Appendix A

A Discussion of Fractal Image Compression[1]

Yuval Fisher[2]

Caveat Emptor.

Anon

Recently, fractal image compression — a scheme using fractal transforms to encode general images — has received considerable attention. This interest has been aroused chiefly by Michael Barnsley, who claims to have commercialized such a scheme. In spite of the popularity of the notion, scientific publications on the topic have been sparse; most articles have not contained any description of results or algorithms. Even Barnsley's book, which discusses the theme of fractal image compression at length, was spartan when it came to the specifics of image compression.

The first published scheme was the doctoral dissertation of A. Jacquin, a student of Barnsley's who had previously published related papers with Barnsley without revealing their core algorithms. Other work was conducted by the author in collaboration with R. D. Boss and E. W. Jacobs[3] and also in collaboration with Ben Bielefeld.[4] In this appendix we discuss several schemes based on the aforementio ed work by which general images can be encoded as fractal transforms.

[1]This work was partially supported by ONR contract N00014-91-C-0177. Other support was provided by the San Diego Supercomputing Center and the Institute for Non-Linear Science at the University of California, San Diego.

[2]San Diego Supercomputing Facility, University of California, San Diego, La Jolla, CA 92093.

[3]Of the Naval Ocean Systems Center, San Diego.

[4]Of the State University of New York, Stony Brook.

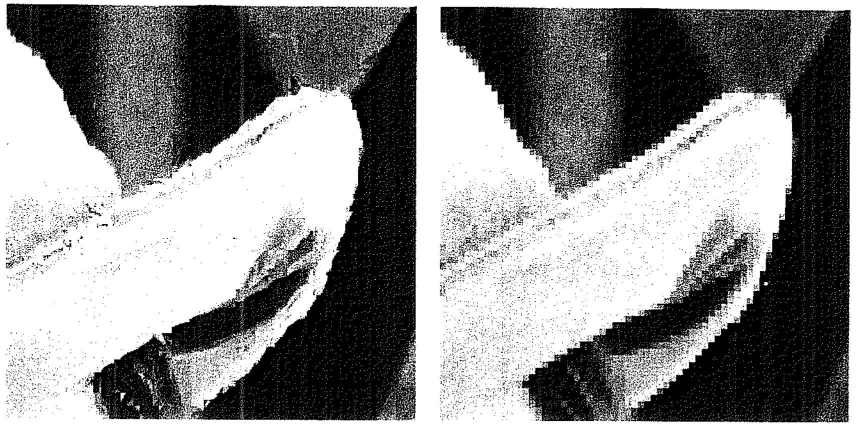
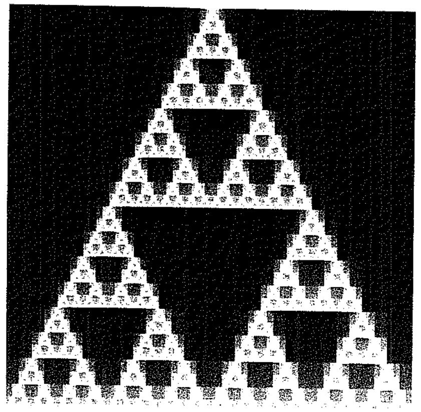


Figure A.1 : A portion of Lenna's hat decoded at 4 times its encoding size (left), and the original image enlarged to 4 times the size (right), showing pixelization.

Why is it "Fractal" Image Compression?

The image compression scheme can be said to be fractal in several senses. First, an image is stored as a collection of transforms that are very similar to the MRCM metaphor. This has several implications. For example, just as the Barnsley fern is a set which has detail at every scale, so does the decoded image have detail created at every scale. Also, if one scales the transformations in the Barnsley fern IFS (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 A.1 shows a detail from a fractal encoding of Lenna along with a magnification of the original. The whole original image can be seen in figure A.4 (left); this is the now famous image of Lenna which is commonly used in the image compression literature. 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.



Grey Scale Version of the Sierpinski Gasket

Figure A.2

Why is it Fractal Image "Compression"?

An image is stored on a computer as a collection of values which indicate a grey level or color at each point (or pixel) of the picture. It is typical to use 8 bits per pixel for grey-scale images, giving $2^8 = 256$ different possible levels of grey at each pixel. This yields a gradation of greys that is sufficient to make monochrome images stored this way look good. However, the image's pixel density must also be sufficiently high so that the individual pixels are not apparent. Thus, even small images require a large number of pixels and so they have a high memory requirement. However, the human eye is not sensitive to certain types of information loss, and so it is generally possible to store an approximation of an image as a collection of transforms using considerably less information than is required to store the original image.

For example, the grey-scale version of the Sierpinski gasket in figure A.2 can be generated from only 132 bits of information using the same decoding algorithm that generated the other encoded images in this section. Because this image is self-similar, it can be stored very compactly as a collection of transformations. This is the spirit of the idea behind the fractal image compression scheme presented in the next sections.

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. The compression ratio for the fractal scheme is hard to measure, since the image can be decoded at any scale. If we decode the grey-scale Sierpinski gasket at, say, two times its size, then we could claim 4 times the compression ratio since 4 times as many pixels would be required to store the decompressed image. For example, the decoded

Graph Generated From the Lenna Image.

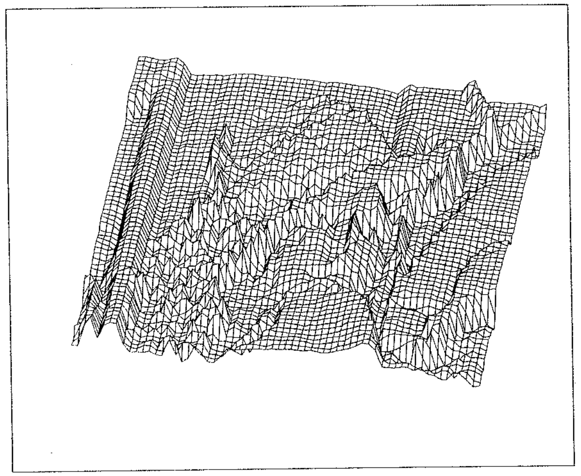


Figure A.3

image in figure A.1 is a portion of a 5.7:1 compression of the whole Lenna 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 91.2:1. This may seem like cheating, but since the 4-times-larger image has detail at every scale, it really isn't.

A.1 Self-Similarity in Images

The images we will encode are different than the images discussed in other parts of the book. Before, when we referred to an image, we meant a set that could be drawn in black and white on the plane, with black representing the points of the set. In this appendix, an image refers to something that looks like a black-and-white photograph.

Images as Graphs of Functions

In order to discuss the compression of images, we need a mathematical model of an image. Figure A.3 shows the graph of a special function $z = f(x,y)$. This graph is generated by using the image of Lenna (see figure A.4) 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 A.3 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). When we are dealing with an image of finite resolution, such as the images that are digitized and stored on computers, we must either average $f(x,y)$ over the pixels of the image or insist that $f(x,y)$ has a constant value over each pixel.

Normalizing Graphs of Images

For simplicity, we assume we are dealing with square images of size 1. We require $(x,y) \in I^2 = \{(u,v) \mid 0 \leq u,v \leq 1\}$, and $f(x,y) \in I = [0,1]$. Since we will want to use the contraction mapping principle, we will want to work in a complete metric space of images, and so we also will require that f is measurable. This is a technicality, and not a serious one since the measurable functions include the piecewise continuous functions, and one could argue that any natural image corresponds to such a function.

A Metric on Images

We also want to be able to measure differences between images, and so we introduce a metric on the space of images. There are many metrics to choose from, but the simplest to use is the sup metric

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

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.

There are other possible choices for image models and other possible metrics to use. In fact just as before, 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 appendix.

Natural Images are not Exactly Self-Similar

A typical image of a face, for example figure A.4 (left) does not contain the type of self-similarity that can be found in the Sierpinski gasket. 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 A.4 (right) shows sample regions of Lenna which are similar at different scales: a portion of her shoulder 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 with ferns and gaskets 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 parts are not identical copies of themselves under affine transformation, 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.

Finally, in what kind of images can we expect to find this type of *local* self-similarity? Experimental results suggest that most images that one

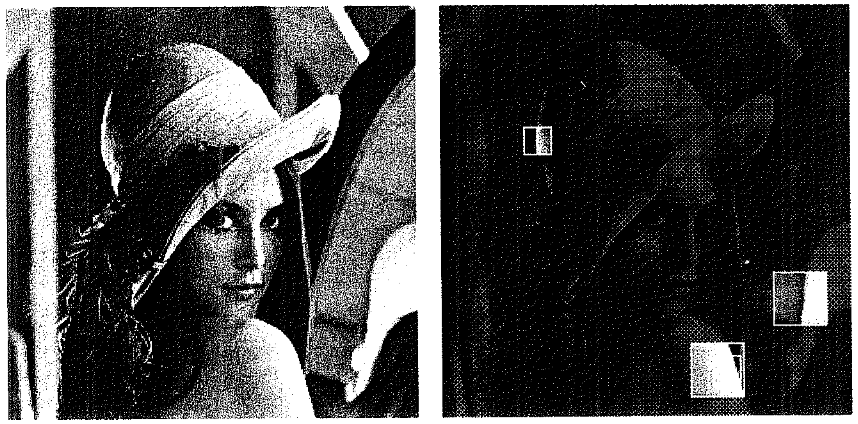


Figure A.4 : The original 256 × 256 pixel Lenna image (left) and some of its self-similar portions (right).

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 local self-similarity and the ability of an algorithm to detect it are distinct issues, and it is the latter which concerns us here.

A.2 A Special MRCM

Partitioned MRCMs

In this section we describe an extension of the multiple reduction copying machine metaphor that can be used to encode and decode grey-scale images. As before, the machine has several dials, or variable components:

Dial 1: number of lens systems,
Dial 2: setting of reduction factor for each lens system individually,
Dial 3: configuration of lens systems for the assembly of copies.

These dials are a part of the MRCM definition from chapter 5; we add to them the following two capabilities:

Dial 4: A contrast and brightness adjustment for each lens,
Dial 5: A mask which selects, for each lens, 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. For this reason, we call this MRCM a partitioned multiple reduction copying machine (PMRCM). By partitioning

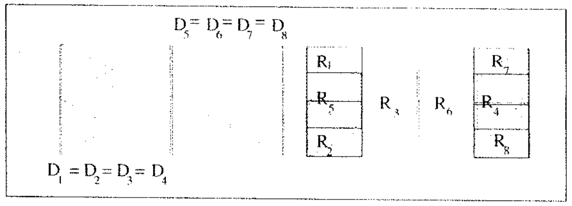
the image into pieces, we allow the encoding of many shapes that are difficult to encode on an MRCM, or IFS.

Let us review what happens when we put an original image on the copy surface of the 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 before. 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.

A PMRCM for a Bowtie

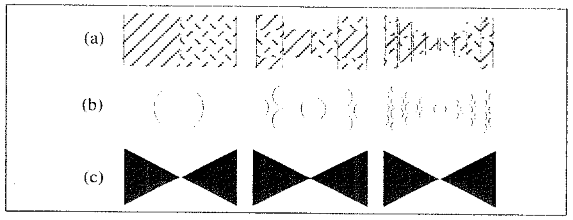
Consider the 8 lens PMRCM indicated in figure A.5. The figure shows two regions, one marked $D_1 = D_2 = D_3 = D_4$ and the other marked $D_5 = D_6 = D_7 = D_8$. These are the partitioned pieces of the original which will be copied by the 8 lenses. The lenses map each domain D_i to a corresponding range R_i, with a reduction factor of $1/2$. For simplicity, we assume that the contrast and brightness are not altered in this example. Figure A.6 shows three iterations of the PMRCM with three different initial images. The attractor for this system is the bow-tie figure shown in (c).

This example demonstrates the utility of a PMRCM. By partitioning the original to be copied, it is very easy to encode the bow-tie image (though the astute reader will notice that this image is also possible to encode using an IFS).



An 8 lens PMRCM encoding a bowtie.

Figure A.5



Three iterations of a PMRCM with three different intial images.

Figure A.6

PMRCM = PIFS

We call the mathematical analogue of a PMRCM, a partitioned iterated function system (PIFS). A PIFS has some features in common with the networked MRCM and Barnsley's recurrent iterated function systems, but they are not at all identical.

We haven't specified what kind of transformations we are allowing, and in fact one could build a PMRCM or PIFS with any transformation one wants. But in order to simplify the situation, and also in order to allow a compact specification of the final PIFS (in order to yield high compression), we restrict ourselves to transformations w_i of the form

$$w_i \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix} . \qquad \text{(A.1)}$$

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 $D_i \times I$. 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 Hutchinson operator 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 $f_n = W^n(f_0)$.

Fixed Points for PIFS

When will W have an attractive fixed point? By the contractive mapping principle, it is sufficient to have W be contractive. Since we have chosen a metric that 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 $s_i < 1$. In fact, the contractive mapping principle can be applied to W^m (for some m), so it is sufficient for W^m to be contractive. This leads to the somewhat surprising result that there is no specific condition on the 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^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^m is composed of a union of compositions of the form

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

Since the product of the contractivities bounds the contractivity of the compositions, the compositions may 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.

Suppose that we take all the $s_i < 1$. This means that when the PMRCM 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 or as a PMRCM. Start with any initial image and repeatedly run the copy machine, or repeatedly apply W until we get 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.

Decoding by Matrix Inversion

When we decode by iterating, we take an initial f_0 and compute $f_n = W(f_{n-1})$. This can also be written as

$$f_n(x,y) = s_i f_{n-1}(v_i^{-1}(x,y)) + o_i ,$$

where i is determined by the condition $(x,y) \in R_i$. Suppose we are dealing with an image of resolution $M \times M$. We can write the image as a column vector, and then this equation can be written as

$$f_n = S f_{n-1} + O ,$$

where S is an $M^2 \times M^2$ matrix with entries s_i that encode the v_i and O is a column vector containing the brightness values o_i. Then

$$f_n = S^n f_0 + \textstyle\sum_{j=1}^{n} S^{j-1} O ,$$

and if each $s_i < c < 1$ then the first term is 0 in the limit. (The condition $s_i < c < 1$ can be relaxed when W is eventually contractive). When $I - S$ is invertible,

$$f_\infty = \textstyle\sum_{j=0}^{\infty} S^j O = (I - S)^{-1} O,$$

where I is the identity matrix. Bielefeld pointed out that when each pixel value $f_n(x,y)$ depends on only one (or a few) other pixel values $f_{n-1}(v_i^{-1}(x,y))$, this matrix is very sparse and can be readily inverted.

A.3 Encoding Images

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

$$\delta(f \cap (R_i \times I), w_i(f)) \quad i = 1, \dots, N \tag{A.2}$$

Finding the pieces R_i (and corresponding D_i) is the heart of the problem.

A Simple Illustrative Example

The following example suggests how this can be done. Suppose we are dealing with a 256×256 pixel image at 8 bits per pixel. Let $R_1, R_2, \dots, R_{1024}$ be the 8×8 non-overlapping sub-squares of $[0, 255] \times [0, 255]$, and let $\mathbf{D}$ be the collection of all 16×16 sub-squares. 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 A.2. 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. 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 sub-square of D_i) or average the 2×2 sub-squares corresponding to each pixel of R_i when we minimize eqn. (A.2).

Minimizing equation (A.2) 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, 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 Point 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 hill on which a solitary figure is perched. They lower the balloon and shout to the man on the hill, "Where are we?". The man pauses for a long time and shouts back, just as the balloon is leaving earshot, "You are in a balloon." So one of the men in the balloon turns to the other 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 ver, convenient for this. But in practice, we are happier using the rms metric which allows us to make least square computations.

Least Squares

Given two squares containing n pixel intensities, $a_1, \dots, a_n$ and $b_1, \dots, b_n$. 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

$$s = \frac{n\left(\sum_{i=1}^{n} a_i b_i\right) - \left(\sum_{i=1}^{n} a_i\right)\left(\sum_{i=1}^{n} b_i\right)}{n\sum_{i=1}^{n} a_i^2 - \left(\sum_{i=1}^{n} a_i\right)^2}$$

and

$$o = \frac{1}{n}\left(\sum_{i=1}^{n} b_i - s\sum_{i=1}^{n} a_i\right).$$

In that case,

$$\begin{aligned} R = \frac{1}{n}\Bigg[\sum_{i=1}^{n} b_i^2 + s\left(s\sum_{i=1}^{n} a_i^2 - 2\sum_{i=1}^{n} a_i b_i + 2o\sum_{i=1}^{n} a_i\right) \\ + o\left(on - 2\sum_{i=1}^{n} b_i\right)\Bigg]. \end{aligned} \tag{A.3}$$

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

A choice of D_i, along with a corresponding s_i and o_i, determines a map w_i of the form of eqn. (A.1). Once we have the collection $w_1, \dots, w_{1024}$ we can decode the image by estimating $|W|$. Figure A.7 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, $W^2(f_0)$, and $W^{10}(f_0)$.

The result is surprisingly good, given the naive nature of the encoding algorithm. The original image required 65536 bytes of storage, whereas the transformations required only 3968 bytes,[5] giving a compression ratio of 16.5:1. With this encoding $R = 10.4$ and each pixel is on average only 6.2 grey levels away from the correct value. These images show 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.

A.4 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 eqn. (A.1). 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, Lenna'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 image is broken up into 4 equally sized sub-squares. Depending on some algorithmic criterion, each of these is again recursively sub-divided.

An algorithm for encoding 256×256 pixel images based on this idea can proceed as follows. 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. If a predetermined tolerance rms value 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 A.8 shows an image of a collie compressed using this scheme. In section A.5 we discuss some of the details of this scheme as well as the other two schemes discussed below.

HV-Partitioning

A weakness of the quadtree based partitioning is that it makes no attempt to select the domain pool $\mathbf{D}$ 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 some criterion is met, as before. This scheme is more flexible, since the position of the partition is variable. We

[5]Each transformation required 8 bits in the x and y direction to determine the position of D_i, 7 bits 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.

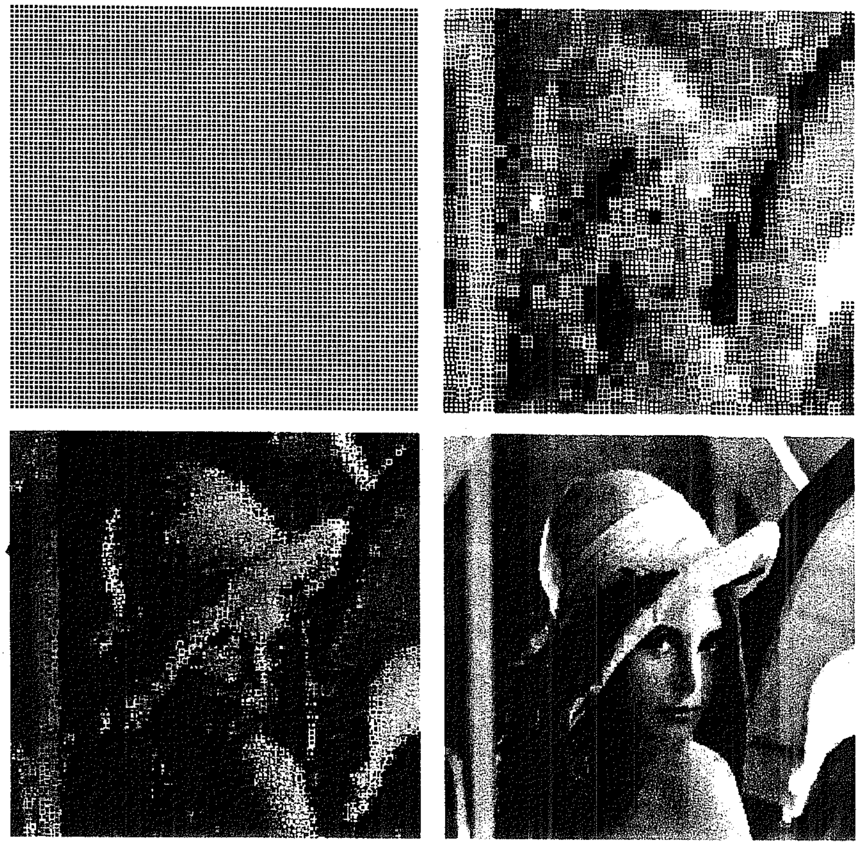


Figure A.7 : An original image, the first, second, and tenth iterates of the encoding transformations.

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 A.10 demonstrates this idea. The figure shows an part of an image (a); in (b) the first partition generates two rectangles, R_1 with the edge running diagonally through it,

A Collie

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

Figure A.8

San Francisco

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

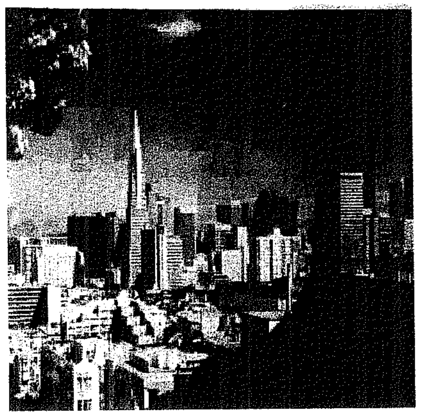
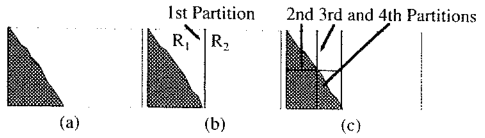


Figure A.9

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 A.9 shows an image of San Francisco encoded using this scheme.

Triangular Partitioning

Yet another way to partition an image is based on triangles. In the triangular partitioning scheme, a rectangular image is divided diagonally



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

Figure A.10

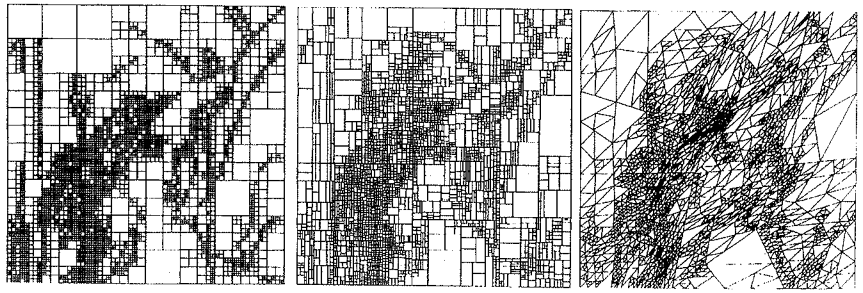


Figure A.11 : A quadtree partition (5008 squares), an HV partition (2910 rectangles), and a triangular partition (2954 triangles).

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. However, the artifacts arising from imperfect 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 A.11 shows sample partitions arising from the three partitioning schemes applied to the Lenna image.

A.5 Implementation Notes

Storing the Encoding Compactly

To store the encoding compactly, we do not store all the coefficients in eqn. (A.1). 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 eqn. (A.3) 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 partitions. 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 A.3, 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 below. In the code, 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.

Optimizing Encoding Time

Another concern is encoding time, which can be significantly reduced by employing a classification scheme on the ranges and domains. Both ranges and domains are classified using some criteria such as their edge-like nature, or the orientation of bright spots, etc. Considerable time savings result from only using domains in the same class as a given range when seeking a cover, the rationale being that domains in the same class as a range should cover it best.

Pseudo-Code

```
a. Pseudo-code targeting a fidelity $e_c$.
• 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 $\mathbf{D}$, find the domain $D_i$ and the
      corresponding $w_i$ which best covers $R_i$ (i.e., which minimizes
      expression (A.2)).
    • If $\delta(f \cap (R_i \times I), w_i(f)) < e_c$ or size$(R_i) \le r_{min}$ then
        • Mark $R_i$ as covered, and write out the transformation $w_i$;
    • else
        • Partition $R_i$ into smaller ranges which are marked as uncov-
          ered, and remove $R_i$ from the list of uncovered ranges.
}
```

```
b. Pseudo-code targeting a compression having $N$ transformations.
• 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
      $D_i \in \mathbf{D}$ 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

        $\delta(f \cap (R_j \times I), w_j(f))$

      (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.
```