Fractal Image Compression

Martin V Sewell

MSc Computing Science project report,
Department of Computer Science, Birkbeck College, University of London

1994

Contents

Abstract

This paper seeks to explain the emerging technology known as fractal image compression. It presents both the mathematical ideas and the practical techniques involved in compressing images by means of fractals. With claims of compression ratios in excess of 10 000 to 1, fractal compression is certainly a promising new technology. However, the method is in fact quite controversial, with some people claiming it doesn't work well, while others claim it works wonderfully; as the technology is fast developing, but as yet unproven. At the moment, very little is known about fractal compression and what is known is difficult to ascertain. The main aspects of the technology are patented by M. F. Barnsley, who pioneered fractal image compression. This paper attempts to clarify the key ideas behind the methodology; and includes an overview of the competition, which includes the JPEG standard and vector quantization. Included as an appendix is the source code for a model digital implementation of fractal image compression of binary black-and-white images, written in C. Examples of decompressed images are also included, as is an evaluation of the program.

Fractal image compression relies on the fact that all real-world images are rich in *affine redundancy*; that is, under suitable affine transformations, large bits of real-world images look like smaller bits of the same real-world image. The basic idea behind the method is to express the image as a set of local iterated function systems (IFSs). An IFS is just a set of contractive affine transformations. The IFS coefficients are then stored in place of the original. No information regarding the pixels is actually stored. When the original is required, the IFSs are iterated on an arbitrary starting image of the desired resolution. The image can then be displayed quickly and zooming will create infinite levels of (synthetic) fractal detail. The problem is how to efficiently generate the IFSs from the original image.

The hypothetical Multiple Reduction Photocopy Machine is used to illustrate the implementation of IFSs. Also, the classical Sierpinski triangle is used as an example of a self-similar shape to illustrate how IFSs work. One important result that is central to fractal image compression is known as the Collage Theorem, which was proved by M. Barnsley in his book Fractals Everywhere. The Collage Theorem states what an IFS must be like in order to represent an image. Also covered is what is known as the inverse problem, which involves going from a given image to an IFS that can generate a good approximation of the original. This remains unsolved.

Other perhaps contrary aspects of the technology are covered, such as why standard IFSs are not actually used in practical image compression. Also explained are the problems encountered with resolution enhancement, when compression ratios are quoted after 'zooming in' on an image. Another problem highlighted is the speed problem, and why compression can take many hours to complete.

The model digital implementation and much of the theory covered deal with black and white images, although an explanation is given as to how grayscale and colour images may be dealt with, and why far superior compression ratios may be achieved with such images. The possibility of fractal video compression is also covered briefly.

Included in the conclusion are some suggestions for future work, some of which may have been tried, while others are quite novel. The promises of 10 000 fold compression turned out to be less of a breakthrough than expected, with 'rigged' images being compressed and/or human assistance used in the compression process. However, this paper aims to put fractal image compression into perspective, as it does indeed appear worthy of investigation.

Chapter 1 Introduction

1.1 Compression

With the advent of multimedia, the necessity for the storage of large numbers of high quality images is increasing. One obstacle that has to be overcome is the large size of image files. For example, a single 800- by 600-pixel true-colour image requires three bytes per pixel, plus a header, which amounts to over 1.37 Mb of disk space, thus almost filling a 1.4Mb high-density diskette. Clearly, some form of compression is necessary. As well as saving storage space, compressed files take less time to transmit via modem, so money can be saved on both counts.

The choice of compression algorithm involves several conflicting considerations. These include degree of compression required, and the speed of operation. Obviously if one is attempting to run programs direct from their compressed state, decompression speed is paramount. The other consideration is size of compressed file versus quality of decompressed image.

There are essentially two sorts of data compression. 'Lossless' compression works by reducing the redundancy in the data. The decompressed data is an exact copy of the original, with no loss of data. Huffman Encoding and LZW are two examples of lossless compression algorithms. There are times when such methods of compression are unnecessarily exact. 'Lossy' compression sacrifices exact reproduction of data for better compression. It both removes redundancy and creates an approximation of the original. The JPEG standard is currently the most popular method of lossy compression. Obviously, a lossless compression method must be used with programs or text files, while lossy compression is really only suitable for graphics or sound data, where an exact reproduction is not necessary. Fractal image compression is a lossy compression method.

1.2 Fractals

The French mathematician Benoit B. Mandelbrot first coined the term *fractal* in 1975. He derived the word from the Latin *fractus*, which means "broken", or "irregular and fragmented". In fact, the birth of fractal geometry is usually traced to Mandelbrot and the 1977 publication of his seminal book *The Fractal Geometry of Nature*. Mandelbrot claimed that classical Euclidean geometry was inadequate at describing many natural objects, such as clouds, mountains, coastlines and trees. So he conceived and developed fractal geometry.

There are two main groups of fractals: linear and nonlinear. The latter are typified by the popular Mandelbrot set and Julia sets, which are fractals of the complex plane. However, the fractals used in image compression are linear, and of the real plane. So, the fractals used are not chaotic; in other words, they are not sensitive to initial conditions. They are the fractals from Iterated Function Theory. An Iterated Function System (IFS) is simply a set of contractive affine transformations. IFSs may efficiently produce shapes such as ferns, leaves and trees.

This presented an intriguing possibility; since fractal mathematics is good for generating natural looking images, could it not, in the reverse direction, be used to compress images? The possibility of using fractals for image encoding rests on two suppositions:

1. many natural scenes possess this detail within detail structure (e.g. clouds);
2. an IFS can be found that generates a close approximation of a scene using only a few transformations.

The first point is true, but excludes many real world images, while the second leads to the so called "inverse problem", which is explained later.

1.3 A Brief History of Fractal Image Compression

1977	Benoit Mandelbrot finishes the first edition of "The Fractal Geometry of Nature"
1981	John E. Hutchinson publishes "Fractals and Self Similarity"
1983	Revised edition of "The Fractal Geometry of Nature" is published
1985	Michael F. Barnsley and Stephen Demko introduce Iterated Function Theory in their paper "Iterated Function Systems and the Global Construction of Fractals"
1987	Iterated Systems Incorporated is founded in Georgia, US
1988	Michael Barnsley and Alan D. Sloan wrote the article "A Better Way to Compress Images" published in Byte, claiming fractal compression ratios of 10 000 to 1
	Barnsley publishes the book "Fractals Everywhere", which presents the mathematics of Iterated Function Systems, and proves a result known as the Collage Theorem
1990	Barnsley's first patent is granted (US Patent # 4 941 193)
1991	M. Barnsley and A. Sloan are granted US Patent # 5 065 447 "Method and Apparatus for Processing Digital Data"
	J.M. Beaumont publishes a paper "Image Data Compression Using Fractal Techniques"
1992	Arnaud E. Jacquin publishes an article that describes the first practical fractal image compression method
	Iterated Systems Inc finally break even and begin to make money
	Microsoft Corporation publish the multimedia encyclopedia "Microsoft Encarta" which uses fractal image compression to great effect
1993	The book "Fractal Image Compression" by Michael Barnsley and Lyman P. Hurd is published
	The Iterated Systems' product line matures

Chapter 2 Topology

2.1 Metric Spaces and Affine Transformations

Notations

R	real numbers
N	natural numbers, 1, 2, 3, …
$f^{\circ n}(x)$	$f(\ldots f(f(f(x)))\ldots)$ where there are n $f(\ )$s
$x \vee y$	the maximum of x and y

Definition
Let **X** be a non-empty set and let d be a function $d : \mathbf{X} \times \mathbf{X} \to \mathbf{R}$ which satisfies the following properties:-
(i) $d(x, y) \geq 0,\ \forall\, x, y \in \mathbf{X}$
(ii) $d(x, y) = 0$ if, and only if, $x = y,\ \forall\, x, y \in \mathbf{X}$
(iii) $d(x, y) = d(y, x),\ \forall\, x, y \in \mathbf{X}$
(iv) $d(x, z) \leq d(x, y) + d(x, z),\ \forall\, x, y \in \mathbf{X}$
Then $(\mathbf{X}, d)$ is called a *metric space*.

Intuitively, we may think of $d(x, y)$ as the distance between the points x and y in **X**.

Definition
A transformation $\omega: \mathbf{R}^2 \to \mathbf{R}^2$ of the form
$$\omega\,(x, y) = (ax + by + e, cx + dy + f),$$
where a, b, c, d, e, and f are real numbers, is called a (two-dimensional) *affine* transformation.

Using equivalent notations:
$$\omega\,(\boldsymbol{x}) = \omega\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} = \mathbf{Ax} + \mathbf{t}$$

So an affine transformation consists of a linear transformation, **A**, which deforms space relative to the origin followed by a translation **t**. Generally an affine transformation can be used to map all the points in one parallelogram into all the points in another parallelogram using some combination of a rotation, a scaling, a shear and a translation.

Definition
Let $(\mathbf{X}, d)$ be a metric space. A sequence $\{x_n\}_{n \in \mathbf{N}} \in \mathbf{X}$ is a *Cauchy sequence* if for every $\varepsilon > 0$ there exists $n_0 \in \mathbf{N}$ such that, for all $m, n \geq n_0$ $d(x_n, x_m) < \varepsilon$.

Definition
A sequence $\{x_n\}_{n \in \mathbf{N}}$ of points in a metric space $(\mathbf{X}, d)$ is said to *converge* to a point $x \in \mathbf{X}$ if, for any given number $\varepsilon > 0$, there is an integer $N > 0$ such that
$$d(x_n, x) < \varepsilon \text{ for all } n > N.$$
The point $x \in \mathbf{X}$, to which the sequence converges, is called the *limit* of the sequence.

Definition
A metric space $(\mathbf{X}, d)$ is *complete* if every Cauchy sequence $\{x_n\}_{n \in \mathbf{N}}$ in **X** has a limit $x \in \mathbf{X}$.

The idea is that a metric space **X** is 'complete' if 'the sequences in **X** that ought to converge do actually converge'.

2.2 The Contraction Mapping Theorem

Definition

Let f: $\mathbf{X} \to \mathbf{X}$ be a transformation on a space. A point $x_f \in \mathbf{X}$ such that $f(x_f) = x_f$ is called a *fixed point* of the transformation.

Definition
A transformation $f : \mathbf{X} \to \mathbf{X}$ on a metric space $(\mathbf{X}, d)$ is called *contractive* or a *contraction mapping* if there is a constant $0 \leq s < 1$ such that

$$d\,(f(x), f(y)) \leq s \cdot d\,(x, y) \quad \forall\ x, y \in \mathbf{X}$$

Any such number s is called a *contractivity factor* for f.

Theorem (The Contraction Mapping Theorem)
Let f: $\mathbf{X} \to \mathbf{X}$ be a contraction mapping on a complete metric space $(\mathbf{X}, d)$. Then f possesses exactly one fixed point $x_f \in \mathbf{X}$, and moreover for any point $x \in \mathbf{X}$, the sequence $\{f^{\circ n}(x) : n = 0, 1, 2, \ldots\}$ converges to x_f; that is,

$$\lim_{n \to \infty} f^{\circ n}(x) = x_f, \text{ for each } x \in \mathbf{X}.$$

Proof
See *Fractal Image Compression* by M F Barnsley and L P Hurd.

2.3 The Hausdorff Space

Definition
Let $S \subset \mathbf{X}$ be a subset of a metric space $(\mathbf{X}, d)$. S is *compact* if every infinite sequence $\{x_n\}_{n \in \mathbf{N}}$ in S contains a subsequence having a limit in S.

Definition
Let $(\mathbf{X}, d)$ be a complete metric space. Then $\mathbf{H}(\mathbf{X})$ denotes the space whose points are the compact subsets of $\mathbf{X}$, other than the empty set.

Definition
Let $(\mathbf{X}, d)$ be a complete metric space, $x \in \mathbf{X}$, and $B \in \mathbf{H}(\mathbf{X})$. Define

$$d(x, B) = \min\{d(x, y) : y \in B\}.$$

$d(x, B)$ is called the *distance* from the point x to the set B.

Definition
Let $(\mathbf{X}, d)$ be a complete metric space. Let A and B belong to $\mathbf{H}(\mathbf{X})$. Define

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

$d(A, B)$ is called the *distance* from the set $A \in \mathbf{H}(\mathbf{X})$ to the set $B \in \mathbf{H}(\mathbf{X})$.

Definition
Let $(\mathbf{X}, d)$ be a complete metric space. Then the *Hausdorff distance* between the points A and B in $\mathbf{H}(\mathbf{X})$ is defined by

$$h(A, B) = d(A, B) \vee d(B, A).$$

We also call h the *Hausdorff metric* on $\mathbf{H}$.

Chapter 3 Iterated Function Systems and the Collage Theorem

3.1 Iterated Function Systems

Definition
A (*hyperbolic*) *iterated function system* consists of a complete space ($\mathbf{X}$, d) together with a finite set of contraction mappings $w_n : \mathbf{X} \to \mathbf{X}$, with respective contractivity factors s_n, for $n = 1, 2, ..., N$. The notation for this IFS is $\{\mathbf{X} ; w_n, n = 1, 2, \ldots, N\}$ and its contractivity factor is $s = \max\{s_n : n = 1, 2, \ldots, N\}$.

N.B.
1. The abbreviation *IFS* may be used for an iterated function system.
2. The word *hyperbolic* is in parentheses in the definition, as it is usually dropped in practice.
3. Almost invariably, affine transformations are used in iterated function systems.

Theorem (The IFS Theorem)
Let $\{\mathbf{X} ; w_n, n = 1, 2, \ldots, N\}$ be a hyperbolic iterated function system with contractivity factor s. Then the transformation W: $\mathbf{H}(\mathbf{X}) \to \mathbf{H}(\mathbf{X})$ defined by

$$W(\mathrm{B}) = \bigcup_{n=1}^{N} w_n(B),$$

for all $B \in \mathbf{H}(\mathbf{X})$, is a contraction mapping on the complete metric space ($\mathbf{H}(\mathbf{X})$, $h(d)$) with contractivity factor s; that is

$$h(W(B), W(C)) \leq s \cdot h(B, C)$$

for all $B, C \in \mathbf{H}(\mathbf{X})$. It has a unique fixed point, $A \in \mathbf{H}(\mathbf{X})$, which obeys

$$A = W(A) = \bigcup_{n=1}^{N} w_n(A),$$

and is given by $A = \lim_{n \to \infty} W^{\circ n}(B)$ for any $B \in \mathbf{H}(\mathbf{X})$.

Proof
See *Fractals Everywhere* by M F Barnsley.

Definition
The fixed point $A \in \mathbf{H}(\mathbf{X})$ described in the IFS Theorem is called the *attractor* of the IFS.

Definition
Let ($\mathbf{X}$, d) be a metric space and let $C \in \mathbf{H}(\mathbf{X})$. Define a transformation $w_0 : \mathbf{H}(\mathbf{X}) \to \mathbf{H}(\mathbf{X})$ by $w_0(B) = C$ for all $B \in \mathbf{H}(\mathbf{X})$. Then w_0 is called a *condensation transformation* and C is called the associated *condensation set*.

Note that a condensation transformation $w_0 : \mathbf{H}(\mathbf{X}) \to \mathbf{H}(\mathbf{X})$ is a contraction mapping on the metric space ($\mathbf{H}(\mathbf{X})$, $h(d)$), with contractivity factor equal to zero, and that its unique fixed point is the condensation set.

Definition
Let $\{\mathbf{X} ; w_1, w_{2, \ldots,} w_{\mathrm{N}}\}$ be a hyperbolic IFS with contractivity factor $0 \leq s < 1$.
Let $w_0 : \mathbf{H}(\mathbf{X}) \to \mathbf{H}(\mathbf{X})$ be a condensation transformation. Then $\{\mathbf{X}; w_0, w_{1, \ldots,} w_{\mathrm{N}}\}$ is called a (*hyperbolic*) *IFS with condensation*.

Note
The IFS Theorem can be generalized to the case of an IFS with condensation.
For proof see *Fractals Everywhere*, by M F Barnsley.

Definition
An *iterated function system with probabilities* consists of an IFS

$$\{\mathbf{X} ; w_1, w_{2, \dots,} w_N\}$$

together with an ordered set of numbers $\{p_1, p_{2, \dots,} p_N\}$, such that

$$p_1 + p_2 + p_3 + \dots + p_N = 1 \text{ and } p_i > 0 \qquad \text{for } i = 1, 2, \dots, N.$$

The probability p_i is associated with the transformation w_i. The nomenclature *IFS with probabilities* may be used as an abbreviation. The full notation for such an IFS is

$$\{\mathbf{X} ; w_1, w_{2, \dots,} w_N ; p_1, p_{2, \dots,} p_N\}.$$

The probabilities are related to the measure theory of IFS attractors, and play a role in the computation of images of the attractor of an IFS attractor using the random iteration algorithm, and also using the multiple reduction photocopy machine algorithm (see section 3.3), as applied to grayscale images. They are not used with the multiple reduction photocopy machine algorithm when it is applied to binary images.

3.2 Two Algorithms for Computing Fractals from IFSs

The Deterministic Algorithm
Let $\{\mathbf{X} ; w_1, w_{2, \dots,} w_N\}$ be a hyperbolic IFS. Choose a compact set $A_0 \subset \mathbf{R}^2$. Then compute successively $A_n = W^{\circ n}(A)$ according to

$$A_{n+1} = \cup_{n \in \mathbf{N}} w_j(A_n) \quad \text{for } n = 1, 2, \dots.$$

Thus construct a sequence $\{A_n : n = 0, 1, 2, 3, \dots\} \subset \mathbf{H}(\mathbf{X})$. Then by the IFS Theorem the sequence $\{A_n\}$ converges to the attractor of the IFS in the Hausdorff metric

The Random Iteration Algorithm
Let $\{\mathbf{X} ; w_1, w_{2, \dots,} w_N ; p_1, p_{2, \dots,} p_N\}$ be an IFS with probabilities. Choose $x_0 \in \mathbf{X}$ and then choose recursively, independently,

$$x_n \in \{w_1(x_{n-1}), w_2(x_{n-1}), \dots, w_N(x_{n-1})\} \qquad \text{for } n = 1, 2, 3, \dots,$$

where the probability of the event $x_n = w_i(x_{n-1})$ is p_i. Thus, construct a sequence

$$\{x_n : n = 0, 1, 2, 3, \dots\} \subset \mathbf{X}.$$

3.3 The Multiple Reduction Photocopy Machine

The Multiple Reduction Photocopy Machine is an imaginary image duplicator, which may be used to find successive approximations for the attractor of an IFS. It functions like a normal photocopy machine with the following exceptions:

1. The machine has several lenses, to create multiple copies of the original.
2. Each lens reduces the size of the original.
3. The copier operates in a feedback loop; the output of one stage becomes the input to the next.

The lenses can be set for different reduction factors, and the reduced (output) images can be positioned at any desired location. Thus, the image may undergo any contractive affine transformation. The initial input can be anything. When the machine is run, the output tends towards a limiting image, which is independent of the initial image. The limiting image is the fixed point, or the attractor.
When dealing with grayscale, rather than black and white images, the lenses also require filters, so that the brightness of illumination may be controlled. The filter on lens i attenuates the light that passes through it by a factor proportional to p_i, where p_i is the probability associated with lens i in the corresponding IFS with probabilities.

3.4 The Sierpinski Triangle

A classical example of a self-similar shape is the Sierpinski triangle, or gasket. It was named after the Polish mathematician Waclaw Sierpinski, who first described it in 1916. The Sierpinski triangle may be created using a Multiple Reduction Photocopy Machine in the following manner. An image is placed on the machine, reduced by one half and copied three times, once onto each vertex of a triangle. If the corresponding contractive affine transformations are written in the form:

$$\omega\,(x, y) = (ax + by + e, cx + dy + f),$$

The IFS can be written as follows:

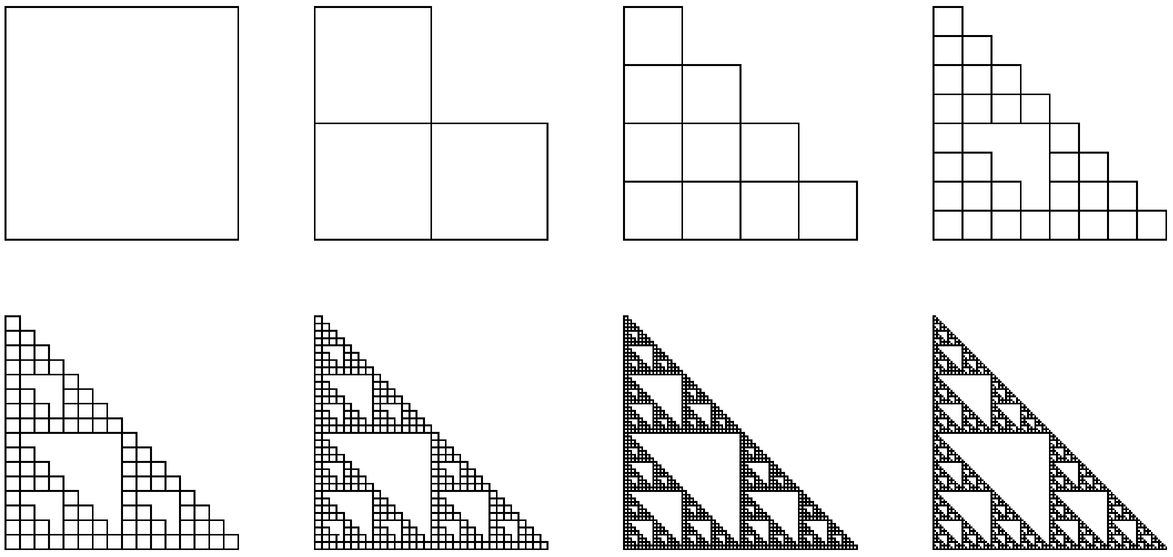
a	*b*	*c*	*d*	*e*	*f*
0.5	0	0	0.5	0	0
0.5	0	0	0.5	0	$w/2$
0.5	0	0	0.5	$w/2$	$w/2$

where *w* is the width of the initial image.

If using an IFS with probabilities, we would use:

a	*b*	*c*	*d*	*e*	*f*	*p*
0.5	0	0	0.5	0	0	$^1/_3$
0.5	0	0	0.5	0	$w/2$	$^1/_3$
0.5	0	0	0.5	$w/2$	$w/2$	$^1/_3$

To illustrate the implementation of the Multiple Reduction Photocopy Machine, or the result of running the Deterministic Algorithm, successive approximations of Sierpinski's triangle are shown below. A larger, more detailed Sierpinski triangle can be found in Appendix A.



3.5 The Inverse Problem

Given an image to encode, finding the optimal IFS that can generate an image which is as close as possible to the original, is known as the "inverse problem". Various algorithms have been proposed for the inverse problem, including:

- genetic algorithms
- moment methods
- the hungry algorithm
- the "graduate student algorithm"

However, it remains unsolved.

3.6 The Collage Theorem

Theorem (The Collage Theorem)
Let ($\mathbf{X}$, d) be a complete metric space. Let $T \in \mathrm{H}(\mathbf{X})$ be given, and let $\varepsilon \geq 0$ be given. Choose an IFS (or IFS with condensation) $\{\mathbf{X} ; (w_0), w_1, w_2, \ldots, w_N\}$ with contractivity factor $0 \leq s < 1$ so that

$$h \left(T, \bigcup_{n=1\,(n=0)}^{N} w_n(T) \right) \leq \varepsilon,$$

where $h(d)$ is the Hausdorff metric. Then

$$h(T, A) \leq \frac{\varepsilon}{1 - s},$$

where A is the attractor of the IFS. Equivalently,

$$h(T, A) \leq (1 - s)^{-1}\, h \left(T, \bigcup_{n=1\,(n=0)}^{N} w_n(T) \right) \text{ for all } T \in \mathrm{H}(\mathbf{X}).$$

Proof
See *Fractals Everywhere*, by M F Barnsley.

The Collage Theorem tells us that to find an IFS whose attractor is "close to" or "looks like" a given set, one must endeavor to find a set of transformations—contraction mappings on a suitable space within which the given set lies—such that the union, or collage, of the images of the given set under the transformations is near to the given set. Nearness is measured using the Hausdorff metric. In other words, if an IFS can be found which, when applied once to an image, gives an image which looks like the original, then iterating the IFS on any starting image gives a result which is even more like the original image. This result is the key to finding an IFS that can be used to compress an image. The Collage Theorem does not solve the inverse problem, but it does provide a way of approaching it.

Continuity Condition
Small changes in the parameters of an IFS will lead to small changes in the attractor, provided that the system remains hyperbolic. In other words, as the coefficients of an IFS are altered, the attractor changes smoothly and without any sudden jumps. This is very useful as it means that one method of performing the compression is to write a program which allows a user to manually alter the coefficients, until a suitable set of transformations is found. Using the Collage Theorem, this is when the IFS transforms the original image into a passable likeness of itself. It also means that one can smoothly interpolate between attractors: this is useful for image animation, for example.

Chapter 4 Automatic Fractal Image Compression and the Fractal Transform

4.1 Local Iterated Function Systems

The Advantages of Representing Images Using Iterated Function Schemes
By utilizing the self-similarity and repetition in nature, a relatively small number of transformations (and probabilities if used) lead to very high compression ratios. Also, given a set of affine transformations, reproduction of the image is computationally straightforward, is well suited to parallel computation, and is stable – small changes in the transformations lead to small changes in the invariant set. The ability of the transformations to define the image at arbitrarily small scales, and the ease at which small regions may be zoomed in on are also points in favour of IFSs.

The Problem With Using Iterated Function Systems to Compress Images
The main disadvantage of the method is the difficulty of obtaining a set of transformations to represent a given image. None of the algorithms proposed for the "inverse problem" have been successfully used in practical image compression. In order to fully automate fractal image compression, an algorithm must be used which requires no 'user interference', such as adjusting the coefficients in an IFS. Standard IFSs are not, in fact, good for general and practical image compression. A high correlation between different parts of a picture are required. The method may be excellent for giving an overall picture of a tree, but is no use if the exact arrangement of the leaves on different branches is important.

Even if the inverse problem were solved, it may be to no avail. Real world scenes are very diverse in subject matter, there may be totally different looking objects in the same scene. They do not, on the whole, obey fractal geometry. Real ferns do not branch down to infinity, and they may be distorted, discoloured, perforated and torn; while other objects are even further from fractal geometry. One theoretical reason why IFSs may never be successfully used in compression is the fact that given the IFS for object A, and the IFS for object B, they cannot be combined to give the IFS for object $A \cup B$, or object $A \cap B$. In practical fractal compression, to capture the diversity of real images, what is used instead is the "fractal transform", which uses local (or partitioned) IFSs, in which, the (contractive) transformations do not map the whole image onto part of itself, but map one part of the image onto another (smaller) part of the image. They allow the compression process to be fully automated.

Definition
Let $(\mathbf{X}, d)$ be a compact metric space. Let R be a non-empty subset of $\mathbf{X}$. Let $w : R \to \mathbf{X}$ and let s be a real number with $0 \le s < 1$. If

$$d(w(x), w(y)) \le s \cdot d(x, y) \text{ for all } x, y \text{ in } R,$$

then w is called a *local contraction mapping* on $(\mathbf{X}, d)$ and s is a contractivity factor for w.

Definition
Let $(\mathbf{X}, d)$ be a compact metric space, and let $w_i : R_i \to \mathbf{X}$ be a local contraction mapping on $(\mathbf{X}, d)$, with contractivity factor s_i , for $i = 1, 2, \ldots, N$, where N is a finite positive integer. Then

$$\{w_i : R_i \to \mathbf{X} : i = 1, 2, \ldots, N\}$$

is called a *local iterated function system* or a *local IFS*.
The number $s = \max \{s_i : i = 1, 2, \ldots, N\}$ is called the *contractivity factor* of the local IFS.

The Collage Theorem for a Local IFS
The Collage Theorem may be applied to a local IFS, if we (cautiously) treat a local IFS just as though it were a standard IFS.

4.2 The Escape Time Algorithm

The Escape Time Algorithm may be used for computing pictures of IFS attractors. More specifically, it may be used to calculate the binary attractor of a local IFS. It determines whether a point x is a member of the attractor. Only points which have not escaped after n iterations are plotted. Some pseudocode follows. In this example, the image screen represents the whole screen; and the black pixels are completely tiled by domain blocks. Each domain block will contain at least one black pixel.

```
counter = 0
max_no_of_iterations = $n$
repeat
    input $x \in$ image_screen
    if $x \in$ domain_block
        $x = f(x)$
    else
        {output "$x \notin$ attractor"
         exit}
    counter = counter + 1
    if counter = $n$
        {output "$x \in$ $n$th approximation of attractor"
         exit}
until false
```

4.3 The Fractal Transform Process for Automatic Image Compression

Compression Algorithm for a Binary Image

1. Input an image G, where G is the set of black points.
2. Partition the image into nonoverlapping domain blocks, D_i. The union of the domain blocks must cover the entire image, G, but they can be any size or shape.
3. Introduce a set of possible range blocks, R. These must be larger than the domain regions, can overlap, and need not cover the entire image.
4. Define a finite set of contractive affine transformations, w_i (which map from a range block R to a domain block D_i).
5. For each domain block

```
    For each range block
        For each transformation
            Calculate the Hausdorff distance $h(w_i(R \cap G), D_i \cap G)$ (or use another metric)
        Next transformation
    Next range block
Record the range block and transformation which produced the minimum Hausdorff distance
Next domain block.
```

6. Output an IFS file comprising a header and the recorded data in the form of a local IFS code.
7. Apply a lossless data compression algorithm to the IFS file.

The compressed image may be decompressed by applying the multiple reduction photocopy machine or the escape time algorithm. In practice, these steps are carried out on a digital image, and each step is expressed digitally. This involves standard quantization and discretization procedures, which are customary in computer graphics.

Chapter 5 Model Digital Implementation in C

5.1 Automatic Image Compression

An Automatic Digital Implementation of Fractal Image Compression - A Working Model

The programs included in Appendix B are written in C, Using Borland Turbo C++ Version 1.01. The programs are designed to run on any IBM-compatible personal computer. The image format chosen to illustrate fractal image compression is that of .BMP. This was due to the wide availability of Windows Paintbrush. Due to availability of printers and ease of illustration, the program only supports black and white images. The numbered steps in the compression process below correspond to those in section 4.3.

Compression

1. The input is a black and white .BMP file, the image is treated as the set of black pixels.
2. The domain blocks are chosen to be 4- by 4-pixel squares.
3. The possible range blocks are chosen to be 8- by 8-pixel squares. A range block starts every 4 pixels, both horizontally and vertically, so there is a 4 pixel overlap. The possible range blocks are numbered, starting from the bottom left corner, row by row. So any block may be uniquely identified by just one integer.
4. The choice of affine transformations is restricted to the group of eight canonical isometries of a square:

Symmetry	**Type of Transformation**
0	identity
1	orthogonal reflection about mid-vertical axis
2	orthogonal reflection about mid-horizontal axis
3	rotation around centre of block, through π
4	orthogonal reflection about first diagonal ($i = j$)
5	rotation around centre of block, through $3\pi/2$
6	rotation around centre of block, through $\pi/2$
7	orthogonal reflection about second diagonal ($i + j = 7$)

   These symmetries are all made to be contractive, when mapping from a range block to a (smaller) domain block.
5. Each domain block is visited in turn, starting from the bottom left of the image, and working along a row at a time. (If the domain block consists only of white pixels, and therefore is not treated as covering the image; it becomes a "dummy" domain block, and is given a 'symmetry value' of 8. This is because it is not treated as a domain block, so is not paired with a range block.) For each domain block each range block is visited, and each symmetry is applied to the range block. (It should be noted that 'range blocks' which do not cover part of the image, i.e. contain only white pixels are excluded from the pool of possible range blocks.) The distance metric is calculated for each combination. It is simply the number of pixels by which the domain block differs from the transformed range block. If at any point this is found to be zero, the current range block and symmetry are recorded and the loop is broken to the point where the next domain block is visited. Otherwise the combination of range block and symmetry which produced the minimum distance is recorded at the end of the loop.
6. The original .BMP header plus the list of affine transformations, one for each domain block, each consisting of range block number, an integer (2 bytes) plus symmetry number, 0-7 (1 byte) is then saved as an .IFS file.
7. The lossless compression program PKZip version 2.04g (for DOS) from PKWare is then applied to the IFS file.

5.2 Automatic Image Decompression

The Fractal Image Decompression Process

1. Decompress the IFS file (using PKUnzip).

```
2.  Read header and affine transformations from the IFS file.
3.  Create memory buffers for the domain and range screens.
4.  Initialize the range screen buffer to an arbitrary initial stage.
5.  Repeat
        For each domain
            Replace this domain with the appropriate transformed range block
        Next domain
        Copy contents of domain screen to range screen
    Until number of iterations required is attained.
6.  Output the final domain screen, to produce a .BMP file with the same header, and of the
    same size as the original.
```

5.3 An Evaluation of the Working Model

Example of Use of the Program

Below is an example of how to use the programs compress.exe, decompress.exe, compare.exe, pkzip.exe and pkunzip.exe on an image 'filename.bmp'.

The examples are in the form of a flow diagram, with the input on the left and the output on the right. The format is as follows:

first line: description of process
second line: description of files
third line: filenames
fourth line: what is actually typed at the command prompt

Compression

fractal compression *lossless compression*

original image ifs file zipped file

filename.bmp $\longrightarrow$ filename.ifs $\longrightarrow$
filename.zip

compress filename.bmp pkzip -ex filename.zip filename.ifs

Decompression

lossless decompression *fractal decompression*

zipped file ifs file decompressed file

filename.zip $\longrightarrow$ filename.ifs $\longrightarrow$
defilena.bmp

pkunzip filename.zip decompress filename.ifs

Error Measurement

compare original image with decompressed image

original image, decompressed image error measurement

filename.bmp, defilena.bmp $\longrightarrow$

compare filename.bmp defilena.bmp

Table of Results

Original Image	Size of Image / pixels width × height	Size of File / bytes
angry.bmp	304 × 145	5862
breeze.bmp	304 × 156	6302
darts.bmp	248 × 166	5374
fgrpoint.bmp	112 × 58	990
horse.bmp	264 × 240	8702
lady.bmp	170 × 235	5702
man.bmp	320 × 270	10862

Original Image	Size of IFS File / bytes	Size of Zipped File / bytes
angry.bmp	4790	1772
breeze.bmp	5412	1290
darts.bmp	4618	1649
fgrpoint.bmp	900	540
horse.bmp	6426	2460
lady.bmp	7738	2728
man.bmp	8844	1699

Original Image	Compression Time	Decompression Time
angry.bmp	47m 13s	2s
breeze.bmp	18m 50s	2s
darts.bmp	43m 44s	2s
fgrpoint.bmp	1m 42s	< 1s
horse.bmp	1hr 23m 28s	2s
lady.bmp	59m 42s	3s
man.bmp	4hr 32m 28s	3s

Original Image	Compression Ratio	Percentage Error
angry.bmp	3.3 : 1	9%
breeze.bmp	4.9 : 1	2%
darts.bmp	3.3 : 1	9%
fgrpoint.bmp	1.8 : 1	10%
horse.bmp	3.5 : 1	7%
lady.bmp	2.1 : 1	11%
man.bmp	6.4 : 1	9%

Assessment of Results

The images, and the decompressed images may be seen in Appendix C. All of the programs were run from the hard disk of a 40MHz 486DX PC. The times quoted are machine dependent and also would be slower if run from a floppy disk. It should be noted that much higher compression ratios would be attainable with grayscale or colour images, as the original black and white images are already down to one bit per pixel. However, in absolute terms, the compressed files are remarkably small for images; with man.bmp compressed to a size where each byte represents over 50 pixels, or over 6 pixels per bit. The image lady.bmp actually increased in size after the fractal compression, due to the relatively low amount of all white areas, and hence a larger proportion of the image needed to be covered by domain blocks. Correspondingly, man.bmp achieved the highest compression ratio, due to the large quantity of plain white space. Compression ratios may be improved by increasing the size of the domain blocks, but this would be at the expense of the quality of the decompressed image. Compression time is obviously related to the size of the initial file, but is also effected by the nature of the image.

Even quite small errors in black and white cartoon style images can be unacceptable, so it is not surprising that the images look quite distorted. The uniform 'grey' areas in particular were poorly reproduced, as the search for a range block which, after a contractive transformation had a similar pixel pattern, was unsuccessful. The quality of the images could be improved by enlarging the range pool, i.e. allowing more possible range blocks, e.g. one beginning every two rather than every four pixels. Alternatively, a larger number of contractive transformations could be made available. Both of these possible solutions would be at the expense of compression time, but would not necessarily decrease the compression ratio. Errors would be harder to detect in grayscale or colour images. The lowest error was attained with the image breeze.bmp, due to a higher than average amount of affine redundancy, in this case all black and all white areas, with fewer blocks of pixels containing fine detail.

To conclude, neither the compression ratios, nor the quality of the images is particularly impressive, however, a sensible balance between the two was reached. The program was written for clarity, and to serve as an example of the method, rather than optimized for speed or image quality. In this respect, it was a success. A program dealing with grayscale or colour images would have achieved better compression ratios and deteriorations in quality would be harder to detect.

Chapter 6 Conclusion

6.1 Grayscale Images, Colour Images and Video Compression

Grayscale and Colour Images
Although the working model is designed only for binary (black and white) images, the same principle may be applied to grayscale and colour images. In a grayscale image, a value is assigned to each pixel, depending on where it lies on the range of gray tones that exist between black and white. With one byte per pixel, this gives a range of 0 to 255.

Three dimensional affine transformations are required for compressing grayscale images. The method works as follows; for each domain block, the mean value of the pixels in the block is calculated. Then, for each range block, in order to be compared to the domain block, it has a spatial rescale of ½, with the mean of the four pixels in each appropriate square being used for the new pixel. It must then undergo an intensity rescale by ¾, so each pixel value is then multiplied by ¾. The mean value of its pixels is then calculated. The difference between the two means is calculated, i.e.

intensity_shift = domain_mean - range_mean.

Then each pixel in the range block has the value intensity_shift added to it. Now the means are equivalent. Then the symmetries are looped over, and the distance metric used between the domain block and the transformed range block with an intensity shift is mean squared error, or the sum of the squared differences of pixel values. So when the block with the smallest error is found, in addition to recording the range block and symmetry, the intensity shift is also required. This will have a value in the range -255...255.

When decompressing a grayscale image, the initial range screen is arbitrary, say a uniform gray, with a pixel value of 128. As above, the range blocks undergo a spatial rescale of ½ and an intensity rescale of ¾, followed by adding the intensity_shift to each pixel before replacing the appropriate domain block.

Alternatively, by transmitting the domain block gray-level means, a blurred version of the original image can be constructed at the decoder, and the offsets do not have to be computed at all.

Colour images can be treated as three grayscale images, one for red, one for green and one for blue. The Collage Theorem and the rest of the method still works.

It should be evident that due to the initial size of the files, grayscale images can be compressed to achieve much higher compression ratios than black and white images. In turn, colour images can be compressed even further.

Fractal Video Compression
As well as still picture coding, fractal techniques can be used for one-dimensional coding, sound and moving picture coding. There is a fractal compression version of MPEG which can be used to compress real-time video. This is done by extending the foregoing scheme to three-dimensional blocks of data, with, say $4 \times 4 \times 4$ domain blocks and $8 \times 8 \times 8$ range blocks. This form of compression could prove invaluable with multimedia and CD-ROMs; an uncompressed 10-second video clip with 30 frames per second at 320 by 200 pixels in true colour requires $10\times30\times320\times200\times3 \approx 55$ Mbytes.

6.2 Resolution Enhancement

One thing that fractal compression offers over other image compression algorithms is the ability to zoom in on the decompressed image. Zooming will generate (theoretically) infinite levels of (synthetic) fractal detail. Although resolution enhancement is a powerful feature, it is also open to abuse, in terms of claimed compression ratios. Barnsley and his company Iterated Systems have used this to their advantage. If one takes a portrait, say a 100- by 100-pixel grayscale image, one byte per pixel, you have a 10 000 byte file. If after compressing it, the compressed file is 250 bytes, that is a compression ratio of 40:1. Now, if you zoom in on the person's hair at a 5× magnification, you would see a texture that still looks like hair. This is now effectively part of a 500- by 500-pixel image, which would require 250 000 bytes uncompressed. So there is now an *effective* compression ratio of 1000:1. The problem is that detail has not been retained, but generated. So there is no guarantee that the zoomed in image will look as it should. For example, zooming in on a person's face will not reveal the pores. Objectively, fractal image compression offers a form of interpolation, which is a useful and attractive property when higher resolutions are required; but care must be taken when interpreting compression ratios. After all, resolution enhancement is the process of compressing an image, expanding it to a higher resolution, saving it, then discarding the iterated function system. In other words, the compressed fractal image (the IFSs) is the means to an end, and not the end itself.

6.3 Compression and the Speed Problem

The fractal-transform process is inherently asymmetric. Far more computation is required for compression than for decompression. The length of compression time is a major drawback of fractal compression. The essence of the compression process is the pairing of each domain block to a range block, such that the difference between the two, under an affine transformation, is minimal This involves a lot of searching, especially when an exact match is never found. For example, let us suppose the image to be compressed (using the implementation written in C) has a width of w pixels, and a height of h pixels (to simplify things assume w and h are both divisible by 4).

number of 4- by 4-pixel domain blocks:	(w/4)(h/4)
number of possible 8- by 8-pixel range blocks:	(w/4-1)(h/4-1)
number of possible symmetries:	8
total number of possible pairings to test:	8(w/4)(h/4)(w/4-1)(h/4-1)

Given an 800- by 600-pixel image that is over 7×10^9 possible combinations, which means the compression time will be measured in hours, rather than seconds or even minutes for large images. However, it could be argued that decompression time is of greater importance than compression time.

6.4 The Competition - JPEG and Error Measurement

Fractal image compression's main competitor is the JPEG standard. The acronym JPEG stands for Joint Photographic Experts Group; and it is a collaborative effort between the two standards committees CCITT and ISO. This compression procedure makes use of the discrete cosine transform (DCT). Compression and decompression take the same amount of time.

Advantages of Fractal Compression
Resolution and device independent.
Decompression is fast (typically 6 times faster than JPEG and twice as fast as reading the original, uncompressed image).
Superior for high compression ratios; gives more aesthetically pleasing results.
Compression ratios can be improved by taking more time during compression without any increase in decompression time or decrease in image quality.

Disadvantages of Fractal Compression
Compression is very slow (typically 12 times longer than JPEG).
An unproven technology.
Patented and expensive for a small user.

Advantages of JPEG
Superior for low compression ratios.
Because it is an industry standard, it is free.

Disadvantages of JPEG
Does not handle binary (1-bit-per-pixel) images, at least not efficiently (JBIG losslessly compresses binary images).
Does not handle colourmapped images.
Does not handle motion picture compression (MPEG may be used).
Does not work well with non-realistic images which are not "natural", e.g. cartoons or line drawings.
Quality of images degrades seriously with high compression ratios.
Blockiness, or "pixelation" as compression ratios rise above about 20:1.
Classical *Gibb's phenomenon* noticeable around discontinuities because higher-frequency data associated with sharp edges is suppressed.

Compressed Images and Error Measurement
With a system such as image compression, it would appear imperative that a formal means of assessing its performance is agreed upon. This would appear especially true if valid comparative statements are to be made regarding different compression technologies.

Indeed, great pains are taken to precisely measure and quantify the error present in an uncompressed image, and great effort is expended toward minimizing the error. These measures include signal-to-noise ratio, root mean square error, and mean absolute error. However, these may be of dubious value. A simple example is systematic shift on a grayscale image: add 10 to every pixel. Standard error measures indicate a large distortion, but the image has merely been brightened.

Proponents of fractal compression claim that signal-to-noise ratio is not a good error measure and that the distortions present are much more 'natural looking' than the blockiness of JPEG, at both low and high bit rates. This is a valid point, but is by no means universally accepted. However, JPEG achieves much of its compression by exploiting known limitations of the human eye, and is intended for compressing images that will be looked at by the humans, rather than machines.

With the absence of an easy to compute error measurement that accurately captures subjective impression of human viewers, it is probably better to judge images by eye. Despite the above, as a general rule, JPEG tends to be superior for compression ratios of up to about 40:1, while fractal compression is better for compression ratios above this point. This figure bodes well for JPEG, since beyond this point, images can become too distorted to use. However, fractal compression's resolution independence may well turn out to be a significant advantage.

6.5 Analogies with Vector Quantization

Vector Quantization (VQ) is another lossy compression method in which a "vector" is a small rectangular block of pixels. The idea is that a few common patterns are stored in a separate file called a codebook.

	Vector Quantization	**Fractal Compression**
1.	Domain blocks and range blocks are the same size	Range blocks are always larger
3.	Range blocks are copied directly	Range blocks undergo luminance scaling and offset (unless dealing with a binary image) plus a transformation
4.	Codebook is stored separately from the image being coded, and is required for both compression and decompression	Codebook is not explicitly stored, the image itself is in effect a "virtual codebook", used only in the decompression
5.	Codebook is shared among many images	Virtual codebook is specific to each image, no "universal codebook" required

The last two points above are probably the main advantages fractal compression has over VQ.

6.6 The Future

Fractal image compression is still in its infancy and there are many aspects of this fast developing technology which are still partially unsolved, or untried. There are many opportunities to develop the ideas further and improve upon the methodology. Below are a few areas which could be explored.

- The geometry of the domain and range blocks need not be square, or even rectangle. This was just an imposition made to keep the problem tractable. Other geometries which might yield less blockiness, could be considered.

- The size of domain blocks, size of range blocks, size of the pool of range blocks and the number of allowable affine transformations was by no means optimized, and this could be investigated.

- A different distance metric between blocks could be used.

- Quadtrees could possibly be used with different sized blocks and multi-level image partitions.

- Work could be done to speed the compression stage up, e.g. possibly testing the range blocks in the locality of the current domain block first, or somehow eliminating a range block earlier in the process if it is unlikely to be a 'best-fit'.

- The pool of range blocks could be split into different sets; e.g. smooth blocks, edge blocks and midrange blocks. For a given domain block, the respective category is then searched for the best match.

- Automatic fractal coding is, of course, a non-intelligent compression scheme, in that it makes no assumptions about the image it is compressing. In an object-oriented scheme, data could be sent representing the types of objects in the image, instead of data representing the pixels. Fractal compression fits in here as many objects have homogeneous surfaces, surrounded by boundaries.

References

Aitken W, "The Big Squeeze", *Personal Computer World*, May 1994, pp 396-402

Anson L F, "Fractal Image Compression", *Byte*, October 1993, pp 195-202

Barnsley M F, *Fractals Everywhere*, Academic Press Professional, 1993, 1988

Barnsley M F and Demko S "Iterated Function Systems and the Global Construction of Fractals", *The Proceedings of the Royal Society of London*, Vol A 399, 1985, pp 243-275

Barnsley M F and Hurd L P, *Fractal Image Compression*, AK Peters Ltd., 1992

Barnsley M F and Sloan A D, "A Better Way to Compress Images", *Byte*, January 1988, pp 215-223

Beaumont J M, "Image Data Compression Using Fractal Techniques", *BT Technology Journal*, Vol 9, No 4, October 1991, pp 93-109

Corcoran E, "Not Just a Pretty Face", *Scientific American*, Vol 262, March 1990, pp 51-52

Demko S, Hodges L and Naylor B, "Construction of Fractal Objects with Iterated Function Systems", *Computer Graphics*, *Siggraph '85 Conference Proceedings*, Vol 19, No 3, July1985, pp271-278

"Pntng by Nmbrs", *The Economist*, Vol 307, 21 May 1988

Falconer K, *Fractal Geometry : Mathematical Foundations and Applications*, John Wiley & Sons Ltd, 1990

Gibbs W W, "Practical Fractal", *Scientific American*, July 1993, pp 89-90

Horner J, *FRAC'Cetera*, Vol 03, Issues 9-11, 1994

Hutchinson J E, "Fractals and Self Similarity", *Indiana University Mathematics Journal*, Vol 30, No 5, 1981, pp 713-747

Jacquin A E, "Image Coding Based on a Fractal Theory of Iterated Contractive Image Transformations, *IEEE Transactions on Image Processing*, Vol 1, No 1, January 1992, pp 18-30

James M, "The Main Squeeze", *Computer Shopper*, June 1994, pp 467-472

Jürgens H, Peitgen H-O and Saupe D, "The Language of Fractals", *Scientific American*, August 1990, pp 40-47

Kandebo S W, "Fractals Research Furthers Digitized Image Compression", *Aviation Week & Space Technology*, Vol 128, 25 April 1988, pp 91-95

Mandelbrot B B, *The Fractal Geometry of Nature*, W.H. Freeman and Company, 1982

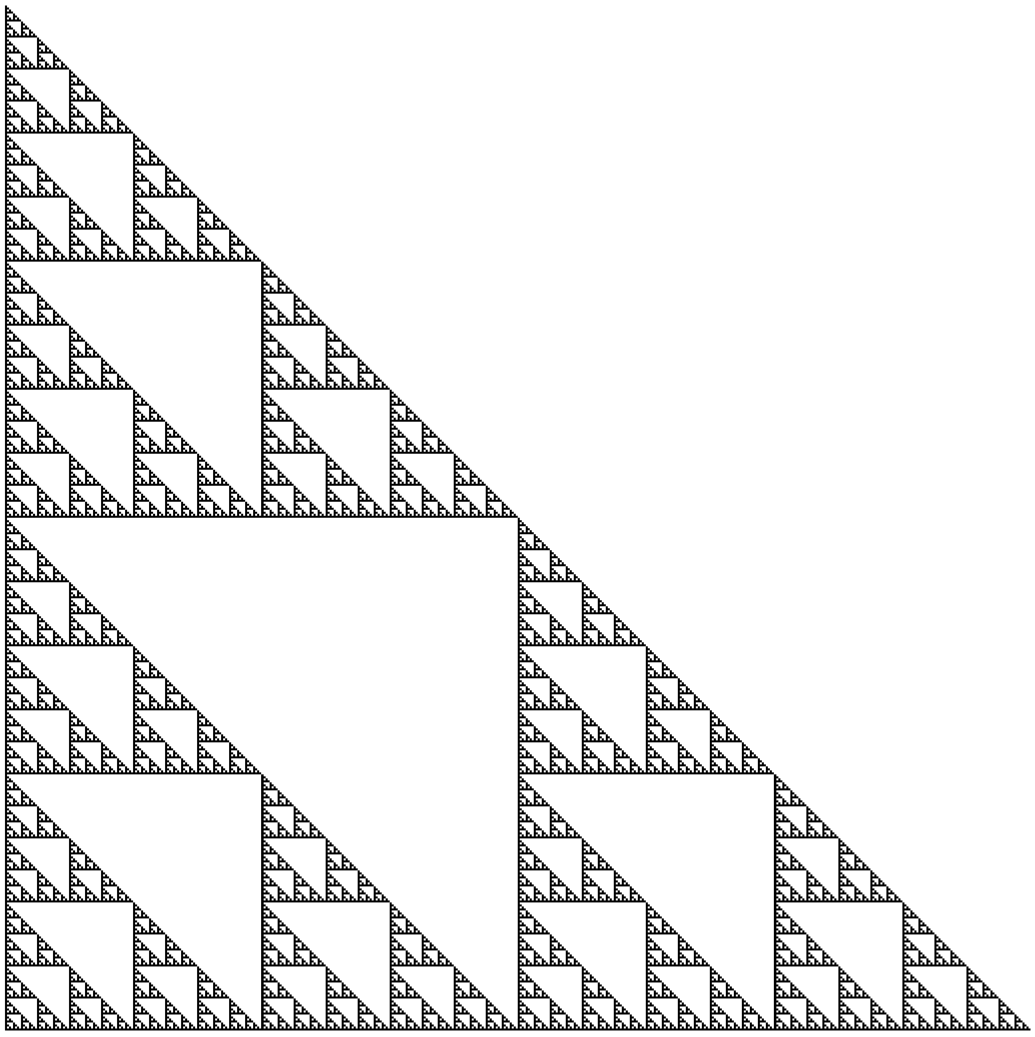
de Rivaz J, *Fractal Report 26*, Vol 5, No 26, April 1993, pp 6-7

Vrscay E R"Mandelbrot Sets for Pairs of Affine Transformations in the Plane", *J. Phys. A: Math. Gen.*, 19, 1986, pp1985-2001

Wood K, "Designing Fractals to Order", *Electronics World + Wireless World*, August 1990, pp 703-708

Ftp: rtfm.mit.edu:/pub/usenet/comp.compression

Appendix A The Sierpinski Triangle



Appendix B Source Code for an Automatic Fractal Image Compression System

B1 Compression Program

```
/* Fractal Image Compression Demonstration */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <alloc.h>

#define IMAGE argv[1]
#define IFS argv[2]

get_image(FILE *ifs_file, unsigned int *assumed_width_of_image, unsigned int
  *assumed_height_of_image, unsigned int *width_of_image, unsigned int *height_of_image, char
                                                                                        *argv[]);
unsigned char domain_block(unsigned long int domx, unsigned char domain[4][4], unsigned int
          assumed_width_of_image, unsigned int width_of_image, unsigned int height_of_image);
unsigned char range_block(unsigned long int ranx, unsigned char ran[8][8], unsigned int
          assumed_width_of_image, unsigned int width_of_image, unsigned int height_of_image);
symmetry(unsigned char ss, unsigned char dom[4][4], unsigned char d1[4][4]);
bestfit(unsigned long int ranx, unsigned char s, unsigned char d1[4][4], unsigned char ran[8][8],
               unsigned long int *bestranx, unsigned char *bestsym, unsigned char *lowest_error);
write_to_file(unsigned long int bestranx, unsigned char bestsym, FILE *ifs_file, unsigned int
                                                          assumed_width_of_image, char *argv[]);
calculate_time(time_t start_time);

unsigned int *image;

main(int argc, char *argv[])
{
   unsigned char dom[4][4], d1[4][4], ran[8][8], lowest_error, sym, bestsym, domain_whites,
                                                                                  range_whites;
   unsigned int assumed_width_of_image, assumed_height_of_image, width_of_image,
                                                                                height_of_image;
   unsigned long int count, domx, domy, ranx, rany, bestranx;
   time_t start_time;
   FILE *ifs_file;

   start_time = time(NULL);
   if (argc != 3) {
      printf("\nUse the following format:\n\ncompress [image_file] [ifs_file]\n\n");
      exit(1);
   }
   if ((ifs_file = fopen(IFS, "wb")) == NULL) {
      fprintf(stderr, "\nError opening file %s\n", IFS);
      exit(1);
   }
   get_image(ifs_file, &assumed_width_of_image, &assumed_height_of_image,
&width_of_image,
                                                                 &height_of_image, argv);
```

```
    printf("\n\nCompressing %s", IMAGE);
    count = 0;
    for (domy = 0; domy < (assumed_width_of_image/4) * assumed_height_of_image; domy +=
                                                                      assumed_width_of_image)
        for (domx = domy; domx < domy + assumed_width_of_image/4; domx++) {
            if (count++ % 200 == 0)
                printf(".");
            lowest_error = 64;
            bestsym = 0;
            domain_whites = domain_block(domx, dom, assumed_width_of_image,
width_of_image,
                                                                             height_of_image);
            if (domain_whites == 16) {
                bestranx = 0;
                bestsym = 8;
                write_to_file(bestranx, bestsym, ifs_file, assumed_width_of_image, argv);
                continue;
            }
            for (rany = 0; rany < (assumed_width_of_image/4) * (assumed_height_of_image - 4);
                                                      rany += assumed_width_of_image) {
                for (ranx = rany; ranx < rany + (assumed_width_of_image/4) - 1; ranx++) {
                    range_whites = range_block(ranx, ran, assumed_width_of_image,
width_of_image,
                                                                             height_of_image);
                    if (range_whites == 64)
                        continue;
                    if (abs(4 * domain_whites - range_whites) > lowest_error)
                        continue;
                    for (sym = 0; sym < 8; sym++) {
                        symmetry(sym, dom, d1);
                        bestfit(ranx, sym, d1, ran, &bestranx, &bestsym, &lowest_error);
                        if (lowest_error == 0)
                            break;
                    }
                    if (lowest_error == 0)
                        break;
                }
                if (lowest_error == 0)
                    break;
            }
            write_to_file(bestranx, bestsym, ifs_file, assumed_width_of_image, argv);
        }
    free(image);
    fclose(ifs_file);
    calculate_time(start_time);
    return 0;
}

get_image(FILE *ifs_file, unsigned int *assumed_width_of_image, unsigned int
  *assumed_height_of_image, unsigned int *width_of_image, unsigned int *height_of_image, char
                                                                                      *argv[])
```

```
{
    FILE *image_file;
    unsigned int buf[24], *header, size_of_header, extra_height;
    unsigned long size_of_file, size_of_image;

    if ((image_file = fopen(IMAGE, "rb")) == NULL) {
        fprintf(stderr, "\nCannot open file %s\n", IMAGE);
        exit(1);
    }
    if (fread(buf, sizeof(unsigned int), 24, image_file) != 24) {
        fprintf(stderr, "\nError reading file %s\n", IMAGE);
        exit(1);
    }
    if (buf[0] != 19778) {
        printf("%s is not a .BMP file", IMAGE);
        exit(0);
    }
    if (buf[23] != 2) {
        printf("%s is not a black and white image", IMAGE);
        exit(0);
    }
    size_of_file = buf[2] * 65536 + buf[1];
    size_of_header = buf[5];
    size_of_image = size_of_file - size_of_header;
    *width_of_image = buf[9];
    if (*width_of_image % 32 != 0)
        *assumed_width_of_image = *width_of_image + 32 - (*width_of_image % 32);
    else *assumed_width_of_image = *width_of_image;
    *height_of_image = buf[11];
    if (*height_of_image%4 !=0)
        *assumed_height_of_image = *height_of_image + 4 - (*height_of_image % 4);
    else *assumed_height_of_image = *height_of_image;
    extra_height = ((*assumed_height_of_image - *height_of_image)
                                                    * *assumed_width_of_image) / 8;
    if (fseek(image_file, 0, SEEK_SET) != 0) {
        fprintf(stderr, "\nError with file %s\n", IMAGE);
        exit(1);
    }
    if ((header = (unsigned int *) calloc(size_of_header/2, sizeof(unsigned int))) == NULL) {
        fprintf(stderr, "\nUnable to allocate sufficient memory for header\n");
        exit(1);
    }
    if (fread(header, sizeof(unsigned int), size_of_header/2, image_file) != size_of_header/2) {
        fprintf(stderr, "\nCannot read header from %s\n", IMAGE);
        exit(1);
    }
    fwrite(header, sizeof(unsigned int), size_of_header/2, ifs_file);
    if ((image = (unsigned int *) calloc((size_of_image + extra_height) / 2, sizeof(unsigned int)))
                                                                        == NULL) {
        fprintf(stderr, "\nUnable to assign enough memory for image\n");
        exit(1);
    }
    if (fread(image, sizeof(unsigned int), size_of_image/2, image_file) != size_of_image/2) {
        fprintf(stderr, "\nError reading file %s\n", IMAGE);
        exit(1);
    }
    fclose(image_file);
    return 0;
}

unsigned char domain_block(unsigned long int dx, unsigned char domain[4][4], unsigned int
            assumed_width_of_image, unsigned int width_of_image, unsigned int height_of_image)
{
    unsigned char i, j, whites = 0;
    unsigned long xi, xa;

    for (xi = dx, i = 0; i < 4 ; xi += assumed_width_of_image / 4, i++) {
        xa = (xi % 4 < 2) ? xi + 2 : xi - 2;
        for (j = 0; j < 4; j++) {
            if (((4 * xi) / assumed_width_of_image < height_of_image) && ((4 * xi + j)
                                                  % assumed_width_of_image < width_of_image))
                domain[i][j] = 1 & (image[xa/4] >> (15 - ((xa % 4) * 4 + j)));
            else domain[i][j] = 1;
            if (domain[i][j] == 1)
                whites++;
        }
    }
    return whites;
}

unsigned char range_block(unsigned long int ranx, unsigned char ran[8][8], unsigned int
            assumed_width_of_image, unsigned int width_of_image, unsigned int height_of_image)
{
    unsigned char i, j, d[4][4], ran_whites = 0;

    ran_whites += domain_block(ranx, d, assumed_width_of_image, width_of_image,
                                                                          height_of_image);
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            ran[i][j] = d[i][j];
    ran_whites += domain_block(ranx + 1, d, assumed_width_of_image, width_of_image,
                                                                          height_of_image);
    for (i = 0; i < 4; i++)
        for (j = 4; j < 8; j++)
            ran[i][j] = d[i][j-4];
    ran_whites += domain_block(ranx + assumed_width_of_image, d, assumed_width_of_image,
                                                          width_of_image, height_of_image);
    for (i = 4; i < 8; i++)
        for (j = 0; j < 4; j++)
            ran[i][j] = d[i-4][j];
    ran_whites += domain_block(ranx+1+assumed_width_of_image, d, assumed_width_of_image,
                                                          width_of_image, height_of_image);
    for (i = 4; i < 8; i++)
        for (j = 4; j < 8; j++)
            ran[i][j] = d[i-4][j-4];
    return ran_whites;
}

symmetry(unsigned char sym, unsigned char dom[4][4], unsigned char d1[4][4])
{
    unsigned char i, j;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            switch (sym) {
            case 0:
                d1[i][j] = dom[i][j];
                break;
            case 1:
                d1[i][j] = dom[3-i][j];
                break;
            case 2:
                d1[i][j] = dom[i][3 - j];
                break;
            case 3:
                d1[i][j] = dom[3 - i][3 - j];
                break;
            case 4:
                d1[i][j] = dom[j][i];
                break;
            case 5:
                d1[i][j] = dom[j][3 - i];
                break;
            case 6:
                d1[i][j] = dom[3 - j][i];
                break;
            case 7:
                d1[i][j] = dom[3 - j][3 - i];
                break;
            }
    return 0;
}


bestfit(unsigned long int ranx, unsigned char s, unsigned char d1[4][4], unsigned char ran[8][8],
                unsigned long int *bestranx, unsigned char *bestsym, unsigned char *lowest_error)
{
    unsigned char i, j, current_error = 0;

    for (i = 0; i < 8 ; i++)
        for (j = 0; j < 8; j++) {
            if (ran[i][j] != d1[i/2][j/2])
                current_error++;
        }
    if (current_error < *lowest_error) {
        *lowest_error = current_error;
        *bestranx = ranx;
        *bestsym = s;
    }
    return 0;
}

write_to_file(unsigned long int bestranx, unsigned char bestsym, FILE *ifs_file, unsigned int
                                                          assumed_width_of_image, char *argv[])
{
    unsigned int abestranx;

    if (putc(bestsym, ifs_file) == EOF) {
        fprintf(stderr, "\nError writing to file %s\n", IFS);
        exit(1);
    }
    if (bestsym != 8) {
        abestranx = bestranx - (3 * assumed_width_of_image / 4 + 1)
                                                        * (bestranx / assumed_width_of_image);
        if (putw(abestranx, ifs_file) == EOF) {
            fprintf(stderr, "\nError writing to file %s\n", IFS);
            exit(1);
        }
    }
    return 0;
}

calculate_time(time_t start_time)
{
    unsigned int hours, mins, secs;
    unsigned long tim;
    time_t finish_time;

    finish_time = time(NULL);
    tim = difftime(finish_time, start_time);
    hours = tim / 3600;
    mins = (tim - 3600 * (tim / 3600)) / 60;
    secs = tim % 60;
    printf("\n\nCompression time: ");
    if (hours == 1)
        printf("%u hour ", hours);
    else if (hours > 1)
        printf("%u hours ", hours);
    if (mins == 1)
        printf("%u minute ", mins);
    else if(mins > 1)
        printf("%u minutes ", mins);
    if (secs == 1)
        printf("%u second", secs);
    else
        printf("%u seconds", secs);
    printf("\n\n");
    return 0;
}
```

B2 Decompression Program

```
/* Fractal Image Decompression Demonstration */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <alloc.h>

#define IFS argv[1]
#define DECOMIMAGE argv[2]
#define NO_OF_ITERATIONS 3

load_header(unsigned int *assumed_width_of_image, unsigned int *height_of_image,
                                 unsigned int *assumed_height_of_image, unsigned int *size_of_header,
                     unsigned long *size_of_image, FILE *ifs_file, FILE *decomp_file, char *argv[]);
initialize_screens(unsigned long size_of_image, unsigned int assumed_width_of_image,
unsigned int height_of_image, unsigned int assumed_height_of_image, unsigned int
*extra_height);
read_ifs_file(unsigned int assumed_width_of_image, unsigned long int *bestranx,
                                                   unsigned char *bests, FILE *ifs_file, char *argv[]);
get_domain_block(unsigned long int dx, unsigned char domain[4][4],
                                                              unsigned int assumed_width_of_image);
get_range_block(unsigned long int ranx, unsigned int assumed_width_of_image,
                                                                           unsigned char ran[8][8]);
replace_domain_block(unsigned int s, unsigned char dom[4][4], unsigned char ran[8][8]);
new_domain_block(unsigned long int dx, unsigned int assumed_width_of_image,
                                                                          unsigned char dom[4][4]);
copy_domain_screen_to_range_screen(unsigned long int size_of_image, unsigned int
extra_height);
copy_domain_screen_to_decomp_file(unsigned long size_of_image, FILE *decomp_file);
calculate_time(time_t start_time);

unsigned int *domain_screen, *range_screen;

main(int argc, char *argv[])
{
   unsigned char iter, i, j, bests, dom[4][4], ran[8][8];
   unsigned int assumed_width_of_image, height_of_image, assumed_height_of_image,
                                                                    size_of_header, extra_height;
   unsigned long count, size_of_image, domx, domy, bestranx;
   time_t start_time;
   FILE *ifs_file, *decomp_file;

   start_time = time(NULL);

   if (argc != 3) {
      printf("\nUse the following format:\n\ndecompress [ifs_file] [image_file]\n\n");
      exit(1);
   }
   if ((ifs_file = fopen(IFS, "rb")) == NULL) {
      printf("\nCannot open file %s\n", IFS);
      exit(1);
   }
   if ((decomp_file = fopen(DECOMIMAGE, "wb")) == NULL) {
      fprintf(stderr, "\nCannot open file %s\n", DECOMIMAGE);
      exit(1);
   }
```

```
    load_header(&assumed_width_of_image, &height_of_image, &assumed_height_of_image,
                                        &size_of_header, &size_of_image, ifs_file, decomp_file, argv);
    initialize_screens(size_of_image, assumed_width_of_image, height_of_image,
                                                        assumed_height_of_image, &extra_height);
    printf("\n\nDecompressing %s", IFS);
    count = 0;
    for(iter = 0; iter < NO_OF_ITERATIONS; iter++) {
        fseek(ifs_file, size_of_header, SEEK_SET);
        for (domy = 0; domy < (assumed_width_of_image/4) * assumed_height_of_image;
                                                             domy += assumed_width_of_image)
            for (domx = domy; domx < domy + assumed_width_of_image/4; domx++) {
                if (count++ % (200 * NO_OF_ITERATIONS) == 0)
                    printf(".");
                read_ifs_file(assumed_width_of_image, &bestranx, &bests, ifs_file, argv);
                if (bests != 8) {
                    get_range_block(bestranx, assumed_width_of_image, ran);
                    replace_domain_block(bests, dom, ran);
                }
                else for (i = 0; i < 4; i++)
                        for (j = 0; j < 4; j++)
                            dom[i][j] = 1;
                new_domain_block(domx, assumed_width_of_image, dom);
            }
        if (iter < NO_OF_ITERATIONS - 1) {
            copy_domain_screen_to_range_screen(size_of_image, extra_height);
        }
    }
    fclose(ifs_file);
    free(range_screen);
    copy_domain_screen_to_decomp_file(size_of_image, decomp_file);
    fclose(decomp_file);
    free(domain_screen);
    calculate_time(start_time);
    return 0;
}

load_header(unsigned int *assumed_width_of_image, unsigned int *height_of_image,
                                  unsigned int *assumed_height_of_image, unsigned int *size_of_header,
                      unsigned long *size_of_image, FILE *ifs_file, FILE *decomp_file, char *argv[])
{
    unsigned int buf[24], *header, width_of_image;
    unsigned long size_of_file;

    if (fread(buf, sizeof(unsigned int), 24, ifs_file) != 24) {
        fprintf(stderr, "\nError reading file %s\n", IFS);
        exit(1);
    }
    if (buf[0] != 19778) {
        printf("%s is not an .IFS file", IFS);
        exit(0);
    }
    if (buf[23] != 2) {
        printf("%s is not a black and white image", IFS);
        exit(0);
    }
    size_of_file = buf[2] * 65536 + buf[1];
    *size_of_header = buf[5];
    *size_of_image = size_of_file - *size_of_header;
    width_of_image = buf[9];
    if (width_of_image % 32 != 0)
        *assumed_width_of_image = width_of_image + 32 - (width_of_image % 32);
    else *assumed_width_of_image = width_of_image;
    *height_of_image = buf[11];
    if (*height_of_image % 4 != 0)
        *assumed_height_of_image = *height_of_image + 4 - (*height_of_image % 4);
    else *assumed_height_of_image = *height_of_image;
    fseek(ifs_file, 0, SEEK_SET);
    if ((header = (unsigned int *) calloc(*size_of_header/2, sizeof(unsigned int))) == NULL) {
        fprintf(stderr, "\nUnable to allocate sufficient memory for header\n");
        exit(1);
    }
    if (fread(header, sizeof(unsigned int), *size_of_header/2, ifs_file) != *size_of_header/2) {
        fprintf(stderr, "\nCannot read header from %s\n", IFS);
        exit(1);
    }
    if (fwrite(header, 1, *size_of_header, decomp_file) != *size_of_header) {
        printf("\nCannot write header\n");
        exit(1);
    }
    free(header);
    return 0;
}

initialize_screens(unsigned long size_of_image, unsigned int assumed_width_of_image,
                  unsigned int height_of_image, unsigned int assumed_height_of_image, unsigned int
                                                                                  *extra_height)
{
    *extra_height = ((assumed_height_of_image - height_of_image) *
assumed_width_of_image)/8;
    if ((domain_screen = (unsigned int *) calloc((size_of_image + *extra_height)/2,
                                                            sizeof(unsigned int))) == NULL) {
        fprintf(stderr, "\nUnable to allocate sufficient memory for domain screen\n");
        exit(1);
    }
    if ((range_screen = (unsigned int *) calloc((size_of_image + *extra_height)/2,
                                                            sizeof(unsigned int))) == NULL) {
        fprintf(stderr, "\nUnable to allocate sufficient memory for range screen\n");
        exit(1);
    }
    return 0;
}
```

```
read_ifs_file(unsigned int assumed_width_of_image, unsigned long int *bestranx,
                                                unsigned char *bests, FILE *ifs_file, char *argv[])
{
    unsigned int abestranx;

    if (fread(bests, sizeof(unsigned char), 1, ifs_file) != 1) {
        fprintf(stderr, "\nError reading file %s\n", IFS);
        exit(1);
    }
    if (*bests != 8) {
        abestranx = getw(ifs_file);
        if (ferror(ifs_file)) {
            fprintf(stderr, "\nError reading file %s\n", IFS);
            exit (1);
        }
        *bestranx = assumed_width_of_image * ((4 * (abestranx)) / (assumed_width_of_image -
4))
                                                + (abestranx % ((assumed_width_of_image - 4) / 4));
    }
    return 0;
}

get_domain_block(unsigned long int dx, unsigned char domain[4][4],
                                                        unsigned int assumed_width_of_image)
{
    unsigned char i, j;
    unsigned long int xi;

    dx = (dx % 4 < 2) ? dx + 2 : dx - 2;
    for (xi = dx, i = 0; i < 4 ; xi += assumed_width_of_image / 4, i++)
        for (j = 0; j < 4; j++)
            domain[i][j] = 1 & (range_screen[xi/4] >> (15 - ((xi % 4) * 4 + j)));
    return 0;
}

get_range_block(unsigned long int ranx, unsigned int assumed_width_of_image,
                                                                        unsigned char ran[8][8])
{
    unsigned char i, j, d[4][4];

    get_domain_block(ranx, d, assumed_width_of_image);
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            ran[i][j] = d[i][j];
    get_domain_block(ranx + 1, d, assumed_width_of_image);
    for (i = 0; i < 4; i++)
        for (j = 4; j < 8; j++)
            ran[i][j] = d[i][j-4];
    get_domain_block(ranx + assumed_width_of_image, d, assumed_width_of_image);
    for (i = 4; i < 8; i++)
        for (j = 0; j < 4; j++)
```

```
            ran[i][j] = d[i-4][j];
    get_domain_block(ranx+1+assumed_width_of_image, d, assumed_width_of_image);
    for (i = 4; i < 8; i++)
        for (j = 4; j < 8; j++)
            ran[i][j] = d[i-4][j-4];
    return 0;
}

replace_domain_block(unsigned int s, unsigned char dom[4][4], unsigned char ran[8][8])
{
    unsigned char i, j;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++) {
            switch (s) {
            case 0:
                dom[i][j] = ran[2*i][2*j] + ran[2*i][2*j+1] + ran[2*i+1][2*j] + ran[2*i+1][2*j+1];
                break;
            case 1:
                dom[i][j] = ran[7-2*i][2*j] + ran[7-2*i][2*j+1] + ran[7-(2*i+1)][2*j]
                                                                + ran[7-(2*i+1)][2*j+1];
                break;
            case 2:
                dom[i][j] = ran[2*i][7-2*j] + ran[2*i][7-(2*j+1)] + ran[2*i+1][7-2*j]
                                                                + ran[2*i+1][7-(2*j+1)];
                break;
            case 3:
                dom[i][j] = ran[7-2*i][7-2*j] + ran[7-2*i][7-(2*j+1)] + ran[7-(2*i+1)][7-2*j]
                                                            + ran[7-(2*i+1)][7-(2*j+1)];
                break;
            case 4:
                dom[i][j] = ran[2*j][2*i] + ran[2*j][2*i+1] + ran[2*j+1][2*i]+ran[2*j+1][2*i+1];
                break;
            case 5:
                dom[i][j] = ran[2*j][7-2*i] + ran[2*j][7-(2*i+1)] + ran[2*j+1][7-2*i]
                                                                + ran[2*j+1][7-(2*i+1)];
                break;
            case 6:
                dom[i][j] = ran[7-2*j][2*i] + ran[7-2*j][2*i+1] + ran[7-(2*j+1)][2*i]
                                                                + ran[7-(2*j+1)][2*i+1];
                break;
            case 7:
                dom[i][j] = ran[7-2*j][7-2*i] + ran[7-2*j][7-(2*i+1)] + ran[7-(2*j+1)][7-2*i]
                                                            + ran[7-(2*j+1)][7-(2*i+1)];
                break;
            }
    if (dom[i][j] < 2)
        dom[i][j] = 0;
    else if (dom[i][j] > 2)
        dom[i][j] = 1;
    else
```

```
        dom[i][j] = rand() % 2;
    }
    return 0;
}

new_domain_block(unsigned long int dx, unsigned int assumed_width_of_image,
                                                        unsigned char dom[4][4])
{
    unsigned char i;
    unsigned int xi;

    union {
        unsigned int word;
        struct {
            unsigned int bit0 : 1;
            unsigned int bit1 : 1;
            unsigned int bit2 : 1;
            unsigned int bit3 : 1;
            unsigned int bit4 : 1;
            unsigned int bit5 : 1;
            unsigned int bit6 : 1;
            unsigned int bit7 : 1;
            unsigned int bit8 : 1;
            unsigned int bit9 : 1;
            unsigned int bit10 : 1;
            unsigned int bit11 : 1;
            unsigned int bit12 : 1;
            unsigned int bit13 : 1;
            unsigned int bit14 : 1;
            unsigned int bit15 : 1;
        } bits;
    } temp;

    for (xi = dx, i = 0; i < 4 ; xi += assumed_width_of_image/4, i++) {
        temp.word = domain_screen[xi/4];
        if (xi % 4 == 0) {
                temp.bits.bit4 = dom[i][3];
                temp.bits.bit5 = dom[i][2];
                temp.bits.bit6 = dom[i][1];
                temp.bits.bit7 = dom[i][0];
        }
        if (xi % 4 == 1) {
                temp.bits.bit0 = dom[i][3];
                temp.bits.bit1 = dom[i][2];
                temp.bits.bit2 = dom[i][1];
                temp.bits.bit3 = dom[i][0];
        }
        if (xi % 4 == 2) {
                temp.bits.bit12 = dom[i][3];
                temp.bits.bit13 = dom[i][2];
                temp.bits.bit14 = dom[i][1];
                temp.bits.bit15 = dom[i][0];
            }
            if (xi % 4 == 3) {
                temp.bits.bit8 = dom[i][3];
                temp.bits.bit9 = dom[i][2];
                temp.bits.bit10 = dom[i][1];
                temp.bits.bit11 = dom[i][0];
            }
            domain_screen[xi/4] = temp.word;
        }
        return 0;
}

copy_domain_screen_to_range_screen(unsigned long int size_of_image, unsigned int
extra_height)
{
    unsigned long int i;

    for (i = 0; i < (size_of_image + extra_height) / 2; i++)
        range_screen[i] = domain_screen[i];
    return 0;
}

copy_domain_screen_to_decomp_file(unsigned long size_of_image, FILE *decomp_file)
{
    fwrite(domain_screen, sizeof(unsigned int), size_of_image/2, decomp_file);
    return 0;
}

calculate_time(time_t start_time)
{
    unsigned int hours, mins, secs;
    unsigned long tim;
    time_t finish_time;

    finish_time = time(NULL);
    tim = difftime(finish_time, start_time);
    hours = tim / 3600;
    mins = (tim - 3600 * (tim / 3600)) / 60;
    secs = tim % 60;
    printf("\n\nDecompression time: ");
    if (hours == 1)
        printf("%u hour ", hours);
    else if (hours > 1)
        printf("%u hours ", hours);
    if (mins == 1)
        printf("%u minute ", mins);
    else if(mins > 1)
        printf("%u minutes ", mins);
    if (secs == 1)
        printf("%u second", secs);
    else
        printf("%u seconds", secs);
    printf("\n\n");
    return 0;
}
```

B3 Error Measurement Program

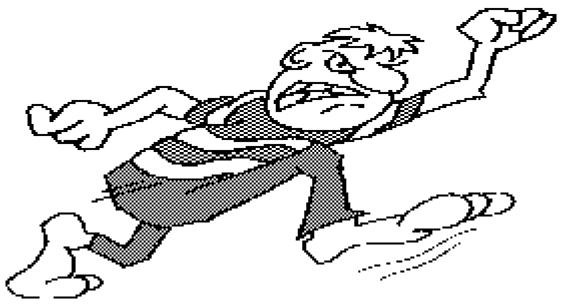
```
/* Comparison of two .BMP images */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <io.h>
```

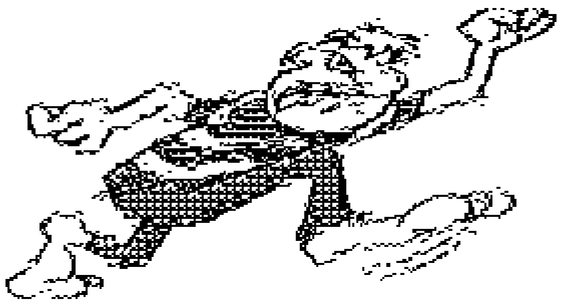
```
#define IMAGE1 argv[1]
#define IMAGE2 argv[2]

main(int argc, char *argv[])
{
    int handle1, handle2;
    unsigned int buf[24], image1_header[BUFSIZ], image2_header[BUFSIZ], size_of_header, i, j,
                                                                                        *image1, *image2;
    unsigned long int size_of_file, size_of_image, error, x, y, nx, width_of_image,
                                                  assumed_width_of_image, height_of_image;
    FILE *image_file_1, *image_file_2;

    if (argc != 3) {
        printf("\nUse the following format:\n\ncompress [image_file_1] [image_file_2]\n\n");
        exit(1);
    }
    if ((image_file_1 = fopen(IMAGE1, "rb")) == NULL) {
        fprintf(stderr, "Cannot open file %s\n", IMAGE1);
        exit(1);
    }
    if (fread(buf, sizeof(unsigned int), 24, image_file_1) != 24) {
        fprintf(stderr, "Error reading file %s\n", IMAGE1);
        exit(1);
    }
    if (buf[0] != 19778) {
        printf("%s is not a .BMP file", IMAGE1);
        exit(0);
    }
    if (buf[23] != 2) {
        printf("%s is not a black and white image", IMAGE1);
        exit(0);
    }
    size_of_file = buf[2] * 65536 + buf[1];
    size_of_header = buf[5];
    size_of_image = size_of_file - size_of_header;
    width_of_image = buf[9];
    if (width_of_image % 32 != 0)
        assumed_width_of_image = width_of_image + 32 - (width_of_image % 32);
    else assumed_width_of_image = width_of_image;
    height_of_image = buf[11];
    if (fseek(image_file_1, 0, SEEK_SET) != 0) {
        fprintf(stderr, "Error with file %s\n", IMAGE1);
        exit(1);
    }
    if (fread(image1_header, sizeof(unsigned int), size_of_header/2, image_file_1)
                                                                     != size_of_header/2) {
        fprintf(stderr, "Cannot read header from %s\n", IMAGE1);
        exit(1);
    }
    if ((image_file_2 = fopen(IMAGE2, "rb")) == NULL) {
```

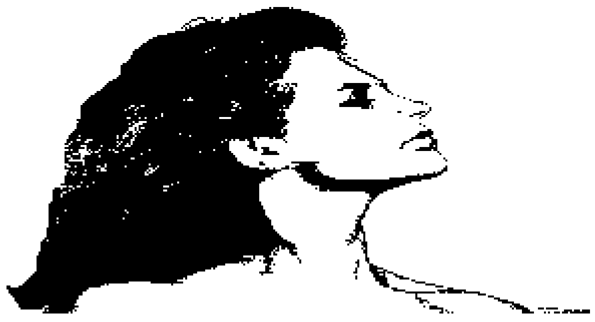
```
    fprintf(stderr, "Cannot open file %s\n", IMAGE2);
    exit(1);
}
if (fread(image2_header, sizeof(unsigned int), size_of_header/2, image_file_2)
                                                                    != size_of_header/2) {
    fprintf(stderr, "Cannot read header from %s\n", IMAGE2);
    exit(1);
}
if (image2_header[0] != 19778) {
    printf("%s is not a .BMP file", IMAGE2);
    exit(0);
}
if (image2_header[23] != 2) {
    printf("%s is not a black and white image", IMAGE2);
    exit(0);
}
if ((handle1 = open(IMAGE1, O_RDONLY)) == -1) {
    printf("Cannot open file %s\n", IMAGE1);
    exit(1);
}
if ((handle2 = open(IMAGE2, O_RDONLY)) == -1) {
    printf("Cannot open file %s\n", IMAGE2);
    exit(1);
}
if (filelength(handle1) != filelength(handle2)) {
    printf("%s and %s are different sized files\n", IMAGE1, IMAGE2);
    exit(0);
}
close(handle1);
close(handle2);
for (i = 0; i < size_of_header/2; i++)
    if (image1_header[i] != image2_header[i]) {
        printf("%s and %s have different headers\n\n", IMAGE1, IMAGE2);
        exit(0);
    }
if ((image1 = (unsigned int *) calloc(size_of_image/2, sizeof(unsigned int))) == NULL) {
    fprintf(stderr, "Unable to assign enough memory for %s\n", IMAGE1);
    exit(1);
}
if ((image2 = (unsigned int *) calloc(size_of_image/2, sizeof(unsigned int))) == NULL) {
    fprintf(stderr, "Unable to assign enough memory for %s\n", IMAGE2);
    exit(1);
}
if (fread(image1, size_of_image/2, sizeof(unsigned int), image_file_1) == NULL)
    fprintf(stderr, "Error reading file %s\n", IMAGE1);
if (fread(image2, size_of_image/2, sizeof(unsigned int), image_file_2) == NULL)
    fprintf(stderr, "Error reading file %s\n", IMAGE1);
fclose(image_file_1);
fclose(image_file_2);
error = 0;
for (y = 0; y < height_of_image * assumed_width_of_image; y+=assumed_width_of_image)
```

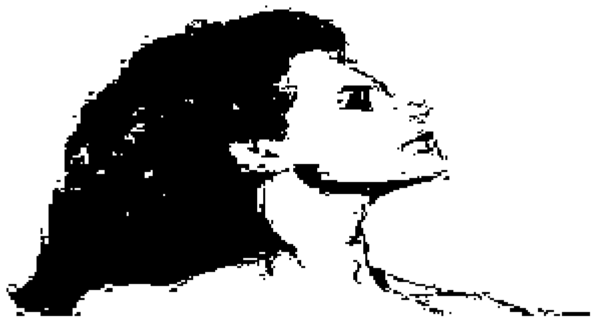
```
        for (x = y; x < y + width_of_image; x++) {
            nx = (x%16 < 8) ? x+8 : x-8;
            error += ((image1[nx/16] >> (15 - nx%16)) ^ (image2[nx/16] >> (15 - nx%16))) & 1;
        }
    free(image1);
    free(image2);
    printf("total number of pixels: %ld\n", width_of_image * height_of_image);
    printf("number of incorrect pixels: %ld\n", error);
    printf("percentage error: %u%%\n\n", (100 * error + (width_of_image * height_of_image) / 2)
                                                    / (width_of_image * height_of_image));
    return 0;
}
```

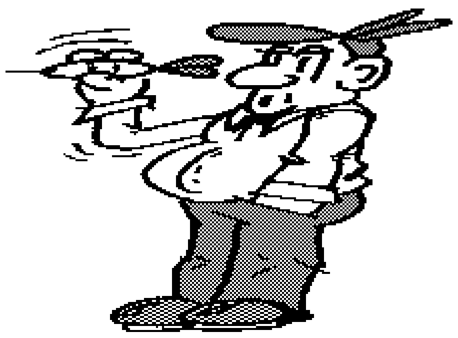
Appendix C Examples of Original and Decompressed Images

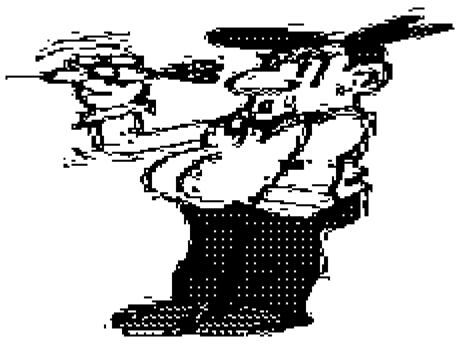

angry.bmp


deangry.bmp


breeze.bmp

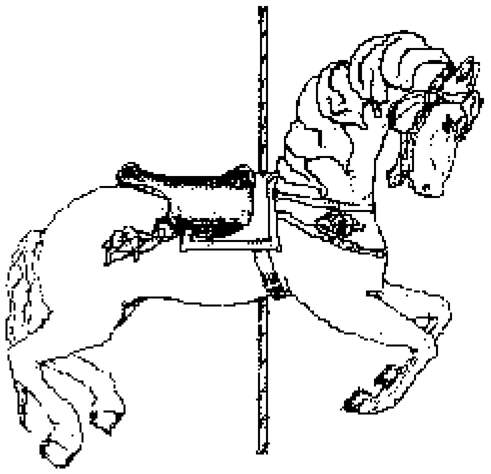

debreeze.bmp

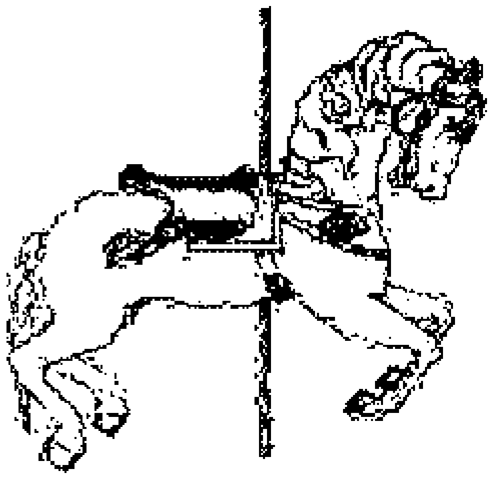

darts.bmp

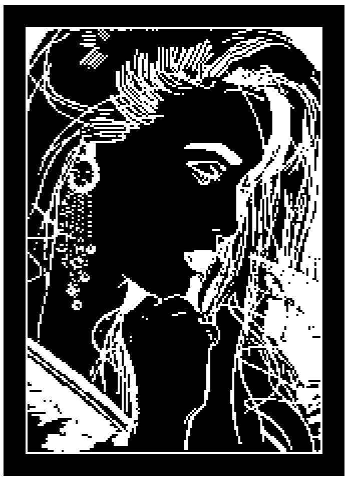

dedarts.bmp

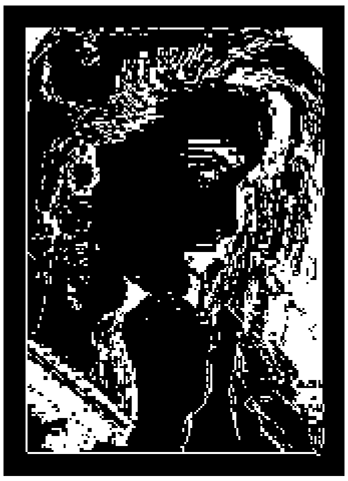
fgrpoint.bmp

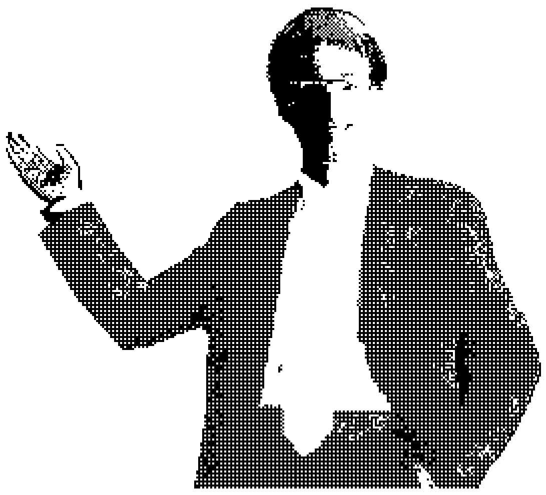
defgrpoi.bmp


horse.bmp


dehorse.bmp


lady.bmp


delady.bmp



man.bmp

deman.bmp