FRACTAL IMAGE COMPRESSION

by

BRUNO LACROIX

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfilment of

the requirements for the degree of

Master of Science

Information and Systems Science

Department of Mathematics and Statistics

Carleton University

Ottawa, Ontario

April, 1998

0-612-36939-0

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my supervisor, Professor Brian Mortimer, for providing me with the right balance of freedom and guidance throughout this research. My thanks are extended to the Department of Mathematics and Statistics at Carleton University for providing a constructive environment for graduate work, and financial assistance.

I wish to express my gratitude to my wife, Martine, for her constant encouragement, patience and support during the course of my graduate studies.

ABSTRACT

This thesis examines the theory of fractal image compression and gives a survey of such techniques for still images and video sequences. Particular attention is given to *partition iterated function systems (pif)*, but *recurrent iterated function systems (rif)* are also discussed. We begin with a discussion of iterated function systems and there applications in image compression and then go on to provide the theoretical basis for such systems, as well as for *pif* and *rif*. In Chapter 3 we discuss different image models as well as the encoding of images using *pif* and *rif*. Different modifications to the brute force *pif* are then discussed in Chapter 4, and we conclude the thesis by presenting four methods which extend the theory of *pif* to video sequences.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In this Chapter, we will discuss the need for fractal image compression techniques. We will also introduce fractals, the Brute Force fractal image compression method and discuss some of its disadvantages. We go on to introduce the history of fractal image compression research and conclude the Chapter with an insight of the following Chapters.

1.1: Why Image Compression

With the Internet growing at an exponential rate, the amount of people transferring digital data from one site to another has increased dramatically. There is now, more than ever, a need for quick data transfer methods and more efficient use of memory space. Unfortunately, images are known for requiring significantly large amounts of memory and hence are not transmittable quickly. This is where fractal image compression techniques come to the rescue. Such techniques allow one to store an image with much less memory than it would normally require, hence allowing it to be transmitted more quickly. For instance, the image in Figure 1.1 requires 263 246 bytes of memory[1], and would take 9.14 seconds to transmit at a data rate of 28.8 Kb/s. However, once this image has been fractally encoded it would require, on average, 189 538 bytes of memory and could thus be transferred in 6.58 seconds, at the same data rate.

[1] At a resolution of 512×512, and a pixel resolution of 256

Figure 1.1: The original image of Lena.

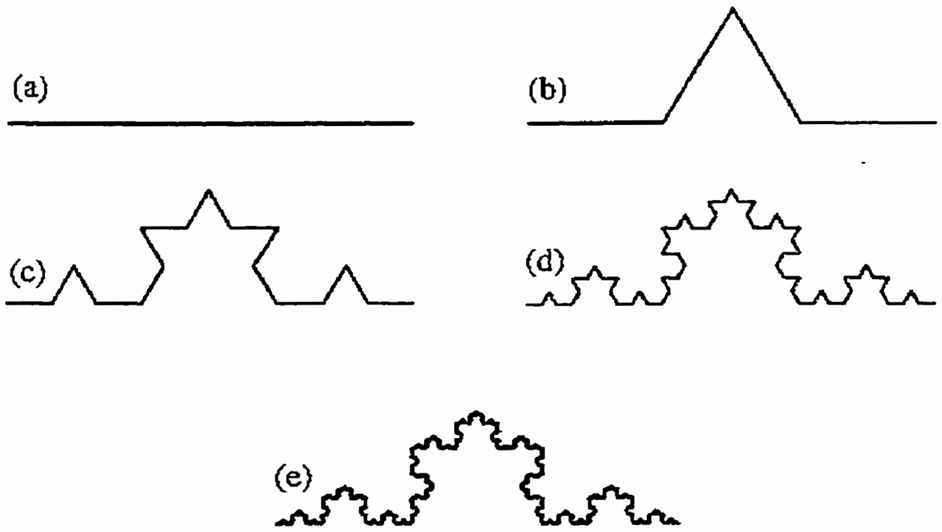
Although fractal image compression was only discovered ten years ago, most fractal image.compression techniques manage to compress a natural image at compression rates between 20:1 and 40:1. This is compatible to the leading techniques used today, which have been around for much longer. Before we explain how fractal image compression techniques work, let us first consider fractals.

A fractal is a geometrical object who's parts are similar to the whole and have infinite resolution. In other words, if we were to expand any part of a fractal, at any ratio, it would never become blurry and one could always see the same original image in that expanded part. For instance, let us consider an image created in the following way: Begin with a straight line divided into three equal parts, with the middle part create an equilateral triangle[3], and then delete its base. Your image should look like the one in Figure 1.2b. With each straight line in the image, repeat the above process ad infinitum. Doing so will create the image in Figure 1.2e, where Figures 1.2a though 1.2d show some of the intermediate steps in achieving the final image. This image is known as the *Koche Curve.* If we had repeated this process on each part of an equilateral triangle, we would have produced the Koche snowflake shown in Figure 1.3[4].

Notice that in the magnification of any part of the Koche curve, we can see parts which are identical to the whole Koche curve. Also, no matter how much we magnify the image, the edges will never be smooth, ie: The Koche Curve has infinite resolution.

[3] An equilateral triangle is a triangle who's sides are all of equal length

[4] The Koche snowflake is a rather peculiar object since it can be contained within a circle of radius one, but has an infinite perimeter.



Figures 1.2a,b,c,d and e: The different steps in the construction of the Koche curve.

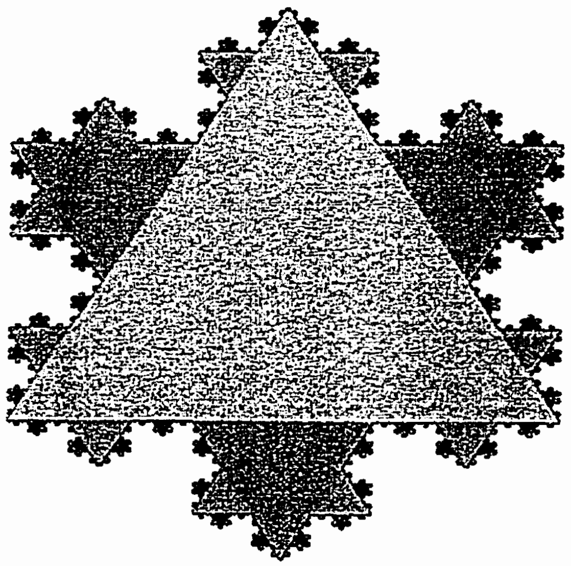


Figure 1.3: The Koche snowflake.

It is important to notice that although Figure 1.2 is quite complex, it can be created extremely easily. Thus, if one wanted to store this image in memory, it would be much more efficient to store the mechanism by which it is created, than it would be to store the image itself. Storing the construction mechanism also allows one to recreate the image at any give resolution, whereas, the stored image would be of a fixed resolution.

1.2: The Brute Force Fractal Image Compression Technique

Geometrical objects created like the ones above are called artificial fractals. Although, naturally occurring objects cannot be created with such techniques, there are some which resemble fractals. For instance, consider the sea coast of an island with a rocky shore. We can magnify the shore at almost any scale and still see a rocky shore type structure, that is until we get to the molecular level of the rocks. Such natural objects are known as natural fractals.

Since the human eye has difficulty distinguishing between certain natural and artificial fractals, we can exploit this for the use of image compression. Fractal image compression algorithms try to find the reconstruction method of an image which is extremely similar to the original one, so as to more efficiently store it. The difference between the reconstructed image and the original can be so small that they appear to be identical to the human observer.

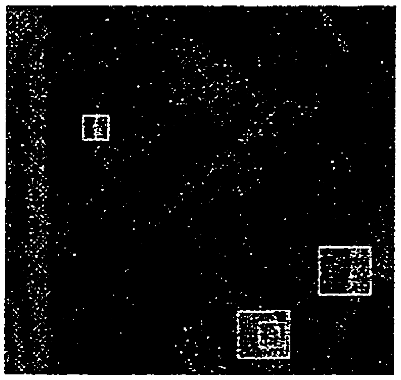


Figure 1.4: The self-similar parts of Lena

Unlike the Koche Curve natural images are not usually self-similar to the whole. However, they usually contain different parts which are self-similar, an example of which is shown in Figure 1.4. Fractal image compression methods take advantage of these self-similar parts by considering them to be identical. They then find a mechanism for reconstructing the image based on the knowledge of the self-similar parts. For a better understanding of how this works, consider the following fractal image compression method for a 256×256 pixel image.

1: Assuming the image is grey-scale, we can then assign an integer value in the range $[0, 255]$ to each pixel.

2: Divide the image into non-overlapping Range blocks of size 8×8 pixels. We therefore have $32 \times 32 = 1024$ range blocks. Let us denote the collection of all such range blocks by $\mathbf{R}$.

3: Also divide the image into possibly overlapping Domain blocks of size 16×16 pixels. We therefore have $241 \times 241 = 58\ 081$ domain blocks.

4: For each range block R_i, find the domain block D_j which, physically, resembles it the most. For a better image quality, you must compare each orientation of the block and its mirror image. Thus performing eight comparisons for each domain block.

It is important to notice that if each range block R_i is compared to each domain block D_j, then you will perform $8 \times 58081 \times 1024 = 475\ 799\ 552$ comparisons in all, or 464 648 comparisons for each range block.

We now have a mapping of a domain block to a range block for each range block in the image. Once this is done, we need only store the mappings for each range block in order to reconstruct the image. Storing such data requires much less information than storing each pixel value in the image. For example, the 256×256 image of Lena requires 65 536 bytes of data, while the mappings only need 3968 bytes, thus resulting in a compression ratio of $16.5 : 1$.

In order to recreate the image, one must choose an arbitrary image to begin with (it may simply be a black image), then iterate the image given the mappings stored for each range block. As Figure 1.4 shows, each iteration will add detail at a finer and finer level, starting at the 8×8 level, then $4 \times 4, 2 \times 2, etc....$ Such a system is known as a *Partitioned (or Local) Iterated Function system*, and is discussed in detail in the following Chapters.

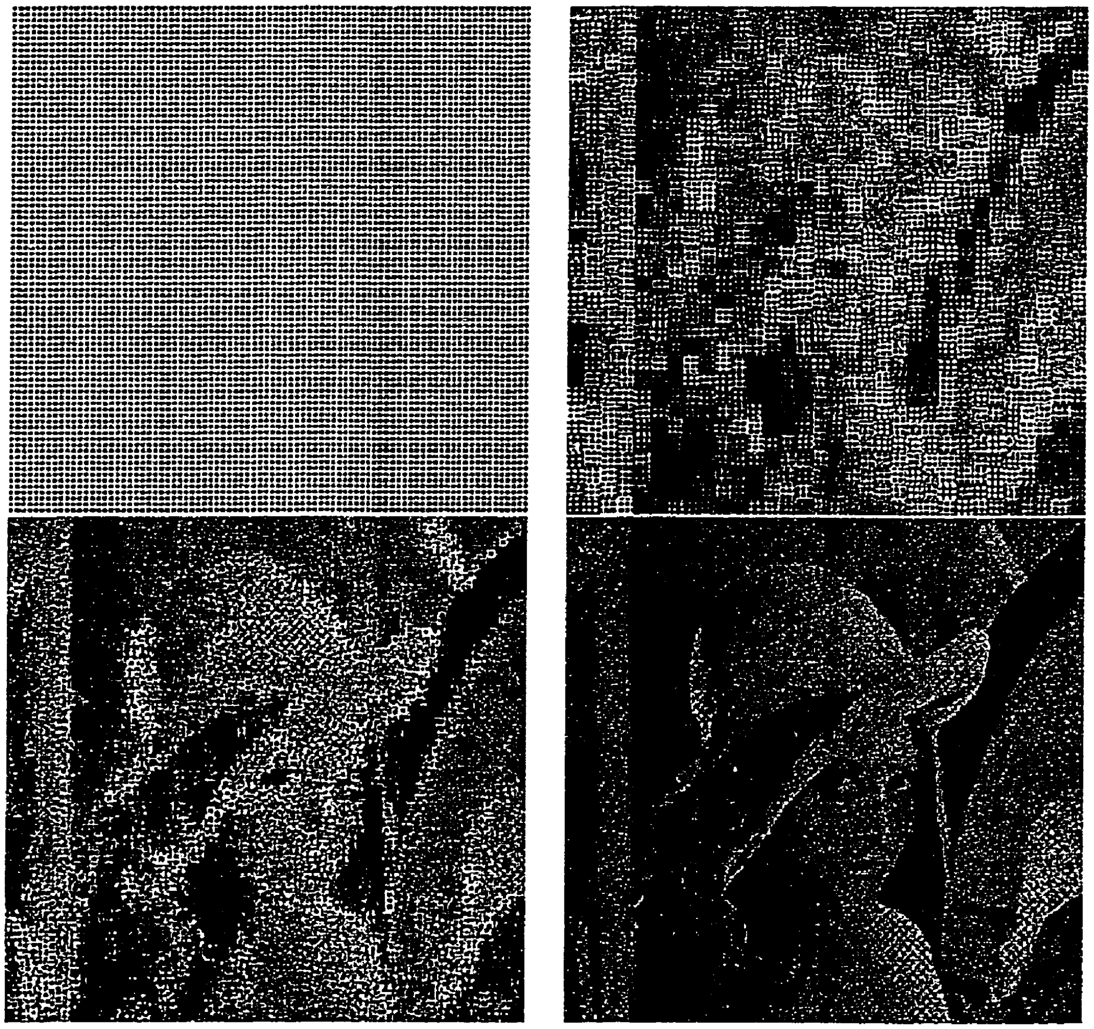


Figure 1.5: The initial image (top left), the first iterate (top right), the second iterate (bottom left) and the tenth iterate (bottom right).

Although the resulting image is of an acceptable quality, given the simplicity of the algorithm, it requires an enormous amount of time to find the domain-range block pair. Most of the work on fractal image compression, has been to decrease the number of comparisons needed and as such, decrease the time needed to encode the image.

Since such images are created in a fractal like manner, they have infinite resolution. However, the resolution created at higher levels is artificial, i.e., one cannot magnify a part of the image and see details that were not in the original. Where other image types result in a blocky effect when the image is magnified, since the value of the pixels is simply extended to the larger area. Fractal image compression does not, since the mappings go on to create artificial resolution.

Once we have the reconstruction mechanism for an image, it is extremely easy and quick to reconstruct it. However, finding the optimal, or suitable, encoding of an image requires a significant amount of time and resources. Therefore, fractal image compression techniques are better suited for applications where the image is encoded once and decoded many times. An example of such a situation is the 1992 version of the Microsoft Encarta compact disk, where all of the images in it are fractals. In order to fit the numerous images on the cd, Microsoft decided to compress them using fractal image compression techniques.

Although most of the research in the past was concentrated on still image compression, video image compression is also becoming extremely popular. This is mainly due to the fact that a suitable video image compression technique can have extremely large economical benefits. Many companies would rather use video-conferencing instead of tele-conferencing. However, because video sequences require large amounts of data, they must be compressed prior to transmitting them. They must also be encoded, transmitted and decoded at such a rate so that the decoded video sequences are not jumpy and still have a suitable image quality. This will be the topic discussed in Chapter 5 of this dissertation.

1.3: The Origins of Fractal Image Compression

The person responsible for discovering fractal image compression is *Micheal F. Barnsley*, who noticed that extremely complicated, naturally appearing objects can be created using transformations with very short codes. He went on to name such transformations *Iterated Function Systems, (IFS)*. IFS's can be used to encode images whose parts resemble the whole. However, as we have mentioned before, most natural images do not contain this type of self-similarity, but rather parts of the image resemble other parts. Barnsley knew this, and in 1988 he generalized the theory of IFS's to *Partitioned Iterated Function Systems (PIFS)*. He used an algorithm similar to the one above to compress an image. However a person had to interact with the program in order to find the domain blocks to be matched to a particular range block. This resulted in high compression ratios, but very poor quality in the decoded images. Much of Barnsley's work remains secret and copyright since he and Sloan founded the company *Iterated Systems Inc* in 1987.

Only in 1990 did any ground breaking work in fractal image compression become public. This was the result of Arnaud Jacquin's Ph.D thesis which automated the search of the domain-range pair. Jacquin, who was Barnsley's Ph.D student, achieved this by restricting the domains and range blocks to a fixed size, and transformations of a particular type, which is discussed in Chapter 2. His algorithm is basically the one described in Section 1.1, with the added capability of adjusting the grey-value of a pixel by one and scaling by a number less than one. He also decreased the number of domain blocks to be searched by classifying them into one of a few categories. For instance, one category was those blocks with a distinctive edge.

1.4: What we will cover in this thesis

Since then, many people have revised Jaquin's method in a variety of ways. In this thesis, we will consider some of the better variations and suggest further improvements. Chapter 2 provides the mathematical background and analysis to understand the reasons such systems work. Chapter 3 introduces different image models and discusses the representation of images using *ifs's, rifs's* and *pifs's*. Chapter 4 introduces many different fractal image compression techniques that are superior to the brute force approach discussed above. Chapter 5 concludes the thesis with a discussion of different techniques used for fractal video compression.

1.5: Conclusion

In this Chapter, we have shown the reasons such compression techniques are needed and taken a brief look at how they work. We also discussed the different applications in which they can be used and have shown some of the history of fractal image compression research. We concluded with a summary of the following Chapters in this thesis.

CHAPTER 2

Iterated Function Systems and the Contractive Mapping Fixed Point Theorem

In this Chapter we will introduce contractive *Iterated Function Systems* and show how to manipulate them so as to create *fractals*. Then we will give a fairly detailed analysis of the properties of such systems and prove the *Generalized Collage Theorem*. After which, we will introduce *Recurrent Iterated Function Systems* and determine the properties of such systems. We will conclude with an introduction of *Partitioned Iterated Function Systems*.

2.1: Iterated Function Systems

An iterated function system is a system of functions which are simultaneously being iterated over a set in $\Re^2$ *st* $F : \Re^2 \rightarrow \Re^2$. When iterated, if the system approaches a fixed point[1], then the graph of the system is called a *fractal*. A fractal, generally speaking, is a geometrical object which has infinite resolution and self similarity at every scale. For a more in depth discussion of fractals, the reader is recommended to read [1].

Example 2.1: Consider the functions defined by

$$F_1\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$$

$$F_2\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1/2 \\ 0 \end{pmatrix}$$

$$F_3\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1/4 \\ 1/2 \end{pmatrix}$$

[1] A fixed point is a point which does not change value when iterated over a function, i.e., $F(\mathbf{x}) = \mathbf{x}$

An interesting characteristic of this system, is that no matter what initial image we begin with, the iterates of the system will always converge to the same final image, i.e., the graph of the system will converge to a fixed point in $\Re^2$. The final image (the fixed point) is called the *attracter* of the system and in this case is known as the *Sierpinski gasket*. Figure 2.1 shows the iterates, of the system in example 2.1, for different initial images.

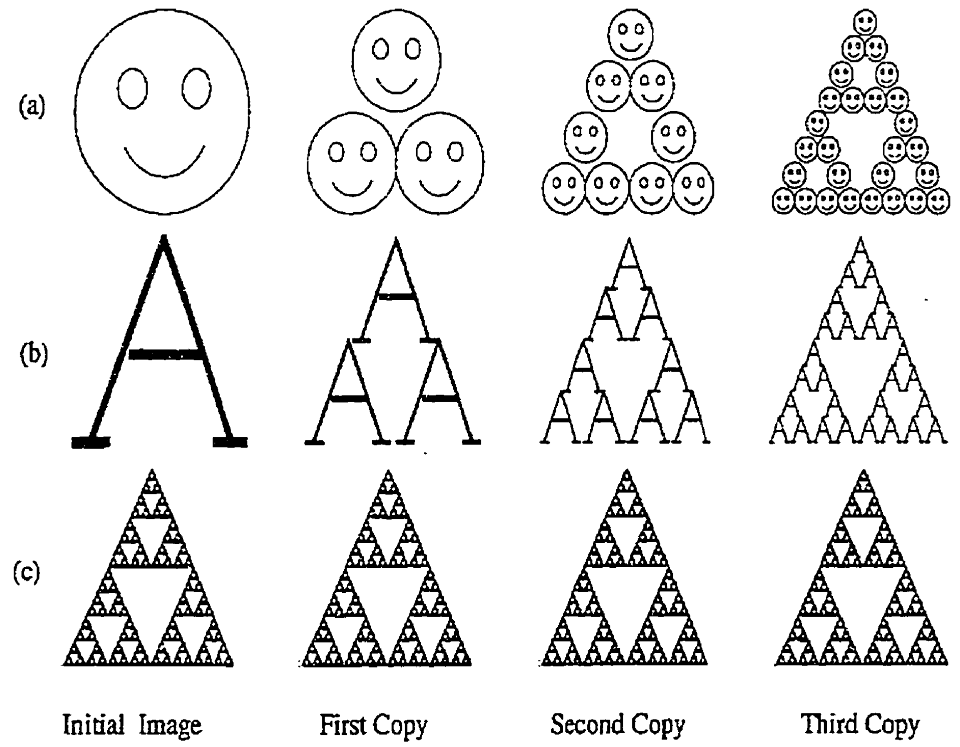


Figure 2.1: The first three iterates of the system in example 2.1, for three different initial images.

Notice that the functions, F_1, F_2, F_3, in Example 2.1, are *contractions*, that is, the distance between the iterates of two points is smaller than the distance between the two points themselves. This can formally be written as $d(F(x), F(y)) < d(x, y)$, where $d(a, b)$ represents the distance between a and b.

In this Chapter, we will discuss the fact such systems, like the one in Example 2.1, always converge to the same fixed point for an arbituary initial image. In the following Chapters, we will discuss how such systems create completely naturally appearing images, to the human observer.

Let us begin by taking a closer look at iterated function systems. Consider the affine function[2] F, given as:

$$F\begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{Ax}+\mathbf{y} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} \tag{2.1}$$

Theorem 2.1: Let λ and μ be eigenvalues of the matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$. The affine function F, defined in equation (2.1), is a contraction if $\lambda \neq \mu$ *st* $|\lambda| < 1$ and $|\mu| < 1$.

Let us now consider the function

$$F\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r\cos\Theta & -r\sin\Theta \\ r\sin\Theta & r\cos\Theta \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}, \quad st\ |r| < 1$$

Notice that F satisfies Theorem 2.1, and is therefore a contraction. Also, F not only shrinks (or expands) the image by a factor of $|r|$, but also rotates it, counterclockwise, by an angle of Θ. See Figure 2.2.

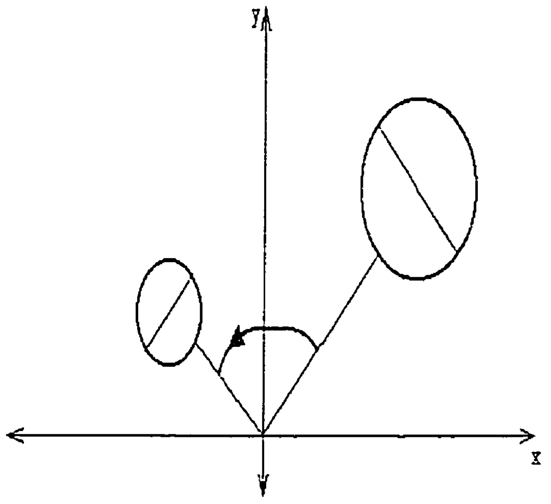


Figure 2.2: The original image x, and its iterate $F(x)$.

Example 2.2: Consider the four functions defined below, where each will begin by shrinking the image by a factor of 1/3.

[2] Affine functions are composites of a linear function, $G(\mathbf{x}) = \mathbf{Ax}$, and a translation function, $H(\mathbf{x}) = \mathbf{x}+\mathbf{y}$.

$$F_1\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}1/3 & 0\\0 & 1/3\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}$$

$$\begin{aligned}F_2\begin{pmatrix}x\\y\end{pmatrix}&=\begin{pmatrix}(1/3)cos(\pi/3) & (-1/3)sin(\pi/3)\\(1/3)sin(\pi/3) & (1/3)cos(\pi/3)\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}+\begin{pmatrix}1/3\\0\end{pmatrix}\\&=\begin{pmatrix}1/6 & -\sqrt{3}/6\\\sqrt{3}/6 & 1/6\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}+\begin{pmatrix}1/3\\0\end{pmatrix}\end{aligned}$$

$$\begin{aligned}F_3\begin{pmatrix}x\\y\end{pmatrix}&=\begin{pmatrix}(1/3)cos(-\pi/3) & (-1/3)sin(-\pi/3)\\(1/3)sin(-\pi/3) & (1/3)cos(-\pi/3)\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}+\begin{pmatrix}1/2\\\sqrt{3}/6\end{pmatrix}\\&=\begin{pmatrix}1/6 & \sqrt{3}/6\\-\sqrt{3}/6 & 1/6\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}+\begin{pmatrix}2/3\\0\end{pmatrix}\end{aligned}$$

$$F_4\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}1/3 & 0\\0 & 1/3\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}+\begin{pmatrix}2/3\\0\end{pmatrix}$$

Note: F_1 will leave the image where it is, F_2 will rotate it by $\frac{\pi}{3}$ and shift it by $\begin{pmatrix}1/3\\0\end{pmatrix}$; F_3 will rotate it by $\frac{-\pi}{3}$ and shift it by $\begin{pmatrix}1/2\\\sqrt{3}/6\end{pmatrix}$ and F_4 will shift it by $\begin{pmatrix}2/3\\0\end{pmatrix}$

The graph of the union of these four contraction mappings is known as the *Koch curve* and is shown in Figure 1.2 on page 3, along with the first few iterations.

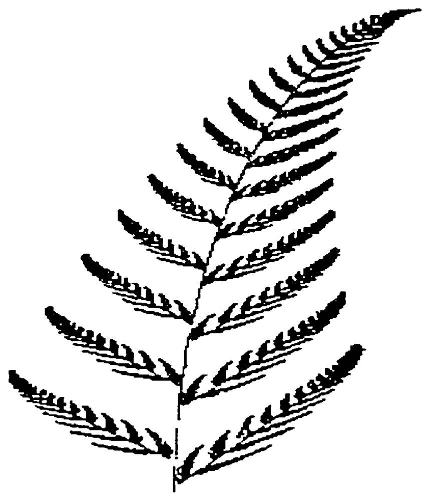


Figure 2.3: Barnsley's fern.

Example 2.3: Another extremely well known *IFS* fractal is *Barnsley's Fern.* It can be described by the following four functions, and is shown in Figure 2.3.

$$F_1\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}0&0\\0&.17\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}\qquad F_3\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}.2&-.26\\.23&.22\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}+\begin{pmatrix}0\\1.4\end{pmatrix}$$

$$F_2\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}.85&.04\\-.4&.85\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}+\begin{pmatrix}0\\3\end{pmatrix}\qquad F_4\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}-.25&.28\\.26&.24\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}+\begin{pmatrix}0\\.4\end{pmatrix}$$

It is a result of the *Contraction Mapping Theorem* that such systems have the characteristic of converging to the same final image regardless of the initial image, this theorem will be formally stated and proven later in this Chapter. Informally the *Contraction Mapping Theorem* states that if F_i are contractions on $\Re^2$, for $i = 1,2,3,...,n$, and $F(A) = \cup_{i=1}^{n} F_i(A)$, such that A is a closed and bounded subset of $\Re^2$, then F has a unique fixed set, which is a closed and bounded subset of $\Re^2$, i.e., $F(A_F) = A_F$. Since all closed and bounded subsets of $\Re^2$ are attracted to the set A_F, it is known as F's *attracter.*

Notice, to evaluate A_F, one could evaluate the sequence [3] $\{F^{(0n)}(z)\}_{n=1}^{\infty}$ for a given z in $\Re^2$, which can be extremely time consuming. A faster method of evaluating A_F was proposed by Barnsley [2]. This method uses random numbers in the interval $(0,1)$ and works for any finite system of contractions, F_i *suchthat* $i = 1,2,...,n$. Barnsley's algorithm [6] is:

Step 1: Choose an arbitrary point z in $\Re^2$.

Step 2: Generate a random number $r \in (0,1)$. Therefore rn is a random number in $(0,n)$.

Step 3: if $k < rn \leq k+1$, then plot the point $F_k(z)$. Otherwise do not.

Step 4: Let $z = F_k(z)$.

Step 5: With the new point z, repeat Steps 2, 3 and 4, then repeat the process as often as needed in order to generate a reasonable representation of the attracter A_F.

2.2: Analysis of ifs and the Generalized Collage Theorem for ifs

Before we can prove the *Contraction Mapping Theorem,* we need a few definitions, lemmas and theorems. We will begin by stating the following three lemmas, for a proof of each, see [12].

[3] $\{F^{(0n)}(z)\}_{n=1}^{\infty}$ represents the n^{th} iterate of F on z, as n approaches infinity

Lemma 2.1: Let F be a continuous function on $\Re^2$, and let A be a closed, bounded subset of $\Re^2$. Then *F(A)* is also closed and bounded.

Lemma 2.2: Let $\Upsilon = \cup_{i=1}^{n} A_i,\ st\ A_i$ is a closed subset of $\Re^2$ for $i = 1, 2, 3, ..., n$, then Υ is closed.

Lemma 2.3: Let $\Upsilon = \cup_{i=1}^{n} A_i,\ st\ A_i$ is a bounded subset of $\Re^2$ for $i = 1, 2, 3, ..., n$, then Υ is bounded.

Definition 2.1: A **metric space** $< \mathbf{X}, \rho >$ is a nonempty set $\mathbf{X}$ of elements (which we call points) together with a real-valued function ρ defined on $\mathbf{X}\times\mathbf{X}$ such that $\forall x, y, z \in \mathbf{X}$:

$$\mathbf{i} : \rho(x,y) \geq 0 \qquad \mathbf{iii} : \rho(x,y) = \rho(y,x)$$
$$\mathbf{ii} : \rho(x,y) = 0 \ \ iff \ \ x = y \qquad \mathbf{iv} : \rho(x,y) \ \leq \ \rho(x,z) + \rho(z,y)$$

Note: The function ρ is called a metric.

Definition 2.2: A metric space $\mathbf{X}$ is compact, if it is both closed and bounded.

Definition 2.3: A sequence $\{x_n\}_{n=1}^{\infty}$ in a metric space is called a **Cauchy Sequence**, if given $\epsilon > 0$ there is an N such that $\forall n, m > N$, we have $\rho(x_n, x_m) < \epsilon$.

Theorem 2.2: Let $< \mathbf{X}, \rho >$ be a metric space, with metric ρ, then every convergent sequence in $\mathbf{X}$ is a Cauchy sequence.

Proof: See [12]

Definition 2.4: Let $< \mathbf{X}, \rho >$ be a metric space, with metric ρ, if every Cauchy sequence in $\mathbf{X}$ is convergent in $\mathbf{X}$, then the space $\mathbf{X}$ is said to be a *complete* metric space.

Definition 2.5: Let $A \subset \Re^2$ and bounded, let $\epsilon > 0$, then $S_{A,\epsilon} = \{y : \exists a \in A\ st\ \rho(a,y) \leq \epsilon\}$ is called the *ϵ−neighbourhood* of A.

Theorem 2.3: Let A be a closed and bounded subset of $\Re^2$, then $S_{A,\epsilon}$ is also closed and bounded.

Proof: See [6]

Definition 2.6: The **Hausdorff Space** of $\mathbf{X}$ is defined as $H(X) = \{B \subset X : \ B\ is\ compact\}$

Definition 2.7: Let $A, B \in H(X)$, then the *Hausdorff metric* is defined as

$$h(A, B) = max\{inf\{\epsilon : B \in S_{A,\epsilon}\}, inf\{\epsilon : A \in S_{B,\epsilon}\}\}$$

For examples of the *Hausdorff metric*, see Figure 2.4, where the line is the Hausdorff distance between the two sets A and B.

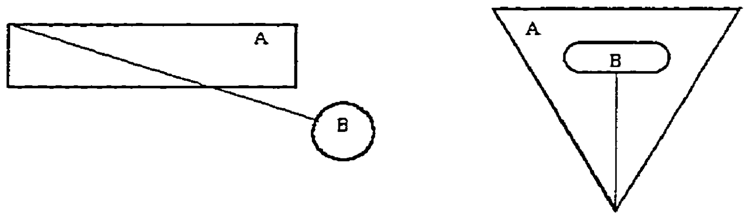


Figure 2.4: Two different examples of how to calculate the Hausdorff distance.

Theorem 2.4: [4]. Let $< X, \rho >$ be a complete metric space with metric ρ, then $H(X)$, with the *Hausdorff* metric h, is a complete metric space.

Proof: See [1]

Definition 2.8: [4]. Let $\mathbf{X}$ be a metric space with metric ρ. A map $w : \mathbf{X} \to \mathbf{X}$ is *Lipschitz* with Lipschitz factor s if $\exists s > 0,\ st\ \ s \in \Re$ and

$$\rho(w(x), w(y)) \leq s\rho(x, y) \quad \forall x, y \in \mathbf{X}$$

Furthermore a mapping is *contractive* iff $s < 1$.

Note: The rest of the theorems and proofs in this Chapter are taken from [4].

Theorem 2.5: If $f : \mathbf{X} \to \mathbf{X}$ is Lipschitz, then f is continuous.

Proof: See [4] pg 34

Theorem 2.6: If $w_i : \Re^2 \to \Re^2$ is contractive, with contractivity factor s_i, $i = 1, 2, ..., n$, then $W = \cup_{i=1}^{n} w_i : H(\Re^2) \to H(\Re^2)$ is contractive, in the Hausdorff metric, with contractivity factor $s = max\{s_i : i = 1, 2, ..., n\}$.

Proof: See [4] pg 34

Note: The notation $f^{(0n)}(x)$ represents the n^{th} interate of x over f.

Theorem 2.7: (The Contractive Mapping Fixed-Point Theorem) Let $\mathbf{X}$ be a complete metric space and $f : \mathbf{X} \to \mathbf{X}$ be a contractive mapping. Then there exists a unique point $x_f \in \mathbf{X}$ such that $\forall x \in \mathbf{X}$

$$x_f = f(x_f) = \lim_{n\to\infty} f^{(0n)}(x).$$

Proof: Let $x \in X$, then for $n > m$ we have

$$\rho(f^{(0m)}(x), f^{(0n)}(x)) < s\rho(f^{(0m-1)}(x), f^{(0n-1)}(x)) < s^m \rho(x, f^{(0n-m)}(x)). \tag{2.2}$$

Now we can use equation 2.2 and the *triangle inequality* repeatedly,

$$\begin{aligned}
\rho(x, f^{(0k)}(x)) &\le \rho(x, f^{(0k-1)}(x)) + \rho(f^{(0k-1)}(x), f^{(0k)}(x)) \\
&\le \rho(x, f(x)) + \rho(f(x), f(f(x))) + \dots + \rho(f^{(0k-1)}(x), f^{(0k)}(x)) \\
&\le (1 + s + \dots + s^{k-2} + s^{k-1})\rho(x, f(x)) \\
&\le \Big(\frac{1}{1-s}\Big)\rho(x, f(x)).
\end{aligned} \tag{2.3}$$

Therefore, we can rewrite equation (2.3) as

$$\rho(f^{(0m)}(x), f^{(0n)}(x)) < \Big(\frac{s^m}{1-s}\Big)\rho(x, f(x))$$

Now since $s < 1$, the left side can be made arbitrarily small for sufficiently large n and m. Therefore, the sequence $\{f^{(0n)}\}_{n=0}^{\infty}$ is a Cauchy sequence. Since $\mathbf{X}$ is assumed to be a complete space, the limit point of the sequence $x_f = lim_{n\to\infty} f^{(0n)}(x)$ is in $\mathbf{X}$. By Theorem 2.5, f is continuous, and so $f(x_f) = f(lim_{n\to\infty} f^{(0n)}) = lim_{n\to\infty} f^{(0n+1)}(x) = x_f$.

To prove the uniqueness of x_f, suppose $x_1 \neq x_2$, *st* x_1 and x_2 are both fixed points. Then $\rho(f(x_1), f(x_2)) = \rho(x_1, x_2)$, but we have shown that $\rho(f(x_1), f(x_2)) < \rho(x_1, x_2)$, therefore we get a contradiction.

Corollary 2.1: (Collage Theorem) With the hypothesis of the *Contractive Mapping Fixed point Theorem*,

$$\rho(x, x_f) \le \Big(\frac{1}{1-s}\Big)\rho(x, f(x)).$$

Proof: In Equation (2.3) we showed that

$$\rho(x, f^{(0k)}(x)) \leq \Big(\frac{s^m}{1-s}\Big)\rho(x, f(x))$$

Therefore, simply let $k \to \infty$.

Definition 2.9: A *Lipschitz function* f is called *eventually contractive*, if $\exists n$ st $f^{(0n)}$ is contractive. n is called the *exponent of eventual contractivity.*

Note: Even if $\exists w_i$, $i = j, ..., l$ *st* $1 < j < ... < l \leq n$ are not contractive, $W = \cup_{i=1}^{n} w_i$ may still be eventually contractive.

Corollary 2.2: (The Generalized Collage Theorem) Let f be eventually contractive with exponent n, then $\exists! x_f \in \mathbf{X}$ *st* $\forall x \in \mathbf{X}$

$$x_f = f(x_f) = \lim_{k \to \infty} f^{(0k)}(x).$$

In this case

$$\rho(x, x_f) \leq \Big(\frac{1}{1-s}\Big)\Big(\frac{1-\sigma^n}{1-\sigma}\Big)\rho(x, f(x)),$$

where s is the contractivity of $f^{(0n)}$ and σ is the Lipschitz factor of f.

Proof: Let $g = f^{(0n)}$. We want to show that $f^{(0k)}$ converges to x_g; that is, $f^{(ok)}(x)$ is arbitrarily close to x_g $\forall k$ sufficiently large. For any k, we can write $k = qn + r$, with $0 \leq r < n$. Therefore,

$$\begin{aligned}
\rho(f^{(0k)}(x), x_g) &= \rho(f^{(0qn+r)}(x), x_g) \\
&\leq \rho(f^{(0qn+r)}(x), f^{(0qn)}(x)) + \rho(f^{(0qn)}(x), x_g) \\
&= \rho(g^{(0q)}(f^{(0r)}(x)), g^{(oq)}(x)) + \rho(g^{(0q)}(x), x_g) \\
&\leq s^q \rho(f^{(0r)}(x), x) + \rho(g^{(0q)}(x), x_g)
\end{aligned}$$

Note: $g(x) = f^{(0n)}(x)$

However, both of these terms can be made arbitrarily small for $0 \leq r < n$ and q sufficiently large. The fixed point condition follows from the continuity of f, and uniqueness follows from the uniqueness of x_g.

For the inequality, we know from Corollary 2.1 that:

$$\rho(x, x_g) \leq \Big(\frac{1}{1-s}\Big)\rho(x, g(x)) \tag{2.4}$$

and

$$\begin{aligned}\rho(x,g(x)) &= \rho(x,f^{(0n)}(x)) \\ &\le \sum_{i=1}^{n} \rho(f^{(0i)}(x), f^{(0i-1)}(x)) \\ &\le \rho(x,f(x)) \sum_{i=1}^{n} \sigma^{i-1} \\ &\le \Big(\frac{1-\sigma^n}{1-\sigma}\Big)\rho(x,f(x)) \qquad (2.5)\end{aligned}$$

The result follows from equations (2.4) and (2.5).

Note: It is sufficient for there to exist an n for which f is contractive, we do not need f to be contractive for all large n.

2.3: Recurrent Iterated Function Systems, rifs

The notion of *ifs* can be extended to *rifs*. Given a finite collection of *ifs*, an *rifs* is simply an *ifs* with the added capability of mapping different *ifs* into one image.

Example 2.4: Say you wanted to create the Barnsley fern with Sierpinski triangle leaves. We then need two *ifs*, one would create the Sierpinski triangle, the other would place the triangles in position of the leaves in the fern and would copy the Sierpinski triangle. The resulting image is shown in Figure 2.5.

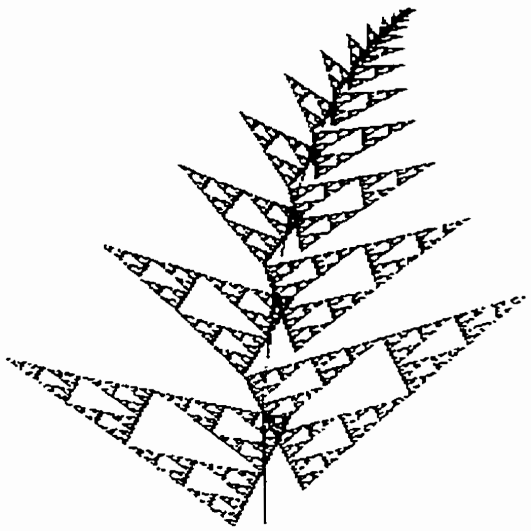


Figure 2.5: Barnsley's fern with Sierpinski triangle leaves.

One can think of $rif's$ as being the collection of affine mappings w_i for $i = 1, 2, ..., n$ and a digraph G such that each node of the digraph represents an affine mapping, and each edge (w_i, w_j) means that the composition $w_j \circ w_i$ is allowed.

Note: It is important to notice that all of the relevant properties of ifs theory are carried over to $rifs$ theory. We will now go on to define what is meant by a mapping between spaces.

Let $< X_i, h_i >$ be a complete metric space for $i = 1, 2, ..., n$, and let $H = H_1 \times H_2 \cdots \times H_n$ st H_i is the set of non-empty compact subsets of X_i. Therefore we have a typical element $(A_1, A_2, ..., A_n) \in H$ st A_i is a non-empty subset of X_i. Define the metric h^* as;

$$h^*((A_1, ..., A_n), (B_1, ..., B_n)) = max\{h_i(A_i, B_i) : i = 1, 2, \ldots n\}$$

Therefore, $< H, h^* >$ is a complete space.

Let $W_{ij} : H_i \rightarrow H_j$ where $W_{ij} = \cup_k w_{ijk}$ st w_{ijk} is the k^{th} contractive mapping from X_i to X_j.

We can now take $W : H \rightarrow H$ st $W(A_1, A_2, ..., A_n) = (\cup_j W_{1j}(A_j), ..., \cup_j W_{nj}(A_j))$. It is important to notice that there must exist at least one mapping W_{ij} (for each i) with a non-empty image.

Theorem 2.8: For W defined above $!\exists x_f = (A_1, A_2, ..., A_n)$ st $x_f = W(x_f)$.

Proof: This follows immediately from theorem 2.7.

2.4: Partitioned Iterated Function Systems, pifs

Most of the methods of fractal image compression techniques described in this dissertation use *pifs*, which are a generalization of *ifs*. Using *pifs* to encode images simplifies the encoding of images who's parts are not self similar to the whole but rather to other parts of the same image. Basically, the method works by limiting the domains of the mapping to parts of the space.

Definition 2.10: from [4]. Let X be a complete metric space, and let $D_i \subseteq X$ for $i = 1, 2, ..., n$. A partitioned iterated function system is a collection of contractive maps $w_i : D_i \rightarrow X$ for $i = 1, 2, ..., n$.

The analysis, for the general form, of *pifs* has not been developed yet. In particular, for *pifs* there is no equivalent theorem to the Contractive Mapping Fixed-Point Theorem (theorem 2.7). However, *pifs* are well understood and work very well for the techniques described in the following Chapters.

2.5: Conclusion

In this Chapter, we have shown how to manipulate *Iterated Function Systems* so as to create fractals, the different characteristics such systems have, and the reasons they have them. Because of this work we were able to prove the *Generalized Collage Theorem*, which will allow us to compress images using *ifs*. We went on to describe two stronger kinds of iterated function systems called recurrent iterated function systems, and partitioned iterated function systems. Recurrent iterated function systems allow us to create complicated images much more easily than using ifs. The analysis of the existence of a unique fixed point in such systems was given. Partitioned iterated function systems where introduced and a formal definition was given.

CHAPTER 3

Image Encoding using IFS's, RIFS's and PIFS's

This Chapter introduces two image models of particular importance to fractal image compression of greyscale and colour images. We then conclude with a discussion concerning the representation of images using *ifs*, *rifs* and *pifs*.

3.1: Image models

To be able to work with and manipulate images, one must decide which of the available image models to use. Since the Computer can only work with the *Pixelized Data* model and our analysis of fractal image compression is much easier when working with the *Functions in* $\Re^2$ model, these are the only two image models we will discuss.

3.1.1: Pixelized Data

Imagine evenly dividing a grey-scale image into small dots, each of which is a shade of grey. We can therefore assign a discrete value to each dot, in the range 0 to b, corresponding to its shade of grey. If we only use one byte of data to represent the grey level value of the dot (which is typical), then it will be in the range of 0 to 255. By increasing the number of bytes used to store the grey level value, we increase the grey-scale resolution. Pixels can be thought of as such dots, such that the number of pixels used in an image corresponds to its resolution.

We can now think of an image as a vector $\mathrm{x} = (x_1, x_2, ..., x_q)$, where q is the total number of pixels used in the image, and x_i is a value in the range 0 to 255. Therefore, to calculate the distance between two images x and y, one would use the *Root Mean Square, rms,* metric defined as;

$$\rho_{rms}(\mathrm{x},\mathrm{y}) = \|x - y\|_2 = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{3.1}$$

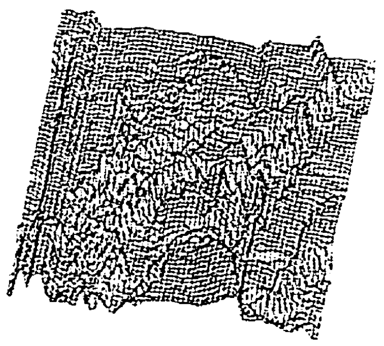
To calculate the difference between two images, one usually uses the *Peak signal-to-noise ratio, PSNR,* which is measured in decibels dB, and is defined as;

$$PSNR = 20\log_{10}\left(\frac{b}{rms}\right)$$

for b defined above. Note that in our example $b = 255$.

3.1.2: Functions on $\Re^2$

Although the computer can only deal with images as sets of pixels, when developing the analysis of *rifs* and *pifs* it is easier to represent an image as the graph of a function. Let $f : I^2 \to I$ represent the grey value of the image, where the unit square $I^2 = \{(x,y) : 0 \leq x, y \leq 1\}$, and $I = [0,1]$. Therefore $f(x,y) = z$ represents the grey-level value at the point $(x,y) \in I^2$. One should notice that the graph of f is formed of the points $(x, y, f(x,y))$, and is therefore a subset of $\Re^3$. See Figure (3.1a), which is the graph of the Lenna image, such that the height is the grey-level going from black (low) to white (high), and see Figure (3.1b), which is the original Lenna image.



Figures 3.1a and 3.1b: The three dimensional graph of Lenna and the original Lenna image.

Before discuss f as an image model, we provide a few definitions.

Definition 3.1: Two functions g and h are said to be equal if the set of points on which they differ has measure 0. In such a case we say that $g = h$ *almost everywhere, a.e.*

Definition 3.2: The *essential supremum, ess sup*, of a function $f(t)$ as defined in terms of Lebesque measure m is;

$$ess\ supf(t) = inf\{N : m\{t : f(t) > N\} = 0\}$$

Definition 3.3: Consider the $L^p(I^2)$ spaces, such that $f \in L^p \ \ iff \ \ \|f\| < \infty$ where

$$\rho_p(f,g) = \|f-g\|_p = \begin{cases} \left(\int_{I^2} |f-g|^p\right)^{\frac{1}{p}} & 1 \le p < \infty \\ ess\ sup|f-g| & p = \infty \end{cases}$$

This metric is known as the L^p metric. For a detailed discussion of the L^p space the reader is recommended to read [12].

In our analysis of *rifs* and *pifs* it is not always useful to use the L^∞ metric, therefore, let us also define the *supremum metric.*

Definition 3.4: Given two functions f and g, the distance between them, with respect to the supremum metric, is;

$$\rho_{sup}(f,g) = sup\{|f(x,y) - g(x,y)| : (x,y) \in I^2\}$$

Definition 3.5: Let us define the space of images as $F = \{f : I^2 \to \Re\}$ *st* f is a measurable function.

Theorem 3.1: $< F, \rho_{sup} >$ is a complete metric space.

Proof: [13]

We are now equipped with all of the necessary tools to discuss image processing with *rifs* and *pifs*. Since most images which one would want to compress are probably colour images, we will provide a brief discussion of image models for colour images.

3.2: Colour Images

The visible light spectrum is composed of a continuous range of frequencies, each describing a colour. However, the sensitivity of the human visual system is limited. A colour perceived by a human observer can be approximated by superposing red, green and blue values. One should not think that the red, green and blue system is the only way of recreating the colour spectrum. for human observers, there are other systems as well. such as YIQ, YUV, HSL and the CMY(K) system.

A brute force approach to fractal compression of colour images would be to work with three versions of the same image, the green, red and blue versions. One could then encode each separately, creating three *pifs* codes for the same image. To recreate the image, one would decode each separately and subsequently superpose them, thus recreating the colour image. Figure 3.2 shows the flow chart depicting this brute force approach.

Obviously, this method is extremely inefficient and completely unpractical. A much better approach would be to consider the high correlations between the fractal codes for the same block in each version of the image and the high correlation between colour planes. For instance, the correlation between the blue and red plane is approximately 0.78, between the red and green it is approximately 0.89 and between the green and blue it is approximately 0.94. Exploiting these correlations results in a much more efficient encoding of the image with very little degradation in quality.

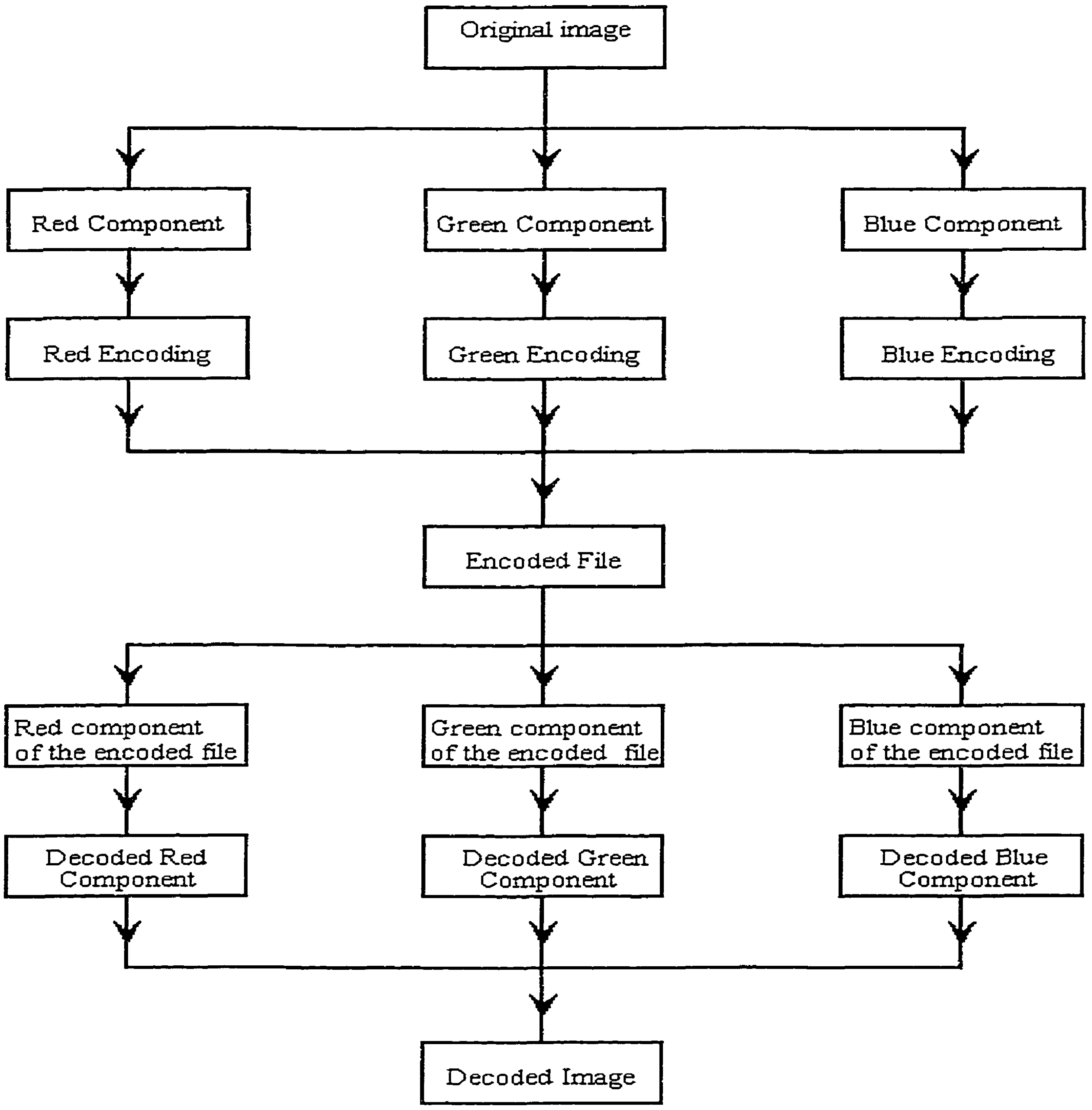
Such correlations between colour planes are not restricted to the red, green, blue system but also exist in the (L,I,Q) and (Y,U,V) systems. As such, one can work with the system of his choice, without a significant effect to the compression ratio and image fidelity.

To convert an image from the (R,G,B) system to the (L,I,Q), one finds the red, green and blue components of each pixel, then convert each (R,G,B) triplet to its corresponding (Y,I,Q) triplet. This can be done with the following transformation;

$$\begin{pmatrix} Y_i \\ I_i \\ Q_i \end{pmatrix} = \begin{pmatrix} .299 & .587 & .114 \\ .596 & -.274 & -.322 \\ .211 & -.523 & .312 \end{pmatrix} \begin{pmatrix} R_i \\ G_i \\ B_i \end{pmatrix}$$

We can now use the same method of encoding the image with this system as with the (R,G,B) system.

By exploiting such correlations, the compression ratio is 2 to 2.7 times more efficient for colour images than the corresponding grey-scale image. The rest of this dissertation, for simplicity, will only deal with grey-scale images.



Figures 3.2: The flow chart of the brute force approach for colour images.

3.3: IFS Representation of Images

In fractal image coding, one tries to exploit self-similarity between parts of the image and the image as a whole. If we can find a contractive affine mapping $f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{y}$ which converges to the desired image when iterated, then we could store the image, much more efficiently than storing the value of each pixel in the image. For example, consider *Barnsley's fern*, to store it in memory, using the value of each pixel, requires 65 536 bytes. However, to store the representation of its corresponding affine mapping, $\mathbf{W}$, only requires 96 bytes [1]. To recreate the image, we iterate $\mathbf{W}$ on any initial image, where each iteration has a closer resemblance to the final image. Therefore, such a representation would be extremely efficient with regards to memory, and hence image transmition.

Once we have $\mathbf{W}$ it is rather easy to recreate the image, however for real images finding $\mathbf{W}$ is difficult and computationaly long. Also, real images are not usually self similar to the whole image, but rather, different parts of the image are similar to other parts. Using mappings from one part of the image to another creates a much better quality than using mappings of the whole image to different parts of the image. Such methods are used when encoding images with *rifs* and *pifs*, which is the subject of the next Section.

3.4: Image Encoding with rifs and pifs

Divide the image, I^2, into non-overlapping squares called *Range blocks*, R_i, st $\cup_{i=1}^{n} R_i = I^2$. Then divide I^2 into squares, called *Domain blocks*, D_i, possibly overlapping, that are twice the size of the range blocks. For clarity, see Figures 3.3 and 3.4.

[1] 4 transformations $\times$ 6 numbers/transformation $\times$ 32 bits/number $\times$ 1 byte/8 bits = 96 bytes

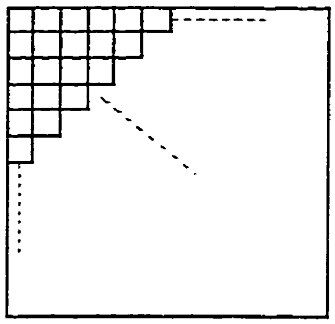


Figure 3.3: The range blocks of an arbitrary image.

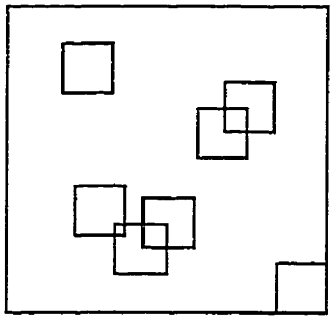


Figure 3.4: A few of the possible domain blocks of the same arbituary image.

In both methods, *rifs* and *pifs*, we will use contractive mappings from domain blocks, D_i to range blocks R_j. However, the way in which each chooses which D_i will map to a given R_j is different.

To encode grey-scale images we will use mappings of the form;

$$w_i \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} e_i \\ f_i \\ o_i \end{pmatrix} \tag{3.2}$$

Where the coordinates x and y, in Equation 3.2, represents a pixel, and z represents its grey-scale value. The coefficient s_i controls the luminance and the coefficient o_i controls the luminance offset. One can think of them as the contrast and the brightness controls, respectively.

In Chapter 2, all of our proofs were based on the *Hausdorf metric*, however, we could also have used the *Root Mean Square Error metric, rms*, defined in Equation 3.1. The reason we used the *Hausdorf metric* and not the *rms*, is to simplify the proofs. However, in practice one should use the *rms* metric since it is computationally easier to calculate.

At this point *pifs* and *rifs* diverge, so we will consider each system separately.

3.4.1: Encoding Images with Recurrent Iterated Function Systems

For the mappings defined in Equation (3.2) and the space $H = (R_1 \times I, ..., R_n \times I)$, we can define W as a mapping from H to H such that;

$$W_{i,j} = \begin{cases} w_i & R_j \subset D_i \\ \phi & otherwise \end{cases}$$

Notice that the domain blocks D_i are simply the union of range blocks, R_i. Consider A_i as the graph of the grey-scale function over R_i, then a point in H is of the form $(A_1, A_2, ..., A_n)$. The proof of the following theorem is lengthy and is omitted.

Theorem 3.2: $< H, \rho^*_{sup} >$ is a complete metric space, where ρ^*_{sup} is defined as;

$$\rho^*_{sup}((A_1, ..., A_n), (B_1, ..., B_n)) = sup\{\rho_{sup}(A_i, B_i) : 1 \leq i \leq n\}$$

In Chapter 2 Section 2.3, we defined the mapping W which can now be written as;

$$W(A_1, ..., A_n) = \left(\cup_j w_1(A_j), ..., \cup_j w_i(A_j), ..., \cup_j w_n(A_j)\right) \; st \; R_j \subset D_i \quad i = 1...n$$

It is important to notice that our definition of ρ^*_{sup} is only sensitive along the *z-axis*, the grey-scale component, and not the x and *y-axis*, the spatial components. Therefore, if W is eventually contractive, then the spatial components may actually expand without effecting the convergence of the system.

3.4.2: Encoding Images with Partitioned Iterated Function Systems

For the mappings defined in Equation (3.2) and the space $H = (R_1 \times I, ..., R_n \times I)$, we can define W as;

$$\mathbf{W} = \cup_{i=1}^n w_i, \; st \; w_i : \Re^3 \rightarrow \Re^3$$

Notice that W, can be contractive even if w_i is only contractive for the *z-axis*.

Definition 3.6: [4], **pg 50** If $w : \Re^3 \to \Re^3$ is a map with $(x', y', z_1') = w(x, y, z_1)$ and $(x', y', z_2') = w(x, y, z_2)$, then w is called *z contractive* if $\exists s \in (0, 1)$ *st*

$$|z_1' - z_2'| \le s|z_1 - z_2|$$

where x' and y' are independent of z_1 or z_2 $\forall x, y, z_1, z_2$.

Theorem 3.3: Let w_i for $i = 1...n$ be z contractive mappings from $\Re^3$ to $\Re^3$. Then the collection of these mappings $\cup_{i=1}^n w_i$ (which we will call W), is contractive in F with the supremum metric.

Proof: [4] **pg 51** Let $s = max\{s_i : 1 \le i \le n\}$, where s_i are the z contractive components of w_i then

$$\begin{aligned}
\rho_{sup}(W(f), W(g)) &= sup\{|W(f)(x,y) - W(g)(x,y)| : (x,y) \in I^2\} \\
&= sup\{z\text{-}component\ of\ |w_i(x,y,f(x,y)) - w_i(x,y,g(x,y))| : \\
&\qquad (x,y) \in D_i, i = 1, ..., n\} \\
&\le sup\{s_i|f(x,y) - g(x,y)| : i = 1, ..., n\} \\
&\le sup\{s|f(x,y) - g(x,y)|\} \\
&\le s \cdot sup\{|f(x,y) - g(x,y)|\} \\
&\le s \cdot \rho_{sup}(f, g)
\end{aligned}$$

Note: By the Contractive Mapping Fixed Point Theorem, we are assured that W has a unique fixed point in F, even if the z-contractive maps are expanding along the x and y axis.

The challenge of using *pifs*, for fractal image compression, is finding a *pifs* representation of an arbitrary digital image such that when iterated, the image produced appears to be the same to the human observer. We will discuss various methods of finding such *pifs* in the following chapters.

3.5: Conclusion

In this Chapter we discussed the different practical image models for fractal image compression and showed how to calculate the difference between two images using each model. We also discussed how to represent colour images efficiently for our models. We concluded with a discussion of image representations using *ifs*, *rifs* and *pifs* for grey-scale images, and showed some of the analysis which causes such systems to work.

CHAPTER 4

Improved Partitioning and Searching Methods

So far we have used a brute force approach to fractal image compression, which divides an image into fixed size non-overlapping range blocks and overlapping domain blocks that are twice the size of the range blocks. Then, each range block is compared to each possible transformation of each domain block, storing the transformations once found. As its name implies, this method is not very efficient. There are many ways in which we could improve the brute force algorithm with respect to compression ratio, fidelity and speed.

The compression ratio is mostly dependant on the number of transformations needed to store the image, by increasing the size of the range blocks used we reduce the number of transformations needed, thus improving the compression ratio. However, there are many ways of reducing the number of transformations, some of which may result in a loss of fidelity. The next Section describes methods of varying the size and/or shape of the blocks to be used. By varying the size, we are able to cover larger parts of an image using only a single range block and thus increasing the compression ratio, or we can cover areas of high detail with smaller range blocks, thus increasing the fidelity.

Another way in which we can improve the brute force approach is by reducing the processing time needed to encode an image. We can reduce encoding times in various ways, such as increasing the strength of the processor, or even better, by limiting the total number of comparisons to be performed, in an intelligent way of course. Algorithms applying this method are discussed in Section 4.2.

4.1: Block Shapes

In this Section we will present different methods of varying the size and/or shape of the blocks to be used, and discuss the effects of doing so.

4.1.1: Variable Sized Square Blocks using Quadtrees

Quadtree partitioning is a rather simple method of dividing an image into variable sized blocks so as to satisfy an error limit. When using the brute force algorithm, we simply find the domain block which resembles a particular range block the most. However, this does not set an upper limit on the difference between the two blocks, and thus it is possible that the best domain-range block pairing is a poor one. If there are a significant number of such matchings, or if the blocks in question are in a particularly high detail area, then it is likely that the decoded image will be of poor quality.

The Quadtree algorithm sets an upper limit for the difference between the domain-range matchings, thus improving the quality of the decoded image. It does so by dividing the original image into overlapping domain blocks of various sizes. For example, an image of size 256×256 could have domain blocks whose sides are of length $8, 12, 16, 24, 32, 48$ and 64. Then it recursively divides the original image into non-overlapping range blocks until they are of a specified size, 32×32 for example. For each range block, it searches the domain pool for the domain block which resembles it the most (minimizing the rms difference). If the rms difference is smaller or equal to the maximum error, then save the transformation and delete the range block from the range pool. If no domain block is found, divide the range block into four equal subsquares, which are the new range blocks, and search the domain pool again, deleting the range block if a match is found. Repeat until the pool of range blocks is empty or the size of the range blocks is less than a specified lower limit.

This method is called the Quadtree algorithm since it can be represented by a tree who's root is the original image and each node has, potentially, four subnodes which correspond to the range blocks of the image. For a better understanding, see Figures 4.1a/b/c/d, such that Figure 4.1a shows an example of the quadtree partitioning of an image, Figure 4.1b shows its corresponding quadtree, Figure 4.1c shows the partitioning of the Lenna image and Figure 4.1d is the decode image of a dog using the quadtree algorithm.

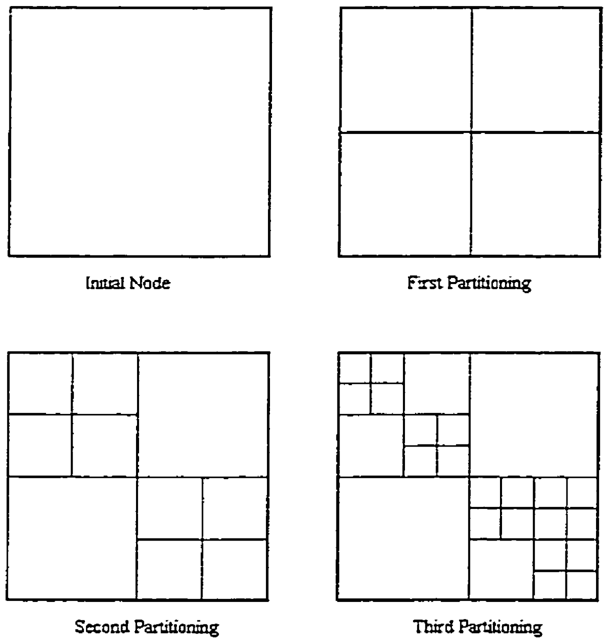


Figure 4.1a: An example of the quadtree partitioning of an image.

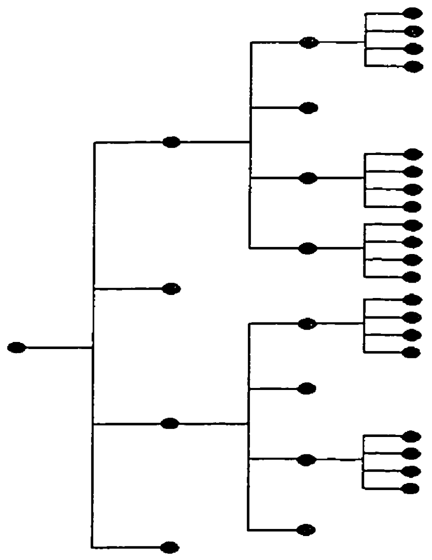


Figure 4.1b: The quadtree corresponding to the partitioning in Figure 4.1a.

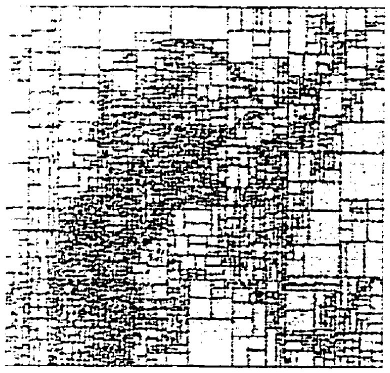


Figure 4.1c: A representation of the Lenna image, partitioned using the quatree algorithm.

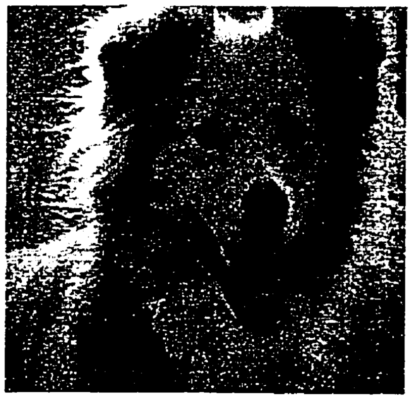


Figure 4.1d: The decoded image of a dog, which was encoded with the quadtree partitioning algorithm.

Notice that in order to compare the domain and range blocks, we must reduce the number of pixels in the larger domain block. If we are comparing the domain blocks which are twice the size of the range blocks, then the number of pixels in the domain blocks must be decreased to one quarter of the original amount. There are two obvious ways of doing so, the preferred way is to take the average pixel values as the new pixel value, which is known as sub-sampling, the other is to choose a representative pixel as the new pixel value. It is also important to note that using the transformations described in Equation 3.2 of Chapter 3, we are multiplying each pixel value by scaling factor $|s_i|$. Thus if $|s_i| > 1$ for the mapping w_i, then it is possible that $W = \cup_i w_i$ may not be eventually contractive, and therefore we are not guaranteed convergence of the decoded image. As such, the values for which $|s_i| > 1$, must be truncated to some fixed s_{max}.

To decode the image. begin by taking an arbitrary image, divide the image into the final quadtree partitioning we had so as to determine the position of the range blocks. For each range block, the domain block that maps to it is spatially reduced by a factor of two using sub-sampling. Each pixel value is then multiplied by the scaling factor s_i, and o_i is added to it before it is placed in its proper place in the range block, which depends of the orientation of the transformation. The whole image is iterated in this way until the difference between two subsequent iterations is smaller than a given threshold value.

For a more profound discussion of this method, the reader is suggested to read [4], in which Fisher discuses, among other things, the effects of varying s_{max} and the domain pool.

4.1.2: Rectangular Block Shapes Using HV-Partitioning

Although the quadtree algorithm is much better than the brute force algorithm, it does not take advantage of natural structures in the image. A better method, which also has variable sized blocks, is the *Horizontal-Vertical partitioning method*, or simply the *HV partitioning method*. Similarly to the Quadtree algorithm, an image is recursively partitioned, however, the partitions are not necessarily and usually not square shaped. This increases the power of the algorithm since the positions of the partitions are not fixed and thus can be exploited to take advantage of self-similar structures. This method also has the advantage of using larger range sizes than the Quadtree method, since the Quadtree method always partitions a range block into four range blocks which are one fourth of its size, where as the HV-method allows you to partition a range block into two rectangles.

Example 4.1: Consider the image in Figure 4.2a. We can then divide the rectangle R_0 into the two rectangles R_1 and R_2, shown in Figure 4.2b, such that R_1 contains the diamond and R_2 contains the diagonal line. We can then divide R_1 vertically into two rectangles R_3 and R_4. Dividing each of R_3 and R_4 into two rectangles creates the partitioning shown in Figure 4.2c. Therefore, each of the smaller rectangles in R_1 can be mapped from R_2 by a simple transformation.

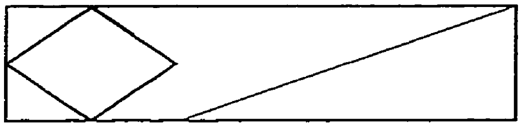


Figure 4.2a: The original image.

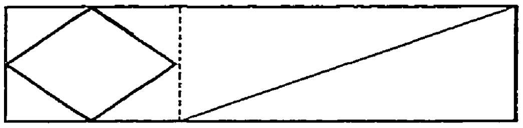


Figure 4.2b: The first partitioning of the image.

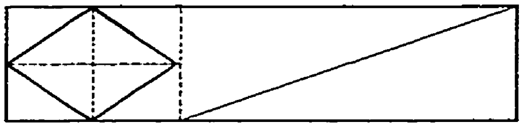


Figure 4.2c: The next two partitions of the image.

Ideally we would like to partition a block along the distinct horizontal and vertical lines of that block and along distinctive edges, without creating extremely narrow rectangular range blocks. To do so, imagine an image created of pixel values $r_{i,j}$ such that $1 \leq i \leq N$ and $1 \leq j \leq M$. A distinct horizontal edge in the image will correspond to a significant difference in pixel values from one row to another. We would like to partition the image along the most significant horizontal or vertical edge present. Therefore, we can begin by calculating the average difference between rows of pixels. Notice the difference between rows i and $i+1$ is $\left(\sum_j r_{i,j} - \sum_j r_{i+1,j}\right)/M$, repeat that calculation for each pair of subsequent rows. In order to find the most distinctive horizontal line, it suffices to take the maximum difference calculated, however, we would like to avoid extremely narrow rectangles. As such the linear biased function $min(i, N-1-i)$ is introduced by multiplying it with the differences calculated. Performing similar calculations along the vertical columns of the image yields

$$h_i = min\{i, N-1-i\}\frac{\left(\sum_j r_{i,j} - \sum_j r_{i+1,j}\right)}{M} \quad st\ 1 \leq i < N$$

$$v_i = min\{i, M-1-i\}\frac{\left(\sum_i r_{i,j} - \sum_i r_{i,j+1}\right)}{N} \quad st\ 1 \leq i < M$$

Without loss of generality, assume $|h_l| \geq |h_p| \ \forall p$ and that $|v_k| \geq |v_q| \ \forall q$. We now partition the image horizontally along row l if $|h_l| \geq |v_k|$, otherwise partition the image vertically along column k.

Given a range block, the domain pool through which we will search for a match will consist of all rectangles who's sides are larger by a factor of 2 or 3, where each side can have a different factor. In order to compare the domain and range blocks, the pixels in the domain blocks are averaged in groups of 2×2, despite the ratio of range to domain side. Similarly to the Quadtree approach, we perform a certain number of partitions before we begin to search for domain-range block matchings and continuously partition the particular range block until a predetermined threshold difference value is satisfied or the range block is smaller than a predetermined size.

Recall that once an image has been encoded, it can be decoded at any size. Y. Fisher and S. Menlove took advantage of this in a decoding algorithm which they describe in [4]. The algorithm considers an $N \times M$ image as an element of $\Re^{N \times M}$. The method of calculating the fixed point, the final image, is by decoding the image at $\frac{1}{16}^{th}$, $\frac{1}{8}^{th}$, $\frac{1}{4}^{th}$ and finally $\frac{1}{2}$ of the final image size, in that order. By doing so, you perform much less calculation than you normally would at the beginning of the decoding process, and once you decode the image at the final size you only need to perform two iterates before achieving the final image.

It is interesting to note that one could modify the H-V partitioning method so as to include partitions along lines of 0, 45, 90 and 135 degrees. This would increase the strength of the method since one would not be confined to strictly vertical or horizontal lines.

4.1.3: Region based Coding with a Heuristic Search

So far we have been confined to square and rectangular shaped range blocks, however there is another method, called FAC-P, which uses irregular shaped range regions, and was developed by *Laster Thomas* and *Farzin Deravi* [15]. This method begins by dividing an image into the standard, square, range and domain blocks. An arbitrary range block, called the seed, is chosen from the range block pool and its corresponding domain block match is found. This sole range block is now referred to as the range region, and the domain block as the domain region. The algorithm then recursively tries to extend the range region both horizontally and vertically by verifying if the extended region has the same parameters as the original seed transformation and that the range block being absorbed into the region is within a given threshold value of the mean squared error distance. Each time a block is absorbed into a range region it is deleted from the range pool, the algorithm continues until the range pool is empty. For a better understanding see Figures 4.3a and 4.3b, such that Figure 4.3a shows the seed and domain block matching along with a possible extension, and Figure 4.3b shows a range region and its corresponding domain region.

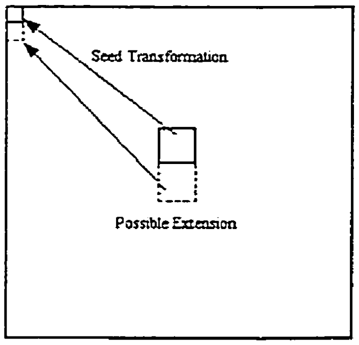


Figure 4.3a: The seed (solid line) and a possible extension (dashed line).

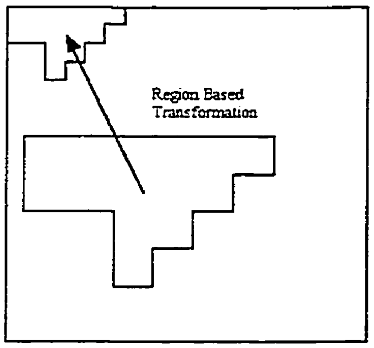


Figure 4.3b: A range region and its corresponding domain region.

One may notice that the region developed about a given seed, depends a lot upon the seed chosen. It may be that the domain region cannot be extended in the corresponding direction because it is at the edge of the image, or the range block being rejected would be accepted if the domain region was shifted. Thomas and Deravi knew this, and as such modified there algorithm, calling it FAC-AP. The FAC-AP algorithm is identical to the FAC-P algorithm with the exception that when a range block is rejected, before considering another range block, the domain region is shifted horizontally or vertically, by one block, in the opposite direction of the rejected range block. The algorithm then checks the new transformation for each block already in the range region, in order to ensure that the change is acceptable for each one as well as the range block being considered as an extension to the region.

Thomas and Deravi also knew that a block along the edge of region S, may have been accepted by an adjacent region T if it was still available when T was being created. It may also have created a better encoding if such a block had belonged to region T. They further improved the algorithm, to take such cases into consideration, and called the modified algorithm the FAC-ACP method. It basically allows for competition between adjacent regions for any range block that can be encoded with a smaller error by one region and that does not breakup the original region to which it belonged, see Figures 4.4a and 4.4b.

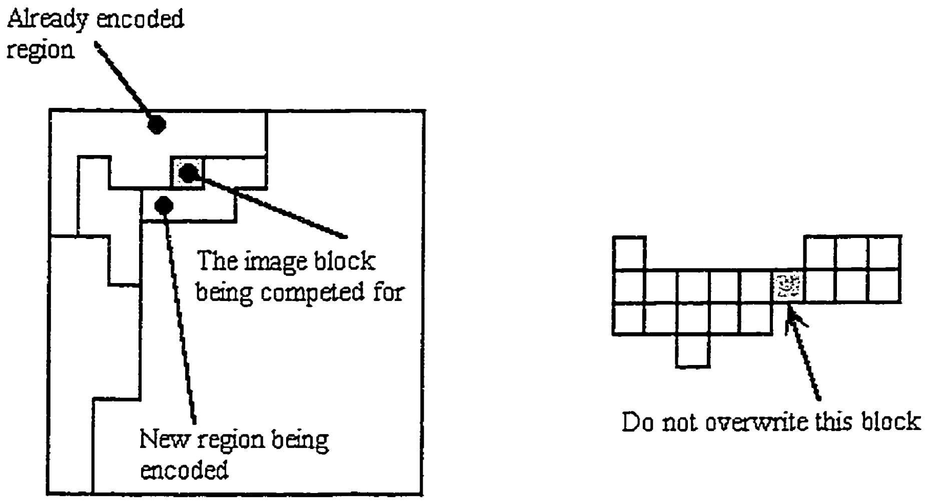

Already encoded region
The image block being competed for
New region being encoded
Do not overwrite this block

Figure 4.4a and 4.4b: Which show two regions competing to include a range block and a block which should not be considered as a candidate to belong to another range region, respectively.

Compared to the standard brute force algorithm, when using the Lenna image, they achieved almost the same PSNR, a difference of .3 dB, but attained a compression ratio of 41:1, compared to 19:1 for the brute force image.

4.1.4: Other Partitioning Methods

There are many different ways in which one can partition an image into domain and range blocks. An interesting method not mentioned so far is one which uses triangular shaped blocks of variable sizes. In triangular partitioning, one divides the original image diagonally into two triangles. We then recursively subdivide each triangle into four triangles by connecting the midpoint of the sides of the triangle. This method is potentially stronger than the previously mentioned methods since the triangles can have any orientation and it can be adjusted so that triangles have self-similar properties. Figure 4.5 shows a partitioning of the Lenna image using triangular partitioning.

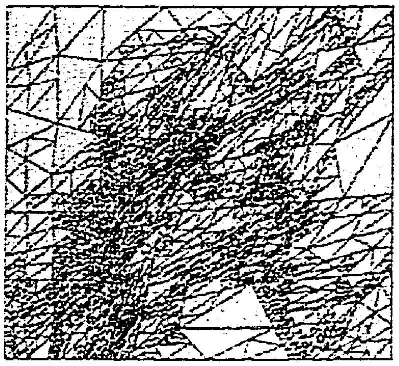


Figure 4.5: A representation of the Lenna image, partitioned using the triangular partitioning method.

Despite the shape of the blocks one may use, the main differences between blocks will be along the edges, since local self-similarity at different scales is unlikely to be perfect. Fisher proposed that the only method of solving this problem, without postprocessing is to use an algorithm which is sensitive to the edges. He went on to describe the algorithm as follows.

1: Vectorize all edges of an image, forming chains of points.

2: Encode the edge chains using a 2-dimensional fractal method, which will map one arc to another. Create domain and range blocks with the two arcs, respectively.

3: Calculate the optimal grey-level scaling and offset for each block.

4: Since all parts of the image containing an edge will be partitioned, those parts of the image remaining can be partitioned using any of the methods previously discussed.

4.2: Improved Searching Methods

In the previously mentioned methods, except the region based method, the domain pool consisted of all possible domain blocks of a particular shape. It is the exhaustive search of all possible domain blocks that takes the largest portion of time to fractally encode an image. Luckily there are many ways one could speed up this process, for instance, one could simply look for the first occurrence of a domain block that is within a given threshold error value, or one could search through the domain pool in an intelligent manner, which is the topic of the next Subsections.

4.2.1: Lean Domain Pools

In [14] Dietmar Saupe published the results of a quadtree partitioning algorithm which limited its search of domain blocks only to those with a high variance. By noticing that most domain blocks used, in the quadtree algorithm described above, were those with a high level of variance, he proceeded to limit the domain blocks that could be placed in the domain pool to those with a high level of variance. As such, only a small portion of blocks would not have an optimal domain-range matching, but the processing time would be significantly decreased. Figure 4.6a shows the variance plotted against the number of domains and Figure 4.6b shows the 8×8 domain blocks used, in black, in the Lenna image.

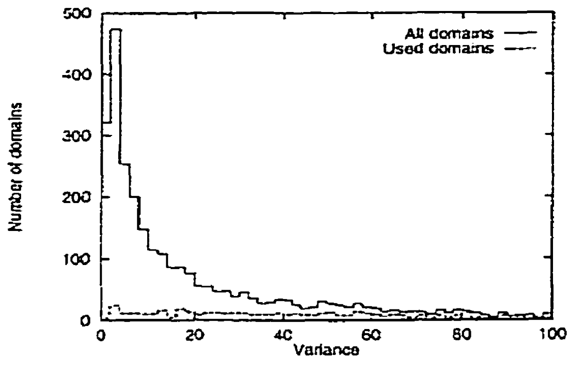


Figure 4.6a: The graph of the number of domain blocks vs variance, for a quadtree partitioning of Lenna.

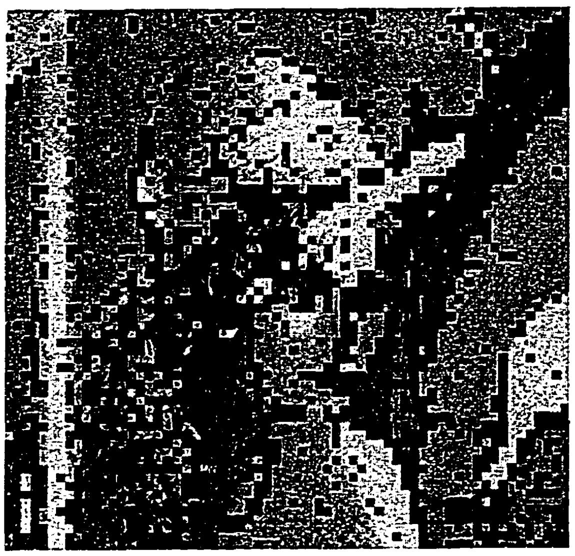


Figure 4.6b: The actual domain blocks used (in black) for the encoding of the Lenna image.

Let α denote the portion of domain blocks which are kept in the pool, therefore as α decreases so will the processing time. This is to be expected since there will be fewer domain blocks to search through in order to find a domain-range block matching. However, since we are using a quadtree approach, the fidelity will actually increase, since some of the large range blocks that could previously be matched must now be subdivided into smaller sub-squares. By increasing the number of range blocks, one increases the quality of the decoded image, but decreases the compression ratio, even though it is only slightly in this case.

In order to increase the compression ratio back to that of the brute force approach, Saupe found a more efficient way of storing the domain indexes. If one considers a 512×512 pixel image with domain blocks of size 8×8, then there are $4096 = 2^{12}$ domain blocks in total. Storing the index value of so many blocks requires 12 bits per block. Since only a fraction of the total number of blocks are used, say 1000, we can use a more efficient index storing system, namely the *white block skipping method, wbs.* Think of the 1000 domain blocks as a bitmap, then the wbs method, described in [17], describes a way of encoding a bitmap as follows: If the whole bitmap is white, mark it as 0 and stop, otherwise, mark it as 1 and partition it in a quadtree manner. Repeat this for each subblock of the bitmap in a counter clockwise direction until you have reached a single value which is either encoded as a 0 for white, or 1 for non-white. For a better understanding see Figure 4.7.

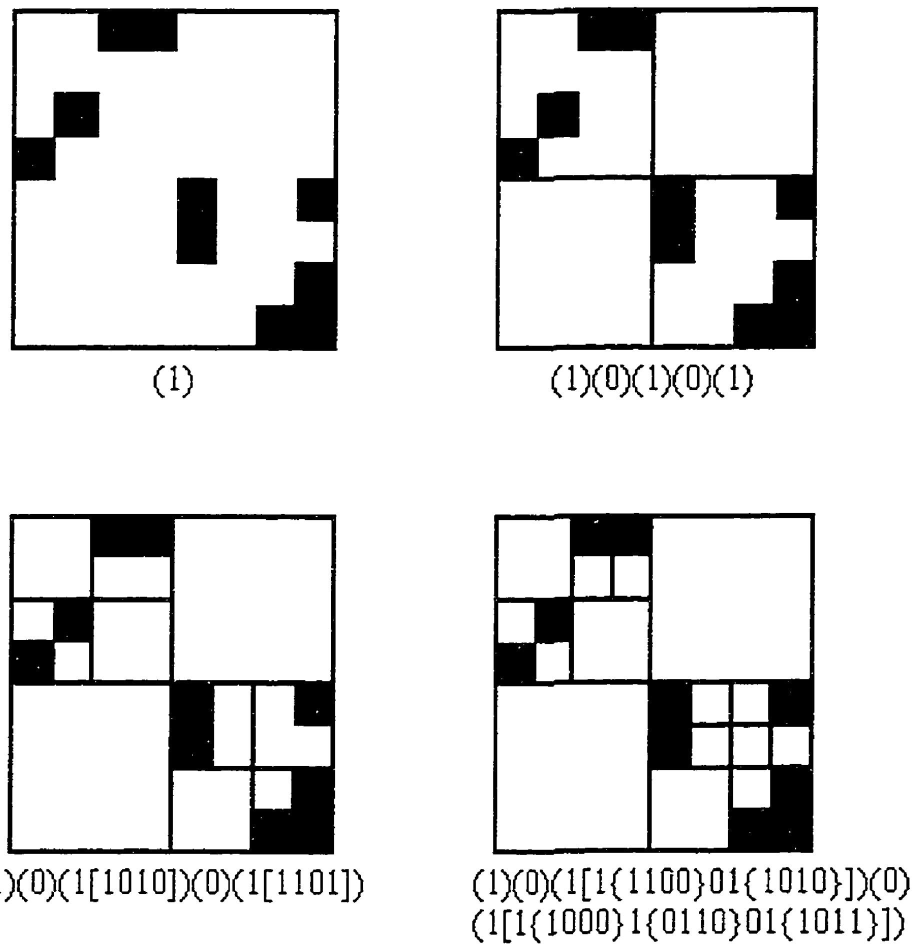


Figure 4.7: An example of the actual derivation of the wbs code.

Experimental results suggest that for $0.7 < \alpha \leq 1$, the above encoding is worse than the standard methods, but is better for values of $\alpha \leq 0.7$. With a value of $\alpha = 0.30$ we achieve a compression ratio of 14.88, which is compatible to the brute force approach, and a PSNR which is better by 0.2 dB. The real advantage to this method is with regards to the encoding time, for the brute force approach Saupe reported compression times of 15.2 seconds, while for the method described above he reported compression times of 6 seconds.

Such a method can be adjusted to work with other compression algorithms and is extremely useful in situations where some quality can be compromised for a significant increase in encoding time.

4.2.2: Points in an Abstract Space with the FFIC Algorithm

Let us consider a block of pixels as a point which can be mapped to an abstract space, such that any two points which are close in the space are perceptually similiar to the human observer. In doing so, one could find the optimal domain-range block matching very quickly, simply by considering those domain blocks which are close to the range block in the space.

John Kominek [10], developed an algorithm he called the *FFIC algorithm*, which uses this idea. The FFIC algorithm begins by partitioning an image in the same way as the brute force approach, ie: non-overlapping range blocks whose union covers the entire image, and overlapping domain blocks twice the size of the range blocks, such that their union covers the entire image. In order to map the blocks into the same space, the domain blocks must be subsampled to the size of the range blocks. We can now consider each $n \times n$ block as an n^2 dimensional vector.

Since we compare each possible affine transformation of a domain block for each range block, we must take this into account when converting our blocks to points. This can be done by normalizing each block so that its pixels have a fixed mean and variance.

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i' = c_1 \qquad var = \frac{1}{n}\sum_{i=1}^{n}(x_i' - \overline{x})^2 = c_2$$

Where x_i' are the normalized pixel values. Thus, when all of the blocks are mapped to the space, any two blocks which are close together will be perceptually similar to the human observer, through some affine transformation.

The FFIC algorithm uses r-trees to map the blocks to the space and limit the domain blocks to be compared, for any particular range block. An r-tree is a data-structure capable of efficiently indexing a multi-dimensional space, for a better understanding see [7]. For the FFIC algorithm, the r-tree basically groups the domain blocks into nested sets of rectangles. Then, for any given range block, the algorithm finds the rectangle in which it belongs and compares the range block to those domain blocks which are in the same rectangle. See the example in Figure 4.8 for a better understanding.

The distance metric used to compare domain and range blocks is a question of choice, but the absolute error and the root mean square metrics are satisfactory.

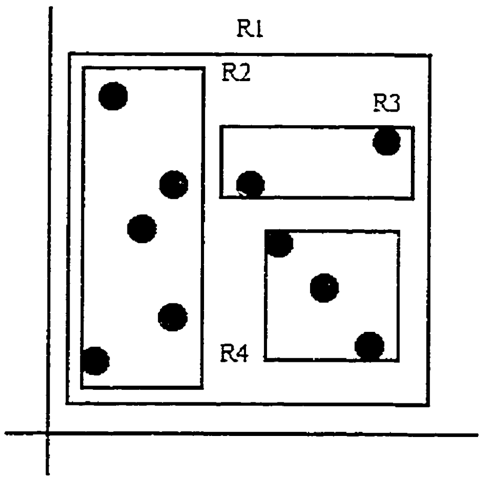
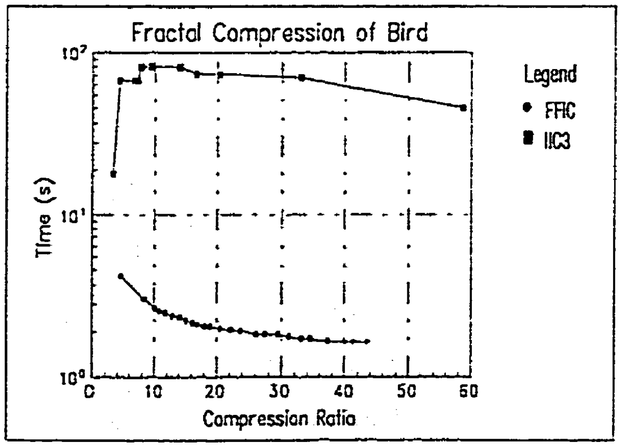


Figure 4.8: An example of an r-tree grouping of domain blocks.

Komenik tested his algorithm on a 486DX2-66 with 16MB RAM and concluded, among other things, that a branching factor of 16 is satisfactory. So as to have a fair evaluation of his method, he compared the compression speed, fidelity and compression ratios achievable to that of the leading fractal image compression program at the time, 1995. As Figure 4.9a shows, for the bird image (Figure 4.9b), the FFIC algorithm encodes much more quickly, about 30 times faster, than the *IIC3*[1] program. Figure 4.10 is a graph of the rms error vs the compression ratio of the same bird image for the FFIC, IIC3 and LBF[2] algorithms. Although the fidelity of the FFIC algorithm is not always better than the IIC3, it does perform well and is faster than the LBF algorithm.

[1] IIC3 is the acronym for *Iterated Systems Incorporated* Images Incorporated III program, which was run on the setting "best".

[2] LBF stands for *light brute force*, which is the same as the brute force algorithm, except that the domain blocks do not overlap.



Figures 4.9a and 4.9b: Time needed to encode the bird image using the FFIC and IIC3 methods, and the original bird image.

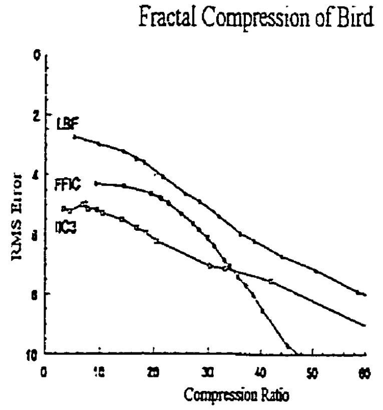


Figure 4.10: The graph of the RMS error vs Compression ratio for the LBF, FFIC and IIC3 algorithms.

Figure 4.11 gives a visual comparison between the FFIC, IIC3 and LBF algorithms. Although the FFIC algorithm has a small rms error, it tends to blur contours and becomes blocky at a compression ratio of 30:1.

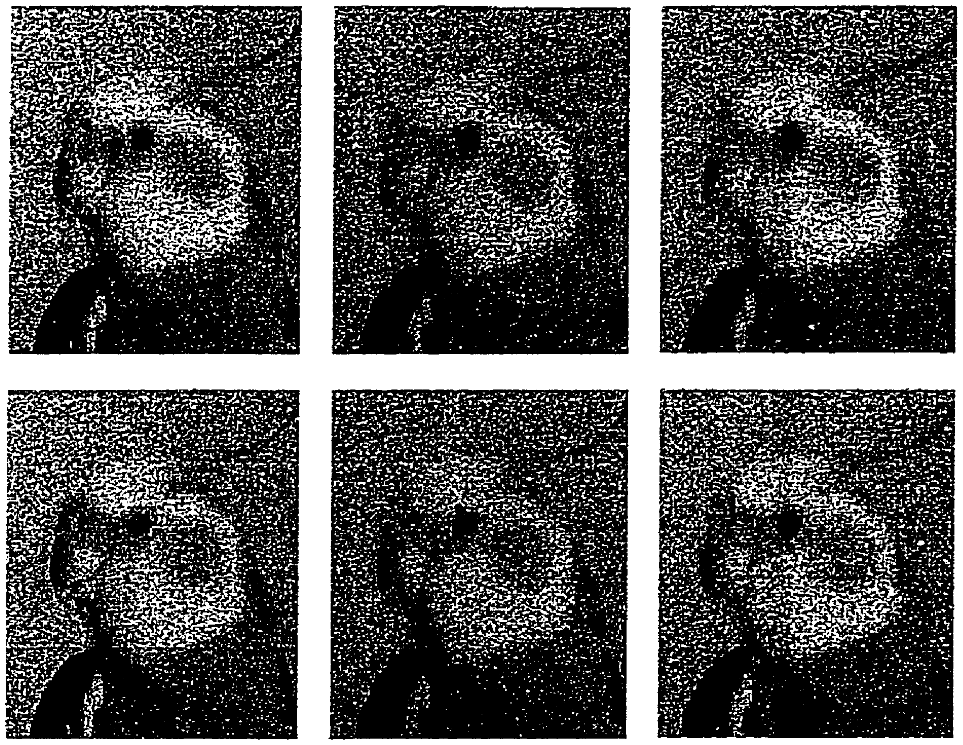


Figure 4.11: Visual comparison of the FFIC, IIC3 and LBF algorithms.
Top Left: LBF 20:1, rms 4.06
Top Centre: IIC3 20:1, rms 6.21
Top Right: FFIC 20:1, rms 4.63
Bottom Left: LBF 30:1, rms 5.40
Bottom Centre: IIC3 30:1, rms 7.05
Bottom Right: FFIC 30:1, rms 6.12

4.3: Conclusion

In this Chapter we introduced various methods of improving the compression ratio, image fidelity and encoding time, compared to the brute force method. We discussed the different effects obtained by varying the block shapes and have presented some non-trivial methods of efficiently searching for a satisfiable domain-range block pairing.

CHAPTER 5

Fractal Compression of Video Sequences

As most people know, video sequences give the appearance of continuous smooth motion by displaying a sequence of still images, known as frames, at a certain rate. With this in mind, there are a few obvious ways in which fractal image compression can be extended to fractal video compression. For instance, one could encode each frame individually, or one could encode a subset of the total frames together by considering three dimensional domain and range blocks, such that time is the third dimension. Each has its potential advantages and each is applied very differently than the other, and as with fractal still image encoding methods, each can be done differently. In this Chapter we will discuss different methods that have been developed using three dimensional blocks and encoding each frame independently.

5.1: Inter/Intraframe Fractal Video Encoding

There are methods which fractally encode the first frame, then will approximate the subsequent frames using domain regions from the current frame, intraframe, and/or previous frames, interframe. By doing so, they can improve the image quality and/or compression ratio. The following subsections describe methods which use these techniques.

5.1.1: Simple Motion Compensation

A very simple motion compensation fractal video algorithm would be to partition each frame in the same way, regardless of the image. Fractally encode the first frame using any method which does not change the domain or range block shapes. This can be the brute force algorithm, or a simple improvement to it using an improved searching method. Compare frames x_i and x_{i-1} and let T denote the set of range blocks which are significantly different, ie: greater than a given error threshold. Such a comparison can be very quick, for instance one may simply find the difference in grey-scale value for each pixel. The range blocks who's difference in pixel values is greater than the threshold can then be placed in T. In order to encode frame x_i, we do not have to encode all of it, but rather only those range blocks which belong to T. Doing so will greatly increase the compression ratio while effecting the quality of the decoded image very little. Notice that we can still control the decoded images quality by adjusting the error threshold for the difference in grey-scale values of consecutive frames.

There are a few options with regards to the domain pool to be considered for the encoding process. We could include those domain blocks of the current frame and/or those of the previous frame or frames. Obviously, a larger domain pool will result in better quality decodings, but will also slow down the encoding time.

Since the partitioning is independent of the image and the domain and range blocks are of a fixed size, this method is not practical. It is however, the basic idea of much better algorithms which improve its compression ratio, encoding time and resulting image quality and, as such, allow for practical approaches to fractal video compression.

5.1.2: Low Rate Video Coding

Bernd Hürtgen and *Peter Büttgen*[8] developed a practical fractal video compression algorithm which is based on the method described in Section 5.1.1. In their research, they concentrated on low-rate video coding with applications in mobile video telephony, teleconferencing and narrow band ISDN distributed audio-visual services from 4.8 to 64 Kb/s.

Their method uses the difference in grey-scale values d_n between the current frame x_n and the previously decoded frame x'_{n-1}, in order to determine which regions must be considered in the next encoding. Since the value of $d_n = x_n - x'_{n-1}$ shows those areas which have changed in grey-level value, we can divide the frame into two regions. The first region is referred to as the background and consists of the union of regions for which the decoded image (which they refer to as the prediction) is satisfactory. The other is called the foreground and consists of those areas where the prediction is unsatisfactory. We can then concentrate solely on encoding and transmition of the foreground image.

The domain pool which Hürtgen and Büttgen use consists only of those blocks in the current frame, both background and foreground. It is important not to use only those domain blocks from the foreground since the foreground may be very small at times, thus resulting in a poorer quality decoding. Using blocks from both the foreground and background also increases the flexibility of the encoding scheme regardless of the size of the foreground.

The partitioning method they employ is different from those that have been discussed in this dissertation, but is based on the simple quadtree approach with variable block sizes of 4×4, 8×8 and 16×16. The partitioning is performed prior to and independently of the encoding of the image. Doing so increases the processing speed since only those blocks chosen by the segmentation must be considered, and forces one to incorporate a priori knowledge into the segmentation algorithm. The way in which they incorporate this knowledge is by employing a gain/cost criterion, which is basically the trade off between the cost of encoding a certain block and the gain in the reduced visual error of doing so. The aim of the segmentation is to find the foreground image which maximizes the total gain/cost relation. It is important to notice that the cost of encoding a block is the same for all blocks, therefore, the only criterion that must be considered is the gain.

Prior knowledge is incorporated in the method by estimating the reconstruction error, or decoding error, for each block. Hürtgen and Büttgen showed that this reconstruction error is highly correlated with the block size, and thus estimate it by only considering the block size. They go on to suggest that considering the grey-level distribution as well as the block size would yield improvements.

For a better understanding of the segmentation procedure, let us denote an N_x by N_y pixel image by an N-dimensional point $x = (x_1, x_2, ..., x_N)^T = (x_i)^T \in \Re$ such that $1 \leq i \leq N = N_x \cdot N_y$ and x_i represents the grey-levels of the image.

Let $b^{(j)}(x) = (x_{j_k})^T$, $st\ x_{j_k} \in \Re$ and $j_k \in \aleph^{(j)}$, be the j^{th} block within image x, where $\aleph^{(j)}$ denotes the set of all indices of elements belonging to the j^{th} block and $1 \leq k \leq M = M_{b_x} \cdot M_{b_y} < N = N_x \cdot N_y$ is the number of elements within the block.

Let $P(x)$ denote the power (level of error) of an image, and let $P(b^{(j)}(x))$ denote the power of the j^{th} block within the image x. Therefore using the euclidean norm we can write

$$P(x) = \sum_{i=1}^{N} |x_i|^2 = ||x||^2 \qquad P(b^{(j)}(x)) = \sum_{j_k \in \aleph^{(j)}} |x_{j_k}|^2 = ||b^{(j)}(x)||^2$$

Recall that we only want to encode those blocks that belong to T, ie: those which are in the foreground. Notice that the total number of blocks N_B which can be encoded is limited by the maximum data rate and the segmentation overhead.

Begin by calculating the power $P_d(b^{(j)}(d_n))$ for each block $b^{(j)}(d_n)$ which is in the foreground. Then calculate the reconstruction error $P_c(b^{(j)}(x_n))$, using the following Equation.

$$P_c(b^{(j)}(x_n)) = p_1 \cdot M^{p_2}$$

where M is the block size and p_1 and p_2 are determined so as to maximize the reconstruction quality. Through experimental results Hürtgen and Büttgen found that p_1 and p_2 only needed to be adjusted once.

The coding improvement is now calculated for each block by using the Equation

$$\Delta P^{(j)} = P_c(b^{(j)}(x_n)) - P_d(b^{(j)}(d_n))$$

If $P_c(b^{(j)}(x_n)) \geq P_d(b^{(j)}(d_n))$, ie: the error introduced by encoding a block is greater than the prediction error, then the block is not taken into consideration for encoding. If $P_c(b^{(j)}(x_n)) < P_d(b^{(j)}(d_n))$, then encoding the block will yield an improvement and is therefore a valid candidate for encoding. However, since we are limited in the number of blocks we can encode, we must then sort the blocks in descending order of improvement $\Delta P^{(j)}$, and only encode the first N_B blocks. This process is then repeated iteratively.

Note: Since we are using a quadtree structure of block partitioning, we are limited in the choice of blocks we can take. Figure 5.1a shows the quadtree segmentation while Figure 5.1b shows the corresponding fractal encoded foreground regions.

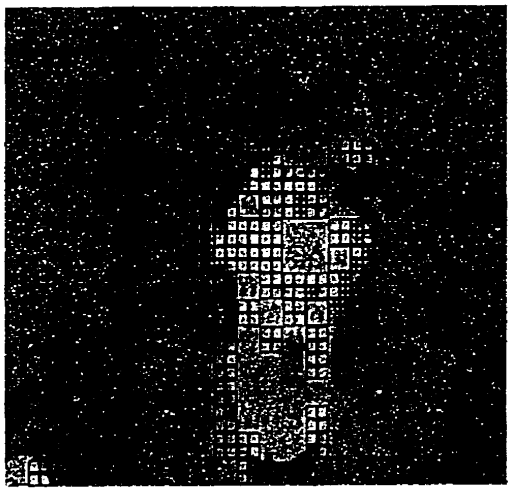


Figure 5.1a: The quadtree segmentation of the foreground region.

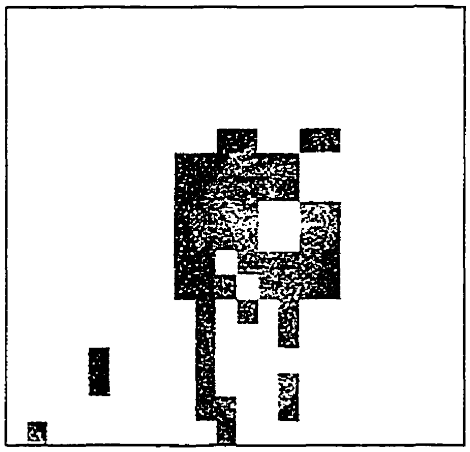


Figure 5.1b: The fractal encoded foreground region.

Hürtgen and Büttgen claim that with a 64 Kb/s ISDN B-channel, a frame rate of 25 Hz and only encoding every third frame (the skipped frames are interpolated at the receiver) we only have 7680 bits of data for each frame. Using the quadtree segmentation requires approximately 1000 bits of overhead, thus leaving approximately 6700 bits of data with which we can encode the foreground image. They tested their method on the Miss America test sequence and obtained a quality of 34-35 dB, showing that their method can encode typical videophone sequences, with reasonable quality, at the practical data rate of 64 Kb/s.

5.2: Three Dimensional Iterated Function Systems

Perhaps the most obvious extension of fractal image compression to fractal video compression, is to add an extra dimension representing time. We can then encode the video sequence as a 3D object which is bounded in two dimensions (spacial dimensions x and y) and unbounded, practically, in the third temporal dimension z. Notice that we can take z to be any finite positive number, if $z = 1$ then it is considered to be a still image, otherwise it can be considered a video sequence. The mathematical analysis showing why such methods work is extremely similar to the analysis shown in Chapter 2 and as such, is not discussed any further.

5.2.1: 3D Fractal Block Coding of Video Sequences

M.S. Lazar and *L.T. Bruton*[11] have developed an algorithm which will use 3D *pifs* to encode video sequences. They consider ten frame sequences at a time and encode them using three dimensional range and domain blocks. The range blocks are of size $B \times B \times T$, where B represents the spacial length of the frame and T the temporal length of the sequence. The domain blocks are taken to be of size $M_1 \cdot B \times M_2 \cdot B \times M_3 \cdot T$, where M_i is a scaler for $i = 1, 2, 3$.

Each range and domain block is chosen from an R-Frame and D-Frame respectively, which consist of consecutive non-overlapping groups of input frames. Each R-Frame is associated to a D-Frame such that the R-Frame is physically inside the D-Frame, but both end at the same temporal location. Notice that a D-Frame can start prior to its associated R-Frame and is limited in spacial size by the size of the frames and limited in temporal size by the number of frames from the first frame to the current. The temporal limitation on the D-Frames may be too large and as such we must limit it in temporal length by $j \cdot M_3 \cdot T$ frames such that $j \in \aleph\backslash\{0\}$.

The indexing of the frames within the R-Frames and D-Frames begins at the most recent frame and increases as we move backwards. We also denote the R-Frame and D-Frame beginning at time t by R-Frame(t) and D-Frame(t) respectively. See Figure 5.2 for a better understanding.

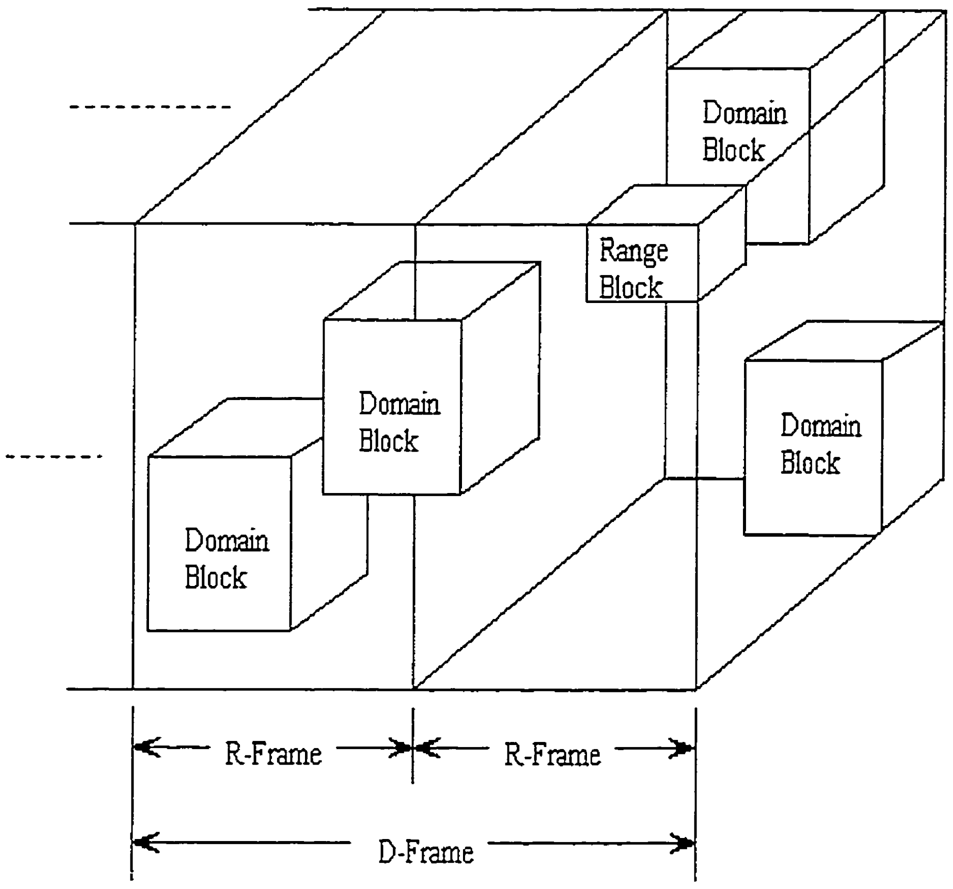


Figure 5.2: An example of an R-Frame, D-Frame, range block and domain block.

Since the motion on most video signals changes smoothly, which is especially true for teleconferencing, each frame will be similar to the last. Thus, we do not have to consider all of the possible pixel shuffling operations, of which there are obviously many more than in 2D fractal encodings. Lazar and Bruton limited the number of isometries by only considering transformations which were the result of first performing intra-frame transformations, then inter-frame transformations. Doing so greatly reduces the number of possible isometries and brings them into the practical realm. We can now describe all possible isometries by $S(I) = S_{inter}(I) + S_{intra}(I)$. The intra-frame isometries are the same as in the 2D case but the inter-frame isometries are limited to two kinds, the frames which remain unaltered and the frames who's order is reversed.

Since the search for a reasonable domain-range block matching is what basically determines the speed of the encoding, of which there are too many possible combinations in our case, Lazar and Bruton employ an efficient searching method. They restricted their search to those domain blocks which are physically near the given range block and denote the matching function by $N(I)$. Therefore if the address of the range block is (N_1, N_2, N_3), then only those domain blocks whose addresses are given by $(N_1 + k_1 \cdot L_1, N_2 + k_2 \cdot L_2, N_3 + k_3 \cdot L_3)$ are considered, such that Where $-K_i \leq k_i \leq K_i$, $i = 1, 2, 3$, (K_1, K_2, K_3) are fixed for all R-Frames and (L_1, L_2, L_3) are the search step sizes. Notice that in order to address a domain block we need only give (k_1, k_2, k_3).

In order to improve the decoded image quality of the frames, they use a three dimensional equivalent to the quadtree splitting method as well as a temporal splitting method. Range blocks can either be split spatially by four or temporally by two depending on the distribution of errors in the original range block and the overall encoding error. When the encoding error for a given block is greater than a fixed threshold, the block is split in one of the two ways. If the errors are distributed evenly throughout the frames of the range block, then the block is split spatially into four equally sized blocks who's depth remains unchange. If the errors are not evenly distributed throughout the frames, then the block is split temporally in half, where the spacial size remains the same but the temporal size is half of the original block. For a better understanding see Figure 5.3.

The method in which the frame encoding errors are determined is by computing the distance between each frame from the encoded block $r_u^{'}$ and the original frames from the block r_i. If the normalized difference between the maximum and minimum of these distances $d(r_i^{'}, r_i)$ is greater than a given threshold, then the error distribution is said to be uneven, otherwise it is said to be evenly distributed.

Once the encoding is done, we can decode each R-Frame in a similar iterative manner as that used for 2D encodings. Notice however that we need all of the frames corresponding to the appropriate D-Frame so as to decode a given R-Frame. If there is data outside of the R-Frame which is contained in the D-Frame, then such data will also be required for decoding the respective R-Frame, but it will not be iterated. The range blocks who's corresponding domain block is completely contained in such an area, outside of the R-Block, can be decode in a single iteration.

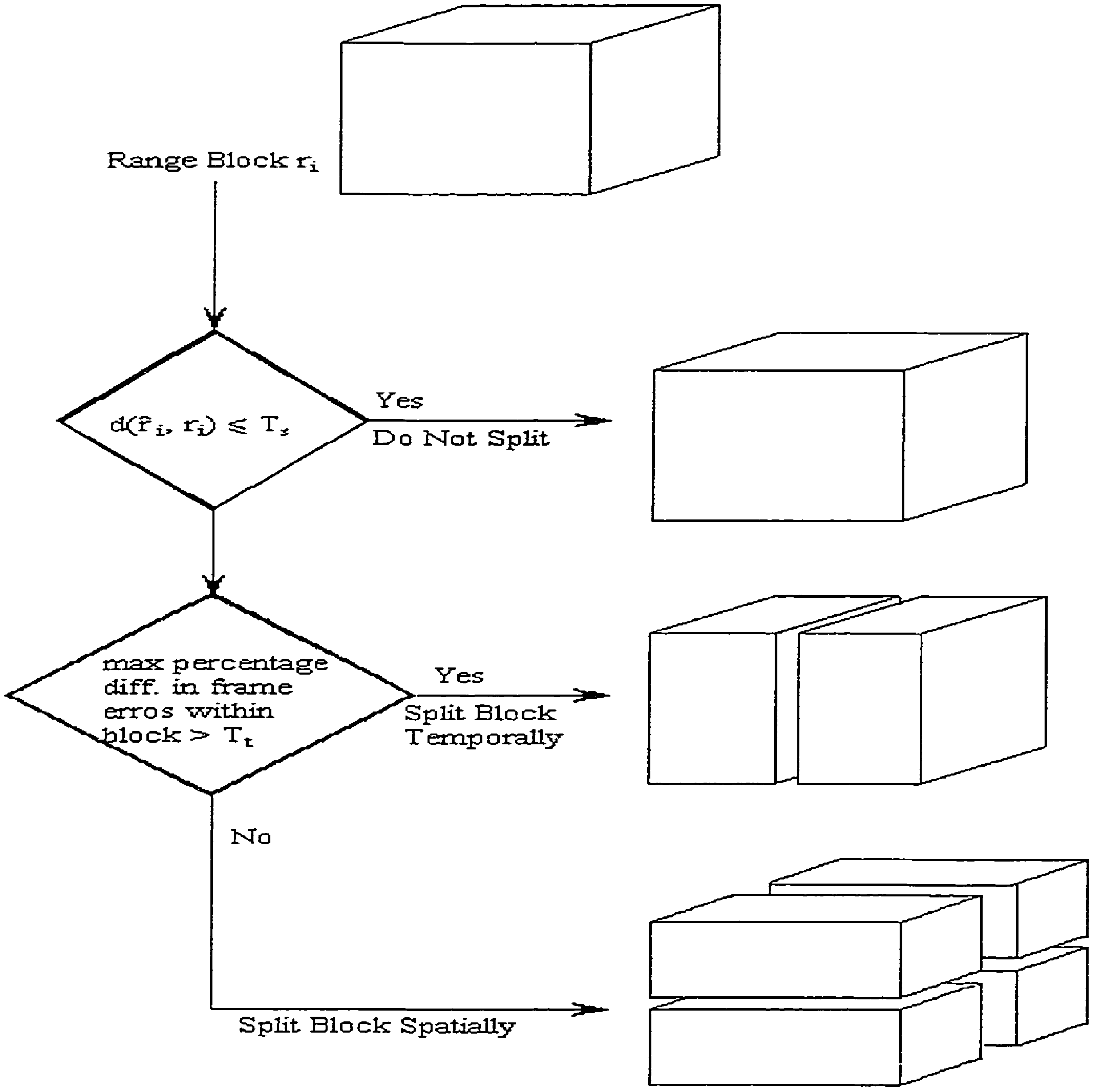

Range Block r_i
$d(\hat{r}_i, r_i) \leq T_s$
Yes
Do Not Split
max percentage diff. in frame erros within block > T_t
Yes
Split Block Temporally
No
Split Block Spatially

Figure 5.3: Flowchart of the algorithm used to determine how to split a given range block.

Since adjacent frames of a video sequence are usually similar, it is better to begin with the previous frame as an initial frame for the current frame and then iterate, ie: The final iteration for R-Frame($t - k \cdot T$) is the same as the first for R-Frame(t).

Using experimental results on the standard salesman and Miss America video sequences, Lazar and Bruton show that their method results in a compression ratio of 40:1 to 77:1, with an acceptable image quality for teleconferancing. They admit however, that the decoded video sequence has a jerkiness effect on those areas of rapid movement. Because of the high computational costs and approximately fixed cost of decoding, their algorithm would be better applied to those applications which need only be encoded once, but decoded many time. An example of this is multi-media.

5.2.2: Improved Searching and Classification Methods for Fractal Volume Compression

Cochran, Hart and Flynn [3] have also researched fractal image compression of video sequences using 3D blocks, and have developed a better method than the one discussed in the previous Section. Their method uses a combination of classification by principal component analysis, a down-sampled nearest neighbour search and macro blocks.

$\lambda_2' - \lambda_1' \leq \epsilon$	$\lambda_3' - \lambda_2' \leq \epsilon$	$\lambda_3' - \lambda_1' \leq \epsilon$	class
true	*true*	*true*	midrange
true	*true*	*false*	mixed edge
true	*false*	*false*	double edge
false	*true*	*false*	single edge
false	*false*	*false*	mixed edge

Table 5.1: The five block categories.

They use the principal component analysis of volumetric blocks to classify the blocks into one of five classes, shade, midrange, mixed edge, double edge and simple edge. Doing so greatly reduces the searching time for an adequate domain-range block pairing. This method begins by calculating the value $v(x)$ of a scalar volumetric dataset of the point $x \in \Re^3$, ie: a 3D domain or range block. The *total mass* M of the block is then defined as $\sum_{x\in\Re} v(x)$ and the centroid $c \in \Re$ as $\frac{1}{M}\sum_{x\in\Re} x \cdot v(x)$. The matrix $S = \sum_{x\in\Re} v(x)(x-c)(x-c)^T$ is then found, along with its associated eigenvalues $\lambda_1 \leq \lambda_2 \leq \lambda_3$ and eigenvectors $\vec{v}_1, \vec{v}_2$ and $\vec{v}_3$. It is important to notice that S is a symmetric 3×3 positive semidefinite matrix, as such, its eigenvalues $\lambda_i,\ i = 1,2,3$ will be non-negative real numbers. We then go on to normalize the eigenvalues as $\lambda_1' = \lambda_1/\lambda_3, \lambda_2' = \lambda_2/\lambda_3$ and $\lambda_3' = 1$ and find a threshold value $\epsilon \leq 1$. Doing so will allow us to place each block into one of the five categories according to their normalized eigenvalues using Table 5.1.

Classifying the blocks in such a way obviously decreases the total number of comparisons performed and consequently reduces the search time considerably. Although when considering mixed edge blocks one must consider all 48 possible isometries. Cochran *et/al* developed a method of reducing the number of isometries considered for single edge and double edge range blocks by considering vectors associated with them. For simple edge blocks, the vector $\bar{w}_1$ is found such that it is normal to the plane that fits the most dense region of the block, where as for double edge blocks, the vector $\bar{w}_2$ is taken as being parallel to each significant edge in the block. Thus, when we are considering an isometry I_i, we can find the eigenvector affected by this isometry and denote it as $I_i(\bar{v}_D)$, such that,

$$\bar{v}_R = \begin{cases} \bar{w}_1 & if\ R\ is\ a\ simple\ edge\ range\ block \\ \bar{w}_2 & if\ R\ is\ a\ double\ edge\ range\ block \end{cases}$$

$$\bar{v}_D = \begin{cases} \bar{w}_1 & if\ D\ is\ a\ simple\ edge\ range\ block \\ \bar{w}_2 & if\ D\ is\ a\ double\ edge\ range\ block \end{cases}$$

We only continue to consider an isometry if $I_i(\bar{v}_D)$ and $\bar{v}_D$ are approximately parallel, or perpendicular, ie: $|\bar{v}_R \cdot I_i(\bar{v}_D)| \approx 1$.

In their test results the authors showed that using this classification scheme only reduces the image quality by a few tenths of a decibel, while decreasing the encoding time significantly.

To further decrease the encoding time, Cochran *et/al* also apply a nearest neighbour searching method. Doing so increases the search space and thus increases the fidelity of the decoded image, however, since we are dealing with 3D blocks, the dimensionality of the search space increases drastically. In order to keep things at a manageable size, the authors suggest down sampling the blocks first, which will reduce the dimension of the *kd-tree* used. Although doing so does decrease the dimension to a manageable size, we are no longer guaranteed an optimal match but only an adequate match. We then need only search through the n nearest neighbours of a range block to find the optimal match from those, or we can take the first acceptable match.

Using such a nearest neighbour search is essential to reducing the encoding time from hours to minutes. The authors did not test the effects of such a search on image fidelity (because of the huge computational cost), but they claim it to be negligible. The algorithm can be further enhanced with the use of macroblocks or localized searching.

Using localized searching, one can search only a small number of spatially close blocks. Using the identity transformation and a brute force approach, we can divide the blocks into two groups. The first and second groups consist of those blocks for which the comparison was acceptable, and those that were not, respectively. We can then perform a second search through the second group, considering all possible transformations. For this search to be manageable, we can employ one of the improved searching methods previously discussed.

To use macroblocks, one must divide the whole volumetric data into large blocks and work with the blocks individually. Finding the kd-tree for each block increases the compression time, but also increases the compression ratio. Experimental results conducted by the authors show that the loss in image fidelity, because of the limited domain pool, is only a few tenths of a decibel. Figure 5.4 shows the fidelity, for different compression ratios, of the above algorithm.

5.3: Conclusion

In this Chapter we have shown how easily fractal image compression can be extended to fractal video compression. Both motion compensation and three dimensional iterated function systems have advantages over the other and over other encoding techniques, but both remain to be fully developed. Because of the short decoding times but lengthy encoding times, fractal video compression techniques, like fractal still image techniques, are better suited for multi-media applications. Even though fractal video compression is only in its early stage of development, it surpasses Vector Quantization methods and is within one decibel PSNR of the Domain Cosine Transform techniques.

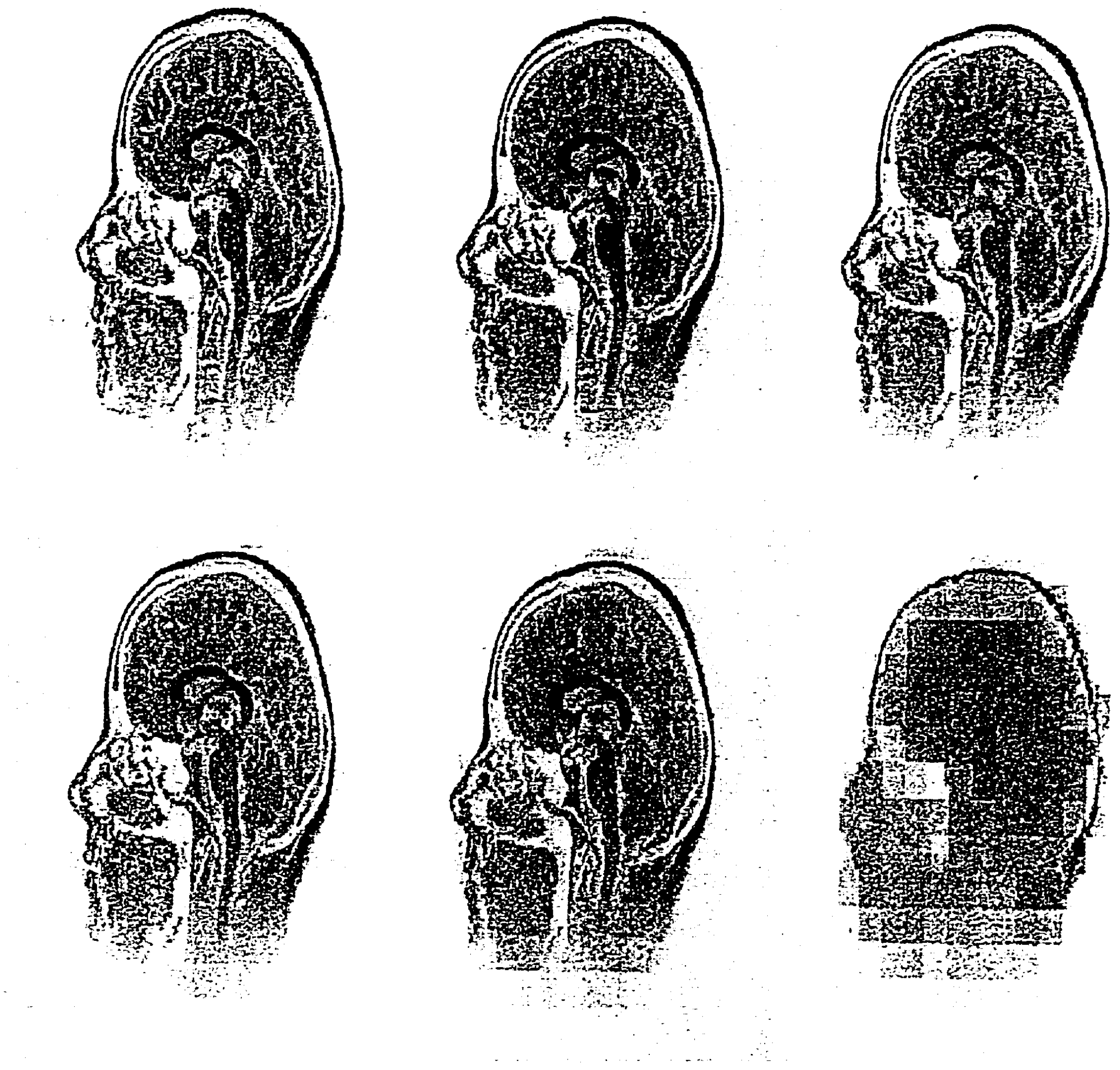


Figure 5.4: One frame of a video sequence of an MRI. Original (upper left), 20:1 (upper center), 25:1 (upper right), 30:1 (lower left), 43:1 (lower center), 729:1 (lower right).

References

[1] M.F. Barnsley; "Fractals Everywhere," Academic Press, San Diego, 1988.

[2] M.F. Barnsley, L.P. Hurd; "Fractal Image Compression," AK Peters, Wellesley MA, 1993.

[3] Wayne Cochran, John Hart, Partick Flynn; "Fractal Volume Compression," IEEE Transactions on Visualization and Computer Graphics, vol. 2, no. 4, pp. 313-322, December 1996.

[4] Yuval Fisher; "Fractal Image Compression. Theory and Applications," Springer Verlag, New York, 1994.

[5] Yuval Fisher; "Fractal Image Compression," SIGGRAPH 92 Course Notes, 1992.

[6] Denny Gulick; "Encounter with Chaos," McGraw-Hill, 1992.

[7] A. Guttman; "R-trees: A Dynamic Index Structure for Spatial Searching," Proceedings of ACM SIGMOD Conference on Management of Data, pp. 47-57, 1984.

[8] Bernd Hürtgen, Peter Büttgen; "Fractal Approach to Low Rate Video Coding," Proceedings from SPIE Visual Communications and Image Processing, vol. 2094, pp. 120-131, 1993.

[9] B. Hrtgen, P. Mols, S. F. Simon; "Fractal Transform Coding of Color Images," Proceedings of the International Conference on Visual Communications and Image Processing, SPIE 94, vol. 2308, pp. 1683-1691, Chicago, Illinois, USA, 1994.

[10] John Kominek; "Algorithm for Fast Fractal Image Compression," Proceedings of SPIE, Volume 2419,1995.

[11] M.S. Lazar, L.T. Bruton; "Fractal Block Coding of Digital Video Coding," IEEE Trans. Circuits and Systems for Video Technology, vol. 4, no. 3, pp. 297-308, 1994.

[12] H.L. Royden; "Real Analysis," Macmillan, New York, 1988.

[13] W. Ruden; "Real and Complex Analysis," McGraw-Hill, New York, 1972.

[14] Dietmar Saupe; "Lean Domain Pools for Fractal Image Compression," Proceedings of SPIE Electronic Imaging 96, Science and Technology Still Image Compression II, vol 2669, 1996.

[15] Lester Thomas, Farzin Deravi; "Region-Based Fractal Image Compression Using Heuristic Search," IEEE Transactions on Image Processing, vol. 4, no. 6, June 1995, pp. 832-838.

[16] Cristopher J. Wein, Ian F. Blake; "On the Performance of Fractal Compression with Clustering" IEEE Transactions on Image Processing, vol. 5, no. 3, March 1996, pp. 522-526.

[17] R.E. Woods, R.C. Gonzales; "Digital Image Processing," Addison-Wesley, Reading, 1992.

IMAGE EVALUATION
TEST TARGET (QA-3)