REDUCED SEARCH FRACTAL BLOCK CODING USING FREQUENCY SENSITIVE NEURAL NETWORKS

by

Larry Marvin Wall

A Thesis presented to the University of Manitoba in

partial fulfillment of the requirements for the degree of

Master of Science

in the

Department of Electrical and Computer Engineering.

Winnipeg, Manitoba

May, 1993

REDUCED SEARCH FRACTAL BLOCK CODING USING FREQUENCY SENSITIVE NEURAL NETWORKS

BY

LARRY MARVIN WALL

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

Lossy signal compression based on *fractals* has attracted a great deal of attention since the introduction of *iterated function systems* (IFSs), a compact fractal representation scheme for complex *self-affine* structures. IFSs are an example of a more general class of emerging fractal coding techniques referred to as *collage coding*. These techniques are all based on a corollary of *contractive transformation theory* called the *collage theorem*. The problem with collage coding techniques is the high computational complexity of the encoding procedure which is even NP complete for IFSs. Recently, a less compact but more manageable collage coding technique called *fractal block coding* (FBC) has been introduced for grey scale images. Using a *divide-and-conquer* encoding strategy, images can be compressed with FBC in known polynomial time. However, this technique still requires a $O(n^4)$ search to encode nxn pixel images. This thesis develops a reduced search FBC encoding procedure employing *neural networks*. A neural network paradigm known as *frequency sensitive competitive learning* (FSCL) assists the encoder in locating fractal *self-similarity* within a source image. For an network of appropriately chosen size this decreases the time complexity of the encoding procedure to $O(n^3)$. For 256x256 images, compression times are improved by a factor of 45 and image quality is reduced by less than 0.2 dB. The reduced search FBC encoding and decoding procedures were implemented as part of a *concatenated* image compression scheme with FBC as the *inner* code, and *arithmetic entropy coding* as the *outer* code. This addition of arithmetic coding improves compression ratios by up to 20% without further effecting image quality. Using the concatenated FBC/arithmetic compression scheme, grey scale images were compressed at ratios in excess of 18:1 with *peak signal-to-noise ratios* (PSNR) of up to 31.0 dB.

ACKNOWLEDGEMENTS

I would like to begin by thanking Dr. W. Kinsner for proposing this topic and giving me his time and patience throughout the course of this thesis. I can think of no other professor who takes such an active interest in every student that sets foot in his classroom or devotes more of his own time to making this university a better place.

I would also like to acknowledge everyone in room 432 engineering, past and present, including Adi Indrayanto, Geoff Stacey, Tom Tessier, Warren Grieder, Armein Langi, and Ken Ferens. Their technical advice and friendship over the past two years have been invaluable contributions to this thesis. I would also like to thank Cam Mayor, Paul Chan, and 'Woj' Ng who, through a variety of discussions (often at 3 am), may or may not have contributed directly to this thesis but did help to preserve my sanity.

Finally, this thesis is dedicated to my Dad. Having a high school science teacher for a father may not always have been to my liking, but it did instill in me a sense of scientific curiosity and an appreciation for learning which has no doubt placed me on my present career path. In this sense, he may have contributed more to this thesis than any other individual. He's also one hell of great guy.

This work was supported in part by the National Sciences and Engineering Research Council of Canada and Manitoba Telephone Systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

CAT	Contractive Affine Transform
CD ROM	Compact Disk Read Only Memory
DCT	Discrete Cosine Transform
DWT	Discrete Wavelet Transform
FBC	Fractal Block Coding
fps	Frames per second
FSCL	Frequency Sensitive Competitive Learning
HDTV	High Definition Television
IFS	Iterated Function System
ISDN	Integrated Services Digital Network
JPEG	Joint Photographics Experts Group
K	$2^{10} = 1024$
LBG	Linde-Buzo-Gray (clustering algorithm)
LZW	Lempel-Ziv-Welch (lossless compression technique)
M	$2^{20} = 1\ 048\ 576$
MHz	Mega Hertz (million cycles per second)
MRCA	Multiple Reduction Copy Algorithm
MS-DOS	Microsoft Disk Operating System
NTSC	National Television System Committee
PSNR	Peak Signal-to-Noise Ratio
RIA	Random Iteration Algorithm
SNR	Signal-to-Noise Ratio
sup	Supremum
VQ	Vector Quantization

CHAPTER I

INTRODUCTION

The demand for digital images in both non-computing and computing related applications is currently undergoing exponential growth. Present and future applications include facsimile, remote sensing, video conferencing, multi-media, and digital television for use in business, entertainment, education, the graphic arts, medicine, and scientific research. Unfortunately, digital images contain an extremely large amount of data. For example, a single 320 by 200 pixel color photograph requires 62.5 Kbytes of storage and is considered small by todays standards. The NTSC broadcast standard for color television, transmitting at 30 *frames per second* (fps) with 525 scanlines per image requires 4 MHz of bandwidth in analog form. A digital representation of this same signal, sampled at 14.3 MHz would require in excess of 100 Mbits per second. The new HDTV standard for North America, when selected, will almost certainly be digital and will require at least 400 Mbits per second.

The advent of larger computer memories, as well increased transmission bandwidth (fiber optics) and mass storage devices (CD ROM), have magnified this problem. By making digital imaging practical for the first time, these developments have dramatically increased user demand for digital images and thereby made apparent the limitations of current technologies. For example, current ISDN channels are confined to 64 Kbits/s and even with a capacity of 650 Mbytes, a single CD ROM is capable of storing only about 6 minutes of moving video.

While it is important to concentrate on improving these technologies, it is equally as important to reduce the demands placed on them. *Data compression* provides the only immediate solution to this problem. By removing unnecessary or *redundant* data from an image, storage and bandwidth requirements can be dramatically reduced. Data compression will not only allow application developers to satisfy current user requirements within the confines of todays technology, but will also ensure that future technologies meet or exceed growing user demand.

Unfortunately, traditional data compression techniques such as *Huffman* [Huff52] and *Lempel-Ziv-Welch* (LZW) [Welc84] coding can compress digital images by less than 50%. These techniques generally result in low compression ratios because they are exact or *lossless*. Lossless compression techniques are intended primarily for encoding critical data such as text and executable files. Data of this nature must be reconstructed exactly from the compressed format. In contrast, images are signals into which small amounts of noise can be injected without noticeably corrupting the data. Images can therefore be compressed using inexact or *lossy* compression techniques. Currently, these techniques can result in compression ratios between 10:1 and 30:1 for still images without introducing unacceptable levels of distortion into the reconstruction.

In lossy compression schemes, distortion is measured in one of two ways; *objectively* and *subjectively*. Objective measurements are quantitative measures of distortion based on some mathematical function such as *signal-to-noise ratio* (SNR). Subjective distortion measures are more difficult to define universally but are equally or even more important. These measures are based on the subjective opinions of human observers. Lossy data compression techniques attempt to maximize compression ratios while minimizing distortion, both objective and subjective.

There are two primary classes of lossy compression techniques; *vector quantization* and *transform coding*. Vector quantization techniques segment an image into blocks called vectors. A table of typical vectors is maintained from which a good match to each image vector is located. Rather than transmitting the vector in its entirety, the vector quantizer transmits or stores only the appropriate table indices. Reconstruction is achieved by a simple table lookup procedure. Transform coding techniques use a mathematical function to transform an image or portion thereof into an equivalent but more implicit representation. Only a subset of significant components in this representation are retained. An approximate reconstruction of the original image is generated by performing an inverse transformation on this subset. The best known and most effective examples of transform coding for images are those based on the *discrete cosine transform* (DCT) [ChSF77] although *discrete wavelet transforms* (DWTs) [Mall89] are emerging as viable competitors to these techniques. The discrete cosine transform produces a representation of an image in the frequency domain, similar to that of the more familiar *discrete Fourier transform* (DFT). The individual frequency coefficients in this representation are then quantized and coded based upon their relative visual significance.

Although both vector quantization and transform coding have produced very good results for image compression, the search continues for new techniques capable of producing higher compression ratios with lower distortion rates. Signal compression using *fractal geometry* or simply *fractals* represents an emerging area of lossy data compression methods. Fractal compression schemes have attracted a great deal of attention since the introduction of *iterated function systems* (IFSs) [Barn88], a remarkably compact scheme for representing intricate self-similar structures. With IFSs, complex binary images can be represented in as few as 10 bytes [Kins91]. These very compact representations have encouraged a number of researchers to investigate the possibility of applying fractals to

lossy signal compression.

Current fractal compression techniques are based on a corollary of *contractive transformation theory* called the *collage theorem* and are commonly referred to as *collage coding*. The collage theorem implies that if an image or portion thereof can be completely described in terms of smaller possibly distorted versions of itself, then the original image can be reconstructed from this description using a simple iterative procedure. The objective of a collage coding must therefore be to represent an image as a function of itself as accurately and compactly as possible.

In this respect, collage coding differs fundamentally from both vector quantization and transform coding. Vector quantization represents an image in terms of prototype vectors stored in a table while transform coding represents the image by transforming it into some other domain. In contrast, collage coding techniques represent an image in terms of a mathematical function which transforms the image into itself. It is this function which constitute the coded version of the image.

The primary difficulty associated with collage coding techniques is computational complexity. The inverse problem of locating the IFS code which describes a given image is an *NP complete* problem for which an adequate automated solution has not yet been found [PeJS92]. Despite this, other researchers have applied collage coding to image compression using less compact representations with promising results. Successful implementations of these techniques are no longer NP or NP complete but instead operate in known polynomial time. One such technique has been proposed by Jacquin for grey scale images [Jacq92] and is referred to as *fractal block coding* (FBC). Unfortunately, the generalized form of the FBC encoding procedure is still computationally intensive and

consists of an $O(n^4)$ search for $n \times n$ images. A number of authors have proposed methods for reducing FBC encoding times but these approaches have generally been heuristic in nature.

This thesis develops a reduced search FBC encoding procedure using a neural network paradigm known as *frequency sensitive competitive learning* (FSCL) [AKCM90]. This new procedure avoids the uncertainties of heuristic techniques in favor of systematically reducing the order of the encoding algorithm. The current implementation performs a hierarchical rather than a complete search within the image and is based on sub-image classification using neural networks. For an appropriately chosen network, the hierarchical approach reduces the time complexity of the encoding procedure from $O(n^4)$ to $O(n^3)$.

The reduced search FBC encoding and decoding procedures are implemented in the context of a concatenated image compression scheme employing both FBC and *arithmetic entropy coding* [WiNC87]. This scheme compresses an image using FBC and then removes any redundancy remaining in the resulting fractal code with arithmetic coding. In contrast to FBC, arithmetic coding is a lossless compression technique which implies that the fractal code can be reconstructed exactly from its arithmetically compressed representation. Using this concatenated FBC/arithmetic compression scheme, compression ratios can be improved by as much as 20% over FBC alone.

This thesis is organized into nine chapters. Chapter 2 provides a very general introduction to fractals, discusses some of their properties, and indicates why fractal geometry may be applicable to data compression. Chapter 3 develops the contractive

transformation and collage theorems upon which all current and practical fractal compression schemes are based. Barnsley's iterated function systems are described in chapter 4 as the first example of a representation scheme which satisfies the preconditions of the contractive transformation theorem. Even in the absence of an automated encoding procedure, IFSs are historically significant and highly illustrative of collage coding techniques. Chapter 5 discusses a generalized form of the basic fractal block coding procedure developed by Jacquin. The chapter concludes with a review of extensions to this generalized procedure appearing in previous literature. In Chapter 6, the systematic reduced search fractal block coding procedure is developed. This development includes a discussion of frequency sensitive competitive learning neural networks. Chapter 7 describes the implementation of the concatenated fractal/arithmetic coding system for grey scale images. A complete description of arithmetic coding and motivation for its inclusion are also provided. Finally in Chapter 8, FBC compression results are presented and compared against other popular image compression schemes based on vector quantization and transform coding. Conclusions and recommendations are presented in Chapter 9.

CHAPTER II

FRACTALS, NATURE, AND SELF-SIMILARITY

The formal definition of a fractal is "a set for which the *Hausdorff-Besicovitch dimension* strictly exceeds the topological dimension" [Mand83]. Unfortunately, this definition is only meaningful to a select group of mathematicians and even then it does not convey any indication of how or why fractals may be applicable to data compression. A true understanding of fractals and fractal data compression requires a more intuitive perspective than that offered by the formal definition. Consequently, the best way to discuss fractals is by considering a few examples – both classical (fractals which exist only in the minds of imaginative mathematicians) and natural (fractals which seem to materialize in every corner of the physical world). These examples will yield some instinctive understanding of fractal geometry, and reveal a few of the properties which make fractals particularly applicable to data compression.

2.1 What are Fractals?

Consider the straight segment line of length L in Fig. 2.1. With a reasonably accurate ruler, the length of this segment is not difficult to measure. One would simply hold the ruler up to the page, and read off the appropriate result. Now consider the circle of Fig. 2.2c. Knowing what mathematicians have known since the time of Archimedes (about 260 B.C.) you could measure the diameter of the circle and say that its circumference is *pi* (π) times its diameter D. However, if you were not aware of

Archimedes equation for the circumference of a circle you might do exactly as Archimedes did. That is, measure or calculate the circumference using a piecewise linear approximation of the circle. This can be accomplished by inscribing a polygon inside the circle and then calculating the total length L_C of its perimeter as shown in Fig 2.2a. Of course the total length determined in this way is only an approximation for the circumference of the circle and depends upon the particular polygon chosen. A polygon with more and shorter sides, as in Fig. 2.2b, results in an approximation which is not only more accurate but, longer. By increasing the number of sides on the polygon, thus improving the accuracy of this approximation, you would find that the approximate circumference of the circle increases towards a limit. This limit is π times the diameter – Archimedes equation for the circumference of a circle.

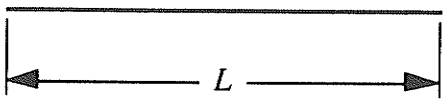


Fig 2.1 A straight line segment of length L.

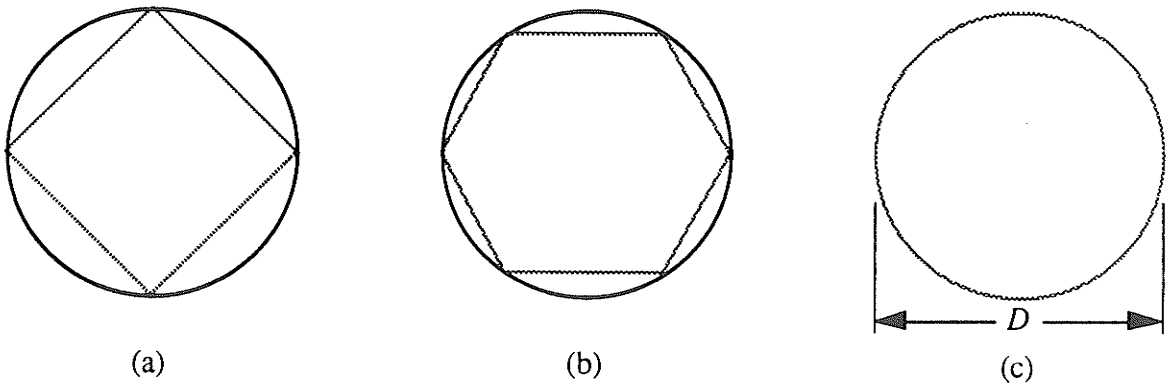


Fig. 2.2. Polygon approximations of the circumference of a circle. Approximated with (a) four line segments for a total length $L_C = 2\sqrt{2}D$, and (b) six line segments for a total length of $L_C = 3D$. (c) The limit as the number of line segments is taken to infinity yields a true circumference of πD.

Now consider the object shown in Fig. 2.3. This object is an approximate representation of what is referred to as the *Koch curve* after the Swedish mathematician von Koch, who first described it. The Koch curve is a non-differentiable function which, like the circle, does not have any straight line components. For this reason, any measurement of its length would have to be a piecewise linear approximation. Following the same logic that resulted in the equation for the circumference of a circle one might suspect that an equation for the length of the Koch curve could be found by taking the limit of a piecewise linear approximation as the pieces are made smaller and smaller. The gross error in this supposition becomes apparent when one considers the particular method by which the Koch curve is constructed.

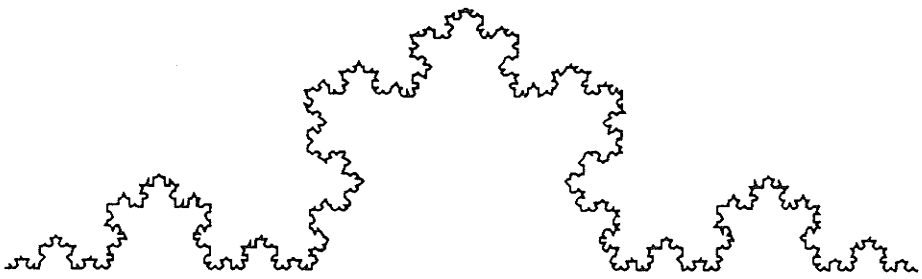


Fig. 2.3. An approximate rendering of the Koch curve.

One technique for constructing the Koch curve is a simple recursive procedure beginning with a straight line segment as shown in Fig. 2.4a. This line is partitioned into three equal parts and the central portion replaced by an equilateral triangle (Fig. 2.4b.). This step is then repeated for each of the remaining segments in the new figure. Each iteration of the construction procedure adds more and more detail to the image but the curve never intersects itself. Repeating this process indefinitely results in the Koch curve.

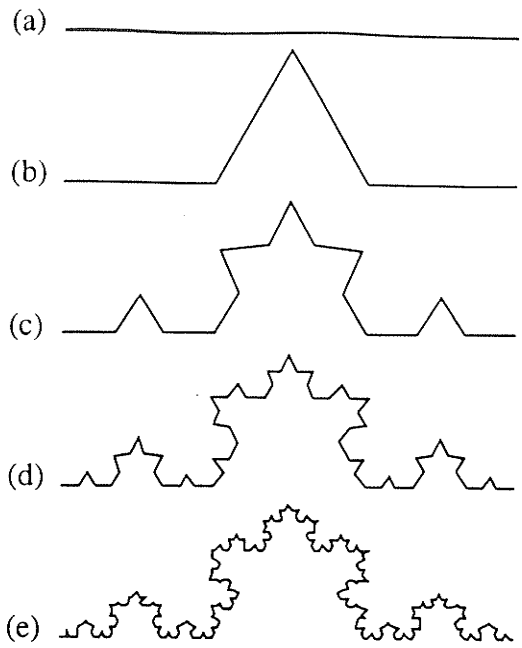


Fig. 2.4. First four iterations of the Koch curve construction procedure.

Now reconsider the length of the Koch curve assuming that the original line segment was of length L. The first iteration of the construction procedure produces a piecewise continuous curve made up of four line segments of equal length. Each line segment is exactly $\frac{1}{3}$ the length of the original line so that the new curve is $\frac{4}{3}$ the length of the original. The next iteration of the procedure replaces each line segment by another piecewise continuous curve of exactly $\frac{4}{3}$ times its length. The length of the curve therefore increases by a factor of $\frac{4}{3}$ at each iteration. After N iterations the length of the curve is given by

$$L_N = \left(\frac{4}{3}\right)^N L \tag{2.1}$$

Since the Koch curve results from an infinite number of iterations of the construction procedure, the length of the Koch curve L_K is given by the limit of Eq. 2.1 as N goes to infinity or

$$L_K = \lim_{N \to \infty} \left(\frac{4}{3}\right)^N L \tag{2.2}$$

But Eq. 2.1 has no limit. As N goes to infinity, so to does the length of the curve. This implies that the length of the Koch curve must be infinite or more appropriately undefined.

Without a finite value for the length of the Koch curve, the following problem arises. Von Koch has described an non-intersecting curve of infinite length which is contained within a bounded two dimensional space. Although confined to a finite area, the Koch curve is still just a line and it is of course impossible to talk about the area of a line. At the same time, as Eq. 2.2 illustrates, we can not describe the Koch curve in a strictly one dimensional sense because its length is undefined. It follows that if we wish to the discuss the nature of Koch curve we must do so in the context of a space whose dimension is somewhere in the interval between one and two. However traditional mathematics, which was drawn primarily from natural observation, only recognized dimensions which are of integer values. Objects like the Koch curve which seemed to be of some form of *fractional dimension* severely upset traditional mathematicians of the 19th and early 20th centuries who called them *pathological curves*, *mathematical monsters*, and *space-filling curves* [Kins72]. Mandelbrot calls them *fractals* [Mand83] and in many ways they have revolutionized the way we view mathematics and its relation to the world around us.

2.2 Fractals in Nature

In proposing objects like the Koch curve with their fractional dimensions, 'surrealist' mathematicians such as Cantor, Peano, Hilbert, Sierpinski, Julia and Hausdorff, set out to illustrate that pure mathematics was capable of describing a 'gallery of

monsters' far beyond anything possible in the 'real world'. These objects led to the redefinition of many traditional concepts in mathematics, like dimension, which had been formerly derived from natural observation. Many mathematicians believed that these expanded definitions would eliminate the limitations previously imposed on traditional mathematics by its natural origins. It was Mandelbrot who pointed out that far from being the exception, in nature fractals appear to be the norm.

In order to illustrate the existence of fractal objects in nature Mandelbrot proposed the apparently simple question – 'How long is the coast of Britain?' [Mand67]. In an attempt to answer this question consider the coastline illustrated in Fig. 2.5. This coastline contains a multitude of 'irregular' bays, inlets, and peninsulas. The only way to resolve the length of the coastline is to measure it and since it is so 'irregular' the only way of doing so is once again by a piecewise linear approximation. Even if it were possible to obtain a very accurate approximation for the length of the curve in Fig. 2.5, consider what would happen if we acquired a map of the same coastline but at a larger scale. On a larger map, previously imperceptible detail would become apparent – a single bay or peninsula might consist of many subbays or subpeninsulas in turn constructed from other features visible only at successively larger scales. In fact, regardless of the scale at which the coastline is measured, there is always another scale just beyond perception containing a plethora of detail equal to the present. These details would have to be accounted for and in doing so the approximate measure for the length of the coastline would increase without bound. Figure 2.6 shows imperical data representing the length of the west coast of Britain measured at different scales which substantiates this claim. Here the total length of the coastline L is measured in small steps of length s. The total length resulting from this measurement then is plotted on a log scale against the inverse of the steps size. As the length of the coastline is measured using smaller and smaller step sizes the total length

measured increases without bound. Consequently a coastline, like the Koch curve, is a fractal object for which length is an elusive and undefinable notion and there is no single universal answer to Mandelbrot's question.

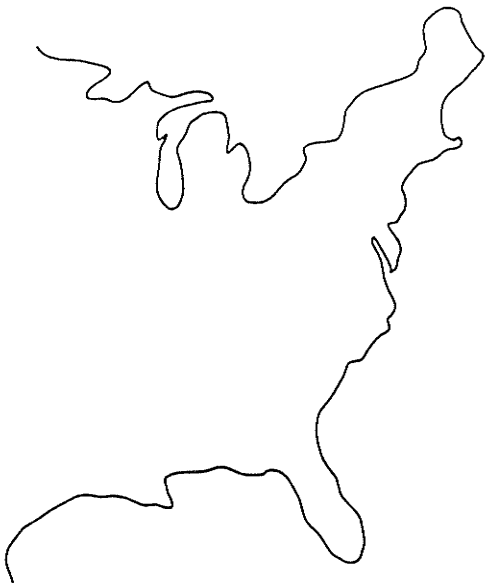


Fig. 2.5. Eastern seaboard of the United States from the Great Lakes to the Gulf of Mexico.

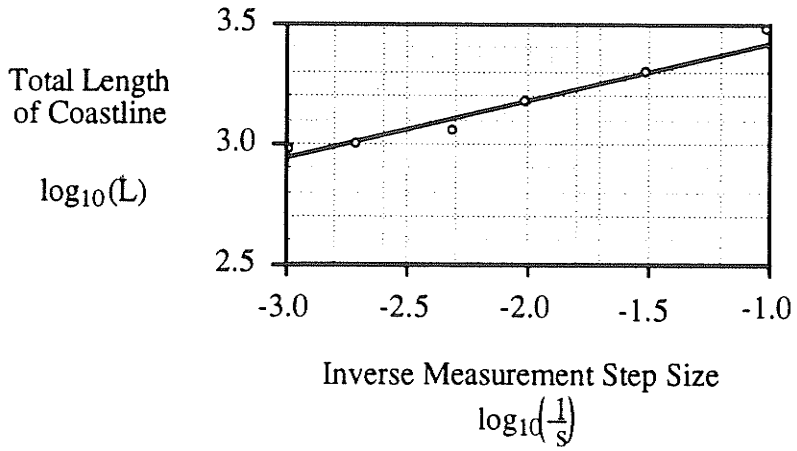


Fig 2.6. Log diagram for the total length of the west coast of Britain L versus the inverse of the measurement step size s [Mand67].

An equally interesting observation is the fact that each of the points in Fig. 2.6 falls roughly on a straight line. Mandelbrot calls the quantity

$$D_F = 1 + d \tag{2.3}$$

where d is the slope of this line, the *fractal dimension* of the coastline. The fractal dimension is a form of the aforementioned Hausdorff-Besicovitch dimension and for the western coastline of Britain it is approximately 1.28.

The idea of a coastline as a fractal object can be easily extended to objects in higher dimensional spaces. Like the length of a coastline, the surface area of a rugged landscape like the mountain range shown in Fig. 2.7 is also ir-rectifiable. A more artificial but equally interesting fractal object is the photograph in Fig. 2.8. At first glance, this photograph may appear to be two dimensional figure with finite area but if we plot it in a three dimensional space with brightness as the third dimension the result (Fig. 2.9) has many characteristics in common with the mountain range. Whether or not the actual photograph is truly a fractal is debatable. A dithered photograph like the one shown actually consists of a finite number of picture elements (pixels) which when enlarged will appear as round areas of uniform brightness. Nevertheless, the original scene of which the photograph is a projection is most definitely a fractal and as such the possibility of modeling digital images as fractal objects has excited many researchers in image processing.

Coastlines, mountain ranges, and photographs are just a few examples of the many fractal objects found in the every day world. In fact, nature abounds with fractals. Snowflakes, the leaves of a fern, the human vascular system, clouds, and even galaxies all exhibit fractal geometry.

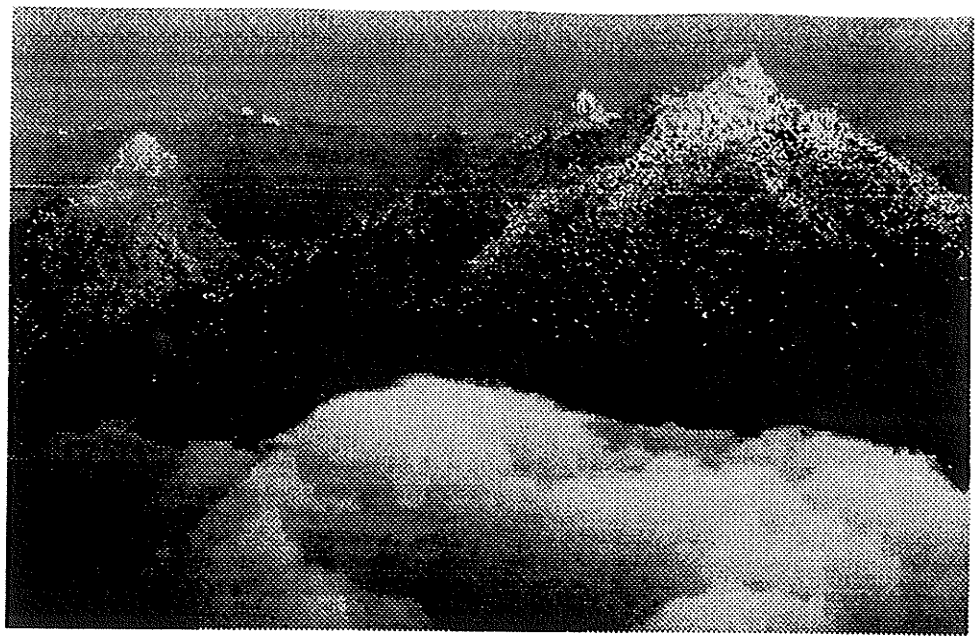


Fig. 2.7. Mountain ranges (and clouds) exhibit fractal geometry.

Fig. 2.8. A black-and-white photograph of Lena.

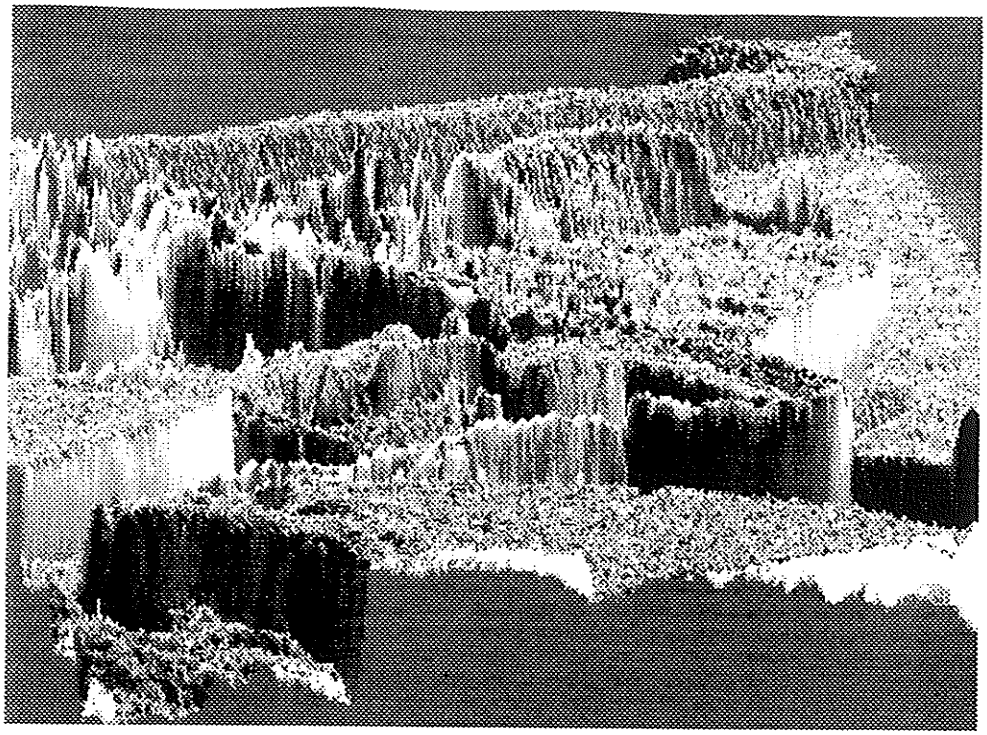


Fig. 2.9. The photograph of Fig. 2.8 rendered in three dimensions with pixel intensity plotted as altitude.

2.3 Fractals and Self-Similarity

Now let us return to the Koch curve in order to discuss another important property of many fractals – *self-similarity*. A close examination of this curve will reveal that, in addition to the recursive procedure described in Section 2.1, this structure can be constructed out of four smaller but otherwise exact copies of itself as shown in Fig. 2.10. Together these four copies form what is referred to as a *collage* of the original image. Objects like the Koch curve which can be constructed from collages of themselves are called *self-similar*. Self-similarity, while not a sufficient condition, is a property common to many fractals.

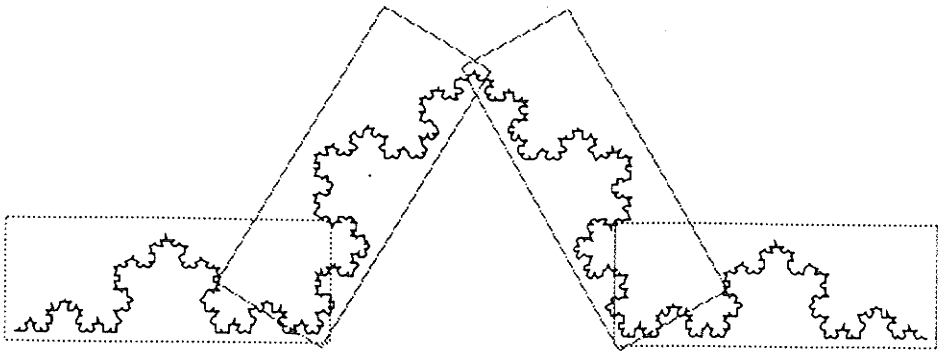


Fig. 2.10. The Koch curve as constructed from four exact copies of itself reduced by a factor of three in all directions [PeJS92].

The self-similar nature of the Koch curve leads to an alternate procedure for its construction called the *multiple reduction copy algorithm* (MRCA). Peitgen *et al.* refer to this a *multiple reduction copy machine* (MRCM) analogous to a photocopy machine with multiple reducing lenses [PeJS92]. This new procedure begins with the same straight line segment of Fig. 2.4a. This line segment is reduced by a factor of three in all directions and four copies of it are placed in the positions indicated by the dotted boxes of Fig. 2.10 representing the self-similar portions of the Koch curve. The resulting image is identicle to that of Fig 2.4b which depicts the first iteration of the previous generation algorithm. This new image is also reduced and copied and the procedure, if repeated, produces exactly the same sequence of images illustrated in Fig. 2.4. An infinite number of iterations yields the Koch curve.

The MRCA may at first seem like just another method of generating the Koch curve but it has a fundamental and remarkable difference. Instead of starting with a straight line as in Fig 2.4, consider the circle of Fig 2.11a. If we perform the MRCA with the circle as the starting image something unexpected happens – the MRCA still converges to the Koch

curve. In fact, no matter what image we begin with the result will always be a Koch curve. The final image is not a function of the starting image but rather of the way in which we map the original Koch curve into itself. Furthermore, since this mapping was derived from the particular collage associated with the Koch curve, this collage must be unique to that object.

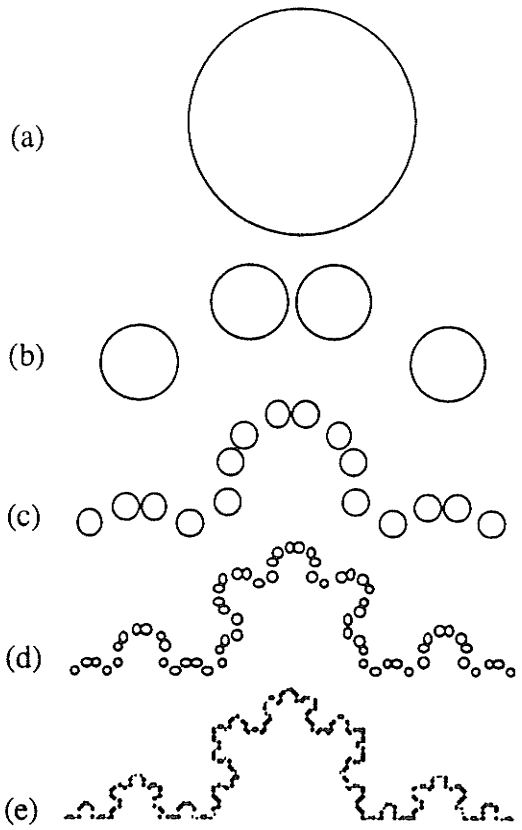


Fig. 2.11. The first four iterations of the *multiple reduction copy algorithm* (MRCA) for the construction of a Koch curve from a circle.

This is a very important and useful result. The Koch curve appears to be a very complex object which, using conventional geometry, might be very difficult to describe. However, using the MRCA it can be described completely with just a handful of

parameters derived from its collage. These parameters include the reduction factor, the number of self-similar portions in the collage, and the x and y coordinates of each portion.

Most natural fractals, however, are not strictly self-similar. Consider the coastline of Fig. 2.5 again. We indicated that zooming in on any portion of the coastline, say a peninsula, would reveal more and more detail. Subpeninsulas and sub-subpeninsulas would become visible. These peninsulas would not be exact copies of the original coastline but they would resemble it in many ways. Objects like the coastline, while not strictly self-similar, are called *self-affine*. Many objects in nature possess self-affinity. For example, each branch of a tree can be thought of as a smaller but inexact copy of the entire tree structure.

The MRCA applies to self-affine as well as self-similar objects. If an object can be represented in terms of smaller distorted copies of itself, then it can be reconstructed from an arbitrary starting image using the MRCA as long as an appropriate set of functions can be identified to perform these distortions. Using these functions, the original self-affine image can be reconstructed by making properly distorted copies of an arbitrary image, reducing these copies, and positioning them correctly to construct a new image. Repeating this process indefinitely results in the MRCA for self-affine images. Of course, the MRCA for strictly self-similar images is just a special case of the MRCA for constructing self-affine images where no distortion of the original object is performed.

The MRCA does not actually require that a self-affine image be represented exactly in terms of itself. Barnsley has developed a theorem which predicts the performance of the of the MRCA for images described only approximately in terms of smaller distorted versions of themselves. Barnsley theorem, called the *collage theorem* [Barn88], implies

that if an image can be approximated using smaller distorted versions of itself, then the original image can be reconstructed approximately using the MRCA. This is a result of considerable importance since it may be very difficult to locate a set of functions which generate an exact collage of the original self-affine structure.

In addition to the collage theorem, Barnsley has developed a set of simple functions which, combined with the MRCA, can compactly represent many intricate self-affine structures like the Koch curve. This representation scheme, called *iterated function systems* (IFSs), is the subject of Chapter 4. However, in order to fully understand IFSs and more importantly the implications and preconditions of the collage theorem, a number of mathematical principles are required. These principles, while furnishing a precise mathematical statement of the collage theorem, will also provide more insight into self-affinity and the workings of the MRCA.

CHAPTER III

MATHEMATICAL FOUNDATIONS: CONTRACTIVE TRANSFORMATION THEORY

Barnsley's collage theorem, briefly discussed in Chapter 2, is actually a corollary of a more general theorem in *metric topology* [NaSe82] called *contractive transformation theory*. The contractive transformation theorem, describes the behavior of infinite sequences like those resulting from repeated application of the MRCA. It was this description that led Barnsley to formulate the collage theorem which provides the mathematical foundation for a group of data compression techniques known collectively as *collage coding*. Contractive transformation theory is itself built upon a number of fundamental principles in metric topology including *metric spaces*, *convergence*, and *contractions*. These principles must of course be developed before a complete understanding of the theorem and its implications is possible.

3.1 Metrics and Metric Spaces

Any discussion of contraction mapping theory must begin with *distortion measures*. A distortion measure is a real valued function $d(\mathbf{x},\mathbf{y})$ which measures the difference or *distance* between two vectors $\mathbf{x}$ and $\mathbf{y}$ in a set X. Mathematically, vectors are simply elements of a set. A vector also consists of discrete elements but unlike a set, the elements of a vector are ordered. It is this order which makes vectors useful as representations of physical objects or phenomena. These objects may be as simple as a

point on the Cartesian plane or as complex as a color photograph. In either case, a distortion measure is a function which places a quantitative value on the difference between two objects of the same type.

Two qualify as a distortion measure a function must simply return a single real valued result for any pair of vectors in the same set. However, a more interesting set of functions are distortion measures known as *metrics* which also satisfy the following four axioms:

(M1) $d(\mathbf{x},\mathbf{y}) \geq 0$ and $d(\mathbf{x},\mathbf{x}) = 0$ for all $\mathbf{x}$ and $\mathbf{y}$ in X.
(M2) If $d(\mathbf{x},\mathbf{y}) = 0$ then $\mathbf{x} = \mathbf{y}$ for all $\mathbf{x}$ and $\mathbf{y}$ in X.
(M3) $d(\mathbf{x},\mathbf{y}) = d(\mathbf{y},\mathbf{x})$ for all $\mathbf{x}$ and $\mathbf{y}$ in X.
(M4) $d(\mathbf{x},\mathbf{y}) \leq d(\mathbf{x},\mathbf{z}) + d(\mathbf{z},\mathbf{y})$ for all $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ in X.

Together, the metric d and the set X are referred to as the *metric space (X,d)*. Some familiar examples of metric spaces include

(1) the set of all real numbers R were

$$d(\mathbf{x},\mathbf{y}) = |x - y|, \tag{3.1}$$

(2) and the Cartesian plane denoted R^2 with the *Euclidian metric* given by

$$d_2(\mathbf{x},\mathbf{y}) = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2}. \tag{3.2}$$

where the notation x_i and y_i refers to the ith elements of the vectors $\mathbf{x}$ and $\mathbf{y}$ respectively.

It should be noted that two different metrics, defined on the same set, form two entirely different metric spaces. For example the function

$$d(\mathbf{x},\mathbf{y}) = |x_1-y_1| + |x_2-y_2| \tag{3.3}$$

is valid metric on the Cartesian plane but results in a metric space which is very different from (R^2,d_2). Which metric is better? This question can only be answered in the context of a particular application. The pilot of an airplane flying in a straight line from point $\mathbf{x}$ to point $\mathbf{y}$ might use the Euclidian metric to measure the distance he or she must travel. However, the metric described by Eq. 3.3 would be far more relevant to a taxi cab driver in Manhattan who must drive from $\mathbf{x}$ to $\mathbf{y}$ along the perpendicular lattice of roadways which make up that cities infrastructure.

Having chosen an appropriate metric space, it becomes possible to talk about *convergent sequences* in that space. Instead of a single vector $\mathbf{x}$ in the set X, consider an infinite sequence of vectors $\{\mathbf{x}_n\} = \{\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3,\ldots\}$ in X. This sequence is said to be *convergent* if there is a point $\mathbf{x}_o$ in (X,d) with the property that for any real number $\varepsilon>0$ there is an integer N such that $d(\mathbf{x}_n,\mathbf{x}_o)<\varepsilon$ for all $n\geq N$. The point $\mathbf{x}_o$ is referred to as the *limit* of the sequence $\{\mathbf{x}_n\}$ and is often written

$$\lim_{n\to\infty} \mathbf{x}_n = \mathbf{x}_o \tag{3.4}$$

This simply means is that successive vectors in the sequence $\{\mathbf{x}_n\}$ become closer and closer to some point $\mathbf{x}_o$.

A metric space is said to be *complete* if any *Cauchy sequence* $\{\mathbf{x}_n\}$ in (X,d) is also a convergent sequence. This means that, if for any $\varepsilon>0$, there exists a positive integer N such that $d(\mathbf{x}_n,\mathbf{x}_m)\leq\varepsilon$ for any $n,m\geq N$, then the sequence $\{\mathbf{x}_n\}$ has a limit $\mathbf{x}_o$ in (X,d). A Cauchy sequence is simply a sequence for which the distance between successive pairs of elements becomes arbitrarily small. In a complete metric space, all such sequences converge to a defined limit.

3.2 Contractions, Fixed Points, and the Collage Theorem

With an understanding of metrics and sequences, it now becomes possible to introduce contractions and the contractive transformation theorem as well as its corollary, the collage theorem.

A *contraction* is a function f, defined on a metric space (X,d) which maps X into itself. Furthermore, to qualify as a contraction there must exists a real number k called the *Lipschitz coefficient* or *contraction factor*, where $0 \leq k < 1$, such that

$$d(f(\mathbf{x}),f(\mathbf{y})) \leq kd(\mathbf{x},\mathbf{y}) \tag{3.5}$$

for all $\mathbf{x}$ and $\mathbf{y}$ in X.

The importance of contractions to fractal coding is described by *Banach's fixed point* or *contractive transformation theorem*. Formally, this theorem states that for a contraction f, defined on a complete metric space (X,d), there is one and only one point $\mathbf{x}_o$ in X such that

$$f(\mathbf{x}_o) = \mathbf{x}_o. \tag{3.6}$$

Moreover, if $\mathbf{x}$ is any point in X and $\{\mathbf{x}_n\}$ is a sequence defined by $\mathbf{x}_1 = f(\mathbf{x})$, $\mathbf{x}_2 = f(\mathbf{x}_1)$, ... $\mathbf{x}_n = f(\mathbf{x}_{n-1}) = f^n(\mathbf{x})$, then $\{\mathbf{x}_n\}$ is convergent and

$$\lim_{n\to\infty} \mathbf{x}_n = \lim_{n\to\infty} f^n(\mathbf{x}) = \mathbf{x}_o. \tag{3.7}$$

More simply put, every contraction in a complete metric space has associated with it a unique fixed point to which successive iterations of the contraction on an arbitrary starting

point will always converge. This fixed point is often called the *attractor* of the contraction.

In addition to a unique attractor, there exists an estimate which relates the distance between the n^{th} iteration of f in the sequence $\{\mathbf{x}_n\}$ and the attractor to the distance between the starting point $\mathbf{x}$ and the first point in the sequence $\mathbf{x}_1$. This estimate, called the *a priori estimate*, is given by

$$d(\mathbf{x}_n,\mathbf{x}_o) \leq \frac{k^n}{1-k}\, d(\mathbf{x},\mathbf{x}_1) \tag{3.8}$$

Letting n equal zero and substituting $\mathbf{x}_1 = f(\mathbf{x})$ into Eq. 3.8 yields

$$d(\mathbf{x},\mathbf{x}_o) \leq \frac{1}{1-k}\, d(\mathbf{x},f(\mathbf{x})) \tag{3.9}$$

which is the generalized form of Barnsley's collage theorem.

What is so interesting and relevant about the contraction mapping theorem is that it describes exactly the type of behavior exhibited by the MRCA. Regardless of initial image $\mathbf{x}$, the MRCA always converges to the same final image $\mathbf{x}_o$. This occurs because the functions which map self-affine objects like the Koch curve into themselves are contractions.

The contractive transformation theorem and its corollary, the collage theorem, have important implications in image coding which should be understood outside of the context of a particular metric space or contractive function. The contractive transformation theorem implies that if an image $\mathbf{x}$ can be described approximately by a contractive function $f(\mathbf{x})$ of itself, successive iterations of the contraction on any initial image will result in an image $\mathbf{x}_o$, unique to f. Furthermore, the collage theorem establishes an upper bound (Eq 3.9) on the error between the original image and the attractor $\mathbf{x}_o$ associated with f. This bound relates

the error between the original image and the attractor to the error between the original image and its collage, $f(\mathbf{x})$.

Encoding/decoding schemes based on contractive transformation theory and the collage theorem are appropriately called *collage coding* or coding by *iterative contractive transformations*. The objective of these coding techniques is to locate a contractive function or set of contractive functions which can be used to generate a collage of a given image. In the techniques described in this thesis, a basic form for these functions is assumed. The encoder then locates the specific parameters within this form that minimize the distance between the original image and the resulting collage. These parameters represent the fractally encoded description of the image and are either transmitted or stored for reconstruction at a latter time.

For a successful collage coding technique both an appropriate metric space and general form for the contraction must be established. As Section 3.1 indicated, it is important that the metric associated with the selected metric space yield relevant measures of the distance between images. In selecting a form for the function one must also ensure that

(1) with appropriately chosen parameters the resulting functions will indeed be a contraction,

(2) functions based on this form will produce adequate collages of the original image with respect to the chosen distance measure,

(3) the parameters which govern the behavior of these functions can be represented more compactly than the original image, and

(4) for a particular image, these parameters can be located efficiently and

systematically.

If the basic form meets these criterion then the original image, or an approximation thereof, can be decoded by iterating the resulting functions with the appropriate parameters on any starting image as per the MRCA.

Both of the fractal coding techniques presented in the remainder of this thesis are examples of collage coding and satisfy at least the first three of the above requirements. *Iterated function systems* (IFSs) are a remarkably compact fractal coding technique used primarily for representing complex self-similar binary images. Unfortunately, it may be tremendously difficult if not impossible to extract the IFS parameters for a particular source image via an automated procedure. Nevertheless, IFSs are highly illustrative of collage coding techniques in general and are therefore worth examining before proceeding to *fractal block coding* (FBC) – a less compact but more manageable fractal compression technique for gray scale images.

CHAPTER IV

ITERATED FUNCTION SYSTEMS

The first difficulty associated with the development of any data compression scheme based on collage coding is locating a set of functions which satisfy the requirements of the contraction mapping theorem. In conjunction with the collage theorem, Barnsley introduced *iterated function systems* (IFSs), a scheme for representing intricate self-similar structures. An IFS consists of a set of simple contractive functions which describe an object in terms of smaller distorted versions of itself. This description is remarkably compact – complex binary images can be represented in as few as 10 bytes.

Unfortunately, the inverse problem of locating the IFS code which describes a given image is an *NP complete* problem for which an adequate automated solution has yet to be found. Despite this, the importance of IFSs can not be overemphasized since they have inspired other collage coding techniques based on more manageable representation schemes. In addition to their historical significance, IFSs are extremely useful for illustrating general collage coding principles and give some indication of the very high compression ratios (eg., 10,000:1 [BaSl88]) which may be possible using fractal compression techniques.

This chapter describes the basic IFS representation scheme and shows how it satisfies the requirements of the contraction mapping theorem. In addition, an alternative form of the MRCA called the *random iteration algorithm* (RIA) is introduced. The RIA,

also developed by Barnsley, has a number of advantages over the traditional MRCA and is useful for establishing the connection between *fractals* and the affiliated field of *chaos* [PeJS92]. Finally image compression based on IFSs is discussed briefly with particular attention paid to the difficulty of designing an automated encoding procedure.

4.1 Overview of Iterated Function Systems

Iterated function systems (IFSs) are a scheme for representing complex self-affine structures. An IFS consists of a set of *contractive affine transformations* (CATs) of the form

$$w(\mathbf{x}) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \tag{4.1}$$

which are used to transform a set of points contained within in the Cartesian plane. In this equation, the parameters e and f represent translations along the horizontal and vertical axes while the matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ is linear operator which can distort an image in four different ways. These distortions, illustrated in Fig. 4.1., include *scaling, rotation, reflection,* and *shearing*. Figure 4.2 illustrates the combined effect of translation and distortion by a single CAT on an object H.

Given a distorted and translated version of some object, the coefficients of the CAT which generate this distortion and translation can be determined by solving a system of simple linear equations. Three points $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ on the original structure are selected and then the three corresponding points $\tilde{\mathbf{x}}$, $\tilde{\mathbf{y}}$, and $\tilde{\mathbf{z}}$ on the distorted version are located. Substituting these points into Eq. 4.1 yields

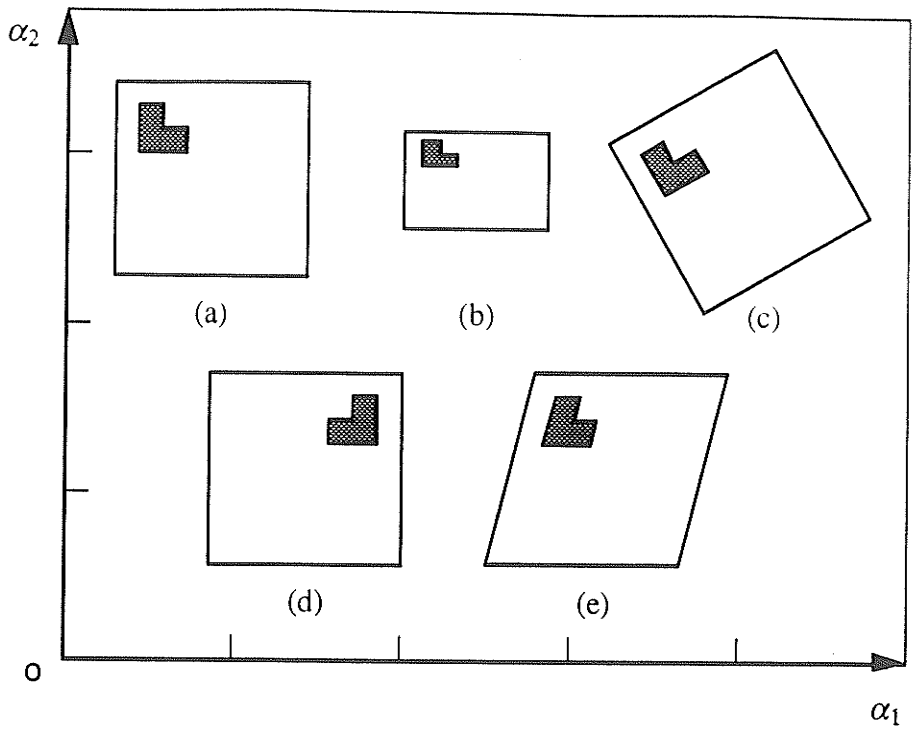


Fig. 4.1. Distortions producible using *contractive affine transformations* CATs. (a) original image, (b) scaling, (c) rotation, (d) reflection, and (e) shearing [PeJS92].

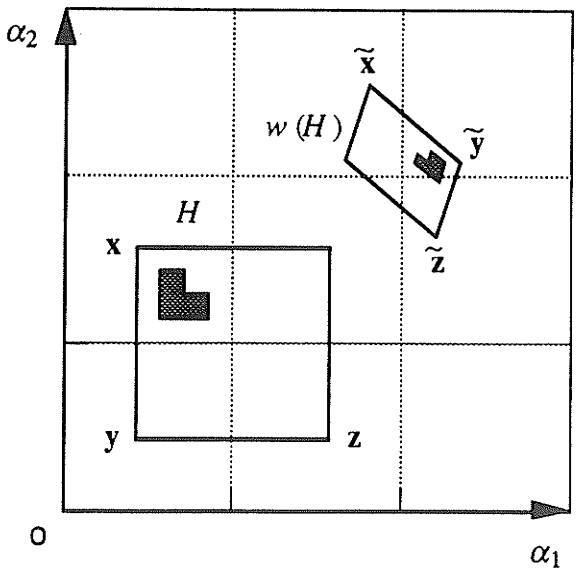


Fig. 4.2. Scaling, rotation, reflection, skewing, and translation of an object H by a CAT.

$$(\tilde{x}_1, \tilde{x}_2) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \tag{4.2}$$

$$(\tilde{y}_1, \tilde{y}_2) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}. \tag{4.3}$$

and

$$(\tilde{z}_1, \tilde{z}_2) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \tag{4.4}$$

Equations (4.2) through (4.4) can be rewritten in terms of the following two systems of linear equations

$$\begin{bmatrix} x_1 & x_2 & 1 \\ y_1 & y_2 & 1 \\ z_1 & z_2 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ e \end{bmatrix} = \begin{bmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \tilde{z}_1 \end{bmatrix} \tag{4.5}$$

and

$$\begin{bmatrix} x_1 & x_2 & 1 \\ y_1 & y_2 & 1 \\ z_1 & z_2 & 1 \end{bmatrix} \begin{bmatrix} c \\ d \\ f \end{bmatrix} = \begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix}. \tag{4.6}$$

Solving Eqs. 4.5 and 4.6 yields

$$\begin{bmatrix} a \\ b \\ e \end{bmatrix} = T \begin{bmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \tilde{z}_1 \end{bmatrix} \tag{4.7}$$

and

$$\begin{bmatrix} c \\ d \\ f \end{bmatrix} = T \begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} \tag{4.8}$$

where

$$T = \frac{1}{t}\begin{bmatrix} y_2-z_2 & z_2-x_2 & x_2-y_2 \\ z_1-y_1 & x_1-z_1 & y_1-x_1 \\ y_1z_2-y_2z_1 & x_2z_1-x_1z_2 & x_1y_2-x_2y_1 \end{bmatrix} \tag{4.9}$$

and

$$t = x_1y_2-x_1z_2-x_2y_1+y_1z_2+x_2z_1-y_2z_1 \tag{4.10}$$

Any self-affine image which can be described in terms of itself under any combination of the distortions illustrated in Fig. 4.1 can be constructed using IFSs. Each self-affine portion in the original images has associated with it a unique CAT. For example, the Koch curve K illustrated in Fig. 4.3 has four self-affine portions and is therefore constructed from an IFS consisting of four CATs. The CAT coefficients for each of the self-affine portions of the curve can be extracted by selecting three points on the original image, locating the corresponding points on each self-affine portion, and solving Eqs. 4.7 through 4.10.

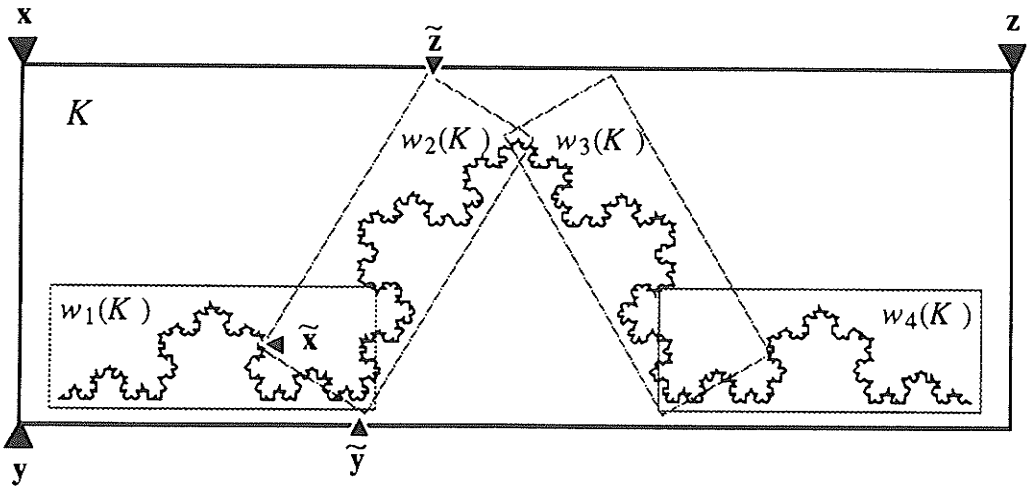
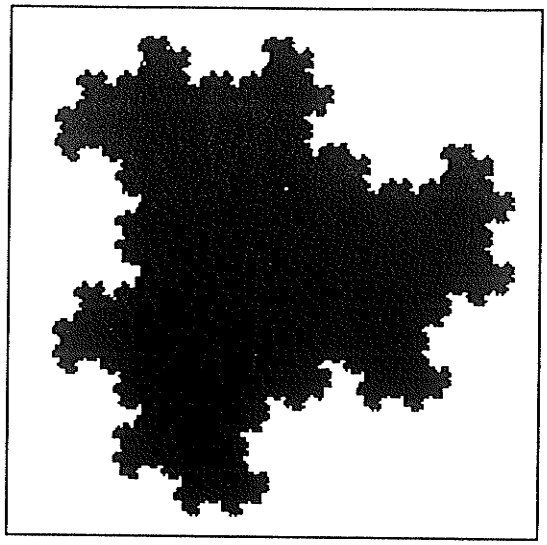


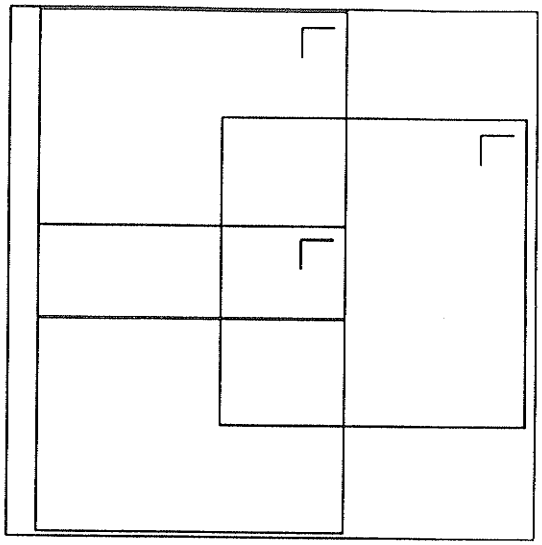
Fig. 4.3. The Koch curve K as constructed from four CATs of itself, w_1 through w_4.

Using the IFS coefficients, the original self-affine structure can be constructed from an arbitrary starting image using the MRCA. Each CAT in the IFS is in turn applied to the starting image to create a new image consisting of N copies of the original. This new image is then transformed in the same way, and this process, if repeated indefinitely will result in a reconstruction of the coded self-affine structure independent of the starting image.

A wide variety of fractal images can be constructed using IFSs. A small subset of these images are 'classical' fractals like the Koch curve but more interesting examples include surprisingly realistic looking renditions of natural objects such as leaves, ferns, and trees. Examples of both types of images are shown in Figs. 4.4 through 4.7 along side their self-affine constructions and IFS coefficients. Each of these images is 200 by 200 pixels in size and would therefore require almost 5 Kbytes of storage ~~each.~~ The IFS storage requirements for these image are significantly less and are included with the corresponding figure. These values assume that only seven bits are required to adequately represent each IFS coefficient. For objects like the fern in Fig. 4.6 which require only four CATS this represents a compression ratios of 238:1.



(a)

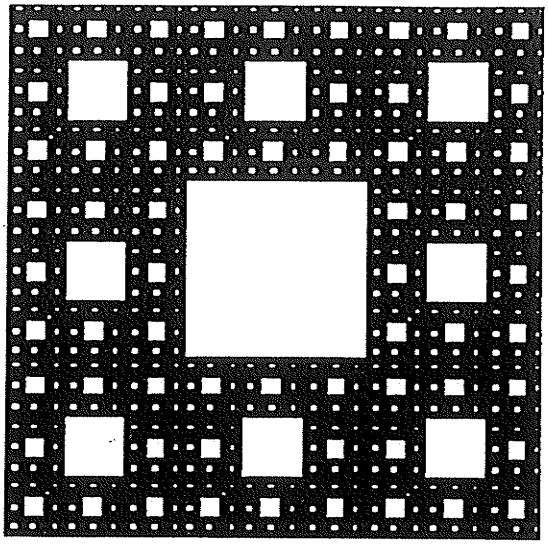


(b)

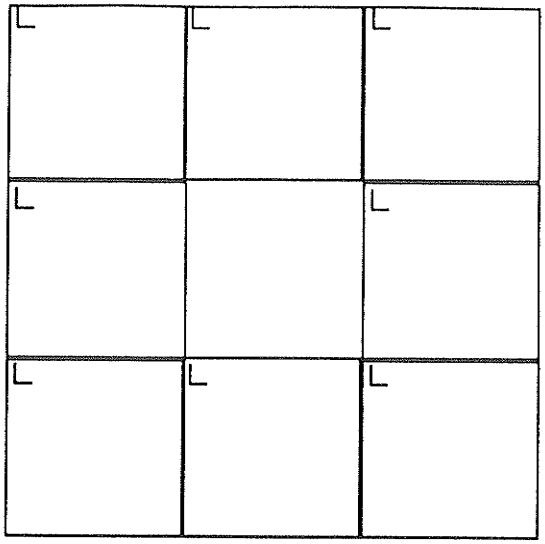
w_i	a	b	c	d	e	f
1	0.00	0.58	-0.58	0.00	0.05	0.59
2	0.00	0.58	-0.58	0.00	0.40	0.79
3	0.00	0.58	-0.58	0.00	0.05	0.98

(c)

Fig. 4.4. (a) A Dragon curve, (b) its self-affine components, and (c) its IFS description; 3 CATs, 16 bytes.



(a)

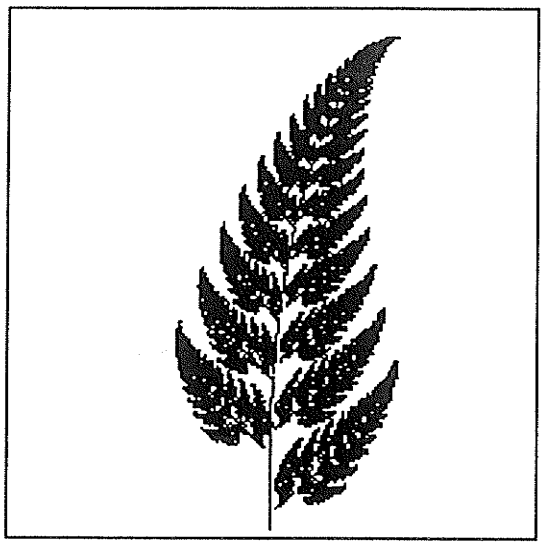


(b)

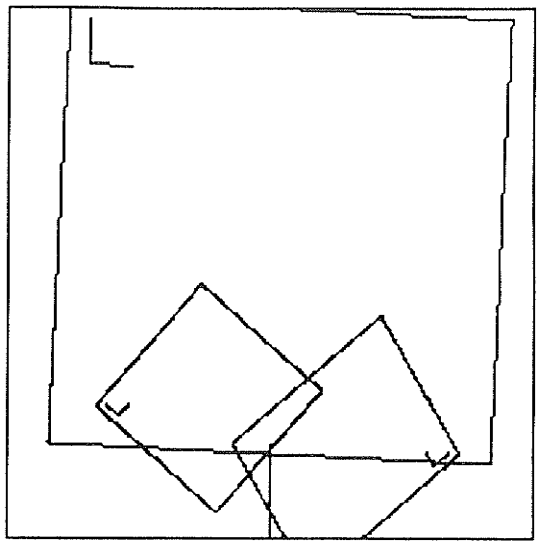
w_i	a	b	c	d	e	f
1	0.34	0.00	0.00	0.34	0.00	0.00
2	0.34	0.00	0.00	0.34	0.00	0.33
3	0.34	0.00	0.00	0.34	0.00	0.67
4	0.34	0.00	0.00	0.34	0.33	0.00
5	0.34	0.00	0.00	0.34	0.33	0.67
6	0.34	0.00	0.00	0.34	0.67	0.00
7	0.34	0.00	0.00	0.34	0.67	0.33
8	0.34	0.00	0.00	0.34	0.67	0.67

(c)

Fig. 4.5. (a) Sierpinski's carpet, (b) its self-affine components, and (c) its IFS description; 8 CATs, 42 bytes.



(a)

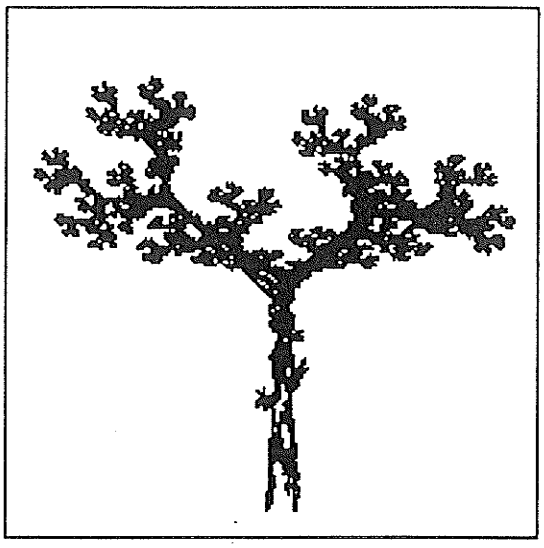


(b)

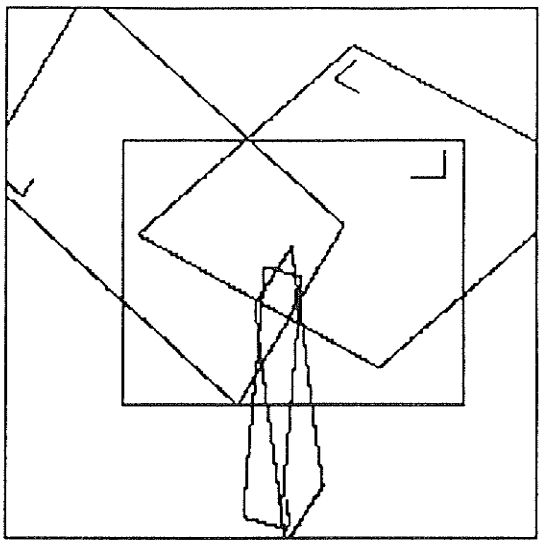
w_i	a	b	c	d	e	f
1	0.85	0.04	-0.04	0.85	0.08	0.18
2	0.20	-0.23	0.23	0.20	0.40	0.05
3	-0.15	0.28	0.26	0.24	0.48	-0.08
4	0.00	0.00	0.00	0.16	0.50	0.00

(c)

Fig. 4.6. (a) Barnsley's fern, (b) its self-affine components, and (c) its IFS description; 4 CATs, 21 bytes.



(a)



(b)

w_i	a	b	c	d	e	f
1	0.20	0.04	-0.04	0.85	0.08	0.18
2	0.46	0.41	-0.25	0.36	0.25	0.57
3	-0.06	-0.07	0.45	0.11	0.60	0.10
4	-0.04	0.70	-0.47	-0.02	0.49	0.51
5	-0.63	0.00	0.00	0.50	0.86	0.25

(c)

Fig. 4.7. (a) A fractal tree, (b) its self-affine components, and (c) its IFS description; 5 CATs, 27 bytes.

4.2 Mathematical Foundations: The Hausdorff Distance

Iterated function systems result in unique attractors when combined with the MRCA because they satisfy the preconditions of Banach's contractive transformation theorem. In particular, an IFS is a contraction with respect to the *Hausdorff distance* defined on the complete metric space formed by sets of ordered pairs in the Cartesian plane. The Hausdorff distance is a metric which measures the distance between two sets of points X and Y, each representing a binary digital image. In order to discuss the distance between sets of points we must first be able to talk about the distance between a point $\mathbf{x}$ and a set Y given by

$$d(\mathbf{x},Y) = \min\{d_2(\mathbf{x},\mathbf{y}) \mid \mathbf{y}\in Y\} \tag{4.11}$$

where d_2 is the familiar *Euclidian metric* for order pairs and is given by

$$d_2(\mathbf{x},\mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \quad . \tag{4.12}$$

The distance between two sets X and Y is in turn given by

$$d(X,Y) = \max\{d(\mathbf{x},Y) \mid \mathbf{x}\in X\} \tag{4.13}$$

The Hausdorff distance follows from this and is simply

$$h(X,Y) = \max\{d(X,Y),\ d(Y,X)\} \tag{4.14}$$

The Hausdorff metric takes into account the relative positions of pixels when determining the distance between two images. For each pixel in an image X, the Hausdorff metric locates the nearest (in the Euclidian sense) pixel in the second image Y. The maximum distance measured between corresponding pairs of nearest pixels is retained

as a measure of the distortion between the image X and the image Y. This procedure is then repeated for every pixel in the the image Y resulting in a measure of the distortion between the image Y and the image X. The Hausdorff distance is the maximum of these two measures.

Under the Hausdorff metric, the contraction factor associated with an IFS can be calculated from the contraction associated with each CAT [Hutc81]. The contraction factor for each CAT is the maximum of the scaling factors in each of two directions given by

$$s_1 = \sqrt{a^2 + c^2} \tag{4.15}$$

and

$$s_2 = \sqrt{b^2 + d^2} \tag{4.16}$$

respectively, so that

$$k_i = \max\{s_1, s_2\} \tag{4.17}$$

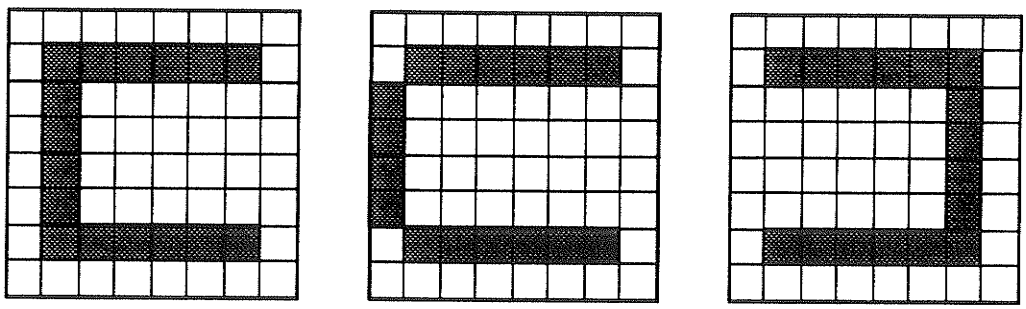
The contraction factor for the entire IFS is simply the maximum contraction factor of the individual contraction factors associated with the N CATs in the IFS or

$$k = \max\{k_i \mid i=1,2,\cdots,N\} \tag{4.18}$$

The motivation for selecting a complex metric like the Hausdorff distance is based, in part, on the psychology of visual perception which determines whether or not a particular metric is in fact meaningful. The two binary images X and Y can also be considered as two nxm vectors $\mathbf{X}$ and $\mathbf{Y}$ for which a simpler metric such as the nxm dimensional Euclidian metric given by

$$d_2(\mathbf{X},\mathbf{Y}) = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}(X_{ij} - Y_{ij})^2} \tag{4.19}$$

could be used. However, Fig. 4.8 illustrates how this measure is not as visually meaningful as the Hausdorff metric. The distance between Figs. 4.8a and 4.8b and the distance between Figs. 4.8a and 4.8c are identical with respect to Eq. 4.19. However, to both the human visual system and the Hausdorff metric, Fig. 4.8a resembles Fig. 4.8b more than Fig. 4.8c. The Hausdorff metric provides distance measures more in line with the human visual system because it takes into account the relative position of pixels in the image.



(a) (b) (c)

Fig. 4.8. Ambiguity of the Euclidian metric. Under the euclidian metric the distance between (a) and (b) is equal to the distance between (a) and (c). Under the Hausdorff metric the distance between (a) and (b) is significantly less than the distance between (a) and (c). The Hausdorff metric reflects the way in which a human being might compare these objects and is therefore a more meaningful distortion measure for binary images [Jean90].

Before concluding this section, a remark on notation is in order. In Chapter 3, images were represented as vectors in a metric space and were therefore denoted by lower case boldface text (eg. **x**). Sometimes, as was the case with the Hausdorff metric, it is

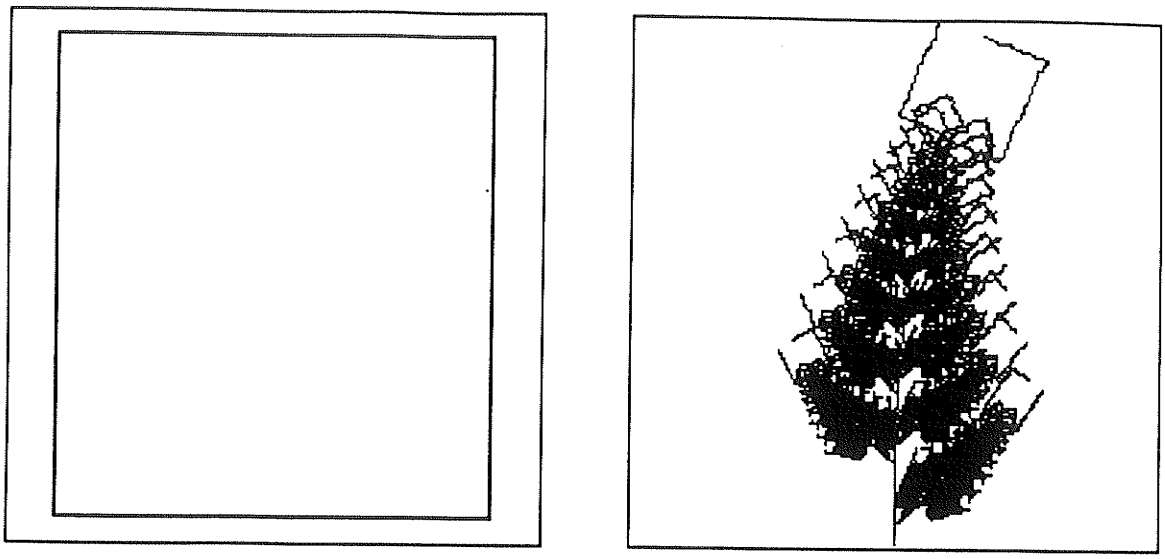
more convenient to think of these images as sets of points denoted X. In future chapters, images will be described as vectors which can be further subdivided into smaller component vectors. To avoid confusion, in the remainder of this thesis images will always be referred to using capital letters consistent with the notation adopted in this chapter. If the image should be viewed in the context of a set of points then it will be represented by capital italics (eg. X). Conversely, if the image is better represented using vector notation then it will be referred to using capital boldface lettering (eg. $\mathbf{X}$).

4.3 The Random Iteration Algorithm

Barnsley's third contribution to fractal data compression, besides the collage theorem and iterated function systems, is an alternative form of the MRCA known as the *chaos game* or *random iteration algorithm* (RIA). The primary disadvantage of the MRCA is that it may take many iterations to converge to an acceptable representation of the attractor associated with a particular IFS. As an example consider Barnsley's fern constructed at a scale of 200x200 using the MRCA. At each iteration of the MRCA, the first CAT in the IFS reduces the current image by only 85%. If the starting image is a 200x200 pixel box as shown in Fig. 4.9a then even after 10 iterations as shown in Fig. 4.9b, artifacts caused by the particular starting image are still visible. In fact, these artifacts will only disappear at the point where the original square has been mapped into a single pixel. This occurs when

$$200 \cdot 0.85^{M} = 1 \tag{4.20}$$

where M is the number of iterations of the MRCA. Solving Eq. 4.20 yields M≈33.



(a) (b)

Fig. 4.9. Barnsley's fern as constructed using the MRCA. (a) The starting image and (b) the reconstructed fern after 10 iterations of the MRCA.

The random iteration algorithm shown in Fig. 4.10, is a less computationally intensive procedure equivalent to the MRCA. The RIA begins with a random point, usually $\mathbf{x} = (0,0)$, then selects a CAT at random from the IFS according to a discrete probability density function. The probabilities p_i reflect the relative area occupied by each CAT in the self-affine representation of the image and are given by

$$p_i = \frac{a_i d_i - b_i c_i}{\sum_{j=1}^{N} |a_j d_j - b_j c_j|} \tag{4.21}$$

Once selected, the CAT is applied to the current point to generate a new point which is then plotted. The procedure is repeated until enough points have been plotted to adequately represent the attractor of the IFS. In most implementations of the RIA, the points resulting from the first 10 iterations of the algorithm are usually not plotted since the initial point is

chosen arbitrarily and may not lie within the attractor of the IFS. However, because the IFS is a contraction the RIA converges very quickly and after 10 iterations the current point seems always to lie within the attractor. An implementation of the RIA with a good starting point and automatic scaling is described by Kinsner [Kins91]. C language source code for both the RIA and the MRCA is also provided in Appendix A of this thesis.

STEP 1: Initialize the starting point $\mathbf{x} = (0,0)$.

STEP 2: Select a random CAT w_i from the IFS $W = \{w_1, w_2, \ldots, w_N\}$ according to the discrete probability density function p_i.

STEP 3: Let $\mathbf{x} = w_i(\mathbf{x})$.

STEP 4: Plot the point $\mathbf{x}$. Goto STEP 2.

Fig. 4.10. The *random iteration algorithm* (RIA) for IFS decoding.

The first 100, 1000, 10 000, and 100 000 iterations of the RIA for the construction of Barnsley's fern are shown in Fig. 4.11. Note that although in excess of 10000 iterations of the RIA were needed to adequately reconstruct the fern, each iteration requires that only one point be plotted. In contrast a single iteration of the MRCA for a 200x200 pixel image would require that 40 000 points be considered. This would amount to over a million points for the 33 iterations of the MRCA required to reconstruct the fern.

Having illustrated that the RIA requires significantly less computation than the MRCA, it is necessary to indicate how the two algorithms are otherwise equivalent. To do so we must show that like the MRCA, the RIA will fill out the attractor of the IFS. A

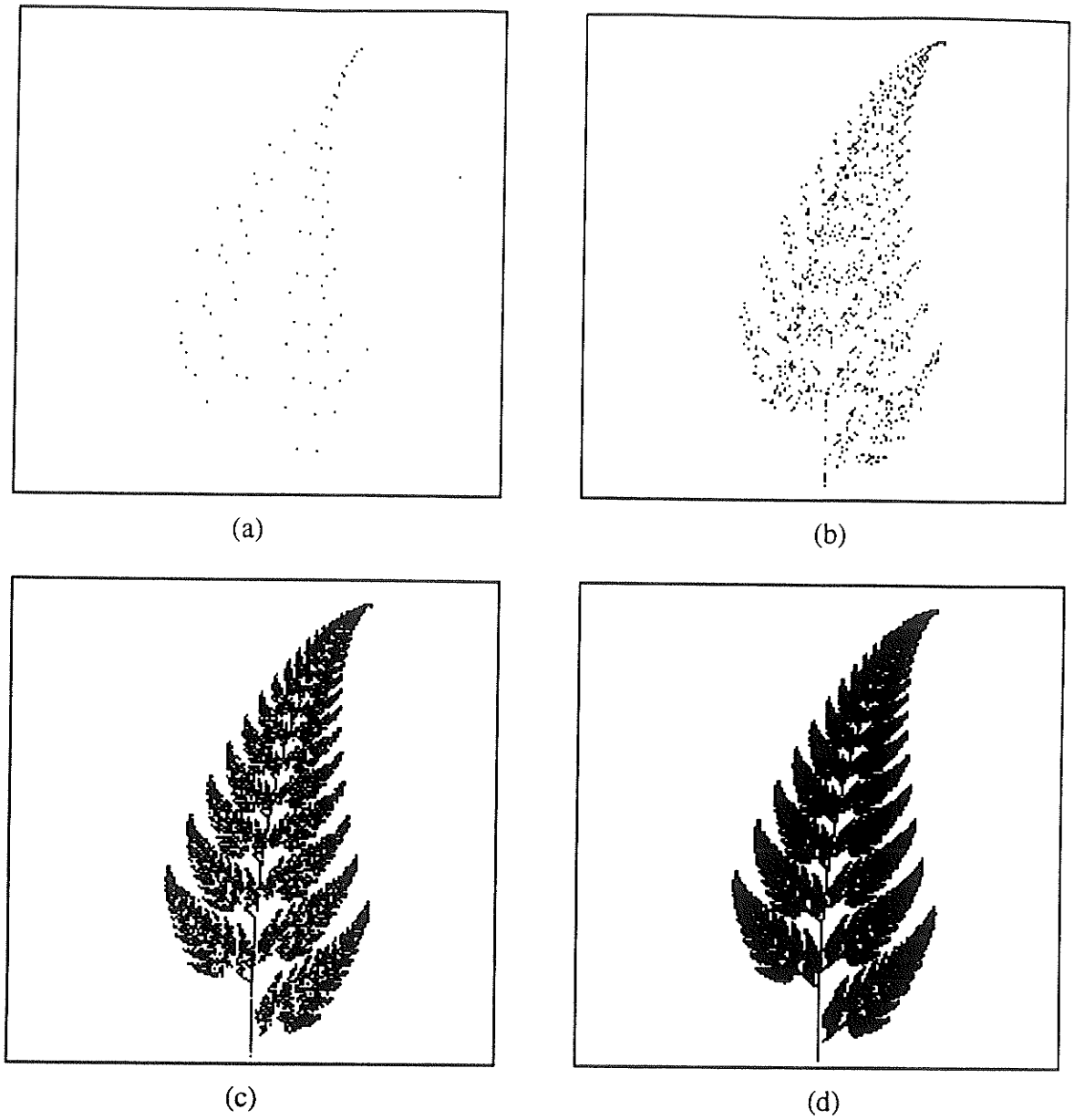


Fig. 4.11. The first (a) 100, (b) 1000, (c) 10 000, and (d) 100 000 iterations of the RIA for the construction of Barnsley's fern. In each case the first 10 points are not plotted.

formal proof of this is based on *ergodic theory* [Barn88] however a simple probabilistic discussion of this is based on the fact that each CAT is associated with a particular self-

affine portion of the original image. By selecting CATs at random, each self-affine portion will be visited a number of times relative to the probability of its associated CAT being chosen. Peitgen *et al.* provide an intuitive discussion of the operation of the RIA based on such a probabilistic approach and in fact suggest that this discussion may be more useful than any formal mathematical proof based on ergodicity [PeJS92].

4.4 Data Compression with IFSs

IFSs can be applied to the compression of binary, grey scale, and color images as well as other lossy signals but are most naturally discussed in the context of binary images. An IFS compression system for binary images would begin with an image or portion thereof and generate a collage of that image using a set of CATs. The encoder must then manipulate the CAT coefficients so as to minimize the Hausdorff distance between the original image and its collage. The collage theorem implies that if the collage is close to the original image in terms of the Hausdorff distance, the attractor of the IFS generated using the MRCA or RIA decoding algorithms will be a good representations of the original image. Of course the original image can be represented exactly by choosing an IFS which has one CAT for each pixel in the original image. However since each CAT increases the storage requirement of the IFS code this will not result in a particularly efficient representation of the original image. Therefore, the IFS encoding algorithm must not only generate a collage which adequately covers the original image but must do so using as few CATs as possible.

Unfortunately, while it is often easy for an intelligent observer to recognize the self-affine portions in an image and thus construct an acceptable collage, an adequate automated solution to this problem has not yet been found. A number of optimization techniques such

as *simulated annealing* [LaAa87] and *genetic algorithms* [Davi91] have been attempted but with only limited success [MoHS90]. The primary disadvantage of such techniques is that they must perform repeated calculations of the Hausdorff distance. Each time a new set of CATs is considered as a possible solution to the encoding problem, a new collage must be generated and the Hausdorff distance calculated to determine the suitability of that particular solution. Although fast algorithms [Shon89] for the calculation of this metric do exist, this is still a computationally intensive procedure. In addition to the computational requirements of the Hausdorff metric, for many images this metric is a non-monotonically decreasing function with many local minima. This non-monotonicity eliminates optimization techniques based on *gradient descent* and may severely impede more global approaches should the Hausdorff metric become particularly erratic. This is often the case for images in which the self-affine components of the collage overlap.

Other researches have attempted to extract the IFS parameters directly from the source image without generating a collage. Such attempts include the use of *morphological skeletal transforms* [MaSh90] and *wavelet transforms* [FrDu90] from which the IFS coefficients may be determined directly. These methods seem to work well for some images and not so well for others since the relationship between images in one domain and the IFS paramters in another has not been clearly established.

In all, no single algorithm has emerged which is capable of solving the IFS encoding problem quickly and for all images. It remains to be seen whether or not the role of the intelligent observer in recognizing self-affinity can in fact be automated. Despite this, since the introduction of IFSs, literally dozens of papers have appeared with titles such as "Image compression using the fractal transform" [Barn90]. More often than not, these papers describe little more than the MRCA and RIA algorithms for fractal image

decoding not encoding. These papers bring to mind a joke amongst information theorists about

> "a fellow who developed a data compression algorithm which could reduce any file of any size down to a single bit. Unfortunately, he's still trying to work a few bugs out of the decompression algorithm."

Nevertheless, IFSs are an efficient way of representing complex self-affine structures and despite the lack of an adequate automated compression algorithm, a scheme does exist for manually extracting the IFS parameters from a certain class of images. Even though IFS encoding is an *NP complete* problem for which a fast general purpose solution may not even exist, other researchers have applied the collage theorem and the concept of self-similarity to image compression using less compact but more manageable representation schemes. One such technique, known as *fractal block coding*, is the subject of the remaining chapters.

CHAPTER V

GENERALIZED FRACTAL BLOCK CODING

Collage coding schemes can be applied to grey scale as well as binary images although the approach is somewhat less intuitive. Jacquin has proposed a fractal encoding technique for self-affine grey-scale images called *fractal block coding* (FBC) [Jacq89][Jacq90a][Jacq90b][Jacq92]. Like iterated function systems, Jacquin's technique attempts to eliminate redundancy by describing an image as a mathematical function of itself. In particular a function is chosen which describes the image at one scale in terms of its self-affine portions at another.

Unlike IFSs, an algorithmic procedure exists for extracting the FBC parameters from a particular image. This procedure is based on a *divide-and-conquer* approach in which both the image and the fractal transformation are segmented into simpler components. Rather than attempting to locate appropriate values for all of the FBC parameters at once, the encoder resolves the parameters associated with each of these segments independently and in succession. Using this approach, an appropriate fractal representation for a source image can be located in polynomial rather than NP or NP complete time.

This chapter outlines the fractal representation scheme for grey scale images developed by Jacquin. Jacquin's fractal transformation and its associated parameters are described as well as generalized forms of the compression and decompression procedures

referred to as *FBC encoding* and *FBC decoding*, respectively. After outlining the FBC representation scheme in general terms, it is discussed in the context of metric spaces and the contractive transformation theorem. This discussion focuses on the particular constraints (or lack thereof) which must be placed on the paramters of the fractal transformation. Finally, a number of extensions to the generalized procedure, concerned primarily with improving its computational efficiency, are described. These extensions, while mostly heuristic in nature, form the basis of a more systematic approach described in Chapter 6.

5.1 Exhaustive Search FBC Encoding Procedure

The generalized form of Jacquin's encoding algorithm begins by subdividing a large image into many smaller square vectors or blocks. Rather than coding the image as a single entity, a method is sought for efficiently representing each of these blocks individually. Jacquin's technique represents each image block in terms of a transformed version of some larger block in the same image as shown in Fig. 5.1. The blocks being coded are referred to as *range blocks* while the larger blocks, from whence the range blocks are represented, are termed *domain blocks*. Domain blocks, being larger than range blocks, represent larger scale features in the image. The set of all possible range blocks is called the *range pool* and consists of all non-overlapping rxr blocks in the image to be coded. The set of all possible domain blocks is likewise referred to as the *domain pool* and consists of all possible dxd blocks in the image – overlapping or otherwise. Jacquin's fractal block coding technique is a collage scheme in so far as the representation of each block in the range pool from some block in the domain pool forms a collage of the original image.

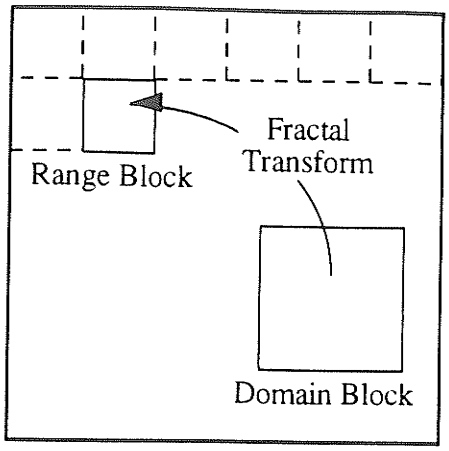


Fig. 5.1. Fractal transformation of domain blocks into range blocks [Jacq92].

The basic FBC coding procedure consists of an exhaustive search of the image for the domain block and set of fractal transformations which most closely represent each range block. The suitability of a particular range/domain block pair is determined according to the *Euclidian metric* denoted by d_2. For a range block $\mathbf{y}$ and a transformed domain block $\tilde{\mathbf{x}}$ each containing rxr pixels, d_2 is given by

$$d_2(\tilde{\mathbf{x}},\mathbf{y}) \doteq \sqrt{\sum_{i=1}^{r}\sum_{j=1}^{r}(\tilde{x}_{ij} - y_{ij})^2} \tag{5.1}$$

The encoder locates the domain block and fractal transformation parameters which minimize d_2 for each range block. Compression results by ensuring that the representation scheme for each range block (in this case the pointer to the domain block and the transformation parameters) is more compact than the explicit description of the block itself.

5.2 Fractal Image Transformation

Since domain blocks are larger than range blocks some form of transformation is

required to map domain blocks into range blocks. In fact, the fractal transformation depicted in Fig. 5.1 consists of three distinct transformations performed sequentially as shown in Fig. 5.2. These transformations are *spatial contraction, isometric block transformation*, and *grey level scaling and translation* respectively.

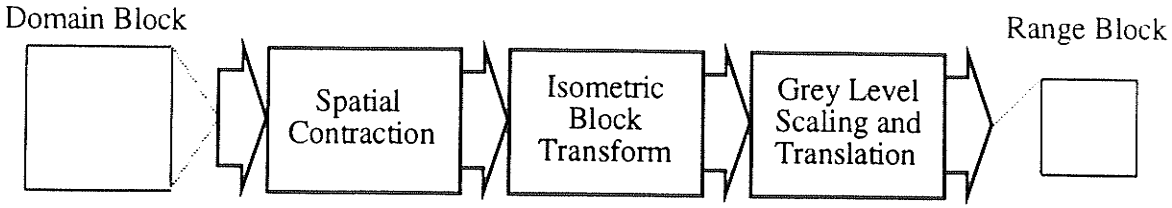


Fig. 5.2. The fractal block transform in terms of its sequential component transforms: spatial contraction, isometric block transformation, and grey level scaling and translation.

Spatial contraction serves to reduce a domain block of size dxd to the size rxr associated with range blocks. If d is an integer multiple of r, this can be accomplished by simply taking the average of every $\left(\frac{d}{r}\right)^2$ adjacent pixels in the domain block.

The isometric block transform redistributes pixels within a contracted domain block in a deterministic manner. This alters the physical orientation of the input block without effecting the individual pixel intensities. In actuality, a set of isometric transforms are maintained from which the most appropriate for each domain/range block pair is chosen. For square domain and range blocks the following eight transforms are used:

(1) identity,
(2) reflection about mid-vertical axis,
(3) reflection about mid-horizontal axis,
(4) reflection about first diagonal,
(5) reflection about second diagonal,
(6) $+90^{o}$ rotation about center,
(7) $+180^{o}$ rotation about center, and
(8) -90^{o} rotation about center.

The effect of each of these transforms is illustrated in Fig. 5.3. Together, the eight isometric block transforms form what is called a *group*. Each transform in a group has an inverse and successive application of two or more transforms results in a single transform already in the group. For this reason combinations of multiple isometries need not be considered.

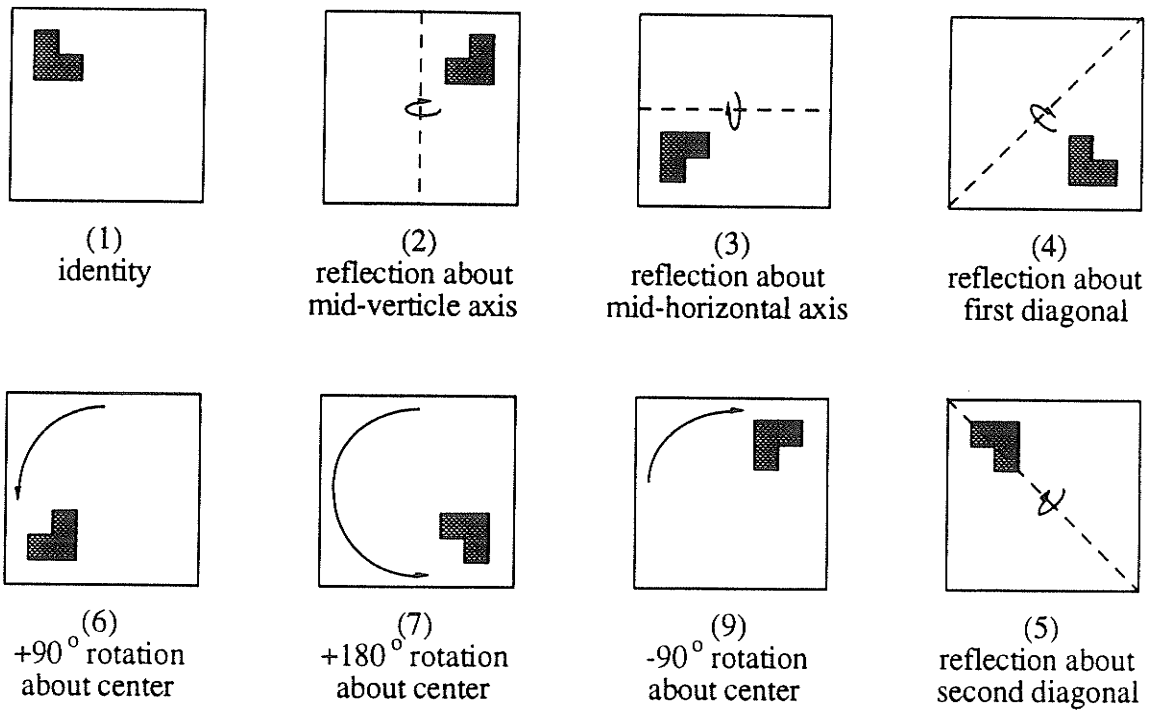


Fig. 5.3. Isometric block transformations.

Finally, grey level scaling and translation modify pixel intensity without effecting the blocks physical orientation. This is accomplished by scaling and translating each block **x** as follows:

$$\tilde{\mathbf{x}} = a\mathbf{x} + t\mathbf{u} \tag{5.2}$$

where a and t are the scaling and translation coefficients, respectively. The vector **u** is such that all components are equal to one; i.e.,

$$u_{ij} = 1 \qquad \text{for all } 1 \le i,j \le r \tag{5.3}$$

Adding a multiple of **u** to the original vector has the effect of altering the original blocks mean intensity.

During the coding procedure, the optimal values of a and t for any given combination of contracted domain blocks and isometric block transforms must be calculated so as to best represent the range block being coded. For a range block **y** and a spatially contracted domain block **x** this is accomplished by minimizing $d_2(\tilde{\mathbf{x}},\mathbf{y})$ with respect to a and t (see Appendix B) for which

$$a = \frac{\|\mathbf{u}\|^2\langle \mathbf{x},\mathbf{y}\rangle - \langle \mathbf{x},\mathbf{u}\rangle\langle \mathbf{y},\mathbf{u}\rangle}{\|\mathbf{u}\|^2\|\mathbf{x}\|^2 - \langle \mathbf{x},\mathbf{u}\rangle^2} \tag{5.4}$$

and

$$t = \frac{\|\mathbf{x}\|^2\langle \mathbf{y},\mathbf{u}\rangle - \langle \mathbf{x},\mathbf{y}\rangle\langle \mathbf{x},\mathbf{u}\rangle}{\|\mathbf{u}\|^2\|\mathbf{x}\|^2 - \langle \mathbf{x},\mathbf{u}\rangle^2}. \tag{5.5}$$

Here the functions $\|\bullet\|$ and $\langle \bullet,\bullet \rangle$ are the *Euclidian norm* and *inner product* and are given by

$$\|\mathbf{x}\| \doteq \sqrt{\sum_{i=1}^{r}\sum_{j=1}^{r} x_{ij}^2} \tag{5.6}$$

and

$$\langle \mathbf{x}, \mathbf{y} \rangle \doteq \sum_{i=1}^{r}\sum_{j=1}^{r} x_{ij}\, y_{ij} \tag{5.7}$$

The Euclidian norm is commonly interpreted as signal *energy* while the inner product is often referred to as the *correlation* of two vectors.

From the preceding description, it follows that each range block is represented by an individual block code containing the following four parameters:

(1) a pointer to the best domain block,
(2) a pointer to the best isometric transform,
(3) an optimal scaling coefficient, and
(4) an optimal translation coefficient.

The set of block codes for all range blocks in the the image is referred to as the *fractal code* for that image.

5.3 Iterative Image Reconstruction: FBC Decoding

To reconstruct the original image from its fractal code, an iterative decoding algorithm is used. This decoding algorithm begins with an arbitrary image referred to as the *domain image*. A second image called the *range image* is then generated by transforming the domain image according to the fractal code. Remember that the fractal code represents each range block in terms of some contractively transformed domain block

in the same image. During coding, the range and domain pools are drawn from the same image. During decoding however, the domain pool is drawn from the domain image while the range pool constitutes the range image and is generated by transforming the appropriate domain blocks from the domain pool as shown in Fig 5.4. This transformation is performed according to the specific parameters for each range block outlined by the fractal code. The algorithm is iterative in that the range image generated in this way, becomes the new domain image and the transformation is repeated. Each successive iteration results in a new range image closer to the source image used during the encoding procedure.

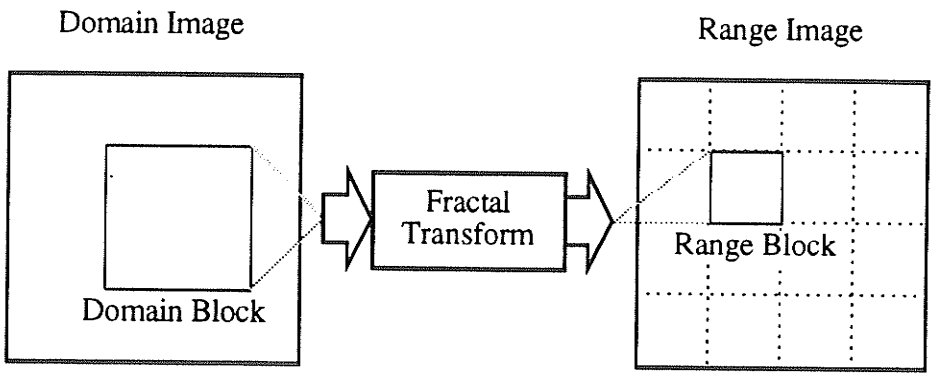


Fig. 5.4. Reconstruction of the range image from a domain image via the fractal code.

The iterative nature of the algorithm is extremely important to the fractal aspect of the coding technique. An important feature associated with scale self similarity is the fact that fractal images possess infinite resolution. Since the spatial reduction portion of the fractal transform maps large domain blocks into smaller range blocks, repeated application of the fractal code will produce images of successively higher resolution (limited of course by the resolution of the computer display). In this way a 256x256 pixel image could be fractal coded and then reconstructed at a resolution of 512x512. Similarly, a small portion of a fractal image could be magnified without exhibiting the blockiness or edge degradation

which results from the magnification of fixed resolution images.

5.4 Mathematical Basis for FBC

In order to establish that the iterative reconstruction procedure will in fact converge to a reasonable representation of the original image, we must discuss the conditions under which fractal block coding satisfies the requirements of the contraction mapping theory. This discussion must of course take place in the context of some complete metric space. In fact we will define two metric spaces valid for grey scale images and use both in the ensuing discussion of the fractal block code as a contractive function.

5.4.1 Metric Spaces for Grey Scale Images

Mathematically, an nxm digital image is simply an element of the set of all possible nxm digital images. Since a digital image can be thought of as a simple two dimensional array of pixels, this set can be described as the set of all real nxm matrices denoted R^{nm}.

To make R^{nm} a metric space requires a metric. Two valid metrics for the set R^{nm} are the *Euclidian metric* and the *sup metric*. For two nxm images **X** and **Y**, the Euclidian metric is given by

$$d_2(\mathbf{X},\mathbf{Y}) \doteq \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}(X_{ij} - Y_{ij})^2} \tag{5.8}$$

which is the same as Eq. 5.1 except defined on the entire image rather than a single rxr block. Similarly on R^{nm} the sup or *Tchebychev* metric, denoted d_∞ is given by

$$d_\infty(\mathbf{X},\mathbf{Y}) \doteq \sup\left\{|X_{ij} - Y_{ij}|: 1\le i\le n,\ 1\le j\le m\right\} \tag{5.9}$$

Combining the set R^{nm} and the metrics d_2 and d_∞ yields the metric spaces (R^{nm},d_2) and (R^{nm},d_∞) which are both complete.

The Euclidian and sup-metrics have very different physical meanings. The Euclidian metric is a measure of the energy in the signal resulting from the difference between the two images **X** and **Y**. In this sense, the Euclidian metric provides a measure of distance which reflects (although not exactly) the average of the errors between corresponding pixels in either image. This means that very large or very small single pixel errors will not appreciably effect the outcome of the Euclidian metric. In contrast, the sup-metric provides an upper bound on the error associated with any single pixel location by implying that no two corresponding pixels in the images **X** and **Y** differ by more than $d_\infty(\mathbf{X},\mathbf{Y})$. The reason for defining two metric spaces for grey scale images becomes apparent when we attempt to discuss the contraction factor of the fractal code. Although the Euclidian metric is visually more meaningful and therefore used in the encoding procedure, it is much easier to establish that a complex function is a contraction in the (R^{nm},d_∞) space.

5.4.2 Contractivity of the Fractal Block Code

Jacquin has provided a formal proof which establishes that the fractal transformations outlined in Section 5.2 are in fact contractive [Jacq89]. The important results of this proof are discussed here in the context of the sup metric.

The spatial contraction function for square range blocks reduces the domain block

by a factor of $\frac{d}{r}$ in both directions. This is accomplished by taking the mean of every $\left(\frac{d}{r}\right)^2$ pixels in the domain block. Under the sup metric, the contraction factor associated with this operation is equal to one.

Since the isometric block transforms redistribute pixels within rxr pixel blocks without effecting the pixels intensities, they all have a contraction factor of one.

The grey level scaling and translation function scales and translates pixel values in the image by the constant factors a and t or

$$\widetilde{\mathbf{x}} = a\mathbf{x} + t\mathbf{u} \tag{5.10}$$

Grey level scaling has the contraction factor

$$k_A = |a| \tag{5.11}$$

while grey level translation has a constant contraction factor of one.

The contraction factor for the fractal block transform representing a single range block $\mathbf{x}_i$ is given by product of the contraction factors associated with each of the individual component functions in the entire transform. Since the contraction factors are one for all but the grey level scaling function this product is given by

$$k_i = k_A = |a_i| \tag{5.12}$$

where a_i represents the scaling coefficient associated with the i^{th} range block.

Under the sup metric, the contraction factor for the entire fractal code is given by the supremum of the contraction factors for each range block or

$$k = \sup_i(k_i) = \sup_i|a_i| \tag{5.13}$$

This means that to ensure that the fractal block code is in fact a contraction and will therefore converge to a fixed image, the absolute value of scaling factor $|a|$ associated with each range block must be strictly less than one or

$$|a_i| < 1 \quad \text{for all } i \tag{5.14}$$

However, the constraint placed on the scaling factor a by Eq. 5.14 is actually much too strong if the complex interaction between overlapping range and domain block is taken into consideration. This interaction may result in a function which is *eventually contractive* even if for certain range blocks the absolute value of the scaling coefficient exceeds one.

A function f is said to be eventually contractive if there exists some integer $q>0$ such that the function f^q is a contraction. The eventually contractive function f has associated with it a the same fixed point as the contractive function f^q. So, even if the fractal code is not in itself contractive, it may be eventually contractive and the collage theorem will still hold. As the selection of domain blocks becomes more uniformly distributed, the interactions between overlapping range and domain blocks makes it more likely that the fractal code will in fact converge. In practice, the fractal codes for non-trivial images such as photographs are all eventually and strongly contractive without any restrictions placed on a or the domain pool [OiLR91].

Finally, it should be pointed out that if a function is a contraction on one metric space then it will yield a fixed point in any metric space defined on the same set since the metric does not in any way effect the behavior of the function. This result is of practical

importance since it is considerably more difficult to determine the contraction factor for the fractal code under the Euclidian metric than the sup-metric.

5.5 Extensions to the Generalized Coding Procedure

The main problem with the generalized coding procedure is the time required to compress images. Since every range block in the image must be compared against every domain block in each of its eight possible isometric configurations, the total number of block comparisons is given by

$$T = 8(n-d)^2\left(\frac{n}{r}\right)^2 \tag{5.15}$$

This is an $O(n^4)$ problem. A number of authors including Jacquin himself have introduced schemes for improving the speed of the coding algorithm. This section outlines a number of methods which have been used in the past to reduce compression time while maintaining acceptable levels of quality and compression ratios.

The actual encoding technique proposed by Jacquin [Jacq89] was much more complicated than the generalized form outlined in Sections 5.1 and 5.2. Jacquin realized that searching the entire image for the ideal domain block for each range block was too time consuming. Instead he proposed a reduced domain pool. This was accomplished by dividing large images (256x256) into 128x128 sub images and encoding these independently. This also had the effect of increasing the compression ratio since a 14 bit pointer is required to address any pixel in a 128x128 image while 256x256 images for example, require 16 bit pointers. Unfortunately, reducing the domain pool decreases image quality since there are fewer domain blocks from which to construct the best fractal block transform.

To further reduce the search time, Jacquin classified each range and domain block according to its *ac* signal energy into one of four categories; shade blocks, simple edges, mixed edges, and midrange blocks [RaGe86]. Domain blocks were only considered as possible sources for a range block if they were classified into the same category. In addition, shade, midrange, and edge blocks were each coded slightly differently. Shade blocks, being the simplest were coded with the fewest number of bits, while edge blocks required the most bits. This led to an overall improvement in compression ratio. Unfortunately, although the four categories chosen had a basis in image analysis, four was not the most appropriate number for an optimal time improvement.

Having severely reduced image quality by reducing the domain pool, Jacquin decided to divide the image up into domain and range blocks of two different sizes. The coding procedure would proceed and attempt to code the image with the largest size of range blocks available. If a particular range block could not be coded acceptably, then it was broken down into four smaller blocks each coded individually. Unfortunately, it took just as many bits to code these smaller blocks as is did the larger ones so the compression, ratio which he had sought to improve by limiting the domain pool and classifying the blocks, was reduced.

In all, Jacquin's attempts to improve quality and compression ratio seemed to cancel each other out although a noticeable improvement in speed was realized. Since $O(n^4)$ is 16 times less for n = 128 than for n = 256 and four 128x128 subimages must be coded, a speed improvement of a factor of four could at most result for 256x256 images.

Oien, Lepsoy, and Ramstaad [OiLR91] have also attempted to improve

compression time by reducing the size of the domain pool. Their version of the fractal block coding technique uses only non-overlapping domain blocks in the image. In a 256x256 image, there are only 256 non-overlapping 16x16 domain blocks compared to 57,600 overlapping blocks. In addition to requiring less search time, this significantly improves the compression ratio. Addressing these non-overlapping range blocks requires only 8 bits rather than the 16 bits required to address every possible overlapping range block in a 256x256 image. Since reducing the domain pool also significantly reduces image quality, two extra components were introduced to the grey level translation function. Instead of shifting the block by a multiple of some vector $\mathbf{u}$, three vectors $\mathbf{u}_1$, $\mathbf{u}_2$, and $\mathbf{u}_3$ were used. As in the generalized form, the vector $\mathbf{u}_1$ consisted of all ones. The vector $\mathbf{u}_2$ was a slope in the i direction or

$$[\mathbf{u}_2]_{ij} = i \tag{5.16}$$

while the vector $\mathbf{u}_3$ consisted of a slope in the j direction or

$$[\mathbf{u}_3]_{ij} = j. \tag{5.17}$$

where the notation $[\mathbf{u}_k]_{ij}$ refers to the ij^{th} element of the k^{th} vector. The expanded grey level translation function resulted in an improvement in image quality at the expense of compression ratio since two new coefficients, associated with $\mathbf{u}_2$ and $\mathbf{u}_3$, where required. However, the bits saved by reducing the domain pool were applied to representing the two new translation coefficients $\mathbf{u}_2$ and $\mathbf{u}_3$ so the overall compression ratio remained unchanged. First and foremost, this improved translation function combined with the large reduction in the domain pool resulted in a dramatic decrease in coding time. In their paper, Oien, Lepsoy, and Ramstaad do however show a small improvement in image quality over the original work of Jacquin and their technique is much simpler since only one block size

is required.

Finally, Beaumont [Beau91] has proposed an improved distortion measure for calculating the error between range and domain blocks. Rather than using the straight Euclidian metric he first transforms each block using the Hadamard transform. He then exploits the varying sensitivity of the visual cortex to frequency by establishing error thresholds for each of the Hadamard coefficients according to their relative importance to the human visual system. This has two effects. Firstly, a domain block can be eliminated as soon as the error threshold for a single Hadamard coefficient is exceeded which will speed up the compression scheme. Secondly, although in terms of *signal-to-noise ratio* (SNR) these images may be of lower quality, in subjective visual tests they will be of superior quality.

5.6 Summary

This chapter has described the basic fractal block coding technique proposed by Jacquin. The fractal representation scheme as well as generalized encoding and decoding procedures were discussed. To ensure that the decoding procedure would in fact converge to an adequate representation of the original image it was necessary to confine the scaling coefficient associated with each range block to values strictly less than one. However, previous experimental results indicate that for non-trivial images, the complex interaction between overlapping range and domain blocks causes the fractal code to become eventually and strongly contractive even when these coefficients are left unconstrained.

It cannot be over emphasized that the success of this technique, as opposed to IFSs, is based on the fact that a divide-and-conquer encoding procedure can be employed to

locate the FBC parameters associated with a particular image. This procedure

(1) partitions the image into smaller range blocks and encodes each of these blocks individually, and

(2) divides the fractal transform into three sub-transformations: spatial contraction, isometric block transformation, and grey level scaling and translation. For each range block the paramters associated with these transformations are then solved consecutively.

For nxn images the generalized form of the encoding procedure is an $O(n^4)$ problem to which a number of heuristic time reducing approaches have been applied. These approaches focus primarily on reducing search time by limiting the size of the domain pool. Unfortunately, a reduced domain pool usually results in reconstructed images of much lower quality and must be compensated for by introducing additional quality improving techniques into the encoding procedure. While all of the extensions to the generalized FBC encoding procedure discussed in this chapter do in fact reduce compression time, they do not address the fundamental issue of computational complexity. That is, even with these extensions, the encoding procedure remains $O(n^4)$. A more systematic approach for improving compression times without adversely effecting image quality is presented in the next chapter. This approach uses neural networks to assist the encoder in its search for appropriate domain/range block pairs and is aimed at actually decreasing the computational complexity of the encoding procedure.

CHAPTER VI

REDUCED SEARCH FRACTAL BLOCK CODING WITH NEURAL NETWORKS

The primary disadvantage of fractal block coding is the time required to search the domain pool for the best domain blocks to represent each range block in the image. Reducing the domain pool is of course the easiest way to tackle this problem but it results in image degradation. Of all the speed improving techniques proposed thus far, the only one that does not dramatically decrease image quality is Jacquin's approach of classifying domain and range blocks and comparing only those which are of the same type. However, Jacquin's choice of classification schemes did not fully exploit this technique. Using four types of block classifiers and assuming that the classification of image blocks was approximately uniform this could at most improve the speed of the algorithm by a factor of four. Unfortunately, the blocks were not classified in a uniform manner and it took time to classify them so the resulting speed improvement was actually less than this. Despite these shortcomings, the general idea of block classification is a promising one if the number of categories is selected appropriately. Unlike Jacquin's classification scheme and other extension to the generalized FBC encoder, the technique described in this chapter concentrates on systematically reducing the actual computational complexity of the encoding procedure using neural networks.

This chapter develops a reduced search FBC encoding scheme based on block classification using *vector quantization* (VQ) and neural networks. The basic VQ

procedure is described and then extended into a complete image classification scheme capable of categorizing image blocks independent of scaling, translation, or physical orientation (isometries). The *frequency sensitive competitive learning* (FSCL) neural network used to develop an appropriate set of image block categories is then introduced. Competitive learning neural networks have become the predominant approach to VQ codebook design and represent very natural implementations of vector quantizers. Finally, equations for the computational complexity for the reduced search procedure are developed. The ideal neural network size for optimal compression time improvement is then established from these equations.

6.1 Block Classification with Vector Quantization

The idea behind a reduced search algorithm employing block classification is a simple one. Rather than considering each domain block in the image as a possible source for coding a particular range block, both the range and domain blocks are classified into a number of pre-determined categories. Only those domain blocks which are classified into the same category as a particular range block are considered as a possible source for that range block. If the number of categories into which blocks may be classified is chosen judiciously, search time may be reduced dramatically.

A wide variety of image classification techniques are available but one of the simplest is *vector quantization* (VQ) a form of classification by table lookup. A basic vector quantizer consists of three components as illustrated in Fig. 6.1. The *codebook* is the most fundamental of these components and is simply a table of *prototype* vectors statistically representative of those vectors found in the actual data to be classified. A *distortion measure* such as the Euclidian metric is used to compare the relative error

between an input vector $\mathbf{x}$ and the individual codebook prototypes $\mathbf{c}_p$. The input vector is classified by searching the codebook for the prototype vector which it most closely resembles according to the distortion measure. The *minimum error detector* keeps track of the error associated with each codebook prototype and determines which is best. This 'best' prototype is commonly referred to as the *winner*. The codebook index of the winner, denoted p_{min}, is assigned to the input vector as its classification. Two or more input vectors which are best represented by the same prototype vector are considered to be of the same class.

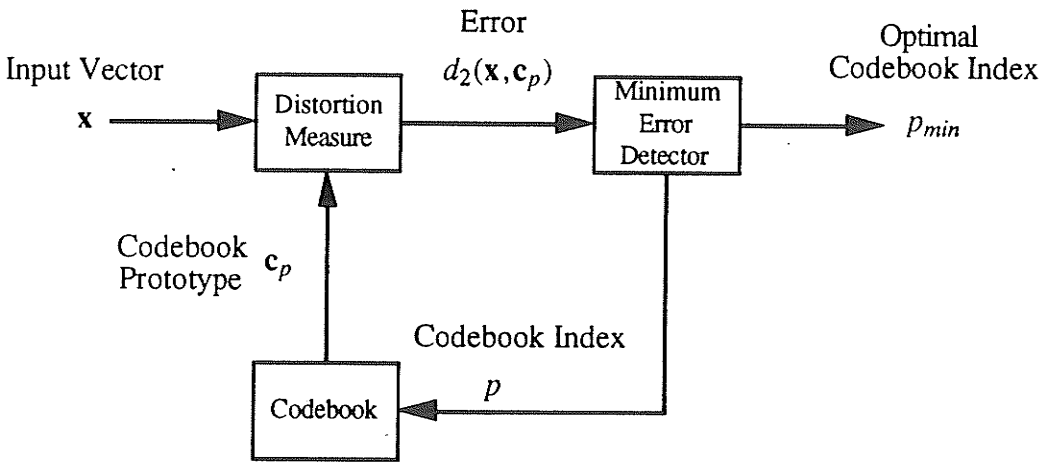


Fig.6.1 The *vector quantization* (VQ) classification scheme.

A reduced search fractal block coding algorithm employing vector quantization would contain a codebook of c vectors representing the categories into which domain and range blocks are to be classified. As indicated in Chapter 5, domain and range blocks are not compared directly. The fractal block transform has three basic components; spatial contraction, isometric block transformation, and grey level scaling and translation. The distortion between domain and range blocks is measured for optimally chosen isometries as

well as scaling and translation coefficients. During the classification procedure, the values of these coefficients are not of interest. Therefore, vectors are classified independent of these coefficients and the codebook prototypes reflect this.

To classify an image block independent of the eight isometries, we can simply compare the block against the codebook prototypes in each of its eight possible isometric configurations and choose the prototype and isometry which result in the minimum distortion. To classify a block independent of scaling or translation, any scaling or translation components already present in the block must be removed. This is procedure is referred to as *orthonormalization*. Orthonormalization is actually a two step procedure consisting of *orthogonalization* and *normalization*. Orthogonalization removes any component of the input vector $\mathbf{x}$ in the $\mathbf{u}$ direction by letting

$$\mathbf{x}' = \mathbf{x} - \langle \mathbf{x}, \mathbf{u} \rangle \mathbf{u} \tag{6.1}$$

where $\mathbf{u}$ is once again the translation vector

$$u_{ij} = 1 \qquad \text{for all } 1 \le i,j \le r \tag{6.2}$$

This eliminates the effect of grey level translation. Normalization eliminates the effect of grey level scaling and is accomplished as follows

$$\hat{\mathbf{x}} = \frac{\mathbf{x}'}{\|\mathbf{x}'\|} \tag{6.3}$$

Normalization also leads to a computational simplification of the vector quantizer. A known property of the Euclidian metric commonly used in signal processing [Fran69] is that for the normalized input vector $\hat{\mathbf{x}}$ and the two codebook prototype vectors $\mathbf{c}_p$, and $\mathbf{c}_q$

$$d_2(\hat{\mathbf{x}}, \mathbf{c}_p) < d_2(\hat{\mathbf{x}}, \mathbf{c}_q) \;\Leftrightarrow\; \langle \hat{\mathbf{x}}, \mathbf{c}_p \rangle > \langle \hat{\mathbf{x}}, \mathbf{c}_q \rangle \tag{6.4}$$

Therefore, rather than calculating the Euclidean metric for each prototype in the codebook, the computationally simpler inner product may be used. Of course since the inequalities of Eq. 6.4 are reversed, the minimum error detector of Fig. 6.1 must be replaced by a *maximum correlation detector*.

A complete isometric configuration, scaling, and translation independent vector quantizer is shown in Fig. 6.2. It is similar to the vector quantizer of Fig. 6.1 but with the addition of the orthonormalization and isometric block transform stages as pre-processing. Moreover, the maximum correlation detector which replaces the minimum error detector, must locate the optimum isometry i_{min} in addition to the winning prototype p_{min}. This is accomplished by correlating each orthonormalized input vector $\hat{\mathbf{x}}$ against each codebook prototype $\mathbf{c}_p$, in all eight of its isometric configurations denoted $\hat{\mathbf{x}}_i$ and finding the combination of p and i which maximizes $\langle \hat{\mathbf{x}}_i, \mathbf{c}_p \rangle$.

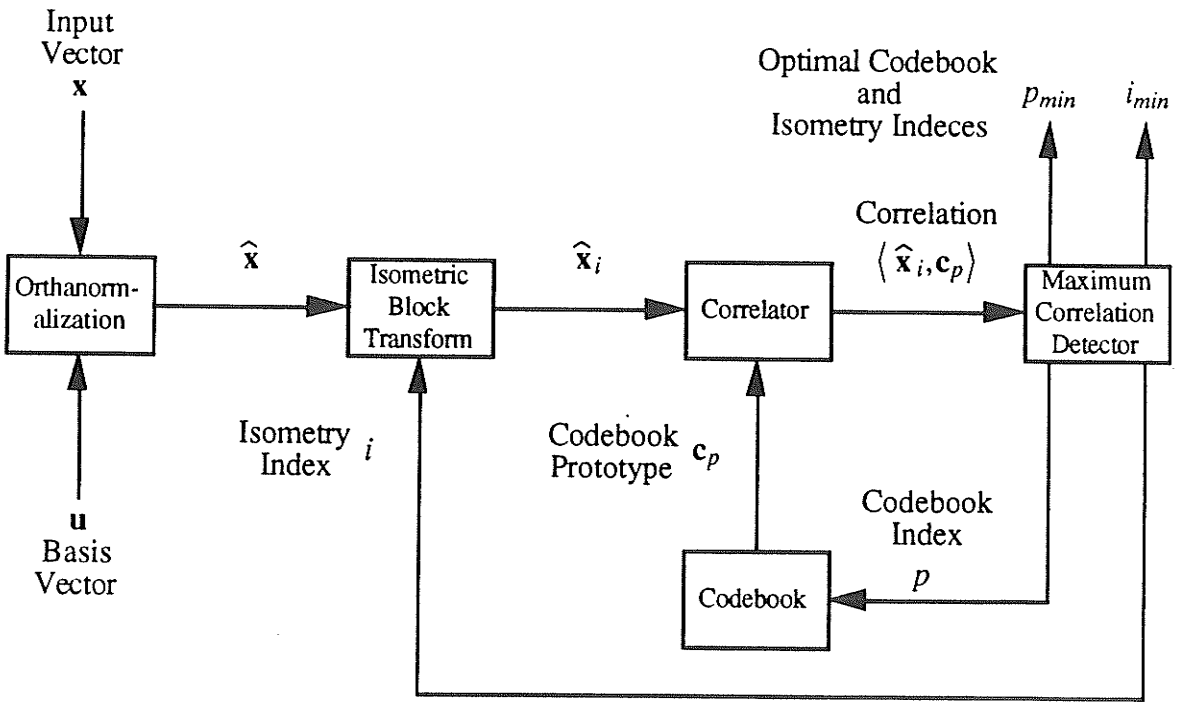


Fig. 6.2. A scaling, translation, and isometric configuration independent vector quantizer.

In the full search algorithm, it was necessary to consider all eight isometries to locate the best isometric block transform between domain and range blocks. Similarly, when classifying image blocks, it is necessary to compare all eight orientations of the input block against each prototype vector in the codebook. However, it has already been indicated that the eight isometries described in Chapter 5 form a group. This implies that repeated application of any two isometries results in a third isometry also in the group and that each isometry has an inverse. These two properties can be exploited to further reduce the search time. Although it is necessary to locate the isometry which best maps each range and domain block into the codebook, the isometry which best maps the domain block into a particular range block of the same class can be calculated directly. Suppose that the contracted domain block **x** is mapped into a codebook prototype $\mathbf{c}_p$ by an isometry I_x, and that the range block **y** is mapped into the same codebook prototype by an isometry I_y. It follows that the function which best maps **x** into **y** is the combined isometry I_{xy} given by

$$I_{xy} = I_x I_y^{-1}. \tag{6.5}$$

6.2 Neural Network Codebook Design

During the discussion of the vector quantizer, it was briefly stated that the VQ codebook consists of a 'table of prototype vectors statistically representative of those vectors found in the actual data to be classified'. Up to this point, nothing has been said regarding how such a table is obtained. Two basic approaches exist for solving this problem; they include *clustering algorithms* and *neural networks*. In the past, clustering algorithms such as the *Linde-Buzo-Gray* (LBG) [LiBG80] and *k-means* algorithms [KaRo90] were used to design appropriate VQ codebooks, however, neural networks have

recently emerged as the approach of choice.

Neural networks are an alternative paradigm for computing which possess the ability to learn from experience. Neural networks, unlike traditional computers, are inherently parallel consisting of two fundamental components; neurons and weighted connections. The neurons are the processing elements of the neural network and transform the neuron inputs into a single output via some simple (usually non-linear) transfer function. The weighted connections serve as both the data paths and memory of the neural network. Each connection has associated with it a particular weight. Neuron activations are transmitted along these connections and are modified as a function of the weight values. The network learns by adapting these weights according to a *learning rule*. A set of *training patterns* is presented to the network for which it learns the combination of weights which best satisfy the constraints of the learning rule. There are many forms of networks but they are most commonly grouped into two categories: *supervised* (training) and *unsupervised* (learning). Supervised networks require that a target activation be associated with each training pattern. The network learns the appropriate mapping between each training pattern and the associated target activations. The best examples of supervised learning networks are those based on the *back-propagation* algorithm [McRu88]. Unsupervised networks perform what is commonly referred to as *regulatory discovery* [RuMc86]. This means that they attempt to discover statistical properties of the training data irrespective of any target values. These types of networks are especially useful for clustering or categorization applications.

One form of unsupervised learning network which is particularly appropriate for the implementation of vector quantizers is the *competitive learning* network [HeKP91]. *Frequency sensitive competitive learning* (FSCL) is an extension of this basic form.

6.2.1 Competitive Learning

Figure 6.3 illustrates a simple competitive learning network. This network consists of two distinct layers of neurons: an input layer and an output layer whose activations are denoted a_q and o_p, respectively. The neurons in the output layer are fully connected to each other by a matrix of inhibitory connections. The neurons in the input layer are fully connected to the output layer by a matrix of excitory connections $\mathbf{w}$. The array of excitory connections $\mathbf{w}_p$ joining the input layer to an individual output neuron p must be normalized to ensure that

$$\|\mathbf{w}_p\| = \sum_{q=1}^{N} (w_{pq})^2 = 1 \tag{6.6}$$

where N is the number of neurons in the input layer and w_{pq} denotes the connection from the q^{th} input to the p^{th} output neuron. This can be accomplished via the same procedure used for normalizing image blocks by letting

$$\widehat{\mathbf{w}}_p = \frac{\mathbf{w}_p}{\|\mathbf{w}_p\|} \tag{6.7}$$

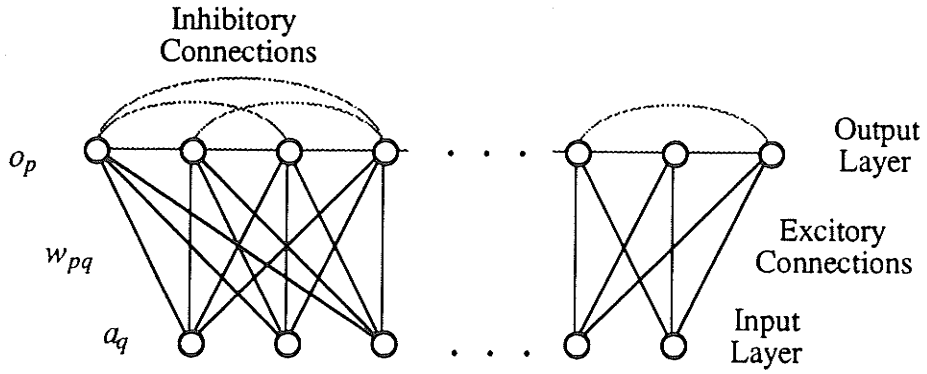


Fig. 6.3. A competitive learning neural network.

A competitive learning neural network is in fact equivalent to a vector quantizer with normalized input and prototype vectors. The input activations a_q are the individual elements of an image block $\mathbf{x}$, the output neurons are the codebook indices p, and the excitory weights $\mathbf{w}_p$ connecting the input layer to each output neuron are the codebook prototypes $\mathbf{c}_p$. The only additional component required to implement the isometric configuration, scaling, and translation independent vector quantizer of Fig. 6.2 is the isometric block transform.

The competitive learning network operates in one of two different modes: *classification* or *learning*. In the classification mode, an input pattern is applied to the input neurons. The activations of the input neurons are then propagated through the excitory connections to the output layer. Each input activation a_q is multiplied by the weight w_{pq} of the connection through which it propagates. The net input i_p to a neuron p is the sum of these weighted activations or

$$i_p = \sum_{q=1}^{N} w_{pq} a_q = \langle \mathbf{w}_p, \mathbf{a} \rangle \tag{6.8}$$

The output units are mutually exclusive and must compete for activation via the inhibitory connections. The neuron with the maximum net input is always the winner of this competition. The activation of a neuron p in the output layer is therefore given by

$$o_p = \begin{cases} 1 & \text{if } i_p > i_u \text{ for all } u \neq p \\ 0 & \text{otherwise} \end{cases} \tag{6.9}$$

The inhibitory weights are fixed while the excitory weights are dynamic. The network determines the set of excitory weights which best represent the statistical properties of the input data according to the algorithm described in Fig. 6.4. During this

procedure, the network is said to be in learning mode.

STEP 1: Initialize the connection weights to the mean of all of the vectors in the training set plus some small random perturbation.

STEP 2: Present a randomly chosen vector from the training set to the network.

STEP 3: Update the connections to the winning neuron in proportion to the learning rate.

STEP 4: Decrease the learning rate. Goto STEP 2.

Fig. 6.4. Competitive learning neural codebook design algorithm.

In the learning mode, vectors are chosen at random from the training set and presented to the network. The input activations are propagated through the network and the output activations are determined as before. The weights of the network are then updated according to the following learning rule

$$\widetilde{w}_{pq} = w_{pq} + \Delta w_{pq} \tag{6.10}$$

where

$$\Delta w_{pq} = \eta o_p(a_q - w_{pq}) \tag{6.11}$$

and η is referred to as the learning rate. The learning rate is initially large (between 0.1 and 0.7) but decreases with time. Since the the weight change Δw_{pq} is proportional to the output neuron activation and only one neuron is active at any given time, only the weights

connecting the input layer to the winning neuron are updated. After updating, the weights must be re-normalized as per Eq. 6.7.

One final note concerning implementation of the scaling, translation, and orientation invariant vector quantizer of Fig. 6.2 is in order . While orthonormalizing the input blocks will remove any effects caused by scaling or translation, block orientation must still be accounted for. This is accomplished by presenting each input block to the network in all eight of its isometric configurations during both the classification and learning modes. The winning neuron and associated isometry are selected only after all eight sets of activations have been propagated through the network and evaluated. In the learning mode, the winning neuron must be updated with the input vector in the best of these eight configurations.

6.2.2 Frequency Sensitive Competitive Learning

One problem with the standard competitive learning algorithm is that some neurons may never win the competition. Should this occur, the neurons in question will never come to represent any significant features within the image and for all intents and purposes be wasted. To ensure that no single neuron is continuously left out of the competition, Ahalt *et al.* have introduced a *conscience mechanism* which they call *frequency sensitive competitive learning* (FSCL) [AKCM90]. The idea behind a conscience mechanism is quite simple. If, during learning, a neuron wins the competition too often it should begin feeling 'guilty' and temporarily shut itself off to allow other neurons to become active [DeSi88]. In FSCL this is accomplished by dividing the net input to each neuron by the number of times f_p that the neuron has won the competition. This results in the following equation for net input:

$$i_p = \frac{1}{f_p} \sum_{q=1}^{N} w_{pq} a_q \qquad (6.12)$$

Actually, FSCL ensures that during learning the weights associated with each neuron will be updated an approximately equal number of times. This is a particularly important result since equal prototype utilization is an assumption which must be made when discussing the overall performance of the reduced search FBC encoding procedure in the next section.

6.3 Performance of the Reduced Search Coding Procedure

From the complete description of the reduced search FBC encoding procedure it is now possible to calculate a time complexity for the reduced search algorithm. Consider an nxn image with domain and range blocks of size dxd and rxr respectively, and a codebook of size c (a competitive learning neural network with c output neurons). Each domain and range block in the image must first be classified. This classification associates a class and isometry with the vector in question. Then, all of the domain/range block pairs belonging to the same class are compared using a combined isometry calculated from the individual isometries associated with each block. Assuming that the vectors in the input data are distributed equally amongst the c categories, the time complexity of the reduced search algorithm is

$$T_c = 8(n-d)^2 c + 8\left(\frac{n}{r}\right)^2 c + \frac{(n-d)^2\left(\frac{n}{r}\right)^2}{c} \qquad (6.13)$$

This includes the time required to classify all of the domain and range blocks and still appears to be $O(n^4)$. However, minimizing Eq. 6.13 with respect to c yields

$$c = \frac{1}{2\sqrt{2}} \frac{n(n-d)}{\sqrt{n^2 + r^2(n-d)^2}} \approx \frac{1}{2\sqrt{2}} \frac{n}{r} \qquad (6.14)$$

Now substituting Eq. 6.14 back into Eq. 6.13 results in the new time complexity

$$T_{min} = 4\sqrt{2}(n - d)^2\frac{n}{r} + 2\sqrt{2}\left(\frac{n}{r}\right)^3 \tag{6.15}$$

which is $O(n^3)$. To further illustrate the effect which this technique has on search time, we can divide Eq. 5.15, the time complexity for the full search algorithm, by Eq. 6.15 to derive the following equation for the total compression time improvement

$$\frac{T}{T_{min}} = \frac{2\sqrt{2}(n - d)^2\frac{n}{r}}{2(n - d)^2 + \left(\frac{n}{r}\right)^2} \approx \sqrt{2}\frac{n}{r} \tag{6.16}$$

For 256x256 images with 8x8 range blocks this represents a speed improvement by a factor of 45.

It must be pointed out that the reduction in compression time described by Eq. 6.16 is achieved at a certain expense since the exhaustive and reduced search coding procedures are not strictly equivalent. The exhaustive search procedure compares every range block against every domain block in each of its eight possible isometric configurations. The exhaustive search procedure is, in this sense, optimal. In contrast, the reduced search procedure places each range and domain block into one of c categories. It is possible for a particular range block and the best domain block, to be placed into different categories. This is illustrated two dimensionally in Fig. 6.5. The vector quantizer divides the set of all domain and range blocks into c disjoint subsets. A range block $\mathbf{y}$ and a domain block $\mathbf{x}_1$ may be very similar but placed into different categories if they lie on or near the borders of these categories. When this happens, a sub-optimal domain block $\mathbf{x}_2$ will be chosen from the same category as the range block. Although this will result in some image degradation, the domain block chosen will usually be a very good match for the range block in question.

In fact, coding results indicate that this degradation is generally not detectable by the human eye.

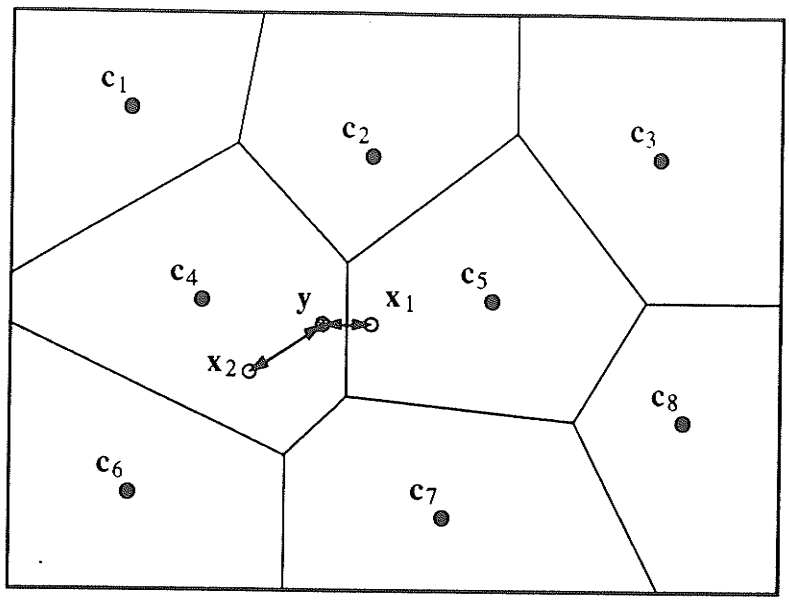


Fig. 6.5. Misclassification of similar vectors. The vector quantizer divides the set of all possible domain and range blocks into disjoint subsets centered on the prototypes $\mathbf{c}_1$ through $\mathbf{c}_8$. The range block $\mathbf{y}$ and the optimal domain block $\mathbf{x}_1$ are placed in the categories $\mathbf{c}_4$ and $\mathbf{c}_5$ respectively. The sub-optimal domain block $\mathbf{x}_2$ will be chosen as the source for the range block $\mathbf{y}$.

6.4 Summary

This chapter described a reduced search FBC encoding procedure based on domain/range block classification using a frequency sensitive competitive learning neural network which is equivalent to a vector quantizer. The network classifies image blocks independent of scaling, translation, and isometric configuration. The basic encoding procedure requires that every range and domain block in the image be classified by the neural network and then domain blocks are only considered as possible sources for a

particular range block if they are of the same class.

The FSCL network was chosen because it exhibits equal prototype utilization. This implies that the vectors in both the domain and range pools will be allocated approximately equally amongst the c classes determined by the network. Equal prototype utilization is a fundamental assumption made in determining the computational complexity of the reduced search coding procedure. For an nxn image with rxr range blocks this complexity is minimized by employing a vector quantizer with $\left\lceil \frac{1}{2\sqrt{2}} \frac{n}{r} \right\rceil$ prototypes or equivalently, a two layer competitive learning neural network with the same number of output neurons. A network of this size will reduce compression time by a factor of $\sqrt{2} \frac{n}{r}$ and reduces the complexity of the encoding procedure from $O(n^4)$ to $O(n^3)$. In addition, the systematic reduced search procedure discussed in this chapter does not preclude the use of heuristic time saving techniques such as those described in Chapter 5. For example, the reduced search FBC procedure could be combined with a smaller domain pool and extended grey level translation functions proposed by Oien *et al.* [OiLR92] to further reduce compression time.

CHAPTER VII

IMPLEMENTATION

Both the reduced and exhaustive search fractal block coding procedures described in Chapters 5 and 6 were implemented in the context of the *concatenated* image compression system illustrated in Fig. 7.1. This system takes gray scale images and compresses them using either the reduced or exhaustive search FBC encoding procedure. The resulting fractal code is then further compressed using *arithmetic entropy coding* resulting in an *arithmetic code stream.* This code stream, representing a doubly encoded version of the original image, is then either stored or transmitted. The original image is reconstructed by first decompressing the arithmetic code stream and then applying the iterative FBC reconstruction procedure to the resulting fractal code. Since the fractal block code is encapsulated within the arithmetic code stream, FBC is referred to as the *inner* code, while arithmetic coding is called the *outer* code.

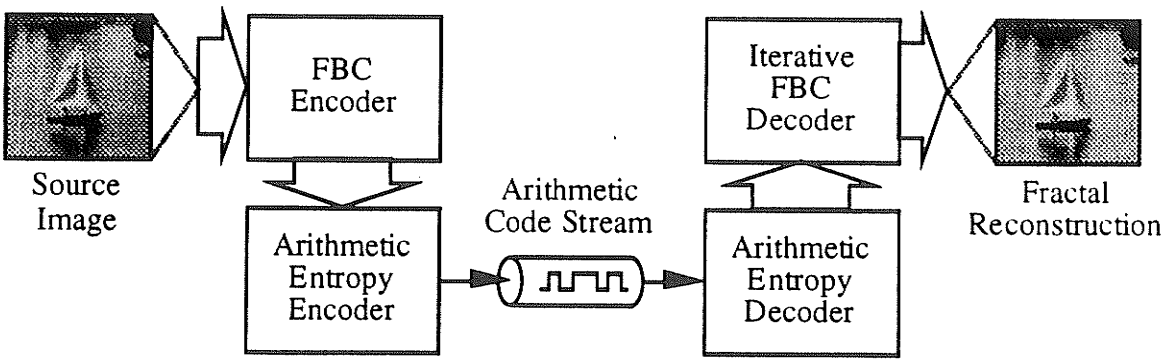


Fig. 7.1. A concatenated image compression system based on fractal block coding and arithmetic entropy coding.

In contrast to FBC, arithmetic coding is a *lossless* data compression technique. This implies that, using arithmetic coding, the FBC paramters can be compressed and then reconstructed exactly. Therefore, the addition of an arithmetic encoder and decoder to the basic FBC technique will improve the overall compression ratio but without introducing any further distortion into the reconstructed image. This improvement, while modest, is still significant and adds relatively little overhead to the entire coding procedure.

This chapter describes the implementation of the concatenated FBC/arithmetic image compression system. The FBC and arithmetic coding subsystems are discussed independently. A description of the FBC encoding algorithms is provided followed by a derivation of the compression ratios resulting from FBC in the absence of entropy coding. Entropy coding is then introduced along with the generalized arithmetic encoding and decoding procedures. The chapter concludes with a brief description of the complete FBC/arithmetic software implementation. The *C* language source code for this implementation is supplied in Appendix D.

7.1 FBC Implementation

The reduced search FBC portions of the concatenated coding system can be further divided in to the four macro-functions `learn codebook`, `classify range image`, `fractal code image`, and `decode fractal image` illustrated in Fig. 7.2. These functions are initiated in the proper sequence from the function `main`.

The `learn codebook` function generates an appropriate set of VQ prototypes from a training image using the FSCL learning rule described in Section 6.2. Training blocks are selected at random from the domain pool of a training image and propagated

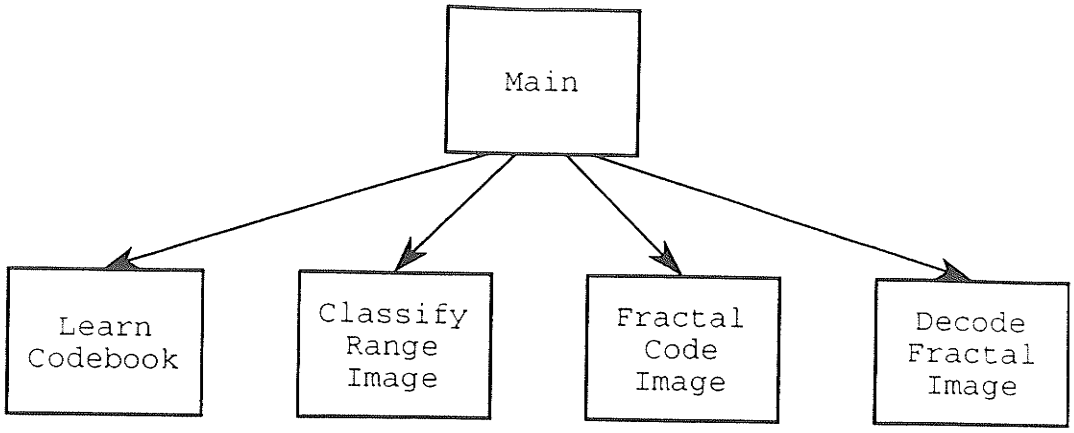


Fig. 7.2. Functions required for the implementation of reduced search FBC encoding and decoding.

through the network. These blocks are orthonormalized and then compared against every prototype in the network in a single isometric configuration before considering the remaining configurations. This improves the overall efficiency of the function since the training block is transformed eight times throughout the entire classification procedure, rather than eight times per codebook prototype. After the best prototype and isometry have been established the network weights are updated using the FSCL learning rule (Eqs. 6.13 and 6.14). The network is trained on a total number of training blocks equal to 500 times the size of the codebook [McAR90] . The learning rate η decreases linearly over time from values of 0.2 to 0.01 so that general features are established during the early stages of learning while fine tuning is accomplished later on.

Training vectors were selected exclusively from the domain pool for two reasons. Firstly, the domain pool is much larger than the range pool. To ensure optimal time performance, it is therefore more important that the domain blocks, rather than the range blocks, are equally distributed amongst the codebook prototypes. Secondly, if the training

set contains both range and domain blocks then the network might locate certain features which are present only in the range pool or the domain pool, not in both. Under these circumstances, the network could assign range blocks to a particular category for which no domain blocks had been allocated. It would not be possible to properly represent these range blocks since the encoder would be unable to locate suitable domain blocks as sources.

The `classify range image` function uses the codebook generated by the `learn codebook` function to classify the individual range blocks in the image. Each range block is presented to the FSCL network which identifies the appropriate class and corresponding isometry using the VQ procedure described in Sections 6.1 and 6.2. These classifications are retained for comparison against domain blocks in the `fractal code image` function.

The `fractal code image` function produces the FBC description of the source image using the reduced search encoding procedure described in Section 6.1. In this function, an individual domain block is extracted from the domain pool and classified by the neural network. This domain block is then considered as a source for all range blocks assigned to the same class by

(1) determining the appropriate isometry as per Eq. 6.4,
(2) calculating the optimal scaling an translation components (Eqs. 5.4 and 5.5),
(3) calculating the Euclidian distance between the range block and the appropriately transformed domain block.

If the distance between the domain block and a particular range block is less than the distance between that range block and any domain block considered thus far, then the domain block and associated transformation parameters are assigned to that range block as

its fractal representation. This procedure is then repeated for the remaining domain blocks in the domain pool.

The `decode fractal image` function generates an approximate reconstruction of the original image from the fractal code using the iterative reconstruction procedure described in Section 5.3. The function begins with an entirely black image (all pixel values set to zero) and, using the fractal code, transforms this image into a second image. This image is similarly transformed and the process is repeated for seven to nine more iterations. The number of iterations has been selected based on results obtained experimentally [Jacq92].

Structure charts describing the complete reduced search FBC procedure are supplied in Appendix C along with technical descriptions of each function. The exhaustive search procedure is simply a subset of the reduced search procedure which does not include the `learn codebook` or `classify range image` functions. As a result, the exhaustive search `fractal code image` function compares every block in the domain pool directly against every range block in the image. The `decode fractal image` function remains the same for both the reduced and exhaustive search procedures.

7.2 Calculation of Compression Ratios for FBC

Section 5.2 described the fractal block code as containing

(1) a pointer to the best domain block,
(2) a pointer to the best isometric transform,
(3) an optimal scaling coefficient, and
(4) an optimal translation coefficient

for each range block in the image. The number of bits b_R required to represent each range block is given by the sum of the number of bits required to represent each parameter in the individual fractal block transform or

$$b_R = b_D + b_I + b_A + b_T \tag{7.1}$$

For an nxn image encoded with dxd domain blocks the number of bits b_D required to address a unique domain block in the domain pool is given by

$$b_D = \lceil 2\log_2(n\text{-}d) \rceil \tag{7.2}$$

The eight isometries were distinguished by $b_I = 3$ bits per range block while the scaling and translation coefficients were represented with $b_A = 11$ and $b_T = 9$ bits, respectively. This was sufficient for representing fixed point scaling values between ±4.0 and integer translation values between ±255.

The compression ratio achieved by the FBC encoder can be calculated based on the number of bits required to represent each range block in both its original and coded and forms. Using rxr range blocks, this ratio is given by

$$compression\ ratio = \frac{r^2 b_P}{b_R} \tag{7.3}$$

where b_P is the number of *bits per pixel* (bpp) in the original image. For a 256x256 eight bpp image encoded using 8x8 and 16x16 range and domain blocks respectively, $b_R = 39$. This results in a compression ratio of 13.1:1.

The compression ratio can be selected by choosing appropriately sized range blocks or by reducing the number of unique blocks in the domain pool thus decreasing b_D.

Alternatively, the number of bits b_A and b_T used to represent the scaling and translation coefficients can be reduced. The values $b_A = 11$ and $b_T = 9$ represent the number of bits required to express these paramters exactly. Both the scaling and translation coefficients can be quantized into a number of distinct values represented with fewer bits. This quantization may be linear or more preferably non-linear but in either case will introduce some distortion into the reconstructed image.

The compression ratios given by Eq. 7.3 represent the amount of compression achieved strictly by the FBC encoder. The overall compression ratio will increase by between 5% and 20% when arithmetic entropy coding is added to the system.

7.3 Arithmetic Entropy Coding

Following the FBC encoding procedure, the resulting fractal code may still contain some statistical redundancy within its parameters. This redundancy can be removed and the fractal code further compressed using *statistical coding* techniques [Kins91]. These techniques attempt to generate a minimal *entropy* representation of a source file based on the relative probability with which each symbol in that file is likely to occur. In this respect, *arithmetic coding* is generally considered to be superior to all other statistical techniques including *Shannon-Fano* and *Huffman* coding [Huff52]. In addition, arithmetic coding possess certain fractal characteristics making it a particularly appropriate addition to any fractal based coding scheme.

Entropy is a thermodynamic principle which can be applied to a data source as a measure its *information* content [Shan49]. The entropy of a randomly generated data stream containing multiple incidents of M unique symbols is given by

$$H = -\sum_{i=1}^{M} p_i \log_2 (p_i) \tag{7.4}$$

where each p_i is the probability of occurrence for the i^{th} unique symbol. Furthermore, it is impossible to precisely represent this data stream in fewer average bits per symbol than the value indicated by its entropy.

It is the objective of lossless data compression techniques, including entropy based or *statistical* coding to develop a code which represents some source message in a minimal number of bits by approaching, as closely as possible, the entropy bound of Eq. 7.4. This can be accomplished by developing a statistical model of the data to be coded and then representing each symbol in this data with a number of bits inversely proportional to its probability of occurrence. The optimal number of bits per symbol, λ_i, follows directly from the entropy measure of Eq. 7.4.

$$\lambda_i = \log_2 \left(\frac{1}{p_i} \right) = -\log_2 (p_i) \tag{7.5}$$

In this way, symbols in the source data occurring more frequently are represented by fewer bits than those which are statistically less common.

7.3.1 Arithmetic Encoding Procedure

An arithmetic code consist of a fixed point value representing an interval between 0.0 and 1.0. This interval is indicative of, and unique to, the particular message being coded. Figure 7.4 shows an example of the arithmetic encoding procedure for the word *eat* followed by an *End of File* (EOF) character. The encoder begins with the open interval [0.0,1.0) and subdivides it into M subintervals, where M is the number of unique

symbols in the source stream. Each subinterval represents a unique source symbol, and the size of the interval is proportional to that symbol's probability of occurrence, p_i. For a given source symbol, the encoder locates the corresponding subinterval, and then divides this interval into subintervals whose ratios are the same as the original cumulative probabilities. The encoder finds the appropriate subinterval for each successive symbol. As this subinterval is located within the previous interval, it represents not only the present but also the past symbols. This process continues recursively until the entire source stream has been encoded, at which time the encoder transmits the final interval.

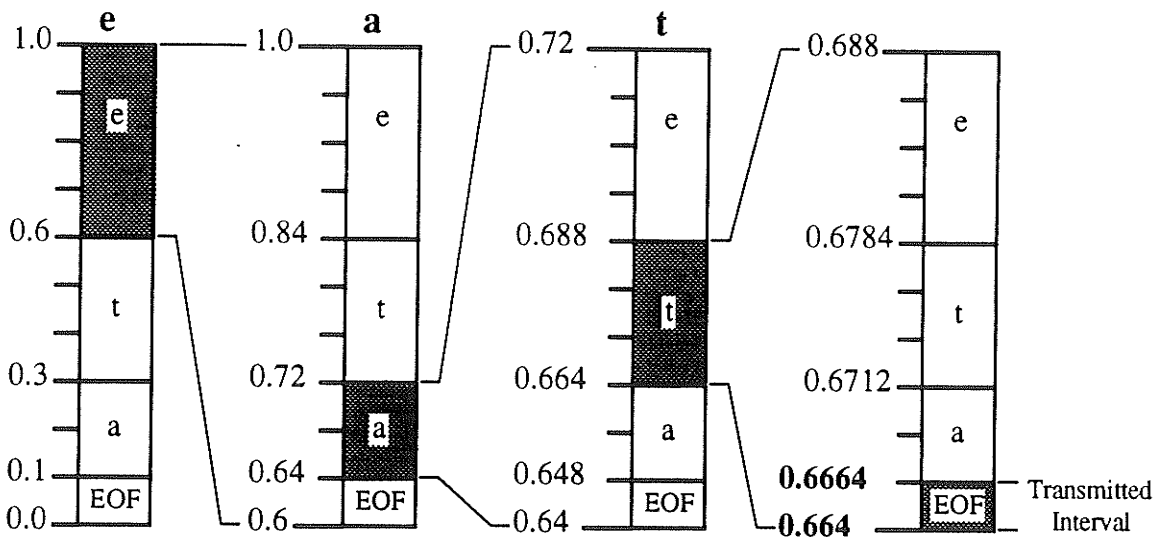


Fig. 7.3. Arithmetic encoder. Example encoding of the symbol stream "e•a•t•EOF" [WiNC87][Kins91].

The arithmetic encoding procedure bears some resemblance to the MRCA since, at each iteration, the entire structure is divided up into M self-similar components. Since each subdivision of the original interval [0.0,1.0) is constructed out of M reduced copies of itself, arithmetic coding appears to be fractal in nature [Kins91]. Although this is a very

recent observation, the basic principle of arithmetic coding was first associated with the Hausdorff-Besicovitch or fractal dimension in an obscure paper on information theory published in 1961 [Bill61]. Unfortunately, since the term fractal did not even exist prior to 1976, it was impossible to make any connection between arithmetic coding and fractals at the time.

7.3.2 Arithmetic Decoding Procedure

The arithmetic decoder recovers the source symbols from the received interval using a procedure similar to that of the encoder, as shown in Fig. 7.4. Like the encoder, the decoder begins with the open interval [0.0,1.0) subdivided into the same M subintervals. The decoder locates the subinterval in which the received interval resides, yielding the first symbol in the stream. This subinterval is further divided in the same manner to recover subsequent symbols. The procedure terminates when the current and received intervals are equivalent. At this point the entire source stream has been decoded.

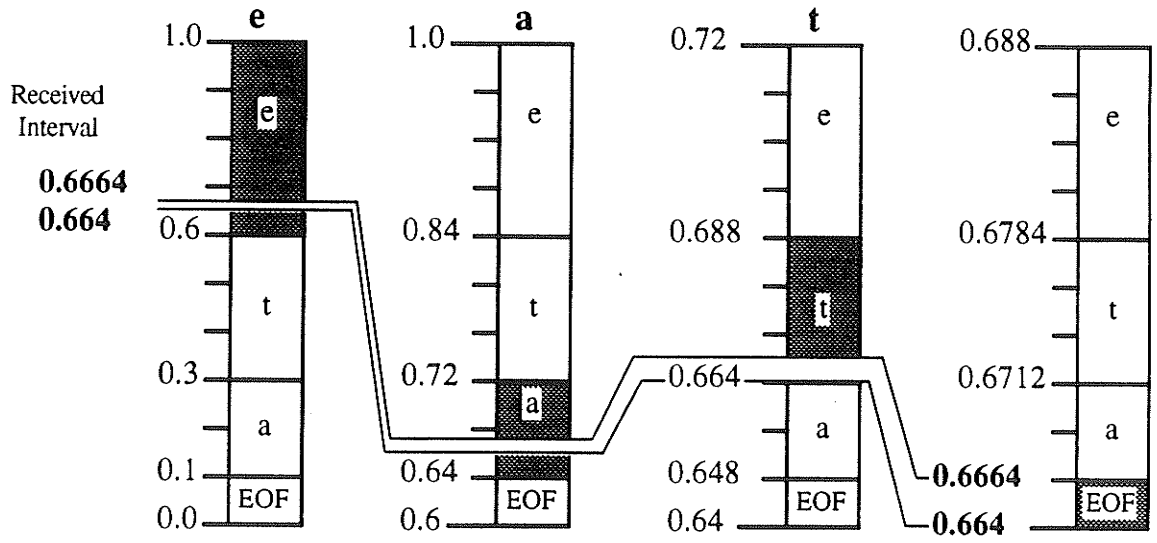


Fig. 7.4. Arithmetic decoder. Example decoding of the interval [0.664, 0.6664) into the data stream "e•a•t•EOF" [WiNC87][Kins91].

A complete description of practical algorithms for arithmetic encoding and decoding is described in [WaFK93]. The implementation of arithmetic coding used in this thesis is based upon that description.

7.3 Software Organization

The complete concatenated FBC/arithmetic compression system was implemented in the *C* programming language because C is versatile, efficient, and popular amongst software developers. Unfortunately, modern software engineering techniques such as *information hiding* [Pfle87] are often more difficult to enact in C than in other languages. C does not enforce 'safe' programming practices and therefore places the onus of software reliability entirely on the programmer. For this reason, it is extremely important to follow a systematic design philosophy when developing C applications.

The FBC and arithmetic coding procedures outlined in Sections 7.1 and 7.3 were implemented using a *modular* design approach [Vela91] based on constructs available in the programming language ADA. This approach requires that an application be divided into a number of smaller carefully organized modules. A single module only contains functions which operate on the same data structures or perform logically related tasks. Each module is further divided into two segments referred to as *interfacing* and *implementation*. The interfacing segment is always contained within a C *header* (**.h**) file and includes only the declarations of *public* functions and data structures required for inter-module communication. The implementation segment contains the executable code for these public functions as well as any *private* data structures, functions, and variables. The implementation segment always takes the form of a C source (**.c**) file. A module which

requires the services of a second module must simply **include** that modules header file in its own implementation segment.

The software developed for the FBC/arithmetic compression system consists of the seven modules shown in Fig. 7.6. These modules are arranged hierarchically with subordinate modules appearing beneath their calling modules. All modules access the module `CONSTANTS` which contains global system constants and data structures. The module `MAIN` contains only one function which initiates other portions of the FBC coding procedure and provides a simple user interface to display coding status. The `FRACTALS` module implements all of the functions associated with the FBC encoding and decoding algorithms as well as utility functions to display the resulting fractal codes. The module `FSCL` contains all of the executable code associated with the frequency sensitive competitive learning neural network. These include functions to initialize the network, learn an appropriate set of image prototypes, and classify image blocks. All of the image block transformations are performed by functions located within the `TRANSFORMS` module. These include the spatial reduction, range block isometries, grey level scaling and translation, as well as orthonormalization functions. Implementations of both the arithmetic encoding and decoding procedures are included within the module `ARITHMETIC`. Finally, the module `IO` contains executable code for dynamically allocating memory for complex data structures such as images and the connection weight matrix of the neural network. Disk I/O routines for saving and retrieving the data contained within these structures are also provided.

C language listings for all seven modules, representing the entire reduced search FBC/arithmetic compression system, are provided in Appendix D. These listings consist of 2200 lines of C source code which was compiled into a 45 Kbyte executable program on

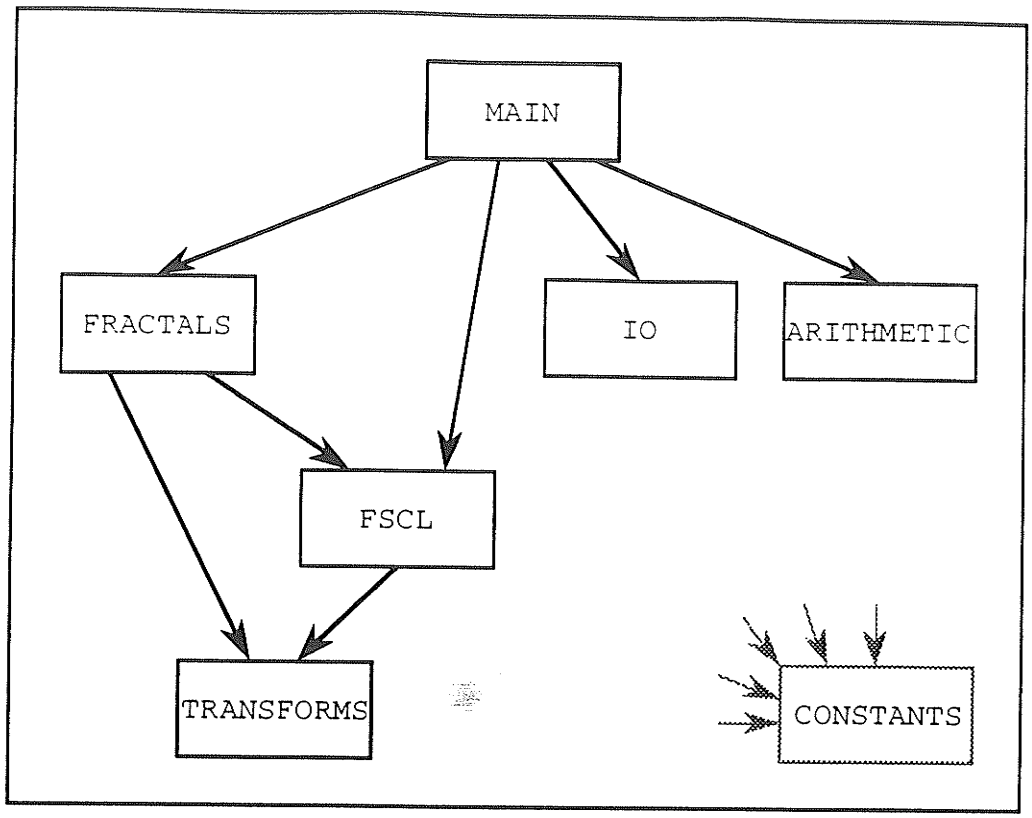


Fig. 7.6. The reduced search FBC program module hierarchy. All modules reference the module Constants.

a SUN SPARCStation 2 using the *cc UNIX* compiler with the -O optimization setting. The FBC image compression experiments described in the next chapter were all conducted using this program.

7.4 Summary

This chapter described the software implementation of a concatenated image compression system with FBC as the inner code followed by arithmetic entropy coding as the outer code. This concatenated scheme will, in general, produce higher compression ratios than either FBC or arithmetic coding alone. Since arithmetic coding is a lossless

compression scheme, the fractal code can be compressed and then reconstructed from the arithmetic code without further effecting image quality. In addition to improving compression ratios, arithmetic coding possesses certain fractal characteristics and therefore seems particularly appropriate as an addition to any fractal compression scheme.

The software was designed using a modular approach and implemented in C under UNIX on the SUN SPARCStation 2 platform. Modularity assisted in software testing and facilitated incremental development. The system was implemented on the SPARC 2 because of that workstations high performance (28 MIPS) and the 'barrier-free' environment provided by UNIX. In contrast, the software could be ported to standard MS-DOS platforms but that operating systems 16 bit framework would limit images to less than 64 Kbytes (256x256 pixels). The software was successfully ported to an *extended* DOS environment using the freeware C/C++ compiler *gcc* and DOS extender *go32* [Delo92]. With the DOS extender, the software was executable only on IBM 386 and 486 type computers running DOS 4.0 or higher, however, image dimensions were limited only by the size of the computers internal memory. On DOS machines one MEG of internal RAM and a floating point processor are highly recommended. Although the experiments in the following chapter were all conducted on the SPARC 2, the extended DOS implementation was tested on a 33-MHz 486 DX under DOS 6.0. With this configuration, compression times were approximately 1.5 times that of the SPARCStation suggesting that compression time might actually be improved on a 50-MHz 486 DX or 66-MHz 486 DX2.

CHAPTER VIII

EXPERIMENTAL RESULTS

Subjective and objective evaluation of the reduced search fractal block coding (FBC) procedure was performed using the software implementation outlined in Chapter 7. Experiments were conducted to

(1) determine the reconstruction quality of FBC encoded images using both objective and subjective means,

(2) demonstrate the ability of the FSCL neural network to generalize,

(3) investigate FBC's ability to reconstruct images at larger than their encoded size without exhibiting edge degradation by the 'staircase' effect, and

(4) compare FBC against two popular image compression techniques based on transform coding and vector quantization.

All experiments were performed on the same SUN SPARCStation 2 platform.

The majority of the encoding experiments were conducted with the 256x256 eight *bits per pixel* (bpp) test image *Lena* shown in Fig. 8.1 (although originally named Lenna Sjööblom with two *n*s [Swed72], the incorrect spelling has become as common in image processing literature as the picture itself). This image was selected as a suitable test image for three reasons. Firstly, *Lena* can not be compressed appreciably using traditional lossless compression schemes. For example, the LZW algorithm [DuKi91] is capable of compressing this image by only 1.58% or 1.02:1. Secondly, the image is composed of a

diverse collection of visually significant features. Portions of the background are very smooth and contain areas in which intensity changes slowly and uniformly. In contrast, the feathers in *Lena's* hat represent an area of very high complexity. Subtle textures occur in the ribbon surrounding this hat while sharp edges are well represented by the border of *Lena's* shoulder and throughout the background. The final motivation for selecting *Lena* as a test image is based simply on its popularity. *Lena* occurs throughout image processing literature more than any other photograph and therefore makes possible direct comparison between the current implementation and the work of others.

Original Image

Fig. 8.1. The original 256x256 eight bpp image Lena.

8.1 FBC Image Compression Experiments

Both the exhaustive and reduced search coding procedures were used to code the original *Lena* image. Both implementations used 8x8 range and 16x16 domain blocks. For 256x256 images, the entire set of overlapping domain blocks comprised the domain pool. For each range block, a total of 16 bits were required to reference the appropriate domain block from the domain pool. The eight isometries were coded with three bits per range block. The scaling and translation coefficients a and t were represented with 11 and 9 bits respectively. Before entropy coding, this produced a compression ration of 13.1:1 or 0.61 bpp. For the reduced search procedure, a codebook containing 11 prototypes was used.

Compression time for the full search procedure was over 13 hours on a SUN SPARCstation 2. On the same workstation, the reduced search algorithm required approximately 18 minutes. The result of compression by the reduced search procedure after eight iterations of the reconstruction procedure is shown in Fig. 8.2. The first six iterations of this same procedure are shown in Fig. 8.3.

8.1.1 Objective Analysis

Quantitatively, the quality of the decompressed images are measured according to the *peak signal-to-noise ratio* (PSNR). For the original image $\mathbf{X}$ and its reconstruction $\mathbf{X}^*$ this is given by

Fig. 8.2. Fractal reconstruction of Lena compressed by 14.3:1 at 0.56 bpp and 29.09 dB using reduced search *fractal block coding* (FBC).

$$PSNR = 10 \log_{10} \left\{ \frac{d_2(\mathbf{X},\mathbf{X}^*)^2}{255^2} \right\}$$

$$= 10 \log_{10} \left\{ \frac{\sum_{i=1}^{n} \sum_{i=1}^{n} \left(X_{ij} - X_{ij}^*\right)^2}{255^2} \right\} \qquad (8.1)$$

For the 256x256 version of *Lena* the exhaustive search and reduced search procedures resulted in reconstructed images with PSNRs of 29.38 dB and 29.22 dB, respectively. After entropy coding, the compression ratio achieved for both these procedures was 14.3:1 or 0.56 bpp – an improvement of 9%. The 0.16 dB loss in quality resulting from the reduced search procedure can be attributed to block misclassification. Range blocks which

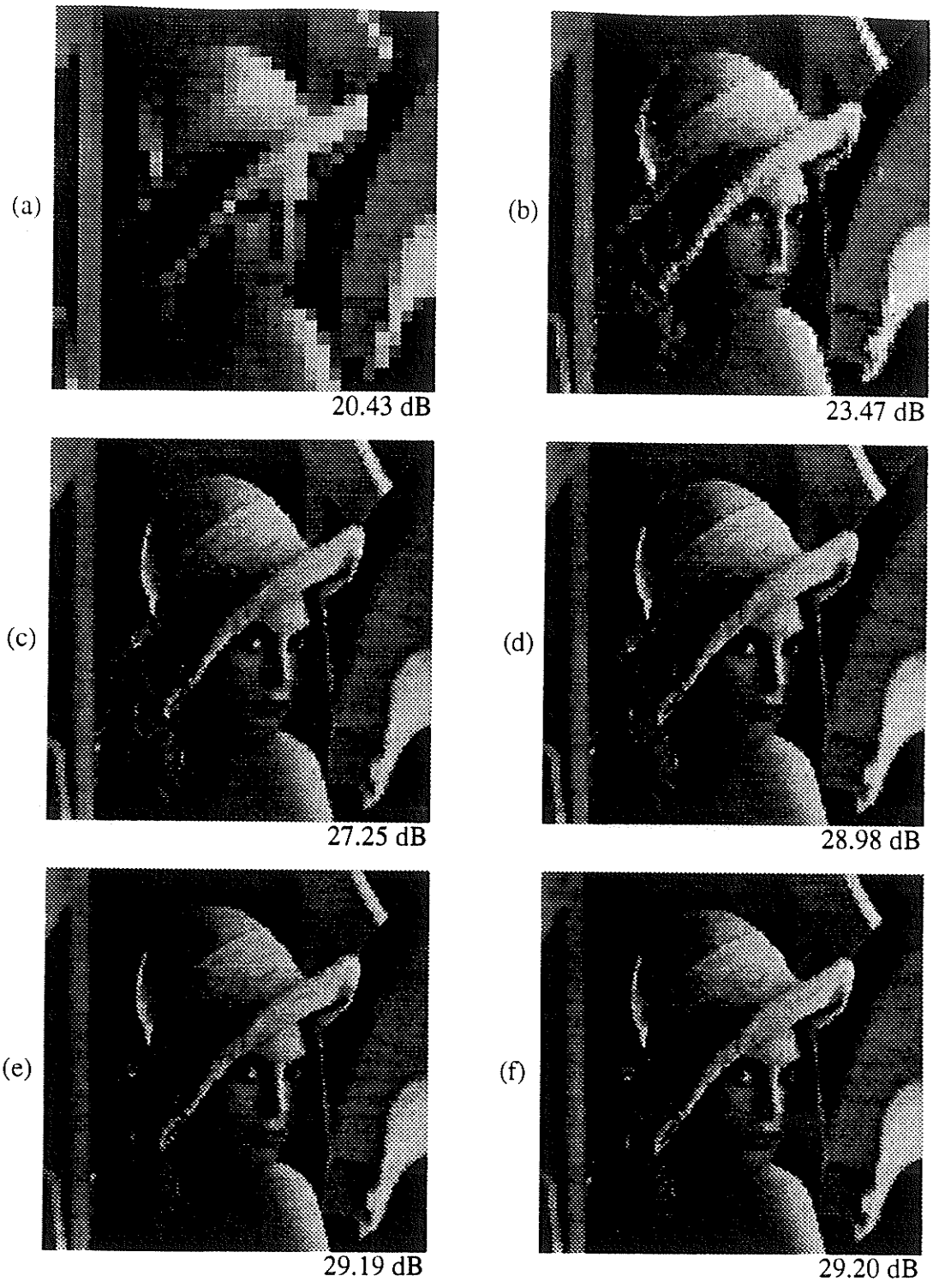


Fig. 8.3. The first six iterations of the fractal image reconstruction procedure. Images are displayed at 60% of their actual size.

lie on or near the border of a category may be placed in an inappropriate category by the vector quantizer. If this is the case, the domain block chosen to represent the range block will be from a suboptimal subset of the domain pool. Despite this, for the images coded, the 0.16 dB loss in quality was not detectable by the human eye.

Besides block misclassification, one of the primary concerns with the reduced search FBC encoder is *generalization*. The codebook prototypes used to code the *Lena* image in the above examples were learned using the FSCL neural network. The image vectors used to train this network were derived from the original *Lena* image. The resulting 11 codebook prototypes are shown in Fig. 8.4. Because these prototypes were derived from *Lena*, the question arises as to how applicable they are to other images. Fortunately, due to the small size of the codebook the neural network was forced to learn very general features. Specifically, the network identified gradients, simple edges, and double edges or stripes, as important features in the image. The network was then trained on a second image *airplane* shown in Fig. 8.5. The codebook derived form this image was used in the reduced search FBC procedure to compress *Lena*, resulting in a PSNR of 29.19 dB. A reduction in PSNR of this magnitude (0.03 dB) is insignificant in image processing applications.

8.1.2 Subjective Analysis

Subjectively while some blocking artifacts were visible, areas of relatively uniform intensity and sharp edges were well preserved by FBC. Blocking was most visible in areas of high complexity such as the feathers in *Lena*'s hat. One bothersome anomaly was FBC's apparent inability to code *Lena's* eyes adequately. This shortcoming reoccurred on a number of test images and resulted from a lack of adequate source features at larger scales.

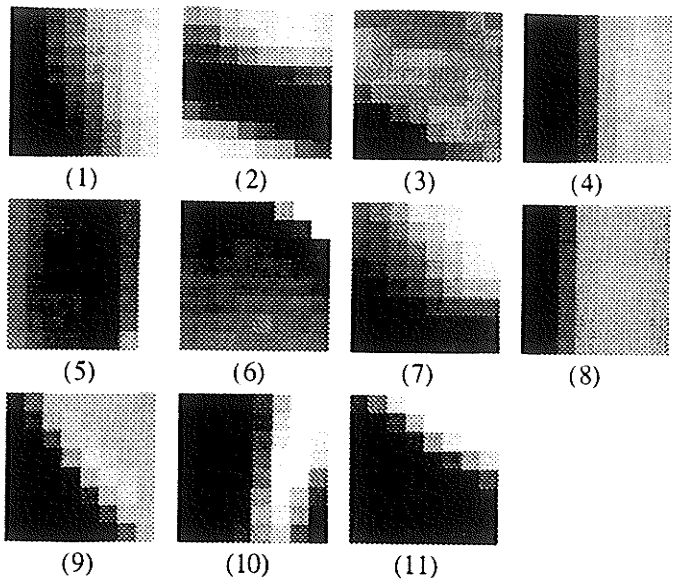


Fig. 8.4. The 11 prototypes in the scaling, translation, and isometric configuration independent VQ codebook as learned by the frequency sensitive competitive learning (FSCL) neural network from the image Lena.

Original Image

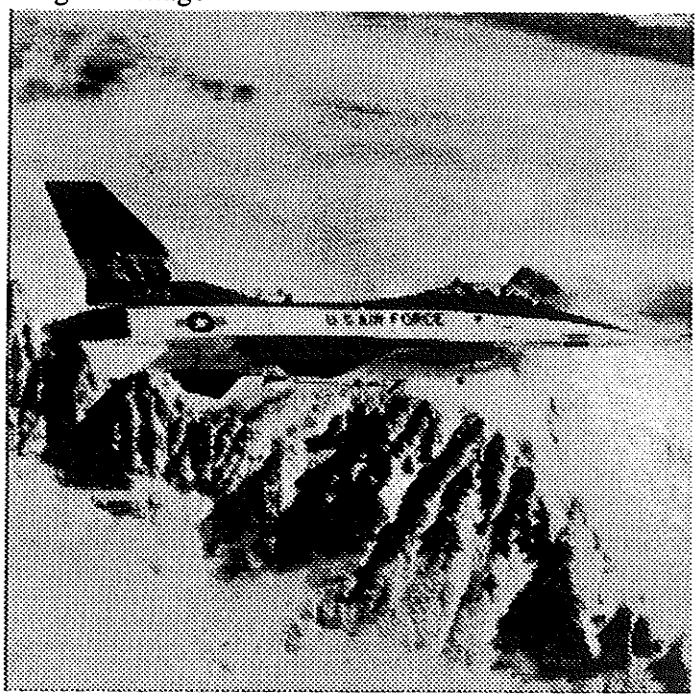
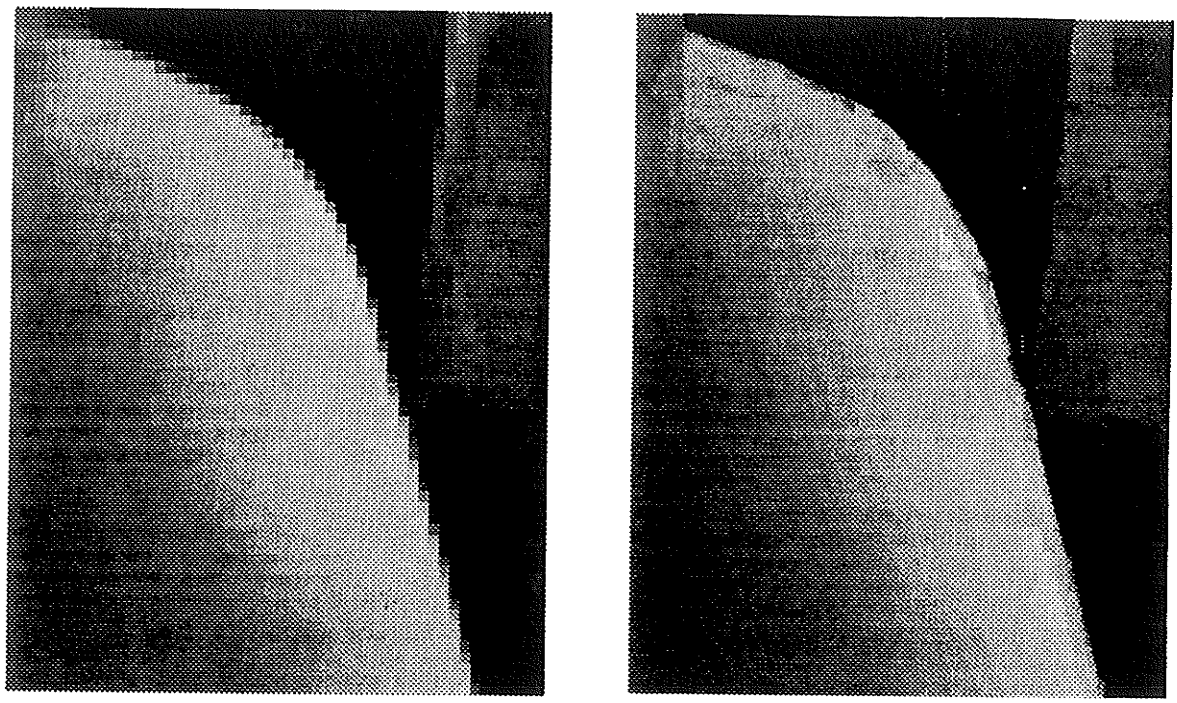


Fig. 8.5. The training image *airplane* used to develop the VQ codebook for the reduced search FBC encoding procedure.

To illustrate fractal coding's ability to interpolate along edges without exhibiting the 'staircase effect', Fig. 8.6 shows Lena's shoulder enlarged by a factor of four in both dimensions. Figure 8.6a shows an enlargement of the shoulder taken directly from the original *Lena* image. Figure 8.6b shows the fractal coded image reconstructed at four times its original size using the iterative reconstruction procedure.



(a) (b)

Fig. 8.6. A portion of Lena's shoulder enlarged to four times its original size. (a) Taken from the original 256x256 eight bpp image. (b) Fractal reconstruction of Lena's shoulder at four times its encoded size.

The reduced search procedure was also applied to a 512x512 version of the same image but this time with 8x8 range and 24x24 domain blocks. Domain blocks were chosen from a reduced domain pool of 24x24 non-overlapping range blocks. Only 10 bits were

required to address a single range block in this pool resulting in a compression ratio of 15.5:1 or 0.52 bpp before entropy coding. With entropy coding, compression increased by 19% to 18.5:1 or 0.43 bpp. The combined reduction in domain pool size and block classification scheme reduced compression time to a manageable 78 seconds. This is comparable to compression times for straight vector quantization techniques. Despite the reduced domain pool, the PSNR for the reconstructed 512x512 image (Fig. 8.7) was 31.00 dB. The improvement in reconstruction quality can be attributed to three factors. First, the 512x512 image had larger areas of uniformity and therefore lower *ac* energy per unit area making it easier to code. Second, more self-similarity existed between the 24x24 domain and 8x8 range blocks than between 16x16 domain and 8x8 range blocks. Third, for 24x24 domain blocks and 8x8 range blocks, the ratio $\frac{d}{r}$ is greater than for 16x16 domain blocks. This tends to improve the overall contraction factor of the fractal code [OiLR91]. The reduced search procedure was repeated with 24x24 overlapping domain blocks on 256x256 images but the net result was a 0.1 dB reduction in image fidelity from earlier experiments.

8.2 FBC Versus Transform Coding and Vector Quantization

As a means of benchmarking FBC, two other commonly employed image compression schemes were used to compress the *Lena* image. Figures 8.8 and 8.9 show *Lena* compressed with the proposed JPEG (Joint Photographic Experts Group) standard [Wall91] and straight *vector quantization* (VQ) [TrMe90], respectively.

The proposed JPEG standard outlines a number of requirements for the compression of continuous tone still images. The method which seems to best satisfy these requirements, in terms of subjective image quality, is a form of adaptive transform coding

Reduced Search Fractal Block Coding Image

18.5:1 (0.43 bpp) PSNR: 31.00 dB

Fig.8.7. Fractal reconstruction of the 512x512 version of *Lena* compressed by 18.5:1 at 0.43 bpp and 31.00 dB using reduced search FBC with a reduced domain pool. Image is cropped to 420x512 for display purposes.

JPEG Image

14.4:1 (0.56 bpp) PSNR: 30.70 dB

Fig. 8.8. Lena compressed by 14.4:1 at 0.56 bpp and 30.07 dB using JPEG.

based on the *discrete cosine transform* (DCT). The implementation of JPEG included within the freeware image processing package *XV* [Brad92] was used to perform the JPEG coding experiments. At compression ratios comparable to those used in the FBC test the DCT based JPEG compression technique performed very well. Quantitatively, DCT-JPEG resulted in the highest PSNR (30.07 dB) of the three techniques tested. Subjectively, a certain 'blotchiness' was observed in areas of slowly changing contrast such as the shadows on *Lena's* shoulder and cheek, or the image background. In addition areas of high contrast, such as the border of *Lena*'s shoulder, or the top of her hat were reconstructed less accurately than with FBC. In particular, a 'ringing' effect was visible along sharp edges. This ringing manifested itself in terms of a faint duplication of the edge adjacent proper edges. Despite this, JPEG was superior to FBC when it came

Vector Quantization Image

14.2:1 (0.56 bpp) PSNR: 29.39 dB

Fig. 8.9. Lena compressed by 14.2:1 at 0.56 bpp and 29.39 dB using *vector quantization* (VQ) based on *frequency sensitive competitive learning* (FSCL).

to representing areas of fine detail like the feathers in *Lena's* hat, the bridge of her nose, and her eyes.

Although superior to FBC in terms of PSNR (29.39 dB), subjectively, straight vector quantization was the worst of the three schemes tested. The implementation used, was based on the simple vector quantizer illustrated in Fig. 6.1. In this scheme, images were divided into 4x4 vectors and presented to the vector quantizer. The codebook was learned using the FSCL neural network algorithm described in Section 6.2. The resulting image displayed the same 'blotchiness' present in the JPEG image and edges were very jagged and fuzzy. VQ reproduced complex areas such as the feathers better than FBC but

the overall effect was less pleasing.

8.3 Summary

The experimental results presented in this chapter have demonstrated that the reduced search fractal block coding procedure employing frequency sensitive learning results in a considerable time saving over the corresponding exhaustive search procedure. Furthermore, this saving is achieved without significantly effecting reconstruction quality. The effectiveness of concatenated compression schemes was also established by illustrating an improvement of compression ratios by up to 20%.

Although objectively, FBC performed poorer than both of the other techniques tested (FBC: 29.22 dB, VQ: 29.39 dB, and JPEG: 30.70 dB), in subjective tests the reconstructed image was of higher quality than images resulting from straight vector quantization. This is consistent with the observations of other researchers [OiLR91] and stems from FBC's ability to preserve sharp edges. Visually, edges are extremely significant features [Scha89] which simple vector quantizers are poor at preserving [RaGe86]. The discrepancy between objective and subjective assessments of image quality for FBC and VQ also illustrates that PSNR is not an absolute measure of image quality and thereby establishes the validity of subjective observations.

In some respects, notably the reconstruction of sharp edges, FBC also outperformed the DCT based JPEG implementation. Nevertheless, based on an overall subjective evaluation, DCT-JPEG must be considered superior to the current FBC implementation. However, it should be pointed out that DCT-JPEG has resulted from the combined efforts of many researchers over an extended period of time. The limited number

of researchers currently involved in fractal coding believe that, with comparable effort, FBC will achieve reconstruction quality equivalent or even superior to DCT-JPEG [Beau91].

Even if FBC can only be elevated to the point at which coding quality is comparable to DCT-JPEG, images compressed using DCT coding can not be reconstructed at larger than their encoded sizes without introducing very visible blocking artifacts and 'straircasing'. For applications such as digital television and remote sensing, reconstruction at higher than coded resolutions may be a very attractive feature. Much to the chagrin of those owning large and expensive television sets, it has often been observed that televisions with smaller pictures tubes (14"-20") yield better picture quality than larger models (34" and above). This is due in part to the fact that the NTSC broadcast standard is fixed at 525 scanlines which can become quite noticeable when observed on large picture tubes. Using fractal encoding schemes, NTSC images could be compressed, transmitted, and then reconstructed at 1050 scanlines on larger televisions. In applications such as remote sensing, FBC's ability to interpolate along sharp edges makes it a natural form of image enhancement. An image or portion thereof could be fractally encoded and then reconstructed at a larger size, allowing small details to be examined more closely. For such applications, the focus of the coding procedure would shift from efficient towards more accurate representation of the source image.

Before concluding this chapter it should be pointed out that both FBC and reduced search FBC do have certain fundamental limitations. The success of FBC techniques in general depends on the encoders ability to locate self-affinity at different scales in an image while generating an eventually contractive function. For non-trivial images, like the photograph of Lena, the encoder is able to adequately satisfy these requirements (although

for some features, like the eyes or feathers in Lena's hat, only marginally sufficient source features could be located). However, the technique may fail completely for trivial images such as simple computer generated graphics. These images often contain objects like circles which do not possess self-affinity and are better represented using traditional Euclidian geometry. For these images, it may be impossible to either locate adequate source features or maintain contractivity. Of course, FBC is a compression scheme intended for digital photographs, not computer graphics for which very compact representation schemes already exist.

The reduced search FBC algorithm has additional ramifications which may lead to the failure of the FBC encoding procedure. Although steps were taken to prevent range blocks from being assigned to a category for which no domain blocks have been allocated, there is no guarantee that this will not in fact occur. However, this situation was never encountered for any of the test photographs and, once again, it is unlikely to occur for non-trivial images.

Despite these limitations, applications in digital television, remote sensing, and other areas, combined with the reduced search FBC coding results presented in this chapter, warrant the continued investigation of FBC and similar collage coding techniques.

CHAPTER IX

CONCLUSIONS AND RECOMMENDATIONS

The work described in this thesis was motivated by the need for new and better image compression techniques. Signal compression using *fractals* represents an emerging area of *lossy* data compression methods which have been applied successfully to digital images. A study of fractals, the *collage theorem*, and *fractal block coding* (FBC) has led to an implementation of a *concatenated* image compression scheme using FBC, *arithmetic coding*, and *neural networks*.

Since the enormous storage and transmission requirements of digital images can not be substantially reduced using traditional *lossless* data compression methods, *lossy* approaches are used. For still images these may result in compression ratios as high as 30:1 but will introduce some distortion into the reconstructed image. It is the objective of lossy compression schemes to maximize compression ratios while minimizing this distortion.

Fractals have been proposed as the basis for good lossy compression techniques. Fractals appear to be well suited to image compression because many objects in nature exhibit fractal geometry and fractals often posses *self-similarity* or *self-affinity* at different scales. This similarity is redundancy and can therefore be removed resulting in data compression.

Existing fractal compression techniques are based on a corollary of *contractive transformation theory* called the *collage theorem.* The collage theorem implies that if an image can be described approximately by a *contractive function* of itself, then it can be reconstructed approximately from that description using an iterative procedure. Coding techniques based on contractive transformation theory and the collage theorem are called *collage coding* techniques. A collage encoding algorithm attempts to represent a source image as a function of itself. This function must be a contraction, produce an adequate and compact representation of the original image, and have an associated systematic encoding procedure.

The fundamental difficulty associated with collage coding techniques is the computational complexity of the encoding procedures. *Iterated function systems* (IFSs), the first proposed collage coding technique provides very compact representations of complex self-affine images. Unfortunately, locating the appropriate IFS coefficients for a particular image is an *NP complete* problem. A less compact compression technique known fractal block coding, operates in known polynomial time by employing a *divide-and-conquer* compression strategy. Using this strategy, both the the source image and the fractal transformation are segmented into more manageable components for which the appropriate fractal parameters can be located systematically. Unfortunately, using even this systematic approach the generalized form of the FBC encoding procedure is still $O(n^4)$. A number of heuristic approaches have been described for reducing FBC encoding time but none of these have dealt specifically with the $O(n^4)$ computational complexity of the encoding procedure.

A reduced search coding procedure based on subimage classification using neural networks provides an alternative to heuristic time-saving approaches, and reduces the

computational complexity of the encoding procedure to $O(n^3)$. In this scheme, domain and range blocks are pre-classified independent of scaling, translation, or isometric configuration using a *frequency sensitive competitive learning* (FSCL) neural network. During the encoding procedure, a domain block is only considered as a possible source for a range block if they are both of the same type. FSCL was selected as an appropriate neural network because it results in equal prototype utilization which is critical in the derivation of compression times for the reduced search algorithm. For 256x256 images, encoding time was reduced by a factor of 45 with further image degradation of less than 0.2 dB.

Reduced search FBC was implemented as the *inner* code of a concatenated FBC/arithmetic image compression scheme. FBC can compress images by up to 16:1 but some statistical redundancy may remain in the fractal code. Lossless arithmetic entropy coding can remove this redundancy and thereby further increase compression ratios by up to 20%. The concatenated compression scheme was capable of compressing grey scale images at ratios in excess of 18:1 with a PSNR of 31.00 dB. The ability of the iterative reconstruction procedure to interpolate along sharp edges in these images also suggests that FBC may have applications in image enhancement.

This thesis has contributed to general and technical knowledge through

(1) study and understanding of collage coding techniques,

(2) the implementation of exhaustive search FBC encoding and decoding algorithms which can serve as both starting points and benchmarks for future FBC developments,

(3) the development and implementation of a reduced search FBC encoding procedure using neural networks to improve on the computational complexity of the exhaustive search encoding procedure, and

(4) an implementation of FBC within a concatenated FBC/arithmetic image compression system, thereby establishing experimentally the effectiveness of concatenated coding.

Recommendations for future work related to this thesis include:

(1) The current implementation relies heavily on floating point calculations. These could be replaced by fixed point calculations using integers to improve performance on computers without floating point processors (eg., 386, 486 SX).

(2) The iterative decoding procedure could be replaced by a random decoding procedure similar to the RIA for IFSs. This procedure would select fractal block transforms at random from the fractal code and apply them one at a time to portions of the same image. This would decrease the memory requirements of the implementation by almost 50%.

(3) The possibility of combining other time, quality, and compression ratio improving techniques with the systematic reduced search procedure with neural networks should be investigated. Specifically,

- the improved translation functions of Oien *et. al* [OiLR91] could be incorporated,
- distortion measures based on the Hadamard transform could be implemented to improve the subjective quality of the reconstructed images [Beau91], and
- the scaling and translation paramters could be quantized to further improve compression ratios using a non-linear scalar or vector quantization scheme. The appropriate quantization step sizes would be derived from extensive objective and subjective experimentation.

(4) A complete investigation into FBC as an image enhancement technique should be performed.

(5) Monro and Dudbridge [MoDu92] have recently proposed their own

collage coding procedure for grey scale images which does not possess the search associated with Jacquin's technique. The authors claim that their technique is less computationally intensive than even the *fast discrete cosine transform* [ChSF77] although it does result in images of lower quality. Nevertheless they are also optimistic that this new fractal technique may achieve performance levels equal to JPEG. It remains to be seen whether or not this new technique also retains the advantageous properties associated with FBC such as superior edge reconstruction.

REFERENCES

[AKCM90] S.C. Ahalt, A.K. Krishnamurthy, P.Chen, and D.E. Melton, "Competitive learning algorithms for vector quantization," *Neural Networks*, Vol. 3, No. 3, pp. 277-290, 1990.

[Barn88] M.F. Barnsley, *Fractals Everywhere*. New York, NY: Academic Press, 1988, 396 pp.

[Barn90] M.F. Barnsley, "Data compression using the fractal transform," *Image Processing 90 - The Key Issues. Conference Proceedings*, Vol. M1, pp. 1-10, 1990.

[BaSl88] M.F. Barnsley and A.D. Sloan, "A better way to compress images," *Byte*, Vol. 13, pp.215-223, January 1988.

[Bill61] P. Billingsley, "On the coding theorem for the noiseless channel," *Ann. Math. Statist.*, Vol. 32, No. 2, pp. 594-601, May 1961.

[Beau91] J.M. Beaumont, "Image data compression using fractal techniques," *British Telecom Tech. Journal*, Vol. 9, No. 4, pp 93-109, 1991.

[Brad92] J. Bradley, *XV version 2.21*. University of Pennsylvania: PA, 1992.

[ChSF77] W.H. Chen, C.H. Smith, and S.C. Fralick, "A fast computational algorithm for the discrete cosine transform," *IEEE Trans. Commun.*, Vol. 25, No. 9, pp. 1004-1009, September 1977.

[Davi91] L. Davis, *Handbook of Genetic Algorithms*. New York, NY: Von Nostrand Rienhold, 1991, 385 pp.

[Delo92] D.G. Delorie, *jdgpp*, Rochester , NH: barnacle.erc.clarkson.edu, 1992

[DuKi91] D. Dueck and W. Kinsner, "Experimental study of Shannon-Fano, Huffman, Lempe-Ziv-Welch and other lossless algorithms," *Proc. 10th Computer Networking Conf.*, (San Jose, CA; Sept. 29-30, 1991), pp. 23-31, 1991.

[DeSi88] D. DeSieno, "Adding a conscience to competitive learning," *Proc. IEEE Intern. Conf. Neural Networks*, Vol. 1, pp. 117-124, 1988.

[Fran69] L.E. Franks, *Signal Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1969, 316 pp.

[FrDu90] G.C. Freeland and T.S. Durrani, "IFS fractals and the wavelet transform," *Proc. IEEE Intern. Conf. Acoustics, Speech & Sign. Processing; ICASSP90* (Albuquerque, NM; Apr. 3-6, 1990), IEEE Cat. No. 90CH2847-2, Vol. 4, pp. 2345-2348, 1990.

[HeKP91] J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computing*. Redwood City, CA: Addison-Wesley, 1991, 327 pp.

[Huff52] D.A. Huffman, "A method for constructing minimum-redundancy codes," *Proc. IRE*, Vol. 40, pp. 1098-1101, September 1952.

[Hutc81] J. Hutchinson, "Fractals and self-similarity", *Indiana University Journ. of Math.*, Vol. 30, pp. 713-747, 1981.

[Jacq89] A.E. Jacquin, "A fractal theory of iterated Markov operators with applications to digital image coding," *Ph.D. Dissertation*, Georgia Tech., 1989.

[Jacq90a] A.E. Jacquin, "A novel fractal block coding technique for digital images," *Proc. IEEE Intern. Conf. Acoustics, Speech & Sign. Processing; ICASSP90* (Albuquerque, NM; Apr. 3-6, 1990), IEEE Cat. No. 90CH2847-2, Vol. 4, pp. 2225-2228, 1990.

[Jacq90b] A.E. Jacquin, "Fractal image coding based on a theory of iterated contractive image transformations," *SPIE Vol. 1360 Visual Communications and Image Processing '90*, pp. 227- 239, 1990.

[Jacq92] A.E. Jacquin, "Image coding based on a fractal theory of iterated contractive image transformations," *IEEE Trans. Image Processing*, Vol. 1, No. 1, pp. 18-30, January 1992.

[Jean90] J.S.N. Jean, "A new distance measure for binary images," *Proc. IEEE Intern. Conf. Acoustics, Speech & Sign. Processing; ICASSP90* (Albuquerque, NM; Apr. 3-6, 1990), IEEE Cat. No. 90CH2847-2, Vol. 4, pp. 2061-2064, 1990.

[KaRo90] L. Kaufman and P.J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. New York, NY: John Wiley & Sons, 1990, 342 pp.

[Kins72] W. Kinsner, "Peano, Sierpinski, and Hilbert space-filling curves," *Unpublished (graphics)*; Hamilton, ON: Department of Electrical Engineering, McMaster University 1972.

[Kins91] W. Kinsner, "Review of data compression methods, including Shannon-Fano, Huffman, arithmetic, Storer, Lempel-Ziv-Welch, fractal, neural network, and wavelet algorithms," *Technical Report*, DEL91-1, Winnipeg, MB: Dept. of Electrical and Computer Engineering, University of Manitoba, January 1991, 157 pp.

[LaAa87] P.J.M. van Laarhoven and E.H.L. Aarts, *Simulated Annealing: Theory and Applications*. Dordrecht, Holland: D. Reidel Publishing, 1987, 186 pp.

[LiGB80] Y. Linde, A. Buzo, and R.M. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Commun.*, Vol. 28, pp.84-95, November 1980.

[Mall89] S.G. Mallat, "A theory of multiresolutional signal decomposition: The wavelet representation," *IEEE Trans. Pattern Analysis Machine Intelligence*, Vol. 11, pp.764-693, July 1989.

[Mand67] B.B. Mandelbrot, "How long is the coast of Britain? Statistical self-similarity and fractional dimension," *Science*, Vol. 156, pp. 636-638, 5 May 1967.

[Mand83] B.B. Mandelbrot, *The Fractal Geometry of Nature*. New York, NY: W.H. Freeman and Co., 1983, 468 pp.

[McAR90] J.D. McAuliffe, L.E. Atlas, and C. Rivera, "A comparison of the LBG alogrithm and Kohonen neural network paridigm for image vector quantization," *Proc. IEEE Intern. Conf. Acoustics, Speech & Sign. Processing; ICASSP90* (Albuquerque, NM; Apr. 3-6, 1990), IEEE Cat. No. 90CH2847-2, Vol. 4, pp. 2293-2296, 1990.

[McRu88] J.L. McClelland and D.E. Rumelhart, *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises*. Cambridge, MA: MIT Press, 1988, 344 pp.

[MoDu92] D.M. Munro and F. Dudbridge, "Fractal block coding of images," *Electronics Letters*, Vol. 28, No. 11, pp. 1053-1055, 21 May 1992.

[MoHS90] B. Moghaddam, K.J. Hintz, and C.V. Stewart, "Fractal image compression and texture analysis," *SPIE Vol. 1406 Image Understanding in the '90s: Building Systems that Work*, pp 42-57, 1990.

[NaSe82] A.W. Naylor and G.R. Sell, *Linear Operator Theory in Engineering and Science*. New York, NY: Springer-Verlag, 1982, 624 pp.

[OiLR91] G.E. Oien, S. Lepsoy, and T.A. Ramstaad, "An inner product space approach to image coding by contractive transformations," *Proc. IEEE Intern. Conf. Acoustics, Speech & Sign. Processing; ICASSP91* (Toronto, Ont., May 14-17, 1991), Vol. 4, pp. 2773-2776, 1991.

[PeJS92] H. Peitgen, H. Jurgens, and D. Saupe, *Fractals for the Classroom*. Vol. 1, New York, NY: Springer-Verlag, 1992, 450 pp.

[Scha89] R.J. Schalkoff, *Digital Image Processing and Computer Vision: An Introduction to Theory and Implementations*. New York, NY: John Wiley & Sons, 1989, 489 pp.

[Shan49] C.E. Shannon, *The Mathematical Theory of Communication*, Urbana, Ill: University of Illinois Press, 1949, 117 pp.

[Shon89] R. Shonkwiller, "An image algorithm for computing Hausdorff distance efficiently and in linear time," *Info. Proc. Lett.*, Vol. 30, pp. 87-89, 1989.

[Swed72] "Swedish Accent," *Playboy*, Vol. 19, No. 11, pp. 134-141, November 1972.

[RaGe86] R. Ramamurthi and A.Gersho, "Classified vector quantization of images," *IEEE Trans. Commun.*, Vol. 34, pp. 1105-1115, November 1986.

[RuMc86] D.E. Rumelhart and J.L. McClelland, *Parallel Distributed Processing 1: Foundations*. Cambridge, MA: MIT Press, 1986, 547 pp.

[Vela91] S.A. Velastin, "An approach to modular programming in C," *Third International Conf. on Software Engineering for Real Time Systems* (Cirencester, UK; Sept. 16-18, 1991), pp. 227-232, 1991.

[WaFK93] L. Wall, K. Ferens, and W. Kinsner, "Real-time dynamic arithmetic coding for low bit rate channels," to appear in*Western Canada Conf. Computers, Power, and Communication Systems, WESCANEX '93* (Saskatoon, SK; May 17-18, 1993).

[Wall91] G.K. Wallace, "The JPEG Still Image Compression Standard," *Comm. ACM*, Vol. 34, No. 4, pp. 30-44, April 1991.

[Welc84] T.A. Welch, "A technique for high-performance data compression," *IEEE Computer*, Vol. 17, pp. 8-19, June 1984.

[WiNC87] I.H. Witten, R.M. Neal, J.G. Cleary, "Arithmetic Coding for Data Compression," *Comm. ACM*, Vol. 30, No. 6, pp. 520-540, June 1987.

APPENDIX A

C LANGUAGE LISTING FOR IFS SYNTHESIS SOFTWARE

```
/*============================================================

Program:      Random Iterartion (RIA) and Multiple Recuction Copy (MRCA)
              Algorithms for IFS Synthesis.

Programmer:   Larry M. Wall
              Department of Electrical and Computer Engineering
              University of Manitoba
              Winnipeg, Canada
              larwall@ee.umanitoba.ca

Version:      1.1

Last Update:  01/12/92                                              */

/*------------------------------------------------------------------

LIBRARIES:                                                          */

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

/*------------------------------------------------------------------

CONSTANTS:                                                          */

#define IMG_SZ      256   /* size of fractal image in pixels       */
#define IMG_SCL     1.0   /* scaling factor of fractal image       */
#define IFS_SZ      6     /* number of data fields in each IFS record */
#define NM_SZ       20    /* maximum filename size                 */

#define SUCCESS     1
#define FAILURE     0

/*------------------------------------------------------------------

FUNCTIONS:                                                          */

unsigned char *img_alloc();
int   img_save();
int   ifs_load();
      void   ifs_alloc();
      void   ifs_free();
      void   calc_prob();
```

```
void  img_box();
void  img_L();
void  img_neg();
void   img_flp();

void  ifs_RIA();
void  ifs_MRCA();
        double  collage();

void  img_box();
void  img_L();
void  img_neg();
void   img_flp();
void  print_code();

/*-------------------------------------------------------------

GLOBAL DECLARATIONS:                                          */

typedef  struct {
        int     l;
        float   *tm[IFS_SZ],
                *p;
}  ifs;

/*=============================================================

MAIN PROGRAM:                                                 */

main()
{

        int     t;
        time_t lt;
        struct tm
                *started,
                *finished;

        unsigned char *img_1,
                      *img_2;

        ifs     ifs_1;

        char    choice;

        char     IFSFile[NM_SZ],
                 ImgFile[NM_SZ];
```

```
        printf("\nEnter IFS Filename: ");
        scanf( "%s", IFSFile );
        puts( "Loading IFS Code." );
        ifs_load( IFSFile, &ifs_1 );
        img_1 = img_alloc();

        printf("\n[r] Random Iteration Algorithm\n");
        printf("[m] Multiple Reduction Copy Algorithm\n");
        printf("\nSelect Function: ");
        scanf( "\n%c", &choice );
        switch( choice )
        {
                case 'r':
                        ifs_RIA( img_1, ifs_1 );
                        break;
                case 'm':
                        ifs_MRCA( img_1, ifs_1 );
                        break;
                default:
                        exit();
        }

        printf("\nEnter Destination Filename: ");
        scanf( "%s", ImgFile );
        img_box( img_1 );
        img_neg( img_1 );
        img_2 = img_alloc();
        img_flp( img_1, img_2 );
        img_save( ImgFile, img_2 );
        free( img_1 );
        free( img_2 );
        ifs_free( &ifs_1 );

}

/*============================================================

: Dynamically allocate memory for image array                    * /

unsigned char *img_alloc()
{
        int x;

        unsigned char *ptr;
```

```
        ptr = (unsigned char *) calloc( IMG_SZ*IMG_SZ, sizeof(unsigned char) );
        if ( !ptr )
        {
                puts( "Memory Allocation Error." );
                exit(0);
        }

        for( x=0; x<IMG_SZ*IMG_SZ; x++ )
                ptr[x] = 0;
        return  ptr;
}

/*-------------------------------------------------------------

: Load IFS code                                                  */

int ifs_load( filename,  code )
char    filename[NM_SZ];
ifs     *code;
{

        int     status,
                record,
        field,
        data;
        FILE    *InFile;
        float   *ptr;

        InFile = fopen( filename, "rt" );
        if ( InFile != NULL )
        {
                status = fscanf(  InFile, "%d\n", &(code->I) );
                if ( ( status == 1 ) && ( code->I != 0 ) )
                {
                        status = 0;
                        ifs_alloc( code );
                        puts( "Allocated" );

                        for ( record=0; record<code->I; record++ )
                        {
                                for ( field=0; field<IFS_SZ; field++ )
                                        if ( !feof(InFile) )
                                        {
                                                status += fscanf( InFile, "%d",  &data );
                                                code->tm[field][record] = (float) data /
100;
                                        }
                                fscanf( InFile, "\n" );
```

```
                        }
                        if ( status != ( code->l * IFS_SZ) )
                        {
                                puts("File Format Error.");
                                code->l = 0;
                        }
                        else
                        {
                                puts( "Loaded." );
                                calc_prob( code );
                                puts( "Calculated." );
                        }
                }
                else
                {
                        puts( "File Format Error." );
                        code->l = 0;
                }
                fclose( InFile );
        }
        else
        {
                puts( "File Not Found." );
                code->l = 0;
        }

        return code->l;

}

/*-------------------------------------------------------------

: Dynamically allocate memory for IFS code                      */

void ifs_alloc( code )
ifs *code;
{
        int     t;

        for( t=0; t<IFS_SZ; t++ )
                code->tm[t] = (float *) calloc( code->l, sizeof(float) );
        code->p = (float *) calloc( code->l, sizeof(float) );

}

/*-------------------------------------------------------------

: Free memory dynamically allocated for IFS code                */
```

```
void ifs_free( code )
ifs *code;
{

        int     t;

        for( t=0; t<IFS_SZ; t++ )
                free( code->tm[t] );
        free( code->p );

}

/*-------------------------------------------------------------------

: Calculate Probabilites for IFS Code                                   */

void calc_prob( code )
ifs *code;
{

        int     t;
        float   area,
                total;

        total = 0.0;

        for ( t=0; t<code->l; t++ )
        {
                area = fabs( ( code->tm[0][t] * code->tm[3][t] ) - ( code->tm[1][t] *
code->tm[2][t] ) );
                if ( area < 0.01 )
                        area = 0.01;
                total += area;
                code->p[t] = total;
        }

        for ( t=0; t<code->l; t++ )
                code->p[t] /= total;

}


/*-------------------------------------------------------------------

: Random Iteration Algorithm (RIA) for Generating Fractal Images from IFS Code */
```

```
void ifs_RIA( image, code )
unsigned char image[IMG_SZ][IMG_SZ];
ifs     code;
{
        int     img_x,
                img_y,
                t;
        unsigned long int itr;
        float   rnd_nm,
                x,
                y,
                new_x,
                new_y,
                x_scl,
                y_scl;

        int     num_itr;

        printf( "\nEnter Number of Iterations: " );
        scanf( "%6d", &num_itr);

        x_scl = y_scl = IMG_SCL;
        for ( img_x=0; img_x<IMG_SZ; img_x++ )
                for ( img_y=0; img_y<IMG_SZ; img_y++ )
                        image[img_x][img_y] = 0;

        x = y = 0.0;
        for ( itr=0; itr<num_itr; itr++ )
        {
                rnd_nm = (float) random() / MAXLONG;
                t = 0;
                while ( rnd_nm > code.p[t] )
                        t++;
                new_x = code.tm[0][t] * x + code.tm[1][t] * y + code.tm[4][t] *
IMG_SZ;
                new_y = code.tm[2][t] * x + code.tm[3][t] * y + code.tm[5][t] *
IMG_SZ;
                x = new_x;
                y = new_y;
                if ( itr>10 )
                {
                        img_x = x + 0.5/* IMG_SZ * IMG_SCL*/;
                        img_y = y + 0.5/* IMG_SZ * IMG_SCL*/;
                        if ( (img_x>0) && (img_x<IMG_SZ) && (img_y>0) &&
(img_y<IMG_SZ) )
                                image[img_y][img_x] = 255;
                }
        }
```

```
}

/*-------------------------------------------------------------

: Multiple Reduction Copy Algorithm (RIA) for Generating Fractal Images from IFS
        Code                                                                  */

void ifs_MRCA( image, code )
unsigned char image[IMG_SZ][IMG_SZ];
ifs     code;
{
        unsigned char *img_temp;

        int     num_itr,
                count;

        printf( "\nEnter Number of Iterations: " );
        scanf( "%6d", &num_itr);

        img_temp = img_alloc();
        img_box( image );
        for( count=0; count<(num_itr/2); count++ )
        {
                collage( image, img_temp, code );
                collage( img_temp, image, code );
        }

        free( img_temp );

}

/*-------------------------------------------------------------

: Create collage image from IFS code                                  */

double collage( s_image, c_image, code )
unsigned char s_image[IMG_SZ][IMG_SZ];
unsigned char c_image[IMG_SZ][IMG_SZ];
ifs code;
{
        int     t,
                img_x,
                img_y,
                col_x,
                col_y;
```

```
        double error;

        error = 0.0;

        for ( col_x=0; col_x<IMG_SZ; col_x++ )
                for ( col_y=0; col_y<IMG_SZ; col_y++ )
                        c_image[col_y][col_x] = 0;

        for ( t=0; t<code.l; t++ )
                for ( img_x=0; img_x<IMG_SZ; img_x++ )
                        for ( img_y=0; img_y<IMG_SZ; img_y++ )
                                if ( s_image[img_y][img_x] != 0 )
                                {
                                        col_x =  code.tm[0][t] * img_x + code.tm[1][t] *
img_y + code.tm[4][t]*IMG_SZ;
                                        col_y =  code.tm[2][t] * img_x + code.tm[3][t] *
img_y + code.tm[5][t]*IMG_SZ;
                                        if  (  (col_x>=0)  &&  (col_x<IMG_SZ)  &&
(col_y>=0) && (col_y<IMG_SZ) )
                                        {
                                                if ( c_image[col_y][col_x] == 0 )
                                                        c_image[col_y][col_x]     =
s_image[img_y][img_x];
                                                else
                                                        error += OVRLAP;
                                        }
                                        else
                                                error += ((IMG_SZ*IMG_SZ)>>2);
                                }
        return  error;

}

/*-------------------------------------------------------------

: Save fractal image to disk                                                    */

int img_save( filename, image )
char filename[NM_SZ];
unsigned char image[IMG_SZ][IMG_SZ];
{

        int     status;
        FILE    *OutFile;

        OutFile = fopen( filename, "wb" );
        if ( OutFile != NULL )
        {
```

```
                if ( fwrite( image, sizeof(unsigned char), IMG_SZ * IMG_SZ, OutFile ) )
                        status = SUCCESS;
                else
                {
                        puts( "File Write Error." );
                        status = FAILURE;
                }
                fclose( OutFile );
        }
        else
        {
                puts( "Cannot Create File." );
                status = FAILURE;
        }

        return status;

}

/*============================================================*/

void print_code( code )
ifs code;
{
        int c;

        for ( c=0; c<code.l; c++ )
                printf( "%f %f %f %f %f %f %f\n", code.tm[0][c], code.tm[1][c],
code.tm[2][c], code.tm[3][c], code.tm[4][c], code.tm[5][c], code.p[c] );

        printf( "\n" );
}


/*------------------------------------------------------------

: Generate a box as the starting image for the MRCA.                    */

void img_box( image )
unsigned char image[IMG_SZ][IMG_SZ];
{
        int     x,
                y;


        for( x=0; x<IMG_SZ; x++ )
        {
                image[x][0] = 255;
```

```
                image[0][x] = 255;
                image[x][IMG_SZ-1] = 255;
                image[IMG_SZ-1][x] = 255;
        }

}

/*-------------------------------------------------------------

: Place an L in the top left corner of the image for diplaying self-affine portions. */

void img_L( image )
unsigned char image[IMG_SZ][IMG_SZ];
{
        int     x,
                y;

        for( x=0; x<(IMG_SZ*0.1); x++ )
        {
                image[(int)(IMG_SZ*0.95)-x][(int)(IMG_SZ*0.05)] = 255;
                image[(int)(IMG_SZ*0.85)][(int)(IMG_SZ*0.05)+x] = 255;
        }
}

/*-------------------------------------------------------------

: Flip the image about the horizontal axis.                    */

void img_flp( s_image, c_image )
unsigned char s_image[IMG_SZ][IMG_SZ];
unsigned char c_image[IMG_SZ][IMG_SZ];
{
        int     x,
                y;

        for( x=0; x<IMG_SZ; x++ )
                for( y=0; y<IMG_SZ; y++ )
                        c_image[x][y] = s_image[IMG_SZ-1-x][y];
}

/*-------------------------------------------------------------

: Produce the negative of an image.                            */

void img_neg( image )
unsigned char image[IMG_SZ][IMG_SZ];
{
```

```
        int     x,
                y;

        for( x=0; x<IMG_SZ; x++ )
                for( y=0; y<IMG_SZ; y++ )
                        if( image[x][y] == 0 )
                                image[x][y] = 255;
                        else
                                image[x][y] = 0;
}

/*=============================================================*/
```

APPENDIX B

DEREVATION OF OPTIMAL SCALING AND TRANSLATION COEFFICIENTS FOR FBC

Equations 5.4 and 5.5 for the optimal FBC scaling and translation coefficients result from the formal definitions of *norms* and *inner products*, and their relationship to *metrics*.

Vector Spaces

Similar to metrics, norms and inner products are defined in terms *normed vector* and *inner product* spaces. While a metric space was defined as a simple set and a corresponding metric, normed vector and inner product spaces are formed from *vector spaces*. A vector space consists of a nonempty set V and two special operators. These operators, called *vector addition* and *scaler multiplication*, are denoted $\mathbf{v} + \mathbf{w}$ and $a\mathbf{v}$ respectively. Vector addition and scalar multiplication must be defined in such a way as to satisfy the following axioms:

(A1) If $\mathbf{u}$ and $\mathbf{v}$ are in V, then $\mathbf{u} + \mathbf{v}$ is in V.
(A2) $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$ for all $\mathbf{u}$ and $\mathbf{v}$ in V.
(A3) $\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$ for all $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ in V.
(A4) There exists a unique element $\mathbf{0}$ in V such that $\mathbf{v} + \mathbf{0} = \mathbf{v}$.
(A5) For each element $\mathbf{v}$ in V there exists an element $-\mathbf{v}$ such that $\mathbf{v} + -\mathbf{v} = \mathbf{0}$.
(S1) If $\mathbf{v}$ is in V, then $a\mathbf{v}$ is in V, for all real a.
(S2) $a(\mathbf{v} + \mathbf{w}) = a\mathbf{v} + a\mathbf{w}$ for all $\mathbf{v}$ and $\mathbf{w}$ in V, and real a.
(S3) $(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$ for all $\mathbf{v}$ in V, and real a and b.
(S4) $a(b\mathbf{v}) = (ab)\mathbf{v}$ for all $\mathbf{v}$ in V, and real a and b.
(S5) $1\mathbf{v} = \mathbf{v}$ for all $\mathbf{v}$ in V.

Norms and Inner Products

A norm is a real valued function $\|\bullet\|$ defined on a vector space V which satisfies the following axioms:

(N1) $\|\mathbf{v}\| \geq 0$ for all $\mathbf{v}$ in V.
(N2) $\| \mathbf{v} + \mathbf{w} \| \leq \|\mathbf{v}\| + \|\mathbf{w}\|$ for all $\mathbf{v}$ and $\mathbf{w}$ in V.
(N3) $\|a\mathbf{v}\| = |a| \, \|\mathbf{v}\|$ for all $\mathbf{v}$ in V, and real a.
(N4) $\|\mathbf{v}\| = 0$ if and only if $\mathbf{v} = \mathbf{0}$.

A normed linear space is simply a vector space V upon which a norm $\|\bullet\|$ is defined and is denoted $(V,\|\bullet\|)$. Associated with any normed vector space is a corresponding metric given by

$$d(\mathbf{v},\mathbf{w}) = \| \mathbf{v} - \mathbf{w} \| \tag{B.1}$$

Similarly, an inner product space is defined to be a vector space V together with an inner product defined on V. The inner product, denoted $\langle \mathbf{v},\mathbf{w} \rangle$, is a real valued function of two vectors $\mathbf{v}$ and $\mathbf{w}$ which satisfies the following axioms:

(P1) $\langle \mathbf{u} + \mathbf{v}, \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{w} \rangle + \langle \mathbf{v}, \mathbf{w} \rangle$ for all $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ in V.

(P2) $\langle a\mathbf{v}, \mathbf{w} \rangle = a\langle \mathbf{v}, \mathbf{w} \rangle$ for all $\mathbf{v}$ and $\mathbf{w}$ in V.

(P3) $\langle \mathbf{v}, \mathbf{w} \rangle = \overline{\langle \mathbf{w}, \mathbf{v} \rangle}$ for all $\mathbf{v}$ and $\mathbf{w}$ in V.

(P4) $\langle \mathbf{v}, \mathbf{v} \rangle > 0$ for $\mathbf{v} \neq \mathbf{0}$.

Once again, of particular interest is the fact that an inner product generates a norm. Specifically

$$\|\mathbf{v}\| = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle} \tag{B.2}$$

It follows from Eq. B.2 that if an inner product generates a norm then an inner product space must also be a normed vector space. Likewise, Eq. B.1 intimates that a normed vector space must also be a metric space. It is important to realize however that the converse of either of these statements is not necessarily true. There exist many examples of metrics which have no associated norms as well as norms for which there are no corresponding inner products.

Calculation of Optimal Grey Level Scaling and Translation Coefficients

During the FBC encoding procedure we are interested in locating the scaling and translation coefficients a and t respectively, which minimize the Euclidian distance between the contracted domain block $\mathbf{x}$ and the range block $\mathbf{y}$ given by $d_2(a\mathbf{x}+t\mathbf{u},\mathbf{y})$. Using Eq. B.1 and B.2 in conjunction with the axioms N1 through N4 and P1 through P4, d_2 can be rewritten as

$$d_2(a\,\mathbf{x}+t\,\mathbf{u},\mathbf{y}) = \|\mathbf{y}\|^2 + a^2\|\mathbf{x}\|^2 + t^2\|\mathbf{u}\|^2 - 2(a\langle \mathbf{x},\mathbf{y} \rangle + t\langle \mathbf{y},\mathbf{u} \rangle - at\langle \mathbf{x},\mathbf{u} \rangle) \tag{B.3}$$

This equation can be minimized by taking partial derivatives with respect to a and t which yields the system of two linear equations

$$\frac{\partial d_2}{\partial a} = 2a\,\|\mathbf{x}\|^2 - 2\langle \mathbf{x},\mathbf{y}\rangle + 2t\,\langle \mathbf{x},\mathbf{u}\rangle = 0 \tag{B.4}$$

and

$$\frac{\partial d_2}{\partial t} = 2t\,\|\mathbf{u}\|^2 - 2\langle \mathbf{y},\mathbf{u}\rangle + 2a\,\langle \mathbf{x},\mathbf{u}\rangle = 0 \tag{B.5}$$

Equations B.4 and B.5 can be rewritten in terms of the following system of linear equations

$$\begin{bmatrix} \|\mathbf{x}\|^2 & \langle \mathbf{x},\mathbf{u}\rangle \\ \langle \mathbf{x},\mathbf{u}\rangle & \|\mathbf{u}\|^2 \end{bmatrix} \begin{bmatrix} a \\ t \end{bmatrix} = \begin{bmatrix} \langle \mathbf{x},\mathbf{y}\rangle \\ \langle \mathbf{y},\mathbf{u}\rangle \end{bmatrix} \tag{B.6}$$

Solving this system of equations yields

$$\begin{bmatrix} a \\ t \end{bmatrix} = \begin{bmatrix} \|\mathbf{x}\|^2 & \langle \mathbf{x},\mathbf{u}\rangle \\ \langle \mathbf{x},\mathbf{u}\rangle & \|\mathbf{u}\|^2 \end{bmatrix}^{-1} \begin{bmatrix} \langle \mathbf{x},\mathbf{y}\rangle \\ \langle \mathbf{y},\mathbf{u}\rangle \end{bmatrix}$$

$$= \frac{1}{\|\mathbf{u}\|^2\|\mathbf{x}\|^2 - \langle \mathbf{x},\mathbf{u}\rangle^2} \begin{bmatrix} \|\mathbf{u}\|^2 & -\langle \mathbf{x},\mathbf{u}\rangle \\ -\langle \mathbf{x},\mathbf{u}\rangle & \|\mathbf{x}\|^2 \end{bmatrix} \begin{bmatrix} \langle \mathbf{x},\mathbf{y}\rangle \\ \langle \mathbf{y},\mathbf{u}\rangle \end{bmatrix} \tag{B.7}$$

from which Eqs. 5.4 and 5.5 follow directly

$$a = \frac{\|\mathbf{u}\|^2\langle \mathbf{x},\mathbf{y}\rangle - \langle \mathbf{x},\mathbf{u}\rangle\langle \mathbf{y},\mathbf{u}\rangle}{\|\mathbf{u}\|^2\|\mathbf{x}\|^2 - \langle \mathbf{x},\mathbf{u}\rangle^2} \tag{B.8}$$

and

$$t = \frac{\|\mathbf{x}\|^2\langle \mathbf{y},\mathbf{u}\rangle - \langle \mathbf{x},\mathbf{y}\rangle\langle \mathbf{x},\mathbf{u}\rangle}{\|\mathbf{u}\|^2\|\mathbf{x}\|^2 - \langle \mathbf{x},\mathbf{u}\rangle^2}. \tag{B.9}$$

The same results can be derived using the *projection theorem* [NaSe82] which would lead directly to Eq. B.6. However this would require the introduction of an number of additional topics in metric topology such as *orthonormal* and *spanning* sets.

APPENDIX C

STRUCTURE CHARTS AND FUNCTIONAL DESCRIPTION OF FBC IMPLEMENTATION

C.1 Data Structures

Image (`img`) - A two dimensional array of pixels which represents an image. Images to be coded as well as decoded images are stored in data structures of this type.

Vector (`blk`) - This data structure describes a range block. It has three basic fields *pixels*, *norm*, and *projection*. *Pixels* is a two dimensional array which contains the actual picture elements of the range block. *Norm* contains the norm of the range block and *projection* is the projection of the block along the **u** axis or $\langle \bullet, \mathbf{u} \rangle$.

Codebook (`cbk`) - A codebook is a table of **vectors** which represent the prototypes of the VQ codebook.

Classes (`ndx`) - This data structure contains the classes and isometries associated with each range block in the range image. The structure has four fields, *class*, *isometry*, *norm*, and a *projection*. The *class* is the index to the VQ prototype which most closely resembles the range vector. The *isometry* field represents that isometry which best maps the range vector into the prototype class. The *norm* and *projection* are as described for the vector data structure. These are calculated and saved to reduce computation in during fractal coding of the image where they would otherwise have to be recalculated repeatedly.

Fractal Code (`fcd`) - This structure contains the fractal code for an image. It contains five fields which represent the parameters of the fractal block transform for each range block in the image. The first two fields are the *x pointer* and *y pointer* to the source domain block . The remaining fields are the *isometry*, *scale*, and *translate* parameters of the fractal transform.

C.2 Structure Charts and Functional Description

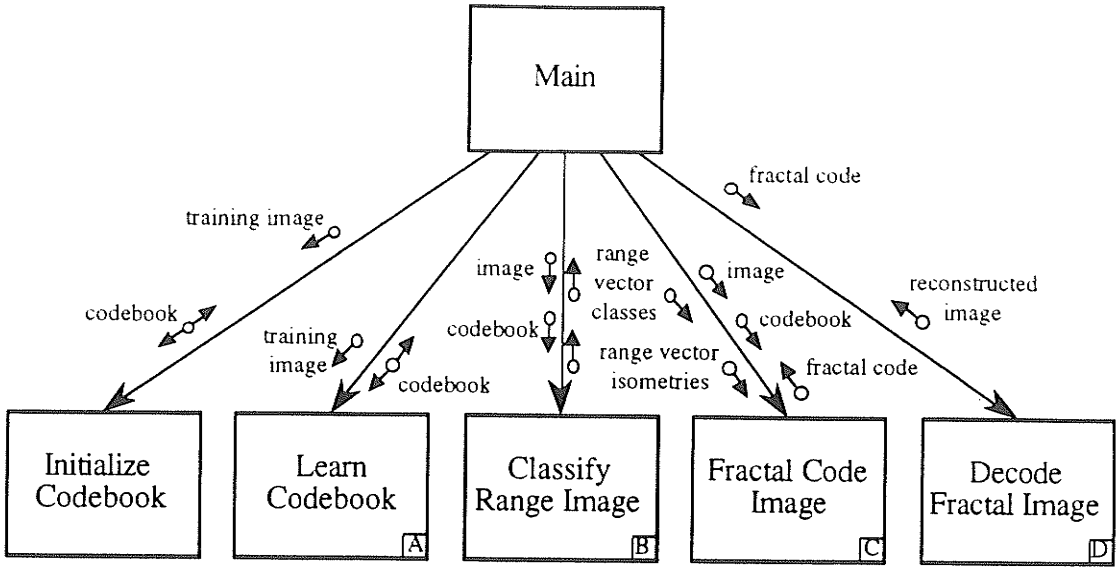


Fig. C.1. Structure chart for reduced search FBC employing FSCL.

Initialize Codebook.
Module: `FSCL`.
Alias: `cbk_init`.

Inputs: training image (type **image**),
codebook (type **codebook**).
Outputs: initialized codebook.

Description: This module initializes the prototype vectors in the VQ codebook based on the training image. Each prototype is initialized to the mean of all of the possible vectors in the training image plus some small random perturbation.

Learn Codebook.
Module: `FSCL`.
Alias: `cbk_learn`

Inputs: training image (type **image**)
randomly initialized codebook (type **codebook**)
Outputs: updated codebook.

Description: This function generates a codebook which is statistically representative of the vectors in the training image using the frequency sensitive competitive learning algorithm. A structure of this function and its subordinates is shown in Fig C.2. Each of the resulting codebook vectors is orthogonal to the vector **u** and normalized. Each training vector is presented to the network in each of its eight possible isometric configurations and the best prototype as well as configuration is chosen. The best prototype and configuration are then used to update the codebook as per the FSCL learning rule.

Search Codebook (Learning)
Module: `FSCL`.
Alias: `cbk_search_learn`.

Inputs: image vector (type **vector**),
codebook (type **codebook**),
frequency.
Outputs: winner,
isometry.

Description: This function searches the codebook for the best prototype for the image vector based on the learning phase of the FSCL algorithm. The image vector to be classified must be compared against every prototype in each of its eight possible isometric configurations. Since the FSCL learning algorithm attempts to ensure that each prototype is chosen an equal number of times, a table of frequencies is maintained for each codebook prototype. The frequency table records the number of times that each prototype has been chosen as the winner. This value is divided by the correlation measure for each prototype and the winner is chosen according to this weighted value. The index of the winning prototype as well as the best isometry are returned.

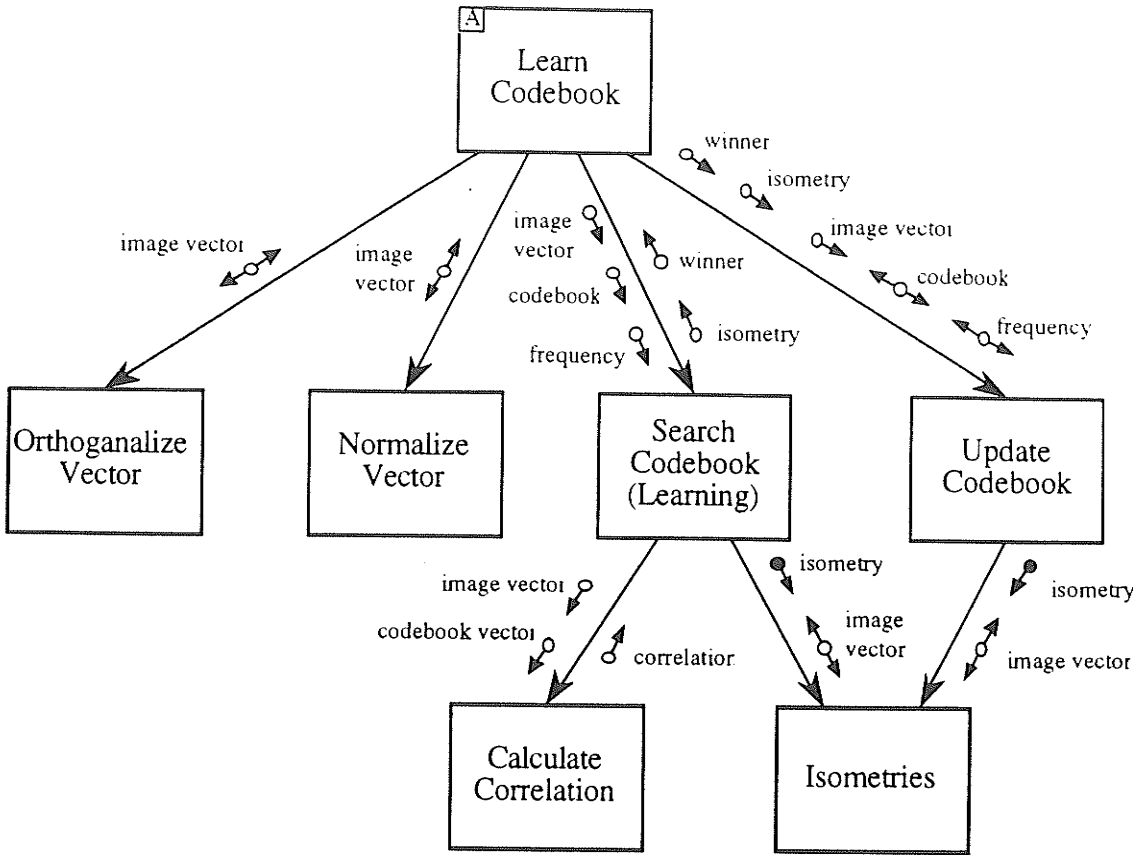


Fig.C.2. The Learn Codebook function and its subordinates.

Update Codebook.
(Physically contained within **Learn Codebook**)

Inputs: winner,
isometry,
image vector (type **vector**),
codebook (type **vector**),
frequency.
Outputs: updated codebook,
updated frequency table.

Description: This function updates the components of the winning prototype in the codebook according to the FSCL learning rule. The training vector and

isometry,are passed to the function. The training vector is then transformed by applying the appropriate isometry to it and the winning prototype in the codebook is then updated using this transformed vector.

Classify Range Image.
Module: `FSCL`
Alias: `ndx_img`.

Inputs: image to be coded (type **image**)
codebook (type **codebook**)
Outputs: range vector classes and isometries (type **classes**)

Description: This function divides the image to be coded into range vectors. Each of these vectors is then presented to the vector quantizer and then classified. The class and optimal isometry are then recorded and returned. A structure chart of this function is illustrated in Fig. C.3

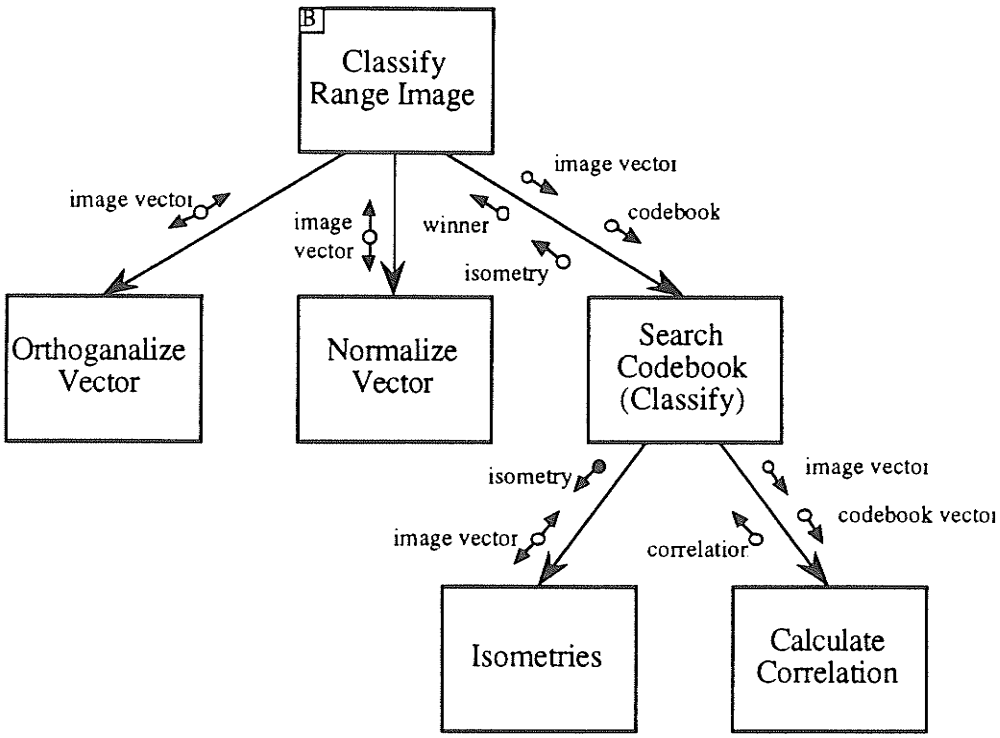


Fig. C.3. The Classify Range Image function and its subordinates.

Search Codebook (Classify)
Module: `FSCL`.
Alias: `cbk_search_range`,
`cbk_search_domain`.

Inputs: image vector (type **vector**),
codebook (type **codebook**).
Outputs: winner,
isometry.

Description: This function classifies an image vector according tho the best matching prototype in the codebook. Because the codebook has been appropriately learned a frequency count is not used here. The function returns the an index to the best prototype indicating the image vector class and the isometry which best maps the image vector into that prototype.

Calculate Correlation
Module: `FSCL`.
Alias: `calc_corr`.

Inputs: image vector (type **vector**),
codebook vector (type **vector**).
Outputs: correlation.

Description: This function receives an image vector and a codebook vector and calculates the inner product or correlation between them.

Fractal Code Image.
Module: FRACTAL.
Alias: fcd_img.

Inputs: image to be coded (type **image**),
codebook(type **codebook**),
range vector classes and isometries (type **classes**).
Outputs: complete fractal code for the image (type **fractal code**).

Description: This function generates the fractal code for a source image. It requires an image, a codebook, and the range classes and isometries for the image to be coded. A structure of the **Fractal Code Image** function and its subordinates are illustrated in Fig. C.4. The function extracts and reduces each domain block in the image. This reduced domain block is then orthonormalized and classified using the vector quantizer and its class as well as best isometry are determined. The reduced domain block is then compared against all of the range blocks in the image which are of the same class. This is done by calculating the isometry which best maps the domain block into the range block, calculating the optimal scaling and translation coefficients, and then determining the error. If for a particular range block the error is less than that associated with any other domain bock, then the scaling, translation and isometry parameters as well as x and y pointers to the domain block are recorded.

Extract and Reduce Domain Block
Module: FRACTAL.
Alias: blk_rdc.

Inputs: image to be coded (type **image**),
vector pointer.
Outputs: reduced domain vector (type **vector**)

Description: This function extracts domain vector from a image at the x and y positions indicated by the vector pointer. This domain vector is then reduced to the same size as range blocks and returned.

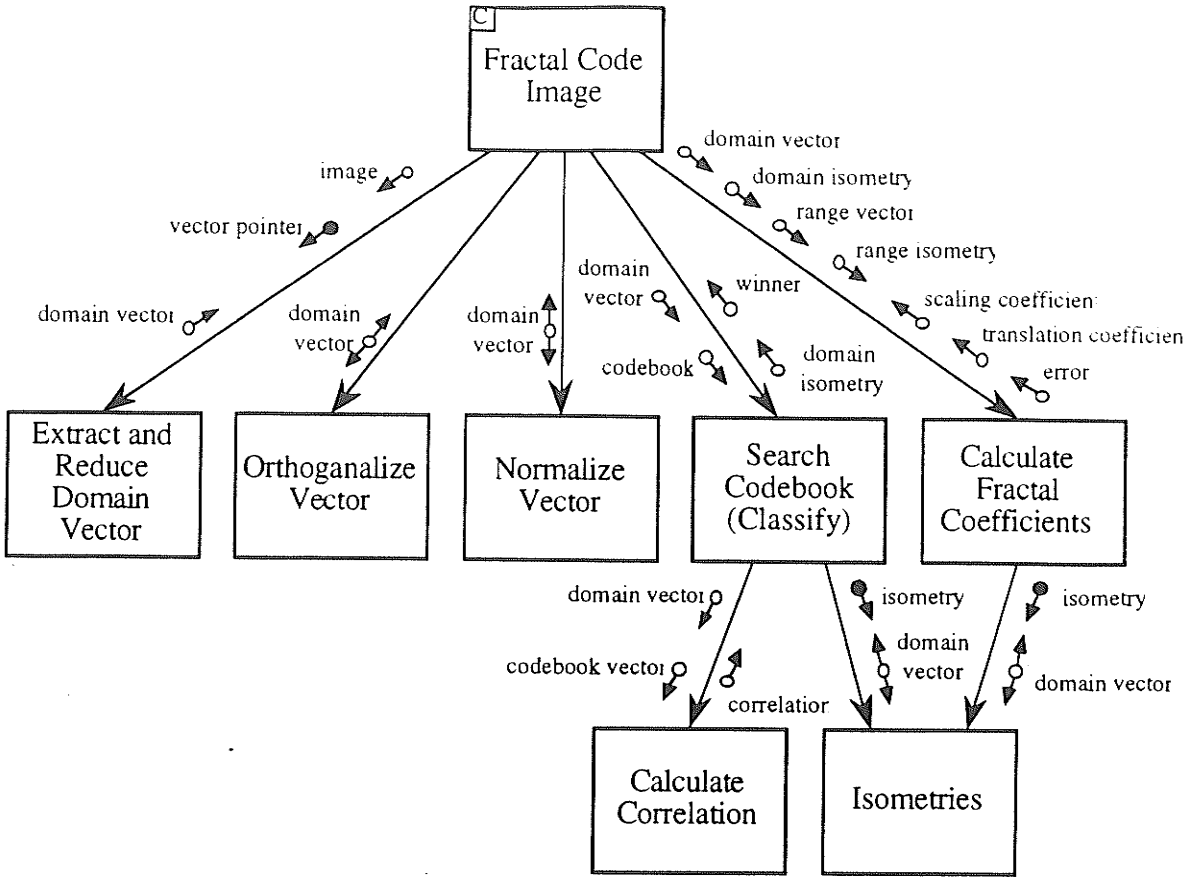


Fig. C.4. The Fractal Code Image function and its subordinates.

Calculate Fractal Coefficients
(Physically contained within **Fractal Code Image**.)

Inputs: reduced domain vector (type **vector**),
range vector (type **vector**),
domain isometry,
range isometry,
Outputs: combined isometry,
scaling coefficient,
translation coefficient,
error.

Description: This function calculates and returns the ideal fractal coefficients for mapping the domain vector into the range vector. The remaining error resulting from these ideal coefficients is also computed and returned.

Decode Fractal Image.
Module: FRACTALS.
Alias: fcd_img.

Inputs: fractal code for image (type **fractal code**)
Output: image reconstructed from the fractal code (type **image**)

Description: This function, shown in Fig. C.5, reconstructs the original image from its fractal code using the iterative reconstruction algorithm. The function begins with an arbitrary image and then maps it into a new image according to the parameters of the fractal code. This image then becomes the source image and the procedure is repeated iteratively.

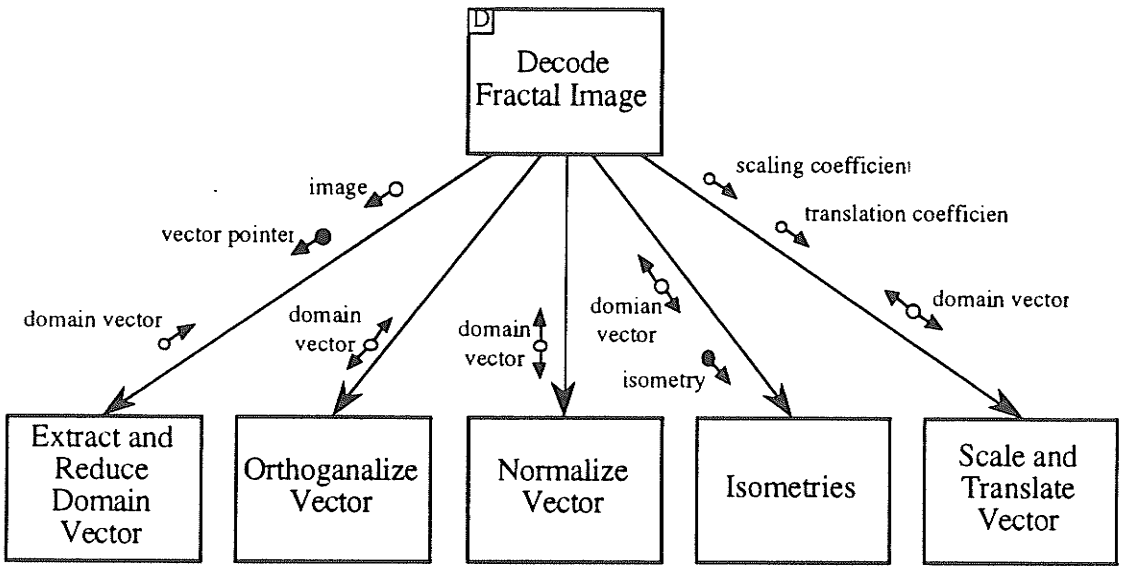


Fig. C.5. The Decode Fractal Image function and its subordinates.

Orthogonalize Vector

Module: TRANSFORMS.
Alias: blk_orth.

Inputs: image vector (type **vector**).
Outputs: orthoganalized vector (type **vector**)

Description: This function removes the component in the **u** direction from a vector.

Normalize Vector

Module: TRANSFORMS.
Alias: blk_nrm.

Inputs: image vector (type **vector**)
Outputs: normalized image vector (type **vector**)

Description: This function normalizes an image vector.

Isometries

Module: TRANSFORMS.
Alias: isometries.

Inputs: isometry,
image vector (type **vector**)
Outputs: isometrically transformed image vector (type **vector**)

Description: This function performs one of eight isometric transformations on an image block. The particular transform is selected according to the isometry index. The actual transformations are handled by eight subordinate functions

(1) idnt - identity,
(2) flp_x - flips the vector about the x axis,
(2) flp_y - flips the vector about the x axis,
(3) flp_d1 - flips the vector about the first diagonal,
(4) flp_d2 - flips the vector about the second diagonal,
(5) rot_90 - rotates the vector 90° about the center,
(6) rot_180 - rotates the vector 180° about the center,
(7) rot_270 - rotates the vector 230° (-90°) about the center.

APPENDIX D

C LANGUAGE LISTINGS FOR CONCATENATED FBC/ARITHMETIC IMAGE COMPRESSION SOFTWARE

```
/*============================================================

Program:        Block Oriented Fractal Data Compression of Digital
                Images.

Programmer:     Larry M. Wall
                Department of Electrical and Computer Engineering
                University of Manitoba
                Winnipeg, Canada
                (larwall@ee.umanitoba.ca)

Version:        2.1

Last Update:    16/02/93

Comments:       V2.0 impliments a reduced range search by employing
                vector quantization.  A number of new data structures are
                introduced to improve modularity.  It also stores the
                fractal code in a format appropriate for additional
                compression later with arithmetic coding.

/*------------------------------------------------------------

LIBRARIES:                                                      */

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

#include "fbc_constants.h"
#include "fbc_fractals.h"
#include "fbc_fscl.h"
#include "fbc_arithmetic.h"
#include "fbc_io.h"

/*============================================================

MAIN PROGRAM:                                                   */

main()
{

        unsigned char *img_1,           /* source image         */
                      *img_2,
                      *img_f;           /* decoded fractal image */

        fractal       *fcd_1;           /* fractal code         */
```

```
vector      *cbk_1;          /* VQ codebook                    */
index       *ndx_1;          /* VQ coded image                 */

time_t lt;
struct tm
       *started,
       *finished;

img_1 = img_alloc();
fcd_1 = fcd_alloc();
cbk_1 = cbk_alloc();
ndx_1 = ndx_alloc();


puts( "Loading Image Data." );
img_load( "len1.img", img_1 );

puts( "Initializing VQ Codebook." );
cbk_init( img_1, cbk_1 );

puts( "Learning VQ Codebook." );
lt = time( NULL );
started = localtime( &lt );
printf( asctime( started ) );
cbk_learn( img_1, cbk_1 );
lt = time( NULL );
finished = localtime( &lt );
printf( asctime( finished ) );

puts( "Saving VQ Codebook." );
cbk_save( "len11.cbk", cbk_1 );

puts( "Loading VQ Codebook." );
cbk_load( "len11.cbk", cbk_1 );

puts( "Generating Picture of the Codebook." );
img_f = img_alloc();
cbk_img( cbk_1, img_f );

puts( "Saving Displayed Codebook." );
img_save( "len11CBK.img", img_f );
free( img_f );

puts( "Quantizing Image." );
lt = time( NULL );
started = localtime( &lt );
printf( asctime( started ) );
ndx_img( img_1, cbk_1, ndx_1);
```

```
        lt = time( NULL );
        finished = localtime( &lt );
        printf( asctime( finished ) );

        puts( "Reduced Fractal Coding Image." );
        lt = time( NULL );
        started = localtime( &lt );
        printf( asctime( started ) );
        img_code_r( img_1, cbk_1, ndx_1, fcd_1 );
        free( img_1 );
        lt = time( NULL );
        finished = localtime( &lt );
        printf( asctime( finished ) );

/*      puts( "Exhaustive Fractal Coding Image." );
        lt = time( NULL );
        started = localtime( &lt );
        printf( asctime( started ) );
        img_code_e( img_1, fcd_1 );
        free( img_1 );
        lt = time( NULL );
        finished = localtime( &lt );
        printf( asctime( finished ) );*/

        puts( "Saving Fractal Coded Image." );
        fcd_save( "len11.fcd", fcd_1 );

        puts( "Loading Fractal Coded Image." );
        fcd_load( "len11.fcd", fcd_1 );

        puts( "Packing Fractal Code." );
        fcd_pack( fcd_1 );

        puts( "Arithmetic Compressing Fractal Coded Image." );
        ac_compress( "len11.ac", fcd_1 );

        puts( "Arithmetic Decompressing Fractal Coded Image." );
        ac_decompress( "len11.ac", fcd_1 );

        puts( "Unpacking Fractal Code." );
        fcd_unpack( fcd_1 );

        puts( "Decoding Fractal Image." );
        lt = time( NULL );
        started = localtime( &lt );
        printf( asctime( started ) );
        img_f = img_alloc();
```

```
        fcd_img( fcd_1, img_f);
        lt = time( NULL );
        finished = localtime( &lt );
        printf( asctime( finished ) );

        puts( "Saving Decoded Fractal Image." );
        img_save( "len11.img", img_f );

        free( img_f );
        free( fcd_1 );

}

/*============================================================*/
```

```
/*============================================================

CONSTANTS:                                                    */

#define IMG_SZ 256 /* size of image in pixels                 */
#define BLK_SZ 8   /* size of range block in pixels           */
#define BLK_MX 7   /* BLK_SZ - 1                              */
#define BLK_NM 32  /* number of blocks per image dimension    */
#define DMN_SZ 16  /* size of domain block in pixels          */
#define SCL_SZ 2   /* scale factor between range and domain blocks */
#define STP_SZ 1   /* step size between domain blocks         */
#define FRC_IT 10  /* number of itterations of decoding algorithm */
#define CBK_SZ 11  /* size of the fscl codebook               */

static unsigned char t_t[8][8] = {  /* Array which indicates the result of */
        0, 1, 2, 3, 4, 5, 6, 7,     /*    multiple applications of block   */
        1, 0, 6, 7, 5, 4, 2, 3,     /*    transformations                  */
        2, 6, 0, 5, 7, 3, 1, 4,
        3, 5, 7, 0, 6, 1, 4, 2,
        4, 7, 5, 6, 0, 2, 3, 1,
        5, 3, 4, 2, 1, 6, 7, 0,
        6, 2, 1, 4, 3, 7, 0, 5,
        7, 4, 3, 1, 2, 0, 5, 6
        };

/*------------------------------------------------------------

TYPE DECLARATIONS:                                            */

        typedef struct {
                int     x,
                        y,
                        translate,
                        scale,
                        transform;
        } fractal;

        typedef struct {
                float   pxls[BLK_SZ][BLK_SZ],
                        s1,
                        s2;
        } vector;

        typedef struct {
                int     ptr;
                unsigned char
                        transform;
```

```
            float   s1,
                    s2;
    } index;

/*=========================================================*/
```

Header:

```
extern void img_code_r();
extern void img_code_e();
extern void fcd_img();
extern void fcd_disp();
extern void fcd_pack();
extern void fcd_unpack();
```

Source:

```
/*============================================================

Module:       FRACTALS

Program:      Block Oriented Fractal Data Compression of Digital
              Images.

Programmer:   Larry M. Wall
              Department of Electrical and Computer Engineering
              University of Manitoba
              Winnipeg, Canada
              (larwall@ee.umanitoba.ca)

Version:      2.2

Last Update:  12/02/93

Comments:     This module contains the code necessary to encode and
              decode a fractal representation of an image using the
              reduced search encoding procedure.

/*------------------------------------------------------------

LIBRARIES:                                                    */

#include <stdlib.h>
#include <math.h>
#include <values.h>

#include "fbc_constants.h"
#include "fbc_io.h"
#include "fbc_transforms.h"

#include "fbc_fractals.h"
```

```
/*-------------------------------------------------------------

PUBLIC FUNCTIONS:                                              */

void  img_code_r();
void  img_code_e();
void  fcd_img();
void  fcd_disp();
void  fcd_pack();

/*=============================================================

: Generate fractal code for source image using the reduced search fractal coding
        procedure.                                             */

void img_code_r( image, cbook, vq_code, fr_code )
unsigned char image[IMG_SZ][IMG_SZ];
vector cbook[CBK_SZ];
index  vq_code[BLK_NM][BLK_NM];
fractal  fr_code[BLK_NM][BLK_NM];
{

        float   low_err[BLK_NM][BLK_NM];

        int     img_x,
                img_y,
                blk_x,
                blk_y,
                b_x,
                b_y,
                x,
                y;

        int     type;

        float   scale,
                translate,
                shift;

        float   rd,
                B,
                D = BLK_SZ * BLK_SZ;

        float   error;

        int     c_transform;
        unsigned char
                t_transform,
```

```
            transform;

        vector r_block,
               s_block,
               t_block;

        for ( blk_x=0; blk_x<BLK_NM; blk_x++ )
               for ( blk_y=0; blk_y<BLK_NM; blk_y++ )
                      low_err[blk_x][blk_y] = MAXFLOAT;

        for ( img_x=0; img_x<(IMG_SZ - DMN_SZ); img_x+=STP_SZ )
               for ( img_y=0; img_y<(IMG_SZ - DMN_SZ); img_y+=STP_SZ )
               {
                      blk_rdc( img_x, img_y, image, &r_block );

                      for( x=0; x<BLK_SZ; x++ )
                             for( y=0; y<BLK_SZ; y++ )
                                    s_block.pxls[x][y] = r_block.pxls[x][y];
                      s_block.s1 = r_block.s1;
                      s_block.s2 = r_block.s2;

                      blk_orth( &r_block );
                      blk_orth( &s_block );
                      type = cbk_search_domain( r_block, cbook, &c_transform );

                      t_transform = (unsigned char) c_transform;

                      for ( blk_x=0; blk_x<BLK_NM; blk_x++ )
                             for ( blk_y=0; blk_y<BLK_NM; blk_y++ )
                             {
                                    if( vq_code[blk_x][blk_y].ptr == type )
                                    {
                                           transform =
t_t[t_transform][vq_code[blk_x][blk_y].transform];
                                           blk_isom( s_block.pxls, t_block.pxls,
transform);
                                           b_x = blk_x * BLK_SZ;
                                           b_y = blk_y * BLK_SZ;
                                           rd = 0.0;
                                           for ( x=0; x<BLK_SZ; x++ )
                                                  for ( y=0; y<BLK_SZ; y++ )
                                                         rd += (float) image[b_x +
x][b_y + y] * t_block.pxls[x][y];
                                           scale = (int) rd ;
                                           shift = vq_code[blk_x][blk_y].s1/D;
                                           error = vq_code[blk_x][blk_y].s2 + scale
* scale * s_block.s2 + shift * shift * D + 2 * ( scale * shift * s_block.s1 - scale * rd
```

```
- shift * vq_code[blk_x][blk_y].s1 );
                                        if ( (low_err[blk_x][blk_y] >= error) &&
( fabs(scale) < 1024 ) )
                                        {
                                                low_err[blk_x][blk_y] = error;
                                                fr_code[blk_x][blk_y].x = img_x;
                                                fr_code[blk_x][blk_y].y = img_y;
                                                fr_code[blk_x][blk_y].transform =
transform;
                                                fr_code[blk_x][blk_y].scale =
scale;
                                                fr_code[blk_x][blk_y].translate =
(int) shift;
                                        }
                                }
                        }
                }

}

/*-------------------------------------------------------------

: Generate fractal code for source image using the exhaustive search fractal
        coding procedure.                                                    */

void img_code_e( image, fr_code )
unsigned char image[IMG_SZ][IMG_SZ];
fractal fr_code[BLK_NM][BLK_NM];
{

        float    low_err[BLK_NM][BLK_NM];

        float   rr,
                r;

        int     img_x,
                img_y,
                blk_x,
                blk_y,
                b_x,
                b_y,
                x,
                y;

        float   scale,
                translate,
                shift;
```

```
        float   rd,
                B,
                D = BLK_SZ * BLK_SZ;

        float   error;

        unsigned char
                transform;

        vector r_block,
                s_block,
                t_block;

        for ( blk_x=0; blk_x<BLK_NM; blk_x++ )
                for ( blk_y=0; blk_y<BLK_NM; blk_y++ )
                        low_err[blk_x][blk_y] = MAXFLOAT;

        for ( img_x=0; img_x<(IMG_SZ - DMN_SZ); img_x+=STP_SZ )
                for ( img_y=0; img_y<(IMG_SZ - DMN_SZ); img_y+=STP_SZ )
                {
                        blk_rdc( img_x, img_y, image, &r_block );

                        for( x=0; x<BLK_SZ; x++ )
                                for( y=0; y<BLK_SZ; y++ )
                                        s_block.pxls[x][y] = r_block.pxls[x][y];
                        s_block.s1 = r_block.s1;
                        s_block.s2 = r_block.s2;

                        blk_orth( &r_block );
                        blk_orth( &s_block );

                        for ( blk_x=0; blk_x<BLK_NM; blk_x++ )
                                for ( blk_y=0; blk_y<BLK_NM; blk_y++ )
                                {
                                        b_x = blk_x * BLK_SZ;
                                        b_y = blk_y * BLK_SZ;
                                        r = 0.0;
                                        rr = 0.0;
                                        for ( x=0; x<BLK_SZ; x++ )
                                                for ( y=0; y<BLK_SZ; y++ )
                                                {
                                                        r += (float) image[b_x + x][b_y +
y];
                                                        rr += (float)  image[b_x + x][b_y
+ y] * image[b_x + x][b_y + y];
                                                }
```

```
                                        for( transform=0; transform<8; transform++ )
                                        {
                                                blk_isom( s_block.pxls, t_block.pxls,
transform);
                                                rd = 0.0;
                                                for ( x=0; x<BLK_SZ; x++ )
                                                        for ( y=0; y<BLK_SZ; y++ )
                                                                rd += (float) image[b_x +
x][b_y + y] * t_block.pxls[x][y];
                                                scale = (int) rd ;
                                                shift = r/D;
                                                error = rr + scale * scale * s_block.s2 +
shift * shift * D + 2 * ( scale * shift * s_block.s1 - scale * rd - shift * r );
                                                if ( (low_err[blk_x][blk_y] >= error) &&
( fabs(scale) < 1024 ) )
                                                {
                                                        low_err[blk_x][blk_y] = error;
                                                        fr_code[blk_x][blk_y].x = img_x;
                                                        fr_code[blk_x][blk_y].y = img_y;
                                                        fr_code[blk_x][blk_y].transform =
transform;
                                                        fr_code[blk_x][blk_y].scale =
scale;
                                                        fr_code[blk_x][blk_y].translate =
(int) shift;
                                                }
                                        }
                                }
                        }

}

/*-------------------------------------------------------------

: Reconstruct image from fractal code via iterative algorithm.                 */

void fcd_img( fr_code, image)
fractal fr_code[BLK_NM][BLK_NM];
unsigned char image[IMG_SZ][IMG_SZ];
{

        unsigned char *img_t;

#define TEMP(j,k) (img_t[j*IMG_SZ+k])

        vector s_block,
                t_block;
```

```
        float   lvl;

        int     i,
                blk_x,
                blk_y,
                x,
                y,
                img_x,
                img_y;

        img_t = img_alloc();

        for ( x=0; x<BLK_SZ; x++ )
                for ( y=0; y<BLK_SZ; y++ )
                        TEMP(x,y) = 0;

        for ( i=0; i<FRC_IT; i++ )
        {
                for ( blk_x=0; blk_x<BLK_NM; blk_x++ )
                        for ( blk_y=0; blk_y<BLK_NM; blk_y++ )
                        {
                                blk_rdc(  fr_code[blk_x][blk_y].x,
fr_code[blk_x][blk_y].y,  img_t,  &s_block );
                                blk_orth( &s_block );
                                blk_isom( s_block.pxls, t_block.pxls,
fr_code[blk_x][blk_y].transform);
                                for ( x=0; x<BLK_SZ; x++ )
                                        for ( y=0; y<BLK_SZ; y++ )
                                        {
                                                lvl = (float) ( t_block.pxls[x][y] *
fr_code[blk_x][blk_y].scale ) + fr_code[blk_x][blk_y].translate;
                                                if ( lvl < 0.0 )
                                                        lvl = 0.0;
                                                if ( lvl > 255.0 )
                                                        lvl = 255.0;
                                                image[( blk_x * BLK_SZ ) + x][( blk_y *
BLK_SZ ) +y] = (unsigned char) lvl;
                                        }
                        }
                for ( img_x=0; img_x<IMG_SZ; img_x++ )
                        for ( img_y=0; img_y<IMG_SZ; img_y++ )
                                TEMP(img_x,img_y) = image[img_x][img_y];
        }

        free( img_t );

}
```

```
/*-------------------------------------------------------------------

: Dispaly fractal code on screen.                                    */

void fcd_disp( code )
fractal  code[BLK_NM][BLK_NM];
{

        int     blk_x,
                blk_y;

        for ( blk_y=0; blk_y<BLK_NM; blk_y++ )
                for ( blk_x=0; blk_x<BLK_NM; blk_x++ )
                        printf("%5d %5d %5d %5d %5d %5d %5d\n", blk_x, blk_y,
code[blk_x][blk_y].x,
                                code[blk_x][blk_y].y,  code[blk_x][blk_y].translate,
code[blk_x][blk_y].scale,
                                (int)  code[blk_x][blk_y].transform  );
}

/*-------------------------------------------------------------------

: Format fractal code for arithmetic compression.                    */

void fcd_pack( code )
fractal code[BLK_NM][BLK_NM];
{
        int     blk_x,
                blk_y,
                max,
                min;

        max = -5000;
        min =  5000;

        for( blk_y=0; blk_y<BLK_NM; blk_y++ )
                for( blk_x=0; blk_x<BLK_NM; blk_x++ )
                {
                        if( code[blk_x][blk_y].scale > max )
                                max = code[blk_x][blk_y].scale;
                        if( code[blk_x][blk_y].scale < min )
                                min = code[blk_x][blk_y].scale;
                }
        for( blk_y=0; blk_y<BLK_NM; blk_y++ )
                for( blk_x=0; blk_x<BLK_NM; blk_x++ )
                {
                        code[blk_x][blk_y].x /= STP_SZ;
```

```
                code[blk_x][blk_y].y /= STP_SZ;
                code[blk_x][blk_y].scale += 1024;
            }

        printf( "Range: %d\n", max - min );


        for( blk_y=0; blk_y<BLK_NM; blk_y++ )
            for( blk_x=0; blk_x<(BLK_NM-1); blk_x++ )
                code[blk_x][blk_y].translate = code[blk_x+1][blk_y].translate
- code[blk_x][blk_y].translate + 256;

        for( blk_y=0; blk_y<BLK_NM-1; blk_y++ )
            code[BLK_NM-1][blk_y].translate = code[BLK_NM-
1][blk_y+1].translate - code[BLK_NM-1][blk_y].translate + 256;

}
/*-------------------------------------------------------------

: Recover fractal code arter arithmetic decompression.                    */

void fcd_unpack( code )
fractal code[BLK_NM][BLK_NM];
{
        int     blk_x,
                blk_y,
                max,
                min;

        for( blk_y=0; blk_y<BLK_NM; blk_y++ )
            for( blk_x=0; blk_x<BLK_NM; blk_x++ )
            {
                code[blk_x][blk_y].x *= STP_SZ;
                code[blk_x][blk_y].y *= STP_SZ;
                code[blk_x][blk_y].scale -= 1024;
            }

        for( blk_y=(BLK_NM-1); blk_y>0; blk_y-- )
            code[BLK_NM-1][blk_y-1].translate = code[BLK_NM-
1][blk_y].translate - code[BLK_NM-1][blk_y-1].translate + 256;

        for( blk_y=0; blk_y<BLK_NM; blk_y++ )
            for( blk_x=(BLK_NM-1); blk_x>0; blk_x-- )
                code[blk_x-1][blk_y].translate = code[blk_x][blk_y].translate
- code[blk_x-1][blk_y].translate + 256;

}
```

```
/*============================================================*/
```

Header:

```
extern  void  cbk_init();
extern  void  cbk_learn();
extern  int   cbk_search_range();
extern  int   cbk_search_domain();
extern  void  ndx_img();
extern  void  cbk_img();
```

Source:

```
/*============================================================

Module:       FSCL

Program:      Block Oriented Fractal Data Compression of Digital
              Images.

Programmer:   Larry M. Wall
              Department of Electrical and Computer Engineering
              University of Manitoba
              Winnipeg, Canada
              (larwall@ee.umanitoba.ca)

Version:      2.0

Last Update:  07/06/92

Comments:     This module contains all of the functions for associated
              with the frequency sensitive competetive learning neural
              network.  Functions are included to intititalize the
              network, learn an appropriate set of weights, and
              classify domain and range blocks.

/*------------------------------------------------------------

LIBRARIES:                                                    */

#include <stdlib.h>
#include <math.h>
#include <values.h>

#include "fbc_constants.h"
#include  "fbc_transforms.h"
```

```
#include "fbc_fscl.h"

/*-------------------------------------------------------------

PUBLIC FUNCTIONS:                                              */

void  cbk_init();
void  cbk_learn();
int   cbk_search_range();
int   cbk_search_domain();
void  ndx_img();
void  cbk_img();

/*-------------------------------------------------------------

PRIVATE FUNCTIONS:                                             */

static int    cbk_search_learn();
static float  calc_error();

/*=============================================================

: Initializes each element in the codebook to the average of all of the blocks in the
      image plus some random pertibation.                      */

void cbk_init( image, cbook )
unsigned char image[IMG_SZ][IMG_SZ];
vector cbook[CBK_SZ];
{

      vector avg;
      float  rnd_nm;

      int    blk_x,
             blk_y,
             x,
             y,
             node;

      avg.s1 = avg.s2 = 0.0;
      for ( x=0; x<BLK_SZ; x++ )
            for ( y=0; y<BLK_SZ; y++ )
                  avg.pxls[x][y] = 0.0;
```

```
        for ( blk_x=0; blk_x<IMG_SZ; blk_x+=BLK_SZ )
                for ( blk_y=0; blk_y<IMG_SZ; blk_y+=BLK_SZ )
                        for ( x=0; x<BLK_SZ; x++ )
                                for ( y=0; y<BLK_SZ; y++ )
                                        avg.pxls[x][y] += (float) ( image[blk_x +
x][blk_y + y] );

        for ( x=0; x<BLK_SZ; x++ )
                for ( y=0; y<BLK_SZ; y++ )
                {
                        avg.s1 += avg.pxls[x][y];
                        avg.s2 += avg.pxls[x][y] * avg.pxls[x][y];
                }

        blk_orth( &avg );

        for ( node=0; node<CBK_SZ; node++)
        {
                cbook[node].s1 = cbook[node].s2 = 0.0;
                for ( x=0; x<BLK_SZ; x++ )
                        for ( y=0; y<BLK_SZ; y++ )
                        {
                                rnd_nm = ( ( (float) random() / MAXINT ) - 0.5 ) * 0.2
+  avg.pxls[x][y];
                                cbook[node].pxls[x][y] = rnd_nm;
                                cbook[node].s1 += rnd_nm;
                                cbook[node].s2 += rnd_nm * rnd_nm;
                        }
                blk_orth( &cbook[node] );
        }

}

/*-------------------------------------------------------------

: Generate the codebook using the Frequency Sensitive Competitive Learning
        (FSCL) algorithm.                                                    */

void cbk_learn( image, cbook )
unsigned char image[IMG_SZ][IMG_SZ];
vector cbook[CBK_SZ];
{
```

```
        vector s_block,
               t_block;

        float  frequency[CBK_SZ],

               i_gn = 0.2,
               f_gn = 0.01,
               c_gn,
               w_chg,
               scale;

        int    rnd_x,
               rnd_y,
               x,
               y,
               node,
               transform;

        long int
               time,
               mx_time = CBK_SZ * 1450;

        for ( node=0; node<CBK_SZ; node++ )
               frequency[node] = 1.0;

        for ( time=0; time<mx_time; time++ )
        {
               c_gn = ( ( i_gn - f_gn ) * ( 1.0 - (float) time / mx_time ) + f_gn );
               rnd_x = (int) random() % ( IMG_SZ - DMN_SZ );
               rnd_y = (int) random() % ( IMG_SZ - DMN_SZ );

               blk_rdc( rnd_x, rnd_y, image, &s_block );
               blk_orth( &s_block );
               node  = cbk_search_learn( s_block, cbook, frequency, &scale,
&transform );
               blk_isom( s_block.pxls, t_block.pxls, transform);
               cbook[node].s1 = cbook[node].s2 = 0.0;
               for( x=0; x<BLK_SZ; x++ )
                      for( y=0; y<BLK_SZ; y++ )
                      {
                             t_block.pxls[x][y] *= scale;
                             w_chg = c_gn * ( t_block.pxls[x][y] -
cbook[node].pxls[x][y] );
                             cbook[node].pxls[x][y] += w_chg;
```

```
                        cbook[node].s1 += cbook[node].pxls[x][y];
                        cbook[node].s2 += cbook[node].pxls[x][y] *
cbook[node].pxls[x][y];
                    }
            blk_norm( &cbook[node] );

            frequency[node] += 1.0;

        }

        for (node=0; node<CBK_SZ; node++)
        {
            blk_orth( &cbook[node] );
            printf( "%f\n", frequency[node] );
        }

}

/*-------------------------------------------------------------

: Search the codebook for the best match to the input vector taking into
        consideration frequency of occurance as per the FSCL learning algorithm. */

static int cbk_search_learn( s_block, cbook, frequency, scale, transform )
vector s_block;
vector cbook[CBK_SZ];
float frequency[CBK_SZ];
float *scale;
int    *transform;
{

        int     x,
                y,
                node,
                t_transform,
                best_node = 0;

        float   error,
                low_err,
                t_scale;

        low_err = MAXFLOAT;

        for ( node=0; node<CBK_SZ; node++ )
```

```
        {
                error = calc_error( s_block, cbook[node], &t_scale, &t_transform ) *
frequency[node];
                if ( error <= low_err )
                {
                        low_err = error;
                        best_node = node;
                        *scale = t_scale;
                        *transform = t_transform;
                }
        }

        return best_node;
}

/*-------------------------------------------------------------------

: Search the codebook for the best match to the input range vector.          */

int cbk_search_range( s_block, cbook, transform )
vector s_block;
vector cbook[CBK_SZ];
int    *transform;
{

        int     x,
                y,
                node,
                t_transform,
                best_node = 0;

        float   error,
                low_err,
                t_scale;

        low_err = MAXFLOAT;

        for ( node=0; node<CBK_SZ; node++ )
        {
                error = calc_error( cbook[node], s_block, &t_scale, &t_transform );
                if ( error <= low_err )
                {
                        low_err      = error;
                        best_node = node;
                        *transform = t_transform;
```

```
                }
        }

        return best_node;
}

/*-----------------------------------------------------------------

: Search the codebook for the best match to the input domain vector.        */

int cbk_search_domain( s_block, cbook, transform )
vector s_block;
vector cbook[CBK_SZ];
int    *transform;
{


        int     x,
                y,
                node,
                t_transform,
                best_node = 0;

        float   error,
                low_err,
                t_scale,
                t_translate;

        low_err = MAXFLOAT;

        for ( node=0; node<CBK_SZ; node++ )
        {
                error = calc_error( s_block, cbook[node], &t_scale, &t_transform );
                if ( error <= low_err )
                {
                        low_err = error;
                        best_node = node;
                        *transform = t_transform;
                }
        }

        return best_node;
}
```

```
/*-------------------------------------------------------------

: Generates the coded version of the image.                    */

void ndx_img( image, cbook, vq_code )
unsigned char image[IMG_SZ][IMG_SZ];
vector cbook[CBK_SZ];
index vq_code[BLK_NM][BLK_NM];
{

        vector s_block;

        int     blk_x,
                blk_y,
                x,
                y,
                t_transform;

        for ( blk_y=0; blk_y<BLK_NM; blk_y++)
                for ( blk_x=0; blk_x<BLK_NM; blk_x++ )
                {
                        s_block.s1 = s_block.s2 = 0.0;
                        for ( x=0; x<BLK_SZ; x++ )
                                for ( y=0; y<BLK_SZ; y++ )
                                {
                                        s_block.pxls[x][y] = (float) image[ blk_x *
BLK_SZ + x][ blk_y * BLK_SZ + y ];
                                        s_block.s1  += s_block.pxls[x][y];
                                        s_block.s2  += s_block.pxls[x][y] *
s_block.pxls[x][y];
                                }
                        vq_code[blk_x][blk_y].s1 = s_block.s1;
                        vq_code[blk_x][blk_y].s2 = s_block.s2;

                        blk_orth( &s_block );

                        vq_code[blk_x][blk_y].ptr = cbk_search_range( s_block, cbook,
&t_transform );
                        vq_code[blk_x][blk_y].transform = ( unsigned char )
t_transform;
                }

}
```

```
/*-----------------------------------------------------------------

: Calculate the difference between two vectors independent of amplitude scaling
        and orientation.  Return the optimum scale, and transform values.      */

static float calc_error( a_block, b_block, scale, transform)
vector a_block;
vector b_block;
float *scale;
int    *transform;
{

        vector  t_block;

        int     x,
                y,
                t_transform;

        float   ab,
                error,
                low_err,
                t_scale;

        low_err= MAXFLOAT;

        for ( t_transform=0; t_transform<8; t_transform++ )
        {
                blk_isom( a_block.pxls, t_block.pxls, t_transform);

                ab = 0.0;
                for ( x=0; x<BLK_SZ; x++ )
                        for ( y=0; y<BLK_SZ; y++ )
                                ab += ( t_block.pxls[x][y] * b_block.pxls[x][y] );

                if( ab < 0.0 )
                {
                        t_scale = -1.0;
                        ab = fabs(ab);
                }
                else
                        t_scale= 1.0;

                error = 2.0 /*b_block.s2*/ - ( 2.0 * ab ) /*+ a_block.s2*/;
                if( error <= low_err )
```

```
                {
                        low_err = error;
                        *scale = t_scale;
                        *transform = t_transform;
                }
        }

        return low_err;

}

/*-------------------------------------------------------------

: Display the codebook in image format.                          */

void cbk_img( cbook, image )
vector cbook[CBK_SZ];
unsigned char image[IMG_SZ][IMG_SZ];
{

        int     x,
                y,
                blk_x,
                blk_y,
                pxl_x,
                pxl_y,
                count;

        float   max,
                min,
                scale,
                trans;

        for( x=0; x<IMG_SZ; x++ )
                for( y=0; y<IMG_SZ; y++ )
                        image[x][y] = 255;

        for( count=0; count<CBK_SZ; count++ )
        {
                blk_x = count / 4;
                blk_y = count % 4;
                for( pxl_x=0; pxl_x<BLK_SZ; pxl_x++)
                        for( pxl_y=0; pxl_y<BLK_SZ; pxl_y++ )
                                for( x=0; x<7; x++ )
```

```
                                for( y=0; y<7; y++ )
                                    image[blk_x*72 + pxl_x*7 +
x][blk_y*64 + pxl_y*7 + y] = (unsigned char) (cbook[count].pxls[pxl_x][pxl_y] *
127 + 128);
        }

}

/*============================================================*/
```

Header:

```
extern  void  blk_rdc();
extern  void  blk_orth();
extern  void  blk_norm();
extern  void  blk_isom();
```

Source:

```
/*============================================================

Module:         TRANSFORMS

Program:        Block Oriented Fractal Data Compression of Digital
                Images.

Programmer:     Larry M. Wall
                Department of Electrical and Computer Engineering
                University of Manitoba
                Winnipeg, Canada
                (larwall@ee.umanitoba.ca)

Version:        2.0

Last Update:    07/06/92

Comments:       This module contains all the functions which perform
                the individual fractal block transforms including
                spatial contraction, isometric block transforms,
                scaling, translation, and orthonormalization.

/*------------------------------------------------------------

LIBRARIES:                                                    */

#include <stdlib.h>
#include <math.h>
#include <values.h>

#include "fbc_constants.h"

#include "fbc_transforms.h"

/*------------------------------------------------------------

PUBLIC FUNCTIONS:                                             */
```

```
void  blk_rdc();
void  blk_orth();
void  blk_norm();
void  blk_isom();

/*-------------------------------------------------------------

PRIVATE FUNCTIONS:                                              */

void  idnt();
void  flp_x();
void  flp_y();
void  flp_d1();
void  flp_d2();
void  rot_90();
void  rot_180();
void  rot_270();

/*=============================================================

: Reduce image block from domain size to range size.            */

void blk_rdc( x, y, image, s_block )
int     x,
        y;
unsigned char image[IMG_SZ][IMG_SZ];
vector  *s_block;
{
        int     rng_x,
                rng_y,
                dmn_x,
                dmn_y;

        float   avg;

        s_block->s1 = s_block->s2 = 0.0;
        for ( rng_x=0; rng_x<BLK_SZ; rng_x++ )
                for ( rng_y=0; rng_y<BLK_SZ; rng_y++ )
                {
                        avg = 0.0;
                        for ( dmn_x=0; dmn_x<SCL_SZ; dmn_x++ )
                                for ( dmn_y=0; dmn_y<SCL_SZ; dmn_y++ )
                                        avg += image[x + ( rng_x * SCL_SZ ) + dmn_x][y
+ ( rng_y * SCL_SZ ) + dmn_y];
                        s_block->pxls[rng_x][rng_y] =  ( avg / ( SCL_SZ * SCL_SZ ) );
                        s_block->s1  +=  s_block->pxls[rng_x][rng_y];
                        s_block->s2  +=  s_block->pxls[rng_x][rng_y]  *  s_block-
```

```
>pxls[rng_x][rng_y];
                }

}

/*-------------------------------------------------------------

: Generate orthonormal vector.                                  */

void blk_orth( s_block )
vector *s_block;
{

        int     x,
                y;

        float   ic;


        ic = s_block->s1 / ( BLK_SZ * BLK_SZ );

        s_block->s1 = s_block->s2 = 0.0;
        for( x=0; x<BLK_SZ; x++ )
                for( y=0; y<BLK_SZ; y++ )
                {
                        s_block->pxls[x][y]  -=  ic;
                        s_block->s2  +=  s_block->pxls[x][y]  *  s_block->pxls[x][y];
                }

        blk_norm( s_block );
}
/*-------------------------------------------------------------

: Select the approriate isometry.                               */

void blk_isom( s_block, t_block, isom )
float  s_block[BLK_SZ][BLK_SZ];
float  t_block[BLK_SZ][BLK_SZ];
int    isom;
{
        switch ( isom )
        {
                case 0: idnt( s_block, t_block);
                        break;
                case 1: flp_x( s_block, t_block);
                        break;
                case 2: flp_y( s_block, t_block );
```

```
                break;
        case 3: flp_d1( s_block, t_block );
                break;
        case 4: flp_d2( s_block, t_block );
                break;
        case 5: rot_90( s_block, t_block );
                break;
        case 6: rot_180( s_block, t_block );
                break;
        case 7: rot_270( s_block, t_block );
                break;
    }

}

/*-------------------------------------------------------------

: Normalize vectors.                                          */

void blk_norm( s_block )
vector *s_block;
{

    int     x,
            y;

    float   norm;

    norm = sqrt( s_block->s2 );

    if( norm > 0.001 )
    {
        for( x=0; x<BLK_SZ; x++ )
            for( y=0; y<BLK_SZ; y++ )
                s_block->pxls[x][y] /= norm;

        s_block->s1 /= norm;
        s_block->s2 = 1.0;
    }

}

/*-------------------------------------------------------------

: Identity transformation.                                    */

void idnt( s_block, t_block )
float s_block[BLK_SZ][BLK_SZ];
```

```
float  t_block[BLK_SZ][BLK_SZ];
{

        int     x,
                y;

        for ( x=0; x<BLK_SZ; x++ )
                for ( y=0; y<BLK_SZ; y++ )
                        t_block[x][y]  =  s_block[x][y];

}

/*-------------------------------------------------------------

: Orthogonal reflection of block about mid-vertical axis.                    */

void flp_x( s_block, t_block )
float  s_block[BLK_SZ][BLK_SZ];
float  t_block[BLK_SZ][BLK_SZ];
{
        int     x,
                y;

        for ( x=0; x<BLK_SZ; x++ )
                for ( y=0; y<BLK_SZ; y++ )
                        t_block[x][y]  =  s_block[BLK_MX  -  x][y];

}

/*-------------------------------------------------------------

: Orthogonal reflection of block about mid-horizontal axis.                  */

void flp_y( s_block, t_block )
float  s_block[BLK_SZ][BLK_SZ];
float  t_block[BLK_SZ][BLK_SZ];
{
        int     x,
                y;

        for ( x=0; x<BLK_SZ; x++ )
                for ( y=0; y<BLK_SZ; y++ )
                        t_block[x][y]  =  s_block[x][BLK_MX  -  y];

}

/*-------------------------------------------------------------
```

```
: Orthogonal reflection of block about first diagonal.                    * /

void flp_d1( s_block, t_block )
float s_block[BLK_SZ][BLK_SZ];
float t_block[BLK_SZ][BLK_SZ];
{
        int     x,
                y;

        for ( x=0; x<BLK_SZ; x++ )
                for ( y=0; y<BLK_SZ; y++ )
                        t_block[x][y] = s_block[BLK_MX - y][BLK_MX - x];

}

/*-------------------------------------------------------------------

: Orthogonal reflection of block about second diagonal.                   * /

void flp_d2( s_block, t_block)
float s_block[BLK_SZ][BLK_SZ];
float t_block[BLK_SZ][BLK_SZ];
{
        int     x,
                y;

        for ( x=0; x<BLK_SZ; x++ )
                for ( y=0; y<BLK_SZ; y++ )
                        t_block[x][y] = s_block[y][x];

}

/*-------------------------------------------------------------------

: Rotate block 270 degrees about center.                                  * /

void rot_270( s_block, t_block )
float s_block[BLK_SZ][BLK_SZ];
float t_block[BLK_SZ][BLK_SZ];
{
        int     x,
                y;

        for ( x=0; x<BLK_SZ; x++ )
                for ( y=0; y<BLK_SZ; y++ )
                        t_block[x][y] = s_block[y][BLK_MX - x];
```

```
}

/*-------------------------------------------------------------

: Rotate block 180 degrees about center.                        */

void rot_180( s_block, t_block )
float s_block[BLK_SZ][BLK_SZ];
float t_block[BLK_SZ][BLK_SZ];
{
        int     x,
                y;

        for ( x=0; x<BLK_SZ; x++ )
                for ( y=0; y<BLK_SZ; y++ )
                        t_block[x][y] = s_block[BLK_MX - x][BLK_MX - y];

}

/*-------------------------------------------------------------

: Rotate block 90 degrees about center.                         */

void rot_90( s_block, t_block )
float s_block[BLK_SZ][BLK_SZ];
float t_block[BLK_SZ][BLK_SZ];
{
        int     x,
                y;

        for ( x=0; x<BLK_SZ; x++ )
                for ( y=0; y<BLK_SZ; y++ )
                        t_block[x][y] = s_block[BLK_MX-y][x];

}

/*=============================================================*/
```

Header:

```
extern void ac_compress();
extern void ac_decompress();
```

Source:

```
/*============================================================
```

Module: ARITMETIC

Program: Block Oriented Fractal Data Compression of Digital Images.

Programmer: Larry M. Wall
Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Canada
(larwall@ee.umanitoba.ca)

Version: 1.0

Last Update: 16/02/93

Comments: This module performs dynamic arithmetic entropy encoding and decoding of FBC paramters.

```
/*------------------------------------------------------------

PRIVATE CONSTANTS:                                          */

#define SBL_NM 2048
#define NEG 1024
#define NDX_NM (SBL_NM + 1)
#define MAX_CUM 32767
#define NM_Code_Bits 17
#define EOF_SBL MAXINT

#define NM_SZ 20
#define SUCCESS 1
#define FAILURE 0

#define TOP ((( unsigned long int )1<<NM_Code_Bits)-1)
#define QTR (TOP/4+1)
#define HALF (2*QTR)
#define THREE_QTR (3*QTR)
```

```
#define FLD_NM 5

/*-------------------------------------------------------------

LIBRARIES:                                                    */

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

#include "fbc_constants.h"
#include "fbc_arithmetic.h"

/*-------------------------------------------------------------

PUBLIC FUNCTIONS:                                             */

void ac_compress();
void ac_decompress();

/*-------------------------------------------------------------

PRIVATE FUNCTIONS:                                            */

void  init_stats();
void update_stats();

void  put_bit();
void  close_bit_out();

void  close_bit_in();

void compress();
void  init_encoder();
void encode_smb();
void  bit_plus_follow();
void  flush_encoder();

void decompress();
void  init_decoder();

void open_data_in();
void close_data_in();

/*-------------------------------------------------------------
```

```
DATA STRUCTURES:                                                    * /

typedef struct {
        int     sbl_nm,
                ndx_nm;
        int     *index,
                *symbl;
        unsigned long int
                *prb,
                *cum;
        } statistics;

/*============================================================

: Initialize the statistical model.                                 * /

void init_stats( stats, sbls )
statistics      *stats;
int             sbls;
{

        int     c;

        stats->sbl_nm = sbls;
        stats->ndx_nm = sbls + 1;

        stats->index = (int *) calloc( stats->ndx_nm, sizeof(int) );
        stats->symbl = (int *) calloc( stats->ndx_nm, sizeof(int) );
        stats->prb   = (unsigned long int *) calloc( stats->ndx_nm + 1, sizeof(unsigned
long int) );
        stats->cum   = (unsigned long int *) calloc( stats->ndx_nm + 1, sizeof(unsigned
long int) );

        for( c=0; c<stats->ndx_nm; c++ )
        {
                stats->index[c] = c;
                stats->symbl[c] = c;
        }

        stats->cum[0] = 0;
        stats->prb[stats->ndx_nm] = 0;
        for( c=0; c<stats->ndx_nm; c++ )
        {
                stats->prb[c] = 1;
                stats->cum[c] = c;
        }
```

```
        stats->cum[stats->ndx_nm] = c;
}

/*-------------------------------------------------------------

: Update the statistical model to reflect the character coded/received.      */

void update_stats( stats, smb )
statistics      *stats;
int smb;
{
        int     ndx_1,
                ndx_2;

        ndx_1 = stats->index[smb];

        if( stats->cum[stats->ndx_nm] >= MAX_CUM )
                for( ndx_2=0; ndx_2<stats->ndx_nm; ndx_2++ )
                {
                        stats->prb[ndx_2] = ( stats->prb[ndx_2] >> 1 ) + 1;
                        stats->cum[ndx_2+1] = stats->cum[ndx_2] + stats-
>prb[ndx_2];
                } .

        for( ndx_2=ndx_1; stats->prb[ndx_2] == stats->prb[ndx_2+1]; ndx_2++ );

        if( ndx_2>ndx_1 )
        {
                stats->index[smb] = ndx_2;
                stats->index[stats->symbl[ndx_2]] = ndx_1;
                stats->symbl[ndx_1] = stats->symbl[ndx_2];
                stats->symbl[ndx_2] = smb;
        }

        stats->prb[ndx_2]+=2;
        for( ndx_1=(ndx_2+1); ndx_1<=stats->ndx_nm; ndx_1++ )
                stats->cum[ndx_1]+=2;
}

/*============================================================*/

static FILE   *BitFile;

static int    buffer,
              buff_count;

/*-------------------------------------------------------------
```

```
: Open the Arithmetic code output bit stream.                    * /

int open_bit_out( filename )
char filename[NM_SZ];
{
        int     status;

        buffer = 0;
        buff_count = 0;

        BitFile = fopen( filename, "wb" );
        if( BitFile != NULL )
                status = SUCCESS;
        else
                status = FAILURE;
}

/*-----------------------------------------------------------------

: Write a bit to the output stream bit buffer.                   * /

void put_bit( bit )
int bit;
{
        buffer = ( buffer << 1 ) | bit;
        buff_count++;
        if( buff_count == 8 )
        {
                putc( buffer, BitFile );
                buffer = 0;
                buff_count = 0;
        }
}

/*-----------------------------------------------------------------

: Close the arithmetic code output stream.                       * /

void close_bit_out()
{
        buffer <<= ( 8 - buff_count );
        putc( buffer, BitFile );

        fclose( BitFile );
}

/*=================================================================*/
```

```
        static int      garbage_count;

/*-----------------------------------------------------------------------

: Open the arithmetic code input stream.                                  */

int open_bit_in( filename )
char filename[NM_SZ];
{
        int     status;

        buffer = 0;
        buff_count = 0;
        garbage_count = 0;

        BitFile = fopen( filename, "rb" );
        if( BitFile != NULL )
                status = SUCCESS;
        else
                status = FAILURE;
}

/*-----------------------------------------------------------------------

: Get a bit from the arithmetic input stream bit buffer.                  */

int  get_bit()
{
        int  bit;

        if( buff_count <= 0 )
        {
                if( !feof( BitFile ) )
                {
                        buffer = getc( BitFile );
                        buff_count = 8;
                }
                else
                {
                        buffer = 0;
                        garbage_count++;
                        if( garbage_count > ( NM_Code_Bits - 2 ) )
                        {
                                puts( "Bad Source Bit File." );
                                exit(-1);
                        }
```

```
            }
        }

        bit = ( buffer & 0x80 ) >> 7;
        buffer <<= 1;
        buff_count--;
        return  bit;

}

/*-------------------------------------------------------------

: Close the input bit stream.                                    */


void  close_bit_in()
{
        fclose( BitFile );
}

/*=============================================================*/

static unsigned long int
                low,
                high;
static int      follow_bits;

/*-------------------------------------------------------------

: Compress FBC paramters using arithmetic encoding.              */

void ac_compress( FileName, fr_code )
char  FileName[NM_SZ];
fractal  fr_code[BLK_NM][BLK_NM];
{
        int     x,
                blk_x,
                blk_y,
                count;

        statistics      stats[FLD_NM];
        int             offset[FLD_NM];
        int             max[FLD_NM];

        init_stats( &stats[0], IMG_SZ/STP_SZ );
        init_stats( &stats[1], IMG_SZ/STP_SZ );
        init_stats( &stats[2], 512 );
        init_stats( &stats[3], 2048 );
```

```
        init_stats( &stats[4], 8 );

        init_encoder();
        open_bit_out( FileName );

        for( blk_x=0; blk_x<BLK_NM; blk_x++ )
                for( blk_y=0; blk_y<BLK_NM; blk_y++ )
                {
                        encode_smb( stats[0], fr_code[blk_x][blk_y].x );
                        update_stats( &stats[0], fr_code[blk_x][blk_y].x );
                        encode_smb( stats[1], fr_code[blk_x][blk_y].y );
                        update_stats( &stats[1], fr_code[blk_x][blk_y].y );
                        encode_smb( stats[2], fr_code[blk_x][blk_y].translate );
                        update_stats( &stats[2], fr_code[blk_x][blk_y].translate );
                        encode_smb( stats[3], fr_code[blk_x][blk_y].scale );
                        update_stats( &stats[3], fr_code[blk_x][blk_y].scale );
                        encode_smb( stats[4], fr_code[blk_x][blk_y].transform );
                        update_stats( &stats[4], fr_code[blk_x][blk_y].transform );
                }

        flush_encoder();
        close_bit_out();

}

/*-------------------------------------------------------------

: Initialize the arithmetic encoder.                          */

void init_encoder()
{
        low = 0;
        high = TOP;
        follow_bits = 0;
}

/*-------------------------------------------------------------

: Encode a single symbol.                                     */

void encode_smb( stats, smb )
statistics      stats;
int smb;
{

        int     ndx;
        unsigned long int
```

```
                range;

        ndx = stats.index[smb];

        range = ( high - low ) + 1;
        high = low + ( range * stats.cum[ndx+1] ) / stats.cum[stats.ndx_nm] - 1;
        low += ( range * stats.cum[ndx] ) / stats.cum[stats.ndx_nm];

        while( ( high < HALF ) || ( low >= HALF ) )
        {
                if( high < HALF )
                        bit_plus_follow(0);
                else
                {
                        bit_plus_follow(1);
                        low -= HALF;
                        high -= HALF;
                }
                low <<= 1;
                high = ( high << 1 ) + 1;
        }
        while( (low>=QTR) && (high<THREE_QTR) )
        {
                follow_bits++;
                low -= QTR;
                low <<= 1;
                high -= QTR;
                high = ( high << 1 ) + 1;
        }
}

/*-------------------------------------------------------------

: Send a bit to the output bit stream.                          */

void bit_plus_follow( bit )
int bit;
{
        put_bit( bit );
        while( follow_bits>0 )
        {
                put_bit( !bit );
                follow_bits--;
        }
}

/*-------------------------------------------------------------
```

```
: Send all remaining bits in the encoder to the output bit stream.                * /

void  flush_encoder()
{
        follow_bits++;
        if( low<QTR )
                bit_plus_follow(0);
        else
                bit_plus_follow(1);
}

/*============================================================*/

static unsigned long int value;

/*------------------------------------------------------------

: Recover FBC paramters from an arithmetic code stream.                * /

void ac_decompress( FileName, fr_code )
char  FileName[NM_SZ];
fractal  fr_code[BLK_NM][BLK_NM];
{
        int     blk_x,
                blk_y;

        statistics      stats[FLD_NM];

        init_stats( &stats[0], IMG_SZ/STP_SZ );
        init_stats( &stats[1], IMG_SZ/STP_SZ );
        init_stats( &stats[2], 512 );
        init_stats( &stats[3], 2048 );
        init_stats( &stats[4], 8 );

        open_bit_in( FileName );

        init_decoder();

        for( blk_x=0; blk_x<BLK_NM; blk_x++ )
                for( blk_y=0; blk_y<BLK_NM; blk_y++ )
                {
                        fr_code[blk_x][blk_y].x = decode_smb( stats[0] );
                        update_stats( &stats[0], fr_code[blk_x][blk_y].x );
                        fr_code[blk_x][blk_y].y = decode_smb( stats[1] );
                        update_stats( &stats[1], fr_code[blk_x][blk_y].y );
                        fr_code[blk_x][blk_y].translate = decode_smb( stats[2] );
                        update_stats( &stats[2], fr_code[blk_x][blk_y].translate );
                        fr_code[blk_x][blk_y].scale = decode_smb( stats[3] );
```

```
                    update_stats( &stats[3], fr_code[blk_x][blk_y].scale );
                    fr_code[blk_x][blk_y].transform = decode_smb( stats[4] );
                    update_stats( &stats[4], fr_code[blk_x][blk_y].transform );

            }

        close_bit_in();

}

/*-------------------------------------------------------------

: Initialize the arithmetic decoder and fill the operating register with the first
        NM_Code_Bits from the input bit stream.                                  */

void  init_decoder()
{
        int  i;

        value = 0;
        for( i=0; i<NM_Code_Bits; i++ )
                value = 2 * value + get_bit();

        low=0;
        high=TOP;
}

/*-------------------------------------------------------------

: Decode a single symbol from the input bit stream.                             */

int decode_smb( stats )
statistics  stats;
{

        unsigned long int
                range;
        int     v_cum,
                ndx,
                smb;

        range = ( high - low ) + 1;
        v_cum = (int) ( ( ( ( value - low ) + 1 ) * stats.cum[stats.ndx_nm] - 1 ) /
range );

        for( ndx = stats.ndx_nm; stats.cum[ndx]>v_cum; ndx-- );
```

```
        high = low + ( range * stats.cum[ndx+1] ) / stats.cum[stats.ndx_nm] - 1;
        low += ( range * stats.cum[ndx] ) / stats.cum[stats.ndx_nm];

        while( ( high < HALF ) || ( low >= HALF ) )
        {
                if( low >= HALF )
                {
                        value -= HALF;
                        low -= HALF;
                        high -= HALF;
                }
                low <<= 1;
                high = ( high << 1 ) + 1;
                value = ( value << 1 ) + get_bit();
        }
        while( (low>=QTR) && (high<THREE_QTR) )
        {
                value -= QTR;
                value = ( value << 1 ) + get_bit();
                low -= QTR;
                low <<= 1;
                high -= QTR;
                high = ( high << 1 ) + 1;
        }

        smb = stats.symbl[ndx];

        return smb;

}

/*============================================================*/
```

Header:

```
extern unsigned char *img_alloc();
extern fractal *fcd_alloc();
extern vector *cbk_alloc();
extern index *ndx_alloc();

extern int img_load();
extern int img_save();
extern void fcd_save();
extern void  fcd_load();
extern int cbk_load();
extern int cbk_save();
```

Source:

```
/*============================================================

Module:       IO

Program:      Block Oriented Fractal Data Compression of Digital
              Images.

Programmer:   Larry M. Wall
              Department of Electrical and Computer Engineering
              University of Manitoba
              Winnipeg, Canada
              (larwall@ee.umanitoba.ca)

Version:      1.0

Last Update   07/06/92

Comments:     Input/Output and Memory Allocation Routines, and Type
              Declarations for Block Oriented Fractal Data Compression
              Program.

/*------------------------------------------------------------

LIBRARIES:                                                    */

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

#include "fbc_constants.h"
```

```
#include  "fbc_io.h"

/*-------------------------------------------------------------

PRIVATE CONSTANTS:                                              */

#define SUCCESS 1
#define FAILURE 0
#define NM_SZ 20

/*-------------------------------------------------------------

PUBLIC FUNCTIONS:                                               */

unsigned char *img_alloc();
fractal       *fcd_alloc();
vector        *cbk_alloc();
index         *ndx_alloc();

int   img_load();
int   img_save();
void  fcd_save();
void  fcd_load();
int   cbk_load();
int   cbk_save();

/*=============================================================

: Dynamically allocate memory for image array                   */

unsigned char *img_alloc()
{

        unsigned char *ptr;

        ptr = (unsigned char *) calloc( IMG_SZ * IMG_SZ, sizeof(unsigned char) );
        if ( !ptr )
                puts( "Memory Allocation Error." );

        return  ptr;

}

/*-------------------------------------------------------------

: Dynamically allocate memory for fractal code                  */

fractal  *fcd_alloc()
```

```
{

        fractal  *ptr;

        ptr = (fractal *) calloc( BLK_NM * BLK_NM, sizeof(fractal) );
        if ( !ptr )
                puts( "Memory Allocation Error." );

        return  ptr;

}

/*-------------------------------------------------------------

: Dynamically allocate memory for codebook array                 */

vector  *cbk_alloc()
{

        vector  *ptr;

        ptr = (vector *) calloc( CBK_SZ, sizeof(vector) );
        if ( !ptr )
                puts( "Memory Allocation Error." );

        return  ptr;

}

/*-------------------------------------------------------------

: Dynamically allocate memory for coded image array              */

index  *ndx_alloc()
{

        index  *ptr;

        ptr = (index *) calloc( BLK_NM * BLK_NM, sizeof(index) );
        if ( !ptr )
                puts( "Memory Allocation Error." );

        return  ptr;

}

/*-------------------------------------------------------------
```

```
: Load image to be compressed into memory.                              * /

int img_load( filename, image )
char  filename[NM_SZ];
unsigned char image[IMG_SZ][IMG_SZ];
{

        int     status;

        FILE   *InFile;

        InFile = fopen( filename, "rb" );
        if ( InFile != NULL )
        {
                if ( fread( image, sizeof(unsigned char), IMG_SZ * IMG_SZ, InFile ) )
                        status = SUCCESS;
                else
                {
                        puts( "File Read Error." );
                        status = FAILURE;
                }
                fclose( InFile );
        }
        else
        {
                puts( "File Not Found." );
                status = FAILURE;
        }

        return  status;

}

/*-------------------------------------------------------------

: Save reconstructed fractal image.                                     * /

int img_save( filename, image )
char  filename[NM_SZ];
unsigned char image[IMG_SZ][IMG_SZ];
{

        int     status;

        FILE   *OutFile;

        OutFile = fopen( filename, "wb" );
```

```
        if ( OutFile != NULL )
        {
                if ( fwrite( image, sizeof(unsigned char), IMG_SZ * IMG_SZ, OutFile ) )
                        status = SUCCESS;
                else
                {
                        puts( "File Read Error." );
                        status = FAILURE;
                }
                fclose( OutFile );
        }
        else
        {
                puts( "File Not Found." );
                status = FAILURE;
        }

        return status;

}

/*-------------------------------------------------------------

: Load fractal code for image.                                  */

void fcd_load( filename, code )
char filename[NM_SZ];
fractal code[BLK_NM][BLK_NM];
{

        FILE *InFile;

        InFile = NULL;
        InFile = fopen( filename, "rb" );
        if ( InFile != NULL )
                fread( code, sizeof(fractal), (BLK_NM * BLK_NM), InFile );
        else
        {
                puts( "File Not Found." );
                exit(3);
        }

        fclose(InFile);

}
/*-------------------------------------------------------------

: Save fractal code for image.                                  */
```

```
void fcd_save( filename, code )
char  filename[NM_SZ];
fractal  code[BLK_NM][BLK_NM];
{

        FILE *OutFile;

        OutFile = NULL;
        OutFile = fopen( filename, "wb" );
        if ( OutFile != NULL )
                fwrite( code, sizeof(fractal), (BLK_NM * BLK_NM), OutFile );
        else
        {
                puts( "Unable to Open File." );
        }

        fclose(OutFile);

}

/*-------------------------------------------------------------

: Load the codebook from disk.                                  */

int cbk_load( filename, cbook )
char  filename[NM_SZ];
vector cbook[CBK_SZ];
{

        int     status;

        FILE    *InFile;

        InFile = fopen( filename, "rb" );
        if ( InFile != NULL )
        {
                if ( fread( cbook, sizeof(vector), CBK_SZ , InFile ) )
                        status = SUCCESS;
                else
                {
                        puts( "File Read Error." );
                        status = FAILURE;
                }
                fclose( InFile );
        }
        else
```

```
        {
                puts( "File Not Found" );
                status = FAILURE;
        }

        return status;

}

/*-------------------------------------------------------------------

: Save the codebook to disk.                                          */

int cbk_save( filename, cbook )
char filename[NM_SZ];
vector cbook[CBK_SZ];
{

        int     status;

        FILE    *OutFile;

        OutFile = fopen( filename, "wb" );
        if ( OutFile != NULL )
        {
                if ( fwrite( cbook, sizeof(vector), CBK_SZ , OutFile ) )
                        status = SUCCESS;
                else
                {
                        puts( "File Write Error." );
                        status = FAILURE;
                }
                fclose( OutFile );
        }
        else
        {
                puts( "Cannot Open Output File." );
                status = FAILURE;
        }

        return status;

}

/*=========================================================*/
```