

A Better Way to Compress Images

Mathematics is providing a novel technique for achieving compression ratios of 10,000 to 1—and higher

Michael F. Barnsley and Alan D. Sloan

THE NATURAL WORLD is filled with intricate detail. Consider the geometry on the back of your hand: the pores, the fine lines, and the color variations. A camera can capture that detail and, at your leisure, you can study the photo to see things you never noticed before. Can personal computers be made to carry out similar functions of image storage and analysis? If so, then image compression will certainly play a central role.

The reason is that digitized images—images converted into bits for processing by a computer—demand large amounts of computer memory. For example, a high-detail gray-scale aerial photograph might be blown up to a 3½-foot square and then resolved to 300 by 300 pixels per square inch with 8 significant bits per pixel. Digitization at this level requires 130 megabytes of computer memory—too much for personal computers to handle.

For real-world images such as the aerial photo, current compression techniques can achieve ratios of between 2 to 1 and 10 to 1. By these methods, our photo would still require between 65 and 13 megabytes.

In this article, we describe some of the main ideas behind a new method for image compression using fractals. The method has yielded compression ratios in excess of 10,000 to 1 (bringing our aerial photo down to a manageable 13,000 bytes). The color pictures in figures 1 through 5 were encoded using the new technique; actual storage requirements for these images range from 100 to 2000 bytes.

A mathematics research team at the

Georgia Institute of Technology is developing the system, with funding provided by the Defense Advanced Research Projects Agency (DARPA) and the Georgia Tech Research Corporation (GTRC). Our description is necessarily simplified, but it will show you how a fractal image-compression scheme operates and how to use it to create exciting images.

Describing Natural Objects

Traditional computer graphics encodes images in terms of simple geometrical shapes: points, line segments, boxes, circles, and so on. More advanced systems use three-dimensional elements, such as spheres and cubes, and add color and shading to the description.

Graphics systems founded on traditional geometry are great for creating pictures of man-made objects, such as bricks, wheels, roads, buildings, and cogs. However, they don't work well at all when the problem is to encode a sunset, a tree, a lump of mud, or the intricate structure of a black spleenwort fern. Think about using a standard graphics system to encode a digitized picture of a cloud: You'd have to tell the computer the address and color attribute of each point in the cloud. But that's exactly what an uncompressed digitized image is—a long list of addresses and attributes.

To escape this difficulty, we need a richer library of geometrical shapes. These shapes need to be flexible and controllable so that they can be made to conform to clouds, mosses, feathers, leaves, and faces, not to mention waving sunflowers and glaring arctic wolves. Fractal

geometry provides just such a collection of shapes. For a hint of this, glance at the pictures in *The Fractal Geometry of Nature* by Benoit Mandelbrot, who coined the term *fractal* to describe objects that are very "fractured" (see references for additional books and articles). Some elementary fractal images accompany this article.

Using fractals to simulate landscapes and other natural effects is not new; it has been a primary practical application. For instance, through experimentation, you find that a certain fractal generates a pattern similar to tree bark. Later, when you want to render a tree, you put the tree-bark fractal to work.

What is new is the ability to start with an actual image and find the fractals that will imitate it to any desired degree of accuracy. Since our method includes a compact way of representing these fractals, we end up with a highly compressed data set for reconstructing the original image.

Overview of Fractal Compression

We start with a digitized image. Using image-processing techniques such as color separation, edge detection, spectrum analysis, and texture-variation analysis, we break up the image into segments. (Some of the same techniques

continued

Michael F. Barnsley and Alan D. Sloan are professors of mathematics at the Georgia Institute of Technology (Atlanta, GA 30332) and officers of Iterated Systems Inc. (1266 Holly Lane NE, Atlanta, GA 30329).

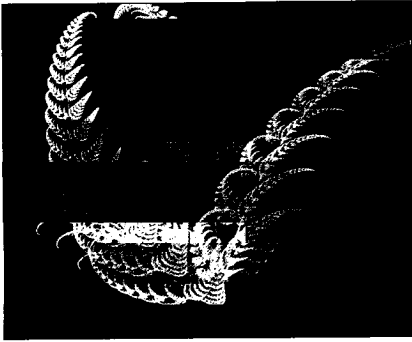


Figure 1: IFS-encoded color image of three-dimensional ferns (4 transformations, 100 bytes).



Figure 2: IFS-encoded color photo of Black Forest, color set adjusted to give winter tones (120 transformations, 2000 bytes).



Figure 3: IFS-encoded color photo of a Bolivian girl (120 transformations, 2000 bytes).

form the basis for the automatic coloring of black-and-white motion pictures.) A segment might be a fern, a leaf, a cloud, or a fence post. A segment can also be a more complex collection of pixels: A seascape, for example, may include spray, rock, and mist.

We then look up these segments in a library of fractals. The library doesn't contain literal fractals; that would require astronomical amounts of storage. Instead, our library contains relatively compact sets of numbers, called *iterated function system* (IFS) codes, that will reproduce the corresponding fractals. Furthermore, the library's cataloging system is such that images that look alike are close together: Nearby codes correspond to nearby fractals. This makes it feasible to set up automated procedures for searching the library to find fractals that approximate a given target image. A mathematical result known as the Collage Theorem (more on that later) guarantees that we can always find a suitable IFS code—and gives a method for doing so.

Once we have looked up all the segments in our library and found their IFS codes, we can throw away the original digitized image and keep the codes, achieving our compression ratio of 10,000 to 1—or even higher.

Iterated Function Systems

We start by explaining how a set of IFS codes can approximate a natural image.

IFS theory is an extension of classical geometry. It uses affine transformations, explained below, to express relations between parts of an image. Using only these relations, it defines and conveys intricate pictures. With IFS theory, we can describe a cloud as clearly as an architect can describe a house.

By studying the following sections,

you should be able to encode and decode fascinating black-and-white image segments, such as leaf skeletons, tree shadows, spirals, and thunderheads. You should also obtain an overview of how a fully automated fractal compression system operates.

Affine transformations can be described as combinations of rotations, scalings, and translations of the coordinate axes in *n*-dimensional space. An example in two dimensions is

$$W(x,y) = (\frac{1}{2}x + \frac{1}{4}y + 1, \frac{1}{4}x + \frac{1}{2}y + 2),$$

which can also be written in matrix form as

$$W \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} .5 & .25 \\ .25 & .5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} .$$

This transformation moves the point (0,0) to (1,2) and moves (-1,0.5) to (0.625, 2). To confirm your understanding of the idea, you should work out where it moves the point (1,1). We denote this transformation by *W*; the notation *W(S)* denotes the subimage of *W* on a set of points *S*.

Now let's see what *W* does to a picture of a smiling face, *F*, lying on the *x,y* plane (see figure 6). The result is a new, squeezed face *W(F)*. The affine transformation has deformed and moved the face. Notice that the eyes in the transformed face *W(F)* are closer together than they are in *F*. We say that the transformation *W* is contractive: It always moves points closer together.

Another example of a contractive affine transformation is shown in figure 7. This time it acts on a leaf to produce a new, smaller leaf.

The general form for an affine transformation is



Figure 4: IFS-encoded color photo of the Monterey coast (60 transformations, 100 bytes).

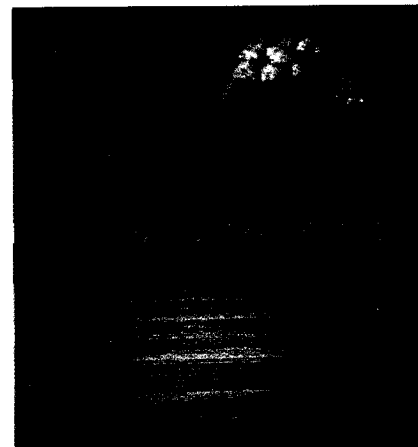


Figure 5: IFS-encoded color image from A Cloud Study (30 transformations, 500 bytes).

$$W \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \\ = \begin{bmatrix} ax+by+e \\ cx+dy+f \end{bmatrix},$$

where the coefficients $a, b, c, d, e,$ and f are real numbers.

If we know in advance the translations, rotations, and scalings that combine to produce W , we can generate coefficient values as follows:

$$a = r \cos \theta, b = -s \sin \phi, \\ c = r \sin \theta, d = s \cos \phi,$$

where r is the scaling factor on x , s is the scaling factor on y , θ is the angle of rotation on x , ϕ is the angle of rotation on y , e is the translation on x , and f is the translation on y .

How can you find an affine transformation that produces a desired effect? Let's show how to find the affine transformation that takes the big leaf to the little leaf in figure 7. We wish to find the numbers $a, b, c, d, e,$ and f for which the transformation W has the property

$$W(\text{big leaf}) \approx \text{little leaf}.$$

Begin by introducing x and y coordinate axes, as already shown in the figure. Mark three points on the big leaf (we've chosen the leaf tip, a side spike, and the point where the stem joins the leaf) and determine their coordinates $(\alpha_1, \alpha_2), (\beta_1, \beta_2),$ and (γ_1, γ_2) . Mark the corresponding points on the little leaf and determine their coordinates $(\tilde{\alpha}_1, \tilde{\alpha}_2), (\tilde{\beta}_1, \tilde{\beta}_2),$ and $(\tilde{\gamma}_1, \tilde{\gamma}_2)$, respectively.

Determine values for the coefficients $a, b,$ and e by solving the three linear equations

$$\alpha_1 a + \alpha_2 b + e = \tilde{\alpha}_1, \quad (1) \\ \beta_1 a + \beta_2 b + e = \tilde{\beta}_1, \quad (2) \\ \gamma_1 a + \gamma_2 b + e = \tilde{\gamma}_1, \quad (3)$$

and find $c, d,$ and f in similar fashion from these equations:

$$\alpha_1 c + \alpha_2 d + f = \tilde{\alpha}_2, \quad (4) \\ \beta_1 c + \beta_2 d + f = \tilde{\beta}_2, \quad (5) \\ \gamma_1 c + \gamma_2 d + f = \tilde{\gamma}_2. \quad (6)$$

We recommend the use of an equation solver such as TK Solver Plus (Universal Technical Systems, Rockford, Illinois) or Eureka (Borland International, Scotts Valley, California) for finding the coefficient values. Doing it manually can be tedious.

Now that we know what a contractive continued

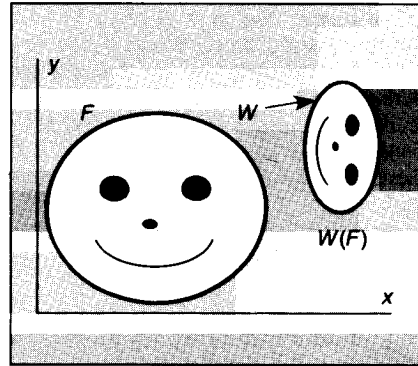


Figure 6: An affine transformation W moves the smiling face F to a new face $W(F)$. The transformation is called contractive because it moves points closer together.

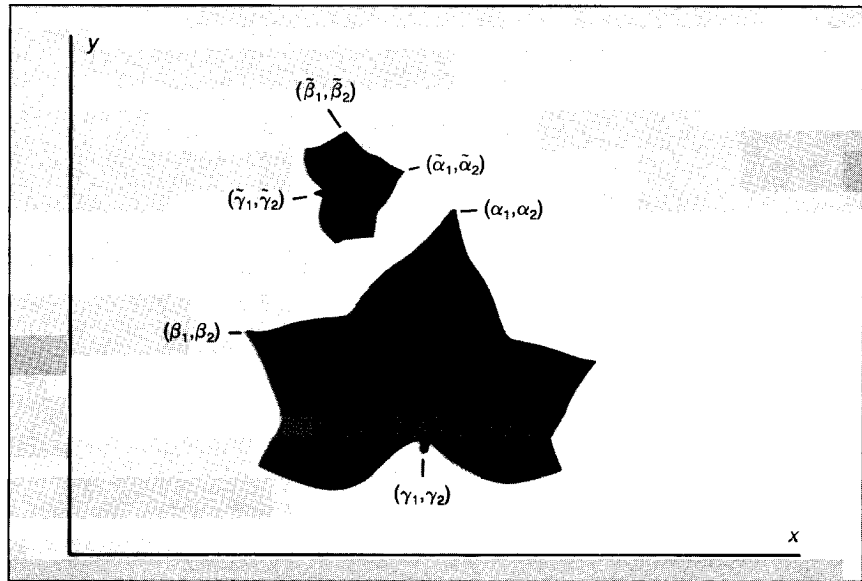


Figure 7: Two ivy leaves fix an affine transformation W .

Table 1: IFS codes for a Sierpiński triangle.

W	a	b	c	d	e	f	p
1	0.5	0	0	0.5	0	0	0.33
2	0.5	0	0	0.5	1	0	0.33
3	0.5	0	0	0.5	0.5	0.5	0.34

Table 3: IFS codes for a fern.

W	a	b	c	d	e	f	p
1	0	0	0	0.16	0	0	0.01
2	0.2	-0.26	0.23	0.22	0	1.6	0.07
3	-0.15	0.28	0.26	0.24	0	0.44	0.07
4	0.85	0.04	-0.04	0.85	0	1.6	0.85

Table 2: IFS codes for a square.

W	a	b	c	d	e	f	p
1	0.5	0	0	0.5	0	0	0.25
2	0.5	0	0	0.5	0.5	0	0.25
3	0.5	0	0	0.5	0	0.5	0.25
4	0.5	0	0	0.5	0.5	0.5	0.25

Table 4: IFS codes for fractal tree.

W	a	b	c	d	e	f	p
1	0	0	0	0.5	0	0	0.05
2	0.1	0	0	0.1	0	0.2	0.15
3	0.42	-0.42	0.42	0.42	0	0.2	0.4
4	0.42	0.42	-0.42	0.42	0	0.2	0.4

affine transformation is and how to find one that maps a source image onto a desired target image, we can describe an iterated function system. An IFS is a collection of contractive affine transformations. Here's an example of an IFS of three transformations:

$$W_1 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0.5 & 0.0 \\ 0.0 & 0.5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$W_2 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0.5 & 0.0 \\ 0.0 & 0.5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

$$W_3 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0.5 & 0.0 \\ 0.0 & 0.5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} .25 \\ .5 \end{bmatrix}.$$

Each transformation must also have an associated probability, p_i , determining its "importance" relative to the other trans-

formations. In the present case we might have p_1, p_2 , and p_3 . Notice that the probabilities must add up to 1. That is, $p_1 + p_2 + p_3 = 1$.

Of course, the above notation for an IFS is cumbersome. Table 1 expresses the same information in tabular form. Other examples of IFS codes are given in tables 2 through 4. Notice that an IFS can contain any number of affine transformations.

The Random Iteration Algorithm

Now let's see how to decode an arbitrary IFS code using the random iteration method. Remember that in general an IFS can contain any number, say m , of affine transformations, $W_1, W_2, W_3, \dots, W_m$, each with an associated probability. The following code summarizes the method:

- (i) Initialize: $x=0, y=0$.
- (ii) For $n=1$ to 2500, do steps (iii)–(vii).
- (iii) Choose k to be one of the numbers 1, 2, ..., m , with probability p_k .
- (iv) Apply the transformation W_k to the point (x,y) to obtain (\tilde{x},\tilde{y}) .
- (v) Set (x,y) equal to the new point: $x=\tilde{x}, y=\tilde{y}$.
- (vi) If $n > 10$, plot (x,y) .
- (vii) Loop.

Applying this procedure to the transformation in table 1 produces the figure shown in figure 8—a fractal known as the Sierpiński triangle. Increasing the number of iterations n adds points to the image. Figure 9 shows the result of the random iteration algorithm applied to the data in table 3, at several stages during the process. By increasing the scale factor used in plotting, you can zoom in on the image (see figure 10). The text box on page 221 contains a BASIC implementation of the method with additional comments on programming.

You may wonder why the first 10 points are not plotted (step (vi)). This is to give the randomly dancing point time to settle down on the image. It is like a soccer ball thrown onto a field of expert players: Until someone gains control of the ball, its motion is unpredictable, or at least is independent of the players' actions. But eventually a player gets the ball, and its motion then becomes a direct result of the skill of the players. The fact that our transformation is contractive guarantees that the "ball" will eventually get to one of the "players," and that it will stay under control after that.

How do we know that the random iter-
continued

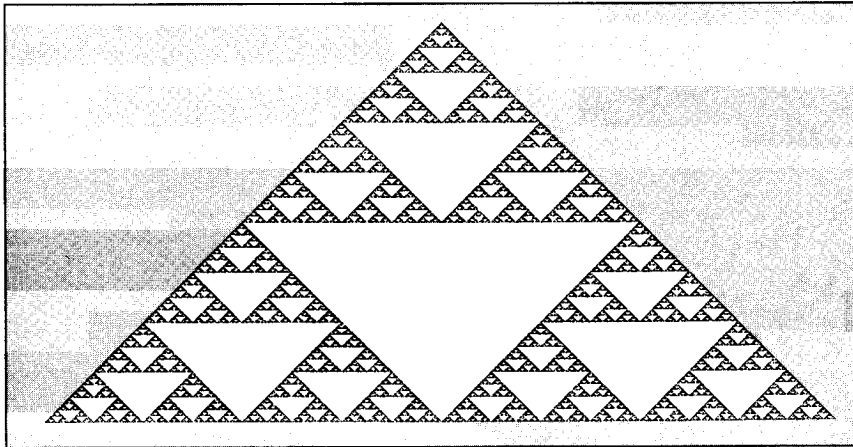


Figure 8: The result of applying the random iteration algorithm to the IFS code in table 1. It is called the Sierpiński triangle.

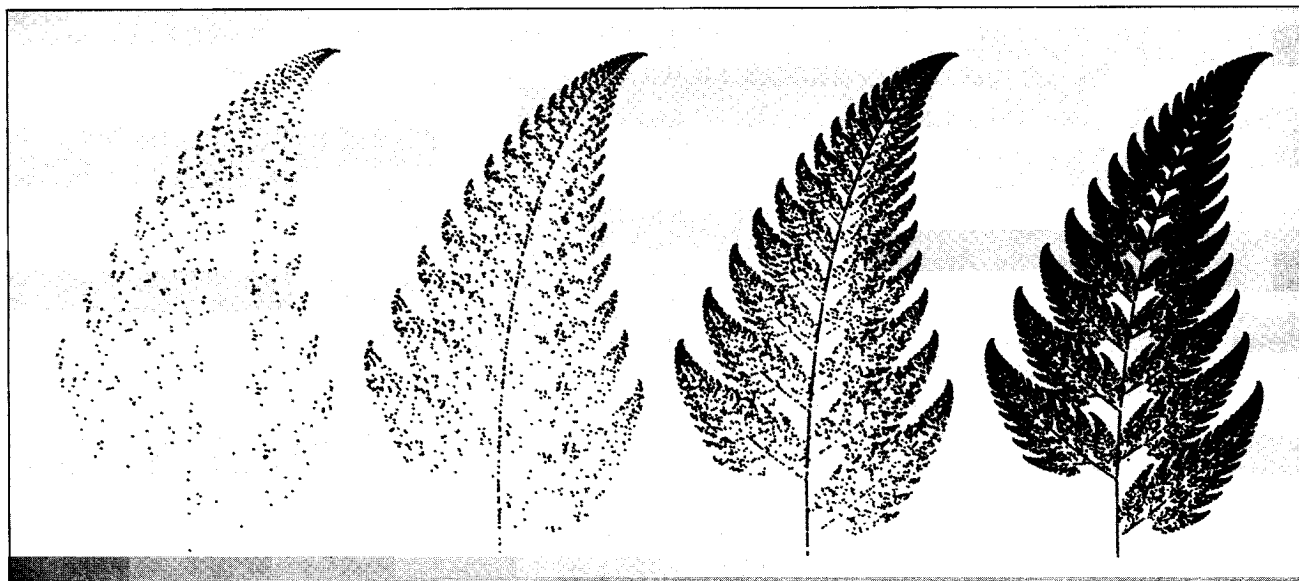


Figure 9: A fern appears when the random iteration algorithm is applied to the IFS code in table 3.

at on algorithm will produce the same image over and over again, independent of the particular sequence of random choices that are made? This remarkable result was first suggested by computer-graphical mathematics experiments and later given a rigorous theoretical foundation by Georgia Tech mathematician John Elton.

The Collage Theorem

Our next goal is to show a systematic method for finding the affine transformations that will produce an IFS encoding of a desired image. This is achieved with the help of the Collage Theorem.

To illustrate the method, we start from a picture of a filled-in square S in the x, y

plane, with its vertices at $(0,0)$, $(1,0)$, $(1,1)$, and $(0,1)$ (see figure 11). The objective is to choose a set of contractive affine transformations, in this case W_1, W_2, W_3, W_4 , so that S is approximated as well as possible by the union of the four sub-images $W_1(S) \cup W_2(S) \cup W_3(S) \cup W_4(S)$. Figure 11 shows, on the left, S together with four noncovering affine transformations of it; on the right, the affine transformations have been adjusted to make the union of the transformed images cover up the square.

To find the coefficients of these transformations, we use the method described earlier in the section on iterated function systems, leading to simultaneous equa-

tions 1 through 3 and 4 through 6. The values one finds in the present case are given in table 2. When the random iteration algorithm is applied to this IFS code, the square is regenerated.

The preceding example typifies the general situation: You need to find a set of affine transformations that shrink distances and that cause the target image to be approximated by the union of the affine transformations of the image. The Collage Theorem says that the more accurately the image is described in this way, the more accurately the transformations provide an IFS encoding of it.

Figure 12 provides another illustration of the Collage Theorem. At the bottom left is shown a polygonalized leaf boundary, together with four affine transformations of that boundary. The transformed leaves taken together do not form a very good approximation of the leaf; in consequence, the corresponding IFS image (bottom right), computed using the random iteration algorithm, does not look much like the original leaf image. However, as the collage is made more accurate (upper left), the decoded image (upper right) becomes more accurate.

So, there's a fundamental stability here. You don't have to get the IFS code exactly right in order to capture a good likeness of your original image. Moreover, the IFS code is robust: Small perturbations in the code will not result in unacceptable damage to the image. In each of the above examples, we have used four transformations to encode the image. However, any number can be used.

For example, the spiral image in figure 13 can be encoded with just two contractive affine transformations. See if you can find them. Then determine the IFS transformation coefficients and input them to the random iteration algorithm to get the spiral back again.

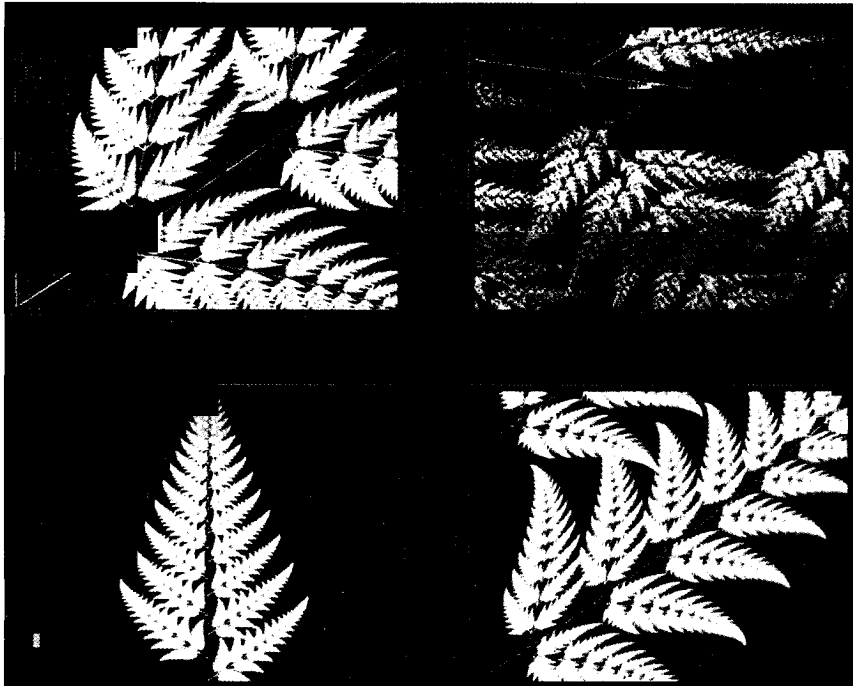


Figure 10: Successive zooms on pieces of an IFS-encoded fern.

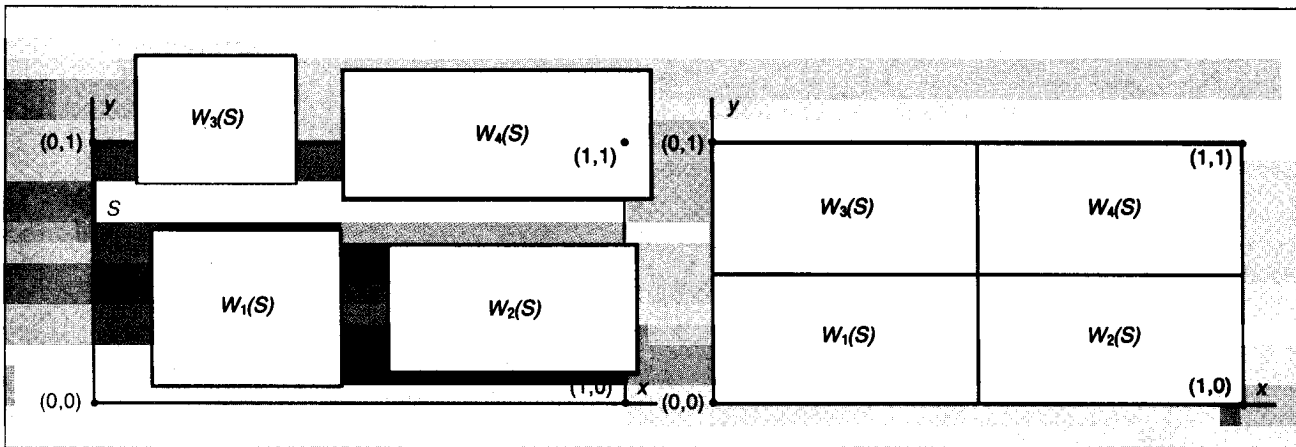


Figure 11: The collage theorem is used to encode a classical square S . The correct IFS code is obtained when the four affine transformations of S cover S , as shown on the right.

Assigning Probabilities

Once you have defined your transformations, you need to assign probabilities to them. Different choices of probabilities do not in general lead to different images, but they do affect the rate at which various regions or attributes of the image are

filled in. Let the affine transformations W_i corresponding to an image I be

$$W_i \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix},$$

where $i = 1, 2, 3, \dots, n$. Then the

amount of time that the randomly dancin g point should spend in the subimage W_i is approximately equal to

$$\frac{\text{area of } W_i}{\text{area of } I}$$

continued

IFS Decoding in BASIC

Listing A is a BASIC implementation of the random iteration algorithm. It includes the data for the Sierpinski triangle, but you can use it to process any IFS tables. In particular, you will want to try the data in tables 2, 3, and 4. Be sure to set the variable m correctly; it tells the program how many transformations are in the IFS.

It is also essential that the probabilities in $p()$ add up to 1. For speed, the transformations should be listed in descending order of probability: the highest probability transformation first, and the lowest probability last.

The program includes variables for rescaling and translating the origin to accommodate the range of the points being plotted to the limits of your screen. If the image is too wide, decrease $xscale$; if the points are too close horizontally, increase $xscale$. Adjust $yscale$ similarly to get a good vertical point spread. To move the image, adjust $xoffset$ and $yoffset$.

You can do these adjustments by trial and error: Run the program; interrupt it and change the offsets and scale factors; and run it again. Or, you can replace the plot command $pset$ with a command to print the values of x and y and run the program to get an exact idea of the range of points being plotted, so you can adjust the scale and offsets more precisely.

Another way to arrange the program is to have it read all the data— m , $a()$, $b()$, $c()$, $d()$, $e()$, $f()$, $p()$, $xscale$, $yscale$, $xoffset$, and $yoffset$ —from a disk file specified by the user. Instead of reading in the coefficients a , b , c , and d , you may want to read in angles θ and ϕ and scale factors r and s , and then calculate the coefficients.

The random iteration method is computation-intensive, so we recommend use of a compiler such as Microsoft's QuickBASIC or Borland's Turbo BASIC. If your computer has a floating-point coprocessor and your compiler supports one, so much the better.

Listing A: A BASIC program demonstrating the use of the random iteration algorithm to reconstruct an IFS-compressed image.

```

10 'Allow for a maximum of 4 transformations in the IFS
20 DIM a(4), b(4), c(4), d(4), e(4), f(4), p(4)
30 '
40 'Transformation data, Sierpinski triangle
50 'First comes the number of transformations
60 'then the coefficients a through f and probability pk
70 'The values for pk should be in descending order.
80 DATA 3
90 DATA .5,0,0,.5,0,0,.34
100 DATA .5,0,0,.5,1,0,.33
110 DATA .5,0,0,.5,.5,.5,.33
120 '
130 'Read in the data
140 READ m
150 pt = 0 'Cumulative probability
160 FOR j = 1 TO m
170   READ a(j), b(j), c(j), d(j), e(j), f(j), pk
180   pt = pt + pk
190   p(j) = pt
200 NEXT j
210 '
220 'Set up for Graphics
230 SCREEN 3 'Select graphics screen
240 xscale = 350 'Map [0,1] onto [0,350]
250 yscale = 325 'Map [0,1] onto [0,325]
260 xoffset = 0
270 yoffset = 0 'Leave the y-origin
280 '
290 'Initialize x and y
300 x = 0
310 y = 0
320 '
330 'Do 2500 iterations
340 FOR n = 1 TO 2500
350   pk = RND
360   'The next line works for m<=4. It must be modified
370   'for values of m > 4.
380   IF pk <= p(1) THEN k = 1 ELSE IF pk <= p(2) THEN k = 2
           ELSE IF pk <= p(3) THEN k = 3 ELSE k = 4
390   newx = a(k) * x + b(k) * y + e(k)
400   newy = c(k) * x + d(k) * y + f(k)
410   x = newx
420   y = newy
430   'Use PRINT x,y instead of the PSET line
440   'to see the range of coordinates. Then fix
450   'xscale, yscale, xoffset, and yoffset
460   IF n > 10 THEN PSET (x * xscale + xoffset, y * yscale
           + yoffset)
470 NEXT n
480 '
490 LOCATE 24, 35
500 PRINT "Press any key to end.";
510 WHILE INKEY$ = ""
520 WEND
530 '
540 'Return to text screen
550 SCREEN 0
560 END

```

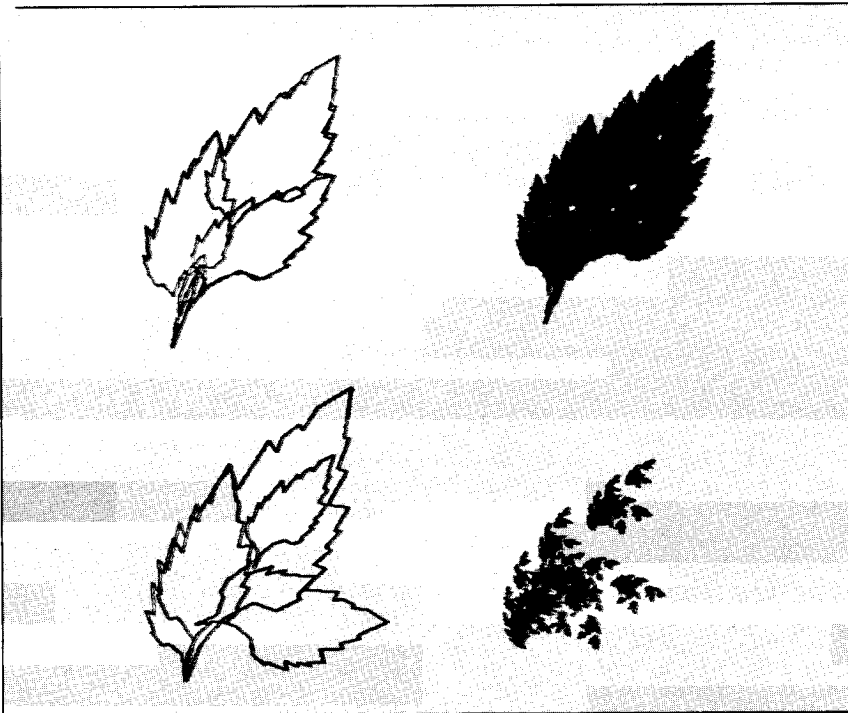


Figure 12: The Collage Theorem is applied to a leaf. The collage at lower left isn't much good, so the corresponding IFS image, shown at lower right, is a poor approximation. But as the collage improves, upper left, so does the IFS image.

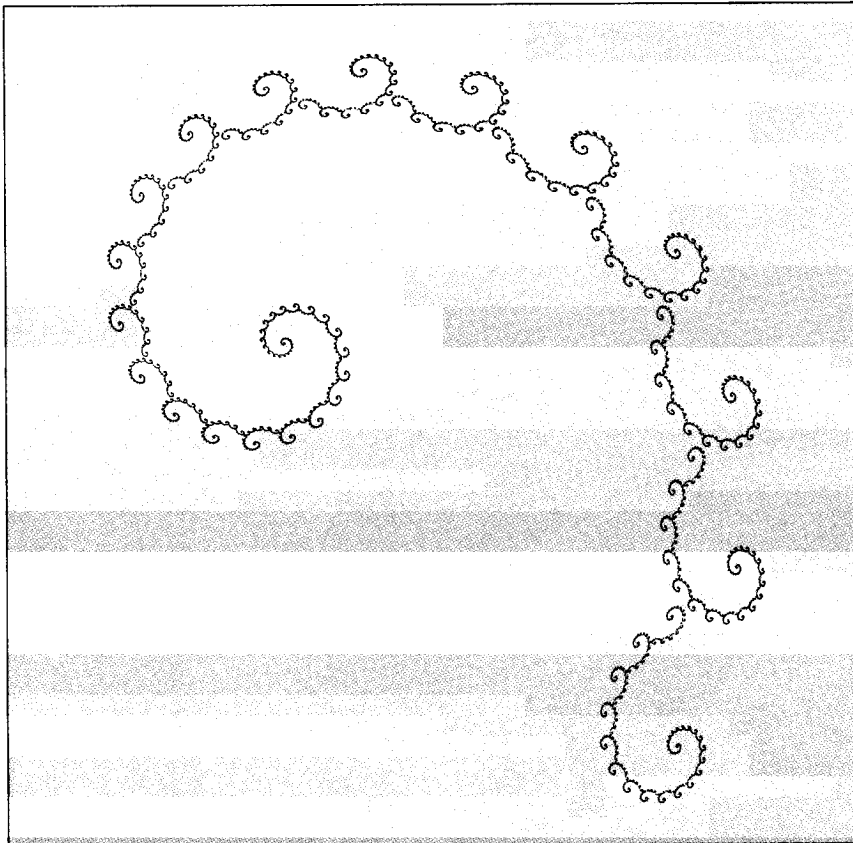


Figure 13: Can you find the IFS codes for this spiral image? Only two transformations are needed.

So long as $ad - cd$ is not 0, it is a standard calculus result that our ratio equals the determinant of the transformation matrix for W_i . So a good choice for the probability p_i is

$$\frac{a_i d_i - b_i c_i}{\sum_{k=1}^n |a_k d_k - b_k c_k|}$$

provided none of these numbers p_i comes out to be 0. A 0 value should be replaced by a very small positive value, such as 0.001, and the other probabilities correspondingly adjusted to keep the sum of all the probabilities equal to 1.

We now summarize the compression and decompression process: An input image is broken up into segments through image-processing techniques. These image components are looked up in the IFS library using the Collage Theorem, and their IFS codes are recorded. When the image is to be reconstructed, the IFS codes are input to the random iteration algorithm. The accuracy of the reconstructed image depends only on the tolerance setting used during the collage mapping stage.

Applications

For graphics applications, we use a more sophisticated procedure that allows full-color images to be encoded. Combinatorial searching algorithms can be used to automate the collage mapping stage. Figures 2, 3, and 4 were obtained using IFS theory at compression ratios in excess of 10,000 to 1. These images were based on photographs in recent issues of *National Geographic*. A full-sequence video animation, *A Cloud Study*, was shown at SIGGRAPH '87. This was encoded at a ratio exceeding 1,000,000 to 1 and can be transmitted in encoded form at video rates over ISDN lines (ISDN stands for integrated services digital network, a concept for integrated voice and data communications). A frame from the animation is shown in figure 5.

The IFS compression technique is computation-intensive in both the encoding and decoding phases. Computations for the color images were all carried out on Masscomp 5600 workstations (dual 68020-based systems) with Aurora graphics. Complex color images require about 100 hours each to encode and 30 minutes to decode on the Masscomp.

For practical applications, you need custom hardware that can speed the encoding and decoding process. An experimental prototype, the IFSIS (iterated function system-image synthesizer), decodes at the rate of several frames per second. The IFSIS device was produced from a cooperative effort between GTRC,

DARPA, Atlantic Aerospace Electronics Corporation, and Iterated Systems, and it was demonstrated on October 5, 1987, at the third annual meeting of the Applied and Computational Mathematics Program of DARPA. It can be connected to a personal computer through a serial port; the personal computer sends the IFS codes to the device, which responds by producing complex color images on a monitor.

The IFSIS is a proof of concept for faster devices with higher resolution. Once the higher-performance IFSIS devices are combined with ISDN telecommunication, full-color animation at video rates over phone lines will be a reality.

Another area for future application of IFS encoding is automatic image analysis. What's in a picture? Does it show a spotted sandpiper or a robin? The more complex the image or the more subtle the question, the harder it becomes for an algorithmic answer to be formulated. But here's the point: Whatever the answer, it will proceed faster if stable, compressed images are used. The reason for this is that image-recognition problems involve combinatorial searching, and searching times increase factorially with the size of the image file.

During the spring of 1987, Iterated Systems was incorporated to develop commercial applications of IFS image compression. It is exciting to see how an abstract field of mathematics research is leading to new technology with implications ranging from commercial and industrial work to personal computing. ■

ACKNOWLEDGMENTS

Figures 2 through 5 were encoded by graduate students François Malassenet, Laurie Reuter, and Arnaud Jacquin. All color images were produced in the Computergraphical Mathematics Laboratory at Georgia Institute of Technology and are copyright 1987, GTRC.

BIBLIOGRAPHY

Barnsley, M. F. and S. Demko. "Iterated Function Systems and the Global Construction of Fractals." *The Proceedings of the Royal Society of London*, A399, 1985, pp. 243-275.

Barnsley, M. F., V. Ervin, D. Hardin, and J. Lancaster. "Solution of an Inverse Problem for Fractals and Other Sets." *Proceedings of the National Academy of Science*, vol. 83, April 1985.

Barnsley, M. F. *Fractals Everywhere*. Academic Press, 1988. Forthcoming.

Elton, J. "An Ergodic Theorem for Iterated Maps." *Journal of Ergodic Theory and Dynamical Systems*. Forthcoming.

Mandelbrot, B. *The Fractal Geometry of Nature*. San Francisco, CA: W. H. Freeman and Co., 1982.

