Fractal Image Compression

SIGGRAPH '92 Course Notes

Yuval Fisher
Visiting the Department of Mathematics
Technion Israel Institute of Technology
from
The San Diego Super Computer Center
University of California, San Diego

With the advance of the information age the need for mass information storage and retrieval grows. The capacity of commercial storage devices, however, has not kept pace with the proliferation of image data. Images are stored on computers as collections of bits (a bit is a binary unit of information which can answer one "yes" or "no" question) representing pixels, or points forming the picture elements. Since the human eye can process large amounts of information, many pixels - some 8 million bits' worth - are required to store even moderate quality images. These bits provide the "yes" or "no" answers to 8 million questions that determine the image, though the questions are not the "is it bigger than a bread-box" variety, but a more mundane "What color is this pixel."

Although the storage cost per bit is currently about half a millionth of a dollar, a family album with several hundred photos can cost over a thousand dollars to store! This is one area in which image compression can play an important role. Storing the images in less memory leads to a direct reduction in cost. Another useful feature of image compression is the rapid transmission of data; less data requires less time to send.

So how can image data be compressed? Most data contains some amount of redundancy, which can sometimes be removed for storage and replaced for recovery, but this redundancy does not lead to high compression. Fortunately, the human eye is not sensitive a wide variety of information loss. That is, the image can be changed in many ways that are either not detectable by the human eye or do not contribute to "degradation" of the image. If these changes are made so that the data becomes highly redundant, then the data can be compressed when the redundancy can be detected. For example, the sequence $2, 0, 0, 2, 0, 2, 2, 2, 0, 0, 2, 0, 2, \ldots$ is similar to $1, 1, 1, 1, 1 \ldots$, but contains random fluctuations of ± 1. If the latter sequence can serve our purpose as well as the first, we are better off storing it, since it can be specified very compactly.

The standard methods of image compression come in several varieties. The current most popular method relies on eliminating high frequency components of the signal by storing only the low frequency Fourier coefficients. Other methods use a "building block" approach, breaking up images into a small number of canonical pieces and storing only a reference to which piece goes where. In this article, we will explore a new scheme based on fractals. Such a scheme has been promoted by M. Barnsley, who founded a company based on fractal image compression technology but who has not released details of his scheme. The first publically available such scheme was due to E. Jacobs and R. Boss of the Naval Ocean Systems Center in San Diego who used regular partitioning and classification of curve segments in order to compress random fractal curves (such as political boundaries) in two dimensions [BJ], [JBF]. A doctoral student of Barnsley's, A. Jacquin, was the first to publish a similar fractal image compression scheme [J]. An improved version of this scheme along with other schemes can be found in work done by the author in [FJB], [JFB], and [FJB1].

We will begin by describing a simple scheme that can generate complex looking fractals from a small amount of information. Then we will generalize this scheme to allow the encoding of an images as "fractals", and finally we will discuss some of the ways this scheme can be implemented.

§1 What is Fractal Image Compression?

Imagine a special type of photocopying machine that reduces the image to be copied by a half and reproduces it three times on the copy. Figure 1 shows this. What happens when we feed the output of this machine back as input? Figure 2 shows several iterations of this process on several input images. What we observe, and what is in fact true, is that all the copies seem to be converging to the same final image, the one in 2(c). We call this image the *attractor* for this copying machine. Because the copying machine reduces the input image, any initial image will be reduced to a point as we repeatedly run the machine. Thus, the initial image placed on the copying machine doesn't effect the final attractor; in fact, it is only the position and the orientation of the copies that determines what the final image will look like.

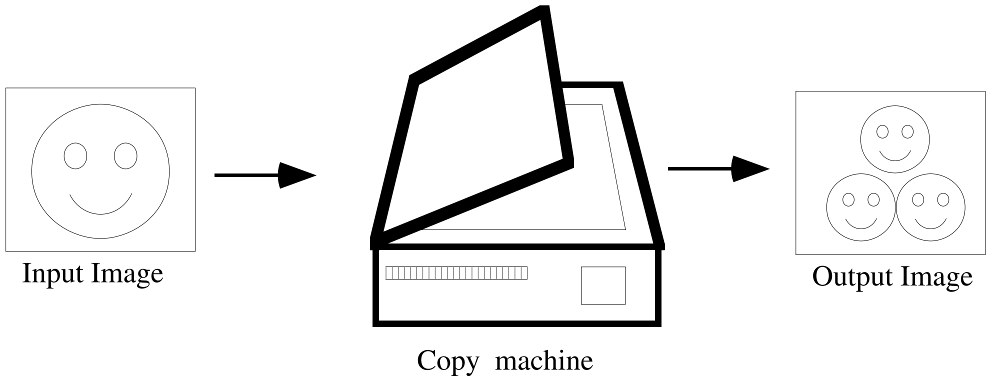

Figure 1. A copy machine that makes three reduced copies of the input image.

Since it is the way the input image is transformed that determines the final result of running the copy machine in a feedback loop, we only describe these transformations. Different transformations lead to different attractors, with the technical limitation that the transformations must be contractive - that is, a given transformation applied to any two points in the input image must bring them closer together in the copy. (See the Contractive Transformations Box). This technical condition is very natural, since if points in the copy were spread out the attractor would have to be of infinite size. Except for this condition, the transformations can have any form. In practice, choosing transformations of the form

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

is sufficient to yield a rich and interesting set of attractors. Such transformations are called *affine transformations* of the plane, and each can skew, stretch, rotate, scale and translate an input image; in particular, affine transformations always map squares to parallelograms.

Figure 3 shows some affine transformations, the resulting attractors, and a zoom on a region of the attractor. The transformations are displayed by showing an initial square marked with an "⊨" and its image by the transformations. The "⊨" helps show when a transformation flips or rotates a square. The first example shows the transformations used in the copy machine of figure 1. These transformations reduce the square to half its size and copy it at three different locations in the same orientation. The second example is very similar to the first, but in it, one transformation flips the square resulting in a different attractor. The last example is the Barnsley fern. It consists of four transformations, one of which is squished flat to yield the stem of the fern.

A common feature of these and all attractors formed this way is that in the position of each of the images of the original square on the left there is a transformed copy of the whole image. Thus, each image is formed from transformed (and reduced) copies of iteslf, and hence it must have detail at every scale. That is, the images are *fractals*. This method of generating fractals is due to John Hutchinson [H], and more information about many ways to generate such fractals can be found in books by Barnsley [B] and Peitgen, Saupe, and Jurgens [P1,P2].

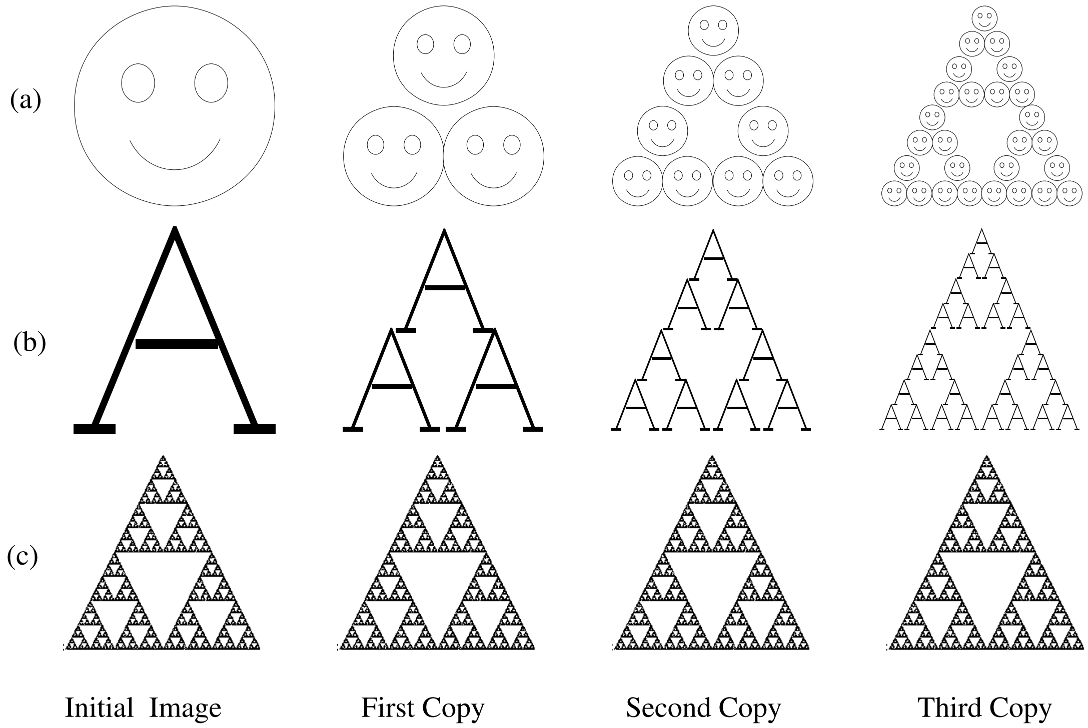

Figure 2. The first three copies generated on the copying machine of figure 1.

Barnsley suggested that perhaps storing images as collections of transformations could lead to image compression. His argument went as follows: the fern in figure 3 looks complicated and intricate, yet it is generated from only 4 affine transforation. Each affine transformation w_i is defined by 6 numbers, a_i, b_i, c_i, d_i, e_i and f_i which do not require much memory to store on a computer (they can be stored in 4 transformations $\times$ 6 numbers/transformation $\times$ 32 bits/number = 768 bits). Storing the image of the fern as a collection of pixels, however, requires much more memory (at least 65,536 bits for the resolution shown in figure 3). So if we wish to store a picture of a fern, then we can do it by storing the numbers that define the affine transformations and simply generate the fern when ever we want to see it. Now suppose that we were given any arbitrary image, say a face. If a small number of affine transformations could generate that face, then it too could be stored compactly. The trick is finding those numbers. The fractal image compression scheme described later is one such trick.

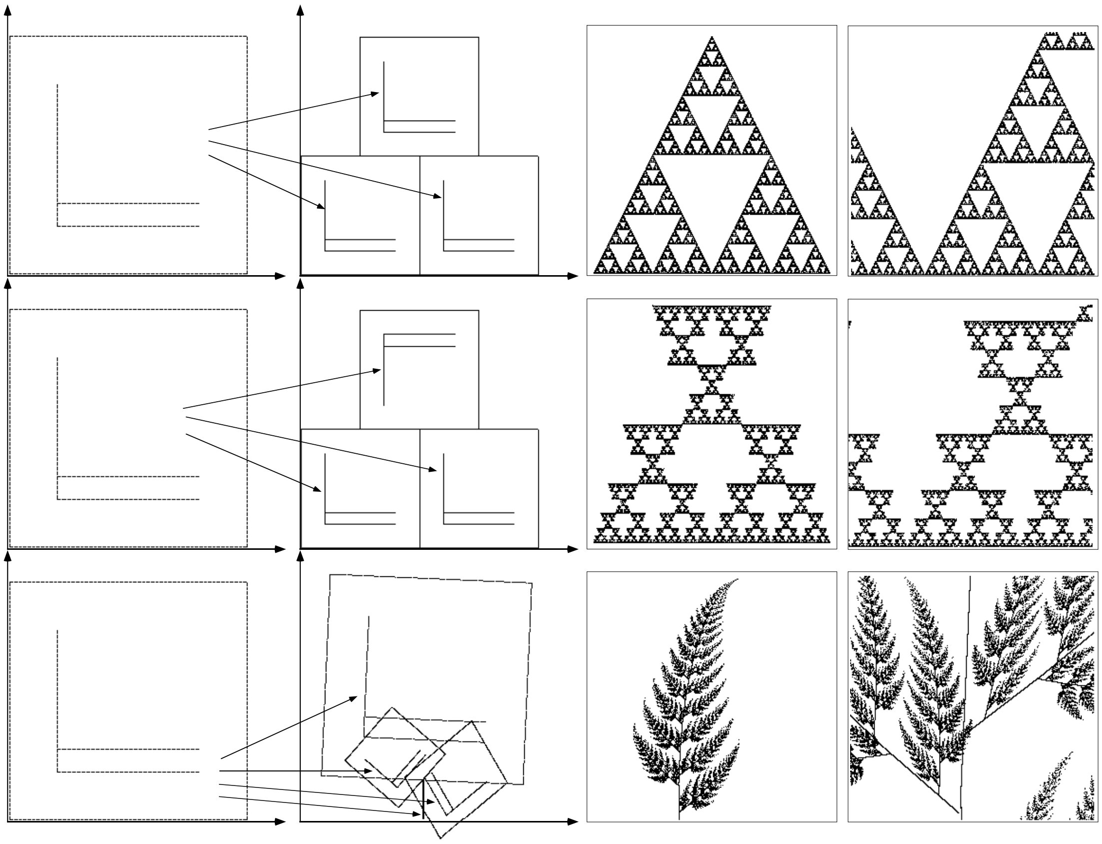

Figure 3. Transformations, their attractor, and a zoom on the attractor.

Why is it "Fractal" Image Compression?

The image compression scheme described later can be said to be fractal in several senses. The scheme will encode an image as a collection of transforms that are very similar to the copy machine metaphor. This has several implications. For example, just as the fern is a set which has detail at every scale, so does the image reconstructed from the transforms have detail created at every scale. Also, if one scales the transformations defining the fern (say by multiplying everything by 2), the resulting attractor will be scaled (also by a factor of 2). In the same way, the decoded image has no natural size, it can be decoded at any size. The extra detail needed for decoding at larger sizes is generated automatically by the encoding transforms. One may wonder (but hopefully not for long) if this detail is "real"; that is, if we decode an image of a person at larger and larger size, will we eventually see skin cells or perhaps atoms? The answer is, of course, no. The detail is not at all related to the actual detail present when the image was digitized; it is just the product of the encoding transforms which only encode the large scale features well. However, in some cases the detail is realistic at low magnifications, and this can be a useful feature of the method. For example, figure 4 shows a detail from a fractal encoding of Lena along with a magnification of the original. The whole original image can be seen in figure 6, the now famous image of Lena which is commonly used in the image compression literature.

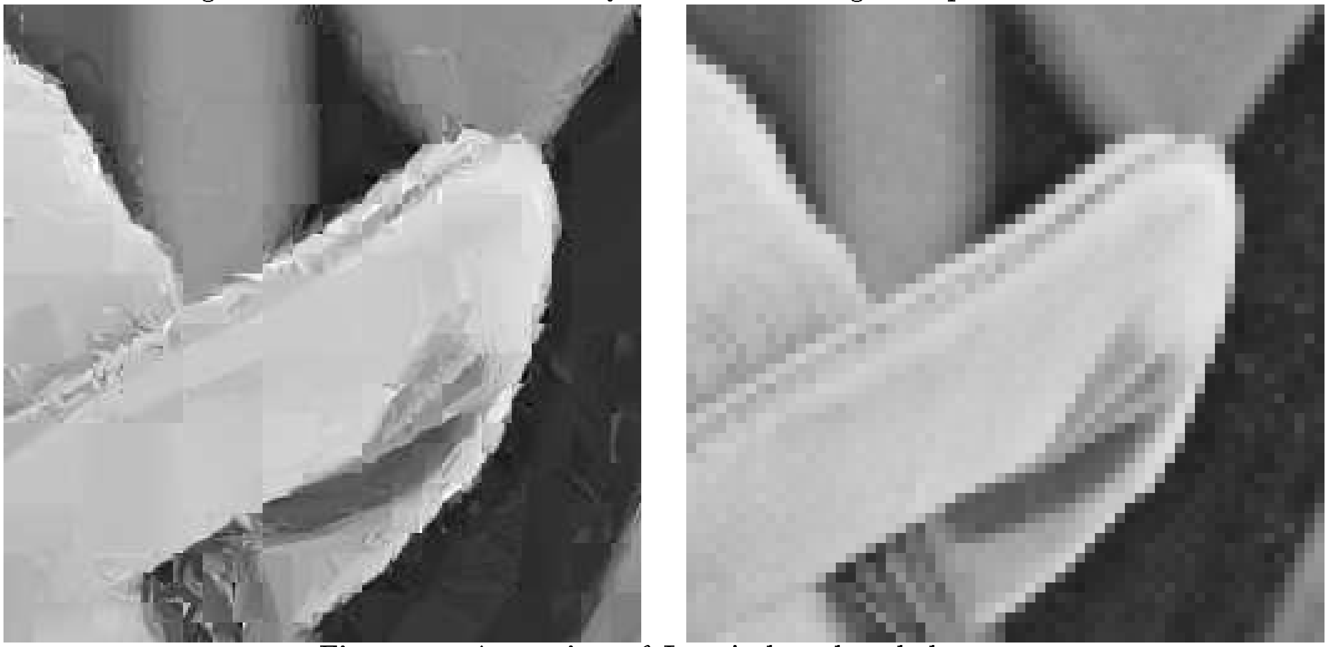

Figure 4. A portion of Lena's hat decoded at 4 times its encoding size (left), and the original image enlarged to 4 times the size (right), showing pixelization.

The magnification of the original shows pixelization, the dots that make up the image are clearly discernible. This is because it is magnified by a factor of 4. The decoded image does not show pixelization since detail is created at all scales.

Why is it Fractal Image "Compression"?

Standard image compression methods can be evaluated using their compression ratio; the ratio of the memory required to store an image as a collection of pixels and the memory required to store a representation of the image in compressed form. As we saw before, the fern could be generated from 768 bits of data but required 65,536 bits to store as a collection of pixels, giving a compression ratio of $65,536/768 = 85.3$ to 1.

The compression ratio for the fractal scheme is hard to measure, since the image can be decoded at any scale. For example, the decoded image in figure 4 is a portion of a 5.7 to 1 compression of the whole Lena image. It is decoded at 4 times it's original size, so the full decoded image contains 16 times as many pixels and hence its compression ratio is 91.2 to 1. This may seem like cheating, but since the 4-times-larger image has detail at every scale, it really isn't.

CONTRACTIVE TRANSFORMATIONS

A transformation w is said to be *contractive* if for any two points P_1, P_2, the distance

$$d(w(P_1), w(P_2)) < sd(P_1, P_2)$$

for some $s < 1$. This formula says the application of a contractive map always brings points closer together (by some factor less than 1). This definition is completely general, applying to any space on which we can define a distance function $d(P_1, P_2)$. In our case, we work in the plane, so that if the points have coordinates $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then

$$d(P_1, P_1) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

An example of a contractive transformation of the plane is

$$w \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

which halves the distance between any two points.

Contractive transformations have the nice property that when they are repeatedly applied, they converge to a point which remains fixed upon further iteration (See the Contractive Mapping Fixed Point Theorem box). For example, the map w above applied to any initial point (x, y) will yield the sequence of points $(\frac{1}{2}x\frac{1}{2}y), (\frac{1}{4}x, \frac{1}{4}y), \ldots$ which can be seen to converge to the point $(0, 0)$ which remains fixed.

Iterated Function Systems.

Before we proceed with the image compression scheme, we will discuss the copy machine example with some notation. Later we will use the same notation for the image compression scheme, but for now it is easier to understand in the context of the copy machine example.

Running the special copy machine in a feedback loop is a metaphor for a mathematical model called an iterated function system (IFS). An iterated function system consists of a collection of contractive transformations $\{w_i : \gtreqless^2 \rightarrow R^2 \mid i = 1, \ldots, n\}$ which map the plane R^2 to itself. This collection of transformations defines a map

$$W(\cdot) = \bigcup_{i=1}^{n} w_i(\cdot).$$

The map W is not applied to the plane, it is applied to sets - that is, collections of points in the plane. Given an input set S, we can compute $w_i(S)$ for each i, take the union of these sets, and get a new set $W(S)$. So W is a map on the space of subsets of the plane. We will call a subset of the plane an image, because the set defines an image when the points in the set are drawn in black, and because later we will want to use the same notation on graphs of functions which will represent actual images. An important fact proved by Hutchinson is that when the w_i are contractive in the plane, then W is contractive in a space of (closed and bounded) subsets of the plane. (The "closed and

The Contractive Mapping Fixed Point Theorem

The contractive mapping fixed point theorem says that something that is intuitively obvious: if a map is contractive then when we apply it repeatedly starting with any initial point we converge to a unique fixed point. For example, the map $\omega(x) = \frac{1}{2}x$ on the real line is contractive for the normal metric $d(x, y) = |x - y|$, because the distance between $\omega(x)$ and $\omega(y)$ is half the distance between x and y. Furthermore, if we iterate ω from any initial point x, we get a sequence of points $\frac{1}{2}x, \frac{1}{4}x, frac18x, \ldots$ that converges to the fixed point 0.

This simple sounding theorem tells us when we can expect a collection of transformations to define image. Let's write it precisely and examine it carefully.

The Contractive Mapping Fixed Point Theorem. *If X is a complete metric space and $W : X \to X$ is contractive, then W has a unique fixed point $|W|$.*

What do these terms mean ? A *complete* metric space is a "gap-less" space on which we can measure the distance between any two points. For example, the real line is a complete metric space with distance between any two points x and y given by $|x - y|$. The set of all fractions of integers, however, is not complete. We can measure the distance between two fractions in the same way, but between any two elements of the space we find a real number (that is, a "gap") which is not a fraction and hence is not in the space. Returning to our example, the map ω can operate on the space of fractions, however the map $x \mapsto \frac{1}{\pi}x$ cannot. This map is contractive, but after one application of the map we are no longer in the same space we began in. This is one problem that can occur when we don't work in a complete metric space. Another problem is that we can find a sequence of points that do not converge to a point in the space; for example, there are sequences of fractions that get closer and closer (in fact, arbitrarily close) to $\sqrt(2)$ which is not a fraction.

A *fixed point* $|W| \in X$ of W is a point that satisfies $W(|W|) = |W|$. Our mapping $\omega(x) = \frac{1}{2}x$ on the real line has a unique fixed point 0 because $\omega(0) = 0$.

Proving the theorem is as easy as finding the fixed point: Start with an arbitrary point $x \in X$. Now iterate W to get a sequence of points $x, W(x), W(W(x), \ldots$ How far can we get at each step ? Well, the distance between $W(x)$ and $W(W(x))$ is less by some factor $s < 1$ than the distance between x and $W(x)$. So at each step the distance to the next point is less by some factor than the distance to the previous point. Since we are taking geometrically smaller steps, and since our space has no gaps, we must eventually converge to a point in the space which we denote $|W| = \lim_{n\to\infty} W^{\circ n}(x)$. This point is fixed, because applying W one more time is the same as starting at $W(x)$ instead of x, and either way we get to the same point.

The fixed point is unique because if we assume that there are two, then we will get a contradiction: Suppose there are two fixed points x_1 and x_2; then the distance between $W(x_1)$ and $W(x_2)$, which is the distance between x_1 and x_2 since they are fixed points, would have to be smaller than the distance between x_1 and x_2; this is a contradiction.

Thus, the main result we have demonstrated is that when W is contractive, we get a fixed point

$$|W| = \lim_{n\to\infty} W^{\circ n}(x)$$

for any initial x.

bounded" part is one of several technicalities that arise at this point. What are these terms and what are they doing there? The terms make the statement precise and their function is to reduce complaint-mail writen by mathematicians. Having W contractive is meaningless unless we give a way of determining distance between two sets. There is such a metric, called the Haussdorff metric, which measures the difference between two closed and bounded subsets of the plane, and in this metric W is contractive on the space of closed and bounded subsets of the plane. This is as much as we will say about these these details.) Hutchinson's theorem allows us to to use the contractive mapping fixed point theorem (see box), which tells us that the map W will have a unique fixed point in the space of all images. That is, whatever image (or set) we start with, we can repeatedly apply W to it and we will converge to a fixed image. Thus W (or the w_i) completely determine a unique image.

In other words, given an input image f_0, we can run the copying machine once to get $f_1 = W(f_0)$, twice to get $f_2 = W(f_1) = W(W(f_0)) \equiv W^{\circ 2}(f_0)$, and so on. The attractor, which is the result of running the copying machine in a feedback loop, is the limit set

$$|W| \equiv f_\infty = \lim_{n \to \infty} W^{\circ n}(f_0)$$

which is not dependent on the choice of f_0. Iterated function systems are interesting in their own right, but we are not concerned with them specifically. We will generalize the idea of the copy machine and use it to encode grey-scale images; that is, images that are not just black and white but which contain shades of grey as well.

§2 Self-Similarity in Images.

In the remainder of this article, we will use the term image to mean a grey-scale image.

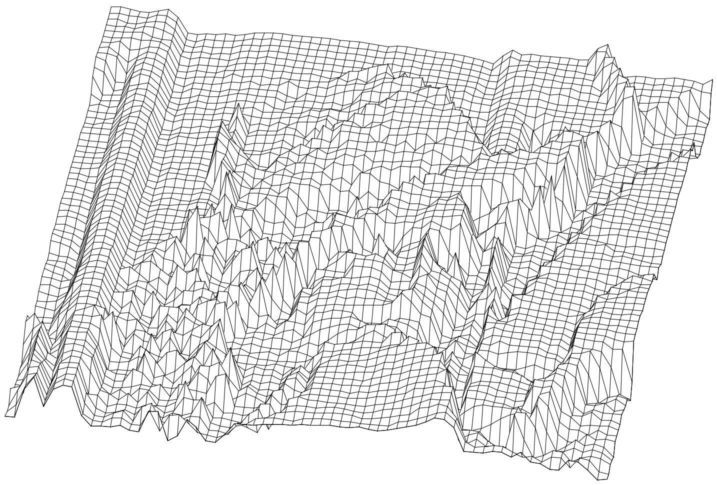

Figure 5. A graph generated from the Lena image.

Images as Graphs of Functions.

In order to discuss the compression of images, we need a mathematical model of an image. Figure 5 shows the graph of a special function $z = f(x, y)$. This graph is generated by using the image of Lena (see figure 6) and plotting the grey level of the pixel at position (x, y) as a height, with white being high and black being low. This is our model for an image, except that while the graph in figure 5 is generated by connecting the heights on a 64×64 grid, we generalize this and assume that every position (x, y) can have an independent height. That is, our model of an image has infinite resolution.

Thus when we wish to refer to an image, we refer to the function $f(x, y)$ which gives the grey level at each point (x, y). In practice, we will not distinguish between the *function* f (which gives us a z value for each x, y coordinate) and the *graph* of the function (which is a set in 3 space consisting of the points in the surface defined by f). For simplicity, we assume we are dealing with square images of size 1; that is, $(x, y) \in \{(u, v) : 0 \leq u, v \leq 1\} \equiv I^2$, and $f(x, y) \in I \equiv [0, 1]$. We have introduced some convenient notation here: I means the interval $[0, 1]$ and I^2 is the unit square.

Figure 6. The original 256×256 pixel Lena image.

A Metric* on Images.

Now imagine the collection of all possible images: clouds, trees, dogs, random junk, the surface of Jupiter, etc. We want to find a map W which takes an input image and yields an output image, just as we did before with subsets of the plane. If we want to know when W is contractive, we will have to define a distance between two images. There are many metrics to choose from, but the simplest to use is the sup metric

$$\delta(f, g) = \sup_{(x,y) \in I^2} |f(x, y) - g(x, y)|. \tag{1}$$

* Recall that a metric is a function that measures distance.

This** metric finds the position (x, y) where two images f and g differ the most and sets this value as the distance between f and g.

Natural Images are not Exactly Self Similar.

A typical image of a face, for example figure 6 does not contain the type of self-similarity that can be found in the fractals of figure 3. The image does not appear to contain affine transformations of itself. But, in fact, this image does contain a different sort of self-similarity. Figure 7 shows sample regions of Lena which are similar at different scales: a portion of her sholder overlaps a region that is almost identical, and a portion of the reflection of the hat in the mirror is similar (after transformation) to a part of her hat. The distinction from the kind of self-similarity we saw in figure 3 is that rather than having the image be formed of copies of its *whole* self (under appropriate affine transformation), here the image will be formed of copies of properly transformed *parts* of itself. These transformed parts do not fit together, in general, to form an exact copy of the original image, and so we must allow some error in our representation of an image as a set of transformations. This means that the image we encode as a set of transformations will not be an identical copy of the original image but rather an approximation of it.

Figure 7. Self similar portions of the Lena image.

In what kind of images can we expect to find this type of self-similarity? Experimental results suggest that most images that one would expect to "see" can be compressed by taking advantage of this type of self-similarity; for example, images of trees, faces, houses, mountains, clouds, etc. However, the existence of this restricted self-similarity and the ability of an algorithm to detect it are distinct issues, and it is the latter which concerns us here.

** There are other possible choices for image models and other possible metrics to use. In fact, the choice of metric determines whether the transformations we use are contractive or not. These details are important, but are beyond the scope of this article.

§3 A Special Copying Machine.

Partitioned Copying Machines.

In this section we describe an extension of the copying machine metaphor that can be used to encode and decode grey-scale images. The partitioned copy machine we will use has four variable components:

- the number copies of the original pasted together to form the output,
- a setting of position and scaling, stretching, skewing and rotation factors for each copy.

These features are a part of the copying machine definition that can be used to generate the images in figure 3. We add to the the following two capabilities:

- a contrast and brightness adjustment for each copy,
- a mask which selects, for each copy, a part of the original to be copied.

These extra features are sufficient to allow the encoding of grey-scale images. The last dial is the new important feature. It partitions an image into pieces which are each transformed separately. By partitioning the image into pieces, we allow the encoding of many shapes that are difficult to encode using an IFS.

Let us review what happens when we copy an original image using this machine. Each lens selects a portion of the original, which we denote by D_i and copies that part (with a brightness and contrast transformation) to a part of the produced copy which is denoted R_i. We call the D_i domains and the R_i ranges. We denote this transformation by w_i. The partitioning is implicit in the notation, so that we can use almost the same notation as with an IFS. Given an image f, one copying step in a machine with N lenses can be written as $W(f) = w_1(f) \cup w_2(f) \cup \cdots \cup w_N(f)$. As before the machine runs in a feedback loop; its own output is fed back as its new input again and again.

Partitioned Copying Machines are PIFS.

We call the mathematical analogue of a partitioned copying machine, a partitioned iterated function system (PIFS). As before, the definition of a PIFS is not dependent on the type of transformations that are used, but in this discussion we will use affine transformations. The grey level adds another dimension, so the transformations w_i are of the form,

$$w_i \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix} \tag{2}$$

where s_i controls the contrast and o_i the brightness of the transformation.

It is convenient to write

$$v_i(x,y) = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix}.$$

Since an image is modeled as a function $f(x,y)$, we can apply w_i to an image f by $w_i(f) \equiv w_i(x, y, f(x,y))$. Then v_i determines how the partitioned domains of an original are mapped to the copy, while s_i and o_i determine the contrast and brightness of the transformation. It is always implicit, and important to remember, that each w_i is restricted to $D_i \times I$, the vertical space above D_i. That is, w_i applies only to the part of the image that is above the domain D_i. This means that $v_i(D_i) = R_i$.

Since we want $W(f)$ to be an image, we must insist that $\cup R_i = I^2$ and that $R_i \cap R_j = \emptyset$ when $i \neq j$. That is, when we apply W to an image, we get some single valued function above each point of the square I^2. Running the copying machine in a loop means iterating the map W. We begin with an initial image f_0 and then iterate $f_1 = W(f_0), f_2 = W(f_1) = W(W(f_0))$, and so on. We denote the n-th iterate by $f_n = W^{\circ n}(f_0)$.

Fixed points for PIFS.

In our case, a fixed point is an image f that satisfies $W(f) = f$; that is, when we apply the transformations to the image, we get back the original image. The contractive mapping theorem says that the fixed point of W will be the image we get when we compute the sequence $W(f_0), W(W(f_0)), W(W(W(f_0))), \ldots$, where f_0 is *any* image. So if we can be assured that W is contractive in the space of all images, then it will have a unique fixed point which will then be some image.

Since the metric we chose in equation 1 is only sensitive to what happens in the z direction, it is not necessary to impose contractivity conditions in the x or y directions. The transformation W will be contractive when each $s_i < 1$; that is, when z distances are shrunk by a factor less than 1. In fact, the contractive mapping principle can be applied to $W^{\circ m}$ (for some m), so it is sufficient for $W^{\circ m}$ to be contractive. This leads to the somewhat surprising result that there is no specific condition on any specific s_i either. In practice, it is safest to take $s_i < 1$ to ensure contractivity. But we know from experiments that taking $s_i < 1.2$ is safe, and that this results in slightly better encodings.

Eventually Contractive Maps.

When W is not contractive and $W^{\circ m}$ is contractive, we call W *eventually contractive*. A brief explanation of how a transformation W can be eventually contractive but not contractive is in order. The map W is composed of a union of maps w_i operating on disjoint parts of an image. The iterated transform $W^{\circ m}$ is composed of a union of compositions of the form

$$w_{i_1} \circ w_{i_2} \circ \cdots w_{i_m}.$$

It is a fact that the product of the contractivities bounds the contractivity of the compositions, so the compositions will be contractive if each contains sufficiently contractive w_{i_j}. Thus W will be eventually contractive (in the sup metric) if it contains sufficient "mixing" so that the contractive w_i eventually dominate the expansive ones. In practice, given a PIFS this condition is simple to check in the sup metric.

Suppose that we take all the $s_i < 1$. This means that when the copying machine is run, the contrast is always reduced. This seems to suggest that when the machine is run in a feedback loop, the resulting attractor will be an insipid, contrast-less grey. But this is wrong, since contrast is created between ranges which have different brightness levels o_i. So is the only contrast in the attractor between the R_i? No, if we take the v_i to be contractive, then the places where there is contrast between the R_i in the image will propagate to smaller and smaller scale, and this is how detail is created in the attractor. This is one reason to require that the v_i be contractive.

We now know how to decode an image that is encoded as a PIFS. Start with any initial image and repeatedly run the copy machine, or repeatedly apply W until we get close to the fixed point f_∞. We will use Hutchinson's notation and denote this fixed point by $f_\infty = |W|$. The decoding is easy, but it is the encoding which is interesting. To encode an image we need to figure out R_i, D_i and w_i, as well as N, the number of maps w_i we wish to use.

§4 Encoding Images.

Suppose we are given an image f that we wish to encode. This means we want to find a collection of maps $w_1, w_2 \ldots, w_N$ with $W = \cup_{i=1}^{N} w_i$ and $f = |W|$. That is, we want f to be the fixed point of the map W. The fixed point equation

$$f = W(f) = w_1(f) \cup w_2(f) \cup \cdots w_N(f)$$

suggests how this may be achieved. We seek a partition of f into pieces to which we apply the transforms w_i and get back f. This is too much to hope for in general, since images are not composed of pieces that can be transformed non-trivially to fit exactly somewhere else in the image. What we can hope to find is another image $f' = |W|$ with $\delta(f', f)$ small. That is, we seek a transformation W whose fixed point $f' = |W|$ is close to, or looks like, f. In that case,

$$f \approx f' = W(f') \approx W(f) = w_1(f) \cup w_2(f) \cup \cdots w_N(f).$$

Thus it is sufficient to approximate the parts of the image with transformed pieces. We do this by minimizing the following quantities

$$\delta(f \cap (R_i \times I), w_i(f)) \quad i = 1, \ldots, N \tag{4}$$

That is, we find pieces D_i and maps w_i, so that when we apply a w_i to the part of the image over D_i, we get something that is very close to the part of the image over R_i. Finding the pieces R_i (and corresponding D_i) is the heart of the problem.

A Simple Illustrative Example.

The following example suggest how this can be done. Suppose we are dealing with a 256×256 pixel image in which each pixel can be one of 256 levels from grey (ranging from black to white). Let $R_1, R_2, \ldots, R_{1024}$ be the 8×8 pixel non-overlapping sub-squares of the image, and let $\mathbf{D}$ be the collection of all 16×16 pixel (overlapping) sub-squares of the image. The collection $\mathbf{D}$ contains $241 \cdot 241 = 58,081$ squares. For each R_i search through all of $\mathbf{D}$ to find a $D_i \in \mathbf{D}$ which minimizes equation 4; that is, find the part of the image that most looks like the image above R_i. This domain is said to *cover* the range. There are 8* ways to map one square onto another, so that this means comparing $8 \cdot 58,081 = 464,648$ squares with each of the 1024 range squares. Also, a square in $\mathbf{D}$ has 4 times as many pixels as an R_i, so we must either subsample (choose 1 from each 2×2

---

* The square can be rotated to 4 orientations or fliped and rotated into 4 other orientations, but that is all.

Least Squares

Given two squares containing n pixel intensities, $a_1, \ldots, a_n$ (from D_i) and $b_1, \ldots, b_n$ (from R_i). We can seek s and o to minimize the quantity

$$R = \sum_{i=1}^{n} (s \cdot a_i + o - b_i)^2.$$

This will give us a contrast and brightness setting that makes the affinely transformed a_i values have the least squared distance from the b_i values. The minimum of R occurs when the partial derivatives with respect to s and o are zero, which occurs when

$$s = \left[n^2 (\sum_{i=1}^{n} a_i b_i) - (\sum_{i=1}^{n} a_i)(\sum_{i=1}^{n} b_i) \right] \Big/ \left[n^2 \sum_{i=1}^{n} a_i^2 - (\sum_{i=1}^{n} a_i)^2 \right]$$

and

$$o = \left[\sum_{i=1}^{n} b_i - s \sum_{i=1}^{n} a_i \right] /n^2$$

In that case,

$$R = \left[\sum_{i=1}^{n} b_i^2 + s(s \sum_{i=1}^{n} a_i^2 - 2(\sum_{i=1}^{n} a_i b_i) + 2o \sum_{i=1}^{n} a_i) + o(on^2 - 2 \sum_{i=1}^{n} b_i) \right] /n^2 \qquad (5)$$

If $n^2 \sum_{i=1}^{n} a_i^2 - (\sum_{i=1}^{n} a_i)^2 = 0$, then $s = 0$ and $o = \sum_{i=1}^{n} b_i / n^2$.

sub-square of D_i) or average the 2×2 sub-squares corresponding to each pixel of R_i when we minimize equation 4.

Minimizing equation 4 means two things. First it means finding a good choice for D_i (that is the part of the image that most looks like the image above R_i). Second, it means finding a good contrast and brightness setting s_i and o_i for w_i. For each $D \in \mathbf{D}$ we can compute s_i and o_i using least squares regression (see box), which also gives a resulting root mean square (rms) difference. We then pick as D_i the $D \in \mathbf{D}$ which has the least rms difference.

A choice of D_i, along with a corresponding s_i and o_i, determines a map w_i of the form of equation 2. Once we have the collection $w_1, \ldots, w_{1024}$ we can decode the image by estimating $|W|$. Figure 8 shows four images: an arbitrary initial image f_0 chosen to show texture, the first iteration $W(f_0)$, which shows some of the texture from f_0, $W^{\circ 2}(f_0)$, and $W^{\circ 10}(f_0)$.

The result is surprisingly good, given the naive nature of the encoding algorithm. The original image required 65536 bytes of storage, where as the transformations required only 3968 bytes*, giving a compression ratio of 16.5:1. With this encoding $R = 10.4$ and each

* Each transformation required 8 bits in the x and y direction to determine the position of D_i, 7 bits

pixel is on average only 6.2 grey levels away from the correct value. Figure 8 shows how detail is added at each iteration. The first iteration contains detail at size 8×8, the next at size 4×4, and so on.

Jacquin [J] encoded images with less grey levels using a method similar to this example but with two sizes of ranges. In order to reduce the number of domains searched, he also classified the ranges and domains by their edge (or lack of edge) properties. This is very similar, coincidentally, to the scheme used by Boss and Jacobs [BJF] to encode contours.

Figure 8. An original image, the first, second, and tenth iterates of the encoding transformations.

---

for o_i, 5 bits for s_i and 3 bits to determine a rotation and flip operation for mapping D_i to R_i.

A Note About Metrics.

Two men flying in a balloon are sent off track by a strong gust of wind. Not knowing where they are, they approach a solitary figure perched on a hill. They lower the balloon and shout the the man on the hill, "Where are we?". There is a very long pause, and then the man shouts back, "You are in a balloon." The first man in the balloon turns to the second and says, "That man was a mathematician." Completely amazed, the second man asks, "How can you tell that?". Replies the first man, "We asked him a question, he thought about it for a long time, his answer was correct, and it was totally useless." This is what we have done with the metrics. When it came to a simple theoretical motivation, we use the sup metric which is very convenient for this. But in practice, we are happier using the rms metric which allows us to make least square computations. (We could have worked with the rms metric, of course, but checking contractivity in this metric is much harder).

§5 Ways to Partition Images.

The example of the last section is naive and simple, but it contains most of the ideas of a fractal image encoding scheme. First partition the image by some collection of ranges R_i. Then for each R_i seek from some collection of image pieces a D_i which has a low rms error. The sets R_i and D_i, determine s_i and o_i as well as a_i, b_i, c_i, d_i, e_i and f_i in equation 2. We then get a transformation $W = \cup w_i$ which encodes an approximation of the original image.

Quadtree Partitioning.

A weakness of the example is the use of fixed size R_i, since there are regions of the image that are difficult to cover well this way (for example, Lena's eyes). Similarly, there are regions that could be covered well with larger R_i, thus reducing the total number of w_i maps needed (and increasing the compression of the image). A generalization of the fixed size R_i is the use of a quadtree partition of the image. In a quadtree partition, a square in the image is broken up into 4 equally sized sub-squares, when it is not covered well enough by a domain. This process repeats recursively starting from the whole image and continuing until the squares are small enough to be covered within some specified rms tolerance. Small squares can be covered better than large ones because contiguous pixels in an image tend to be highly correlated.

An algorithm that works well for encoding 256×256 pixel images based on this idea can proceed as follows (see [FJB1]). Choose for the collection $\mathbf{D}$ of permissible domains all the sub-squares in the image of size $8, 12, 16, 24, 32, 48$ and 64. Partition the image recursively by a quadtree method until the squares are of size 32. For each square in the quadtree partition, attempt to cover it by a domain that is larger; this makes the v_i contractive. If a predetermined tolerance rms value e_c is met, then call the square R_i and the covering domain D_i. If not, then subdivide the square and repeat. This algorithm works well. It works even better if diagonally oriented squares are used in the domain pool $\mathbf{D}$ also. Figure 9 shows an image of a collie compressed using this scheme. In section 6 we discuss some of the details of this scheme as well as the other two schemes discussed below.

Figure 9. A collie (256×256) compressed with the quadtree scheme at 28.95:1 with an rms error of 8.5.

HV-Partitioning.

A weakness of the quadtree based partitioning is that it makes no attempt to select the domain pool **D** in a content dependent way. The collection must be chosen to be very large so that a good fit to a given range can be found. A way to remedy this, while increasing the flexibility of the range partition, is to use an HV-partition. In an HV-partition, a rectangular image is recursively partitioned either horizontally or vertically to form two new rectangles. The partitioning repeats recursively until a covering tolerance is satisfied, as in the quadtree scheme.

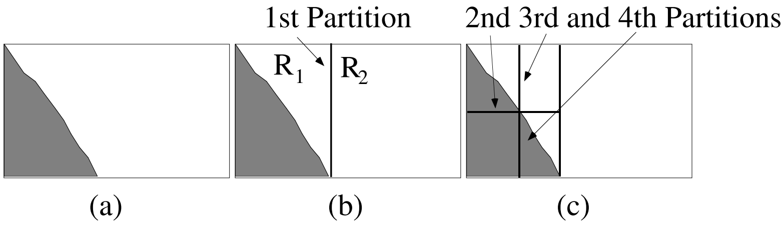

Figure 11. The HV scheme attempts to create self similar rectangles at different scales.

This scheme is more flexible, since the position of the partition is variable. We can then try to make the partitions in such a way that they share some self similar structure. For example, we can try to arrange the partitions so that edges in the image will tend to run diagonally through them. Then, it is possible to use the larger partitions to cover the smaller partitions with a reasonable expectation of a good cover. Figure 11 demonstrates this idea. The figure shows a part of an image (a); in (b) the first partition generates two rectangles, R_1 with the edge running diagonally through it, and R_2 with no edge; and in (c) the next three partitions of R_1 partition it into 4 rectangles, two rectangles which can be well covered by R_1 (since they have an edge running diagonally) and two which can be covered by R_2 (since they contain no edge). Figure 10 shows an image of San Francisco encoded using this scheme.

Figure 10. San Francisco (256×256) compressed with the HV scheme at 7.6:1 with an rms error of 7.1.

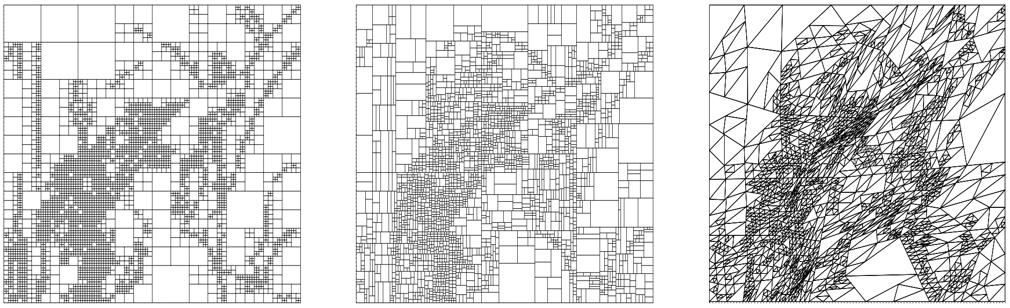

Figure 12. A quadtree partition (5008 squares), an HV partition (2910 rectangles), and a triangular partition (2954 triangles).

Triangular Partitioning.

Yet another way to partition an image is based on triangles. In the triangular partitioning scheme, a rectangular image is divided diagonally into two triangles. Each of these is recursively subdivided into 4 triangles by segmenting the triangle along lines that join three partitioning points along the three sides of the triangle. This scheme has several potential advantages over the HV-partitioning scheme. It is flexible, so that triangles in the scheme can be chosen to share self-similar properties, as before. The artifacts arising from the covering do not run horizontally and vertically, and this is less distracting. Also, the triangles can have any orientation, so we break away from the rigid 90 degree rotations of the quadtree and HV partitioning schemes. This scheme, however, remains to be fully developed and explored. Figure 12 shows sample partitions arising from the three partitioning schemes applied to the Lena image.

§6 Implementation Notes.

The pseudo-code in Table 1 shows two ways of encoding images using the idea presented. One method attempts to target a fidelity by finding a covering such that equation 4 is below some criterion e_c. The other method attempts to target a compression ratio by limiting the number of transforms used in the encoding.

Storing the Encoding Compactly.

To store the encoding compactly, we do not store all the coefficients in equation 2. The contrast and brightness settings are stored using a fixed number of bits. One could compute the optimal s_i and o_i and then discretize them for storage. However, a *significant* improvement in fidelity can be obtained if only discretized s_i and o_i values are used when computing the error during encoding (and equation 5 facilitates this). Using 5 bits to store s_i and 7 bits to store o_i has been found empirically optimal in general. The distribution of s_i and o_i shows some structure, so further compression can be attained by using entropy encoding.

The remaining coefficients are computed when the image is decoded. In their place we store R_i and D_i. In the case of a quadtree partition, R_i can be encoded by the storage order of the transformations if we know the size of R_i. The domains D_i must be stored as a position and size (and orientation if diagonal domain are used). This is not sufficient, though, since there are 8 ways to map the four corners of D_i to the corners of R_i. So we also must use 3 bits to determine this rotation and flip information.

In the case of the HV-partitioning and triangular partitioning, the partition is stored as a collection of offset values. As the rectangles (or triangles) become smaller in the partition, fewer bits are required to store the offset value. The partition can be completely reconstructed by the decoding routine. One bit must be used to determine if a partition is further subdivided or will be used as an R_i and a variable number of bits must be used to specify the index of each D_i in a list of all the partition. For all three methods, and without too much effort, it is possible to achieve a compression of roughly 31 bits per w_i on average.

In the example of section 4, the number of transformations is fixed. In contrast, the partitioning algorithms described are adaptive in the sense that they utilize a range size which varies depending on the local image complexity. For a fixed image, more transformations lead to better fidelity but worse compression. This trade-off between compression and fidelity leads to two different approaches to encoding an image f - one targeting fidelity and one targeting compression. These approaches are outlined in the pseudo-code in table 1. In the table, size(R_i) refers to the size of the range; in the case of rectangles, size(R_i) is the length of the longest side.

Table 1. Two pseudo-codes for an adaptive encoding algorithm

```
• Choose a tolerance level e_c.
• Set R_1 = I^2 and mark it uncovered.
• While there are uncovered ranges R_i do {
    • Out of the possible domains D, find the domain D_i
      and the corresponding w_i which best covers R_i (i.e.
      which minimizes expression (4)).
    • If δ(f ∩ (R_i × I), w_i(f)) < e_c or size(R_i) ≤ r_min then
        • Mark R_i as covered, and write out the transfor-
          mation w_i;
    • else
        • Partition R_i into smaller ranges which are
          marked as uncovered, and remove R_i from the list
          of uncovered ranges.
  }
```

a. Pseudo-code targeting a fidelity e_c.

```
• Choose a target number of ranges N_r.
• Set a list to contain R_1 = I^2, and mark it as uncovered.
• While there are uncovered ranges in the list do {
    • For each uncovered range in the list, find and store
      the domain D_i ∈ D and map w_i which covers it best,
      and mark the range as covered.
    • Out of the list of ranges, find the range R_j with
      size(R_j) > r_min which has the largest

                    δ(f ∩ (R_j × I), w_j(f))

      (i.e. which is covered worst).
    • If the number of ranges in the list is less than N_r
      then {
        • Partition R_j into smaller ranges which are added
          to the list and marked as uncovered.
        • Remove R_j, w_j and D_j from the list.
      }
  }
• Write out all the w_i in the list.
```

b. Pseudo-code targeting a compression having N transformations.

Acknowledgements

This work was partially supported by ONR contract N00014-91-C-0177. Other support was provided by the San Diego Super Computer Center; the Institute for Non-Linear Science at the University of California, San Diego; and the Technion Israel Institute of Technology.

References

[B] Barnsley, M. Fractals Everywhere. Academic Press. San Diego, 1989.

[BJ] R.D. Boss, E.W. Jacobs, "Fractal-Based Image Compression," NOSC Technical Report 1315, September 1989. Naval Ocean Systems Center, San Diego CA 92152-5000.

[FJB] Y. Fisher, E.W. Jacobs, and R.D. Boss, "Fractal Image Compression Using Iterated Transforms," to appear in Data Compression, J. Storer, Editor, Kluwer Academic Publishers, Norwall, MA.

[FJB1] Y. Fisher, E.W. Jacobs, and R.D. Boss, "Fractal Image Compression Using Iterated Transforms," NOSC Technical Report ???, Naval Ocean Systems Center, San Diego CA 92152-5000.

[H] John E. Hutchinson, *Fractals and Self Similarity.* Indiana University Mathamatics Journal, Vol. 35, No. 5. 1981.

[J] Jacquin, A., A Fractal Theory of Iterated Markov Operators with Applications to Digital Image Coding, Doctoral Thesis, Georgia Institute of Technology, 1989.

[JBF] R.D. Boss, E.W. Jacobs, "Fractal-Based Image Compression II," NOSC Technical Report 1362, June 1990. Naval Ocean Systems Center, San Diego CA 92152-5000.

[JFB] E.W. Jacobs, Y. Fisher, and R.D. Boss, "Image Compression: A Study of the Iterated Transform Method," to appear in *Signal Processing.*

[P1] "The Science of Fractals", H.-O. Peitgen, D. Saupe, Editors, Springer Verlag, New York, 1989.

[P2] "Fractals For Class Room", H.-O. Peitgen, D. Saupe, H. Jurgens, Springer Verlag, New York, 1991.

[WK] E. Walach, E. Karnin, "A Fractal Based Approach to Image Compression", Proceedings of ICASSP Tokyo, 1986.