IMAGE COMPRESSION WITH DENOISED REDUCED-SEARCH FRACTAL BLOCK CODING

by

Shamit Bal

A Thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg, Manitoba

(x + 174 = 184 pp.)

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

IMAGE COMPRESSION WITH DENOISED REDUCED-SEARCH

FRACTAL BLOCK CODING

BY

SHAMIT BAL

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University

of Manitoba in partial fulfillment of the requirements of the degree

of

MASTER OF SCIENCE

ABSTRACT

The research reported in this thesis improves the efficiency of the reduced-search fractal block coding algorithm of greyscale images. The problem addressed here is that additive noise increases the first-order entropy of the image. The increased entropy is equivalent to a lower level of redundancy in the image, and therefore a lower efficiency of the reduced-search fractal block algorithm. This thesis examines a technique of reducing the first-order entropy with a minimum distortion of the signal itself. This is in contrast to spatial filtering techniques such as smoothing which affect not only the noise but also the signal. The technique used in this thesis is called wavelet denoising. Wavelet denoising involves performing the forward discrete wavelet transform of the image, reducing the coefficients by some specified threshold, and performing the inverse discrete wavelet transform. The resulting image, with a lower entropy, is then coded by the reduced-search fractal block compression algorithm. This approach increases the compression ratio by 9.1% to 11% with an acceptable image reconstruction quality.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. W. Kinsner, for giving me the opportunity and the guidance to complete the Master of Science degree in an exciting field of research. I would also like to thank Telecommunications Research Labs (TR*Labs*) for providing an excellent working environment, in both equipment and atmosphere, and financial support. TR*Labs*' assistance in starting a new career is also greatly appreciated.

During most of the time spent on this thesis, I had opportunities to discuss ideas with Epiphany Vera which I found very useful. Also, I extend my thanks to Martin Jacobs for helping the weekends working at TR*Labs* go quickly and giving me lifts to bus stops.

I thank my family for their support and staying out of my way when I needed to get things done (most of the time) and for allowing me to use their vehicles. Finally, I would like to show my appreciation to Ilo, Kurt, and Heather who gave me much needed diversions and helped me keep my spirits up enough to finish and to look forward to the future.

TABLE OF CONTENTS

Abstract ii

Acknowledgments iii

Table of Contents iv

List of Figures vi

List of Abbreviations x

Chapter 1 Introduction 1

1.1 Background and Motivation 1
1.2 Objectives 4
1.3 Summary of Thesis 4

Chapter 2 Theoretical Background 6

2.1 Self-Similarity and Self-Affinity 7
2.2 Scale, Measurement, and Dimension 9
2.3 Fractals in Nature 15
2.4 Metric Spaces and Metrics 16
2.5 Affine Transforms 20
2.6 The Contraction Mapping Principle and the Collage Theorem 20
2.7 Iterated Function Systems 22
2.8 Frequency Sensitive Competitive Learning 26
2.9 Vector Spaces 29
2.10 Orthonormal Bases 30
2.11 The Wavelet Transform 31
2.12 Summary 37

Chapter 3 Fractal Block Coding 39

3.1 Generalized Fractal Block Coding 39
3.2 Reduced-Search Fractal Block Coding 42
3.3 Arithmetic Entropy Encoding 44
3.4 Summary 47

Chapter 4 Denoising Images 49

4.1 Spatial Smoothing 50
4.2 Wavelet Denoising 51
4.3 Summary 54

Chapter 5 Implementation of Denoised Fractal Block Coding 55

5.1 Software Organization 56
5.2 Development Environment 57
5.3 Implementation of Experiments 58
5.4 Definition of Compression Ratio and Quality 60
5.5 Summary 61

Chapter 6 Experimental Results and Discussion 63
6.1 The Source Images 63
6.2 Fractal Block Coding Without Pre-Processing 67
6.3 Fractal Block Coding With Pre-Processing 74
6.3.1 Spatial Filtering 74
6.3.2 Wavelet Denoising 78
6.4 Scale Invariance Of Reconstructed Images 87
6.5 Summary 87

Chapter 7 Conclusions and Recommendations 89

References 93
Appendix A Additional Images and Plots 97
Appendix B Software Structure 107
Appendix C Source Code for Denoised Reduced-Search FBC 111
Appendix D Source Code for Solving the Inverse Problem of IFS 170

LIST OF FIGURES

Fig. 2.1. The construction of the Koch curve is initiated by a single line segment and generated by the four line segments [Mand83]. 8

Fig. 2.2. Measuring the perimeter of a circle. 10

Fig. 2.3. The coast of Britain. 11

Fig. 2.4. (a) Plot of coastline length vs. Length of measurement device. (b) Plot of log of coastline length vs. log of measurement precision (after [Kins94b]). 13

Fig. 2.5. The fractal nature of a mountain range. 16

Fig. 2.6. (a) Greyscale image of *Lena*. (b) Three-dimensional representation of *Lena* using greyscale values as altitudes (Image produced with Pov-Ray 3.0 [Youn96].). 17

Fig. 2.7. Self-similar portions of *Lena* [Fish94]. 18

Fig. 2.8. Sierpinski Triangle. 23

Fig. 2.9. Barnsley's fern. 24

Fig. 2.10. Frequency sensitive competitive learning neural network. 28

Fig. 2.11. A few examples of Daubechies wavelets: (a) Daubechies-4, (b) Daubechies-8, (c) Daubechies-12, (d) Daubechies-20 31

Fig. 2.12. Sub-band decomposition of an image [Lang96]. 35

Fig. 2.13. Forward (a) and inverse (b) discrete wavelet transform (after [ABMD92]). 36

Fig. 3.1. The eight isometric transforms of fractal block coding. 41

Fig. 3.2. Reduced-search fractal block coding. 43

Fig. 3.3. Arithmetic entropy encoding of the string "EAT" with an end of file character. The interval returned is [0.664, 0.6664). 46

Fig. 3.4. Arithmetic entropy decoding of the interval [0.664, 0.6664) [Kins91]. 47

Fig. 4.1. Filter mask of odd dimension n. The pixel underneath the centre element x is replaced with average of the pixel values that lie beneath the mask. 50

Fig. 4.1. Wavelet denoising of an image with a threshold of 2.4×10^{-5}. 53

Fig. 5.1. The six modules of the reduced-search fractal block coding software. 56

Fig. 5.2. Structure of wavelet denoising portion of the software. 57

Fig. 5.3. The Graphical User Interface (GUI) for the reduced-search fractal block coding program. 58

Fig. 5.4. Flow of experiments. 59

Fig. 6.1. The image *Lena* . (Shown at 84.3% original size.) 64

Fig. 6.2. The image *Goldhill.* (Shown at 84.3% original size.) 65

Fig. 6.3. The image *Peppers* . (Shown at 84.3% original size.) 66

Fig. 6.4. Domain block sampling. 69

Fig. 6.5. The incremental reconstruction of *Lena* . The images are at 42% of the original size. 72

Fig. 6.6. Reconstructed *Lena* with PSNR = 31.25 dB and compression ratio = 17.4:1. No pre-processing was used. This image is shown at 84.3% size. 73

Fig. 6.7 (a) Filtered with 3x3 mask. 74

Fig. 6.7. (b) Filtered with 7x7 mask. 75

Fig. 6.7. (c) Filtered with 11x11 mask. 75

Fig. 6.8. Reconstructed image of *Lena* with 3x3 filtering. The PSNR = 30.49 dB and the compression ratio was 17.7:1. The image is shown at 50% actual size. 78

Fig. 6.9. Entropy trend of the image *Lena* after wavelet coefficient shrinking. 80

Fig. 6.10. Reconstruction quality trend with respect to the pre-processed image of *Lena* after wavelet coefficient shrinking. 81

Fig. 6.11. Reconstruction quality trend with respect to the original image of *Lena* after wavelet coefficient shrinking. 82

Fig. 6.12. Compression ratios of *Lena* after wavelet coefficient shrinking. 83

Fig. 6.13. Reconstructed image of *Lena* at T=0.006. PSNR = 30.42 dB; CR = 18.8:1. (Shown at 84.3% original size.) 84

Fig. 6.14. Reconstructed image of *Lena* at T=0.015. PSNR = 30.14 dB; CR = 19.0:1. (Shown at 84.3% original size.) 85

Fig. 6.15. Reconstructed image of *Lena* at t=0.020. PSNR = 29.92 dB; CR =19.1:1. (Shown at 84.3% original size.) 86

Fig. 6.16. A portion of *Lena* 's shoulder enlarged four times in (a) from the original image and (b) through the reconstruction procedure. 87

Fig. A.1. Reconstructed *Goldhill* with PSNR = 29.46 dB and CR = 17.6:1. No pre-processing was used. (Shown at 84.3% original size.) 97

Fig. A.2. Reconstructed *Peppers* with PSNR = 32.79 dB and CR = 17.5:1. No pre-processing was used. (Shown at 84.3% original size.) 98

Fig. A.3. Entropy trend of the image *Goldhill* after wavelet coefficient shrinking. 99

Fig. A.4. Entropy trend of the image *Peppers* after wavelet coefficient shrinking. 99

Fig. A.5. Reconstruction quality trend with respect to the wavelet coefficient reduced image of *Goldhill.* 100

Fig. A.6. Reconstruction quality trend with respect to the to the wavelet coefficient reduced image of *Peppers.* 100

Fig. A.7. Reconstruction quality trend with respect to the original image of *Goldhill* after wavelet coefficient shrinking. 101

Fig. A.8. Reconstruction quality trend with respect to the original image of *Peppers* after wavelet coefficient shrinking. 101

Fig. A.9. Compression ratios of *Goldhill* after wavelet coefficient shrinking. 102

Fig. A.10. Compression ratios of *Peppers* after wavelet coefficient shrinking. 102

Fig. A.11. Root mean square error of the reconstruction with respect to the wavelet coefficient reduced image of *Lena.* 103

Fig. A.12. Root mean square error of the reconstruction with respect to the wavelet coefficient reduced image of *Goldhill.* 103

Fig. A.13. Root mean square error of the reconstruction with respect to the wavelet coefficient reduced image of *Peppers*. 104

Fig. A.14. Reconstructed image of *Goldhill* at T=0.006. PSNR = 29.34 dB; CR = 17.6:1. (Shown at 84.3% original size.) 105

Fig. A.15. Reconstructed image of *Peppers* at T=0.006. PSNR = 32.53 dB; CR = 17.6:1. (Shown at 84.3% original size.) 106

Fig. B.1. Main function hierarchy. 107

Fig. B.2. Train codebook function hierarchy. 108

Fig. B.3. Compress image function hierarchy. 109

Fig. B.4. Reconstruct image function hierarchy. 110

LIST OF ABBREVIATIONS

CAT	Contractive Affine Transformation
CR	Compression Ratio
DCT	Discrete Cosine Transform
DEC	Digital Equipment Corporation
DWT	Discrete Wavelet Transform
FBC	Fractal Block Code or Fractal Block Coding
FSCL	Frequency Sensitive Competitive Learning
IDWT	Inverse Discrete Wavelet Transform
IFS	Iterated Function Systems
JPEG	Joint Photographic Expert Group
MHz	Megahertz (1 million cycles per second)
PSNR	Peak Signal-to-Noise Ratio
rms	root mean square
VQ	Vector Quantization

CHAPTER 1
INTRODUCTION

1.1 Background and Motivation

The compression of still images is necessary for present day methods of communication and data storage, so the problem of improving image compression algorithms is always an issue in compression research. Fractal compression has recently been recognized as a viable method for image compression. Stemming from the work involving iterated function systems [Barn88], and later, fractal block coding [Jacq92], reduced-search fractal block coding [Wall93] evolved. This thesis extends that work in order to increase the efficiency of the compression algorithm through the use of wavelets. First, however, an introduction to compression would be useful.

The nature of data compression is to reduce or remove redundancies and irrelevancies in the data. There are two forms of compression, lossless and lossy. In lossless compression, only redundancies are removed so that the compressed form has the smallest number of bits without any loss of information. Information is defined by Shannon entropy, which determines the number of bits that can be expected per symbol. In the case of greyscale images, a symbol is a single grey level. A higher entropy implies fewer redundancies which is less susceptible to lossless compression. Lossless compression of images often results in relatively low compression ratios because the images to be compressed quite often have high entropy. However, in applications where the image is viewed by humans, the entropy can be reduced by reducing or removing irrelevant data so that there is no perceptual change because the human brain is capable of interpolating the missing information. Compression that uses this technique is classified as lossy. It can be argued, however, if useful signal can be separated from unwanted noise, that it can be considered *pseudo*-lossless, if the unwanted noise is contained within one

quantization step [LaKi96]. For example, if two 8-bit greyscale images are compared pixel by pixel and the difference is one grey level or less, the process in considered lossless.

Vector quantization and transform coding are two principal classes of lossy image compression. In vector quantization, image samples are quantized into a finite set of image samples. In transform coding, the image is transformed into a domain where the image is decomposed into its constituent frequencies or frequency plus temporal components.

Of the transform coding class, one of the current standards for image compression is JPEG (Joint Photographic Experts Group). In the baseline JPEG algorithm [Teka95], the image is first subdivided into 8 x 8 blocks. Each block is then transformed into the cosine domain through the use of the discrete cosine transform (DCT) which yields a set of coefficients that are quantized and recorded in a zigzag fashion to obtain a one-dimensional signal. This signal is then coded by the Huffman compression algorithm. Currently, work is being done to use the discrete wavelet transform (DWT) [Mall89] instead of the DCT in JPEG. In this case, the DWT yields frequency plus temporal components of the image. Another class of lossy compression is vector quantization.

Vector quantization involves subdividing the image into blocks, or vectors. Each vector is classified by finding, in another set of vectors, the best matching vector. The index of the vector in the set is used to replace the original vector. The set of indices representing the image vectors is a much more compact way of representing the image. The image is reconstructed by using the vector in the set that is pointed to by the index of each vector in the original image. While transform coding and vector quantization are effective means of compressing images, fractal compression has also received a great deal of focus in the research of image compression.

Fractal image compression is a technique in which the image is treated as being a fractal or collage of fractals, where a fractal is characterized as self-similar because portions of that image are transformed copies of other portions the image. If a real-world image can be treated the same way, a very compact representation of the image can be generated.

The earliest form of fractal image compression, which was developed by Barnsley [Barn88], [Barn93], took a complex binary image and represented it with a short set of transform functions. This set and the method that applies the set to regenerate the image is called an iterated function system, which will be described later in this work. Barnsley applied this method to real world images and obtained extremely high compression ratios. However, automatically determining the correct set of transform functions that will reconstruct any given image is an NP complete problem. In Barnsley's approach, a human operator would provide the computer with points in the image that can be used to generate the transform functions. This is not practical if it is desired to have a fully automatic, stand-alone process.

A student of Barnsley's, Jaquin [Jacq92], [Jacq93], applied the concept of iterated function systems to develop an automatic process called fractal block coding. While the results were not as good as Barnsley's, the process was entirely automatic. The process worked by scanning through the entire image and finding pairs of blocks, where one could be transformed into the other. However, the act of searching the entire image for a best matching pair is a very time intensive procedure and is of a complexity $O(n^4)$. The process was inefficient because many hours would be spent to search thousands of blocks for just one block that best matched, via a transform, another given block. The efficiency could be improved if the number of blocks to be searched could be reduced.

The number of blocks to be searched can be reduced by pre-classifying the blocks . In a technique called *reduced-search fractal block coding*, developed by Kinsner and Wall [KiWa93] and [Wall93], only the blocks of the same class as the block before the transform are examined for the possibility of being the best match. The reduced search resulted in a complexity of $O(n^3)$ as opposed to Jacquin's $O(n^4)$. The blocks are classified by matching them to one of a set of vectors which is trained with a frequency sensitive competitive learning (FSCL) neural network. The resulting fractal code is also further

compressed using a lossless technique called arithmetic entropy encoding [WiNC87] which allows for a 20% better compression than the fractal code alone.

A problem that exists with the compression of images is that images are generally contaminated with some kind of noise which is often encountered during the digitization process but can also become evident after digital manipulation of the image following the digitization. Noise, being random and unpredictable, raises the first-order entropy of the image. Higher entropy generally indicates a lower relative number of redundancies. For that reason, compression techniques that exploit the inherent redundancies of the image perform poorly. The noise can be reduced but problems exist in the common noise reduction techniques. Most noise reduction can be achieved by a filter that affects the pixels of the image to make the noise less prominent. In doing so, however, the pixels that compose the signal that we wish to preserve are distorted.

1.2 Objectives

This thesis studies the reduced-search fractal block coding algorithm, with the addition of a pre-processing stage called wavelet denoising that does not filter the signal but attempts to separate the desired signal from the noise. Since data that is not part of the signal is removed, wavelet denoising is considered to be a lossless technique. By reducing the amount of noise, the entropy of the signal is reduced. This effectively increases the level of redundancy in the image, giving it more fractal characteristics. If the image is perfectly self-similar, every portion of the image would be one of a finite set of transformations of another portion. While this is not the case in real-world images, reducing the entropy moves the image closer to that state. With the addition of the wavelet denoising stage, the efficiency of the reduced-search fractal block coding is increased by as much as 9.1% to 11%.

1.3 Summary of Thesis

The chapters of this thesis are broken down as follows. Chapter 2 gives a theoretical background to fractals, affine transformations, and wavelets. Chapter 3 provides a description of how reduced-search fractal block coding works, complete with the post-processing stage of arithmetic entropy encoding. Chapter 4 describes the two techniques of pre-processing the images that were used in this work. While, wavelet denoising is the focus, spatial filtering was used to offer a comparison. The implementation of reduced-search fractal block coding with and without pre-processing the images is described in Chapter 5. Chapter 6 describes the results of the experiments and provides some discussion. Here, the effects of wavelet denoising and spatial filtering on reduced-search fractal block coding are compared. Finally, Chapter 7 states some conclusions and recommendations for future work.

CHAPTER 2

THEORETICAL BACKGROUND

This chapter is intended to provide a theoretical background to all the work presented in this thesis. Fractal coding takes advantage of the self-similar or self-affine nature of our natural environment. In light of this, a discussion of classical fractals is necessary with the discussion beginning with the description of self-similarity and self-affinity. Following that is a discussion of the concepts of scale, measurement, and dimension which establish a definite method of measuring and analyzing fractals so that fractals need not be as complex as they appear to be. The analysis is instrumental in developing a deterministic algorithm to construct fractals. The purpose of this is to show that complex fractal images can be constructed and easily repeated. If the fractals can be represented in a compact form, then by representing natural objects as being constructed with fractals they can also be represented in a compact form. The discussion also includes metric spaces, affine transforms, the collage theorem, the contraction mapping principle, and iterated function systems.

In order for fractal coding to work well, the system performing the procedure must have the ability to identify similarities between two regions of the image so that a contractive transform can be designed that maps one of the regions on to the other. This identification has been shown to be performed well with an artificial neural network. In that respect, a brief background on one type of artificial neural network, the frequency sensitive competitive learning neural network, is given.

Having established the nature of fractals, the chapter continues by describing how the wavelet transform works. The wavelet transform is used to denoise the images in preparation for fractal coding. Additive noise reduces the efficiency of fractal coding techniques. The development of the theoretical background of the wavelet transform is supported with a discussion of vector spaces and orthonormal bases.

2.1 Self-Similarity and Self-Affinity

The redundancy that exists in a natural object can be characterized as being *self-similar*. To illustrate self-similarity, take a simple line segment, called the *initiator*, and replace it with another set of line segments arranged in a specific way, called the *generator*, as shown in Fig. 2.1. Then take each of those line segments and replace them with the generator but scaled and rotated to match the orientation of each segment. The process can continue an infinite number of times, resulting in an object of infinite complexity. At any given scale, the object is made up of scaled down replicas of the whole object making it strictly self-similar everywhere i.e. it has the same scale in all directions. The characteristic where the amount of detail seen does not change no matter what the scale of observation is called *scale invariance*. The object shown in Fig. 2.1 is an example of a fractal called the Koch curve [Mand77].

While the Koch curve is strictly self-similar everywhere, all fractals do not have to be. There can be varying degrees of self-similarity to the point where the scaled down replicas no longer resemble the original object but are still related to the whole object through an affine transformation. Objects with this kind of characteristic have more than one scaling direction, and are called *self-affine* . The transformation is a one-to-one mapping from an object in n-dimensional space on to the same space. The transformation, w, called an affine transformation, in the n-dimensional space of real numbers, is written as

$$w : \Re^n \rightarrow \Re^n. \tag{2.1}$$

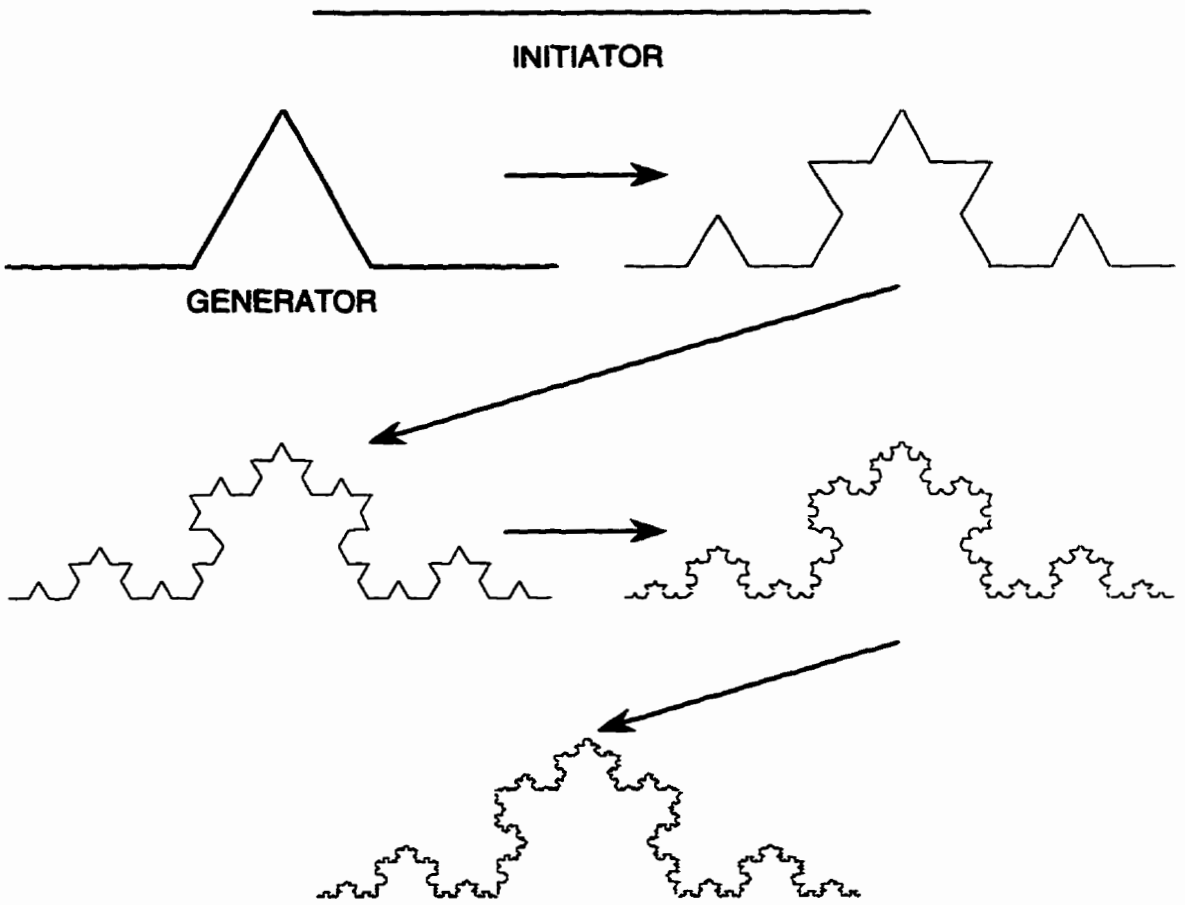


Fig. 2.1. The construction of the Koch curve is initiated by a single line segment and generated by the four line segments [Mand83].

where $\Re$ is the space of real numbers. A fractal object may be constructed with a collage of any number of affine transformations. This has the potential of generating fractal objects with a variety of detail and textures. In this manner, fractals that resemble natural, real-world objects can be deterministically constructed.

At this point, we have a simple method for generating self-affine or self-similar fractals which will be instrumental in the reconstruction of fractal coded images. In both cases, relatively complex images can be constructed from a very compact description which results in extremely high compression ratios. While the generated fractals appear complex, they can be measured and a value of how self-similar they are can be obtained. This

requires a discussion of how the scale at which an object is measured affects the total overall measurement.

2.2 Scale, Measurement, and Dimension

A comprehensive and unified review of fractal dimensions has been presented by Kinsner [Kins94a]. This section summarizes the key ideas pertinent to this thesis. When one wishes to obtain the length of a simple, finite geometrical object such as a rectangle, one simply uses a ruler or other measuring device and measures the length of one side. The length obtained is accurate to within some margin of error that is dependent on the measuring device used. To measure the perimeter, the sum of the lengths of all four sides is taken. This method is simple for objects with straight sides. However, for an object such as a circle, the task is a little more complicated. A possible approach would be to trace the circle with thread or other flexible tool and then measure the length of the thread to obtain the length of the circumference of the circle. Another method would be to inscribe the circle with a polygon and measure the length of the perimeter of the polygon by taking the sum of the lengths of each side as shown in Fig. 2.2. This results in an approximation of the length of the circumference of the circle. By decreasing the length of each side, thus increasing the number of sides, the approximation would approach the actual value of the circumference. The length of a side of the polygon indicates the scale at which the measurement is taken. At smaller scales, the accuracy of the total length increases. However, to apply this method to fractal objects, where the amount of detail does not change as the scale decreases, problems arise.

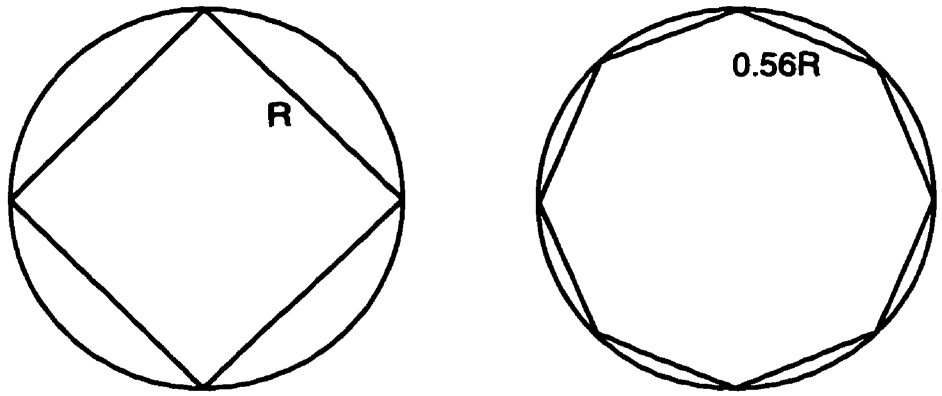


Fig. 2.2. Measuring the perimeter of a circle.

A famous question posed by Mandelbrot [Mand83], "How long is the coast of Britain?" seems simple enough but is actually difficult to determine. A portion of a coastline is at least as long as the distance between the two endpoints. However, as shown in Fig. 2.3, the coastline is winding and irregular which would add to the distance already measured between the endpoints. The length of the coastline can be measured by approximating it with an irregular polygon and summing the lengths of the sides of the polygon. In order to approximate the coastline as closely as possible, the lengths of the sides would have to become smaller and smaller to accommodate every indentation and excursion. Unlike that of the circle, the length of the coastline does not converge to a finite value. The amount of detail visible on the coastline depends on the scale at which it is viewed. A portion of coastline viewed at 1/100 000 scale that is reexamined at 1/10 000 scale reveals smaller bays and peninsulas. A portion of *that* portion can be examined again at 1/1000 scale to reveal even more inlets and other irregularities. It is seen from this example that it is impossible to obtain an exact measure of the length of the coastline, that is, the length increases infinitely. The accuracy of the measure depends on the scale at which the coastline is measured.

Fig. 2.3. The coast of Britain.

As noted earlier, a method of measuring the length of the perimeter of a closed polygon is to sum the lengths of the sides. This can be extended by stating that the length of the perimeter of a closed polygon is the sum of the lengths of the sides each raised to the power $D = 1$ which is the dimensional space in which the perimeter of the polygon is embedded. In the same manner, the area enclosed by the polygon can be measured by covering it with squares and summing the lengths of the squares each raised to the power $D = 2$ yielding the sum of the areas of the squares. As with measuring the length of the coastline, the accuracy of the measure increases as the size of the squares decreases and the number of squares increases. When measuring the length of the circumference of a circle, the length, L_r, approaches a finite limit as the measuring distance, r, approaches 0. The length can be calculated by

$$L_r = N_r r \tag{2.2}$$

where N_r is the total number of line segments. It is clear that r is raised to the power 1. Equation 2.2 will not apply to the coastline because L_r approaches infinity as r approaches zero. So, let r be raised to some dimension D_L so that L_r is equal to one.

$$L_r = N_r r^{D_L} = 1, \quad r \to 0. \tag{2.3}$$

Since r can be arbitrarily less than 1, the total length can be set to equal 1. The total length of the coastline with r raised to the power 1.0 is

$$L_r = N_r r = \left(\frac{1}{r}\right)^{D_L} r = \left(\frac{1}{r}\right)^{D_L - 1} \tag{2.4}$$

where the substitution for N_r comes from Eq. 2.3.

By examining the results of making several measurements of the length of the coastline with varying sizes of measurement devices, it can be seen that the measured length of the coastline is inversely proportional to the length of the measurement device as shown in the plot in Fig.2.4a. By plotting the log of the total length vs. the log of the measurement precision, $1/r$, a straight line is plotted as shown in Fig. 2.4b. The slope, p, of the log-log plot is the exponent that satisfies the power-law relationship

$$L_r = \left(\frac{1}{r}\right)^p \tag{2.5}$$

so by equating Eqs. 2.4 and 2.5, the length or *fractal* dimension [Mand77], of the coastline is

$$D_L = 1 + p. \tag{2.6}$$

To show that this calculation works, the procedure can be implemented on the perimeter of the circle or any other non-fractal object. The length of any non-fractal object would yield a log-log plot with a slope $p = 0$. The length dimension would, as expected, be equal to one. The same procedure can be performed on the coastline. The results of the calculation show that the fractal dimension of the coast of Britain is 1.38. As a comparison, the fractal dimension of the coast of Norway is 1.66. It can be observed that the coast of Norway has more convolutions than the coast of Britain. Both coasts cover a certain degree of area through their meandering to exist beyond single dimension but not quite enough to constitute a two dimensional object.

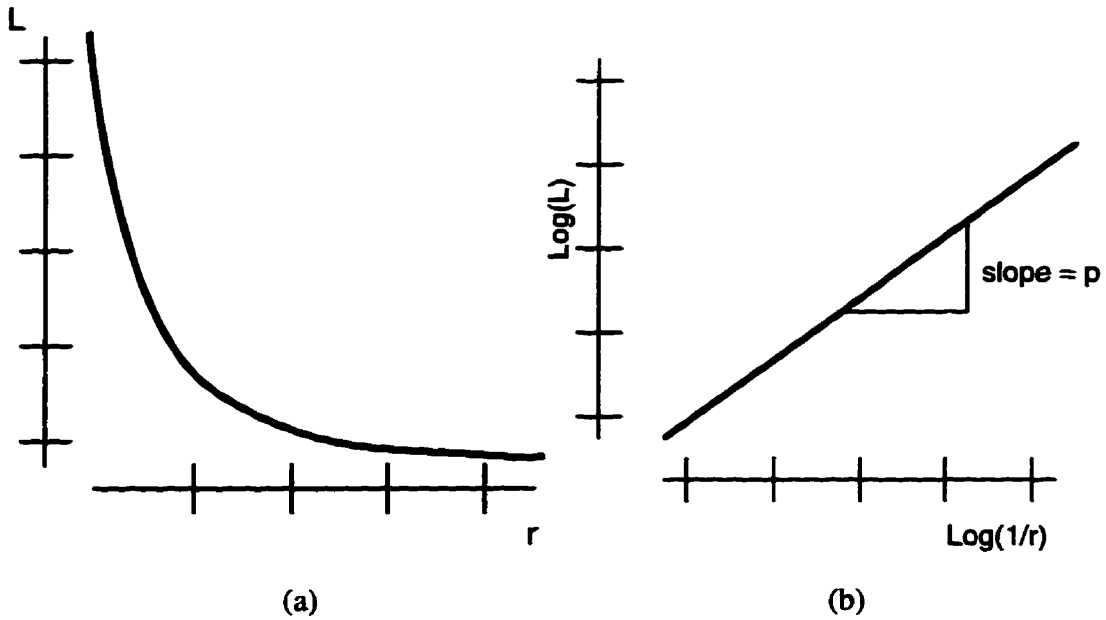


Fig. 2.4. (a) Plot of coastline length vs. length of measurement device. (b) Plot of log of coastline length vs. log of measurement precision (after [Kins94b]).

The Koch curve, being a strictly self-similar fractal object, can be analyzed in the same way. The Koch curve, by the nature of its construction, is already partitioned into line segments. The length of the curve is the product of the total number of line segments and the length of the line segments as in Eq. 2.4. The Koch curve, however, contains an infinite number of line segments so the product is infinite or undefined. At each scale level, the length of the line segment is scaled by 1/3. For each drop in scale, the number of line segments increases by a factor of four. The length of the Koch curve increases without bound. By following the established calculation for the fractal dimension, the result is 1.262.

The length dimension of a strictly self-similar object such as the Koch curve can be generalized to the fractal self-similarity dimension, D_S [Kins94b], which avoids the procedure necessary to calculate the slope of the log-log plot. Since an object that is reduced by a factor r produces N exact copies of itself, it satisfies the power-law equation

$$N = \left(\frac{1}{r}\right)^{D_S} \quad for\ r \to 0 \tag{2.7}$$

then

$$D_S = \frac{\log(N)}{\log\left(\frac{1}{r}\right)}. \tag{2.8}$$

For objects such as the Koch curve or the Cantor set, which are strictly self-similar, the self-similarity dimension can be calculated at any iteration of their construction. However, for objects that reach exact self-similarity at an infinite number of iterations, Eq. 2.8 can be further generalized to

$$D_{S_k} = \lim_{k \to \infty} \frac{\log(N_k)}{\log\left(\frac{1}{r_k}\right)}. \tag{2.9}$$

The above derivation will work on objects that are strictly self-similar. Felix Hausdorff introduced the concept of subdividing an object that is not necessarily strictly self-similar into equal size neighbourhoods [Mand77]. Blockwise subdivision is the simplest to implement but they can be any arbitrary shape. The procedure starts with a coarse subdivision of the object. The number of neighbourhoods of size that cover the object is counted. On the next iteration, the size of the neighbourhoods is decreased and the number of neighbourhoods that cover the object is again counted. This process is repeated until no more detail can be observed. The fractal dimension is calculated as

$$D_H = \lim_{r \to 0} \frac{\log(N_r)}{\log\left(\frac{1}{r}\right)} \tag{2.10}$$

where N_r is the number of neighbourhoods that cover the object with a neighbourhood size r. Abram S. Besicovitch extended this by stating that if the exponent of the measuring precision is less than D_H, the length is infinite and if the exponent is greater than D_H, the length vanishes. Equation 2.10, then, is called the Hausdorff-Besicovitch dimension [Mand77]. There are other fractal dimensions [Kins94a] which are suitable for different

kinds of fractals. A formal definition of a fractal can now be stated as a set whose Hausdorff-Besicovitch dimension strictly exceeds its topological dimension [Mand77].

The fractal dimension, whether it be the length dimension, self-similarity dimension, or the Hausdorff-Besicovitch dimension, measures the roughness of the fractal. The higher the dimension the rougher the fractal. It should be stressed, however, that this class of morphological fractal dimensions is incapable of revealing lower-order fractals [Kins94a]. Multifractal measures must be used to show the entire spectrum of fractals in an object. Fractals of varying dimensions can be thought to make up our natural environment as was already indicated by studying the coastline of a land mass.

2.3 Fractals in Nature

Our natural environment can be described through the use of fractals. In his book, *The Fractal Geometry of Nature*, Benoit Mandelbrot [Mand77] shows the connection between fractals and nature. For example, the bark of a tree, when viewed from a distance appears to have a rough surface. On closer inspection, that same roughness can be seen at finer and finer scales which demonstrates the concept of scale invariance. A mountain range, shown in Fig. 2.5, has a characteristic jagged appearance at all scales. Natural objects are self-affine, rather than strictly self-similar, which implies there are transformations other than scaling involved in the construction of the object. The class of images that are studied in this work are of the natural world, particularly of head-and-shoulder images of people.

By treating the greyscale values as altitudes, a three dimensional representation can be constructed, Fig 2.6. The result is very similar to the mountain range. This seems to confirm the connection that exists between greyscale images and fractals. In addition, by studying the features in an image self-similar portions can be located as shown in Fig. 2.7. While the construction process of such fractals itself is fairly simple, determining

specifically what process is required to construct a particular feature and to actually find the features is difficult.

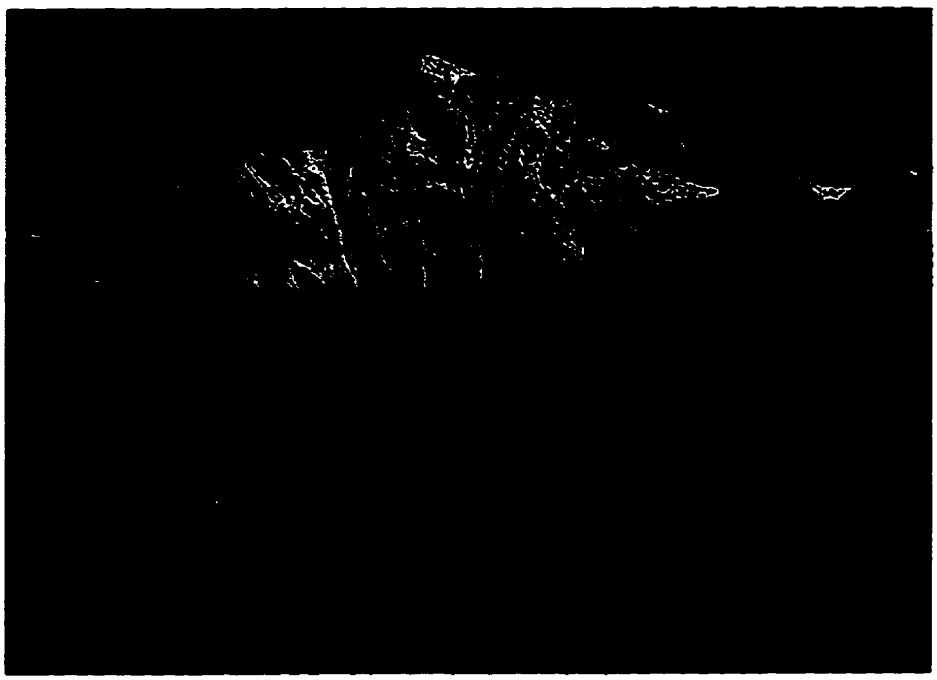
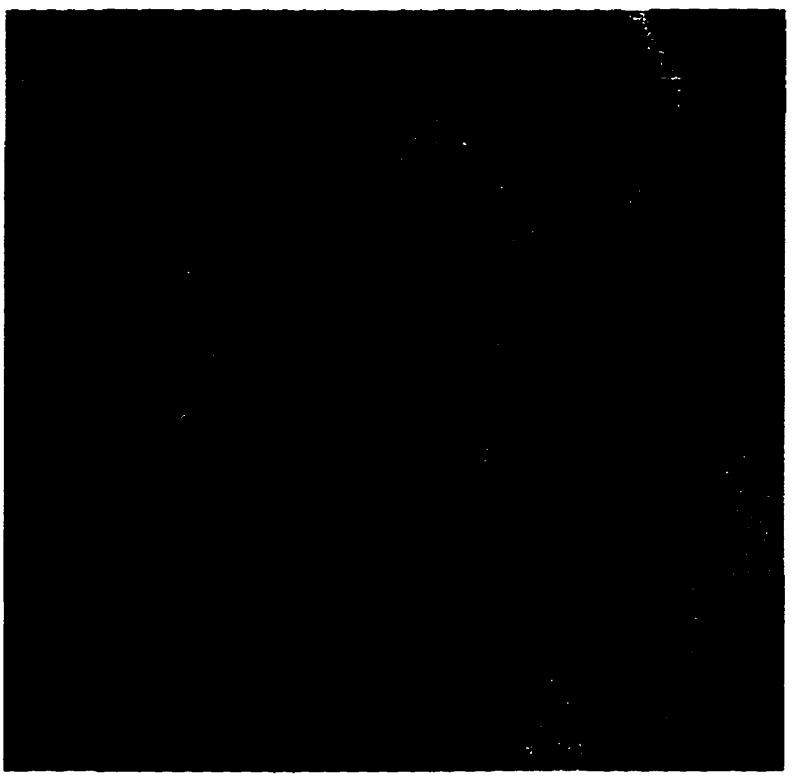


Fig. 2.5. The fractal nature of a mountain range.

In Section 2.1, the concept of self-affine structures was briefly introduced as objects that are composed of scaled down and distorted copies of itself or portions of itself. These transforms, in order to create a fractal, illustrate the contraction mapping principle. In order to understand how contractive transformations work, it is first required to understand the subject of metric spaces and metrics.

2.4 Metric Spaces and Metrics

Before any discussion of mappings or transformations can occur, metric spaces must first be defined. A metric space [Kreyszig] is a pair (X,d) where X is a set and the metric, d, is a distance measure defined on X. A metric space must satisfy the following for all $x, y, z \in X$:



(a)

(b)

Fig. 2.6. (a) Greyscale image of *Lena*. (b) Three-dimensional representation of *Lena* using greyscale values as altitudes (Image produced with Pov-Ray 3.0 [Youn96].).

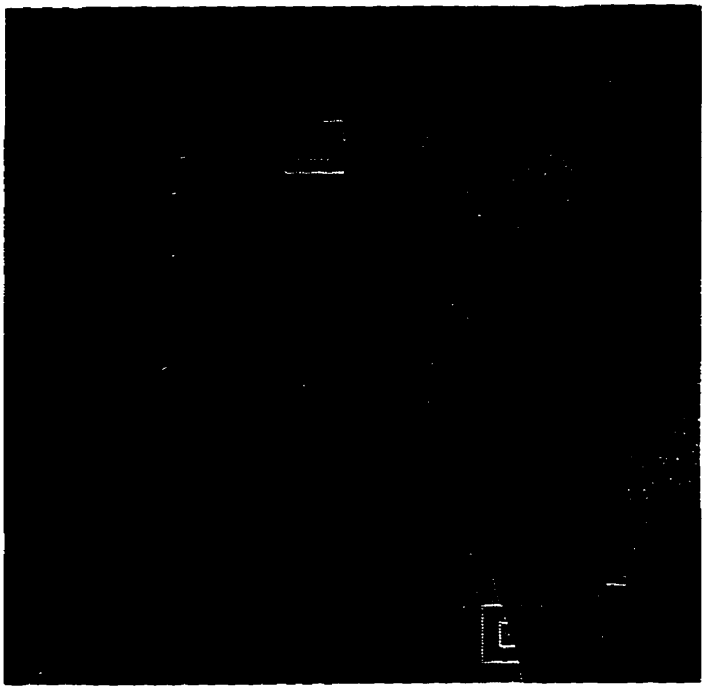


Fig. 2.7. Self-similar portions of *Lena* [Fish94].

1. d is real, finite, and non-negative
2. $d(x,y) = 0$ if and only if $x = y$
3. $d(x,y) = d(y,x)$
4. $d(x,y) \leq d(x,z) + d(z,y)$

An example of a metric space is the Euclidean plane, $\Re^2$. This metric space contains the set of ordered pairs of the form (x_1,y_2). The Euclidean metric is defined by

$$d(x,y) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{2.11}$$

Objects such as a square or circle exist in the Euclidean plane. For the discussion of fractals it is important to establish a space of reference. It had been determined earlier that fractals are not of the same complexity as other one or two dimensional objects which

reside in the two dimensional space. Therefore, the Hausdorff metric space [Barn93], which is complete, is introduced.

A *complete* metric space is one in which any Cauchy sequence in the space converges in that space. In other words, a complete metric space is a space in which all convergent sequences converge to points that are still in the space. A *Cauchy* sequence is such that for every $\varepsilon > 0$, there is an $N = N(\varepsilon)$ so that for a sequence $\{x_n\}$

$$d(x_m, x_n) < \varepsilon \tag{2.12}$$

for every $m, n > N$.

Given a complete metric space (X,d), the Hausdorff space $\mathcal{H}(X)$ contains points that are compact subsets of X, other than the empty set. The term compact refers to the requirement that all sequences in the space have convergent subsequences. The distance measure for the Hausdorff space is a measure of the distance between two points in $\mathcal{H}(X)$. Given a point $x \in X$ and a set $B \in \mathcal{H}(X)$ the distance between the point x and a set B is

$$d(x, B) = \min\{d(x, y) : y \in B\}. \tag{2.13}$$

Having established the distance between a point in X and a set in $\mathcal{H}(X)$, the distance between sets $A, B \in \mathcal{H}(X)$ is

$$d(A, B) = \max\{d(x, B) : x \in A\}. \tag{2.14}$$

This distance measure first, for every point in A, finds the distance to the nearest point in B. Then the maximum of those measures is determined. This measure is not necessarily equal to $d(B,A)$. The Hausdorff distance between points $A, B \in \mathcal{H}(X)$ is now defined as follows

$$h(A, B) = d(A, B) \vee d(B, A). \tag{2.15}$$

The notation $a \vee b$ is used to indicate the maximum of the real values a and b. The Hausdorff metric space is now defined as $(\mathcal{H}(X),h)$ in which fractals can be generated using affine transforms.

2.5 Affine Transforms

By definition [Barn93], an affine transform is of the form

$$g = a \cdot x + b \tag{2.16}$$

where a is a scaling factor and b is a translation. The transform g is a transform along the real line but can be extended into the two dimensional plane as

$$g(x) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \tag{2.17}$$

The 2 x 2 matrix performs a distortion which includes scaling, rotation, reflection, and shearing. The vector $\begin{bmatrix} e \\ f \end{bmatrix}$ is a translation along the horizontal and vertical axes. If the determinant of the matrix is less than 1, the transform is classified as contractive.

2.6 The Contraction Mapping Principle and the Collage Theorem

Given an affine mapping $f:X \rightarrow X$, on a metric space (X,d), where d is some distance measure, the contractive mapping theorem states that there is a constant $0 \leq c \leq 1$ such that

$$d(f(x),f(y)) \leq c \cdot d(x,y) \forall x, \quad y \in \mathbf{X}. \tag{2.18}$$

The constant c is called the contractivity factor of the contraction, f.

When the contractivity factor is restricted between 0 and 1, the recursive applications of the mapping will result in a convergent sequence. If $a_0, a_1, a_2, \ldots$ is a sequence of points in X defined by $a_{n+1} = f(a_n)$, then the sequence converges to a unique fixed point called an *attractor*, a_∞.

$$a_\infty = \lim_{n\to\infty} a_n \tag{2.19}$$

The attractor is characterized as being invariant so that $f(a_\infty) = a_\infty$, to which there exists an estimate of the distance, $d(a_n,a_\infty)$, from a_n.

$$d(a_n,a_\infty) \le \frac{c^n d(a_0,a_1)}{1-c}. \tag{2.20}$$

This measure gives an approximation of how long the sequence will take to reach an acceptable distance to the attractor. The attractor does not have to be composed of just one mapping but can be a *collage* of transforms.

The collage theorem states that, given a space (X,d), there exists a set of mappings, $\{f_0, f_1, f_2, \ldots, f_n\}$, an attractor, a_∞, a contractivity factor, c, and an arbitrary constant, ε, in X such that

$$d\left(L, \bigcup_{i=0}^{n} f_i(L)\right) \le \varepsilon \tag{2.21a}$$

and then

$$d(L,a_\infty) \le \frac{\varepsilon}{1-c} \tag{2.21b}$$

or equivalently,

$$d(L,a_\infty) \le \frac{d\left(L, \bigcup_{i=0}^{n} f_i(L)\right)}{1-c} \tag{2.21c}$$

for all $L \in \mathbf{X}$. This corollary of the contractive mapping theorem implies that given an attractor in $\mathbf{X}$ there exists a set of mappings, f, whose union is a close approximation of the attractor.

Affine transforms, the contraction mapping principle, and the collage theorem, being necessary for the construction of fractals, all have a role in fractal compression. The

earliest form of fractal image compression, *iterated function systems*, is quite dependent on them.

2.7 Iterated Function Systems

The collage theorem implies that a set of mappings exist for a given unique attractor. Iterated function systems [Barn93], IFS, consist of sets of contractive affine transformations (CATs), which are affine transforms as described in Section 2.5 that exhibit the contractive mapping theorem. An affine transform of the form of Eq. 2.16 would have to satisfy Eq. 2.18. That is, the distance between the transformations of two points in the space must be less than the distance between the points themselves. For the one dimensional case, this can be satisfied by restricting a to be between 0 and 1, not including the value 1. For the two dimensional case, the 2x2 matrix of Eq. 2.22 must have a determinant less than 1 in order for the transform to satisfy the contractive mapping theorem. Thus, the IFS will converge to a unique fixed point.

$$w\begin{pmatrix}\tilde{x}_1\\ \tilde{x}_2\end{pmatrix}=\begin{bmatrix}a & b\\ c & d\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix}+\begin{bmatrix}e\\ f\end{bmatrix} \tag{2.22}$$

As mentioned above, the IFS consists of a set of contractive affine transformations. The attractor is approached as the output of each CAT becomes the input of another. This continues through every CAT, with each one chosen with a predetermined probability. Since every CAT chosen is contained within the IFS, the IFS is considered a recursive system. This introduces a self-affinity in the attractor which, as described earlier, is a characteristic of fractals.

For example, the Sierpinski triangle can be modeled with IFSs, as shown in Fig. 2.8. Three affine transforms create three copies of a basic shape and places them in positions relative to each other. The parameters for each CAT are shown in Table 2.1. The system is initiated with any arbitrary point. The parameter p, indicates the probability at which that particular CAT is selected.

Table 2.1. Parameters for each CAT of the IFS to generate the Sierpinski gasket [PeJS92].

a	b	c	d	e	f	p
0.500	0.000	0.000	0.500	0.000	0.000	0.333
0.500	0.000	0.000	0.500	0.500	0.000	0.333
0.500	0.000	0.000	0.500	0.000	0.500	0.333

As the number of iterations increases, as a result of the collage theorem, the system converges to the attractor which, in this case is the Sierpinski gasket. The IFS is composed of 3 CATs, so the collage mapping of the IFS is

$$W(a) = w_1(a) \cup w_2(a) \cup w_3(a) \tag{2.23}$$

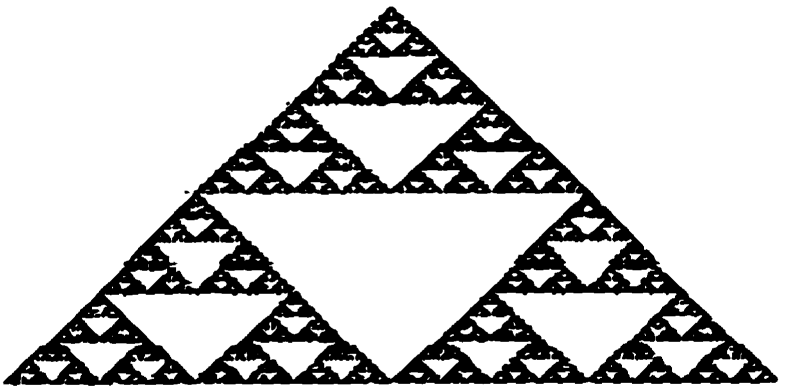


Fig. 2.8. Sierpinski Triangle.

The result of each iteration is plotted to create the image. This system will always converge to the Sierpinski gasket because, as required by the contractive mapping principle, the IFS should have a unique attractor so that $W(a_\infty) = a_\infty$. However, the inverse is not true. For a given attractor, there could be many IFSs that could converge to it. The size of the image of the Sierpinski gasket shown is 256x256 which is 65536 bits. The file containing the IFS coefficients can be as little as 576 bits, resulting in a compression ratio of approximately 114:1.

Another example of an IFS is shown in Fig. 2.9. The image created is a very realistic looking branch of a fern. These coefficients can be stored in 768 bits resulting in a compression ratio of 85:1. This immediately poses the possibility that IFSs can be used to

compress natural, real-world images as well. However, while it is fairly easy to generate the attractor when the IFS parameters are known, it is difficult to reverse the process. More specifically, it is difficult to have an automatic process determine the IFS parameters, since it is desired to have this processed through the use of a computer.

The problem with automating this process is that it must be known what points would be the range of the function, $w(\tilde{x})$, and what points would be the domain. The problem becomes a matter of solving for six unknown parameters so three range-domain pairs would be needed.

$$\begin{aligned} x_1a + y_1b + e &= \tilde{x}_1 \\ x_2a + y_2b + e &= \tilde{x}_2 \\ x_3a + y_3b + e &= \tilde{x}_3 \end{aligned} \tag{2.24a}$$

$$\begin{aligned} x_1c + y_1d + f &= \tilde{y}_1 \\ x_2c + y_2d + f &= \tilde{y}_2 \\ x_3c + y_3d + f &= \tilde{y}_3 \end{aligned} \tag{2.24b}$$

Fig. 2.9. Barnsley's fern.

Table 2.2. Parameter for each CAT of the IFS to generate Barnsley's fern [Barn88], [Barn93].

a	b	c	d	e	f	p
0	0	0	0.16	0	0	0.01
0.85	0.04	-0.04	0.85	0	1.6	0.85
0.2	-0.26	0.23	0.22	0	1.6	0.07
-0.15	0.28	0.26	0.24	0	0.44	0.07

The equations in Eq. 2.24 makes it possible to solve for the coefficients of the matrix and the translation vector given points (x_1,y_1), (x_2,y_2), and (x_3,y_3) and their corresponding affine transforms $(\bar{x}_1,\bar{y}_1)$, $(\bar{x}_2,\bar{y}_2)$, and $(\bar{x}_3,\bar{y}_3)$. A decision must be made as to which pairs of points are affine transforms of each other. This decision is not an easily automated task.

In a report by Bal [Bal95], a systematic approach to solving the inverse problem has been addressed. Software was written that provides a graphical user interface where an image can be displayed. The user is able to select what is believed to be three pairs of points where the second point of each pair is an affine transformation of the first point. The software calculates the parameters for the CAT and is then able to construct an image based on those parameters. More CATs are determined by selecting more sets of three pairs of points until the final constructed image converges to the original source image. The source code that calculates the parameters and construct the image is presented in Appendix D.

It will be shown how it can be possible to determine a set of affine transformations that is able to construct an image. This is done by finding a transformation between a pair of image blocks that are similar to each other. The transformation is such that it transforms one block to closely approximate the other block. This process is repeated for all image blocks that cover the image. The problem is that there are too many blocks to in an image to locate a pair that match. The number of blocks to be searched can be reduced by using an artificial neural network.

2.8 Frequency Sensitive Competitive Learning

Artificial neural networks have the ability to separate features in a data set that would not be separable using heuristical methods. The data set can be separated into classes whose members are members of the data set that share similar features. In order for the neural network to perform this task, it must be trained to recognize members of different classes and identify them as belonging to such classes. The method of training is competitive because individual cells or "neurons," referred to here as codewords or codebook vectors, in the network compete to classify a given data sample.

The structure of the network contains a single layer of neurons and a single input layer. Each neuron contains a vector, called a codeword, of N connection weights which indicate the strength of the connection between the neuron and each of the N elements in the input layer. The weight vector must be normalized so that the length of the vector is equal to one. The network in this form is then used in either a classification mode or a training mode.

In classification mode, an input vector is presented to the input layer of the network. A distortion measure, the Euclidean metric, is calculated for each codeword. The neuron whose codeword resulted in the smallest distortion is declared the winner. Another method of determining the winning neuron is with the greatest input activation, a. The input activation is calculated by

$$a = \sum_{i=1}^{N} w_i x_i = \langle \mathbf{w}, \mathbf{x} \rangle \tag{2.25}$$

The classification associated with that neuron is then associated with the input vector. In order for accurate classification to occur, the network must be trained properly.

Competitive learning [AKCM90] is a method for training the network. A neural network trained in this method tunes its codewords to be near data vectors that can be grouped together if they occur frequently enough. In other words, data vectors in a set of data samples that are statistically prominent become members of a separate class defined by

the neural network. The problem with this is that features that are statistically prominent will cause a select few of the codewords to win the competition. The remaining codewords will not be used resulting in inefficient use of the neural network. It would be desired that if the data samples are not evenly distributed among the classes, then the neural network train itself so that there is an even distribution among the codewords. A simple way of achieving this is to keep track of how often a particular codeword wins the competition. This method is referred to as *frequency sensitive competitive learning* (FSCL) [AKCM90].

The training algorithm for the FSCL neural network occurs in four stages. In the first stage the weight, $\mathbf{w}_i$, must be initialized. To offer good distribution, the elements of the vectors are set to the mean of all vectors in the training set plus or minus some random perturbation. The vectors are normalized to length equal to one. In the second stage, a training vector, $\mathbf{x}$, is chosen at random from the training set and presented to the network. The distance measures are compared and the codeword that is closest to $\mathbf{x}$ is declared the winner. Stage three updates the winning codeword in the following manner

$$\tilde{\mathbf{w}}_i = \mathbf{w}_i + \Delta\mathbf{w}_i \tag{2.26}$$

where i is the index of each vector element and $\Delta\mathbf{w}_i$ is defined by

$$\Delta\mathbf{w}_i = \eta(\mathbf{x}_i + \mathbf{w}_i) \tag{2.27}$$

with η being the learning rate. Before the start of training, the learning rate is set to equal a large value between 0.1 and 0.7. In the fourth stage, the learning rate is decreased and the process repeats at stage two. The entire process continues until the learning rate reduces to zero. The rate at which the learning rate falls should be set so that there will be enough training samples to properly condition the network.

The training algorithm described above describes the basic competitive learning algorithm. In FSCL, shown in Fig. 2.10, a counter is associated with each codeword that keeps track of how often that codeword wins the competition. It can be thought of as a conscience mechanism [AKCM90]. This frequency value, f, alters the distance measure

between the codeword and the training vector so that that codeword appears to be further away from the training vector than it really is. Thus, it will not be chosen when it wins enough times to be pushed further away than another codeword which then will have the opportunity to win the competition. The input activation, then, is also recalculated as

$$a = \frac{1}{f}\sum_{i=1}^{N} w_i x_i = \langle \mathbf{w}, \mathbf{x} \rangle \tag{2.28}$$

After training is complete, the codebook will be conditioned to recognize as many feature classes as there are codewords with each codeword getting an equal number of adjustments so that the training set is evenly distributed over the neural network.

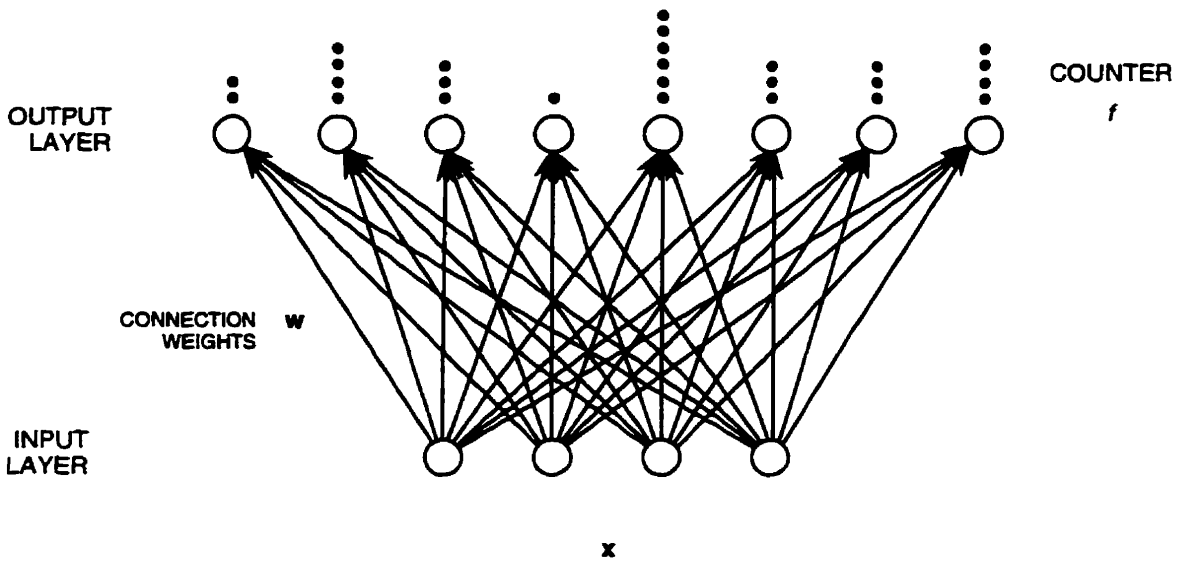


Fig. 2.10. Frequency sensitive competitive learning neural network.

An optimal number of codewords must be chosen to best represent the features in the training set. If too few are used, there will not be enough distinction made between various feature classes. Thus, one codeword would be chosen for different feature classes. If too many codewords are used, the codewords will become too specifically conditioned for a particular data vector. Thus, while one codeword will be chosen when one data vector is presented to the network, another codeword will be chosen when a different data

vector is presented to the network, even though the two data vectors belong to the same class.

The FSCL neural network will pre-classify the blocks so that finding affine transformable block pairs will be easier, but the block pairing is affected by additive noise. Noise, being random and unpredictable, reduces the number of similar block pairings existing in the image. The goal of this thesis is to reduce the noise, thus increasing the number of similar block pairings. This can be achieved through wavelet denoising. To describe how wavelet denoising works, a background of the wavelet transform is required which begins with a discussion of vector spaces and orthonormal bases.

2.9 Vector Spaces

A vector space consists of a set of vectors to which any of two algebraic operations can be applied with the result still being a member of the set [Krey78] and a norm which defines the magnitude of a vector. The two algebraic operations are vector addition and multiplication by scalars. For work with images, the vector space used is $L^2(\Re)$ where the vectors are of limited energy, as shown here.

$$\int_{\Re} |x(t)|^2 \, dt < \infty, \quad t \in \Re \tag{2.29}$$

The norm, $\|\bullet\|$, is defined by the inner product of a vector, $\mathbf{x}$, with itself, shown as

$$\|\mathbf{x}\| = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle} \tag{2.30}$$

where the inner product of two vectors, $\langle \bullet, \bullet \rangle$, is given by

$$\langle \mathbf{x}, \mathbf{y} \rangle = \int_{\Re} x(t) y(t) \, dt, \quad t \in \Re. \tag{2.31}$$

2.10 Orthonormal Bases

The algebraic operation of vector addition allows linear combinations of vectors whose result still resides in the set of vectors. A *linearly independent* set of vectors, $\mathbf{x}_1$, $\mathbf{x}_2$,..., $\mathbf{x}_r$, satisfies the equation,

$$\alpha_1\mathbf{x}_1 + \alpha_2\mathbf{x}_2 + \cdots + \alpha_r\mathbf{x}_r = 0, \tag{2.32}$$

where α_1, α_2, ..., α_r, are scalars all equal to 0. If this is not satisfied, then the set of vectors are *linearly dependent*. A linearly independent set of vectors forms a *basis* of a set M if every vector $\mathbf{v} \in M$ can be written as a unique linear combination, or superposition, of the linearly independent set of vectors. This can be written as

$$\mathbf{v} = \sum_{i=1}^{r} \alpha_i \mathbf{x}_i . \tag{2.33}$$

The set $\{\alpha_i \in \Re\}$, called the coefficients of $\mathbf{v}$, represents $\mathbf{v}$ completely and uniformly. Each scalar α_i, indicates the amplitude of each vector $\mathbf{x}_i$, that is, they indicate how much each vector $\mathbf{x}_i$ contributes to the vector $\mathbf{v}$. When the basis $\{\mathbf{x}_i\}$ and vector $\mathbf{v}$ are given, it is desired that the set of scalars $\{\alpha_i \in \Re\}$ be found. This is difficult for an arbitrary basis. However, if the basis is orthonormal, the scalars are easily found.

An orthonormal set [Krey78] is a set whose elements have a norm of one and the inner product of any two vectors in the set satisfy

$$\langle x_i, x_j \rangle = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}. \tag{2.34}$$

Now, each scalar can be calculated by the inner product

$$\alpha_i = \langle \mathbf{x}_i, \mathbf{v} \rangle. \tag{2.35}$$

This is an orthogonal projection of $\mathbf{v}$ on to $\mathbf{x}_i$. The orthogonal projection gives the component of $\mathbf{v}$ in the direction of $\mathbf{x}_i$. By taking the superposition of all the vectors in the basis with each scaled by the appropriate coefficient, the vector $\mathbf{v}$ is produced. The act of

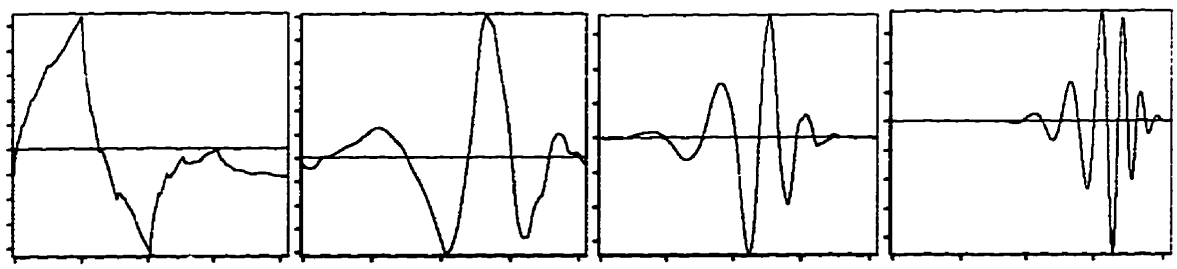


Fig. 2.11. A few examples of Daubechies wavelets: (a) Daubechies-4, (b) Daubechies-8, (c) Daubechies-12, (d) Daubechies-20

generating the coefficients is called the forward transform and the act of generating the vector, **v**, is called the inverse, or reverse, transform.

2.11 The Wavelet Transform

The idea of the forward transform using an orthonormal basis can be used on a signal that is represented by a vector to decompose it into a superposition of a set of vectors. Wavelets are scaled and translated copies of a single function, in a manner not unlike the transformations that exist in fractals.

$$\psi_{a,b}(t) = \sqrt{|a|}\psi_{1,0}\left(\frac{t-b}{a}\right) \tag{2.36}$$

Equation 2.36 [Daub92] illustrates the generation of wavelets, where a is a scaling factor, b is a translation, t is the index of the domain of the function, and $\psi_{1,0}$ is a function referred to as the *mother wavelet* scaled by a factor of 1 and translated by 0. With $a < 1$, the wavelet produced is a high frequency wavelet with a narrow width. With $a > 1$, the wavelet is low frequency with wide width. The shape, in all cases, however, remains the same. With a set of wavelets varying in scale, the wavelet transform can be performed. A few examples of wavelets are shown in Fig. 2.11.

In order for the set of wavelets to be suitable for wavelet transforms, there must be enough of them to span the entire signal space. If there are insufficient wavelets, then different input signals will result in identical transforms resulting in a poor reconstruction

when the inverse transform is performed. The optimal wavelet transform is a series of wavelets whose basis is a countable set of wavelets with a minimum number of basis signals that span the signal space. From Eq. 2.36, the basis signals are derived as

$$\mathbf{x}_{j,k} = \psi_{j,k}(t) = \sqrt{2^j}\psi_{1,0}\left(\frac{t-2^j k}{2^j}\right) \tag{2.37}$$

where the scale $a = 2^j$ and the translation $b = 2^j k$.

The wavelet transform is a superposition of a set of wavelets where a set of scalar coefficients are generated. If the set of wavelets forms an orthogonal basis, then given a real-valued signal, $v(t)$, and using Eq. 2.35, the wavelet coefficients can be generated by

$$c_{j,k} = \left\langle v(t), \mathbf{x}_{j,k} \right\rangle. \tag{2.38}$$

The value $W_v(a,t)$ is the wavelet coefficients at scale a. The inner product given above in $L^2(\Re)$ is defined as [Krey78]

$$\left\langle v(t), x_{j,k} \right\rangle = \int_{\Re} v(t)\psi_{j,k}(t)\, dt \tag{2.39}$$

To reconstruct the signal, $v(t)$, the inverse transform,

$$v(t) = \sum_{j,k} c_{j,k} x_{j,k} = \sum_{j,k} c_{j,k}\psi_{j,k}(t) \tag{2.40}$$

is used. Small values of j imply small values of a resulting in a narrow wavelet which is more representative of fine details. Coefficients corresponding to narrow wavelets will have a greater magnitude if there are many fine details in the input signal $v(t)$. Larger values of j increase the scale a, producing a wider wavelet which is more representative of coarse details.

In this thesis, instead of the continuous wavelet transform, the discrete wavelet transform, DWT, is more practical. The input signal $v(t)$ is sampled at equal intervals to produce N samples. The discrete signal is $\mathbf{v}$ = (v[0], v[1], ..., v[N-1]) so that it is now in the $\Re^N$ space. Using the same sampling rate, the wavelet basis, $\psi^{a,b}(t)$, is also transferred to $\Re^N$ space to yield $\psi^{a,b}[n]$. The forward transform becomes

$$W_v(a,n) = \langle v[n], \psi^{a,b}[n] \rangle \tag{2.41}$$

inner product becomes

$$\langle \mathbf{v}, \mathbf{x}_{j,k} \rangle = \sum_{i=0}^{N-1} v[i] \psi_{j,k}[i]. \tag{2.42}$$

Since, in practice, there will not be an infinite amount of signal, the signal will be limited by time or space, therefore, the amount of translation must also be limited. The shift k is limited to

$$0 \le k < 2^{-j} N \tag{2.43}$$

or $b = 2^j k$ will result in translation values greater then N-1 if k becomes too high. The value of j must also be restricted to remain above zero because $j<0$ will result in non-integer signal vector indices. That kind of precision is not necessary, however, because the level of detail in the input signal is limited to the number of samples in $v[n]$. Using $j=0$ will not be practical because this will cause the wavelet to vanish resulting in the coefficient being 0. In Eq. 2.43, $2^{-j}N$ can not be less than one or there will be non-integer signal vector indices. This requires that j be limited by

$$1 \le j \le J \tag{2.44}$$

where $J \le \log_2(N)$ is a user defined parameter. The upper limit on J ensures that k does not exceed its limitation in Eq. 2.43. The upper limit on J, however, restricts the ability of the wavelets to span the entire signal space because there will be coarse details that require j to be higher than J. To accommodate this, the DWT incorporates an orthonormal scaling basis signal, $\phi_{J,k}$, which scales the input signal so that the coarse details for $j > J$ fall within the span of the wavelets. The scaling coefficients are the low frequency components of the signal because the scaling signals scale down the coarse, or low frequency, parts of the signal. The forward DWT then consists of $\{c_{j,k}, d_{j,k}\}$ which is defined by

$$c_{j,k} = \sum_{n=0}^{N-1} v[n] \psi_{j,k}[n], \quad d_{J,k} = \sum_{n=0}^{N-1} v[n] \phi_{J,k}[n] \tag{2.45}$$

and the inverse by

$$v[n] = \sum_{j=1}^{J} \left(\sum_{k=0}^{(2^{-j}N-1)} c_{j,k} \psi_{j,k}[n] + \sum_{k=0}^{(2^{-j}N-1)} d_{J,k} \phi_{J,k}[n] \right). \tag{2.46}$$

It can be shown that the total number of scaling coefficients plus the number of wavelet coefficients, is equal to N, the total number of samples in the input signal.

The above discussion of the wavelet transform is intended to work with 1-dimensional signals. For applications involving images of size NxN, a 2-dimensional transform is required [ABMD92]. This is carried out by performing the 1-dimensional DWT on the individual rows and columns of pixels of the image with a process called sub-band coding [Mall89], where the image is divided in a pyramid fashion into four sub-bands. This is accomplished by first performing the one-dimensional DWT on the rows of the image. The first half of the row is then replaced with the scaling coefficients, $d_{J,k}$, and the second half of the image is replaced with the wavelet coefficients, $c_{j,k}$. The DWT is then performed on the columns of the image, which at this point, has been replaced with the transforms of the rows. The top half of each column is replaced with the scaling coefficients and the bottom half is replaced with the wavelet coefficients. The result is shown in Fig. 2.12. Each sub-band is labeled in binary from 00 to 11. The sub-band labeled 10 shows the wavelet coefficients of the DWT of the rows, the sub-band labeled 01 shows the wavelet coefficients of the DWT of the columns, and the sub-band labeled 11 shows the coefficients of the DWT performed on both the rows and columns. The sub-band labeled 00 contains scaling coefficients only so it is a scaled down copy of the original image. The scaling signals scale the low frequency parts of the image so this sub-band shows the lower frequency parts of the image. The production of the four sub-bands is one stage of the multiresolution decomposition of the image. The decomposition continues by performing the sub-band coding to the sub-band labeled 00 until there is one pixel left.

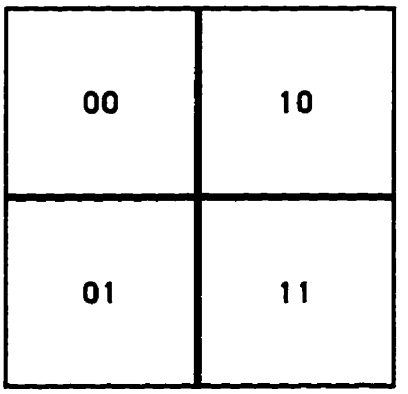


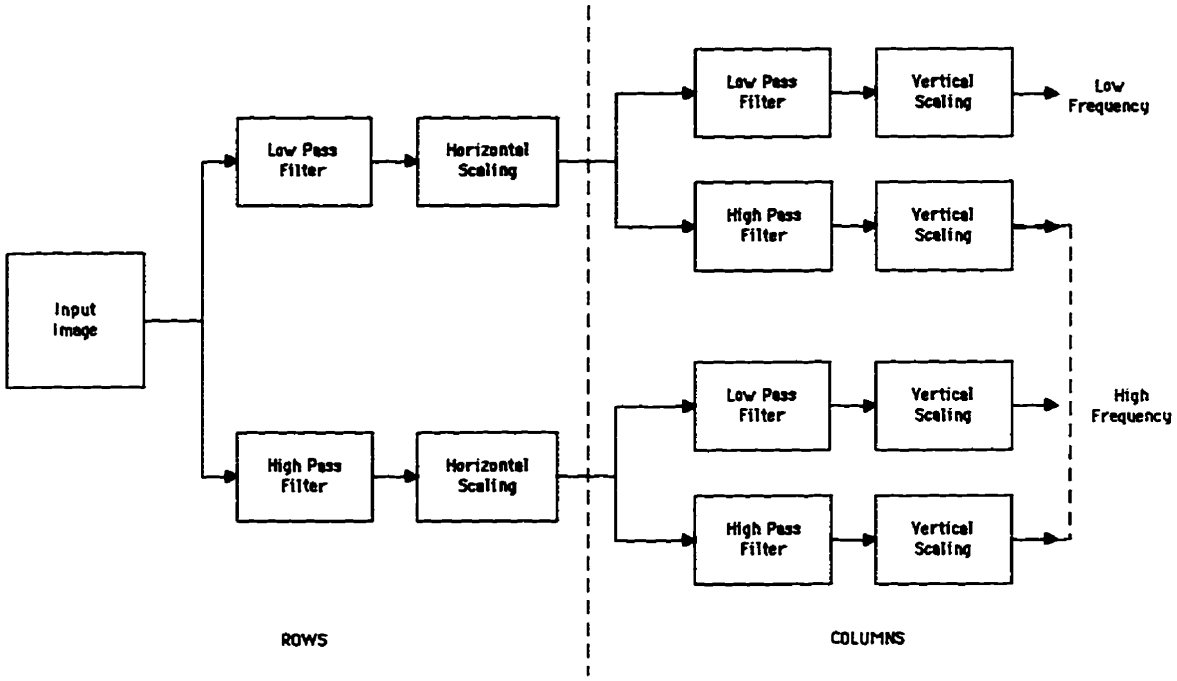
Fig. 2.12. Sub-band decomposition of an image [Lang96].

Following the multiresolution decomposition, the result is an NxN array of wavelet coefficients at various scales. To reconstruct the image, the inverse DWT is performed in the reverse order of the forward DWT. The procedure commences at the small scale and the inverse one-dimensional DWT is first performed on the columns. The first half of the columns is replaced with the scaling coefficients of each row and the second half is replaced with the wavelet coefficients of each row. Following this, the inverse one-dimensional DWT is performed on the rows. Each row is replaced with the low pass components of the image. This is one stage of the inverse DWT of the image. The scale level is increased by a factor of two and the procedure is repeated. The final result is the reconstructed image. The flow of the forward and inverse DWT of an image can be shown graphically in Fig. 2.13.

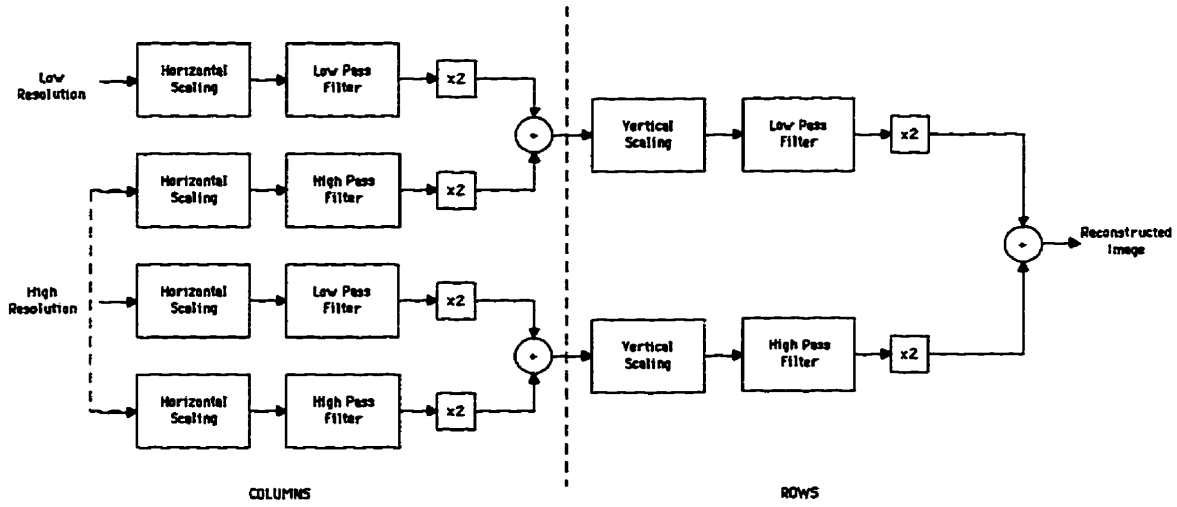
Performing the wavelet transform requires proper selection of the mother wavelet itself. The wavelet must possess properties that depend on the signal being analyzed. If the signal has many smooth features, it would be desireable to use wavelets that are smooth. For rough signals, rough wavelets would be appropriate. Wavelets are characterized by vanishing moments, regularity, and compact support. The number, M, of vanishing moments a wavelet has reflects the ability for the wavelet to vanish a polynomial of order M in a time integration. A vanishing moment is defined as an integer m that satisfies

$$\int_T t^m \psi_{1,0}(t)\,dt = 0, \quad 0 \le m < M. \tag{2.47}$$

A wavelet transform using a wavelet with a large number of vanishing moments will efficiently model smooth signals because smooth signals are considered to be high-order polynomials. The reverse is true for rough signals.



(a)



(b)

Fig. 2.13. Forward (a) and inverse (b) discrete wavelet transform (after [ABMD92]).

For practical purposes, the wavelets should have a compact support. Compact support means that the wavelet is restricted to a finite period of time or space. If the wavelet is confined to a finite domain, it can be easily implemented. An infinite domain is impossible to implement practically so that an approximation would have to be used instead.

The set of wavelets used in this work is the Daubechies family of wavelets. The set contains wavelets of varying smoothness and compactness. The Daubechies-4 wavelet is not very smooth but it has a very compact support. The Daubechies-20 wavelet is very smooth but its support is not as compact as that of the Daubechies-4 wavelet. In order for wavelet transforms to be effective in the reduction of noise, a wavelet must be chosen so that it models the noise signal better than it models the desired signal. The coefficients of the noise will be prominent and thus are easier to make smaller. When the image is reconstructed with less prominent noise coefficients, the image will have been denoised.

2.12 Summary

This chapter has given a theoretical background to the work presented in this thesis. A brief explanation of fractals was given in terms of how they can be constructed, measured and how they are related to nature. Fractals can be constructed directly by drawing a recursive set of generators each at different scales or through the use of predefined affine transformations contained in iterated function systems. In doing so, it is seen that fractals can be represented in a very compact way even if the fractal itself is very complicated. Being able to measure a fractal in terms of its fractal dimension shows the reader that fractals are tangible and increases our understanding of them so that constructing and using fractals becomes easier. It was then illustrated that our natural environment can be constructed with fractals. Everything from the bark of a tree to vast mountain ranges are fractal in nature. By representing our natural environment as made up of a collage of

fractals, which, in turn, can be represented in a very compact form, an image of a portion of the natural environment can be represented in a very compact form.

The image in question also contains additive noise which decreases the 'fractalness' of the image. The noise can be reduced through wavelet denoising so a description of the wavelet transform was given. A two-dimensional wavelet transform of an image is performed by taking the one-dimensional wavelet transform of the rows and columns of pixels in the image. Some of the characteristics of wavelets and how they affect the result of the transform were described. Short, jagged wavelets are more suitable for modeling rough, impulse-type signals, while longer, smooth wavelets are suitable for smooth, large scale features.

Having established the theory of fractals, competitive learning, and wavelets, the methodology of fractal block coding can be described. This description includes generalized fractal block coding with $O(n^4)$ complexity and reduced-search fractal block coding with $O(n^3)$ complexity. The remainder of the thesis will focus on a pre-processing stage to reduced-search fractal block coding to increase the efficiency of the compression algorithm.

CHAPTER 3

FRACTAL BLOCK CODING

In Chapter 2, the concept of self-affine features was introduced. Portions of the image can be represented with scaled down, distorted copies of other portions of the image. This redundancy in images can be exploited in a blockwise fashion to compress the image using a scheme called fractal block coding (FBC) [Jacq92], [Jacq93]. This chapter describes the generalized FBC process and the results that are presently attainable with that technique. A problem that exists with the generalized method is that it has a time complexity of $O(n^4)$ which makes it impractical where short processing times are needed. Through the use of a frequency sensitive competitive learning (FSCL) neural network [AKCM90], the time complexity can be reduced to $O(n^3)$ [WaKi93] which makes it more practical to use. This new method is called reduced-search FBC. The fractal code can then be further compressed losslessly through arithmetic entropy encoding [Kins91]. These points are described in this chapter followed by a summary.

3.1 Generalized Fractal Block Coding

In the generalized FBC process, the image to be compressed is divided into non-overlapping square image vectors of size *rxr*. Each block represents the range of a contractive image transformation. The problem to be solved is to find the domain of that transformation. The domain of the transformation is also a block, sampled from the same image, where one must be found such that, after the contractive transform, it closely approximates a given range block. The image is divided into domain blocks of size *dxd* that may or may not overlap. The best approximation is chosen by finding the domain block that results in an approximation that minimizes the Euclidean distance measure

$$d(\tilde{\mathbf{x}},\mathbf{y}) = \sqrt{\sum_{i=1}^{r}\sum_{j=1}^{r}\left(\tilde{\mathbf{x}}_{ij} - \mathbf{y}_{ij}\right)^2} \tag{3.1}$$

where $\tilde{\mathbf{x}}$ is the transformed domain block and $\mathbf{y}$ is the range block.

The domain block is transformed through a process called the fractal block transform. The fractal block transform consists of a spatial contraction, isometric transformation, and grey level scaling and translation. The spatial contraction reduces the size of the domain block from dxd to rxr. If d is an integer multiple of r then this is achieved by taking the average of every $\frac{d}{r}x\frac{d}{r}$ block of pixels. The isometric transformation is performed on the spatially reduced block so that the pixels of the block are rearranged in one of eight ways. The eight isometric transforms, shown in Fig. 3.1, are

1. identity
2. reflection about the vertical axis
3. reflection about the horizontal axis
4. reflection about the first diagonal
5. reflection about the second diagonal
6. 90° rotation about the centre
7. 180° rotation about the centre
8. 270° rotation about the centre.

This set of transforms is referred to as a group in relation to set theory where each transform has an inverse and the application of two or more transforms is equivalent to one transform.

Following the spatial reduction and isometric transformation, the pixel values of the block are scaled and translated to better approximate the source range block shown by

$$\tilde{\mathbf{x}} = a\mathbf{x}' + t\mathbf{u} \tag{3.2}$$

where a is the scaling factor, t is the translation amount, $\mathbf{x}'$ is the spatially reduced and isometrically transformed vector, and $\mathbf{u}$ is a vector whose dimensions are the same as $\mathbf{x}'$ and whose values are equal to one.

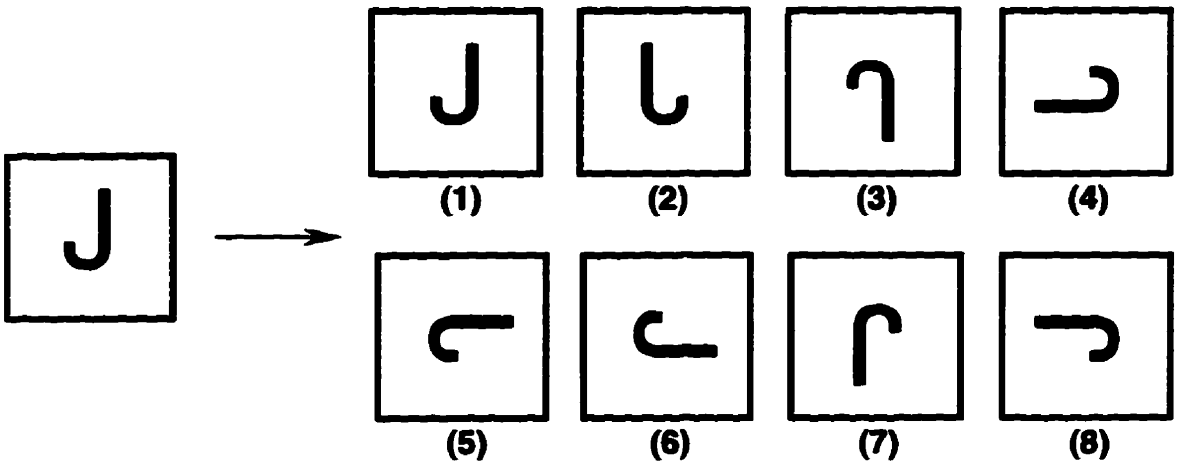


Fig. 3.1. The eight isometric transforms of fractal block coding.

The optimal values for a and t must be found for the best approximation of $\mathbf{y}$. By using the Euclidean norm, $\|\bullet\|$, and inner product, $\langle\bullet,\bullet\rangle$, the optimal value for a is [Wall93]

$$a = \frac{\|\mathbf{u}\|^2\langle\mathbf{x},\mathbf{y}\rangle - \langle\mathbf{x},\mathbf{u}\rangle\langle\mathbf{y},\mathbf{u}\rangle}{\|\mathbf{u}\|^2\|\mathbf{x}\|^2 - \langle\mathbf{x},\mathbf{u}\rangle^2} \tag{3.3}$$

and the optimal value for t is

$$t = \frac{\|\mathbf{x}\|^2\langle\mathbf{y},\mathbf{u}\rangle - \langle\mathbf{x},\mathbf{y}\rangle\langle\mathbf{x},\mathbf{u}\rangle}{\|\mathbf{u}\|^2\|\mathbf{x}\|^2 - \langle\mathbf{x},\mathbf{u}\rangle^2}. \tag{3.4}$$

Having established the optimal values for a and t, the fractal code for each range block is listed as follows:

1. the pointer to the best domain block
2. the spatial reduction scale
3. the isometric transform
4. the optimal grey level scaling coefficient, a
5. the optimal grey level translation coefficient, t.

The fractal code is a more compact representation of the range block than the range block itself.

The image must be able to be reconstructed from the fractal code to verify FBC as a valid image coding technique. The reconstruction occurs in an iterative fashion where one pass through the entire fractal code is considered to be one iteration. The fractal code is

processed sequentially, taking one range block representation at a time. The pointer to the best domain block is used to sample a domain block from an arbitrary image support. The block is spatially reduced and isometrically transformed according to the values in the fractal code. The resulting block is then grey-level scaled and translated and mapped on to the position in the arbitrary image support corresponding to the range block being reconstructed. After the entire fractal code has been processed in this way, one iteration is complete and another iteration may commence with the now slightly altered set of pixels in the image support. The more iterations that occur, the closer the reconstruction will be to the attractor, which is the original image.

The manner in which the image is constructed is similar to iterated functions systems where repeated recursive iterations of a set of CATs move the system closer and closer to a fixed attractor. In the case of FBC, the entire image is the point that is passed from function to function. The FBC technique supports the collage theorem in that the union of the reconstruction of every range block forms a collage of the original image.

The determination of the optimal domain block in the FBC process requires an exhaustive search of the entire domain space as was done in [Jacq92], [Jacq93]. The computational complexity of this is $O(n^4)$ so the time requirement makes the process impractical for most purposes. Fractal coding a single image would take half a day or more. In [WaKi93], a reduced-search method is introduced that makes use of a FSCL neural network.

3.2 Reduced-Search Fractal Block Coding

In the generalized FBC process, the entire domain space must be searched for a best match for a particular range block. This exhaustive process is extremely computationally intensive having order $O(n^4)$. With the introduction of a FSCL neural network [WaKi93], the domain space can be divided into subclasses each of which alone is searched, instead of the entire domain space.

During the FBC process, shown in Fig. 3.2, each sampled range block is first classified using a pre-trained FSCL neural network. Then the domain block search commences, but instead of taking each domain block and performing the fractal transform to see if it is a match for the range block, it is classified with the same neural network first. The domain block is scaled to the same size as the range block before the classification occurs. If the class of the domain block matches that of the range block, then the fractal transform may commence and the FBC process continues as before. If the classes do not match, the domain block is discarded. This process of discarded domain blocks reduces the complexity to the order $O(n^3)$. A process that took half a day to perform, now is executed in a matter of minutes.

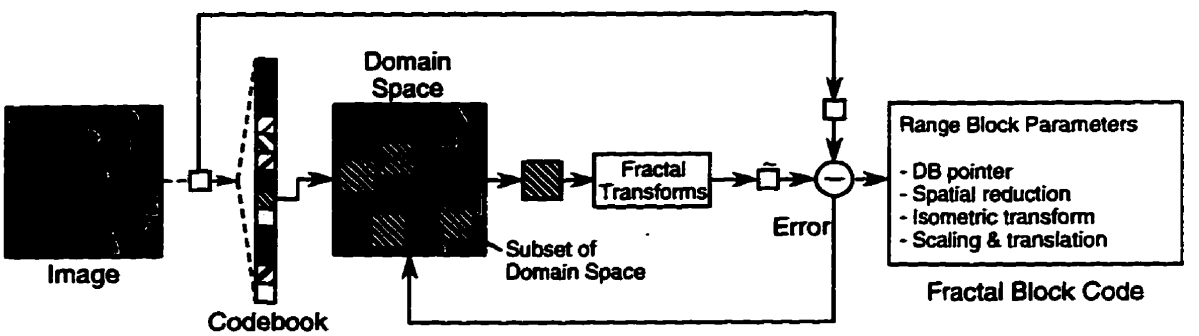


Fig. 3.2. Reduced-search fractal block coding.

The time complexities are calculated based on the number of operations that must be performed on the blocks and how many blocks there are. Thus, for the exhaustive approach, with 8 transforms that must be performed for every domain block and range block, the time required is calculated by

$$T = 8(n-d)^2\left(\frac{n}{r}\right)^2 \tag{3.5}$$

where n is the dimension of the image, d is the size of the domain block, and r is the size of the range block. As can be seen in Eq. 3.5, the exhaustive process is an $O(n^4)$ problem.

The time required for the reduced-search algorithm with c codewords, which implies there are c number of classes, is

$$T_c = 8(n-d)^2 + 8\left(\frac{n}{r}\right)^2 + \frac{(n-d)^2\left(\frac{n}{r}\right)^2}{c} \tag{3.6}$$

By minimizing with respect to c, the minimum time required is

$$T_{\min} = 4\sqrt{2}(n-d)^2\frac{n}{r} + 2\sqrt{2}\left(\frac{n}{r}\right)^3. \tag{3.7}$$

This is a complexity of $O(n^3)$.

3.3 Arithmetic Entropy Encoding

The fractal code generated by the exhaustive process or the reduced-search process contains many statistical redundancies. The better the fractal coding, the more redundancies there will be. Through a lossless compression technique, such as arithmetic entropy encoding, these redundancies can be exploited to produce an even more compact code. Like other statistical coding techniques [Kins91], arithmetic entropy encoding attempts to produce a representation of the data stream with a lower entropy based on the relative probability at which each symbol in the stream is likely to occur.

Entropy is a measure of the information content of the data stream. If entropy is low, there is a high level of redundancy in the data stream. If entropy is high, then there is a high information content and low redundancy. The calculation of entropy is based on the probability of occurrence of each symbol in the data stream. Entropy, H, is calculated by

$$H = -\sum_{i=1}^{M} p_i \log_2(p_i) \tag{3.8}$$

where p_i is the probability of occurrence of the i^{th} unique symbol and M is the number of unique symbols in the data stream. The units of this measure is average number of bits per pixel (bpp). Entropy also gives a measure of how well the data stream can be losslessly

compressed because it is impossible to precisely represent the data stream with fewer than the average number of bits per pixel indicated by the entropy.

Arithmetic entropy encoding develops a single codeword from the data stream by interval scaling using fractal self-similarity. The semi-open interval [0.0,1.0) is divided into M subintervals each representing a unique symbol in the data stream. The size of each interval is proportional to the probability of occurrence for that particular symbol. The interval representing the first symbol in the data stream is selected and divided into M subintervals in the same manner as the original interval. The interval representing the second symbol is selected and the process continues until the last symbol is reached. The final interval is the representation for the data stream. The repeated subdivisions form a fractal self-similarity.

In the decoding process, the interval that was output from the encoding process selects intervals representing the appropriate symbol. The process starts with the open interval [0.0,1.0) that is divided into M subintervals as before. The subinterval in which the encoded interval falls is selected. The symbol that is represented by that interval is the first symbol in the decoding process. The subinterval is divided into M subintervals and the subinterval that contains the encoded interval is selected. This yields the second symbol. The process continues until the subinterval selected is the same size as the encoded interval. The encoding and decoding algorithms are shown in Figs. 3.3 and 3.4 respectively.

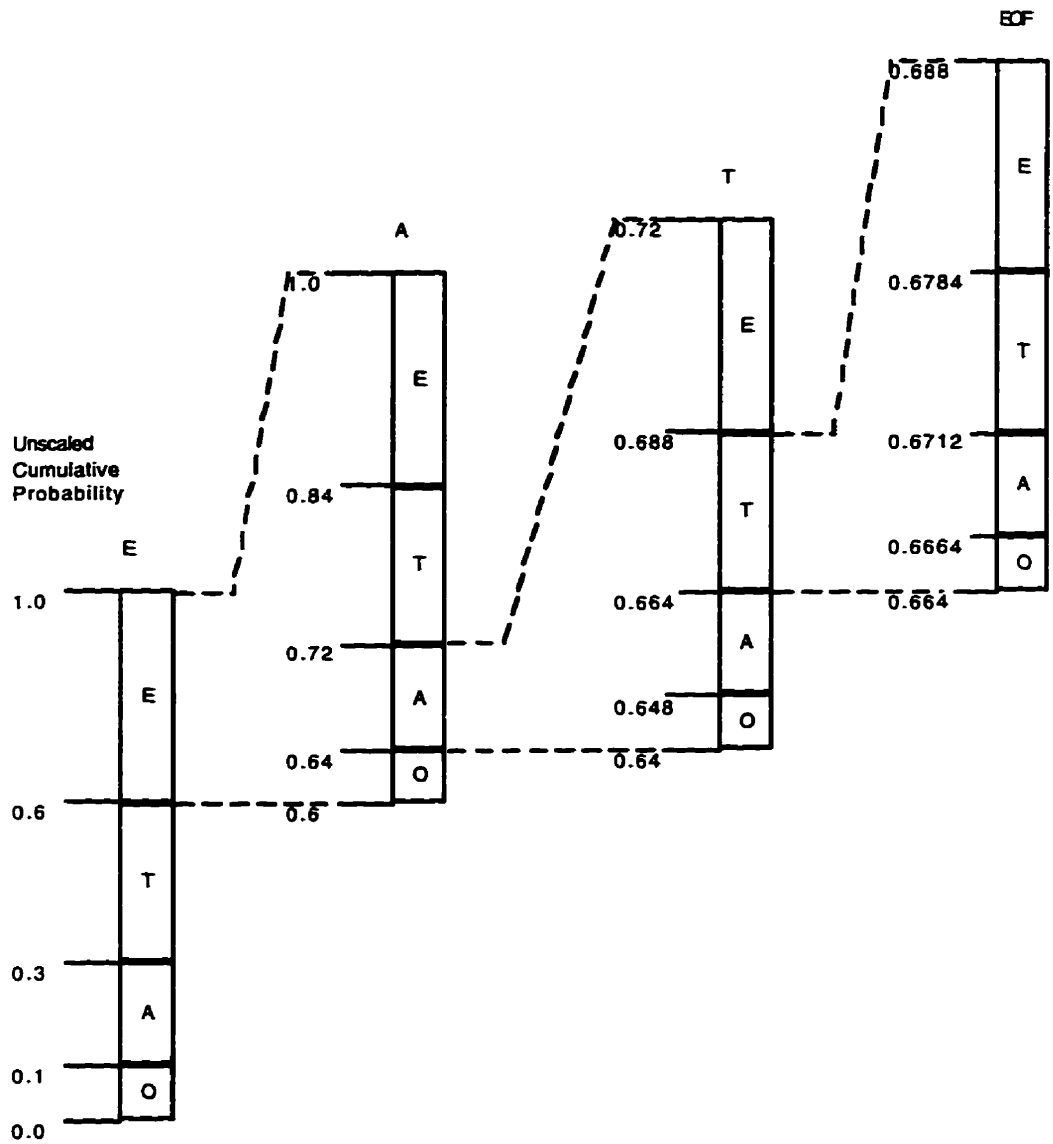


Fig. 3.3. Arithmetic entropy encoding of the string "EAT" with an end of file character. The interval returned is [0.664, 0.6664).

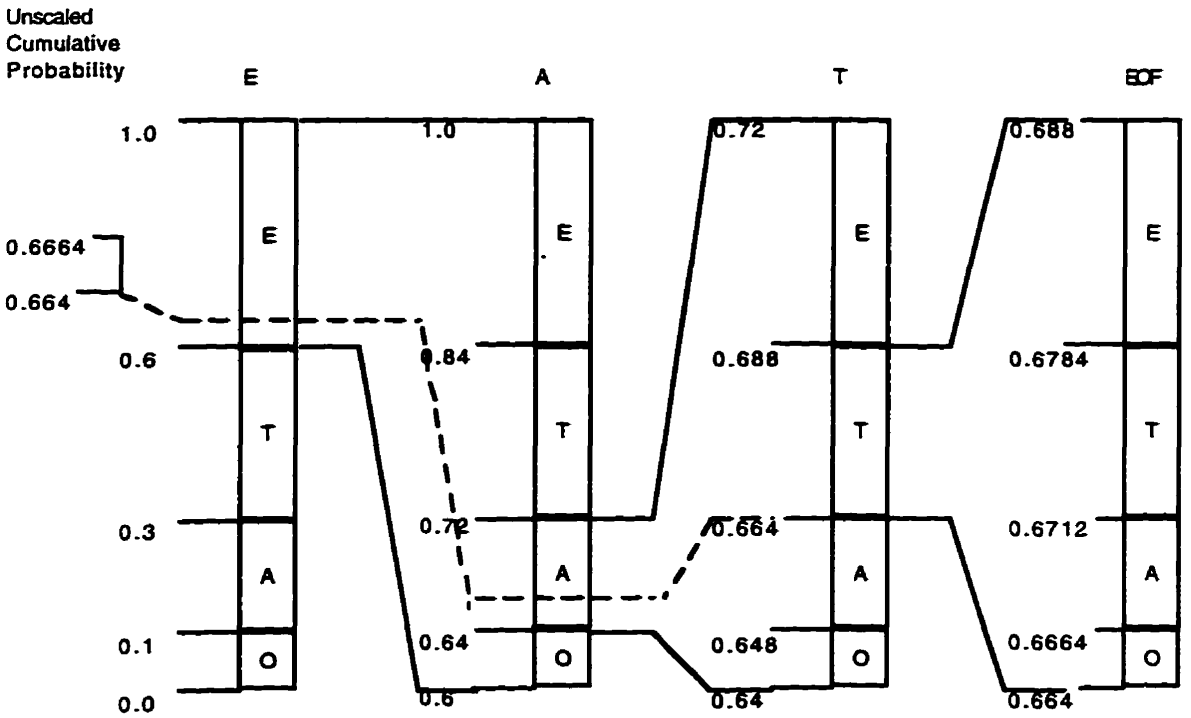


Fig. 3.4. Arithmetic entropy decoding of the interval [0.664, 0.6664) [Kins91].

3.4 Summary

This chapter has described how fractal block coding works. Beginning with the generalized form, the fundamental principle behind FBC was described. The image is searched for a domain block, that, when spatially reduced, isometrically transformed, and grey level scaled and translated, closely approximates a given range block. The fractal code contains five items: the pointer to the best domain block, the spatial reduction factor, the isometric transform, the optimal grey level scaling coefficient, and the optimal grey level translation coefficient. In the generalized form, the exhaustive search for the best matching domain block has a time complexity of $O(n^4)$ which, for most purposes, is impractical. With the incorporation of an FSCL neural network, the time complexity can be improved.

In reduced-search FBC, the FSCL neural network is used to pre-classify the range blocks and the domain blocks before they are transformed. If the class of a selected domain block does not match the class of the range block, the domain block is discarded

and time is not wasted on a domain block that would be eventually discarded anyway. It was shown that the reduced-search method has a time complexity of $O(n^3)$. The resulting fractal code, in either the exhaustive or reduced-search methods, quantizes the redundancies that are present in the image.

The redundancies present in the image are present in the fractal code. Arithmetic entropy encoding is a lossless compression technique that can take advantage of those redundancies to compress the fractal code even further. Arithmetic entropy encoding was described as a technique that produces a representation for the data stream with a lower entropy based on the relative probability at which each symbol in the stream is likely to occur.

With fractal block coding explained, it can now be shown how the process can be improved. As mentioned earlier, additive noise reduces the 'fractalness' of an image by reducing the level of redundancy. The next chapter will describe methods that can be used to reduce the amount of additive noise to increase the efficiency of reduced-search fractal block coding.

CHAPTER 4

DENOISING IMAGES

The fractal coding process described in Chapter 3 exploits the inherent fractal nature of the image being coded. The fractal nature comes from the redundancies that occur from correlated pixels in the image. However, uncorrelated noise often contaminates the image, reducing the efficiency of the fractal block coding process. As with many compression techniques, the efficiency of FBC increases with a lower entropy which implies a greater level of redundancy. FBC becomes more efficient as the level of redundancy increases because fractal block coding searches for redundancies in the image so the more redundancies that are found, the more the image can be compressed. This chapter uses concepts as implemented by Kinsner and Langi [LaKi96].

Naturally occurring signals are usually contaminated with noise. A pixel in a contaminated image, $g(x,y)$, contains the uncontaminated signal, $f(x,y)$, plus an additive noise, $n(x,y)$ as shown below.

$$g(x,y) = f(x,y) + n(x,y) \tag{4.1}$$

Additive noise raises the entropy of the signal. The goal of image denoising is to suppress the noise enough to lower the entropy of the signal, thus raising the level of redundancy of the signal. The resulting signal ideally, would be

$$\hat{g}(x,y) = f(x,y) + \hat{n}(x,y) \tag{4.2}$$

where $\hat{n}(x,y)$ is the reduced noise signal.

This chapter describes two methods that were used in this work to reduce additive noise. The first, spatial smoothing affects the raw pixels themselves, thus affecting the desired signal itself as well as the noise. The second method, wavelet denoising, transforms the image from the spatial domain to the wavelet domain where the noise can be affected without greatly disturbing the desired signal.

4.1 Spatial Smoothing

Spatial smoothing [GoWo92], or spatial filtering, is a process that works directly on the image pixels themselves. A square mask of odd dimensions is passed over the image, one pixel at a time as shown in Fig. 4.1. Spatial filters are also referred as lowpass filters.

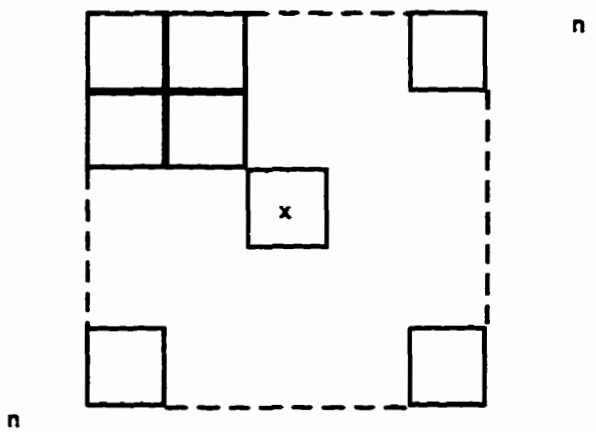


Fig. 4.1. Filter mask of odd dimension n. The pixel underneath the centre element x is replaced with average of the pixel values that lie beneath the mask.

Low-pass filters attenuate high frequency components of the signal. High frequency components are sharp features such as edges and points. Noise is generally high frequency in nature. The value of the pixel that falls beneath the centre element of the mask is replaced by the average of the values of all the pixels that lie beneath the mask. This method is based on an assumption that the values of the pixels surrounding a particular pixel are the same. Any discrepancies would be due to noise and, therefore, by averaging, the noise is suppressed. If sharp features are attenuated, the effect is blurring. In images, this would make any sharply defined features without clear definition. The new signal is shown here by

$$\hat{g}(x,y) = \hat{f}(x,y) + \hat{n}(x,y) \tag{4.3}$$

which defers from Eq. 4.2 by containing the altered signal $\hat{f}(x,y)$. Ideally it would be desired that there is no change in $f(x,y)$, but in lossy compression methods, some loss of detail is tolerated, but only to a certain point. The blurring effect becomes more evident as the size of the mask is increased. Distortion of the desired signal is unavoidable. A better situation would be to process the image in a space that avoids operating directly on the raw pixels themselves and directly works on the noise. Denoising, in contrast to filtering, with wavelets offers such a solution.

4.2 Wavelet Denoising

In general, wavelet denoising, or wavelet smoothing, is a lossy technique. However, it can be considered lossless because the data that is removed, i.e. the noise, is not considered part of the signal. Wavelet denoising is a technique that preserves signal smoothness where the image is separated from the noise based on its membership in a set of smooth prototype signals. The image, $f(x,y)$, belongs to a space that the noise, $n(x,y)$, does not. It is desired to reduce $n(x,y)$ to $\hat{n}(x,y)$ to change Eq. 4.1 to:

$$\hat{g}(x,y) = f(x,y) + \hat{n}(x,y) \tag{4.4}$$

If $\hat{g}(x,y)$ obtains the same membership as $f(x,y)$ in the set of prototype signals, then the noise has been suppressed.

The image is wavelet transformed with an appropriate wavelet. The coefficients of the resulting transform are then reduced through soft-thresholding [LaKi96], [Dono92] in the following manner:

$$\hat{G}_j = \begin{cases} G_j(x,y) - T; & G_j(x,y) > T \\ 0; & -T \le G_j(x,y) \le T \\ G_j(x,y) + T; & G_j(x,y) < -T \end{cases} \tag{4.5a}$$

$$T = Median(|G_1|)\frac{\sqrt{2\log(N)}}{\sqrt{N}} \tag{4.5b}$$

where $G_j(x,y)$ is the discrete wavelet transform of $g(x,y)$ in Eq. 4.1, T is the optimal reduction soft-threshold value, and N is the number of pixels in the image. The median of the coefficients of the first level of the wavelet transform of g is considered to be the noise level estimate based on the fine level coefficients of the image and the latter portion of the equation is the pure noise estimate [DoJo94]. The inverse transform of $\hat{G}(x,y)$ is performed to yield the denoised image.

An important issue to consider is selecting an appropriate wavelet to use in the wavelet transform. The wavelet must be chosen so that when the coefficients are reduced and the inverse wavelet transform is performed, there will be minimal loss of signal. A wavelet that models noise very well would be suitable. The coefficients would be more representative of the noise portion of the signal rather than the desired part of the signal. By shrinking the coefficients of the noise, only the noise is affected. So if very few of the coefficients modeled the desired signal, then the signal would not be affected.

Noise is generally a narrow, high frequency signal. A wavelet would have to be chosen so that it models such signals well but not broader, low frequency signals. The Daubechies-4 wavelet has a short support which would be expected to model short, impulse signals very well. Noise is uncorrelated so the removed signal, $g(x,y)- \hat{g}(x,y)$, should appear to have been generated through a random process.

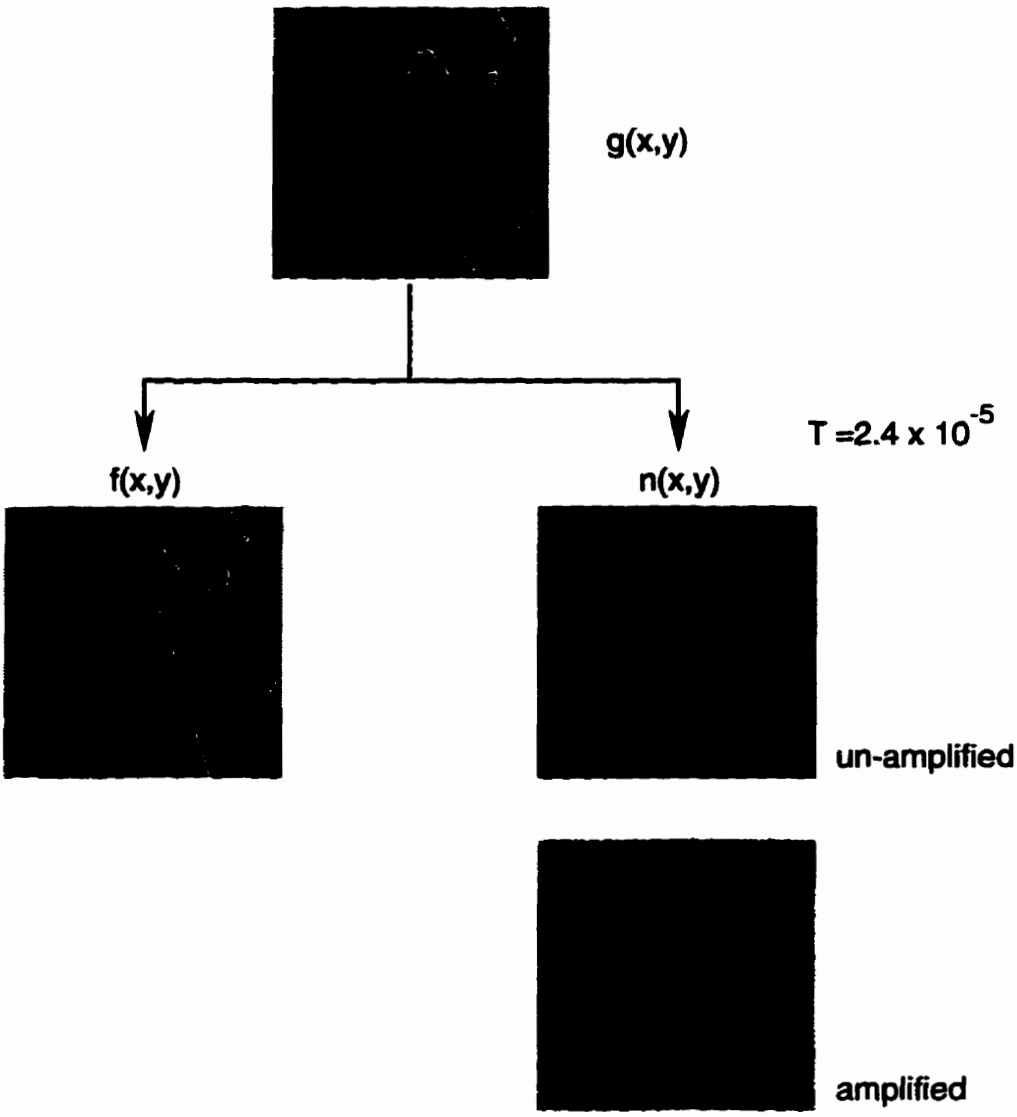


Fig. 4.2. Wavelet denoising of an image with a threshold of 2.4 x 10^{-5}. The un-amplified residue is shown to be uncorrelated, random noise.

Figure 4.2 shows the separation of the signal from noise contained in $g(x,y)$ using wavelet denoising with a threshold of $T = 2.4 \times 10^{-5}$. Uncorrelated noise is visible in the un-amplified residue. The amplified residue begins to show the pattern of the orginal image which indicates the technique is lossy.

4.3 Summary

This chapter has described two methods in which the additive noise contained in an image can be reduced. The first method, spatial smoothing, is a filtering technique that affects the raw pixels, thus affecting the desired signal as well as the noise. The implementation would be fast but the cost is the final quality of the image. Wavelet denoising, on the other hand, is a technique that attempts to alter the noise without greatly affecting the signal. More processing is required than with spatial smoothing, but the resulting quality is more satisfactory.

At this point, all of the necessary background and methodologies for fractal block coding have been established. In addition, two techniques have been described that will reduce the level of noise in the image resulting in more efficient fractal coding. The next chapter will discuss how the experimentation to support this thesis was implemented.

CHAPTER 5

IMPLEMENTATION OF DENOISED FRACTAL BLOCK CODING

This chapter describes the experimental implementation of the thesis. The organization of the software is described and illustrated using structured charts. While this thesis is independent of the hardware platform used, the hardware used for the experiments is described. Following that, a description of the flow of processing will be given. This section describes the steps that were performed for the experiments.

5.1 Software Organization

The software that is used to perform the experiments is divided into six main modules, shown in Fig. 5.1. The input/output, or I/O, module and the TRANSFORMS module perform low level block preparation and manipulation. The isometric transformations required for the generation of the fractal transform is executed in the TRANSFORMS module. The FSCL module handles the operations associated with the frequency sensitive competitive learning neural network. Codebook initialization and training is carried out here as well as the actual usage for the FSCL neural network, that is, image block classification. The FRACTALS module handles all operations that are necessary to generate the fractal code for each range block of the image and to reconstruct the image from the fractal code. The fifth module, ARITHMETIC, handles the arithmetic entropy encoding of the fractal code. The sixth module, MAIN, links the other five modules into a working program.

Each module is divided into two portions. There is the interfacing portion that allows for communication of the module to the other modules in the program. It includes declarations of public functions and data structures that other modules will need to have access to. The other portion of the module is the implementation which contains

declarations for private functions, data structures, and variables as well as the code to carry out the operation itself.

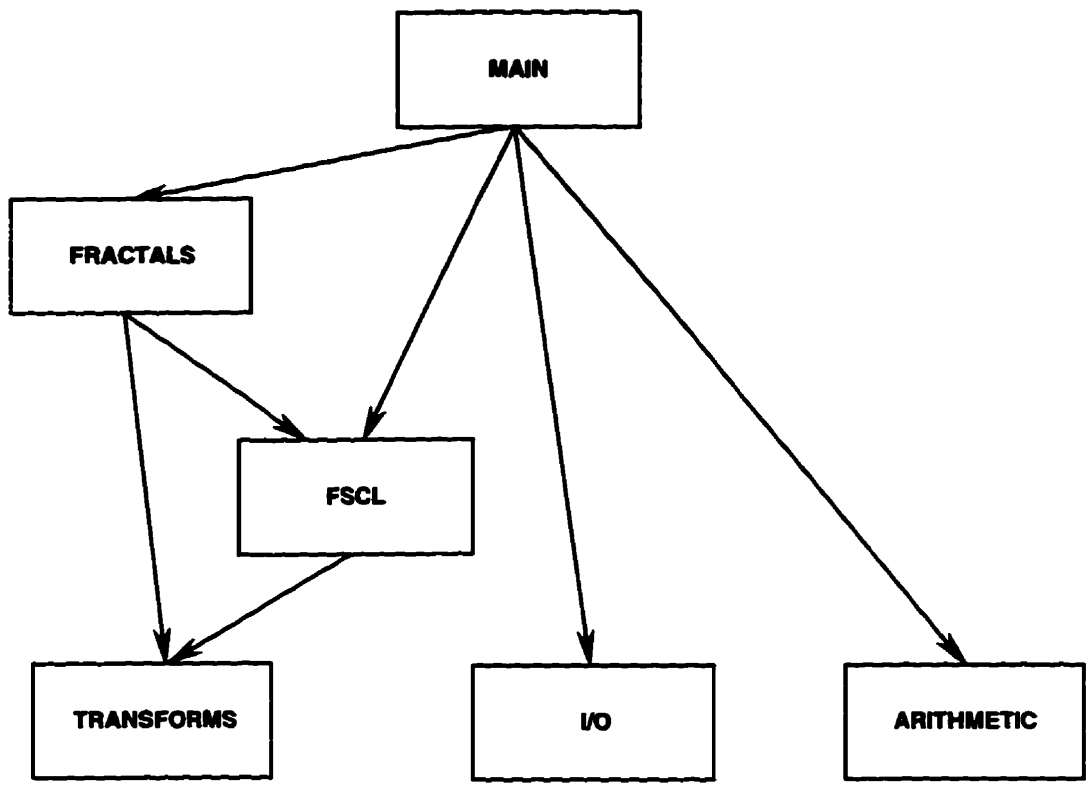


Fig. 5.1. The six modules of the reduced-search fractal block coding software.

The wavelet denoising portion of the software is divided into three main modules. As shown in Fig. 5.2, they are DWT, which performs the forward discrete wavelet transform, REDUCE, which reduces the coefficients, and IDWT, which performs the inverse discrete wavelet transform.

5.2 Development Environment

The initial work [WaKi93] was implemented on a Sun SparcStation 2. The second phase of this work, described in this thesis, was implemented on DEC Alpha workstations operating at 175 MHz with the DEC OSF/1 operating system. To illustrate the speed difference between the two machines, a process that took approximately 18 minutes on the

SparcStation 2 took only 5 minutes on the Alpha. While the FBC algorithm does not depend on what processor it is running on, a fast machine lends itself to providing quicker results so that batch execution of many experiments is possible in a very short period of time.

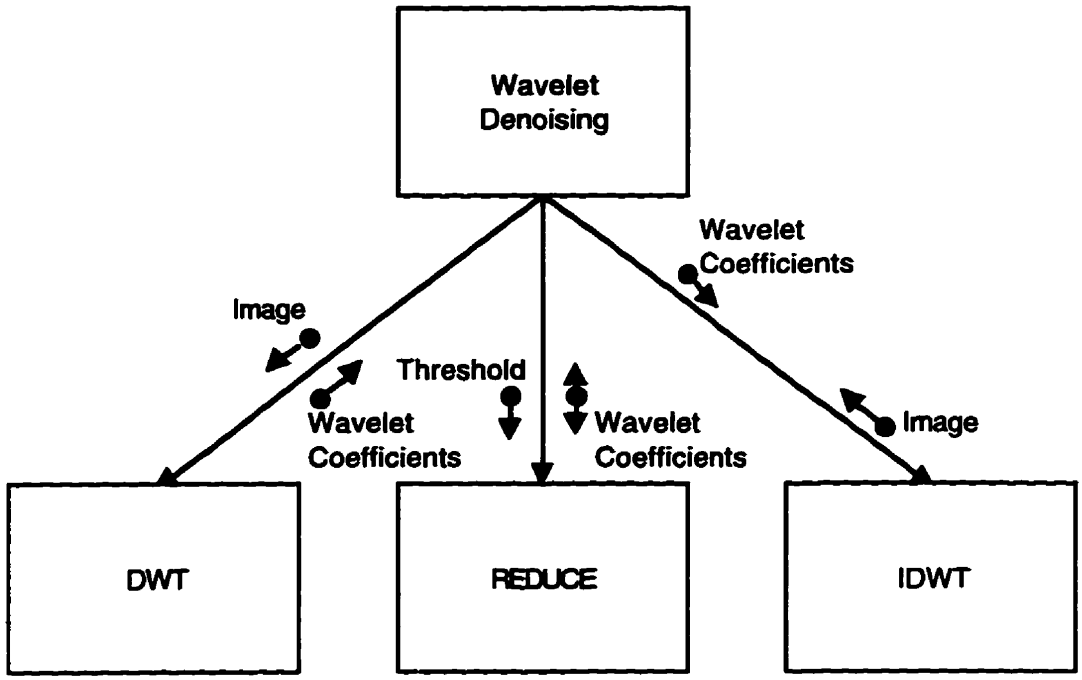


Fig. 5.2. Structure of wavelet denoising portion of the software.

The software was developed under Unix using a combination of X-Motif and X-Toolkit constructs. A graphical user interface (GUI) was designed to simplify experimentation. The GUI allows the user to change some parameters without having to recompile the software. It also allows the user to activate a batch processing mode in which a parameter is gradually changed and the image is fractal coded for each change in that parameter. The program window is composed of three parts. On the left of the window, the original image to be compressed is displayed. On the right, the reconstruction of the compressed image is shown. Below these two images, a graphical representation of the FSCL codebook is displayed. The GUI is shown in Fig. 5.3.

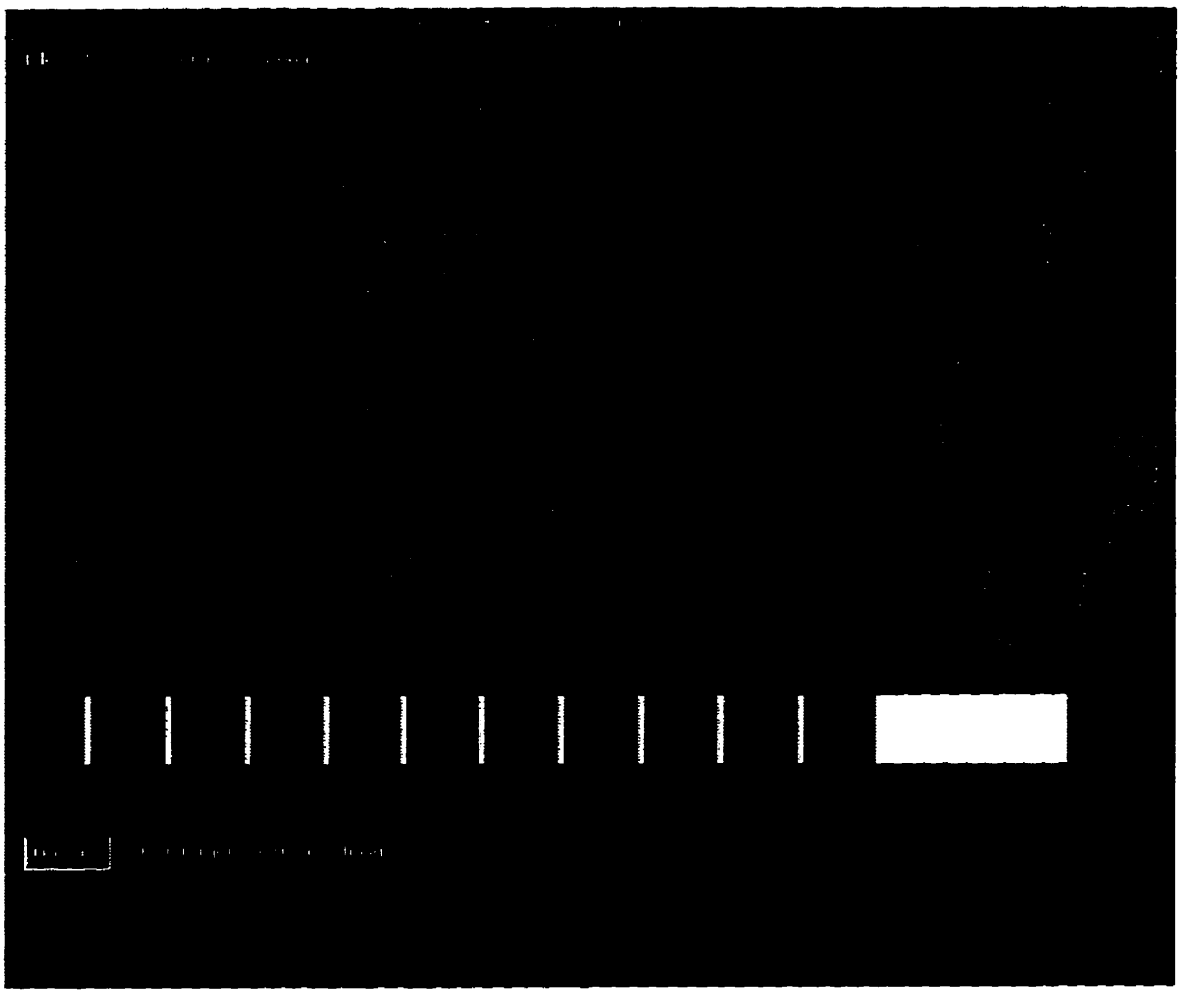


Fig. 5.3. The Graphical User Interface (GUI) for the reduced-search fractal block coding program.

5.3 Implementation of Experiments

The experiments conducted in this thesis all followed the same order as shown in Fig. 5.4. Initially, the FSCL codebooks must be trained but this occurred only once. To compress an image, it was subdivided into range blocks which are classified using the neural network. Following that was the actual fractal block coding process. The resulting fractal code was then losslessly compressed using arithmetic entropy encoding. At this point, the compression ratio was determined to indicate how efficient the FBC process was. However this does not complete the experimentation because a compression

technique can not be used unless there is a good reconstruction. The code was then entropy decoded and the FBC process is reversed.

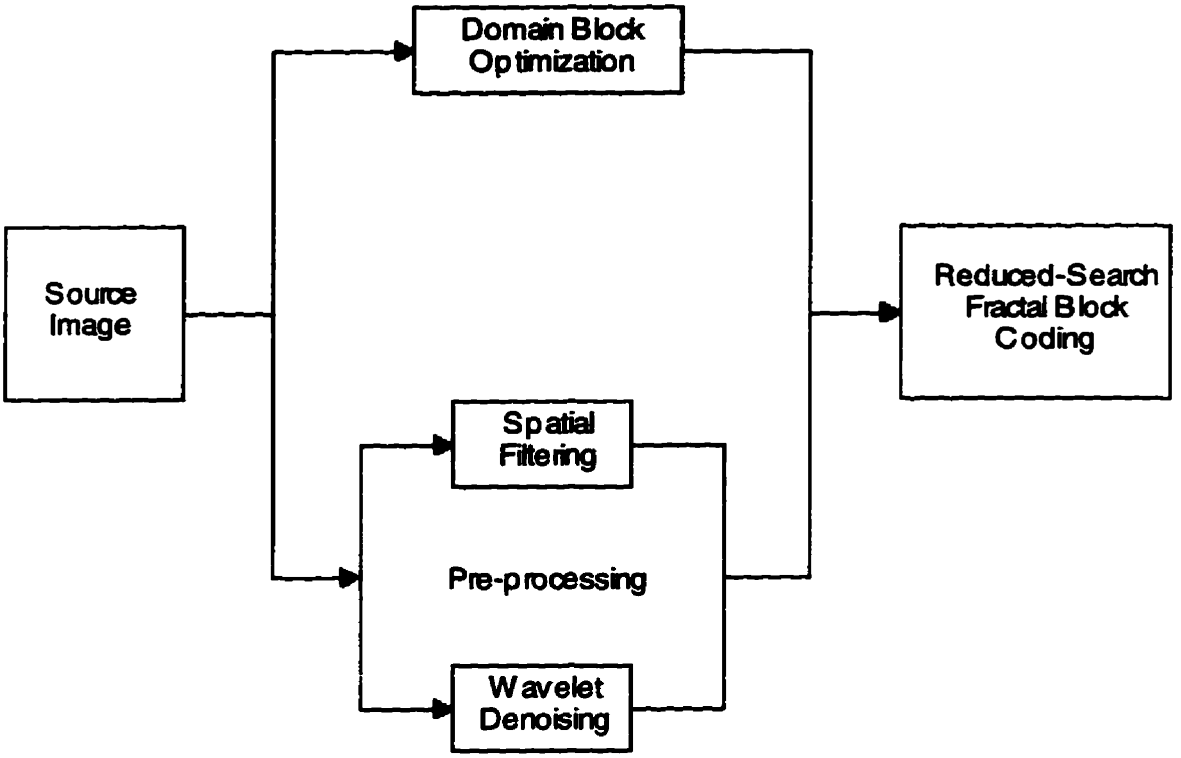


Fig. 5.4. Flow of experiments.

During the FBC process, domain blocks were sampled from the image sequentially from the top-left corner to the bottom-right corner of the image. For the experiments in this thesis, the FBC was attempted with the blocks being sampled at increments of one pixel, half of the domain block size, and one complete domain block width. In addition, domain blocks of different sizes were used. For simplicity, the domain block sizes were integer multiples of the range block size, which was eight. This way, the scaling factor in the fractal code would be an integer.

The pre-processing stage of FBC occurred before the image to be compressed entered the sequence described above. The sequence of the pre-processing stage started by spatially filtering the image or taking the DWT of the image. Following that, if, in the case of the DWT, the coefficients were shrunk by a pre-determined value and the inverse DWT

was performed. The result was then fed into the FBC sequence. The entire sequence was repeated with a different filter mask size or coefficient shrinking value.

5.4 Definition of Compression Ratio and Quality

The measures necessary to determine how good the compression is, are compression ratio and peak-signal-to-noise ratio. The compression ratio is an indication of how efficient the fractal block coding was for a particular image. The compression ratio depends on the image used. The more self-affine features there are, the more efficient the coding will be, resulting in a higher compression ratio. The compression ratio is calculated by taking a measure of the size of the original image and dividing by a measure of the size of the fractal code. Each range block of the image is represented by a pointer to a domain block, a pointer to the best isometric transform, a scaling coefficient, and a translation coefficient. For an *N*x*N* image with *dxd* domain blocks, the number of bits, b_D, needed to address a unique domain block is [Wall93]

$$b_D = 2 \cdot \log_2(N-d). \tag{5.1}$$

There are eight isometric transforms so $b_I = 8$ bits are needed to point to the best transform. For the scaling and translation coefficients, $b_A = 11$ bits and $b_T = 9$ bits respectively are sufficient to allow fixed point scaling values in the range, ±4.0, and integer translation values in the range, ±255. The total number of bits, then, of a coded range block is

$$b_R = b_D + b_I + b_A + b_T. \tag{5.2}$$

The compression ratio, *CR*, can be calculated by dividing the number of bits needed for an *rxr* range block in the coded image by b_R shown by

$$CR = \frac{r^2 b_p}{b_R}, \tag{5.3}$$

where b_p is the number of bits per pixel in the original image. The compression ratio can be adjusted by choosing differently sized range blocks or changing the size of the domain

pool. By reducing the size of the domain pool, few bits are needed to address a unique domain block which will result in a higher compression ratio. The sacrifice, though, is the quality of the reconstructed image. The image quality is determined by calculating the peak-signal-to-noise ratio, *PSNR*.

The measure of quality of the reconstructed image is calculated by

$$PSNR = 10 \cdot \log_{10} \left(\frac{R^2}{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left(\hat{f}(x,y) - f(x,y) \right)^2 / MN} \right) \tag{5.4}$$

where R is the full range of values that a pixel in the image has, $\hat{f}(x,y)$ is the value of a pixel in the reconstructed image, $f(x,y)$ is the value of a pixel in the original image, and M and N are the dimensions of the image. The PSNR is measured in dB. A good quality reconstruction would have PSNR value of approximately 32 dB. However, it should be noted here that PSNR is not the absolute method of determining the quality of a reconstruction. An image with a low PSNR is not necessarily of poor quality nor is an image with a high PSNR necessarily a high quality image. However, by establishing standard quality measures, comparisons can be made among the various reconstructed images in these experiments.

Another measure of quality is *root-mean-square error*, rms. Given an MxN image, the root-mean-square error, $error_{rms}$, is

$$error_{rms} = \sqrt{\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left(\hat{f}(x,y) - f(x,y) \right)^2} . \tag{5.5}$$

This error is a function of the noise only, given by the difference between two images as opposed to the PSNR. The unit of rms error, therefore, is grey level.

5.5 Summary

This chapter has described the implementation of the experiments to support this thesis. The software was described to be organized into six modules: FRACTALS,

FSCL, IO, TRANSFORMS, ARITHMETIC, and MAIN. The environment in which the experiments were performed was described. All experiments were performed on a DEC Alpha workstation. Following that, was a description of the flow of processing of the experiments with and without image pre-processing. Finally, the calculations that must be performed to interpret the results were described. The compression ratio and reconstruction image quality, measured by PSNR, were given as formulae.

CHAPTER 6

EXPERIMENTAL RESULTS AND DISCUSSION

This chapter describes the results of the experiments performed to study a technique to increase the efficiency of reduced-search fractal block coding by reducing additive noise. Greyscale images were first reduced-search fractal block coded with various alterations to the domain block configuration to obtain results of coding without image pre-processing. These results were then compared with pre-processed images that have had their additive noise reduced. Comparisons were also made between spatial filtering and wavelet denoising.

This chapter first describes the choice of the images used in the experiments. The images contain features that span from smooth to highly detailed areas. These images will help determine how effective the fractal block coding technique is on different kinds of features and what kind of effects will result from noise reduction. Following that, is a presentation of the results of the experiments along with a discussion to explain why the results are as they are. The chapter is closed with a summary of the key results that were used to draw conclusions.

6.1 The Source Images

The images used for the experiments are named *Lena*, *Goldhill*, and *Peppers* and are shown in Figs. 6.1, 6.2, and 6.3, respectively. The image *Lena* contains fine details in the hat and smooth features in the background and shoulder. Since one of the applications of FBC is for the coding of head and shoulder images of people, this image is a good test of that application. The image *Goldhill* contains many fine details that test how well the FBC process handles such input. Finally, the image *Peppers* contains large smooth features with sharply defined edges. This image was chosen to show the effect fractal block coding has on sharp edges. Another reason why these images were selected is that they have become quite common in the image compression research community. By using

such standards, results can be shared among researchers who are already familiar with the features contained within the images.

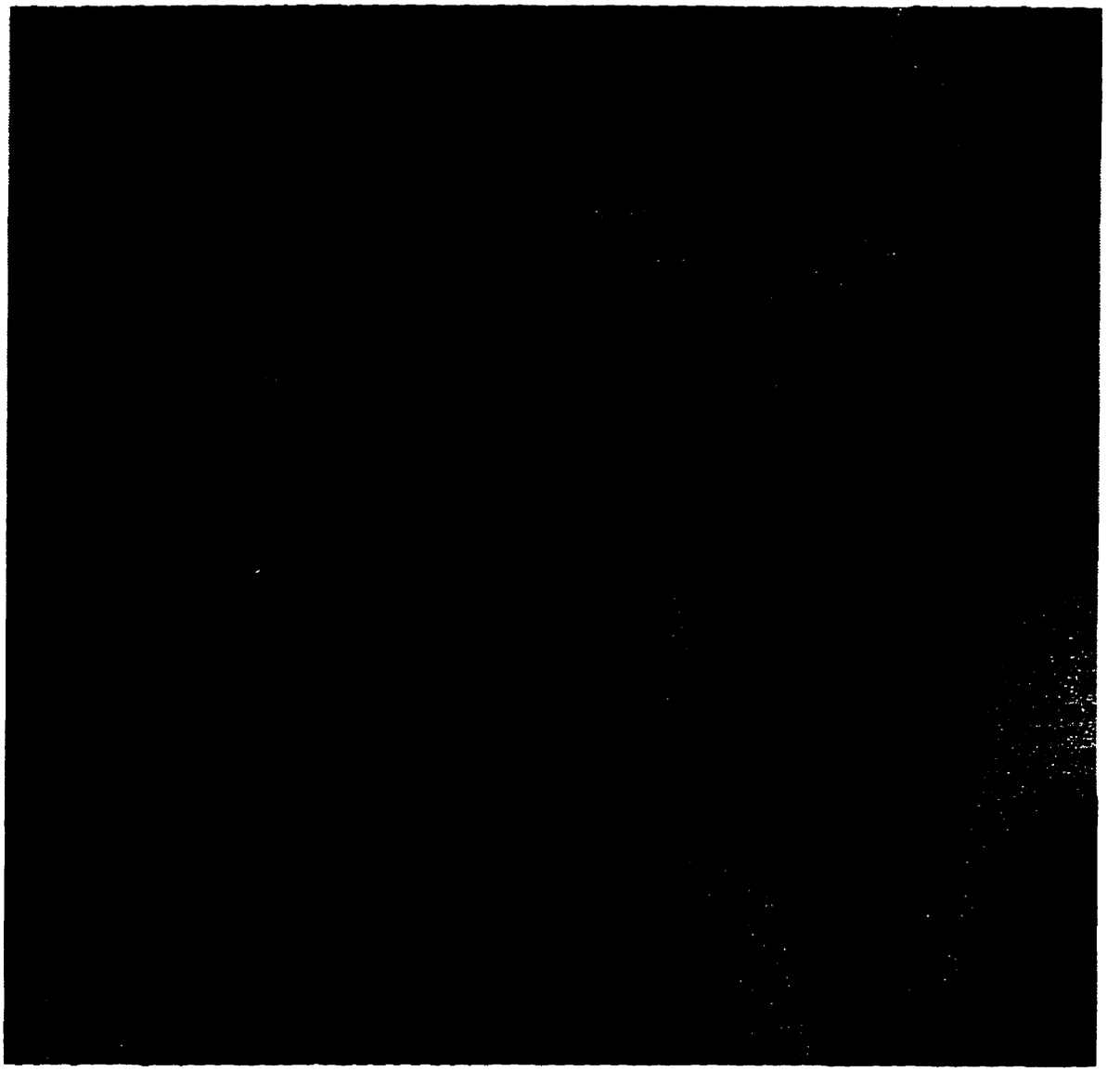


Fig. 6.1. The image *Lena* . (Shown at 84.3% original size.)

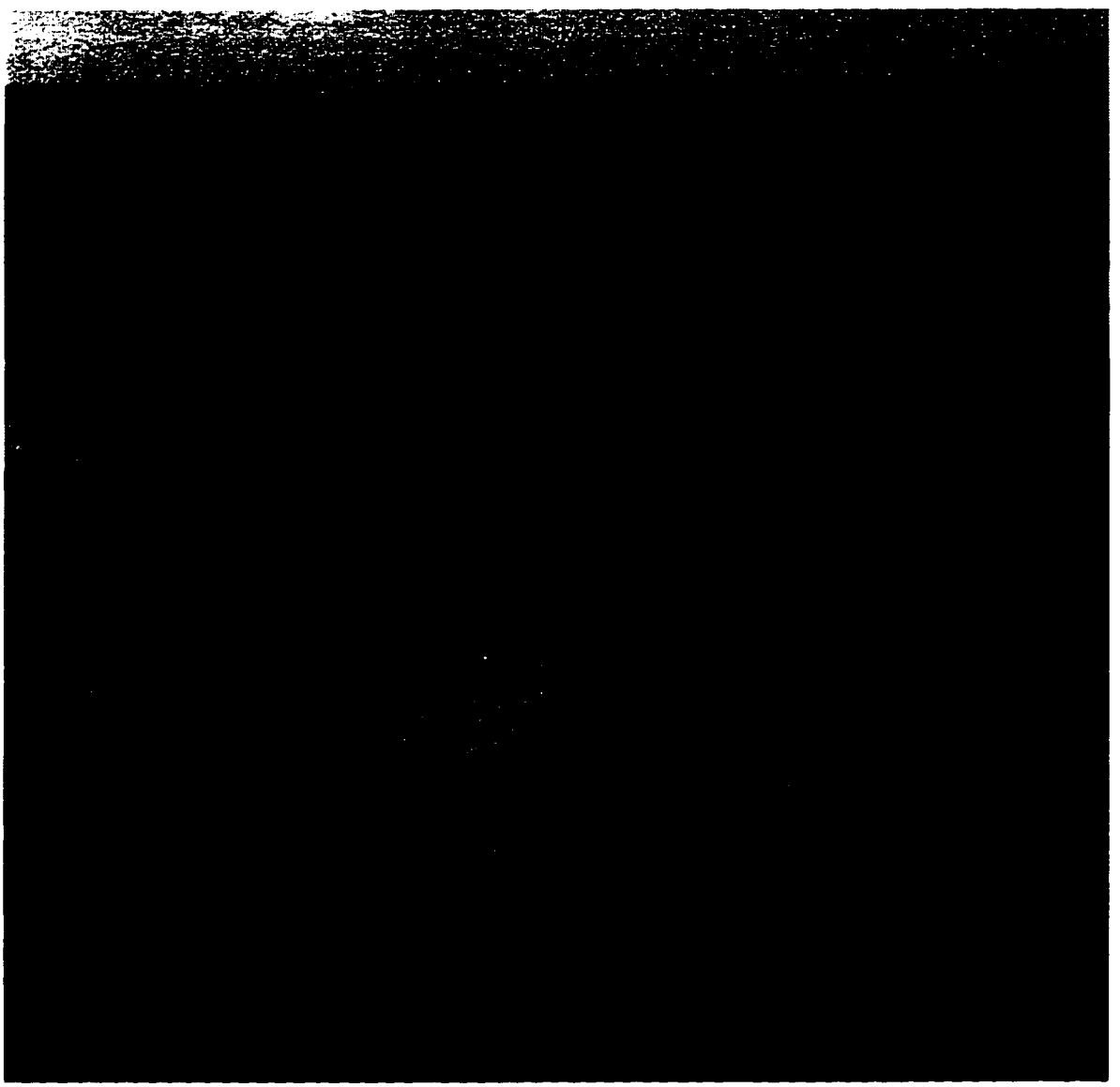


Fig. 6.2. The image *Goldhill*. (Shown at 84.3% original size.)

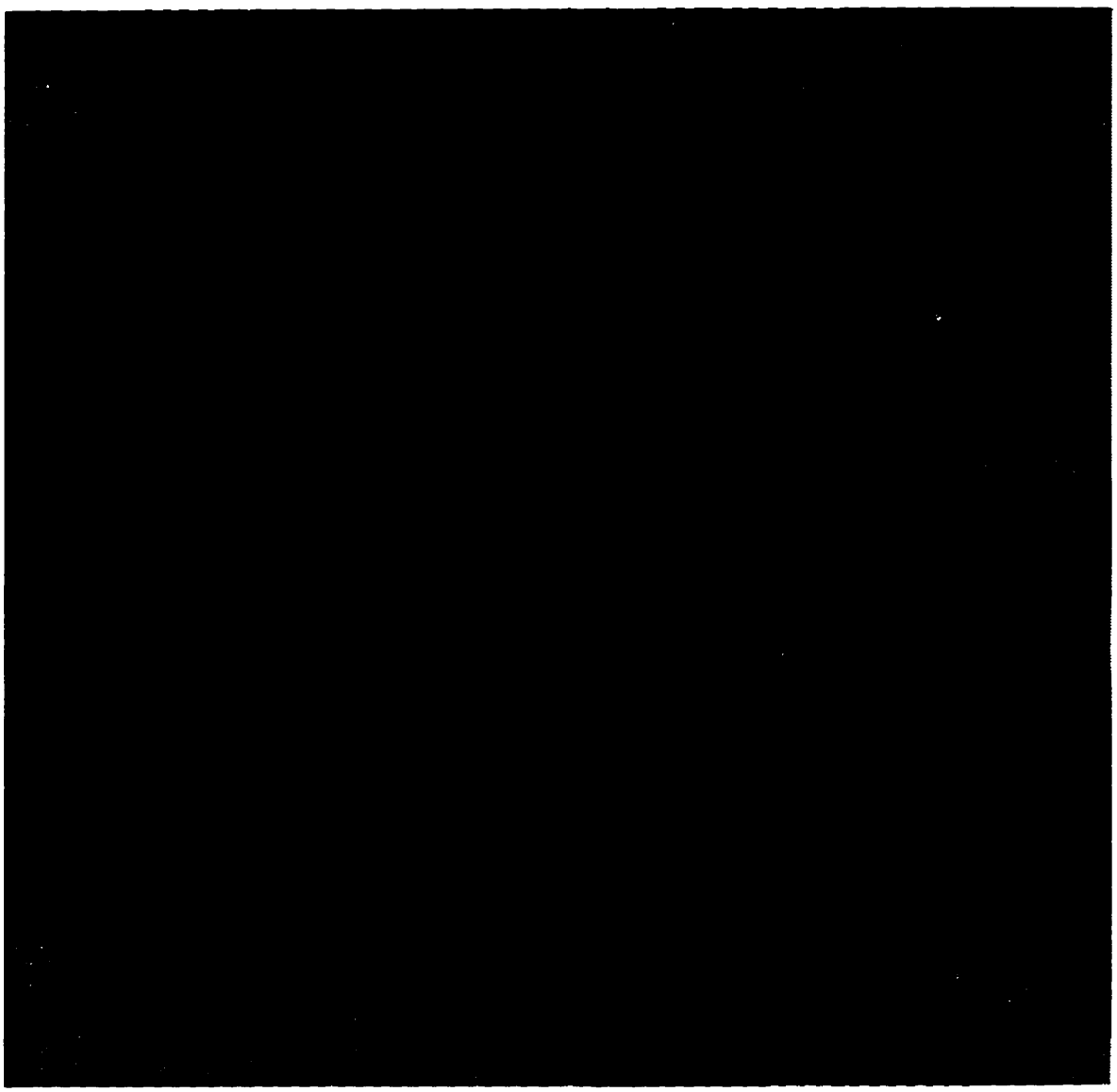


Fig. 6.3. The image *Peppers* . (Shown at 84.3% original size.)

Table 6.1. Entropies and lossless compression ratio of the test images.

Image	Entropy (bpp)	Best Lossless Compression Ratio
Lena	7.445526	1.0745:1
Goldhill	7.477780	1.0698:1
Peppers	7.37436	1.0848:1

The images are each 512x512 pixels in size and are 8 bits per pixel (bpp) greyscale. Table 6.1 shows their respective entropies and their best lossless compression ratios. The best lossless compression ratios gives an idea of how much the image can be compressed without any loss of signal. It can be seen that the lossless compression ratio is not very high which makes lossy compression techniques, such as FBC, favourable for the compression of these images. Also, note that images with lower entropies can be compressed further than images with higher entropies. This point is relevant when the images are pre-processed to lower their entropies.

As described in Chapter 4, images generally contain additive noise as in Eq. 4.1 and repeated here

$$g(x,y) = f(x,y) + n(x,y). \tag{6.1}$$

The additive noise, $n(x,y)$, raises the entropy of the signal, $g(x,y)$. By pre-processing the image, either by spatial filtering or wavelet denoising, the noise can be reduced, thus reducing the entropy. The reduced entropy should allow for more efficient coding because the level of redundancy contained within the image has been increased.

6.2 Fractal Block Coding Without Pre-Processing

This section describes the results that have already been obtained with reduced-search fractal block coding of images [Wall93] without any pre-processing. The key measurements that were made of the experiments were of the reconstruction image quality

and the compression ratio. It was desired to have both a high image quality and a high compression ratio.

The FSCL codebook was trained using the image of *Lena* because it contains a range of features from smooth, low frequency features to rough, high frequency features. The codebook generated from it would then span a similar range of features. Through experimentation [Wall93], it was found that the optimal number of codewords to be sufficient to code the image is 11. In that work it was also found that the image used to train the codebook did not greatly affect the fractal coding process. In other words, any image can be used to train the codebook.

The set of experiments performed were based on the domain block size and how the domain blocks are selected, which in turn, affected the number of domain blocks in the set. Three sizes of domain blocks were tested: 16x16, 24x24, and 32x32, all of which are integer multiples of 8 which is the width of a range block. The manner in which the domain blocks were sampled from the source image was also modified. During the domain block search, each domain block is sampled from the image consecutively, starting in the top-left corner. There can be a varying number of pixels that separate the top-left corner of each domain block as shown in Fig. 6.4. In these experiments, that value was varied from an increment of one pixel, half the width of the domain block, and the whole width of the domain block. Table 6.2 shows the number of domain blocks available in the set of domain blocks for each block size and increment value in a 512x512 image.

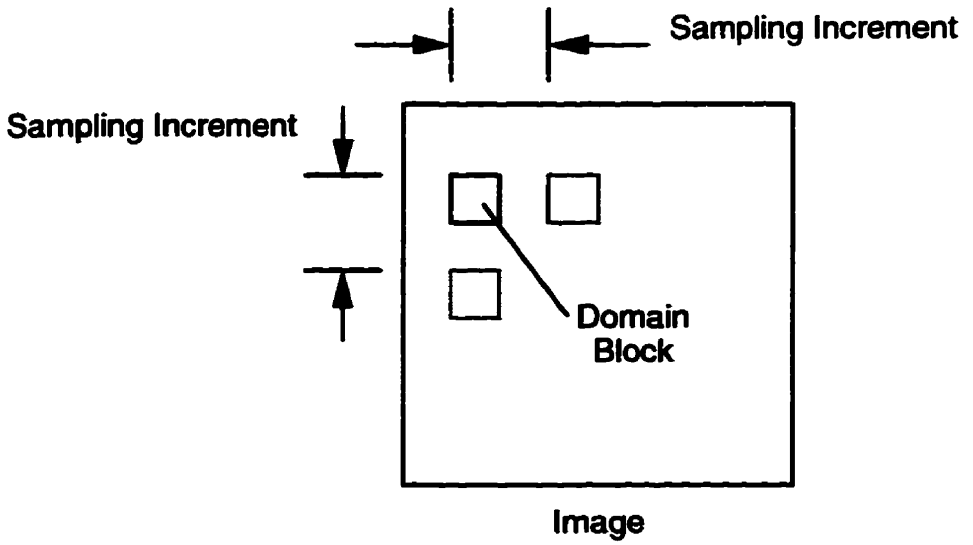


Fig. 6.4. Domain block sampling.

Table 6.2. Number of available domain blocks in a 512x512 image for various domain block specifications.

Domain block size (pixels)	Domain block increment (pixels)	Number of available domain blocks
16x16	1	246 016
	8	3 844
	16	961
24x24	1	238 144
	12	1 653
	24	413
32x32	1	230 400
	16	900
	32	225

It is clear from these numbers that with each domain block size, more time is needed for single pixel increments but the domain space is more feature-rich than that of the domain spaces with larger increments. In that case, one would expect higher quality reconstructions. Furthermore, with larger domain spaces, there exists the potential that more domain blocks will be used to code the image than with smaller domain spaces. This implies that the compression ratio will be lower. Experiments were performed on all three test images to determine optimal domain block parameters based on compression ratio and reconstruction quality in terms of PSNR and rms error.

Every combination of domain block size and domain sample increment from Table 6.2 was attempted on each of the test images. For each domain block size, the domain block size was viewed as being constant while the sampling increment was increased from one pixel, to half the domain block width, to the full domain block width. Tables 6.3, 6.4, 6.5 and show the results of the experiments for *Lena*, *Goldhill*, and *Peppers* respectively. The number of iterations indicate how many iterations were necessary to reconstruct the image where any more iterations would not appreciably change the quality of reconstruction.

The highest compression ratio was observed for the set with the highest domain block sampling increment. As expected, with the smaller domain space, there are fewer blocks to choose from and therefore, a particular domain block appears more frequently in the fractal code. This redundancy is exploited at the arithmetic entropy stage to increase compression. The best quality reconstruction was observed for the set with the sampling increment of one pixel. Again, this is expected because of the larger domain space, the feature space is larger which produces a better quality reconstruction. The compression ratio was the lowest in this set as well.

Table 6.3. Domain block variations for reduced-search FBC of *Lena*.

Domain Block Size	Sampling Increment	Compression Ratio	Quality (PSNR)	Quality (rms error)	No. of Iterations
16x16	1	13.8400:1	32.8051	5.8383	5
	8	16.7493:1	31.5269	6.7639	4
	16	17.9563	30.8430	7.3180	4
24x24	1	13.8627:1	32.6436	5.9479	4
	12	17.4495:1	31.2504	6.9824	3
	24	18.7005:1	30.5319	7.5848	2
32x32	1	13.9091:1	32.4164	6.1055	3
	16	18.0168:1	30.8663	7.2983	3
	32	19.4498:1	30.0777	7.9920	3

Table 6.4. Domain block variations for reduced-search FBC of *Goldhill*.

Domain Block Size	Sampling Increment	Compression Ratio	Quality (PSNR)	Quality (rms error)	No. of Iterations
16x16	1	13.9135:1	30.7675	7.3819	9
	8	16.8289:1	29.6847	8.3619	4
	16	18.0577:1	29.1094	8.9345	4
24x24	1	13.9298:1	30.7659	7.3832	8
	12	17.5594:1	29.4592	8.5818	5
	24	18.8335:1	28.8965	9.1562	5
32x32	1	13.9505:1	30.6362	7.4943	5
	16	18.1427:1	29.1577	8.8849	3
	32	19.5717:1	28.3175	9.7873	3

Table 6.5. Domain block variations for reduced-search FBC of *Peppers*.

Domain Block Size	Sampling Increment	Compression Ratio	Quality (PSNR)	Quality (rms error)	No. of Iterations
16x16	1	13.9290:1	34.2829	4.9249	8
	8	16.8203:1	33.0912	5.6491	8
	16	18.0416:1	32.3982	6.1183	8
24x24	1	13.9394:1	34.3301	4.8982	5
	12	17.5101:1	32.7915	5.8474	9
	24	18.8079:1	31.9863	6.4154	8
32x32	1	13.9535:1	34.1048	5.0269	6
	16	18.1164:1	32.4058	6.1129	3
	32	19.5542:1	31.3423	6.9092	3

In all cases, it was also observed that compression time ranged from 20 to 30 minutes for all images with the 16x16 domain block and smallest sampling increment. The time was reduced to seconds when the 24x24 and 32x32 domain block were used with the half domain block width increment, which in this case, is 12 and 16 pixels respectively. The use of 24x24 domain blocks with 12 pixel increments gave a reasonable compromise between compression ratio, reconstruction quality, and implementation time. From Table 6.2 there are 1653 unique blocks in the domain pool which would take 11 bits to address each one. The compression ratio, then, before entropy encoding is 15.06:1. The pre-processing experiments were performed on that subset of domain block parameters. Also, note that fewer iterations were required at reconstruction with the largest domain blocks and largest sampling increment. For the case of 24x24 domain blocks with twelve pixel

increment the number of iterations required for *Lena*, *Goldhill*, and *Peppers*, were 3, 5, and 9, respectively. The incremental reconstruction of *Lena* is shown in Fig. 6.5. The reconstructed image of *Lena* is shown in Fig. 6.6. The range block

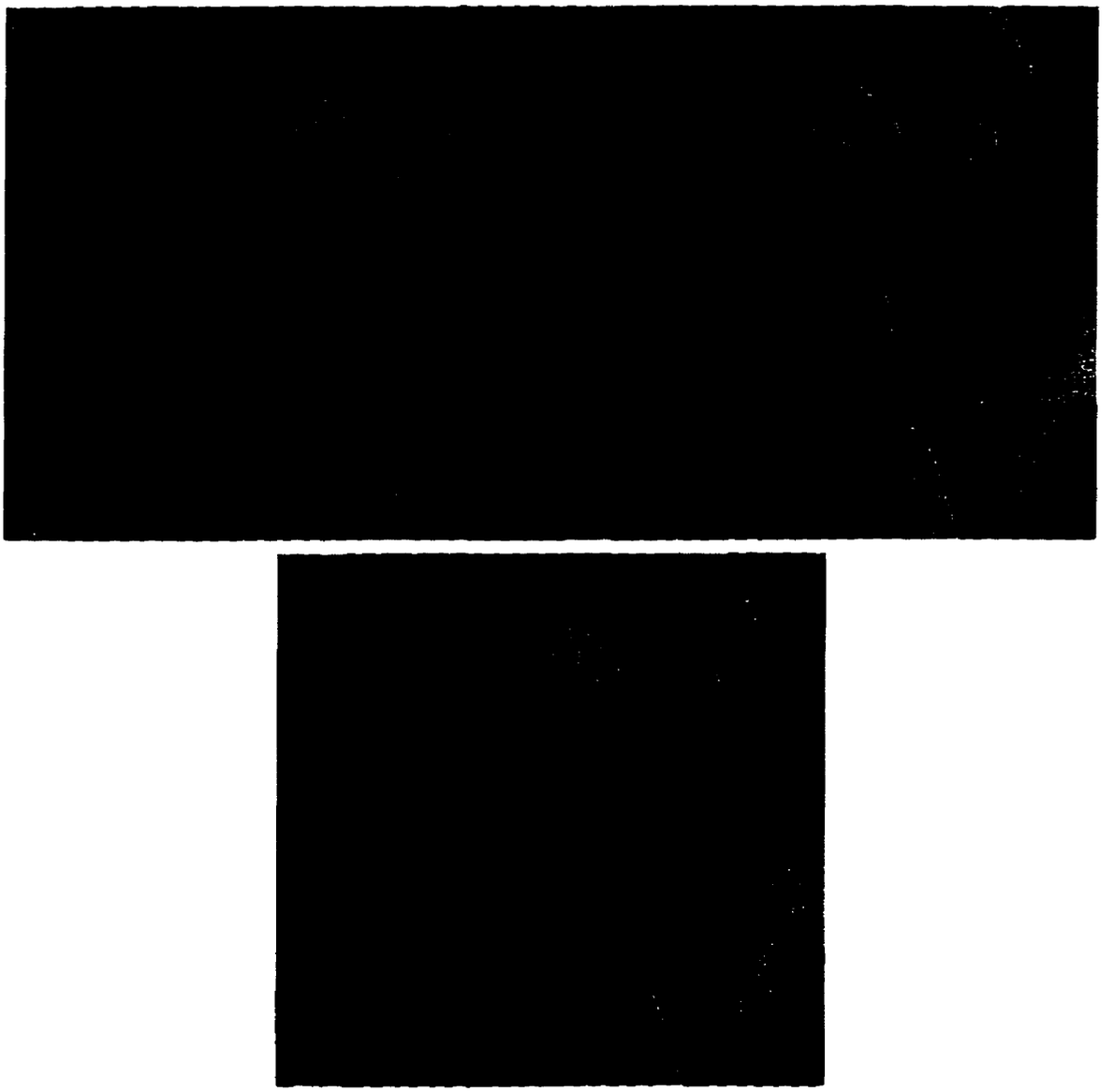


Fig. 6.5. The incremental reconstruction of *Lena*. The images are at 42% of the original size.

size and the domain block size was 8x8 and 24x24 respectively, resulting in a block scaling factor of 3. The reconstruction PSNR was 31.25 dB with a compression ratio of 17.4:1. While the reconstruction is recognizable, many fine details are lost. The details in the feathers of the hat and the eyes are distorted but the smooth features of the hat and shoulder

are well reconstructed. Some artifacts resulting from the block oriented coding algorithm are evident in the feathers in the hat. Sharp edges were also well preserved. Using the same block sizes, the compressed images of *Goldhill* and *Peppers* are shown in Figs. A.1 and A.2 respectively. A considerable amount of fine detail was lost in the reconstruction of *Goldhill* but the reconstruction *Peppers* , which contains very few fine features, was of higher quality.

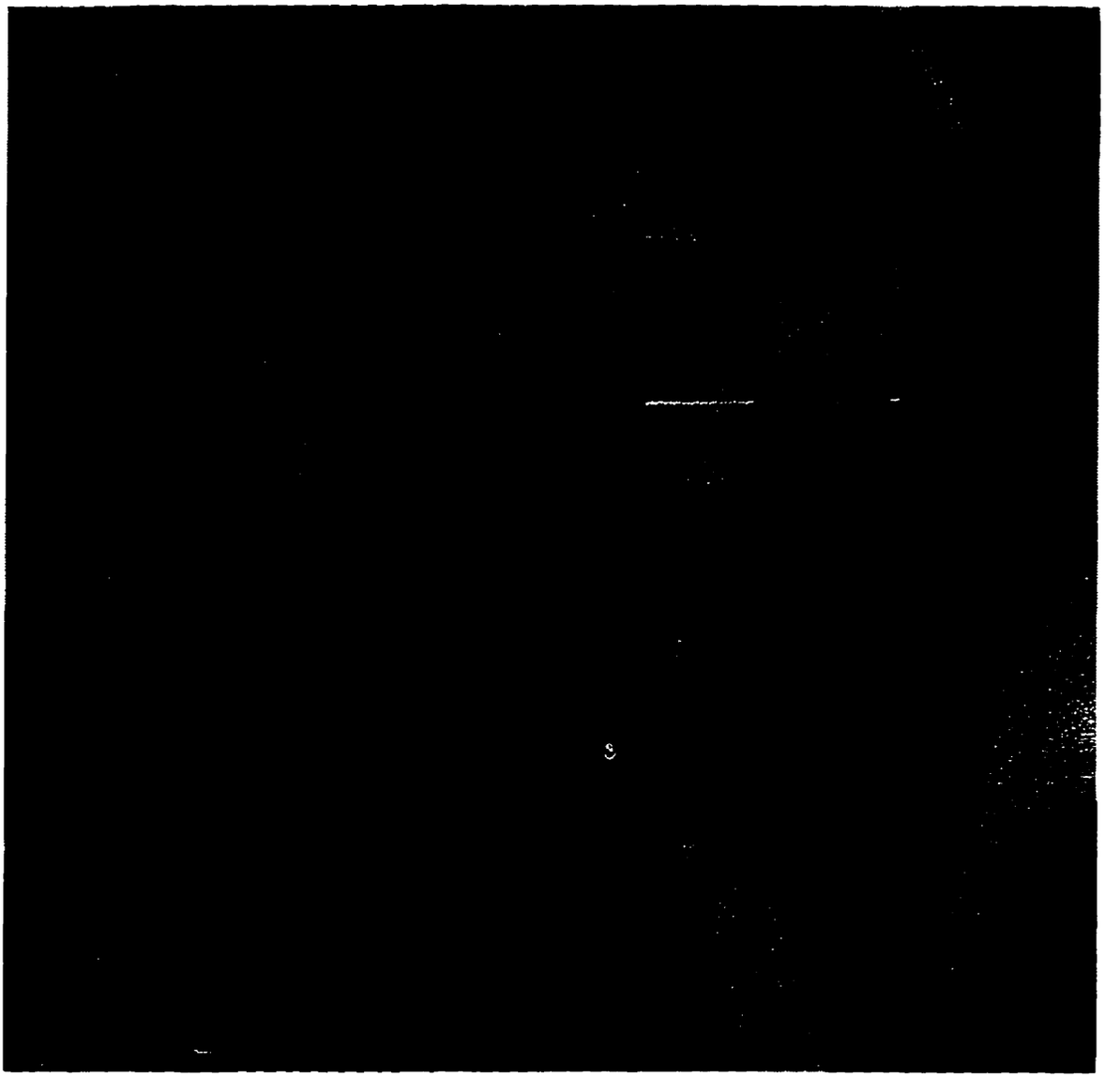


Fig. 6.6. Reconstructed *Lena* with PSNR = 31.25 dB and compression ratio = 17.4:1. No pre-processing was used. This image is shown at 84.3% size.

6.3 Fractal Block Coding with Pre-Processing

6.3.1 Spatial Filtering

To illustrate the effects of spatial filtering, each image was pre-processed by convolving an averaging, or smoothing, filter with each image. The mask sizes that were tested were 3x3, 5x5, 7x7, 9x9, and 11x11. Table 6.6 and Table 6.7 shows the PSNRs relative to the original and entropies of each image after filtering. This indicates how much the image degraded and became more redundant before the image is actually compressed. Some examples are shown in Fig. 6.7.

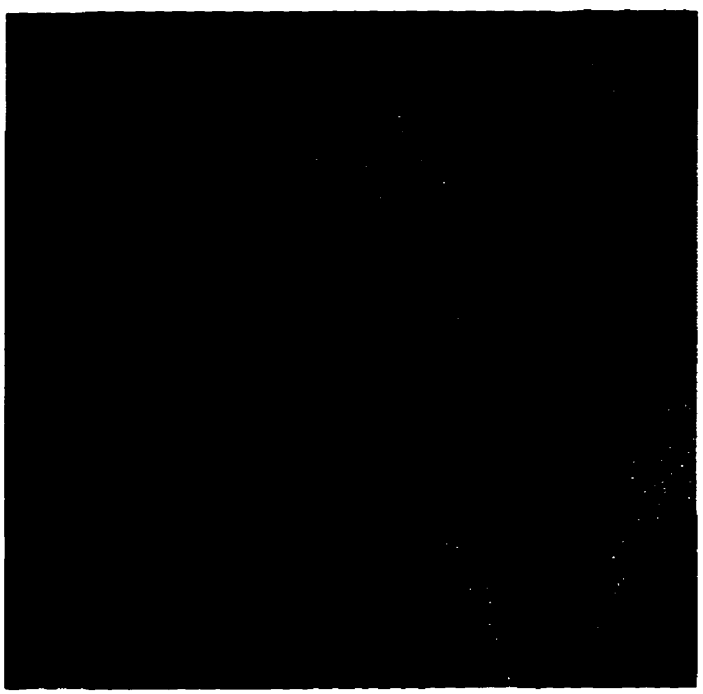


Fig. 6.7 (a) Filtered with 3x3 mask.

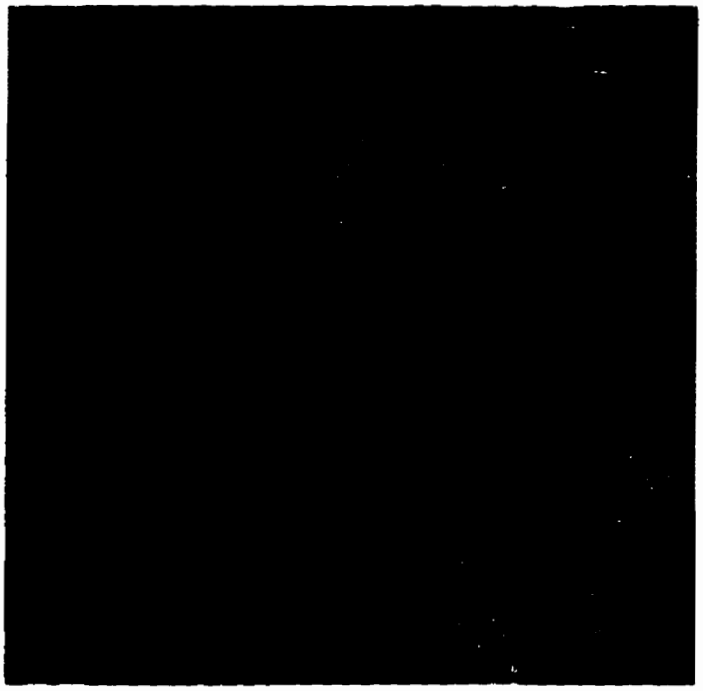


Fig. 6.7. (b) Filtered with 7x7 mask.

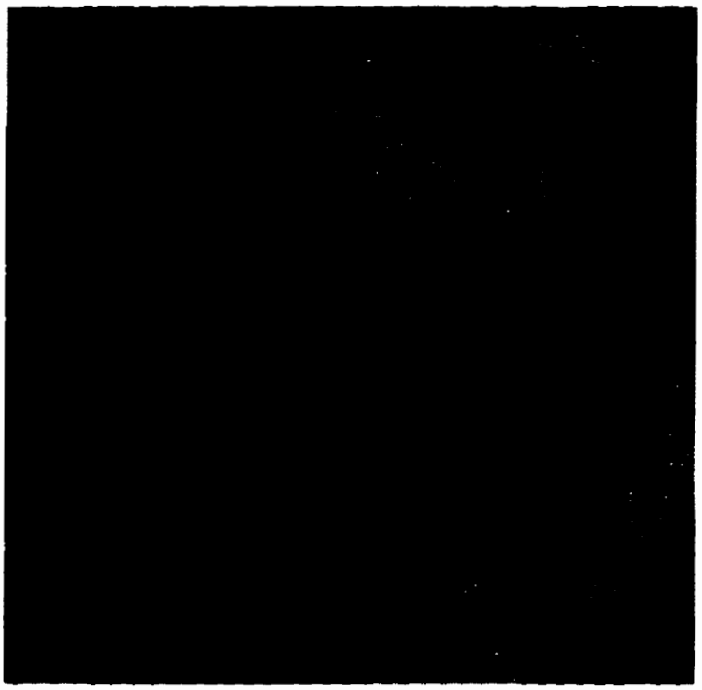


Fig. 6.7. (c) Filtered with 11x11 mask.

Table 6.6. Signal degradation (PSNR) with varying filter mask sizes.

	3x3	5x5	7x7	9x9	11x11
Lena	33.716145	29.620741	27.520794	26.181154	25.203754
Goldhill	31.059834	28.139589	26.701481	25.680073	24.971323
Peppers	30.875515	29.159481	27.823071	26.728603	25.814837

Table 6.7. Entropy (bpp) with varying filter mask sizes.

	No Filtering	3x3	5x5	7x7	9x9	11x11
Lena	7.445526	7.409656	7.382763	7.359675	7.337998	7.317272
Goldhill	7.477780	7.430872	7.400693	7.374064	7.351524	7.331128
Peppers	7.374360	7.347542	7.335463	7.323257	7.311070	7.297602

It is clear that while the results of using large mask sizes lowers the entropy, it is at the cost of relevant detail. Since the filter works directly on the raw pixels themselves, image degradation can not be avoided. Subjectively, only the images filtered with the 3x3 mask appear to be of reasonable quality to be useable.

Table 6.8. Reconstruction quality of smoothed images.

		With respect to smoothed			With respect to original		
Test Image	Filter Mask nxn	PSNR (dB)	rms (bpp)	Its.	PSNR (dB)	rms (bpp)	Its.
Lena	3	34.50	4.80	2	30.49	7.62	3
	5	37.87	3.26	2	28.75	9.31	4
	7	40.24	2.48	2	27.22	11.10	3
	9	42.00	2.03	2	26.04	12.72	3
	11	43.30	1.74	2	25.11	14.16	8
Goldhill	3	35.28	4.39	2	28.81	9.24	2
	5	38.25	3.12	4	27.59	10.64	2
	7	40.58	2.39	3	26.46	12.12	2
	9	42.35	1.95	4	25.57	13.43	2
	11	43.54	1.70	6	24.90	14.51	2
Peppers	3	37.14	3.55	2	29.72	8.33	3
	5	39.51	2.70	2	28.66	9.41	4
	7	41.34	2.19	2	27.59	10.65	4
	9	42.81	1.85	3	26.60	11.93	3
	11	43.84	1.64	3	25.73	13.18	6

The set of images were reduced-search fractal block coded using 24x24 pixel domain blocks and 12 pixel domain block sampling increments. The results are show in Table 6.8 and Table 6.9. With increasing mask sizes, the compression ratio increases as

did the PSNR relative to the filtered image. This was expected because with the decreased entropy, the image should be more susceptible to fractal block coding. However, the PSNR relative to the original source image is considerably less than that of the PSNR of the filtered images before coding with respect to the original source images. The increase in the quantitative measure of quality simply indicates that the reduced-search FBC algorithm performs well on a highly self-similar image, which is the result of severe smoothing. The result, however, at large filter mask sizes, is a severely distorted image relative to the original. Fig. 6.8 shows a sample of a fractal block coded image of *Lena* with 3x3 filtering. The increase in compression ratio was 1.3%.

Table 6.9. Compression ratios of smoothed images.

Test Image	Filter Mask nxn	Compression Ratio
Lena	3	17.67
	5	17.80
	7	17.93
	9	18.04
	11	18.10
Goldhill	3	17.77
	5	18.00
	7	18.21
	9	18.45
	11	18.57
Peppers	3	17.69
	5	17.84
	7	17.97
	9	18.05
	11	18.17

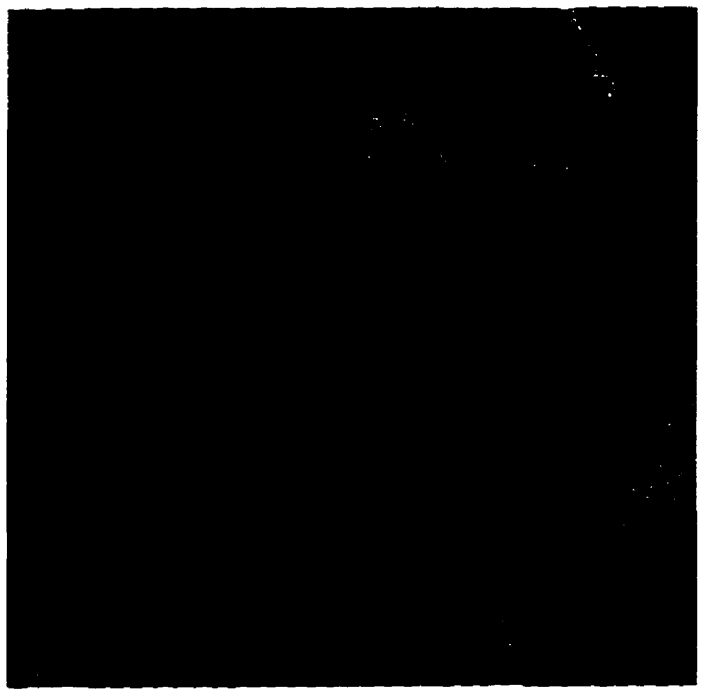


Fig. 6.8. Reconstructed image of *Lena* with 3x3 filtering. The PSNR = 30.49 dB and the compression ratio was 17.7:1. The image is shown at 50% actual size.

6.3.2 Wavelet Denoising

For the wavelet denoising experiments, the member of the Daubechies family of wavelets was tested as well as the wavelet coefficient shrinkage threshold. Daubechies-4, -8, -12, and -20 were used. They vary in roughness from the very rough Daubechies-4 to the very smooth Daubechies-20. Each test image was first discrete wavelet transformed with the given wavelet. The wavelet coefficients were reduced by a threshold, T, by Eq. 4.5 and the image was then reconstructed through the inverse DWT. Fifty images were produced using values of T in the range {0, 0.050} at increments of 0.001. Each image was then fractal block coded using domain blocks of size 24x24 and domain sampling increment of 12 pixels.

The optimal values of the threshold given by Eq. 4.5b are functions of the median of the first-scale coefficients of the wavelet transform and the pure noise level. For each

wavelet, the calculated optimal T for each image was approximately the same and are listed in Table 6.10. The pure noise level, which is a function of the size of the image only, was calculated to be 0.006.

Table 6.10. Optimal threshold values.

Test Image	Optimal Threshold
Lena	2.4×10^{-5}
Goldhill	3.6×10^{-5}
Peppers	2.8×10^{-5}

Table 6.11. Wavelet coefficient shrinking of *Lena*.

Threshold	Compression Ratio	PSNR (dB)
2.4×10^{-5}	17.46	31.15
0.006	18.82	30.42
0.020	19.06	29.92
0.040	19.18	29.04
1.000	19.43	19.13

While the threshold values were ranged from 0.000 to 0.050 at 0.001 increments to establish a trend, the results of using a few particular threshold values with the Daubechies-4 wavelet on the image *Lena* are shown in Table. 6.11. The PSNR value indicates the change in quality relative to the original image. It was also observed with all three test subjects, the number of iterations required to reconstruct the image until there is no significant change in PSNR gradually decreased to approximately 3.

At the optimal threshold value, there was no appreciable increase in compression ratio from the case with no pre-processing. However, the reconstruction quality experienced a drop from 31.25 dB to 31.15 dB. The compression ratio at T=0.006, the pure noise level, gave a 7.8% increase in compression ratio. The first results that were studied, however, were the entropy values of the coefficient reduced images.

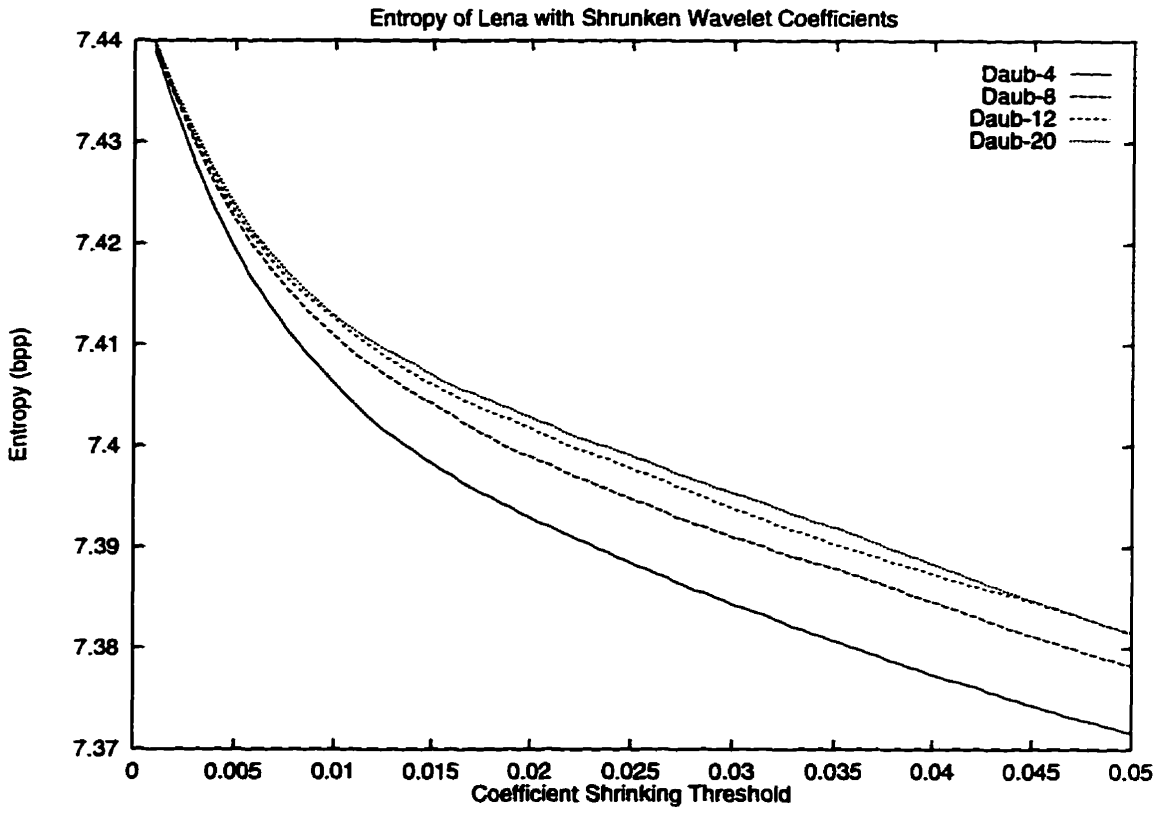


Fig. 6.9. Entropy trend of the image *Lena* after wavelet coefficient shrinking.

The plot shown in Fig. 6.9 shows the entropies of *Lena* after shrinking the wavelet coefficients with all four Daubechies wavelets. The plots for *Goldhill* and *Peppers* are shown in Figs. A.3 and A.4 respectively. With *Lena* and *Goldhill*, the Daubechies-4 wavelet produced the greatest drop in entropy and with *Peppers*, Daubechies-4 and 8 were closely matched as having the greatest effect on entropy. In the plot for *Peppers* , there is a marked change approximately at T=0.006 which is the pure noise threshold. There are more rough features in *Lena* and *Goldhill* than in *Peppers* to make them more sensitive to changes in the Daubechies-4 coefficients. Visually, it was observed that the quality of the images degrades as the threshold increases.

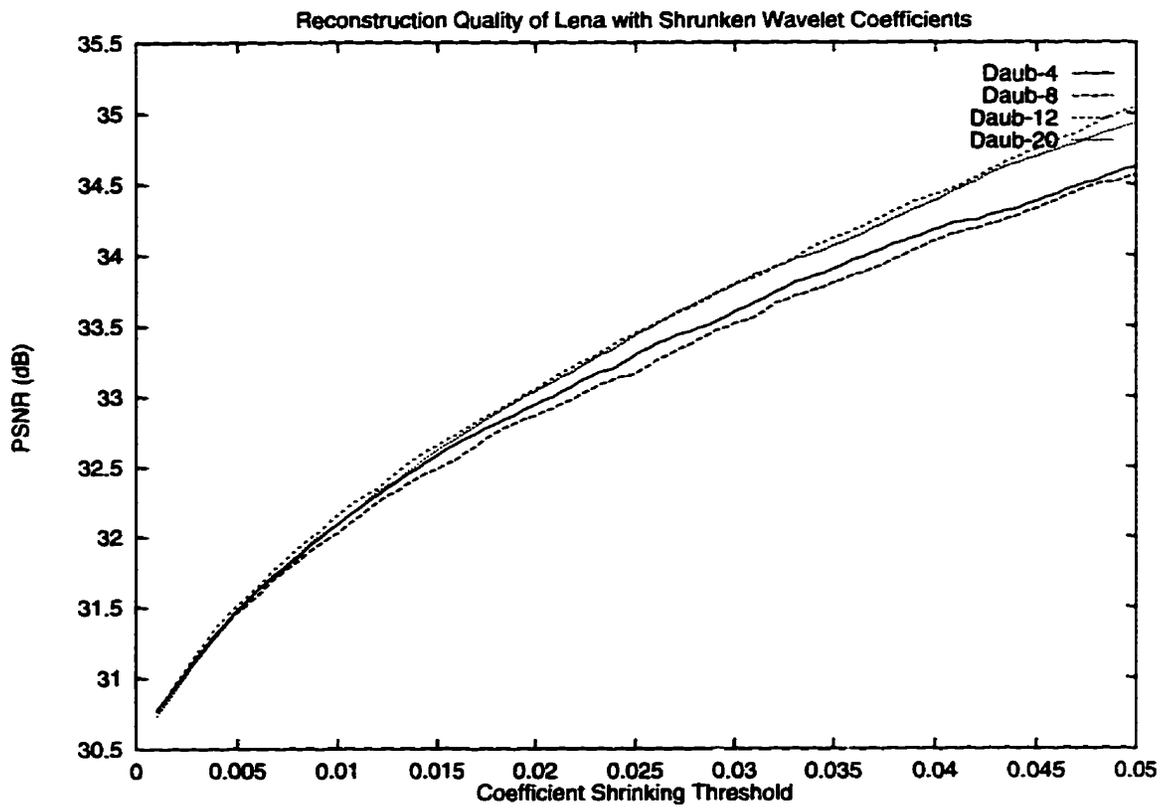


Fig. 6.10. Reconstruction quality trend with respect to the pre-processed image of *Lena* after wavelet coefficient shrinking.

The plot shown in Fig. 6.10 shows the reconstruction quality of the image with respect to the wavelet smoothed images after fractal coding. The increasing PSNR means that as the shrinking threshold increases, the reduced-search FBC algorithm becomes more capable of accurately coding and reconstructing the image. Subjectively, the image becomes noticeably distorted at T=0.020. The coefficients representing major parts of the signal other than fine details and noise are being affected. The plots for *Goldhill* and *Peppers* are shown in Figs. A.5 and A.6. How much the signal is changed with respect to the original image is shown in the PSNR measure of the image with respect to the original image.

The next plot, shown in Fig. 6.11, shows the change in fidelity of the wavelet reduced images with respect to the original after reduced-search FBC. In all three test images, the plot stays fairly level until T is equal to approximately 0.005 which is close to

the calculated value of the pure noise level of 0.006. After T=0.005, more than just the noise was being removed and the plots reflect this with an increased downward slope. It is interesting to note that while the image fidelity is fairly constant before T=0.005, the compression increases throughout the range of threshold values. The plots for *Goldhill* and *Peppers* are shown in Figs. A.7 and A.8 respectively.

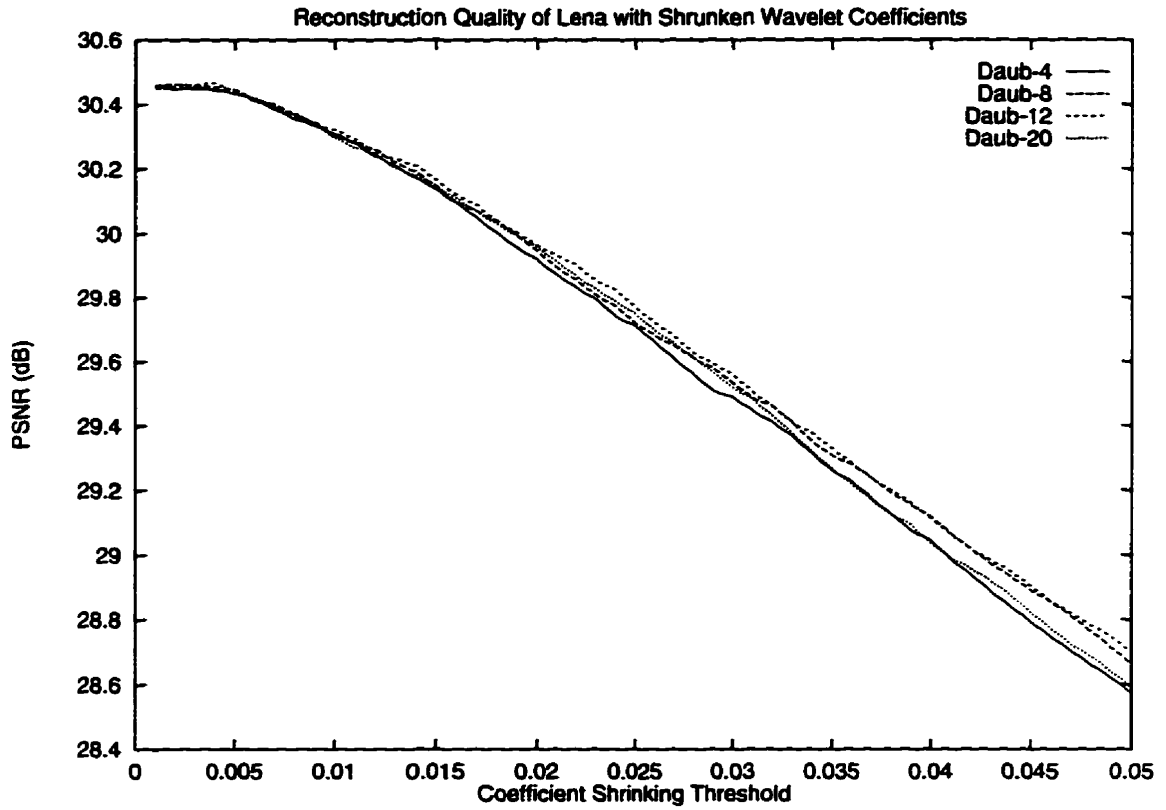


Fig. 6.11. Reconstruction quality trend with respect to the original image of *Lena* after wavelet coefficient shrinking.

After reduced-search fractal block coding, the compression ratios increased with increasing coefficient reduction threshold. However, with the Daubechies-4 wavelet, it can be seen that the compression ratio increased noticeably faster than that of Daubechies-8, 12, and 20 with the *Lena* and *Peppers* (Fig. A.10) images. The image, *Goldhill*, did not have as noticeable an effect with the Daubechies-4 wavelet and all four wavelets followed the same trend as shown in Fig. A.9. The compression ratio of Lena at T=0.02 was 19.1:1 as compared to 17.4:1 before wavelet shrinking which is a 9.8% increase. The image

Goldhill experienced a 1.1% increase and *Peppers* a 1.7% increase at the same threshold level and wavelet.

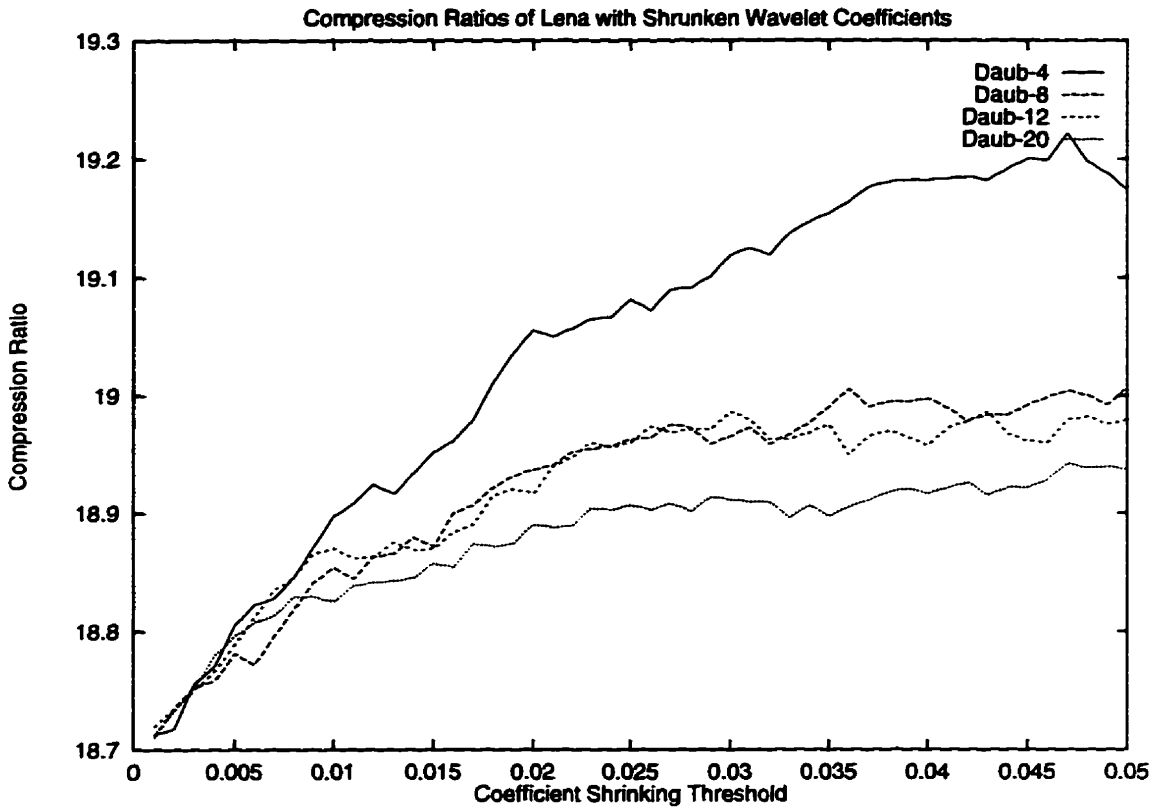


Fig. 6.12. Compression ratios of *Lena* after wavelet coefficient shrinking.

The reconstructed images of *Lena* at T=0.006, T=0.015 and T=0.020 are shown in Figs. 6.13, 6.14 and 6.15 respectively. It was observed that the artifacting due to block coding was still present. Since T=0.006 appeared to be a significant threshold value, the reconstructed images of *Goldhill* and *Peppers* at that threshold are shown in Fig. A.13 and A.14 respectively.

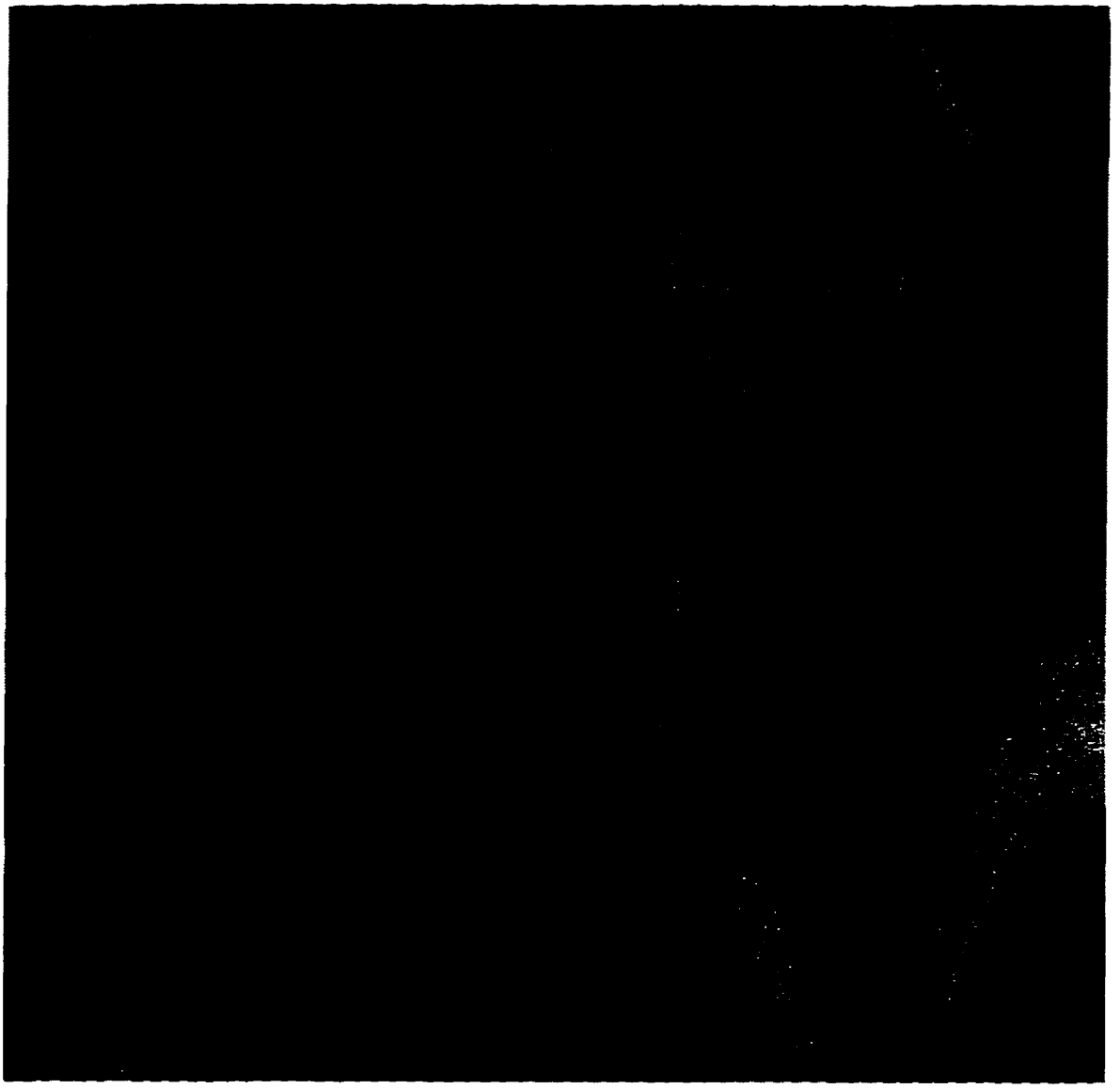


Fig. 6.13. Reconstructed image of *Lena* at T=0.006. PSNR = 30.42 dB; CR = 18.8:1. (Shown at 84.3% original size.)

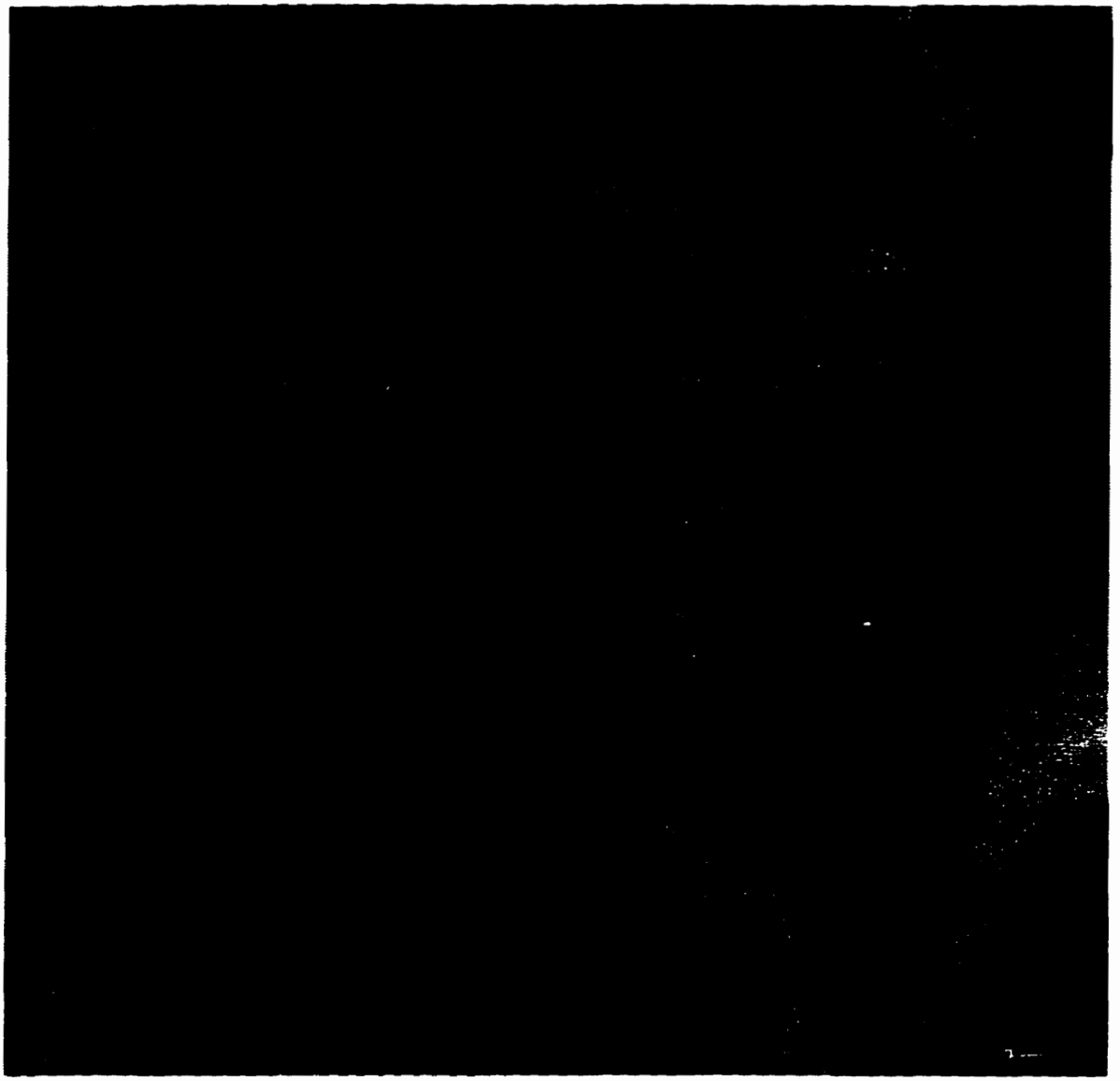


Fig. 6.14. Reconstructed image of *Lena* at T=0.015. PSNR = 30.14 dB; CR = 19.0:1. (Shown at 84.3% original size.)

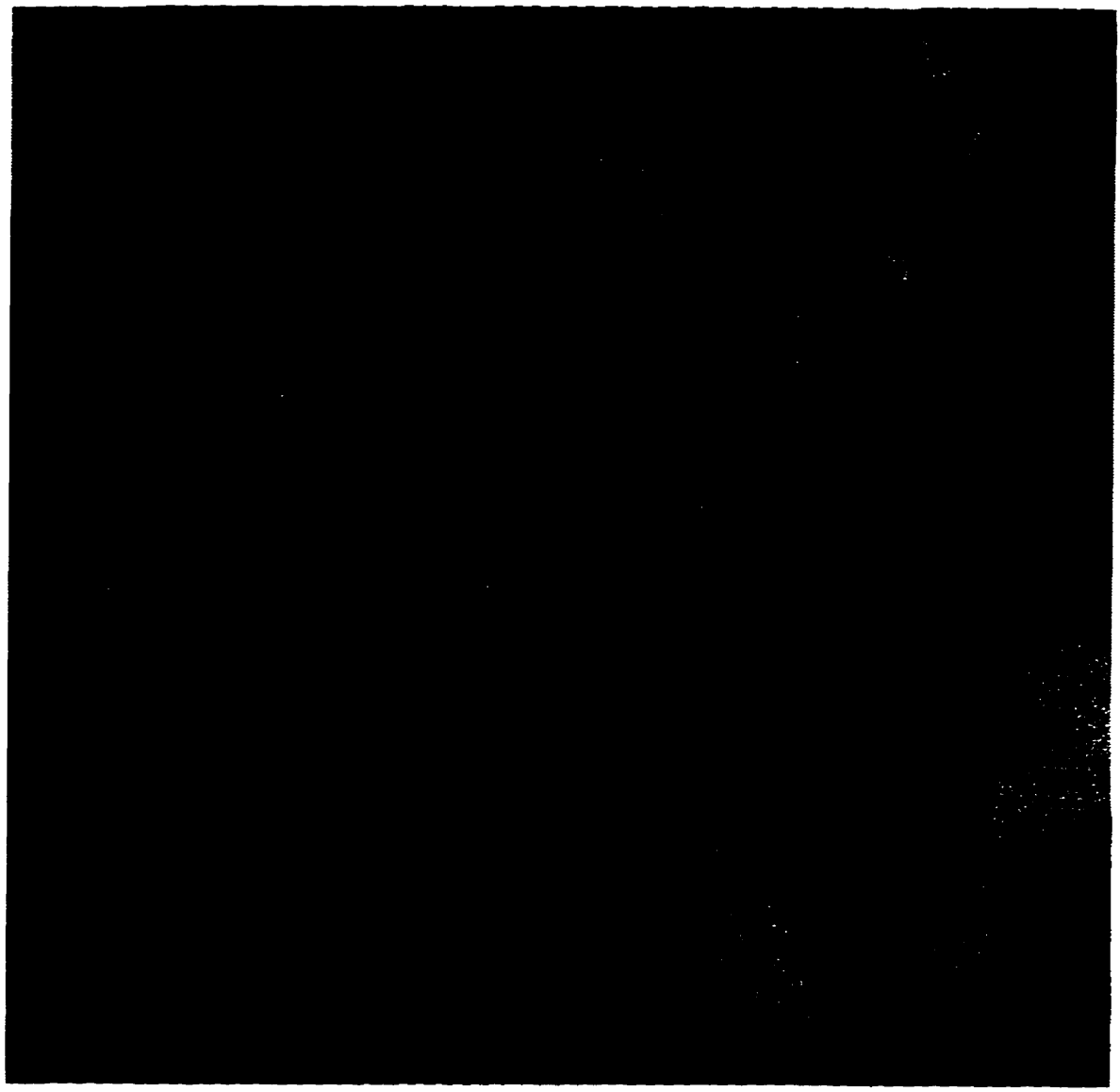
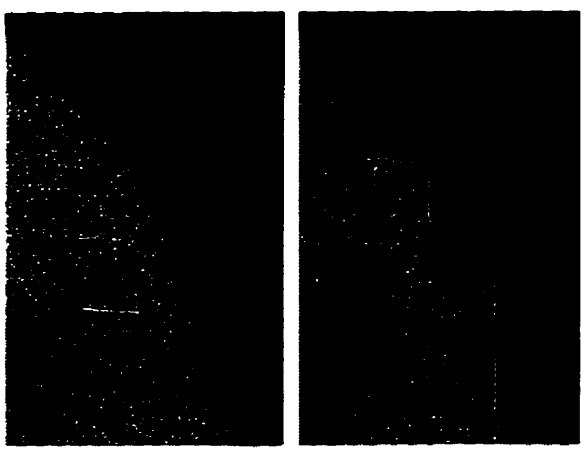


Fig. 6.15. Reconstructed image of *Lena* at T=0.020. PSNR = 29.92 dB; CR =19.1:1. (Shown at 84.3% original size.)

6.4 Scale Invariance of Reconstructed Images

In Chapter 2, fractals were described as being scale invariant where the detail seen is not dependent on the scale at which the observation was made. Fig. 6.16a shows a portion of *Lena* 's shoulder enlarged by a factor of four while Fig. 6.16b shows the same portion reconstructed with the iterative procedure at four times the original size.



(a) (b)

Fig. 6.16. A portion of *Lena* 's shoulder enlarged four times in (a) from the original image and (b) through the reconstruction procedure.

It can be seen that in the enlargement of the original image, pixellation occurs but in the enlargement through the iterative procedure, the structure of the shoulder retains its smooth appearance. The artifacting due to block coding is still visible, however, but this can be removed with post processing.

6.5 Summary

In this chapter, the results of experiments of two forms of noise reduction were presented. It was observed that with spatial filtering, degradation of the image was unavoidable. This was expected because the nature of the filter implied that the value of every pixel would be affected. Only the 3x3 filter mask yielded a result of useable quality with a 1.3% increase in compression ratio over compression without any pre-processing.

With wavelet denoising, a useable range of threshold values out of the range [0.000, 0.050] was found to decrease the entropy enough to increase the compression ratio by as much as 9.1% at T=0.020 without introducing a noticeable amount of distortion. At higher threshold values, distortion was observed due to the fact that wavelet coefficients representing critical information were being reduced. The fidelity of the wavelet coefficient reduced images relative to the original showed a distinct change in trend at a threshold equal to the pure noise level. This indicated a point at which the pure noise was being removed and the desired signal left untouched. The compression ratio for *Lena* at this threshold was 18.8:1 or 7.8% higher than the compression ratio of the image without pre-processing.

In both forms of noise reduction, the number of iterations required to reconstruct the image to the point where there is no more significant change in PSNR dropped dramatically to as low as 3 from 9 without pre-processing. This implies that reducing the entropy of the image moves it closer to the attractor in analogy to iterated function systems, further supporting the idea of treating the image as a fractal.

Additional support for the fractal nature of the reconstructed image was observed in the scale invariance of the fractal coded image. Smooth features remained smooth as the scale of observation decreased, that is, as the enlargement factor increased. In comparison, an enlargement of the original image, yields pixellation.

CHAPTER 7

CONCLUSIONS AND RECOMMENDATIONS

This thesis focused on the reduced-search fractal block coding algorithm as developed by Kinsner and Wall with the addition of a pre-processing wavelet denoising stage. The denoising was implemented to reduce entropy increasing additive noise contained in an image.

The initial set of experiments were performed to establish the characteristics of the reduced-search FBC process without any pre-processing. A series of experiments were performed that varied the size of the domain block and the domain block sampling increment. It was found, as expected, that smaller domain blocks with small sampling increments produced higher quality reconstructions than larger domain blocks and sampling increments. The reason for this is that there is a larger feature base available with smaller domain blocks. However, the use of smaller domain blocks implies that there are more to be tested so that the processing time is large and the compression ratio is relatively low because more bits are needed to address the higher number of domain blocks.

A higher compression ratio can be achieved with larger domain blocks and larger sampling increments but at the cost of lower reconstruction quality. Compression ratios ranged from 13.8:1 for 16x16 pixel domain blocks with one pixel sampling increment to 19.6:1 for 32x32 pixel domain blocks with 32 pixel increment. Reconstruction qualities followed a reversed trend where for 16x16 pixel domain blocks with one pixel sampling increment, the reconstruction quality was as high as PSNR = 34.3 dB. For the largest domain block and sampling increment, the quality dropped to as low as PSNR = 28.3 dB. A domain block size of 24x24 pixels and 12 pixel sampling increment was used for the rest of the experiments as those domain block parameters offered an acceptable balance between the two criteria.

The second set of experiments performed involved the pre-processing stage to the reduced-search FBC process. As a comparison to wavelet denoising, spatial filtering was used. By the nature of the filtering process, the desired signal is affected along with the additive noise. While this process does lower the entropy of the signal, thus making it more susceptible to fractal block coding, enough of the signal is distorted to render all but the smallest filter mask impractical. Filter masks larger than 3x3 resulted in noticeable distortions.

Wavelet denoising, on the other hand, allows for the reduction of additive noise without damaging the signal. The entropy was lowered enough to result in as much as a 9.1% increase in compression ratio. With the onset of some distortion as a result of signal degradation, the compression ratio increased to 11% over the same method without pre-processing. The four wavelets used for the denoising followed similar trends in reconstruction quality but the Daubechies-4 wavelet produced results that differed slightly from the rest.

The Daubechies-4 wavelet was observed to have a more accelerated increase in compression ratio as the coefficient reduction threshold increased. There are two possibilities that explain this occurrence. One possibility is that noise, being a rough signal, is modeled well by the Daubechies-4 wavelet. By reducing coefficients that are modeling the noise, the noise is effectively reduced. The other possibility is that as the threshold increased to the point where image degradation occurred, coefficients were being reduced that represent features in the desired signal because the signal itself has features that are modeled well by the Daubechies-4 wavelet. However, in the image *Peppers*, which contains many smooth features, the same effect was not observed with the wavelets that model a smooth signal well. This suggests that the first possibility is more probable.

The reconstruction quality of the images compressed with wavelet coefficient shrinking did not differ significantly from the same process without pre-processing. Quantitatively, the PSNR values relative to the original decreased as the threshold increased

because the denoised images were, numerically, different from the originals. The PSNR values relative to the denoised images before they entered the FBC process increased with increasing threshold indicating that they were more easily fractal coded and thus more easily reconstructed. The reduced entropy gave the image more fractal characteristics.

Reducing the entropy of an image with wavelet denoising to give the image more fractal characteristics has been shown to be a viable method of increasing the efficiency of reduced-search fractal block coding. At this point, it is a possibility that rough wavelets will perform better than smooth wavelets. Further study should be done on other rough wavelets of other families of wavelets such as Coifman [Wick94]. Selecting which wavelet would be suitable for a particular image depends on qualities of the image itself. Classifying the image would lead to a practical approach to selecting the appropriate wavelet. In this work, the thresholding was applied to all sub-bands. Further experiments should be performed on individual sub-bands. If there is a sub-band which represents the noise part of the signal more that any other sub-band, reducing those coefficients would result in an optimally denoised image.

In this thesis, entropy provided one estimate of the "fractalness" of the image. Another method of determining the "fractalness" of the image hinges on the discussion of fractal dimension in Chapter 2. Multifractal measures [Kins94a], [ArFG91], [FeKi94] offer ways of determining to what degree the image is a fractal. This measure could lead to a better selection of a wavelet for wavelet denoising as well as affecting the selection of other parameters of the reduced-search FBC process itself.

Presently, the reduced-search FBC process uses range blocks that are 8x8 pixels in size and domain block sizes that can vary. It has been observed [ArFG91], [FeKi94] that an image is not of a uniform fractal dimension, that is, the fractality of the image various throughout the image. The fractal dimension of a region or pixel of the image should be used to determine a range block that is suitable for that particular region. A region with a

higher fractal dimension would be more detailed thus requiring a higher resolution of range blocks. Smaller range blocks yield higher resolution.

The above mentioned methods still do not address the use of iterated function systems which has already been shown to yield high compression ratios. Some work has been done that automatically determines iterated function systems for one-dimensional signals [MaHa92]. This could possible by extended to two dimensional signals by operating on one line at a time.

This thesis has shown that reducing the entropy of an image through wavelet denoising increases the efficiency of reduced-search fractal block coding. Supported by the theories of collage coding and contractive affine transformation, reduced-search fractal block coding is a comparable alternative to present image compression standards. The theories also indicate that fractal compression has the potential of achieving much higher compression ratios. With the increasing need for rapid communication, higher compression ratios are definitely needed and fractal image compression is clearly a path in that direction.

REFERENCES

[ABMD92] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using wavelet transform," *IEEE Trans. Image Proc.*, vol. 1, no. 2, pp. 205-220, April 1992.

[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.

[ArFG91] F. Arduini, S. Fioravanti, D. D. Giusto, "A multifractal-based approach to natural scene analysis," *Intl. Conf. Acoustics, Speech, and Signal Proc.* (Toronto, ON; May 14-17, 1991), pp. 2681-2684.

[Bal95] S. Bal, "A systematic approach to solving the inverse problem of iterated function systems," *Report.* Winnipeg, MB: Dept. Electrical and Computer Eng., Univ. of Manitoba, April 1995, 17 pp.

[Barn88] M. Barnsley, *Fractals Everywhere.* Boston, Mass: Academic Press Professional, 1988, 396 pp.

[Barn93] M. Barnsley, *Fractals Everywhere 2nd Ed.* Boston, Mass: Academic Press Professional, 1988, 531 pp.

[BCDJ95] J. Buckheit, S. Chen, D. Donoho, I. Johnstone, J. Scargle, "Wavelab reference manual," Matlab Software, Stanford University & NASA-Ames Research Center, 239 pp., 1995. (available through the Internet at ftp://playfair.stanford.edu)

[Dono92] D. L. Donoho, "De-noising via soft-thresholding," *Technical Report*, Dept. of Statistics, Stanford University, 37p., 1992. (available through the Internet at ftp://playfair.stanford.edu/pub/reports).

[FeKi94] K. Ferens and W. Kinsner, "A multifractal entropy measure for feature extraction of images," *Can. Conf. Elec. & Comp. Eng.*, (Halifax, NS; Sept. 25-28, 1994).

[FeLK93] K. Ferens, W. Lehn, W. Kinsner, "Image compression using learned vector quantization," *Proc. IEEE Communications, Computers & Power Conf., WESCANEX'93* (Saskatoon, SK; May 17-18, 1993), pp. 299-312.

[Fish94] Y. Fisher, Ed., *Fractal Image Compression: Theory and Application*: New York, NY: Springer-Verlag, 1994, 341 pp.

[GoWo92] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Reading, Mass: Addison-Wesley, 1992, 716 pp.

[HwMa93] W.-L. Hwang and S. Mallat, "Characterization of self-similar multifractals with wavelet maxima," *Technical Report 641*, New York University, July 1993, 25 pp.

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

[Jacq93] A. E. Jacquin, "Fractal image coding: a review," *Proc. of the IEEE*, vol.81, no. 10, pp. 1451-1465, October 1993.

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

[Kins94a] W. Kinsner, "Fractal dimension: Morphological, entropy, spectrum, and variance classes," *Technical Report*, DEL94-4, Winnipeg, MB: Dept. Electrical and Computer Eng., Univ. of Manitoba, May 31, 1994, 146 pp.

[Kins94b] W. Kinsner, *Fractals and Chaos Engineering. Course Notes*. Winnipeg, MB: Dept. Electrical and Computer Eng., Univ. of Manitoba, January 1994, 380 pp.

[Krey78] E. Kreyszig, *Introductory Functional Analysis with Applications*. New York, NY: John Wiley & Sons, 1978, 688 pp.

[Lang96] A. Langi, "Wavelet and fractal processing and compression of nonstationary signals," *Ph.D. Thesis*, University of Manitoba, 1996.

[LaKi96] A. Langi and W. Kinsner, "Compression of aerial ortho images based on images denoising,"*Proc. NASA Data Compression Workshop*, (Snowbird, UT; April 4, 1996).

[MaHa92] D. S. Mazel and M. H. Hayes, "Using iterated function systems to model discrete sequences," *IEEE Trans. Signal Proc.*, vol. 40, no. 7, pp. 1724-1734, July 1992.

[MaHw92] S. Mallat and W.L. Hwang, "Singularity detection and processing with wavelets," *IEEE Trans. Info. Theory*, vol. 38, no. 2, pp. 617-643, March 1992.

[Mall89] S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. 11, no. 7, pp. 674-693, July 1989.

[Mand77] B. Mandelbrot, *The Fractal Geometry of Nature*. New York, NY: W. H. Freeman and Company, 1977, 468 pp.

[PeJS92] H.-O. Peitgen, H. Jürgens, D. Saupe, *Chaos and Fractals New Frontiers of Science*: New York, NY: Springer-Verlag, 1992, 984 pp.

[Teka95] A. M. Tekalp, *Digital Video Processing*. Upper Saddle River, NJ: Prentice-Hall, 1995, 526 pp.

[Wick94] M. V. Wickerhauser, *Adapted Wavelet Analysis from Theory to Software*. Wellesley, Mass: A K Peters, 1994, 486 pp.

[WaKi93] L. Wall and W. Kinsner, "A fractal block coding technique employing frequency sensitive competitive learning," *Proc. IEEE Communications, Computers & Power Conf., WESCANEX'93* (Saskatoon, SK; May 17-18, 1993), IEEE Cat. No. 93CH3317-5; pp. 320-329.

[Wall93] L. Wall, "Reduced search fractal block coding using frequency sensitive neural networks," *M.Sc. Thesis*, University of Manitoba, 1993.

[Youn96] C. Young, POV-Team Co-ordinator, POV-Ray 3.0 (Persistence of Vision Ray Tracer). Indianapolis, IN, 1996.

APPENDIX A

ADDITIONAL IMAGES AND PLOTS

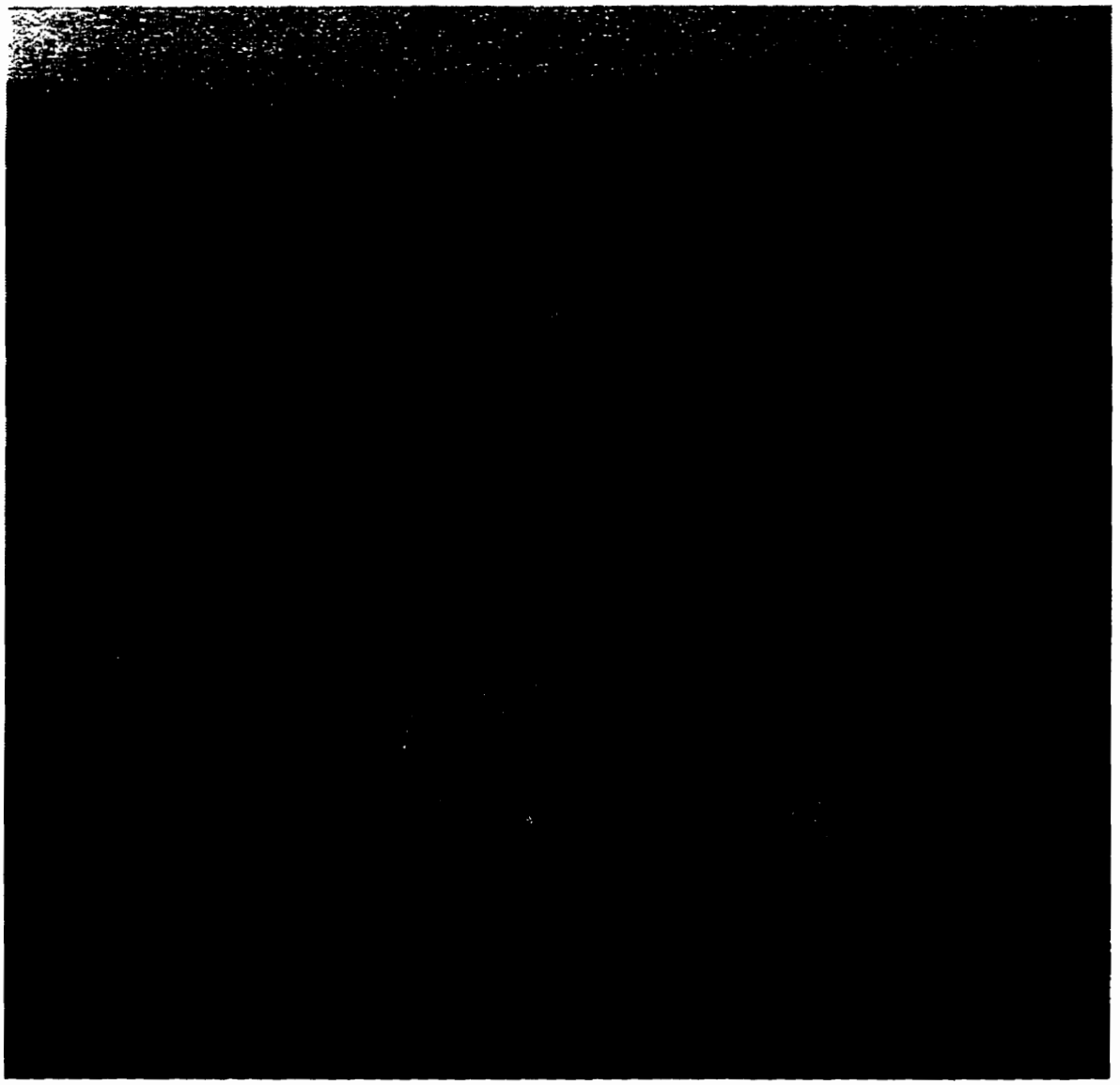


Fig. A.1. Reconstructed *Goldhill* with PSNR = 29.46 dB and CR = 17.6:1. No pre-processing was used. (Shown at 84.3% original size.)

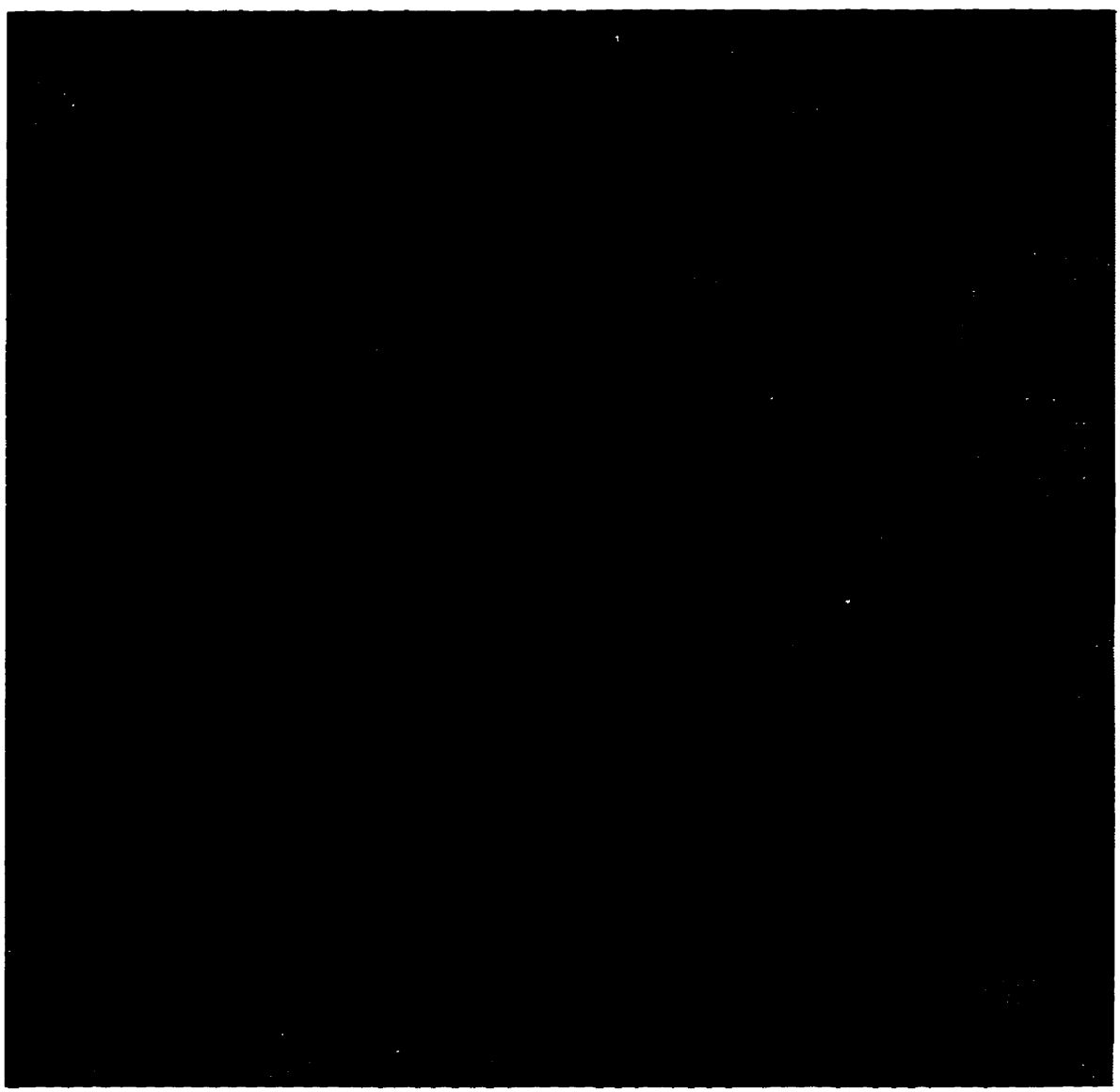


Fig. A.2. Reconstructed *Peppers* with PSNR = 32.79 dB and CR = 17.5:1. No pre-processing was used. (Shown at 84.3% original size.)

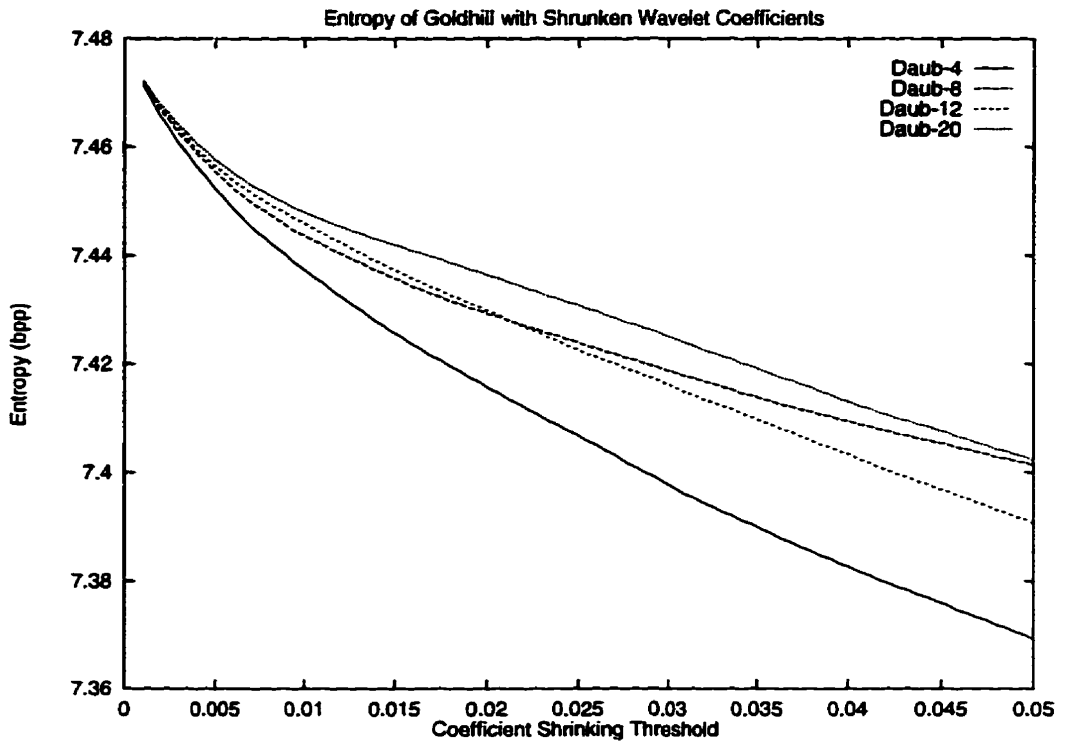


Fig. A.3. Entropy trend of the image *Goldhill* after wavelet coefficient shrinking.

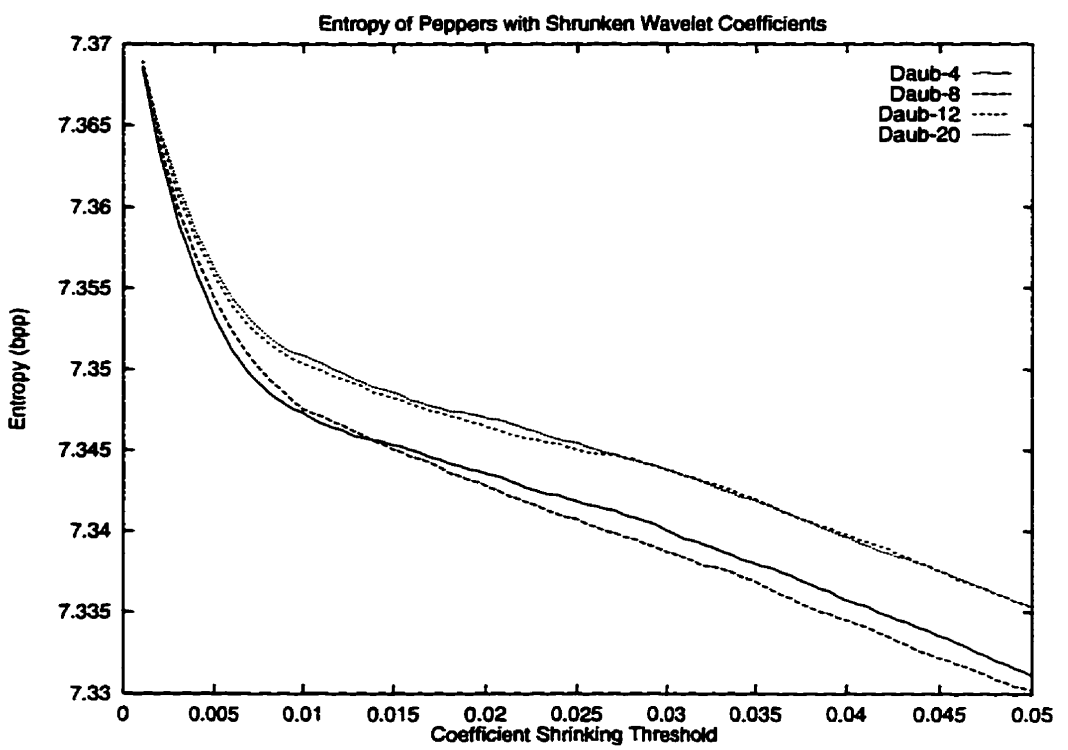


Fig. A.4. Entropy trend of the image *Peppers* after wavelet coefficient shrinking.

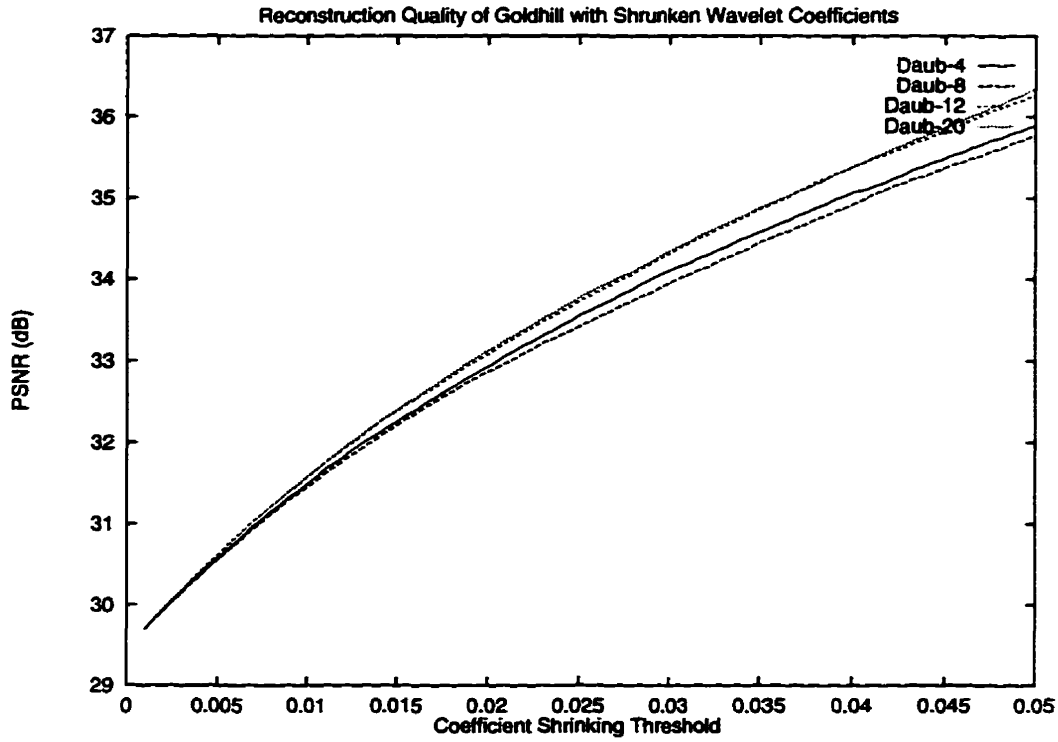


Fig. A.5. Reconstruction quality trend with respect to the wavelet coefficient reduced image of *Goldhill.*

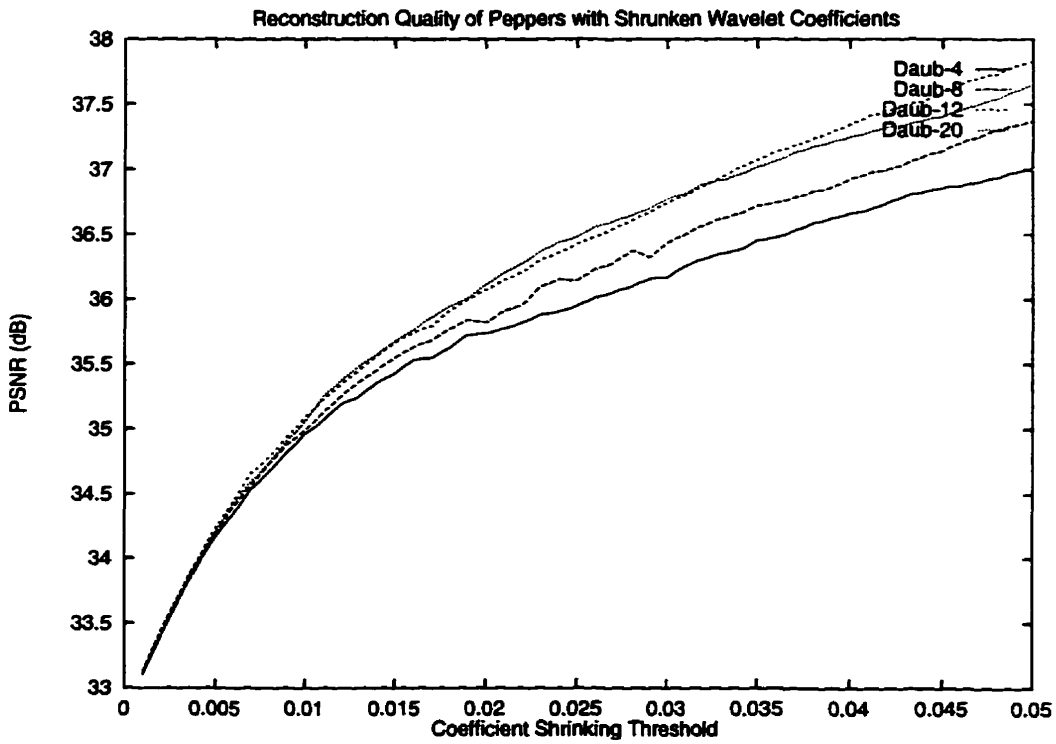


Fig. A.6. Reconstruction quality trend with respect to the to the wavelet coefficient reduced image of *Peppers.*

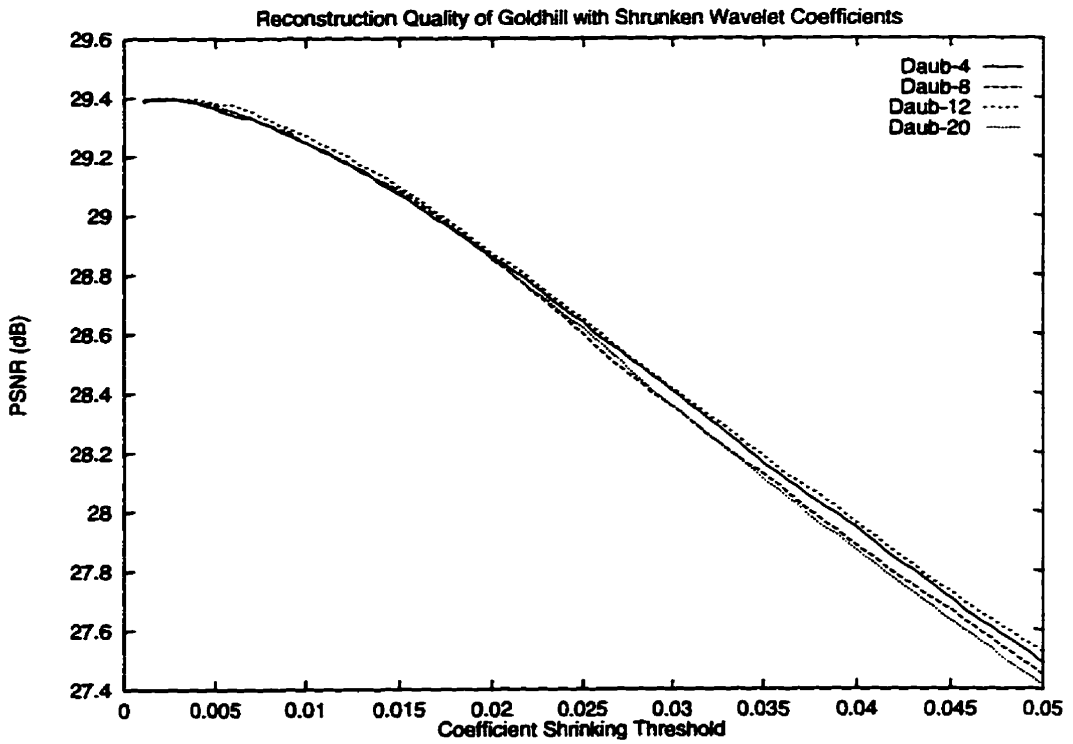


Fig. A.7. Reconstruction quality trend with respect to the original image of *Goldhill* after wavelet coefficient shrinking.

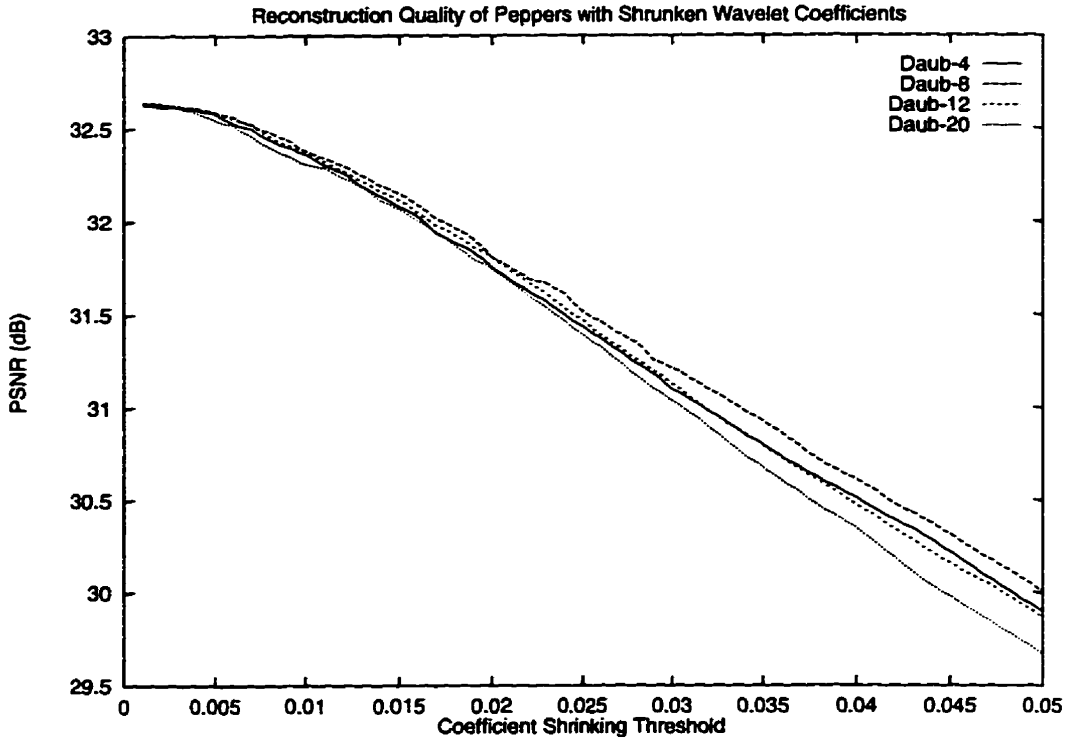


Fig. A.8. Reconstruction quality trend with respect to the original image of *Peppers* after wavelet coefficient shrinking.

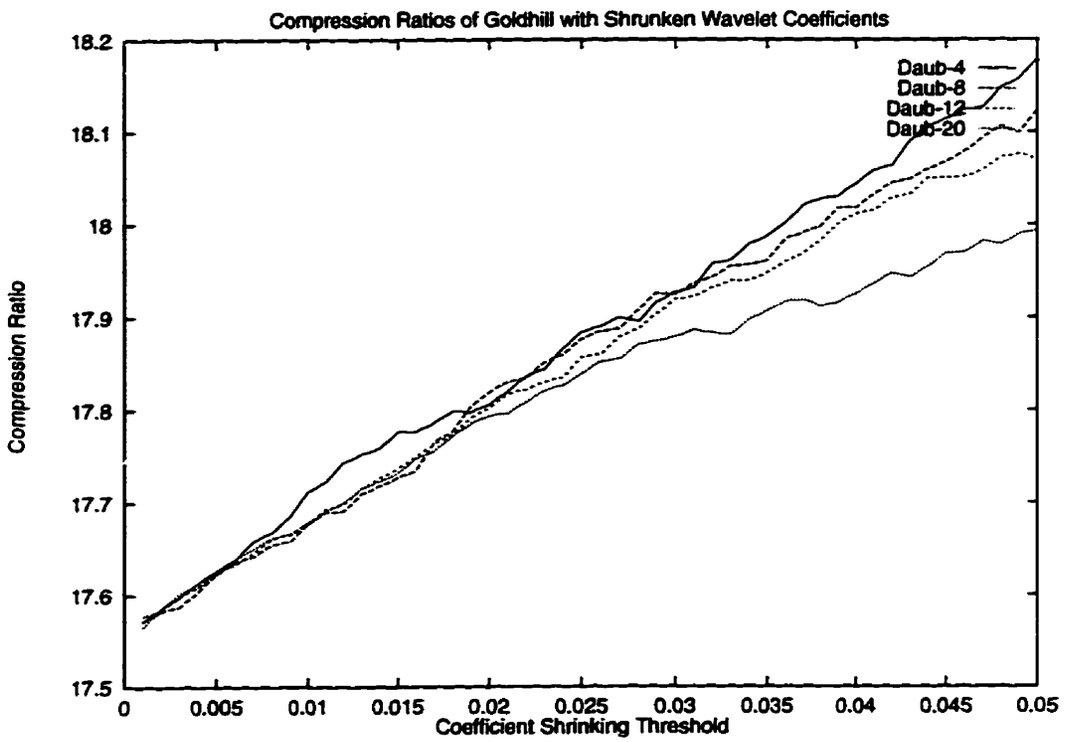


Fig. A.9. Compression ratios of *Goldhill* after wavelet coefficient shrinking.

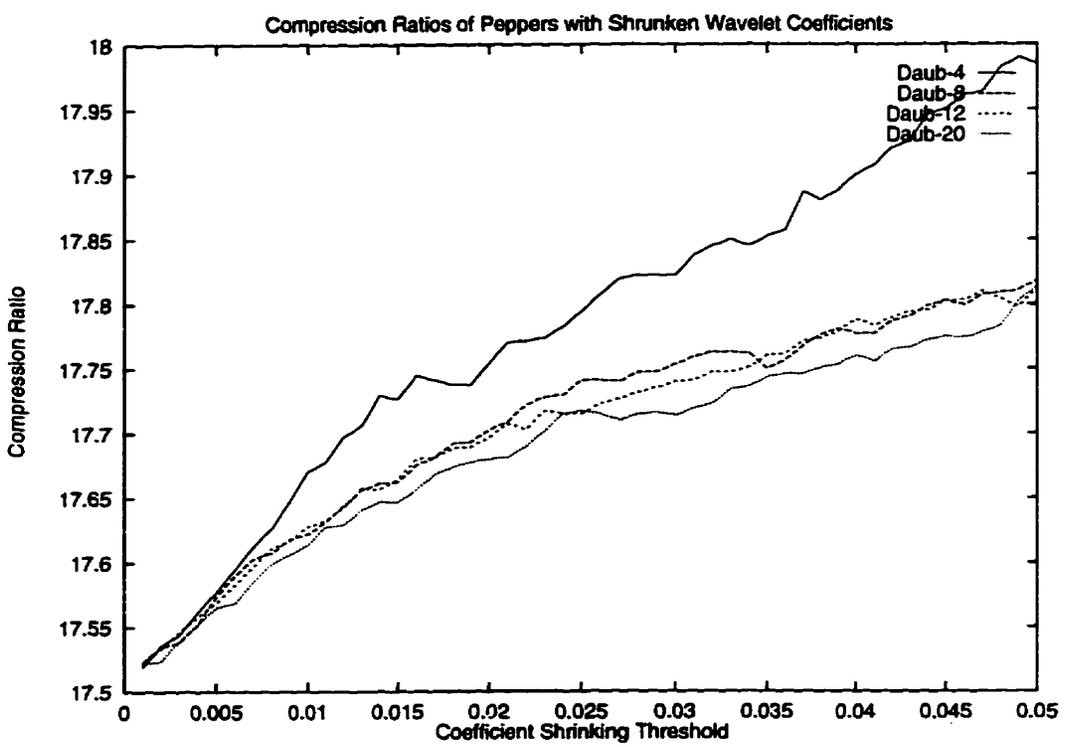


Fig. A.10. Compression ratios of *Peppers* after wavelet coefficient shrinking.

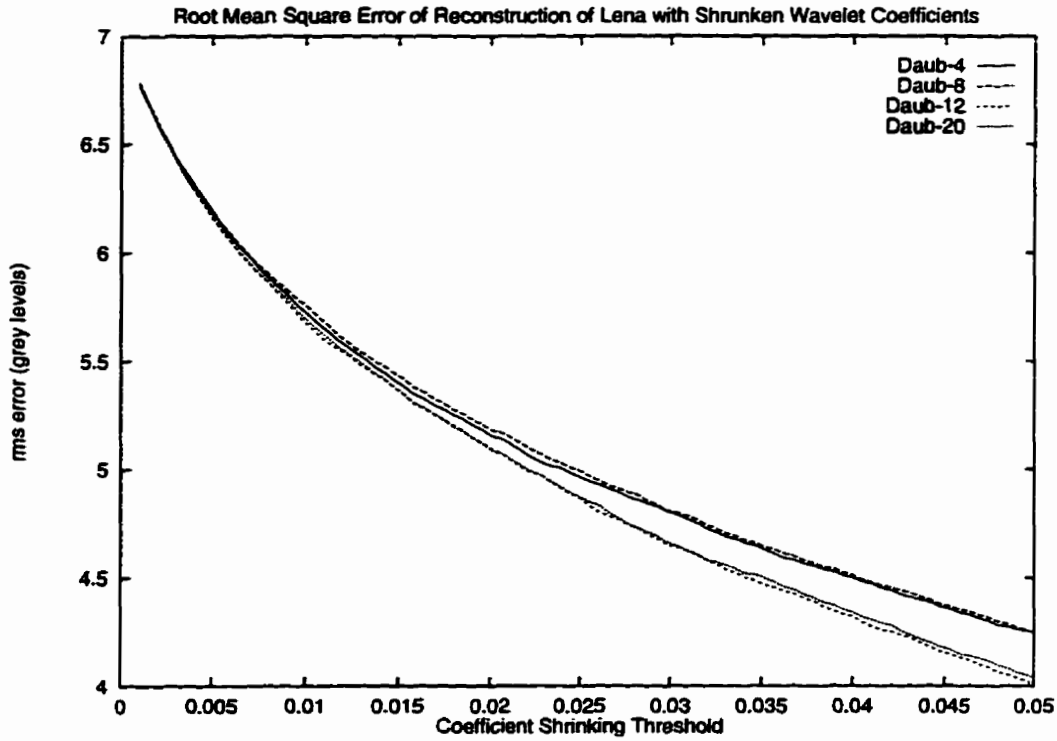


Fig. A.11. Root mean square error of the reconstruction with respect to the wavelet coefficient reduced image of *Lena.*

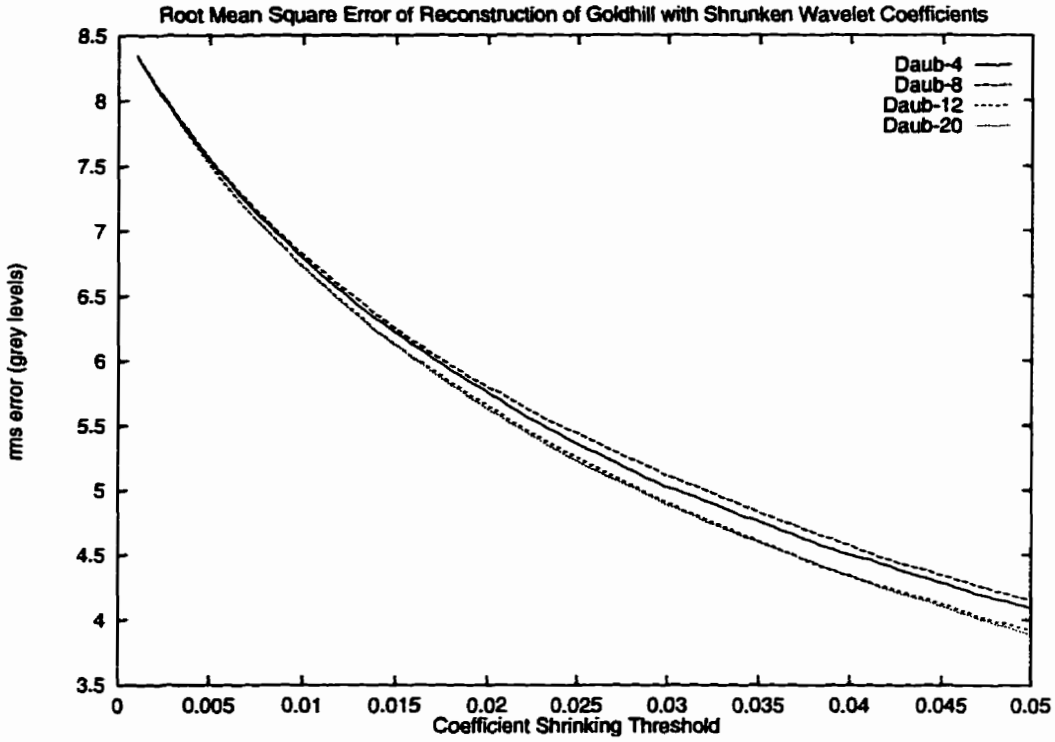


Fig. A.12. Root mean square error of the reconstruction with respect to the wavelet coefficient reduced image of *Goldhill.*

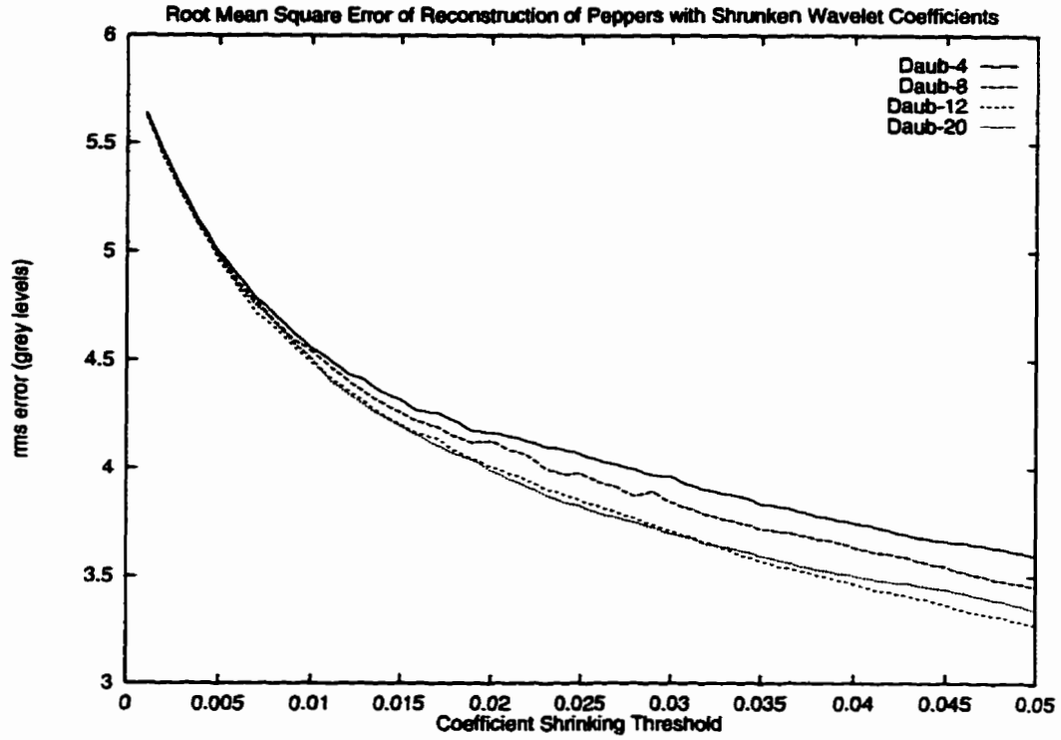


Fig. A.13. Root mean square error of the reconstruction with respect to the wavelet coefficient reduced image of *Peppers*.

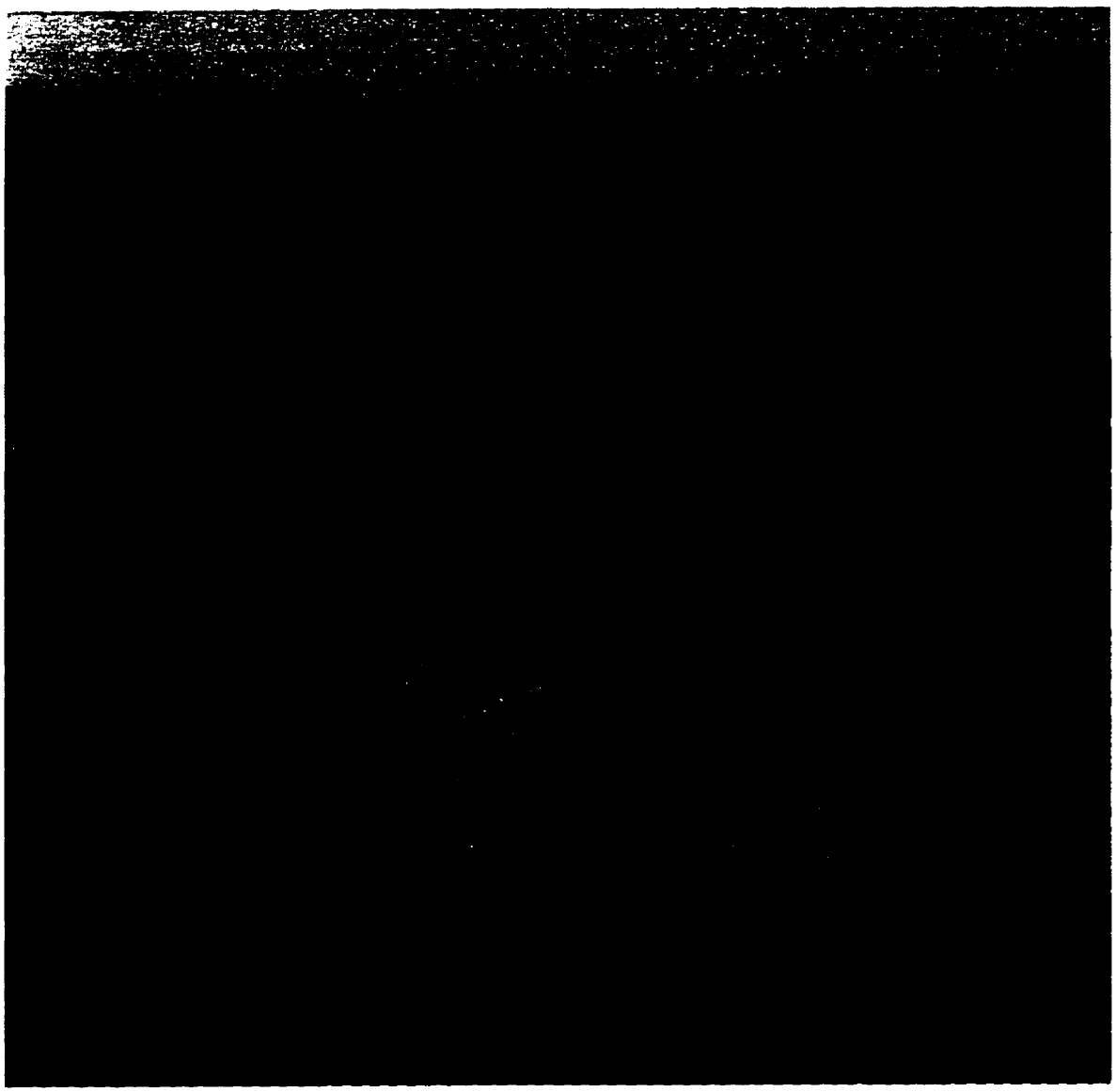


Fig. A.14. Reconstructed image of *Goldhill* at T=0.006. PSNR = 29.34 dB; CR = 17.6:1. (Shown at 84.3% original size.)

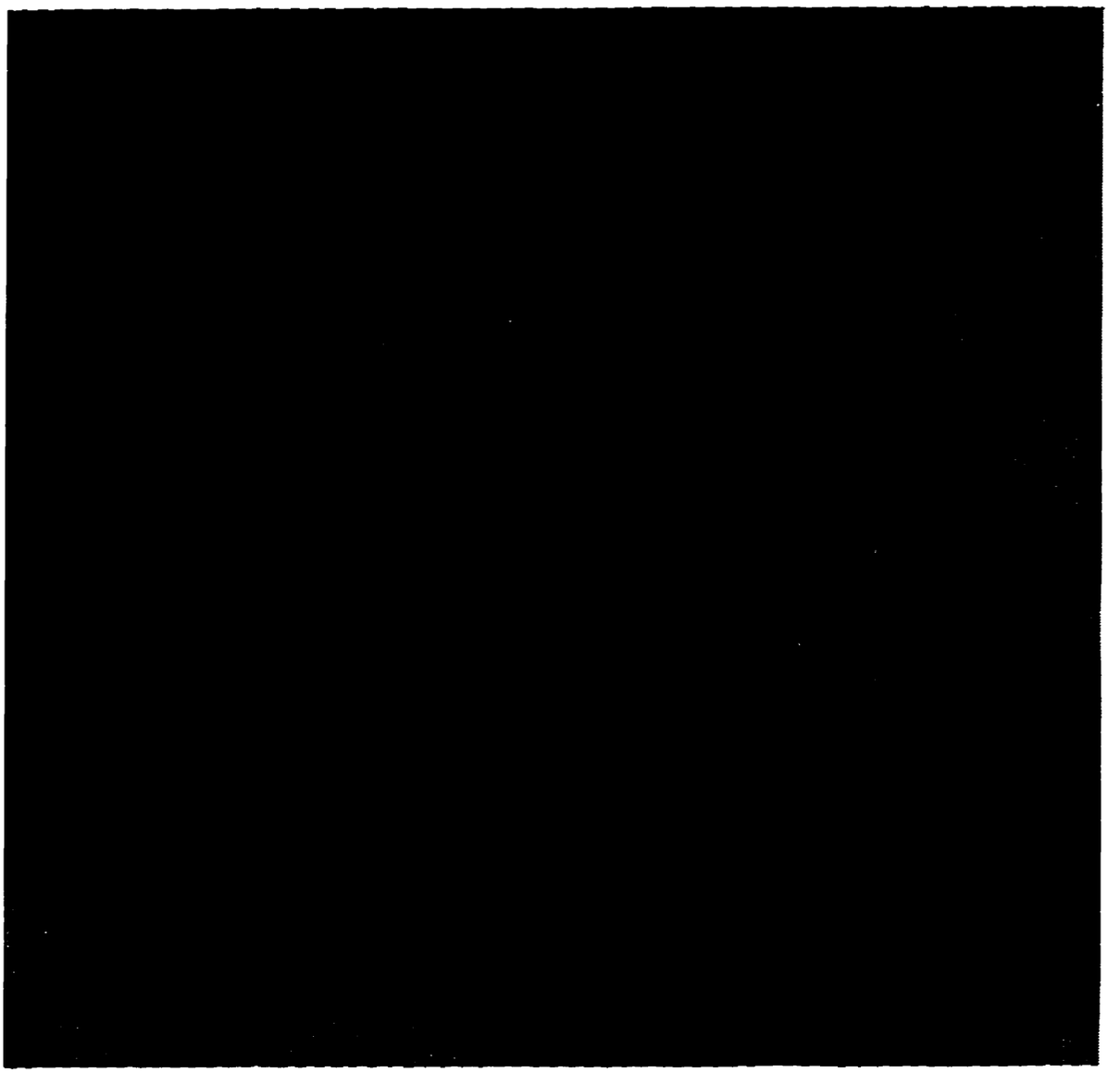


Fig. A.15. Reconstructed image of *Peppers* at T=0.006. PSNR = 32.53 dB; CR = 17.6:1. (Shown at 84.3% original size.)

APPENDIX B

SOFTWARE STRUCTURE

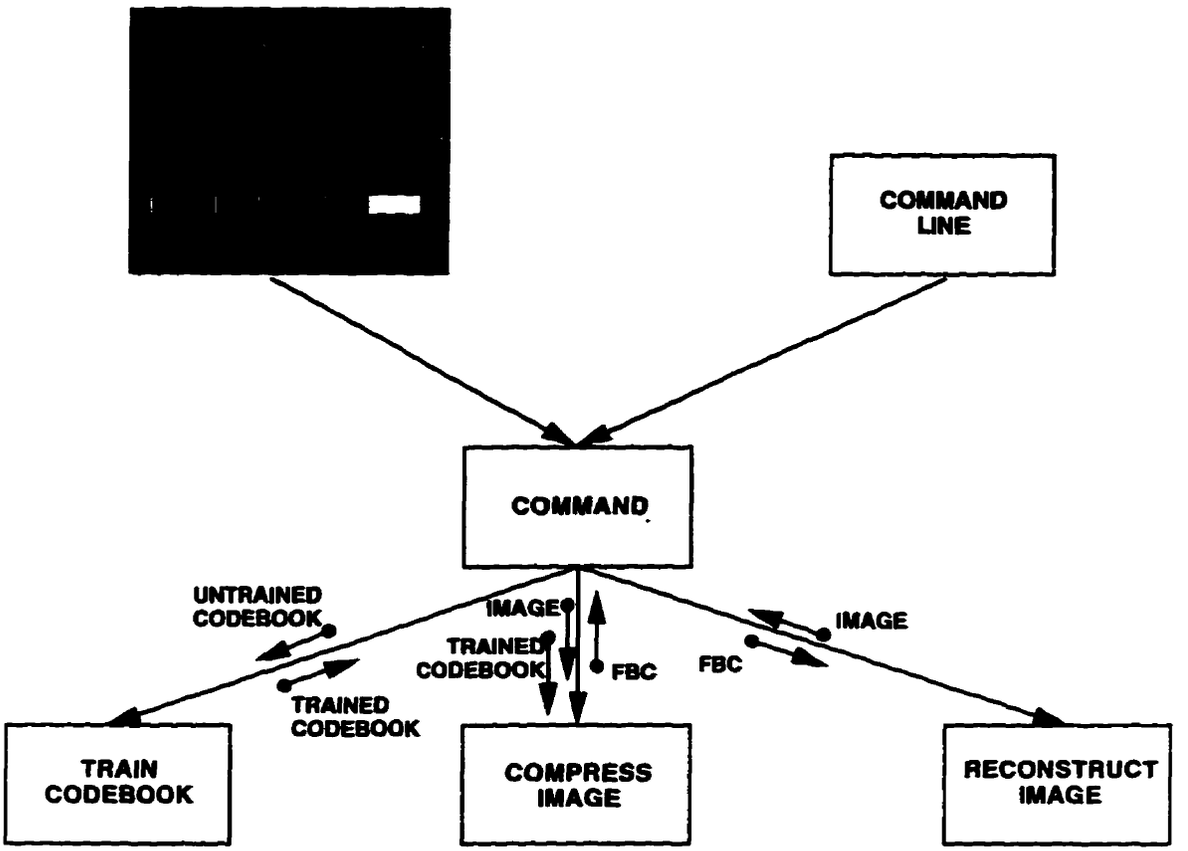


Fig. B.1. Main function hierarchy.

The main functional hierarchy of reduced-search fractal block coding (FBC) is shown in Fig. B.1. The COMMAND module received input from the graphical user interface or the command line. Based on that input, the COMMAND module calls the appropriate module. The main functions are Train Codebook, Compress Image, and Reconstruct Image.

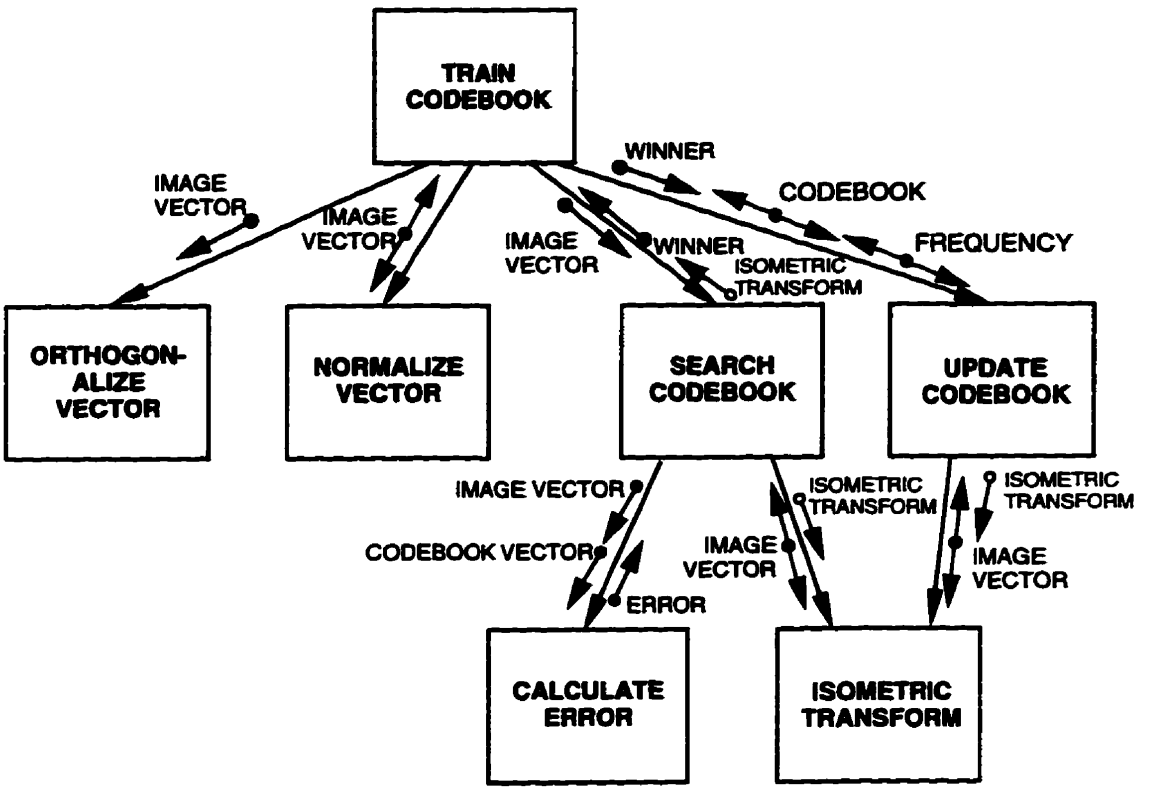


Fig. B.2. Train codebook function hierarchy.

Train Codebook

Input: Untrained codebook, training image

Output: Trained codebook

This module initially sets each codeword to be the mean of very possible input vector from the training image plus or minus some random value. An image vector is then sampled from the training image and orthogonalized and normalized. The codebook is searched for the winning codeword whose index is returned along with the best isometric transform and calculated error. Using the error data, the codeword and its frequency counter is updated.

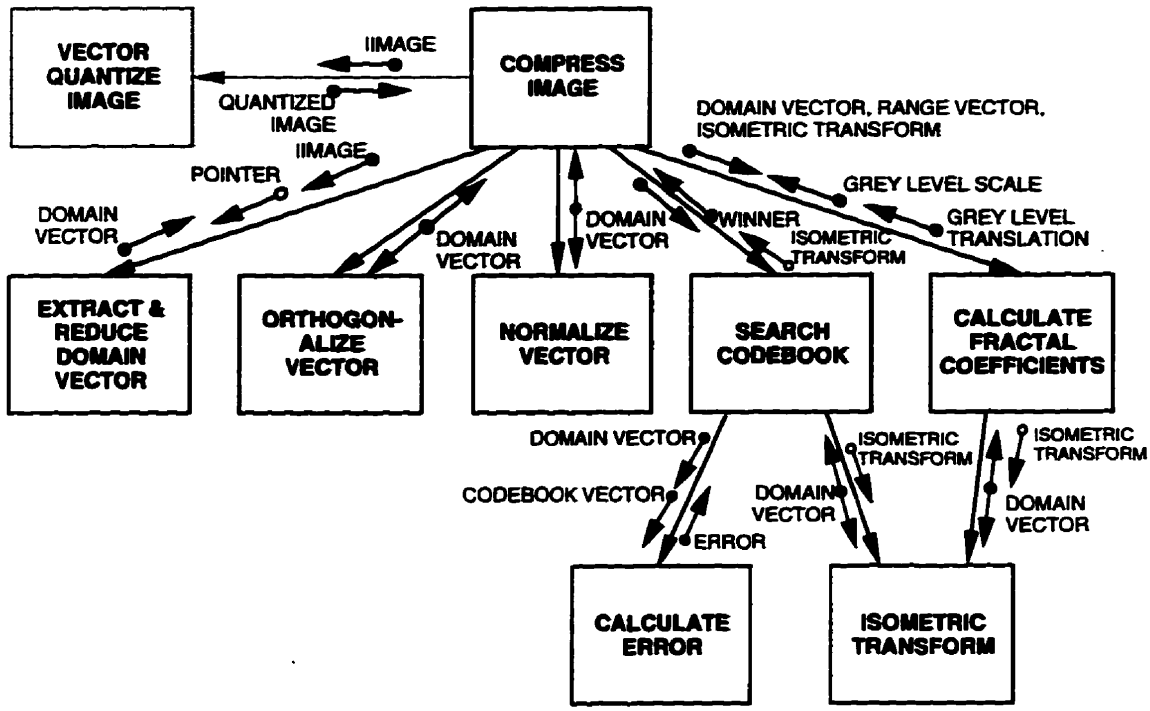


Fig. B.3. Compress image function hierarchy.

Compress Image

Input: Source image, trained codebook

Output: Fractal block code

In this module, the image is first vector quantized to classify each range block. The domain blocks or vectors are sequentially extracted and reduced to the size of the range block. The vector is then orthogonalized and normalized so that it can be compared with the vectors in the codebook. The Search Codebook module is used to classify the domain vector. The domain block whose classification matches that of the range block has its fractal coefficients determined. The domain block that matches the range block the closest has its fractal coefficients preserved as part of the fractal code.

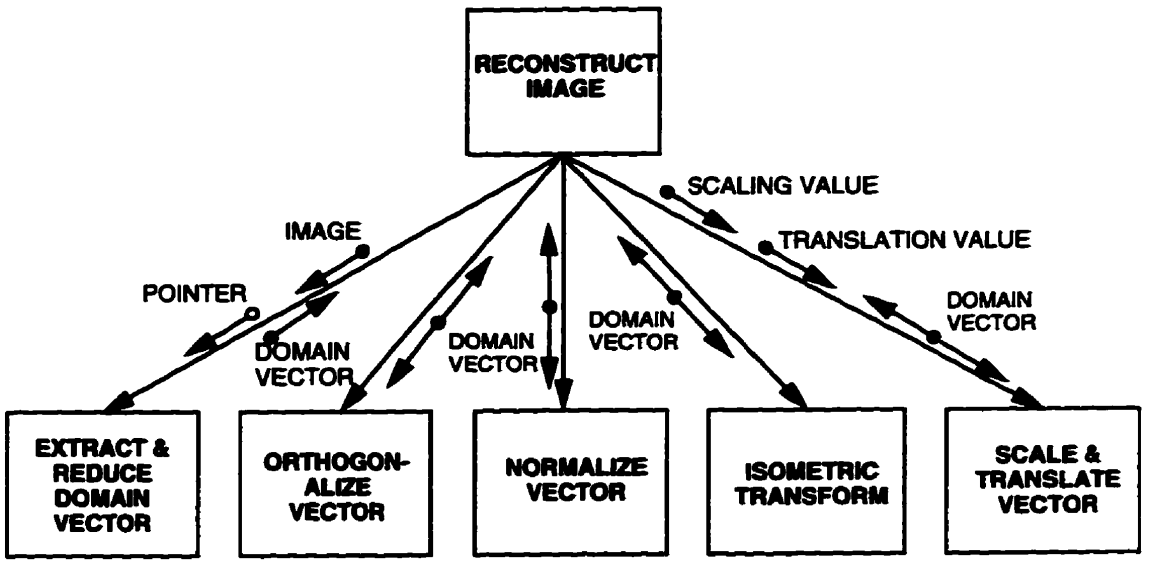


Fig. B.4. Reconstruct image function hierarchy.

Reconstruct Image

Input: Fractal block code

Output: Reconstructed image

A domain vector from an arbitrary image support is extracted, reduced, orthogonalized and normalized. Using the given isometric transform index from the fractal code, the domain block is appropriately transformed. The block is then grey level scaled and translated to produce a range block that can be mapped on to the corresponding position to the fractal code on the image support. The process is repeated for all range blocks.

APPENDIX C

SOURCE CODE FOR DENOISED REDUCED-SEARCH FBC

This appendix contains all the source code that was used to perform the fractal coding and reconstruction process (after [Wall93]). Following that is the source code to perform the forward and inverse discrete wavelet transforms with coefficient shrinking (after [BCDJ95]).

```
**************************************************
File:  constants.h
This is the global type and constant definition file.
**************************************************

#define RANGE_SIZE 8        //size of range block in pixels
#define CODEWORDS 11        //size of the fscl codebook
#define SRCX 10
#define SRCY 22             //Source image co-ords
#define CBKX 2
#define CBKY 574            //Codebook image co-ords
#define RECX 532
#define RECY 22             //Reconstructed image co-ords

// TYPE DECLARATIONS

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
        };


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

typedef struct {
        float  pxls[24][24],
               s1,
               s2;
        } vector;

typedef struct {
        float  pxls[RANGE_SIZE][RANGE_SIZE],
               s1,
               s2;
        } vector2;

struct indx {
   int x,y;
   unsigned char transform;
   float  s1, s2;
   struct indx *next;
} ;
```

```
************************************************
File:  fbc.h
This file contains global variables.
************************************************

int SCALE, BLK_TEMP, IMAGE_SIZE, NUM_BLOCKS, DOMAIN_SIZE, DISP,
VERB, COMPRESS;
float ITERATIONS, SAMP_INCR;
char *SRC_FILE, *DEST_FILE, *CODE_FILE, *BATCH_FILE, *FBC_FILE;
int MAG;
float PSNR,SIG,RMS;
image_type source,codebook,recon;
fractal *FCODE1,*FCODE2;
vector2 *TEMP_CODEBOOK;
vector *CODEBOOK;
struct indx *VECTOR_CODE;
Dimension height,width;
char *text;


************************************************
File:  fbc.c
This file prepares the GUI and parses the command line for input.
************************************************

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

#include "xutils.h"
#include "constants.h"
#include "fbc.h"
#include "io.h"

#define XSIZE 1054
#define YSIZE 880
#define NUM_BUTS 3

void SetUp();
void CleanSet();

char *commands_list="-nodisplay-domain-source-code-dest-step-batch-verbose-u-h";

void Redraw(w, client_data,event) Widget w; XtPointer client_data; XEvent *event;
{
 char buffer[30];
 DisplayImage(MAIN,SRCX,SRCY,&source);
 DisplayImage(MAIN,CBKX,CBKY,&codebook);
 DisplayImage(MAIN,RECX,RECY,&recon);
 Text(SRCX,SRCY+IMAGE_SIZE+13,(String)"Source Image");
 Text(CBKX,CBKY+64+13,(String)"Codebook");
 Text(RECX,RECY+IMAGE_SIZE+13,(String)"Reconstructed Image");
 Text(RECX+IMAGE_SIZE/2,RECY+IMAGE_SIZE+13,(String)"PSNR = ");
 Text(RECX,RECY+IMAGE_SIZE+27,(String)"Number of Iterations:");
```

```
 gcvt(PSNR,8,buffer);
 strcat(buffer,"          ");
 Text(RECX+IMAGE_SIZE/2+45,RECY+IMAGE_SIZE+13,buffer);
 gcvt((double) ITERATIONS,8,buffer);
 strcat(buffer,"    ");
 Text(RECX+IMAGE_SIZE/2+10,RECY+IMAGE_SIZE+27,buffer);
 XtVaGetValues(MAIN,XtNheight,&height,XtNwidth,&width,NULL);
 XDrawLine(DISPLAY,XtWindow(MAIN),GC1,0,height-25,width,height-25);
}

void SetUp()
{
 source.xsize=source.ysize=recon.xsize=recon.ysize=IMAGE_SIZE;
 codebook.xsize=960;
 codebook.ysize=64;
 source.image = img_alloc(IMAGE_SIZE,IMAGE_SIZE);
 codebook.image = img_alloc(960,64);
 recon.image = img_alloc(IMAGE_SIZE,IMAGE_SIZE);
 NUM_BLOCKS=IMAGE_SIZE/RANGE_SIZE;
 FCODE1 = fcd_alloc();
 FCODE2 = fcd_alloc();
 CODEBOOK = cbk_alloc();
 TEMP_CODEBOOK=(vector2 *)calloc(CODEWORDS,sizeof(vector2));
 VECTOR_CODE = ndx_alloc();
 ITERATIONS=10;
 SCALE=DOMAIN_SIZE/RANGE_SIZE;
 MAG=0;
 BLK_TEMP=RANGE_SIZE;
 if(VERB)
    {
        printf("\n");
        printf("Image Size = %d\n",IMAGE_SIZE);
        printf("Range Block Size = %d\n",RANGE_SIZE);
        printf("Domain Block Size = %d\n",DOMAIN_SIZE);
        printf("Scale factor = %d\n",SCALE);
        printf("Step Size = %f\n",SAMP_INCR);
        printf("Diplay = %d\n",DISP);
        printf("Source File = %s\n",SRC_FILE);
        printf("Destination File = %s\n",DEST_FILE);
        printf("Codebook File = %s\n",CODE_FILE);
        printf("Batch File = %s\n",BATCH_FILE);
        printf("FBC File = %s\n",FBC_FILE);
        if(COMPRESS)
           printf("Compressing...\n");
        else
           printf("Uncompressing...\n");
    }
}

void CleanSet()
{
  XtAddEventHandler(MAIN,ButtonPressMask,FALSE,GetPosition,0);
  XtAddEventHandler(TOP_LEVEL,VisibilityChangeMask,False,Redraw,(XtPointer) 0);
}
```

```
void main(argc,argv)
 unsigned int argc;
 char **argv;
{
 button_list *buttons = (button_list *) calloc(NUM_BUTS,sizeof(button_list));
 int arg_ptr,com,bye=0;
 char *buffer;
 DISP=1;
 DOMAIN_SIZE=24;
 SAMP_INCR=12;
 IMAGE_SIZE=256;
 VERB=0;
 COMPRESS=1;
 SRC_FILE="NONE";
 CODE_FILE="lena512.cbk";
 DEST_FILE="NONE";
 BATCH_FILE="NONE";
 FBC_FILE="NONE";
 if(argc>1)
    {
        for(arg_ptr=1;arg_ptr<argc;arg_ptr++)
           {
               buffer=argv[arg_ptr];
               com=strstr(commands_list,buffer)-commands_list;
               switch(com)
                  {
                  case 0:
                        DISP=0;
                        break;
                  case 10:
                        DOMAIN_SIZE=atoi(argv[++arg_ptr]);
                        break;
                  case 17:
                        SRC_FILE=argv[++arg_ptr];
                        break;
                  case 24:
                        CODE_FILE=argv[++arg_ptr];
                        break;
                  case 29:
                        DEST_FILE=argv[++arg_ptr];
                        break;
                  case 34:
                        SAMP_INCR=atof(argv[++arg_ptr]);
                        break;
                  case 39:
                        BATCH_FILE=argv[++arg_ptr];
                        break;
                  case 45:
                        VERB=1;
                        break;
                  case 53:
                        COMPRESS=0;
                        FBC_FILE=argv[++arg_ptr];
```

```
                    break;
                case 55:
                    printf("Usage:  fbc [-nodisplay] [-source file] [-dest file] [-code file]
[-domain N] [-step S] [-batch file] [-verbose] [-u file] [h]\n");
                    exit(0);
                    break;
                }

            }
    }
 SetUp();
 if(DISP)
    {
        buttons[0].label="Transfer";
        buttons[1].label="Clear Target";
        buttons[2].label="Quantize Image";
        XtToolkitInitialize();
        CONTEXT=XtCreateApplicationContext();

DISPLAY=XtOpenDisplay(CONTEXT,NULL,"Test","Test",NULL,0,&argc,argv);
        MAIN = (Widget *) CreateMainWindow(XSIZE,YSIZE,"Reduced Search Fractal
Block Coding");
        CleanSet();
        CreateButtonSet(MAIN,0,700,NUM_BUTS,buttons,XmHORIZONTAL);
        CreateMenu("fbc_menu.script");
        XtRealizeWidget(TOP_LEVEL);
        XtAppMainLoop(CONTEXT);
    }
 if(!DISP)
    {
        if(strcmp(SRC_FILE,"NONE")==0)
            {
                printf("Need source file.\n");
                bye=1;
            }
        if(strcmp(DEST_FILE,"NONE")==0)
            {
                printf("Need destination file.\n");
                bye=1;
            }
        if(bye) exit(0);
        if(COMPRESS)
            {
                FILENAME=SRC_FILE;
                open_source();
                FILENAME=CODE_FILE;
                open_code();
                code();
                FILENAME=DEST_FILE;
                save_fbc();
            }
        else
            {
                if(strcmp(FBC_FILE,"NONE")==0)
```

```
                {
                    printf("Need FBC file.\n");
                    exit(0);
                }
            FILENAME=SRC_FILE;
            open_source();
            FILENAME=FBC_FILE;
            open_fbc();
            reconstruct();
            FILENAME=DEST_FILE;
            save_dest();
        }
    }
}
```

```
************************************************
File:  commands.c
This file operates the COMMAND module which routes commands from the GUI or the
command line.
************************************************

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

#include "xutils.h"
#include "constants.h"
#include "fbc.h"
#include "fractals.h"
#include "fscl.h"
#include "arithmetic.h"
#include "io.h"

void from_v2();
void to_v2();
void code();
void reconstruct();
void open_source();
void open_code();
void save_fbc();
void open_fbc();
void save_dest();

void ButtonRoute(w,client,call)
        Widget w;
        XtPointer client,call;
{
 int command = (int) client;
 switch(command)
        {
        case 0:
           memcpy(source.image,recon.image,source.xsize*source.ysize);
           DisplayImage(MAIN,SRCX,SRCY,&source);
           break;
        case 1:
           memset(recon.image,255,recon.xsize*recon.ysize);
           DisplayImage(MAIN,RECX,RECY,&recon);
           break;

        case 2:
           VQ_Image(source.image,CODEBOOK,VECTOR_CODE);
           DisplayImage(MAIN,RECX,RECY,&recon);
           printf("Done.\n");
           break;
        }

}
```

```
void MenuRoute(w,client,call)
        Widget w;
        XtPointer client,call;
{
 int command = (int) client;
 time_t ptime;
 struct tm *start, *finish;

XDefineCursor(DISPLAY,XtWindow(MAIN_WINDOW),XCreateFontCursor(DISPLAY
,150));
 switch(command)
        {
   case 11: CallFileSelection("Open","Open Image","*.pgm",0);
        break;
   case 12: CallFileSelection("Open","Open Codebook","*.cbk",1);
        break;
   case 13: CallFileSelection("Open","Open Fractal Block Code","*.fcd",2);
        break;
   case 21: CallFileSelection("Save","Save Codebook","*.cbk",3);
        break;
   case 22: CallFileSelection("Save","Save Fractal Block Code","*.fcd",4);
        break;
   case 23: CallFileSelection("Save","Save Fractal Reconstruction","*.pgm",5);
        break;
   case 31: PrintImage(&source);
        break;
   case 32: PrintImage(&codebook);
        break;
   case 33: PrintImage(&recon);
        break;
   case 40: printf("Bye.\n");
            free(source.image);
          free(codebook.image);
          free(recon.image);
          free(FCODE1);
          free(CODEBOOK);
          free(VECTOR_CODE);
          exit(0);
            break;
        case 110:
          ptime=time(NULL);
          InitializeCodebook(source.image,CODEBOOK);
          start=localtime(&ptime);
          printf("Codebook training start time:  ");
            printf(asctime(start));
          printf("\n");
            TrainCodebook(source.image,CODEBOOK);
          ptime=time(NULL);
          start=localtime(&ptime);
          printf("Codebook training finish time:  ");
            printf(asctime(start));
          printf("\n");
          ImageCodebook(CODEBOOK,codebook.image);
          DisplayImage(MAIN,CBKX,CBKY,&codebook);
```

```
        Text(0,height-7,"Codebook has been trained.                    ");
       break;
      case 120:
       code();
       break;
      case 130:
        reconstruct();
        break;
      case 140:
      {
        FILE *list;
        int i;
        list=fopen("list","r");
        while(feof(list)==0)
              {
                char *buffer;
                fscanf(list,"%s\n",FILENAME);
                buffer=FILENAME;
                printf("%s\n",FILENAME);
                img_load(FILENAME,&source);
                code();
                reconstruct();
                printf("PSNR = %f\n",PSNR);
                printf("-----------------------------\n");
                strcat(FILENAME,".rec.pgm");
                img_save(FILENAME,&recon);
                FILENAME=buffer;
                strcat(FILENAME,".rec.fcd");
                PreArithmetic(FCODE1);
                ArithmeticEncode(FILENAME,FCODE1);
                Text(0,height-7,"Fractal block code has been saved.        ");
              }
        fclose(list);
        break;
      }
      case 210:
      {
        gcvt(SAMP_INCR,2,PROMPT_DATA);
        CallPrompt("Enter step size.",&SAMP_INCR,PROMPT_DATA);
     break;
      }
      case 220:
        gcvt(ITERATIONS,2,PROMPT_DATA);
        CallPrompt("Enter number of iterations.",&ITERATIONS,PROMPT_DATA);
       break;
      case 310:

XDefineCursor(DISPLAY,XtWindow(MAIN_WINDOW),XCreateFontCursor(DISPLAY
,130));
        MAG=1;
        break;
      }
```

```
 if(MAG==0)
XDefineCursor(DISPLAY,XtWindow(MAIN_WINDOW),XCreateFontCursor(DISPLAY
,132));
}

void FileRoute(w,client,call)
        Widget w;
        XtPointer client,call;
{
 int command = (int) client;
 XtRemoveCallback(w,XmNokCallback,FileRoute,client);
 switch(command)
        {
        case 0:
          open_source();
          break;
        case 1:
          open_code();
          break;
        case 2:
          open_fbc();
          break;
        case 3:
          to_v2(CODEBOOK,TEMP_CODEBOOK);
          SaveCodebook(FILENAME,TEMP_CODEBOOK);
          Text(0,height-7,"Codebook has been saved.                ");
          break;
        case 4:
          save_fbc();
          break;
        case 5:
          save_dest();
          break;
        case 6:
            printf("%s\n",FILENAME);
         ArithmeticDecode(FILENAME,FCODE2);
            printf("Decompressed.\n");
            PostArithmetic(FCODE2);
            printf("Unpacked.\n");
            Text(0,height-7,"Second fractal block code has been loaded.      ");
         break;
        }

}

void from_v2(codebook,new)
vector2 codebook[CODEWORDS];
vector new[CODEWORDS];
{
  int i,x,y;
  for(i=0;i<CODEWORDS;i++)
        {
          for(y=0;y<RANGE_SIZE;y++)
               for(x=0;x<RANGE_SIZE;x++)
```

```
                    new[i].pxls[x][y]=codebook[i].pxls[x][y];
            new[i].s1=codebook[i].s1;
            new[i].s2=codebook[i].s2;
        }
}

void to_v2(codebook,new)
vector codebook[CODEWORDS];
vector2 new[CODEWORDS];
{
   int i,x,y;
   for(i=0;i<CODEWORDS;i++)
        {
          for(y=0;y<RANGE_SIZE;y++)
               for(x=0;x<RANGE_SIZE;x++)
                    new[i].pxls[x][y]=codebook[i].pxls[x][y];
          new[i].s1=codebook[i].s1;
          new[i].s2=codebook[i].s2;
        }
}

void code()
{
   time_t ptime;
   struct tm *start;
   ptime=time(NULL);
   start=localtime(&ptime);
   if(VERB)
        {
          printf("Quantizing image start time: ");
          printf(asctime(start));
          printf("\n");
        }
   VQ_Image(source.image,CODEBOOK,VECTOR_CODE);
   ptime=time(NULL);
   start=localtime(&ptime);
   if(VERB)
        {
          printf("Quantizing image finish time: ");
          printf(asctime(start));
          printf("\n");
        }
   ptime=time(NULL);
   start=localtime(&ptime);
   if(VERB)
        {
          printf("Fractal coding image start time: ");
          printf(asctime(start));
          printf("\n");
        }
   GenFractalCode(source.image,CODEBOOK,VECTOR_CODE,FCODE1);
   ptime=time(NULL);
   start=localtime(&ptime);
   if(VERB)
```

```
        {
          printf("Fractal coding image finish time:  ");
          printf(asctime(start));
          printf("\n");
        }
    if(DISP)
        Text(0,height-7,"Image has been fractal block coded (Reduced Search).");
}

void reconstruct()
{
    time_t ptime;
    struct tm *start;
    ptime=time(NULL);
    start=localtime(&ptime);
    if(VERB)
        {
          printf("Reconstruct image start time:  ");
          printf(asctime(start));
          printf("\n");
        }
    RecImage(FCODE1,recon.image);
    ptime=time(NULL);
    start=localtime(&ptime);
    if(VERB)
        {
          printf("Reconstruct image finish time:  ");
          printf(asctime(start));
          printf("\n");
        }
    PSNR=psnr(&source,&recon);
    RMS=rms(&source,&recon);
    if(DISP)
        {
          DisplayImage(MAIN,RECX,RECY,&recon);
          Text(0,height-7,"Image has been reconstructed.                    ");
        }
    if(VERB)
        printf("PSNR = %f, RMS = %f in %.0f
interations.\n",PSNR,RMS,ITERATIONS);
    MAG=0;
}

void open_source()
{
    img_load(FILENAME,&source);
    IMAGE_SIZE=source.xsize;
    recon.xsize=recon.ysize=IMAGE_SIZE;
    free(recon.image);
    recon.image = img_alloc(IMAGE_SIZE,IMAGE_SIZE);
    NUM_BLOCKS=IMAGE_SIZE/RANGE_SIZE;
    free(FCODE1);
    free(VECTOR_CODE);
    free(FCODE2);
```

```
  FCODE2 = fcd_alloc();
  FCODE1 = fcd_alloc();
  VECTOR_CODE = ndx_alloc();
  if(DISP)
      {
        DisplayImage(MAIN,SRCX,SRCY,&source);
        DisplayImage(MAIN,RECX,RECY,&recon);
        Text(0,height-7,"Source image has been loaded.          ");
      }
  if(VERB)
      {
        printf("The image file %s has been loaded.\n",FILENAME);
        printf("IMAGE_SIZE = %d\n",IMAGE_SIZE);
      }
}

void open_code()
{
  LoadCodebook(FILENAME,TEMP_CODEBOOK);
  from_v2(TEMP_CODEBOOK,CODEBOOK);
  ImageCodebook(CODEBOOK,codebook.image);
  if(DISP)
      {
        DisplayImage(MAIN,CBKX,CBKY,&codebook);
        Text(0,height-7,"Codebook has been loaded.          ");
      }
  if(VERB)
      printf("The codebook %s has been loaded.\n",FILENAME);
}

void save_fbc()
{
  PreArithmetic(FCODE1);
  ArithmeticEncode(FILENAME,FCODE1);
  if(DISP)
      Text(0,height-7,"Fractal block code has been saved.      ");
  if(VERB)
      printf("Fractal block code, %s has been saved.\n",FILENAME);
  PostArithmetic(FCODE1);
}

void open_fbc()
{
  ArithmeticDecode(FILENAME,FCODE1);
  PostArithmetic(FCODE1);
  if(DISP)
      Text(0,height-7,"Fractal block code has been loaded.      ");
  if(VERB)
      printf("The fractal block code %s has been loaded.\n",FILENAME);
}

void save_dest()
{
  img_save(FILENAME,&recon);
```

```
    if(DISP)
        Text(0,height-7,"Reconstructed image has been saved.        ");
    if(VERB)
        printf("The reconstucted image has been saved to %s.\n",FILENAME);
}
```

```
*************************************************
File:  fractals.h
This file contains function prototypes of the FRACTALS module.
*************************************************
extern void GenFractalCode();
extern void RecImage();
extern void PreArithmetic();
extern void PostArithmetic();


*************************************************
File:  fractals.c
This file operates the FRACTALS module.
*************************************************

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

#include "xutils.h"
#include "constants.h"
#include "fbc.h"
#include "io.h"
#include "transforms.h"
#include "fractals.h"



*************************************************
This function generates the reduced-search fractal block code.
*************************************************

void GenFractalCode(image, cbook, vq_code, fr_code)
 unsigned char *image;
 vector cbook[CODEWORDS];
 struct indx vq_code[CODEWORDS];
 fractal *fr_code;
{
#define  img(x,y) (image[y*IMAGE_SIZE+x])
/* #define  vq_c(a,b) (vq_code[b*NUM_BLOCKS+a]) */
#define  fr_c(i,j) (fr_code[j*NUM_BLOCKS+i])

 float   *low_err=(float *)calloc(NUM_BLOCKS*NUM_BLOCKS,sizeof(float));

 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 = RANGE_SIZE*RANGE_SIZE;
 float   error;
 int     c_transform;
 unsigned char  t_transform, transform;
 vectorr_block, s_block, t_block;
 struct indx *point;
```

```
 for (blk_y=0; blk_y < NUM_BLOCKS; blk_y++)
        for (blk_x=0; blk_x < NUM_BLOCKS; blk_x++)
                low_err[blk_x+blk_y*NUM_BLOCKS] = MAXFLOAT;
 for (img_y=0; img_y < (IMAGE_SIZE - DOMAIN_SIZE); img_y+=(int) SAMP_INCR)
        for (img_x=0; img_x < (IMAGE_SIZE - DOMAIN_SIZE); img_x+=(int)
SAMP_INCR)
        {
                ScaleBlock(img_x, img_y, image, &s_block);
                OrthBlock(&s_block);
                type = SearchCodebookDomain(s_block, cbook, &c_transform);
                t_transform = (unsigned char) c_transform;
                point=&vq_code[type];
                while(point->next != NULL)
                {
                        blk_x=point->x;
                        blk_y=point->y;
                        transform = t_t[t_transform][point->transform];
                        TransformBlock(s_block.pxls, t_block.pxls, transform);
                        b_x = blk_x * RANGE_SIZE;
                        b_y = blk_y * RANGE_SIZE;
                        rd = 0.0;
                        for (y=0; y < RANGE_SIZE; y++)
                                for (x=0; x < RANGE_SIZE; x++)
                                        rd += (float) img((b_x + x),(b_y + y)) *
t_block.pxls[x][y];
                        scale = (int) rd;
                        shift = point->s1/D;
                        error = point->s2 +
scale*scale*s_block.s2+shift*shift*D+2*(scale*shift*s_block.s1-scale*rd-shift*point-
>s1);
                        if ((low_err[blk_x+blk_y*NUM_BLOCKS] >= error) &&
(fabs(scale) < 1024))
                          {
                                low_err[blk_x+blk_y*NUM_BLOCKS] = error;
                                fr_c(blk_x,blk_y).x = img_x;
                                fr_c(blk_x,blk_y).y = img_y;
                                fr_c(blk_x,blk_y).transform = transform;
                                fr_c(blk_x,blk_y).scale = scale;
                                fr_c(blk_x,blk_y).translate = (int) shift;
                          }
                        point=point->next;
                }
        }

}


************************************************
This function reconstructs the image.
************************************************
void RecImage(fr_code, image)
fractal *fr_code;
unsigned char *image;
{
```

```
    unsigned char *img_t,*src,*temp;
#define  img(x,y) (image[y*IMAGE_SIZE+x])
#define TEMP(j,k) (img_t[k*IMAGE_SIZE+j])
#define fr_c(a,b) (fr_code[b*NUM_BLOCKS+a])

        vector  s_block, t_block;
        float   v,snr=0.0;
        int     i, blk_x, blk_y, x, y, img_x, img_y, tx,
ty,blk_x_start,blk_y_start,RANGE_SIZE,mx,my,iters;

        img_t = img_alloc(IMAGE_SIZE,IMAGE_SIZE);
        temp = img_alloc(IMAGE_SIZE,IMAGE_SIZE);
        if (MAG==0)
          {
                src=img_t;
                SCALE=DOMAIN_SIZE/RANGE_SIZE;
                BLK_TEMP=RANGE_SIZE;
                RANGE_SIZE=RANGE_SIZE;
                blk_x_start=0;
                blk_y_start=0;
          }
        else
          {
                src=source.image;
/*              src=img_t;  */
                BLK_TEMP=24;
                RANGE_SIZE=24;
                SCALE=1;
                mx=MOUSE[0]-SRCX;
                my=MOUSE[1]-SRCY;
                blk_x_start=my/RANGE_SIZE;
                blk_y_start=mx/RANGE_SIZE;
          }
        memset(img_t,0,IMAGE_SIZE*IMAGE_SIZE);
//      ITERATIONS = 10.0;
        for (i=0; i < (int) ITERATIONS; i++)
        {
                for (blk_y=blk_y_start; blk_y < NUM_BLOCKS; blk_y++)
                        for (blk_x=blk_x_start; blk_x < NUM_BLOCKS; blk_x++)
                        {
                                ScaleBlock(fr_c(blk_x,blk_y).x, fr_c(blk_x,blk_y).y,src,
&s_block);
                                OrthBlock(&s_block);
                                TransformBlock(s_block.pxls, t_block.pxls,
fr_c(blk_x,blk_y).transform);
                                for (y=0; y < RANGE_SIZE; y++)
                                        for (x=0; x < RANGE_SIZE; x++)
                                        {
                                                v = (float) (t_block.pxls[x][y] *
fr_c(blk_x,blk_y).scale) + fr_c(blk_x,blk_y).translate;
                                                if (v < 0.0)
                                                        v = 0.0;
                                                if (v > 255.0)
                                                        v = 255.0;
```

```
                                        tx=((blk_x-blk_x_start) * RANGE_SIZE) +
x;
                                        ty=((blk_y-blk_y_start) * RANGE_SIZE) +
y;
                                        if ((tx >= 0) && (tx < IMAGE_SIZE) &&
(ty >= 0) && (ty < IMAGE_SIZE))
                                            img(tx,ty) = (unsigned char) v;

                                    }
                }
        for (img_y = 0; img_y < IMAGE_SIZE; img_y++)
                for (img_x=0; img_x < IMAGE_SIZE; img_x++)
                        TEMP(img_x, img_y) = img(img_x,img_y);
        if(DISP)
          DisplayImage(MAIN,RECX,RECY,&recon);
/*
        if(psnr(&source,&recon)>snr)
          {
                snr=psnr(&source,&recon);
                if(DISP)
                  DisplayImage(MAIN,RECX,RECY,&recon);
                iters=i;

        memcpy(temp,recon.image,IMAGE_SIZE*IMAGE_SIZE*sizeof(unsigned char));
                }
                */
        }
//      memcpy(recon.image,temp,IMAGE_SIZE*IMAGE_SIZE*sizeof(unsigned char));
        free(img_t);
        free(temp);
//      ITERATIONS=(float) iters;
}

************************************************
This function formats the fractal block code for arithmetic entropy encoding.
************************************************

void PreArithmetic(code)
fractal *code;
{
#define fr_c(a,b) (code[b*NUM_BLOCKS+a])

        int     blk_x, blk_y, max, min;

        max = -5000;
        min = 5000;

        for(blk_y=0;blk_y < NUM_BLOCKS; blk_y++)
                for(blk_x=0;blk_x < NUM_BLOCKS; blk_x++)
                {
                        if (fr_c(blk_x,blk_y).scale > max)
                                max = fr_c(blk_x,blk_y).scale;
                        if(fr_c(blk_x,blk_y).scale < min)
                                min = fr_c(blk_x,blk_y).scale;
```

```
        }
    for(blk_y=0;blk_y < NUM_BLOCKS; blk_y++)
        for(blk_x=0;blk_x < NUM_BLOCKS; blk_x++)
        {
            fr_c(blk_x,blk_y).x /= (int) SAMP_INCR;
            fr_c(blk_x,blk_y).y /= (int) SAMP_INCR;
            fr_c(blk_x,blk_y).scale += 1024;
        }
    if(VERB)
    printf(" Range:  %d\n", max - min);

    for(blk_y=0;blk_y < NUM_BLOCKS; blk_y++)
        for(blk_x=0;blk_x < (NUM_BLOCKS-1); blk_x++)
            fr_c(blk_x,blk_y).translate = fr_c((blk_x+1),blk_y).translate -
fr_c(blk_x,blk_y).translate + 256;

    for(blk_y=0; blk_y < NUM_BLOCKS-1;blk_y++)
        fr_c((NUM_BLOCKS-1),(blk_y)).translate = fr_c((NUM_BLOCKS-
1),(blk_y+1)).translate - fr_c((NUM_BLOCKS-1),blk_y).translate + 256;

}

************************************************
This function recovers fractal code after arithmetic decompression.
************************************************

void PostArithmetic(code)
fractal *code;
{
#define fr_c(a,b) (code[b*NUM_BLOCKS+a])

    int     blk_x, blk_y, max, min;

    for(blk_y=0; blk_y < NUM_BLOCKS; blk_y++)
        for(blk_x=0; blk_x < NUM_BLOCKS; blk_x++)
        {
            fr_c(blk_x,blk_y).x *= (int) SAMP_INCR;
            fr_c(blk_x,blk_y).y *= (int) SAMP_INCR;
            fr_c(blk_x,blk_y).scale -= 1024;
        }

    for(blk_y=(NUM_BLOCKS-1); blk_y > 0; blk_y--)
        fr_c((NUM_BLOCKS-1),(blk_y-1)).translate = fr_c((NUM_BLOCKS-
1),blk_y).translate - fr_c((NUM_BLOCKS-1),(blk_y-1)).translate + 256;

    for(blk_y=0; blk_y<NUM_BLOCKS; blk_y++)
        for(blk_x=(NUM_BLOCKS-1); blk_x > 0; blk_x--)
            fr_c((blk_x-1),blk_y).translate = fr_c(blk_x,blk_y).translate -
fr_c((blk_x-1),blk_y).translate + 256;

}


************************************************
```

```
File:  fscl.h
This file contains function prototypes of the FSCL module.
*************************************************
extern void InitializeCodebook();
extern void TrainCodebook();
extern int SearchCodebookRange();
extern int SearchCodebookDomain();
extern void VQ_Image();
extern void ImageCodebook();


*************************************************
File:  fscl.c
This file operates the FSCL module.
*************************************************

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

#include "xutils.h"
#include "constants.h"
#include "fbc.h"
#include "transforms.h"
#include "fscl.h"



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

*************************************************
This function initializes the codebook so that it spans the image vector space with a random
pertibation.
*************************************************
void InitializeCodebook( image, cbook)
unsigned char  *image;
vector cbook[CODEWORDS];
{
#define  img(x,y) (image[y*IMAGE_SIZE+x])

        vector  avg;
        float   rnd_nm,*sig=malloc(RANGE_SIZE*RANGE_SIZE*sizeof(float));

        int     blk_x, blk_y, x, y, node;

        avg.s1 = avg.s2 = 0.0;
        for (y=0; y < RANGE_SIZE; y++)
                for (x=0;x < RANGE_SIZE; x++)
                        avg.pxls[x][y] = 0.0;
        for(blk_y = 0; blk_y < IMAGE_SIZE; blk_y+=RANGE_SIZE)
                for (blk_x=0; blk_x < IMAGE_SIZE; blk_x+=RANGE_SIZE)
                        for(y=0;y<RANGE_SIZE; y++)
```

```
                                for(x=0;x < RANGE_SIZE; x++)
                                        avg.pxls[x][y] += (float) (img(blk_x + x,blk_y +
y));
        node=0;
        for(y=0; y<RANGE_SIZE;y++)
                for(x=0; x<RANGE_SIZE;x++)
                {
                        avg.s1 += avg.pxls[x][y];
                        avg.s2 += avg.pxls[x][y]*avg.pxls[x][y];
                }

        OrthBlock(&avg);

        for(node=0;node<CODEWORDS;node++)
        {
                cbook[node].s1 = cbook[node].s2 = 0.0;
                for(y=0; y<RANGE_SIZE; y++)
                        for(x=0;x<RANGE_SIZE;x++)
                        {
                                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;
                        }
                OrthBlock(&cbook[node]);
        }
}

************************************************
This function trains the codebook on a given training imge.
************************************************
void TrainCodebook(image, cbook)
unsigned char *image;
vector cbook[CODEWORDS];
{

        vector  s_block, t_block;

        float   frequency[CODEWORDS],
                i_gn = 0.2,
                f_gn = 0.1,
                c_gn, w_chg, scale;

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

        long int        time, mx_time = CODEWORDS*1450;

        for (node=0;node<CODEWORDS;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() % (IMAGE_SIZE - DOMAIN_SIZE);
```

```
                rnd_y = (int) random() % (IMAGE_SIZE - DOMAIN_SIZE);
                ScaleBlock(rnd_x, rnd_y, image, &s_block);
                OrthBlock(&s_block);
                node = cbk_search_learn(s_block, cbook, frequency, &scale, &transform);
                TransformBlock(s_block.pxls, t_block.pxls, transform);
                cbook[node].s1 = cbook[node].s2 = 0.0;
                for(y=0;y<RANGE_SIZE;y++)
                        for(x=0;x<RANGE_SIZE;x++)
                        {
                                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];
                        }
                NormalizeBlock(&cbook[node]);
                frequency[node] += 1.0;
        }

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

*************************************************
This function searches the codebook for the best match to an input vector taking into
account the frequency count of the codewords.
*************************************************

static int cbk_search_learn(s_block, cbook, frequency, scale, transform)
vector s_block;
vector cbook[CODEWORDS];
float frequency[CODEWORDS];
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<CODEWORDS;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;
}

*************************************************
This function searches the codebook to classify a given range block.
*************************************************

int SearchCodebookRange(s_block,cbook,transform)
vector s_block;
vector cbook[CODEWORDS];
int *transform;
{
        int     x,
                y,
                node,
                t_transform,
                best_node = 0;

        float   error,
                low_err,
                t_scale;

        low_err = MAXFLOAT;

        for(node=0;node<CODEWORDS;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;
}

*************************************************
This function searches the codebook to classify a given domain block.
*************************************************

int SearchCodebookDomain(s_block,cbook,transform)
vector s_block;
vector cbook[CODEWORDS];
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<CODEWORDS;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;
}

************************************************
This function classifies each image vector of the image to be fractal coded.
************************************************

void VQ_Image(image,cbook,vq_code)
unsigned char *image;
vector cbook[CODEWORDS];
struct indx vq_code[CODEWORDS];
{
#define  img(x,y) (image[y*IMAGE_SIZE+x])

        vector  s_block;
        struct indx *point,*temp=NULL;
        int     blk_x,
                blk_y,
                x,
                y,
                t_transform,cx;
        float ts1,ts2;
        for(x=0;x<CODEWORDS;x++)
          {
                point=&vq_code[x];
                point=point->next;
                while(point != NULL)
                  {
                        temp=point->next;
                        free(point);
                        point=temp;
                  }
                point=&vq_code[x];
                point->next=NULL;
          }
        for (blk_y=0;blk_y<NUM_BLOCKS;blk_y++)
                for(blk_x=0;blk_x<NUM_BLOCKS;blk_x++)
                {
                        s_block.s1 = s_block.s2 = 0.0;
```

```
                        for(y=0;y<RANGE_SIZE;y++)
                                for(x=0;x<RANGE_SIZE;x++)
                                {
                                        s_block.pxls[x][y] = (float)
img((blk_x*RANGE_SIZE+x),(blk_y*RANGE_SIZE+y));
                                        s_block.s1 += s_block.pxls[x][y];
                                        s_block.s2 +=
s_block.pxls[x][y]*s_block.pxls[x][y];
                                }
                        ts1=s_block.s1;
                        ts2=s_block.s2;
                        OrthBlock(&s_block);
                        cx = SearchCodebookRange(s_block,cbook,&t_transform);
                        point = &vq_code[cx];
                        while(point->next != NULL) point=point->next;
                        point->s1=ts1;
                        point->s2=ts2;
                        point->x=blk_x;
                        point->y=blk_y;
                        point->transform = (unsigned char) t_transform;
                        point->next=(struct indx *)calloc(1,sizeof(struct indx));
                        point=point->next;
                        point->next=NULL;

                }
}

*************************************************
This function calculates 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++)
        {
                TransformBlock(a_block.pxls, t_block.pxls, t_transform);
```

```
                ab = 0.0;
                for(y=0;y<RANGE_SIZE;y++)
                        for (x=0;x<RANGE_SIZE;x++)
                                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;
}

*************************************************
This function generates a graphic version of the codebook.
*************************************************

void ImageCodebook(cbook,image)
vector cbook[CODEWORDS];
unsigned char *image;
{
#define imgc(x,y) (image[y*960+x])
  int       x,
                y,
                blk_x,
                blk_y,
                pxl_x,
                pxl_y,
                count;

  float max,
                min,
                scale,
                trans;

  memset(image,252,61440);
  for(count = 0; count < CODEWORDS; count++)
        {
          blk_x=count % CODEWORDS;
          blk_y=count / CODEWORDS;
          for(pxl_y=0;pxl_y<RANGE_SIZE;pxl_y++)
                for(pxl_x=0;pxl_x<RANGE_SIZE;pxl_x++)
```

```
                    for(y=0;y<8;y++)
                        for(x=0;x<8;x++)
                            imgc((blk_x*72+pxl_x*8+x),(blk_y*64 + pxl_y*8+y)) =
(unsigned char) (cbook[count].pxls[pxl_x][pxl_y]* 127 + 128);
            }
}
```

```
************************************************

File:  arithmetic.h

This file contains protype functions of the ARITHMETIC module.

************************************************

extern void ArithmeticEncode();
extern void ArithmeticDecode();


************************************************
File:  arithmetic.c
This file operates the ARITHMETIC module.
************************************************
#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

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

#include "constants.h"
#include "arithmetic.h"

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();
void daub4();


typedef struct {
        int     sbl_nm,
                ndx_nm;
        int     *index,
                *symbl;
        unsigned long int
                *prb,
                *cum;
        } statistics;

*************************************************
This function initializes 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;
}
*************************************************
This function updates the statistical model to reflect the character coded/received.
*************************************************
```

```
void update_stats(stats,smb)
statistics *stats;
int smb;
{
        int     VECTOR_CODE,
                ndx_2;

        VECTOR_CODE = 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=VECTOR_CODE;stats->prb[ndx_2] == stats->prb[ndx_2
+1];ndx_2++);

        if (ndx_2>VECTOR_CODE)
        {
                stats->index[smb] = ndx_2;
                stats->index[stats->symbl[ndx_2]] = VECTOR_CODE;
                stats->symbl[VECTOR_CODE] = stats->symbl[ndx_2];
                stats->symbl[ndx_2] = smb;
        }

        stats->prb[ndx_2] += 2;
        for(VECTOR_CODE=(ndx_2 + 1); VECTOR_CODE <= stats->ndx_nm;
VECTOR_CODE++)
                stats->cum[VECTOR_CODE] += 2;
}

static FILE *BitFile;
static int buffer, buff_count;
************************************************
This file opens 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;
}
************************************************
This function writes 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;
        }
}
*************************************************
This function closes the arithmetic code output stream.
*************************************************
void close_bit_out()
{
        buffer <<= (8-buff_count);
        putc(buffer,BitFile);

        fclose(BitFile);
}

static int garbage_count;
*************************************************
This function opens 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;
}
*************************************************
This funciton gets 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;
}
*************************************************
This function closse input bit stream.
*************************************************
void close_bit_in()
{
        fclose(BitFile);
}

static unsigned long int low, high;
static int follow_bits;

*************************************************
This function compresses FBC parameters using arithmetic encoding.
*************************************************
void ArithmeticEncode(FileName, fr_code)
char FileName[NM_SZ];
fractal *fr_code;
{

#define fr_c(a,b) (fr_code[b*NUM_BLOCKS+a])

        int     x,
                blk_x,
                blk_y,
                count;

        statistics      stats[FLD_NM];
        int     offset[FLD_NM];
        int     max[FLD_NM];
        if(VERB)
          printf("Starting arithmetic compression.\n");
        init_stats(&stats[0],IMAGE_SIZE/(int) SAMP_INCR);
        init_stats(&stats[1],IMAGE_SIZE/(int) SAMP_INCR);
        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<NUM_BLOCKS;blk_x++)
                for(blk_y=0;blk_y<NUM_BLOCKS;blk_y++)
                {
                        encode_smb(stats[0],fr_c(blk_x,blk_y).x);
                        update_stats(&stats[0],fr_c(blk_x,blk_y).x);
                        encode_smb(stats[1],fr_c(blk_x,blk_y).y);
                        update_stats(&stats[1],fr_c(blk_x,blk_y).y);
                        encode_smb(stats[2],fr_c(blk_x,blk_y).translate);
                        update_stats(&stats[2],fr_c(blk_x,blk_y).translate);
                        encode_smb(stats[3],fr_c(blk_x,blk_y).scale);
                        update_stats(&stats[3],fr_c(blk_x,blk_y).scale);
                        encode_smb(stats[4],fr_c(blk_x,blk_y).transform);
                        update_stats(&stats[4],fr_c(blk_x,blk_y).transform);
                }

        flush_encoder();
        close_bit_out();
}
*************************************************
This function initializes the arithmetic encoder.
*************************************************
void init_encoder()
{
        low=0;
        high=TOP;
        follow_bits=0;
}
*************************************************
This fucntion encodes 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;
        }
}
*************************************************
This function sends a bit to the output bit steam.
*************************************************
void bit_plus_follow(bit)
int bit;
{
        put_bit(bit);
        while(follow_bits>0)
        {
            put_bit( !bit );
            follow_bits--;
        }
}
*************************************************
This function sends 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;
*************************************************
This function recovers FBC parameters from an arithmetic code stream.
*************************************************
void ArithmeticDecode(FileName,fr_code)
char FileName[NM_SZ];
fractal *fr_code;
{
#define fr_c(a,b) (fr_code[b*NUM_BLOCKS+a])

        int     blk_x,
                blk_y;
```

```
        statistics      stats[FLD_NM];

        init_stats(&stats[0], IMAGE_SIZE / (int) SAMP_INCR);
        init_stats(&stats[1],IMAGE_SIZE/(int) SAMP_INCR);
        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 <NUM_BLOCKS; blk_x++)
                for(blk_y=0;blk_y<NUM_BLOCKS;blk_y++)
                {
                        fr_c(blk_x,blk_y).x = decode_smb(stats[0]);
                        update_stats(&stats[0],fr_c(blk_x,blk_y).x);
                        fr_c(blk_x,blk_y).y = decode_smb(stats[1]);
                        update_stats(&stats[1],fr_c(blk_x,blk_y).y);
                        fr_c(blk_x,blk_y).translate = decode_smb(stats[2]);
                        update_stats(&stats[2],fr_c(blk_x,blk_y).translate);
                        fr_c(blk_x,blk_y).scale = decode_smb(stats[3]);
                        update_stats(&stats[3],fr_c(blk_x,blk_y).scale);
                        fr_c(blk_x,blk_y).transform = decode_smb(stats[4]);
                        update_stats(&stats[4],fr_c(blk_x,blk_y).transform);
                }
        close_bit_in();
}
*************************************************
This function initializes the arithmetic decoder and fill the operating register with the first
NM_Code_Bits from the intput 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;
}

*************************************************
This function decodes 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;
}
```

```
************************************************

File:  transforms.h

This file contains function prototypes of the TRANSFORMS module.

************************************************

extern void ScaleBlock();
extern void OrthBlock();
extern void NormalizeBlock();
extern void TransformBlock();


************************************************
File:  transforms.c
This file operates the TRANSFORMS module.
************************************************
#include <stdlib.h>
#include <math.h>
#include <values.h>

#include "constants.h"
#include "transforms.h"



void identity();
void flip_x();
void flip_y();
void flip_d1();
void flip_d2();
void rotate_90();
void rotate_180();
void rotate_270();

************************************************
This function scales an image block from domain size to range size.
************************************************
void  ScaleBlock(x,y,image,s_block)
int x,y;
unsigned char *image;
vector *s_block;
{

#define  img(x,y) (image[y*IMAGE_SIZE+x])

        int     rng_x,
                rng_y,
                dmn_x,
                dmn_y;

        float   avg;
        s_block->s1 = s_block->s2 = 0.0;
```

```
        for(rng_y=0;rng_y<BLK_TEMP;rng_y++)
                for(rng_x=0;rng_x<BLK_TEMP;rng_x++)
                {
                        avg = 0.0;
                        for(dmn_y=0;dmn_y<SCALE;dmn_y++)
                                for(dmn_x=0;dmn_x<SCALE;dmn_x++)
                                        avg+=img((x+(rng_x*SCALE)+dmn_x),(y + (rng_y
* SCALE)+dmn_y));
                        s_block->pxls[rng_x][rng_y] = (avg / (SCALE * SCALE));
                        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];
                }
}

*************************************************
This function orthonormalizes a block.
*************************************************
void OrthBlock(s_block)
vector *s_block;
{
        int     x,
                y;
        float   ic;

        ic = s_block->s1 / (BLK_TEMP * BLK_TEMP);

        s_block->s1 = s_block->s2 = 0.0;
        for(y=0;y<BLK_TEMP;y++)
                for(x=0;x<BLK_TEMP;x++)
                {
                        s_block->pxls[x][y] -= ic;
                        s_block->s2 += s_block->pxls[x][y] * s_block->pxls[x][y];
                }
        NormalizeBlock(s_block);
}

*************************************************
This function selects and performs an isometric transform.
*************************************************
void TransformBlock(s_block, t_block, isom)
float s_block[24][24];
float t_block[24][24];
int isom;
{
        switch(isom)
        {
                case 0:identity(s_block,t_block);
                        break;
                case 1:flip_x(s_block,t_block);
                        break;
                case 2:flip_y(s_block,t_block);
                        break;
                case 3:flip_d1(s_block,t_block);
```

```
                break;
        case 4:flip_d2(s_block,t_block);
                break;
        case 5:rotate_90(s_block,t_block);
                break;
        case 6:rotate_180(s_block,t_block);
                break;
        case 7:rotate_270(s_block,t_block);
                break;
    }
}

*************************************************
This function normalizes a given block.
*************************************************
void NormalizeBlock(s_block)
vector *s_block;
{
    int     x,
            y;
    float   norm;

    norm = sqrt(s_block->s2);
    if (norm > 0.001)
    {
        for(y=0;y<BLK_TEMP;y++)
            for(x=0;x<BLK_TEMP;x++)
                s_block->pxls[x][y] /= norm;

        s_block->s1 /= norm;
        s_block->s2 = 1.0;
    }
}

*************************************************
This function performs the identity transform
*************************************************
void identity(s_block,t_block)
float s_block[24][24];
float t_block[24][24];
{
    int     x,
            y;

    for (y=0;y<BLK_TEMP;y++)
        for(x=0;x<BLK_TEMP;x++)
            t_block[x][y] = s_block[x][y];
}

*************************************************
This function performs the orthogonal reflection of a block about mid-vertical axis.
*************************************************
void flip_x(s_block,t_block)
float s_block[24][24];
```

```
float t_block[24][24];
{
        int     x,
                y;
        for(y=0;y<BLK_TEMP;y++)
                for(x=0;x<BLK_TEMP;x++)
                        t_block[x][y] = s_block[BLK_TEMP-1-x][y];
}

*************************************************
This function performs the orthogonal reflection of a block about mid-horizontal axis.
*************************************************

void flip_y(s_block,t_block)
float s_block[24][24];
float t_block[24][24];
{
        int     x,
                y;

        for(y=0;y<BLK_TEMP;y++)
                for(x=0;x<BLK_TEMP;x++)
                        t_block[x][y] = s_block[x][BLK_TEMP-1-y];
}


*************************************************
This function performs the orthogonal reflection of a block about the first diagonal.
*************************************************

void flip_d1(s_block,t_block)
float s_block[24][24];
float t_block[24][24];
{
        int     x,
                y;

        for(y=0;y<BLK_TEMP;y++)
                for(x=0;x<BLK_TEMP;x++)
                        t_block[x][y] = s_block[BLK_TEMP-1-y][BLK_TEMP-1-x];
}

*************************************************
This function performs the orthogonal reflection of a block about the second diagonal.
*************************************************

void flip_d2(s_block,t_block)
float s_block[24][24];
float t_block[24][24];
{
        int     x,
                y;

        for(y=0;y<BLK_TEMP;y++)
```

```
                for(x=0;x<BLK_TEMP;x++)
                        t_block[x][y] = s_block[y][x];
}


*************************************************
This function rotates a block 270 degrees about center.
*************************************************
void rotate_270(s_block,t_block)
float s_block[24][24];
float t_block[24][24];
{
        int     x,
                y;

        for(y=0;y<BLK_TEMP;y++)
                for(x=0;x<BLK_TEMP;x++)
                        t_block[x][y] = s_block[y][BLK_TEMP-1-x];
}


*************************************************
This function rotates a block 180 degrees about center.
*************************************************

void rotate_180(s_block,t_block)
float s_block[24][24];
float t_block[24][24];
{
        int     x,
                y;

        for(y=0;y<BLK_TEMP;y++)
                for(x=0;x<BLK_TEMP;x++)
                        t_block[x][y] = s_block[BLK_TEMP-1-x][BLK_TEMP-1-y];
}


*************************************************
This function rotates a block 90 degrees about center.
*************************************************

void rotate_90(s_block,t_block)
float s_block[24][24];
float t_block[24][24];
{
        int     x,
                y;

        for(y=0;y<BLK_TEMP;y++)
                for(x=0;x<BLK_TEMP;x++)
                        t_block[x][y] = s_block[BLK_TEMP-1-y][x];
}
```

```
*************************************************
File:  io.h
This file contains function prototype of the IO module.
*************************************************
extern void SaveFractalCode();
externvoid LoadFractalCode();
extern int LoadCodebook();
extern int SaveCodebook();
extern fractal *fcd_alloc();
extern vector *cbk_alloc();
extern struct indx *ndx_alloc();

*************************************************
File:  io.c
This file operates the IO module.
*************************************************

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

#include "constants.h"
#include "io.h"


#define SUCCESS 1
#define FAILURE 0
#define NM_SZ 20

*************************************************
This function allocates memort for the fractal block code.
*************************************************

fractal *fcd_alloc()
{
        fractal  *p;

        p = (fractal *) calloc(NUM_BLOCKS * NUM_BLOCKS,sizeof(fractal));
        if (!p) printf("Memory Allocation Error.\n");

        return p;
}
*************************************************
This function allocates memort for the codebook.
*************************************************

vector *cbk_alloc()
{
        vector  *p;
        p = (vector *) calloc(CODEWORDS,sizeof(vector));
        if (!p)
                printf("Memory Allocation Error.\n");
        return p;
}
```

```
*************************************************
This function allocates memort for the indexed image.
*************************************************

struct indx *ndx_alloc()
{
        struct indx *p;
        int i;
        p = (struct indx *) calloc (CODEWORDS, sizeof(struct indx));
        if(!p)
                printf("Memory Allocation Error.\n");
        for(i=0;i<CODEWORDS;i++) p[i].next = NULL;
        return p;
}

*************************************************
This function loads a fractal block code from a file.
*************************************************
void LoadFractalCode(filename,code)
char filename[NM_SZ];
fractal *code;
{
        FILE    *InFile;

        InFile = NULL;
        InFile = fopen(filename, "rb\n");
        if(InFile != NULL)
                fread(code,sizeof(fractal), (NUM_BLOCKS * NUM_BLOCKS), InFile);
        else
        {
                printf("File Not Found.\n");
                exit(3);
        }
        fclose(InFile);
}

*************************************************
This function saves a fractal block code to a file.
*************************************************
void SaveFractalCode(filename,code)
char filename[NM_SZ];
fractal *code;
{
        FILE    *OutFile;

        OutFile = NULL;
        OutFile = fopen(filename,"wb\n");
        if (OutFile != NULL)
                fwrite(code,sizeof(fractal), (NUM_BLOCKS * NUM_BLOCKS),
OutFile);
        else
        {
                printf("Unable to Open File.\n");
```

```
        }
        fclose(OutFile);
}

*************************************************
This function loads a codebook from a file.
*************************************************
int LoadCodebook(filename,cbook)
char filename[NM_SZ];
vector2 cbook[CODEWORDS];
{
        int     status;
        FILE    *InFile;

        InFile = fopen(filename, "rb\n");
        if(InFile != NULL)
        {
                if(fread(cbook,sizeof(vector2),CODEWORDS,InFile))
                        status=SUCCESS;
                else
                {
                        printf("File Read Error.\n");
                        status = FAILURE;
                }
                fclose(InFile);
        }
        else
        {
                printf("File Not Found.\n");
                status = FAILURE;
        }
        return status;
}


*************************************************
This function saves a codebook to a file.
*************************************************

int SaveCodebook(filename,cbook)
char filename[NM_SZ];
vector2 cbook[CODEWORDS];
{
        int     status;
        FILE    *OutFile;

        OutFile = fopen(filename,"wb\n");
        if (OutFile != NULL)
        {
                if (fwrite(cbook, sizeof(vector2), CODEWORDS, OutFile))
                        status = SUCCESS;
                else
                {
                        printf("File Write Error.\n");
```

```
                status = FAILURE;
            }
            fclose(OutFile);
        }
        else
        {
            printf("Cannot Open Output File.\n");
            status = FAILURE;
        }
        return status;
}
```

```
************************************************
File:  wave.h
This function contains constants and global variables for the Wavelet Denoising module.
************************************************
#define SX 10
#define TX 532
#define IX 542
#define DX 808

int IMG_SZ,FILTER,IT;
image_type IMG_1,IMG_2;
double *SIG,*LINE,*G,*H;
float THRESH,INCR,LOW,HIGH,MASK;


************************************************

File:  wave.c

This file prepares the GUI and routes input to the command module.

************************************************



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

#include "xutils.h"
#include "wave.h"

#define NUM_BUTS 8

void Refresh(w,client,event)
   Widget w;
   XtPointer client;
   XEvent *event;
{
 DisplayImage(MAIN,SX,10,&IMG_1);
 DisplayImage(MAIN,TX,10,&IMG_2);
 if((LOW != 0.0) && (HIGH==0.0))
    {
         gcvt(HIGH,8,PROMPT_DATA);
         CallPrompt("Enter upper limit.",&HIGH,PROMPT_DATA);
    }
 if((HIGH != 0.0) && (INCR==0.0))
    {
         gcvt(INCR,8,PROMPT_DATA);
         CallPrompt("Enter increment.",&INCR,PROMPT_DATA);
    }

}

main()
```

```
{
 int i;
 char **b;
 button_list *buttons = (button_list *) calloc(NUM_BUTS,sizeof(button_list));
 buttons[0].label="Open";
 buttons[1].label="Quit";
 buttons[2].label="Transform";
 buttons[3].label="I-Transform";
 buttons[4].label="Save";
 buttons[5].label="Load Filter";
 buttons[6].label="Cycle";
 buttons[7].label="Smooth";
 IMG_SZ=256;
 IMG_1.xsize=IMG_1.ysize=IMG_SZ;
 IMG_2.xsize=IMG_2.ysize=IMG_SZ;
 THRESH=0.0;
 IT=0;
 MASK=3.0;
 IMG_1.image=img_alloc(IMG_1.xsize,IMG_1.ysize);
 IMG_2.image=img_alloc(IMG_2.xsize,IMG_2.ysize);
 XtToolkitInitialize();
 CONTEXT=XtCreateApplicationContext();

DISPLAY=XtOpenDisplay(CONTEXT,NULL,"DISPLAY","DISPLAY",NULL,0,&i,b);
 MAIN=(Widget *) CreateMainWindow(TX+522,620,"Wavelets");
 XtAddEventHandler(TOP_LEVEL,VisibilityChangeMask,TRUE,Refresh,(XtPointer) 0);
 CreateButtonSet(MAIN,0,522,NUM_BUTS,buttons,XmHORIZONTAL);
 CreateMenu("wave.script");
 XtRealizeWidget(TOP_LEVEL);
 XtAppMainLoop(CONTEXT);
}
```

```
************************************************
File:  commands.c
This file routes commands to the appropriate module.
************************************************

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>

#include "xutils.h"
#include "wave.h"
#include "wt.h"

void transform();
int sign();
void inverse();
int comp(const void *,const void *);

void ButtonRoute(w,client,call)
        Widget w;
        XtPointer client,call;
{
 int command = (int) client;
 switch(command)
        {
        case 0:
           CallFileSelection("Open","Open Image","*.pgm",0);
           break;
        case 1:
        {
           printf("%f   %f   %f\n",LOW,HIGH,INCR);
           exit(0);
           break;
        }
        case 2:
        {
           transform();
           break;
        }
        case 3:
        {
           inverse();
           break;
        }
        case 4:  {
           CallFileSelection("Save","Save Image","*.pgm",1);
           break;
        }
        case 5:  {
           CallFileSelection("Load","Load Filter","*.fil",2);
           break;
        }
        case 6:
```

```
        {
          float i;
          for(i=LOW;i<=HIGH;i+=INCR)
                {
                  THRESH=i;
                  transform();
                  inverse();
                  gcvt(THRESH,8,FILENAME);
                  strcat(FILENAME,".pgm");
                  img_save(FILENAME,&IMG_2);
                }
          break;
        }
        case 7:
        {
          int x,y,i,j,ysize=IMG_1.ysize,xsize=IMG_1.xsize,mx,my,c;
          float t;
          for(y=0;y<ysize;y++)
                for(x=0;x<xsize;x++)
                  {
                        t=0.0;
                        c=0;
                        for(my=0;my<(int)MASK;my++)
                          for(mx=0;mx<(int)MASK;mx++)
                                {
                                  i=(x-(int)(MASK/2.0))+mx;
                                  j=(y-(int)(MASK/2.0))+my;
                                  if((i>0) && (i < xsize) && (j>0) && (j<ysize))
                                        {
                                          t+=(float)IMG_1.image[j*xsize+i];
                                          c++;
                                        }
                                }
                        t/=(float) c;
                        IMG_2.image[y*xsize+x]=(unsigned char) t;
                  }
          DisplayImage(MAIN,TX,10,&IMG_2);
          IMG_2.scale=IMG_1.scale;
          printf("PSNR = %f\n",psnr(&IMG_1,&IMG_2));
          break;
        }

        }

}


void MenuRoute(w,client,call)
        Widget w;
        XtPointer client,call;
{
 int command = (int) client;
 switch(command)
        {
```

```
        case 10:
        {
          gcvt(THRESH,8,PROMPT_DATA);
          CallPrompt("Enter threshold.",&THRESH,PROMPT_DATA);
          break;
        }
        case 20:
        {
          gcvt(LOW,8,PROMPT_DATA);
          CallPrompt("Enter lower limit.",&LOW,PROMPT_DATA);
          break;
        }
        case 30:
        {
          gcvt(MASK,8,PROMPT_DATA);
          CallPrompt("Mask size",&MASK,PROMPT_DATA);
          break;
        }
        }

}

void FileRoute(w,client,call)
        Widget w;
        XtPointer client,call;
{
 int command = (int) client;
 XtRemoveCallback(w,XmNokCallback,FileRoute,client);
 switch(command)
        {
        case 0:
          ClearBlock(SX,10,IMG_1.xsize,IMG_1.ysize);
          ClearBlock(TX,10,IMG_2.xsize,IMG_2.ysize);
          IMG_1.xsize=IMG_1.ysize=IMG_SZ;
          img_load(FILENAME,&IMG_1);
          free(IMG_2.image);
          IMG_2.xsize=IMG_1.xsize;
          IMG_2.ysize=IMG_1.ysize;
          IMG_2.image=img_alloc(IMG_2.xsize,IMG_2.ysize);
          DisplayImage(MAIN,SX,10,&IMG_1);
          DisplayImage(MAIN,TX,10,&IMG_2);
          break;
        case 1:
          {
                img_save(FILENAME,&IMG_2);
                printf("Image has been saved.\n");
                break;
          }
        case 2:
        {
          FILE *in;
          int i;
          double sign;
          free(G);
```

```
            free(H);
            in=fopen(FILENAME,"r");
            fscanf(in,"%d\n",&FILTER);
            G=(double *)calloc(FILTER,sizeof(double));
            H=(double *)calloc(FILTER,sizeof(double));
            for(i=0;i<FILTER;i++)
                  {
                    fscanf(in,"%30le\n",&sign);
                    H[i]=sign;
                  }
            sign=1.0;
            fclose(in);
            printf("done.\n");
            for(i=0;i<FILTER;i++)
                  {
                    G[i]=sign*H[i];
                    sign*=-1.0;
                  }
            break;
          }

          }
}


************************************************
This function performs the foward wavelet transform and applies threholding.
************************************************

void transform()
{
   int i,j,nn,count;
   double  value,min=99999.0,max=0.0,median,*buf,res,thr;
   unsigned char ch;
   printf("Threshold = %10f\n",THRESH);
   SIG=(double *)malloc((IMG_1.xsize*IMG_1.ysize+1)*sizeof(double));
   buf=(double *)malloc((IMG_1.xsize*IMG_1.ysize+1)*sizeof(double));
   for(i=0;i<IMG_1.xsize*IMG_1.ysize;i++)
         {
           SIG[i]=(double)IMG_1.image[i]/IMG_1.xsize;
         }
    fwt2d(SIG,IMG_1.xsize,IMG_1.xsize,G,H,1);
   nn=0;
   for(i=0;i<IMG_1.xsize;i++)
         for(j=0;j<IMG_1.ysize;j++)
           if((i>=IMG_1.xsize/2) || (j>=IMG_1.ysize/2))
                buf[nn++]=fabs(SIG[j+i*IMG_1.xsize]);
   qsort((void *) buf,(size_t)nn,(size_t)sizeof(double),comp);
   median =(buf[(nn/2)-1]+buf[nn/2])/2;
   thr=median*sqrt(2*log10((double)(IMG_1.xsize*IMG_1.ysize)))/IMG_1.xsize;
   printf("Transform complete.  Median = %10f   Calc. threshold = %10f
%d\n",median,thr,nn);
   IT=1;
   for(i=0;i<IMG_1.xsize*IMG_1.ysize;i++)
```

```
        {
          value=SIG[i];
          if(value>(double)THRESH)
                SIG[i]=value-(double)THRESH;
          else if(value<(double)-THRESH)
                SIG[i]=value+(double)THRESH;
          else SIG[i]=0.0;

          if(value < min) min=value;
          if(value > max) max=value;
          IMG_2.image[i]=0;
        }
   for(i=0;i<IMG_1.xsize*IMG_1.ysize;i++)
        {
          value=SIG[i];
          if (value<0) ch=0;
          else ch=(unsigned char) (((value-min)/(max-min))*255.0);
          IMG_2.image[i]=(unsigned char) (fabs(value)*255.0);
        }
   DisplayImage(MAIN,TX,10,&IMG_2);
   printf("The smallest coefficient is: %f\n \n",min);
   printf("The biggest coefficient is: %f\n",max);
   free(buf);
}



int sign(num)
double num;
{
   if (num<0) return(-1);
   if (num==0) return (0);
   if (num>0) return(1);
}
*************************************************
This function performs the inverse wavelet transform
*************************************************
void inverse()
{
   int i;
   double  value,min=99999.0,max=0.0;
   unsigned char ch;
   if(IT==0)
        {
          SIG=(double *)malloc((IMG_1.xsize*IMG_1.ysize+1)*sizeof(double));
          for(i=0;i<IMG_1.xsize*IMG_1.ysize;i++) SIG[i]=(double)IMG_1.image[i];
        }
    iwt2d(SIG,1,IMG_1.xsize,G,H);
   for(i=0;i<IMG_1.xsize*IMG_1.ysize;i++)
        {
          value=SIG[i]*(double) IMG_1.xsize;
//        ch=(unsigned char) (((value-min)/(max-min))*255.0);
          IMG_2.image[i]=(unsigned char) value;
        }
```

```
    DisplayImage(MAIN,TX,10,&IMG_2);
    IMG_2.scale=IMG_1.scale;
    printf("PSNR = %f\n",psnr(&IMG_1,&IMG_2));
    free(SIG);
    IT=0;
}
int comp(const void *a, const void *b)
{
  double x = *((double *) a);
  double y = *((double *) b);

  if (x < y) return(-1);
  if (x == y) return(0);
  return(1);
}
```

```
*************************************************

File:  wt.h

This file contains function prototypes for the DWT and IDWT modules.

*************************************************

extern void fwt2d();
extern void iwt2d();
extern void downlo();
extern void downhi();
extern void updyadlo();
extern void updyadhi();
extern void iconv();
extern void aconv();


*************************************************

File:  wt.c

This file operates the DWT and IDWT modules.

*************************************************

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

#include "xutils.h"
#include "wave.h"
#include "wt.h"

void downlo();
void downhi();
void updyadlo();
void updyadhi();
void upsample();
void iconv();
void aconv();

*************************************************
This function performs the foward 2-d wavelet transform.
*************************************************
void fwt2d(in,dim,size,g,h,limit)
double *in;
int dim,size;
double *g,*h;
{
   double *row,*wk,*out;
   int i,j,k;
   wk=(double *)calloc(dim+FILTER,sizeof(double));
```

```
    row=(double *)calloc(dim,sizeof(double));
    out=(double *)calloc(size*size,sizeof(double));
    while((dim/2)>=limit)
          {
            for(i=0;i<dim;i++)
                  {
                    memcpy(row,in+(i*size),dim*sizeof(double));
                    downlo(row,wk,h,dim);
                    memcpy(in+(i*size),wk,dim/2*sizeof(double));
                    downhi(row,wk,g,dim);
                    memcpy(in+(i*size+dim/2),wk,dim/2*sizeof(double));
                  }
            for(i=0;i<dim;i++)
                  {
                    for(k=0;k<dim;k++)  row[k]=in[k*size+i];
                    downlo(row,wk,h,dim);
                    for(k=0;k<dim/2;k++) in[k*size+i]=wk[k];
                    downhi(row,wk,g,dim);
                    for(k=dim/2;k<dim;k++) in[k*size+i]=wk[k-dim/2];
                  }
            dim/=2;
          }
    free(wk);
    free(row);
    free(out);
}

*************************************************
This function performs the inverse 2-D wavelet transform.
*************************************************
void iwt2d(in,dim,size,g,h)
double *in;
int dim,size;
double *g,*h;
{
    double *row,*temp,*sum1,*sum2;
    int i,j,k;
    row=(double *)calloc(size,sizeof(double));
    sum1=(double *)calloc(size+FILTER,sizeof(double));
    sum2=(double *)calloc(size+FILTER,sizeof(double));
    temp=(double *)calloc(size*size,sizeof(double));
    memcpy(temp,in,size*size*sizeof(double));
    while(dim<=size/2)
          {
            for(i=0;i<2*dim;i++)
                  {
                    for(k=0;k<dim;k++)  row[k]=temp[k*size+i];
                    updyadlo(row,sum1,h,dim);
                    for(k=dim;k<2*dim;k++)  row[k-dim]=temp[k*size+i];
                    updyadhi(row,sum2,g,dim);
                    for(k=0;k<2*dim;k++) temp[k*size+i]=sum1[k]+sum2[k];
                  }
            for(i=0;i<2*dim;i++)
                  {
```

```
                for(k=0;k<dim;k++) row[k]=temp[i*size+k];
                updyadlo(row,sum1,h,dim);
                for(k=dim;k<2*dim;k++) row[k-dim]=temp[i*size+k];
                updyadhi(row,sum2,g,dim);
                for(k=0;k<2*dim;k++) temp[i*size+k]=sum1[k]+sum2[k];
            }
        dim*=2;
    }
  for(i=0;i<size*size;i++) in[i]=temp[i];
  free(row);
  free(sum1);
  free(sum2);
  free(temp);
}

*************************************************
This function downscales and convolves with the low frequency filter.
*************************************************
void downlo(in,out,f,dim)
double *in,*out;
double *f;
int dim;
{
  int i;
  aconv(in,out,f,dim);
  for(i=0;i<dim;i++) out[i]=out[2*i];
}
*************************************************
This function downscales and convolves with the high frequency filter.
*************************************************
void downhi(in,out,f,dim)
double *in,*out;
double *f;
int dim;
{
  int i;
  double *temp;
  temp=(double *)calloc(dim,sizeof(double));
  temp[dim-1]=in[0];
  memcpy(temp,in+1,(dim-1)*sizeof(double));
  iconv(temp,out,f,dim);
  for(i=0;i<dim;i++) out[i]=out[2*i];
  free(temp);
}
*************************************************
This function ipscales and convolves with the low frequency filter.
*************************************************
void updyadlo(in,out,f,dim)
double *in,*out;
double *f;
int dim;
{
  double *wk;
  wk=(double *)calloc(2*dim,sizeof(double));
```

```
   upsample(in,wk,dim);
   iconv(wk,out,f,2*dim);
   free(wk);
}



*************************************************
This function ipscales and convolves with the high frequency filter.
*************************************************

void updyadhi(in,out,f,dim)
double *in,*out;
double *f;
int dim;
{
   int i;
   double *temp,*wk;
   wk=(double *)calloc(2*dim,sizeof(double));
   temp=(double *)calloc(2*dim,sizeof(double));
   upsample(in,wk,dim);
   temp[0]=wk[2*dim-1];
   for(i=2*dim-1;i>0;i--) temp[i]=wk[i-1];
   aconv(temp,out,f,2*dim);
   free(wk);
   free(temp);
}

*************************************************
This function upsamples a line.
*************************************************
void upsample(in,out,dim)
double *in,*out;
int dim;
{
   int i;
   for(i=0;i<2*dim;i++) out[i]=0.0;
   for(i=0;i<dim;i++) out[2*i]=in[i];
}

*************************************************
This function
void iconv(in,out,f,dim)
double *in,*out;
double *f;
int dim;
{
   int i,j;
   double *temp,b;
   temp=(double *)calloc(dim+FILTER,sizeof(double));
   for(i=FILTER-1;i<dim+FILTER;i++)
         {
```

```
            for(j=i-(FILTER-1);j<=i;j++)
                {
                  if((j>=0) && (j<dim+FILTER))
                    {
                        b=in[(FILTER*dim-FILTER+j)%dim];
                        temp[i]+=f[i-j]*b;
                    }
                }
        }
    memcpy(out,temp+FILTER,dim*sizeof(double));
    free(temp);
}

void aconv(in,out,f,dim)
double *in,*out;
double *f;
int dim;
{
    int i,j;
    double *temp,b;
    temp=(double *)calloc(dim+FILTER,sizeof(double));
    for(i=FILTER-1;i<dim+FILTER;i++)
        {
          for(j=i-(FILTER-1);j<=i;j++)
                {
                  if((j>=0) && (j<dim+FILTER))
                    {
                        b=in[j%dim];
                        temp[i]+=f[(FILTER-1)-(i-j)]*b;
                    }
                }
        }
    memcpy(out,temp+FILTER-1,dim*sizeof(double));
    free(temp);
}
```

APPENDIX D

SOURCE CODE FOR SOLVING THE INVERSE PROBLEM OF IFS

This appendix contains calc.c and ria.c that were written to solve the CAT parameters and construct an image based on those parameters given user selected points [Bal95].

```
************************************************

File:  calc.c

This file implements the calculation of the CAT parameters.

************************************************

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

#include "/home/unix/sbal/bin/include/ifs.h"
#include "/home/unix/sbal/bin/include/calc.h"

void Calculate_parameters(xyi,ifs)
 int xyi[6][2];
 ifs_struct ifs[10];
{
 float xy[6][2],c[2],s[2],p[4],cat[6];
 int i,j;
 for(i=0;i<6;i++)
 {
  xy[i][0]=(float) xyi[i][0];
  xy[i][1]=(float) xyi[i][1];
 }
 for(i=0;i<2;i++)
   {
    c[i]=xy[i+1][0]/xy[0][0];
    s[i]=1-c[i];
   }
 for(i=0;i<2;i++)
   {
    p[0]=xy[5][i]-c[1]*xy[3][i];
    p[1]=xy[4][i]-c[0]*xy[3][i];
    p[2]=xy[2][1]-c[1]*xy[0][1];
    p[3]=xy[1][1]-c[0]*xy[0][1];
    cat[i*3+2]=(p[0]-(p[2]/p[3])*p[1])/(s[1]-(p[2]/p[3])*s[0]);
    cat[i*3+1]=(p[1]-cat[i*3+2]*s[0])/p[3];
    cat[i*3]=(xy[3][i]-cat[i*3+1]*xy[0][1]-cat[i*3+2])/xy[0][0];
   }
 for(i=0;i<6;i++) ifs[cat_count].cat[i] = cat[i];
 DisplayIFS(ifs);
}

void DisplayIFS(ifs)
 ifs_struct ifs[10];
{
 int i,j,k;
 char buf1[50],buf2[10];
 ClearBlock(IFS_X,IFS_Y-13,219,300);
 Text(IFS_X,IFS_Y-20,"IFS Parameters");
 for(i=0;i<=cat_count;i++)
 for(j=0;j<2;j++)
 {
```

```
    gcvt(ifs[i].cat[j*3],4,buf1);
    for(k=1;k<3;k++)
    {
     gcvt(ifs[i].cat[k+j*3],4,buf2);
     strcat(buf1," ");
     strcat(buf1,buf2);
    }
    strcat(buf1," ");
    gcvt(ifs[i].p,4,buf2);
    strcat(buf1,buf2);
    Text(IFS_X,j*13+IFS_Y+i*40,buf1);
  }
}
```

****

File: ria.c

This file implements the construction of the image using CAT parameters.

****

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

#include "/home/unix/sbal/bin/include/xtools.h"
#include "/home/unix/sbal/bin/include/ifs.h"


void CalcProbs();

void RIA(ifs,cats,image)
    ifs_struct ifs[10];
    int cats;
    unsigned char image[IMG_SZ][IMG_SZ];
{
  int i,j,ix,iy,w,t,numt=50000;
  float x,y,nx,ny,rndm,prob;
  CalcProbs(ifs,cats);
  XSetForeground(DISPLAY,GC1,0);
  x=y=0.0;
  randomize();
  for(t=0;t<numt;t++)
  {
   rndm=(float) random(100)/100.0;
   prob=0.0;
   for(i=0;i<=cats;i++)
         {
        prob+=ifs[i].p;
        if(prob>=rndm)
```

```
            {
                w=i;
                break;
            }
        }
    nx=ifs[w].cat[0]*x + ifs[w].cat[1]*y + ifs[w].cat[2];
    ny=ifs[w].cat[3]*x + ifs[w].cat[4]*y + ifs[w].cat[5];
    ix=nx;
    iy=ny;
    if (ix < IMG_SZ && iy < IMG_SZ  && ix > -1 && iy > -1 && t>10)
    {
        image[iy][ix]=255;
//      XDrawPoint(DISPLAY,XtWindow(MAIN),GC1,DESX+ix,DESY+iy);
    }
    x=nx;
    y=ny;
  }
 DisplayImage(MAIN,DESX,DESY,IMG_SZ,IMG_SZ,IMG_2);
}


void CalcProbs(ifs,cats)
    ifs_struct ifs[10];
    int cats;
{
  float sum=0.0,det,probs[10],temp;
  int i,j,index,order[10];
  switch(PROB)
     {
     case 0:
          for(i=0;i<=cats;i++)
             {
                det=(ifs[i].cat[0]*ifs[i].cat[4])-(ifs[i].cat[1]*ifs[i].cat[3]);
                det=sqrtf(det*det);
                sum+=det;
             }
          for(i=0;i<=cats;i++)
             {
                ifs[i].p = ((ifs[i].cat[0]*ifs[i].cat[4])-(ifs[i].cat[1]*ifs[i].cat[3]))/sum;
                ifs[i].p = sqrtf(ifs[i].p*ifs[i].p);
             }
          break;
     case 1:
          for(i=0;i<=cats;i++)
            ifs[i].p = 1/(double) (cats+1);
          break;
     case 2:
          for(i=0;i<=cats;i++)
             {
                det=(ifs[i].cat[0]*ifs[i].cat[4])-(ifs[i].cat[1]*ifs[i].cat[3]);
                det=sqrtf(det*det);
                sum+=det;
             }
          for(i=0;i<=cats;i++)
```

```
            {
                ifs[i].p = ((ifs[i].cat[0]*ifs[i].cat[4])-(ifs[i].cat[1]*ifs[i].cat[3]))/sum;
                ifs[i].p = sqrtf(ifs[i].p*ifs[i].p);
            }
        for(i=0;i<=cats;i++)
            {
                temp=ifs[i].p;
                index=i;
                for(j=i+1;j<=cats;j++)
                    {
                        if (ifs[j].p<temp)
                            {
                                temp=ifs[j].p;
                                index=j;
                            }
                    }
                order[i]=index;
            }
        for(i=0;i<=cats/2;i++)
            {
                temp=ifs[order[i]].p;
                ifs[order[i]].p=ifs[order[cats-i]].p;
                ifs[order[cats-i]].p=temp;
            }
    }
        DisplayIFS(ifs);
}
```