very similar to each other, only the last having been shown here. In fact,* f^5 *and* f^6 *are so close that the system is very likely convergent and* f^6 *is quite close to the attractor! In the sixth iterate the two whiskers are nearly completely visible, but with some errors: some pixels are much paler or much darker than in the original image. This is largely due to the limited alphabet of large tiles that we were restricted to working with.*

To obtain the complete partitioned iterated function system of the entire image we made a few concessions. (Recall that the number of individual transformations to be explored is over a billion!) In fact, for each small tile, each large tile, and each of the eight transformations we calculate a pair (s_j, g_j)*. Thus, for each small square we must repeat the calculation eight times the number of large squares. To make this process more efficient we have decide to abandon the search as soon as a large tile* G_i *and associated transform* T_i *are found that are within a distance of* $d_4 = 10$ *to the original small tile. Is this a large distance in the Euclidean space* $\mathbb{R}^{h \times v} = \mathbb{R}^{16}$*? No; in fact, it is quite close! If the distance is* 10*, then the square distance is* 100*. In each small square there are* 16 *pixels; thus we can expect an average squared error of* $\frac{100}{16} \approx 6.3$ *per pixel, corresponding to an expected gray tone error of* $\sqrt{6.3} \approx 2.5$ *per pixel, a relative error of* 1% *on the scale from* 0 *to* 255*. As we will see, the eye is easily able to overlook such a small error. The second compromise we have made is to reject all transformations in which* $|s_i| > 1$*. We have done this to improve the chances that the resulting system is convergent.*

Figure 11.11 presents the first, second, fourth, and sixth iterates of the reconstruction. Again, you can clearly see the 4×4 *uniform blocks in the first iterate and the* 2×2 *uniform blocks in the second iterate. As for* f^4 *and* f^6*, the two are nearly identical and distinguished only by small details. The quality of the sixth iterate is quite good and generally comparable to the original image, the exceptions being areas of fine detail and high contrast, such as the white whiskers against the shadowed background under*



(a) The first iterate f^1 (b) The second iterate f^2

(c) The fourth iterate f^4 (d) The sixth iterate f^6

Fig. 11.11. Reconstructing the entire image of a cat (see Example 11.26).

the table. It should be noted that a majority of the small tiles were approximated by transformations with a distance less than 10 *from the original. However, roughly* 15% *of the tiles were approximated by transformations with a larger error, and the worst offender had a distance of roughly* 280.

Compression ratio. As of 2007, consumer-level digital cameras are commonly available that capture images of up to 8 million pixels (and professional cameras can reach

up to 50 million!). We consider the compression ratio achieved on a 3000×2000 pixel grayscale image with $2^8 = 255$ gray tones. The gray tone of each pixel can be specified using exactly 8 bits, thus one byte,[1] and thus the original image requires $3000 \times 2000 = 6 \times 10^6$ B $= 6$ MB. Now consider the space required to represent the partitioned iterated function system.

Each small tile has an associated transformation T_i and large tile G_i. Consider:

(i) the number of bits necessary to represent a transformation T_i of the form in (11.15):
- 3 bits to specify one of the $2^3 = 8$ possible affine transformations L;
- 8 bits to specify s_i, the gray tone scaling factor; and
- 9 bits to specify g_i, the gray tone shift (we must permit negative values, requiring another bit).

(ii) the number of bits necessary to identify the associated large tile G_i. If we permit all possible overlapping large tiles, then each of them may be uniquely specified by indicating the upper left corner of the block. However, since we limited ourselves to nonoverlapping blocks, there are only $3000/8 \times 2000/8 = 93{,}750$ possible choices. Since $2^{16} = 65{,}536 < 93{,}750 < 2^{17} = 131{,}072$, we require 17 bits to specify a large tile.

(iii) the number of small tiles in the image: $\frac{3000}{4} \times \frac{2000}{4} = 375{,}000$.

Thus, we require $3 + 8 + 9 + 17 = 37$ bits per small tile, yielding $37 \times 375{,}000$ bits or roughly 1.73 MB, yielding that the compression ratio is roughly 3.46 times. In this approach we see that it is possible to vary the number of candidate large tiles. Had we restricted the search of large tiles to the one-fourth of them immediately neighboring the small tile in question, we could have reduced the number of bits necessary to encode each small tile by 2 (from 37 to 35). The resulting compression ratio would improve to a factor of $\frac{37}{35} \times 1.73 \approx 3.66$.

A more substantial gain is achieved by making small tiles 8×8 and large tiles 16×16. A factor of 4 is immediately gained, but at the expense of reconstructed image quality. Finally, one last improvement is to let the size of both the small and large tiles vary. In areas with little detail we can increase the tile size, while we could correspondingly decrease it in areas of fine detail. Thus, the compression ratio may be smoothly varied according to storage needs or desired quality of reconstruction.

Iterated function systems and JPEG. The method described here is very different from that employed by the JPEG standard. Which image compression technique is the best? This depends greatly on the type of images, the desired compression ratio, and the amount of computational power available. As with the improvements discussed above, the compression ratio of JPEG may be smoothly varied (at the expense of image quality) by changing the quantization tables (see Section 12.5). Digital cameras typically store images in the JPEG format, offering the user two or three resolution settings. The degree of compression actually obtained for a given resolution depends on the

[1] One byte equals eight bits and is abbreviated B. One megabyte is 10^6 bytes and is abbreviated MB.

photograph itself (in contrast to the algorithm presented here), but is typically between 6 and 10 times. These are compression ratios that are comparable to those we have just calculated. Compression using iterated function systems has been studied for quite some time but is not used in practice. Its weak point is the amount of time required to compress an image. (Recall that in our earliest discussion of the algorithm the number of steps was proportional to the square of the number of pixels, $(h \times v)^2$. In comparison, the complexity of the JPEG algorithm grows only linearly with image size, and is proportional to $h \times v$. For a photographer in the field snapping photos one after the other, this is a big advantage. For research images being processed on a high-powered computer, it is less so. Regardless, the domain moves quite fast, and iterated function systems may not have spoken their last words.

11.8 Exercises

Certain of the following fractals have been constructed based on the figures found in [1].

1. **(a)** For the fractals of Figure 11.12, find iterated function systems describing them. In each case clearly specify the coordinate system you have chosen. Afterward, reconstruct each of the figures in software.
(b) Given your chosen coordinate system, find two different iterated function systems describing the fractal (b).

2. For the fractals of Figure 11.13, find iterated function systems describing them. In each case clearly specify the coordinate system you have chosen. Afterward, reconstruct each of the figures in software.

3. For the fractals of Figure 11.14, find iterated function systems describing them. In each case clearly specify the coordinate system you have chosen. Afterward, reconstruct each of the figures in software. Attention: here the triangle in Figure 11.14(b) is equilateral, in contrast to the Sierpiński triangle in our earlier example.

4. For the fractals of Figure 11.15, find iterated function systems describing them. In each case clearly specify the coordinate system you have chosen. Afterward, reconstruct each of the figures in software.

5. Amuse yourself by constructing arbitrary iterated function systems and trying to intuit their attractors. Afterward, confirm or disprove your intuitions by plotting them on a computer.

6. Calculate the fractal dimensions of the fractals in Exercises 1 (except (a)), 2, 3, and 4. (In certain cases you will be required to pursue numeric approaches.)

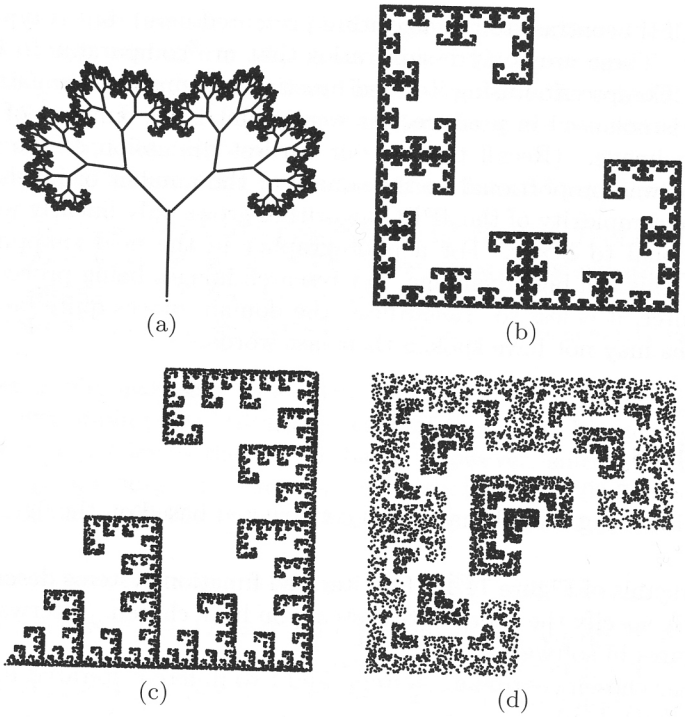


Fig. 11.12. Exercise 1.

7. The Cantor set is a subset of the unit interval $[0, 1]$. It is obtained as the attractor of the iterated function system $\{T_1, T_2\}$, where T_1 and T_2 are the affine contractions defined by $T_1(x) = x/3$ and $T_2(x) = x/3 + 2/3$.

(a) Describe the Cantor set.

(b) Draw the Cantor set. (You may pursue the first few iterations by hand, but it is easiest to use a computer.)

(c) Show that there exists a bijection between the Cantor set and the set of real numbers with base-3 expansions of the form

$$0.a_1a_2 \ldots a_n \ldots,$$

where $a_i \in \{0, 2\}$.

(d) Calculate the fractal dimension of the Cantor set.

8. Show that the fractal dimension of the Cartesian product $A_1 \times A_2$ is the sum of the fractal dimensions of A_1 and A_2:

$$D(A_1 \times A_2) = D(A_1) + D(A_2).$$

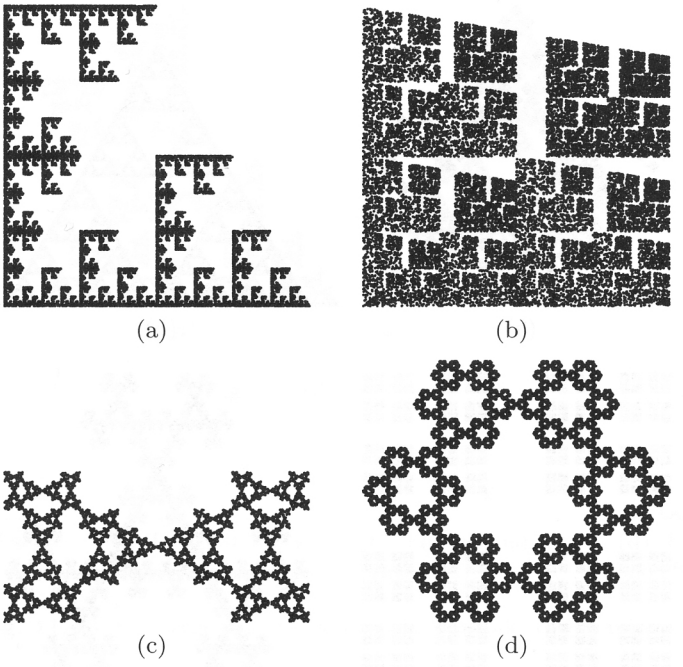


Fig. 11.13. Exercise 2.

9. Let A be the Cantor set, as described in Exercise 7. This is a subset of $\mathbb{R}$. Find an iterated function system on $\mathbb{R}^2$ whose attractor is $A \times A$.

10. The Koch snowflake (or von Koch snowflake) is constructed as the limiting object of the following process (see Figure 11.16):

- Begin with the segment $[0, 1]$.
- Replace the initial segment with four segments, as shown in Figure 11.16(b)).
- Iterate the process, at each step replacing each segment by four smaller segments (see Figure 11.16(c)).

(a) Give an iterated function system that constructs the von Koch snowflake.
(b) Can you give an iterated function system for building the von Koch snowflake that requires just two affine contractions?
(c) Calculate the fractal dimension of the von Koch snowflake.

11. Explain how to modify an iterated function system on $\mathbb{R}^2$

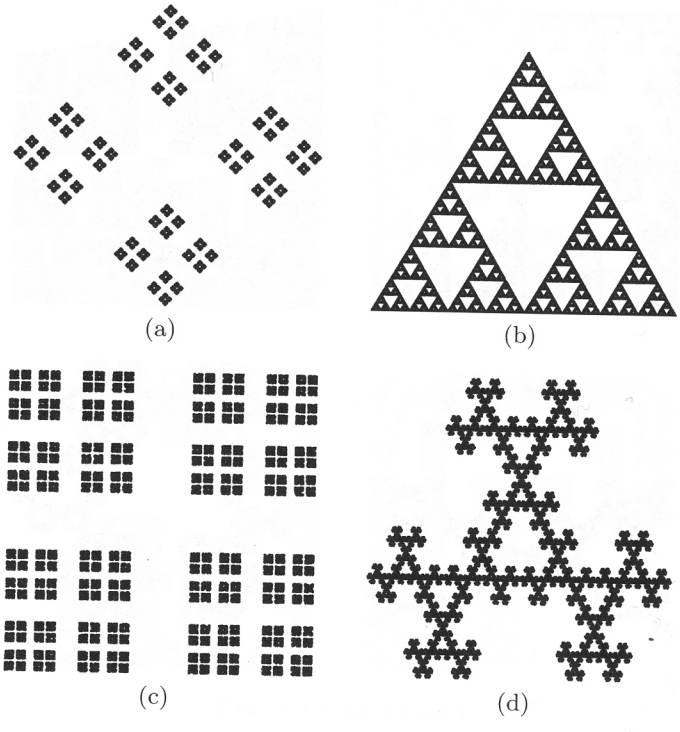


Fig. 11.14. Exercise 3.

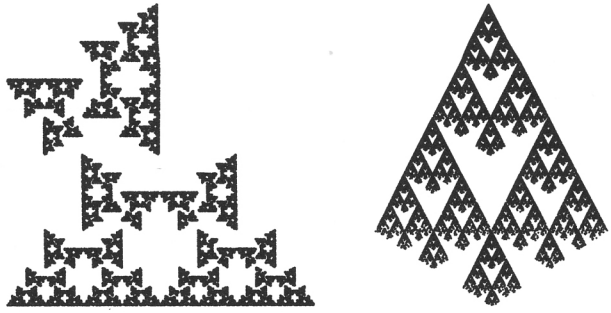
(a) such that its attractor will be twice as large in both dimensions;
(b) to translate the location of its bottom leftmost point.

12. Consider an affine transformation $T(x, y) = (ax + by + e, cx + dy + f)$.
(a) Show that T is an affine contraction if and only if the associated linear transformation $U(x, y) = (ax + by, cx + dy)$ is a contraction.
(b) Show that U contracts distances if

$$\begin{cases} a^2 + c^2 < 1, \\ b^2 + d^2 < 1, \\ a^2 + b^2 + c^2 + d^2 - (ad - bc)^2 < 1. \end{cases}$$

Suggestion: it suffices to show that the square of the length of $U(x, y)$ is less than the square of the length of (x, y) for all nonzero (x, y).

13. Let $P_1, \ldots, P_4$ be four noncoplanar points in $\mathbb{R}^3$. Let $Q_1, \ldots, Q_4$ be four other points of $\mathbb{R}^3$. Show that there exists a unique affine transformation $T : \mathbb{R}^3 \to \mathbb{R}^3$ such that $T(P_i) = Q_i$.



(a) (b)

Fig. 11.15. Exercise 4.

(a) The initial segment (b) The first iteration (c) The second iteration (d) The von Koch snowflake

Fig. 11.16. Constructing the von Koch snowflake of Exercise 10.

Remark: We can consider systems of iterated functions in $\mathbb{R}^3$. As an example, we could use an iterated function system in this space to describe a fern leaf bent under its own weight. We could then project this image to the plane in order to display it.

14. Consider $v \in \mathbb{R}^2$ and A, B, two closed and bounded subsets of $\mathbb{R}^2$. Show that $d(v, A \cup B) \leq d(v, A)$ and $d(v, A \cap B) \geq d(v, A)$.

15. Proceeding numerically, find the contraction factors of the individual transforms T_i for the fern leaf. Are any of these exact contraction factors?

16. **(a)** Let B_1 and B_2 be two disks in $\mathbb{R}^2$ with radius r, and whose centers are at a distance of d from each other. Calculate $d_H(B_1, B_2)$.
(b) Let B_1 and B_2 be two concentric disks in the plane with radii r_1 and r_2, respectively. Calculate $d_H(B_1, B_2)$.