IMAGE COMPRESSION USING WAVELET TRASFORM

BY

MUHAMMAD SHOAIB

QUAID-I-AZAM UNIVERSITY

DEPARTMENT OF ELECTRONICS
QUAID-I-AZAM UNIVERSITY
ISLAMABAD PAKISTAN

July, 1998

Certificate

Certified that the work contained in this dissertation was carried out by Mr. Muhammad Shoaib under my supervision.

(Dr. Ijaz Mansoor Qureshi)
Department of Electronics
Quaid-i-Azam University
Islamabad, Pakistan

Submitted through:

(Dr. M. H. Chohan)
Chairman
Department of Electronics
Quaid-i-Azam University
Islamabad, Pakistan

This work is submitted as dissertation
in partial fulfilment of
the requirements for the degree of

MASTER OF PHILOSOPHY

in

ELECTRONICS

Department of Electronics
Quaid-i-Azam University
Islamabad, Pakistan

Dedicated to my
Parents, Brothers and
Sisters

Acknowledgment

In The Name of ALLAH, The most gracious and merciful, who gave me an opportunity to complete this dissertation. I'm deeply debted to my supervisor Dr. Ijaz Mansoor Qureshi for his continued guidance and inspiration, His insight and suggestions have been indispensable for my research work. It has been rewarding, fruitful and above all fun to work with him.

I'm thankful to my friends, Ahsan, Faisal and Zahid for providing me a valuable material relevant to my research work. Specially Ahsan who helped me beyond my expectation. I'd like to thank Dr. Q. A. Naqvi, Dr. A. A. Rizvi and Dr. Nisar for providing me computer facilities and setting a high standard for me, which really helped me - learn and grow. Special thanks goes to Akram, and Zahoor for their care and attention during my M.Phil.

My groupmates Aliya, Ateka, Safia, Sarwar and classmates deserve credit for fostering a healthy and friendly environment, their valuable suggestions and discussions were a real help during this dissertation.

Finally, the debt I owe to my parents, brothers, sisters and family, who has always remembered me in their prayers, has no measure.

Muhammad Shoaib

ABSTRACT

A wavelet based algorithm for digital image compression is presented. The algorithm has good compression ratio as well as PSNR. A comparison with the standard JPEG algorithm for a number of images is also presented. This algorithm has been applied to gray scale images but can be extended for color.

Contents

Chapter 1

Introduction

This work deals with the image coding using wavelet transform . It can be better described by casting it in the framework of transform image coding . In this typical state of the art, transform image coding system , the encoder consists of a linear transform operation, followed by quantization of the transform-domain coefficients, and lossless compression of the quantized coefficients using an entropy coder . After the encoded bit stream of an input image is transmitted over the channel (assumed to be perfect), the decoder undoes all the functionalities applied in the encoder and tries to reconstruct a decoded image that looks as close as possible to the original input image, based on the transmitted information . This source coding paradigm has become the de-facto standard for lossy image compression application such as JPEG [1] and MPEG [2] , where the only loss of information occurs in the quantizer . See Figure 1.1 .

The basic idea behind using a linear transformation is to make the task of compressing an image in the transform domain after quantization easier than direct coding in the spatial domain . A good candidate transformation should be able to offer flexible image representation decorrelation (to facilitate efficient entropy coding) and good energy compaction in the transform domain (so that fewer quantized coefficients are needed to be encoded and the rest can be discarded for minimum distortion) . It is also desirable for the transform to be orthogonal so that the energy is conserved from the spatial domain to the transform domain , and the distortion in the spatial domain introduced by quantization of transform coefficients can be directly examined in the transform domain . Finding the optimal orthogonal transform for an $N \times N$ image necessitates a search over the set of all $N^2 \times N^2$ unitary matrices , which is clearly inpossible , since such a set of unitary matrices is infinite . In practice , suboptimal approximations such as the discrete cosine transform (DCT) are used for computational efficiency and being image independent [3] .

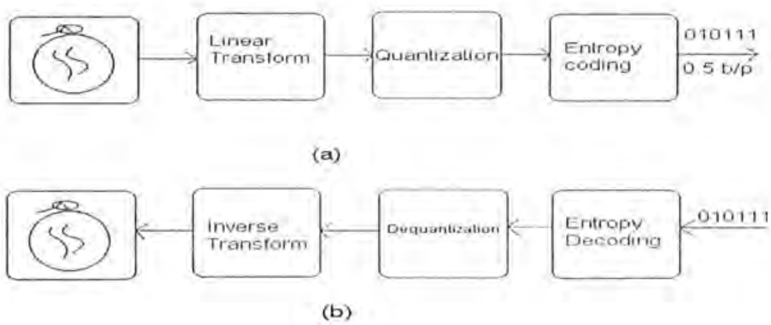

Figure 1.1: Block diagram of typical transform coding system (a) The encoder block (b) The decoder block diagram

Transform coding (DCT) is an efficient block oriented image compression techniques that is now being widely used in the image compression industry. while various discrete transforms have been investigated for application to transform coding, only the discrete cosine transform (DCT) has emerged as the most practical and efficient transform [11]. The principle used for transform coding is to resolve the original subpicture into a linear combination of a set of predefined subpictures, called *basis function*.The transform coefficients, which form the transmitted informations, are the multiplying factors of the basis functions. The bit rate reduction is operated by quantizing the transform coefficients before transmission.

To achieve a high compression ratio, most of the transform coefficients are coarsely quantized. Coarse quantization of the transform coefficients results in various artifacts in the coded images. Methods for reducing the block-artifacts in the transform coding of the images has been extensively studied. One of them is to incorporate the HVS (Human Visual System) properties with designing the quantization matrix (Q-matrix) for the transform coding since the human eye has discriminative sensitivity to different spatial frequencies. Another approach to cope with this problem is to replace the DCT basis functions by discrete wavelet basis [4], which has shorter basis functions for higher frequencies, and longer basis functions for lower frequencies as shown in figure[3.1]. There are more samples to represent the higher frequency sub-bands, than the lower frequencies ones. Therefore, sharp edges, which are well localized spatially and have significant high-frequency contents, can be represented more compactly with the DWT than with the DCT.

Recently , wavelets and filter bank theory [5, 6, 7, 8, 9, 10,] , together with their generalization such as wavelet packet (W P) [5, 6, 20,] , have appeared as alternatives to the DCT basis due to their ability to provide more flexible space-frequency resolution trade-off image representation.

1.1 Layout of the dissertation

The research work carried out and described in this dissertation constitutes a lossy compression algorithm. it is based on wavelet transform. Before going into details of the actual work existing image compression methods and wavelet based algorithm are discussed. chapter 2 discusses the need for image compression and standard image compression techniques. Chapter 3 deals with the wavelet analysis. Compression using wavelet transform and its results are discussed in chapter 4. Some real images along with the processed images are presented in chapter 4.

Chapter 2

Image Compression

2.1 Introduction

Digital image contain large amounts of information therefore from the stand points of data storage and transmission, one would like to have an image stored in a way that requires fewer bits. If one is processing a large number of images, then efficient representation is necessary in order not to overwhelm memory and if images are to be transmitted then there are bandwidth limitations so that timely transmission requires efficient representation. Thus data need to be compressed, or coded in such a way as to facilitate storage and transmission. Further more, various image representations enhance the speed of various algorithms. A key here is elimination of redundancy, and various transformation serve to reduce various type of redundancy. More than simply finding an efficient image representation, image are often altered in a noninvertible manner. Since this involve the loss of information, such compression is termed 'lossy' as opposed to invertible encoding, which are termed lossless.

2.2 Why compression is necessary?

"A picture is worth a thousand words." This English aphorism reminds us of the importance of images. It is especially true in the age of information highway and multimedia. Computers, fax machines, video phones, teleconferencing system and storage devices impact our workplace. Text, data, sound, images and video clip are grouped together to send over data networks or to store The amount of data is astronomical . Compression increases the throughput of the network and the capacity of the storage device .For satellite transmission,compression greatly reduces cost .

To understand why compression is needed it is a good idea to start with the analog video signal .The analog video signal is mostly the source for a digital video, because the most common form of the video signal in use today

The full resolution of PAL video signal is 720*576=414,720 pixels for one picture . For true color 24 Bites per pixel are needed, so 720*576*24/8=1,244,160. bytes are needed for one picture. The PAL video signal contains 25 pictures per second. so we need 31,104,000 bytes per second digital video. This means we have bit rate of 248,832,000 Bytes per second or about 249Mbit/sec.

If you now compute how much space you need to store one 90 minute movie you will recognize why compression is needed:

90 Minutes contain 90*60=5400 seconds. One 90 minute movie needs 5400*31,104,000= 16,796,160,000 bytes or 156 G Byte.

This astronomical bit rate can not be handled by any computer system today. The logical solution to this problem is digital image compression.

2.3 Lossless And Lossy Compression

Compression techniques are classified into two categories lossless and lossy as shown in figure[2.1]. Lossless techniques are capable to recover the original data perfectly. These algorithms are used to archive computer data which have to recovered perfectly.

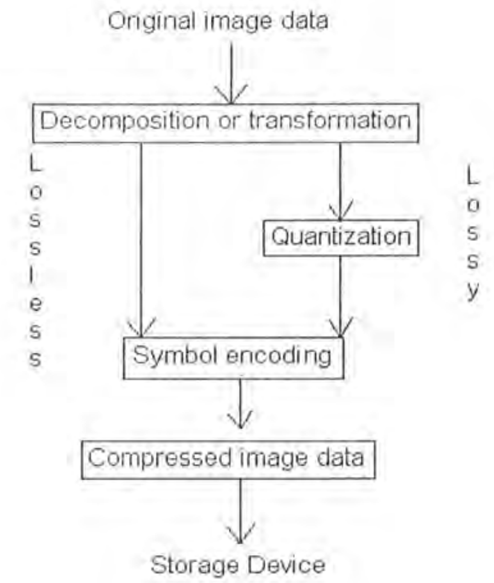

Figure 2.1: Basic elementt of lossey and lossless image compression

Lossy techniques involve algorithms which recover only similar data to the original one. The lossy techniques provide higher compression ratios, and therefore they are more often applied in the image compression than lossless techniques.

The lossy compression algorithm make use of the characteristic of the human eye, because a picture contains some information which is not necessary for the picture quality. The human visual system does not treat all the visual information with equal sensitivity . For example , the eye is more sensitive for changes in the luminance than in the chrominance. For this reason it suffies to transfer only one chrominance pixel U and V for four luminance pixel Y. This indicates that four adjacent pixels have the same color information but various brightness on the display screen. These make it possible to reduce the color information .

The human eye is also less sensitive to high frequencies. So it is a good idea to transfer the lower frequencies more exactly than the high frequencies or to clip the high frequencies. This reduction is not exactly reversible . But because of the human eye there are some possibilities to cut high frequencies and to reduce the color information without visible artifacts.

2.4 JPEG

The first standard for image compression was developed by the JPEG (Joint Photographic Experts Group) committee in the eighties.

The JPEG committee was introduced by the ISO / IEC (International Standard Organization / International Electro technical Commission). The JPEG compression algorithm is used for still image applications. The JPEG standard is targeted for full-color still frames, achieving 15:1 average compression. You can get a compression factor up to 30 nearly without visible quality difference. The JPEG standard provides four modes of operation:

1. *sequential DCT – based encoding* in which each image component is encoded in a single left-to-right and top-to bottom scan.
2. *progressive DCT – based encoding* in the image is encoded in multiple scan, in order to produce a quick rough decoded image when the transmission time is long. This encoding mode is often used in the World Wide Web to provide a quick transmitted picture which gets better and better.
3. *lossless encoding* in which the image is encoded to guarantee the exact reproduction. But the compression ration is only about 2:1
4. *hierarchical encoding* in which the image is encoded in multiple resolution.

The JPEG encoding is based on the transformation in the frequency domain using the Discrete Cosine Transform (DCT). The compression is made by a quantization in the frequency domain and this means that JPEG is a lossy compression technique.

2.4.1 Transformation From The Spatial Domain To The Frequency Domain

The standardized image compression techniques transform a spatial domain into the frequency domain. This transformation into the frequency domain exploits the spatial correlation of the pixels by converting them to a set of independent coefficients. This transformation is done on a block basis. The picture is divided mostly into 8*8 pixel blocks and then these blocks are transformed using a 2-D transformation into the frequency domain.

The video compression algorithms use the discrete cosine transform (DCT). The DCT offers the advantage in comparison with other transformations, that the coefficients are all in real domain. There are other transformations available, but in the JPEG and MPEG standard the transformation is done by the DCT.

The forward 2-D DCT and the inverse 2-D IDCT are defined as follows :

DCT

$$F(u,v) = \frac{1}{4} * C(u) * C(v) * \sum_{j}^{n}\sum_{k}^{n} f(j,k) * \left[\cos\frac{(2*j+1)*u*\pi}{16}\right] * \left[\cos\frac{(2*k+1)*u*\pi}{16}\right]$$

IDCT

$$F(J,k) = \frac{1}{4} \sum_{j}^{n}\sum_{k}^{n} C(u) * C(v) * \left[\cos\frac{(2*j+1*u*\pi}{16}\right] * \left[\cos\frac{(2*k+1*v*\pi}{16}\right]$$

with

$$C(w) = \{ \tfrac{1}{\sqrt{2}} \quad with \quad w = 0$$

The result of the DCT is an 8×8 matrix with the frequency coefficient. In the value at the upper left corner is the DC coefficient. The frequency increases to the right and to the bottom.

Normally a picture contain many low frequency components. Mostly the coefficients are concentrated in the upper left corner of a matrix.

After the transformation into the frequency domain the results of the DCT are quantized. The higher frequencies are more quantized than the lower frequencies, because the human eye is not very sensitive for the higher frequency components. The result of the quantization of higher frequencies is mostly zero. Because the high frequency coefficient are more quantized as the low ones.

2.5 Quantization

A quantizer is essentially a staircase function that maps the possible input values into a smaller number of output levels. In this way the number of symbols that need to be encoded are reduced at the expense of introducing error in the reconstructed image. The type and degree of quantization has a large impact on the final bit rate and the reconstructed picture quality of a lossy scheme. The individual quantization of each signal value is called scalar quantization (SQ), and the joint quantization of a block of signal value is called vector quantization (VQ). The selection of a quantizer is usually based on the minimization of some distortion measure for a given average output bit rate.

2.5.1 Scalar Quantization

As mentioned previously, scalar quantization (SQ) refers to the independent quantization of each signal value. The main advantage of SQ is its implementation simplicity and optimal performance in many situation.

2.5.2 Vector Quantization

The joint quantization of a block of signal value is called vector quantization (VQ). In VQ, an N-dimentional input vector X=$[x_{1,}x_{2,}......x_n]$, whose components represent the discrete or continuous signal value , is mapped into one of N possible reconstruction vectors Y_i, donated by d [X , Y_i] and is defined according to the application. The most common distortion measurement is mean-squared (MSE), given by

$$d_{mse}(X, Y) = \frac{1}{n} \sum (x_i - y_i)^2$$

The set Y is sometimes referred to as the reconstruction *codebook* and its members are called *codevectors* or *templates*. The *codebook design* problem is to find the optimal code book (in the sense of minimizing average distortion) for given input signal statistics, distortion measure, and code book size, N.

Hierarchical VQ

Compute the variance of an $(N \times N)$ block. If it is above a given threshold, divide it into four$(N/2 \times N/2)$ block. Compute the variances of the four $(N/2 \times N/2)$ blocks. If all or some of the variances of the $(N/2 \times N/2)$ blocks are above an other threshold, then divide each $(N/2 \times N/2)$ block (that is, those above the threshold) into four $(N/4 \times N/4)$ blocks. This is a variable block size and, hence, a variable vector dimension VQ reflecting the image activity. Separate code book for each block size are to be designed and, of course, stored at the encoder and decoder. Overhead by bits indicating the actual coding mode need to be transmitted. This adaptivity improves the coding efficiency at the cost of increased complexity.

2.6 Symbol Encoding

The last component in common compression algorithms is symbol encoding. i.e., mapping of output symbols (values) resulting from the decomposition and / or quantization stages into channel symbols. The mapping operation is also referred to as noiseless coding, lossless coding or data compaction coding. Symbol encoding may be as simple as using fixed-length binary code words to represent symbols, or it might us variable-length code words for better efficiency.

2.6.1 Fixed Length Codes

In some application it is desirable to use fixed length code words to minimize implementation complexity or to satisfy certain channel constraints, such as the need for a constant data rate. In fixed length coding, each source symbol is assigned a fixed length code, where code length depends upon the number of symbols the source can generate. The inefficiency of this type of coding results when the number of symbols is not a power of 2. The variable-length coding is better choice in such cases.

2.6.2 Variable - Length Codes

Variable-length codes are efficient than the fixed-length codes in terms of bit rate but they have complex implementation. Also , they are not suitable for applications requiring fixed bit rate .

2.6.3 Run Length And Huffman Coding

After the quantization the 8*8 matrix contains many zero coefficients particularly in the high-frequency part. These can be coded by the run-length coding efficiency. The run-length coding produce a couple of numbers which represent the following :

The run-length coding counts the number of zeros until a coefficient unequal zero occurs. Then one pair of numbers is generated, the first value is the count of zero coefficient and the second value is the value of the coefficient which is unequal zero

For example the run level code (5,20) represent the following 0 0 0 0 0 20.

For the run-length coding the 2-D 8*8 matrix has to be arranged to a 1-D sequence. This ordering is done by the so called Zig Zag scan way through the matrix. This Zig Zag scan way orders the quantized DCT coefficient in ascending spatial frequencies from the dc coefficient to the highest frequency coefficient. The frequency increases from the left to the right of the result matrix and also from the top to the bottom. The Zig Zag scan way is shown in Figure 1. The run-length coding is a lossless compression.

After the run-length coding each run-length couple is coded by an entropy coding. The entropy coding is also known as Huffman or variable length coding.

The appearance probability P_s of one single run-length couple is investigated. The run-length couple with the largest appearance probability is coded with a shorter bit code then run-length couple with appearance probability. The optimum code length for a symbol, Ls is given by [11]:

$$Ls = \log_2(1/Ps)$$

The entropy, which is simply the average number of bits per symbol, is given by [11]:

$$\text{Entropy} = \sum P_s \log_2(1/P_s)$$

The Huffman codes are constructed by pairing two symbols with the lowest probability combining them to a branch of a tree. The branches of the tree are assigned a1 and a0 bit. The tree is developed until every symbol is covered by a branch of a tree. The Huffman codes are constructed by containing the bits of the branches, starting from the root and going back to the symbol to code.

The following example shows the results of the transformation to the frequency domain, the quantization and the run-length coding.

* * * * * * * * * * 8*8 pixelblock: * * * * * * * * * *

$$\begin{bmatrix} 130 & 125 & 133 & 136 & 139 & 149 & 135 & 137 \\ 119 & 132 & 150 & 150 & 135 & 128 & 124 & 122 \\ 135 & 136 & 127 & 120 & 122 & 117 & 133 & 137 \\ 88 & 106 & 133 & 138 & 140 & 134 & 126 & 104 \\ 142 & 151 & 142 & 134 & 116 & 120 & 125 & 140 \\ 120 & 113 & 118 & 148 & 165 & 149 & 147 & 130 \\ 129 & 139 & 141 & 127 & 124 & 120 & 129 & 150 \\ 132 & 126 & 122 & 121 & 134 & 147 & 157 & 149 \end{bmatrix}$$

Quantization Results:

$$\begin{bmatrix} 132 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Results of Run-length Coding:

(0, 132) (0, -1) (0, -1) (0, 1) (0, 1) (0, -1) (1, -1) (0, -1)

(0, 1) (2, 1) (7, -1) (13, -1) (1, 1) (10, 1) (0, 2)

These run-length couples are coded using the Huffman coding. In this example a Huffman code is developed for these Run-length couples.

Run − level code	*Count*	*Probability(P)*	*Code*
$0, -1$	4	0.266	1
$0, 1$	3	0.200	01
$0, 132$	1	0.066	00111
$0, 2$	1	0.066	00110
$1, 1$	1	0.066	00101
$1, -1$	1	0.066	00100
$2, 1$	1	0.066	00011
$7, -1$	1	0.066	00010
$10, 1$	1	0.066	00001
$13, -1$	1	0.066	00000

• Table : Probability of the run-length couples

The Huffman code can be developed with the following tree shown in figure 2.2.

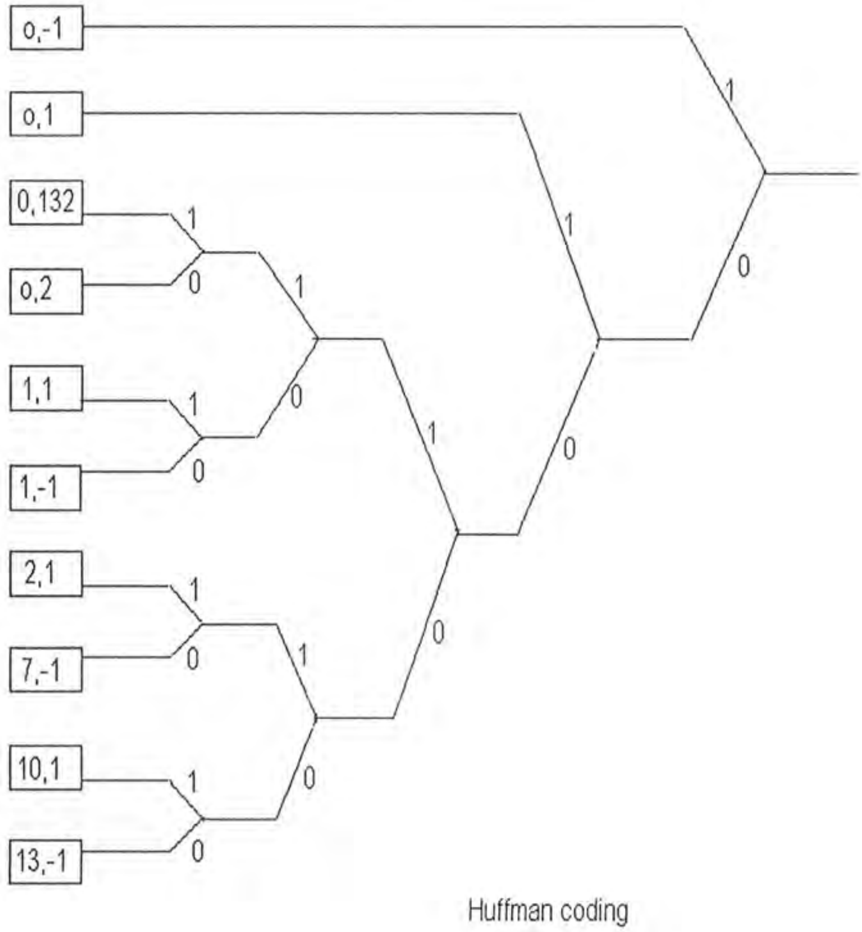

Figure 2.2:

The run-length couples are coded as follows:

00111 1 1 01 01 00100 1 01 0011 00010 00000 00100 00001 00110 (50 Bit)

2.7 Wavelet based image coding

There are many techniques for image coding but presently subband coding is the successful one. DCT based transform coding was popular in 1980's due to less complexity and effective bit allocation, so became the JPEG standard in image coding. JPEG image suffers from blocking artifacts. Wavelet based subband coding avoids blocking at medium bit rate, because its basis function have variable length. Wavelet based coder transforms the image into subimages using two channel filter bank.

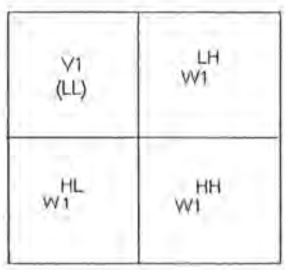

figure 2.3: Image decomposition : first stage of the 2-D dyadic DWT

The upper left subimage is obtained by low pass filtering in both horizontal and vertical directions, represented by LL as in figure 2.3. The other subimages have high frequencies. The algorithm will assign few bits to HH and many bits to LL. The LL image is 4 to 5 time is iterated. These subbands are quantized and then scanned. These scanned coefficients are coded using lossless compression and then transmitted or stored. We will study in details of image compression using wavelet transform in chapter 4.

Chapter 3

Wavelet Analysis

In Fourier analysis frequency information can only be extracted for the complete duration of signal f(t). Since the integral in the Fourier transform extends over all the time therefore the information rises over whole the length of signal. If there is a local oscillation at some point in f(t), will contribute to the calculated Fourier transform F(w). But its location in the time domain will be lost.

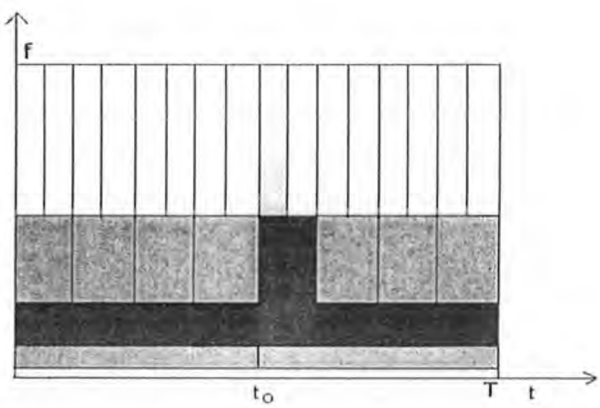

Figure 3.1: Discrete time wavelet series.

This disadvantage is overcome in wavelet analysis because the wavelet transform has shorter basis functions for the higher frequencies and longer basis function for the lower frequencies as shown in figure 3.1, which provides an alternative way of breaking a signal down into its constituent parts.

3.1 Dilation Equations

Now we will examine how the wavelets are generated from dilation equations. The basic function of $\phi(x)$ is a dilated version of $\phi(2x)$. In dilation equation $\phi(x)$ is expressed as a finite series of terms involving $\phi(2x)$. Each of these $\phi(2x)$ terms in positioned at a different place and different argument on the horizontal axis. The basic dilution equation has a form

$$\phi(x) = \sum_k c_k \phi(2x - k) \tag{3.1}$$

where c's are numerical constants.

It is very difficult to solve equation 1 directly to find out $\phi(x)$ therefore we have to construct $\phi(x)$ indirectly. The simplest approach is to set up an iterative algorithm in which each new approximation is calculated by the previous one.

$$\phi_j(x) = C_0\phi_{j-1}(2x) + C_1\phi_{j-1}(2x-1) + C_2\phi_{j-1}(2x-2) + C_3\phi_{j-1}(2x-3) \tag{3.2}$$

the process of iteration is continued until $\phi_j(x)$ and $\phi_{j-1}(x)$ becomes almost same (indistinguishable). Consider a box function $\phi_0 = 1$ $0 \leq x \leq 1$. The interval x = 0 to 1 has developed a stairless function over the interval x = 0 to 2. The added contribution is shown in this figure [3.2]. A particular set of co-efficient is used as defined below

$$\begin{aligned} C_0 &= (1+\sqrt{3})\ /\ 4\ , \qquad C_1 = (3+\sqrt{3})\ /\ 4 \\ C_2 &= (3-\sqrt{3})\ /\ 4\ , \qquad C_3 = -\ (\sqrt{3}-1)\ /\ 4\ , \end{aligned}$$

these co-efficient generates (see later) D_4 wavelet. The D stands for Daubechies who first discovered their properties [10].

If the iterative process is continued, the function $\phi(x)$ approaches to limiting shape as shown in figure (3.2) and is discontinuous in nature. By magnifying the figure, we observed the graph has a fractal nature and its irregular out line remain always there. We can get smoother function by adding more terms in the dilation equation (1). This function $\phi(x)$ is called scaling function and the corresponding wavelet function will be constructed in the next section.

The scaling function $\phi(x)$ generated by iteration follows this matrix scheme

$$[\phi_2] = \begin{bmatrix} C_0 & & & \\ C_1 & & & \\ C_2 & C_0 & & \\ C_3 & C_1 & & \\ & C_2 & C_0 & \\ & C_3 & C_1 & \\ & & C_2 & C_0 \\ & & C_3 & C_1 \\ & & & C_2 \\ & & & C_3 \end{bmatrix} \begin{bmatrix} C_o \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} [1] \; = M_2 M_1 \; [1]$$

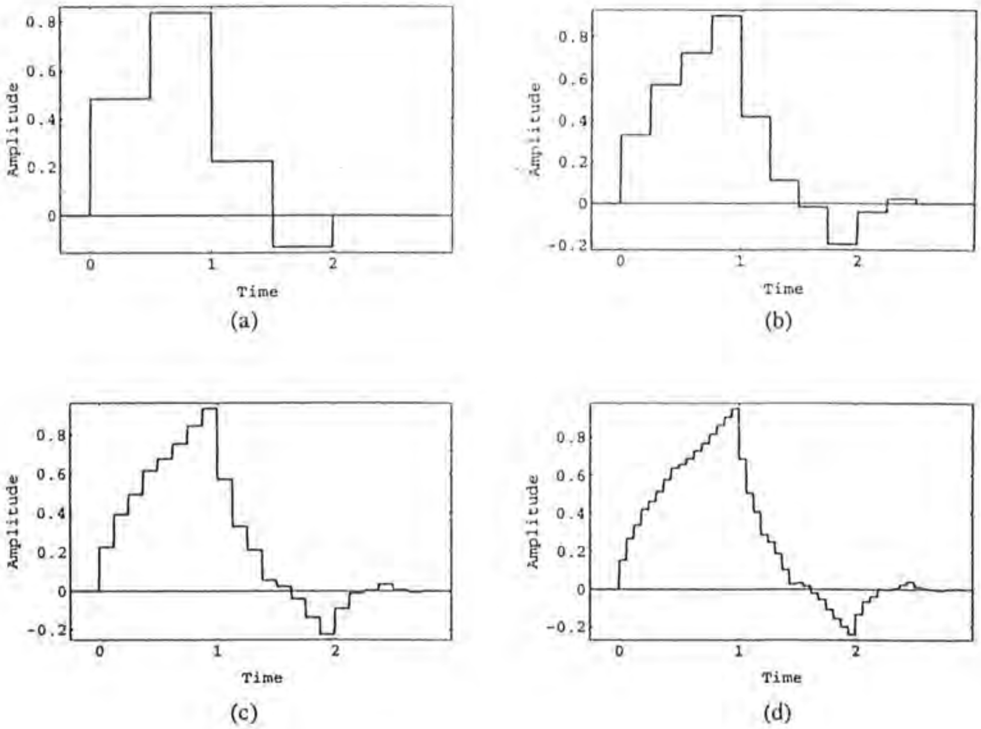

Figure 3.2: (a) $\phi^{(1)}(t)$.(b) $\phi^{(2)}(t)$.(c) $\phi^{(3)}(t)$.(d) $\phi^{(4)}(t)$. Generated by iteration

where M_r denotes a matrix of order $(2^{r+1} + 2^r - 2) \times (2^r + 2^{r-1} - 2)$ in which each column has a submatrix of coefficients C0 C_1 , C_2 , C_3 , positioned two places below of its left submatrix. As the iteration increases, the number points increases on the graph in sequence, 1, 4, 10, 22, 46, $\cdots$ $2^{r+1} + 2$ so that after eight iteration it reaches $2^9 + 2^8 - 2 = 766$ with each point spaced $1 / 2^8 = 1 / 256$ unit apart. This is not most efficient method but it is simple.

3.2 Dilation Wavelet

Wavelet function is derived from the corresponding scaling function with same co-efficient but in reverse order and with terms having their sign changed.

$$\psi(x) = \sum_{k=0}^{N-1} (-1)^k \, c_k \, \phi(2x + k - N + 1) \tag{3.3}$$

where k is positive integer and N is the total number of co-efficient. Like scaling function, it also retains the discontinuous and fractal nature but have a surprising shape for the bases function. Suppose we want to generate $\psi(x)$ (wavelet function) from $\phi(x)$ just after single iteration then it is generated by the following matrix scheme.

$$[\psi_2] = \begin{bmatrix} C_0 & & & \\ C_1 & & & \\ C_2 & C_0 & & \\ C_3 & C_1 & & \\ & C_2 & C_0 & \\ & C_3 & C_1 & \\ & & C_2 & C_0 \\ & & C_3 & C_1 \\ & & & C_2 \\ & & & C_3 \end{bmatrix} \begin{bmatrix} -C_3 \\ C_2 \\ -C_1 \\ C_0 \end{bmatrix} [1] = M_2 M_1 \; [1] \tag{3.4}$$

similarly

$$[\psi_3] = M_3 M_2 M_1 \; [1]$$

and so on where M_3 is matrix of order 22×10

3.3 Condition for Wavelet co-efficient

A good set of co-efficient must satisfy the following conditions.

i $\sum_{k=0}^{N-1} C_k = 2$

ii $\sum_{k=0}^{N-1} (-1)^k \, k^m$

where m = 0, 1, 2, $\cdots\cdots$N / 2 -1

iii $\sum_{k=0}^{N-1} C_k \, C_{k+2m} = 0 \qquad m \neq 0$

iv $\sum_{k=0}^{N-1} C_k^2 = 2$

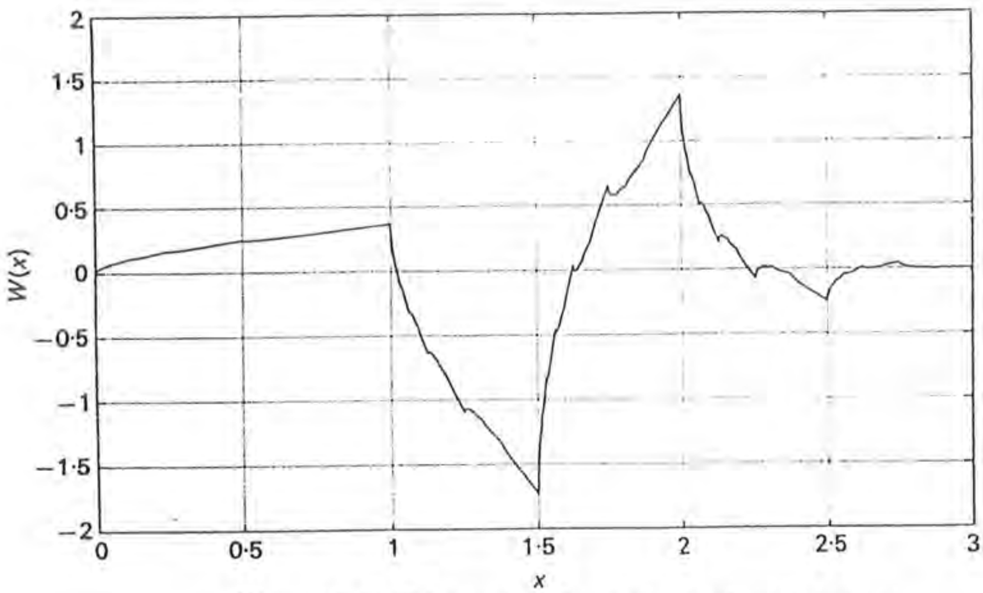

Figure 3.3: D4 wavelet from the scaling function

3.4 Discrete wavelet transforms

The DWT algorithm was discovered by Mallat [12] and is called Mallat's pyramid algorithm or sometimes Millat' tree algorithm. We shall approach the algorithm by considering first its inverse. Suppose that the DWT has been computed to generate the sequence

$$a = [a_0\ a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ .\ .\ .\ a_{2j+k}\ .\ .\ .]$$

Suppose, for example, that we consider an expansion with the primary scaling function $\phi(x)$ and wavelet of scale 0, 1 and 2. Then a will have $(1 + 1 + 2 + 4) = 2^3$ terms. In order to include all the wavelets at any particular scale, the total number of terms in the transform must always be a power of 2. Consider the case when there are only eight terms so that

$$a = [a_0\ a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7] \tag{3.5}$$

The first element a_0 is the amplitude of the scaling function term $\phi(x)$. Since $\phi(x)$ can be generated by iteration from a unit box over the integral $0 \leq \mathrm{x} < 1$ (see figure),

$a_0\phi(x)$ can be generated by iteration starting from a box of height a_0. Suppose we chosen a wavelet with four coefficients. Then the first step in the iteration is from (3.4)

$$\phi_1 = \begin{bmatrix} C_o \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} [a_o]$$

The initial box function occupied from interval $0 \leq x < 1$, but we see in figure (3.1) that the first iteration extends over $0 \leq x < 2$. If the part that lies outside the interval is wrapped round to fall back into the unit interval, we get

$$\phi_1 = \begin{bmatrix} C_o + C_2 \\ C_1 + C_2 \end{bmatrix} [a_o]$$

On taking the second iterative step, without including wrap-around, we have, from (3.4),

$$\phi_2 = \begin{bmatrix} C_o & & & \\ C_1 & & & \\ & C_2 & C_o & & \\ & C_3 & C_1 & & \\ & & C2 & C_o & \\ & & C_3 & C_1 & \\ & & & C_2 & C_o \\ & & & C_3 & C_1 \\ & & & & C_2 \\ & & & & C_3 \end{bmatrix} \begin{bmatrix} C_o \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} [a_o] \qquad (3.6)$$

Allowing for wrap-around and adding terms at the same position in $0 \leq x < 1$, we can check by multiplication that this is the same as

$$\phi_2 = \begin{bmatrix} C_o & C_2 \\ C_1 & C_3 \\ C_2 & C_0 \\ C_3 & C_1 \end{bmatrix} \begin{bmatrix} C_o + C_2 \\ C_1 + C_3 \end{bmatrix} [a_o] \tag{3.7}$$

Notice how the left-hand matrix is formed, by taking a submatrix of order 4 × 2 from the left-hand matrix in (3.6) and then transposing the C_2 and C_3. This recipe also applies for the matrix in (3.5) which is formed by transposing the same two elements

$$\begin{bmatrix} C_0 \\ C_1 \end{bmatrix}$$

For the third iterative step, the calculation is

$$\phi_3 = \begin{bmatrix} C_o & & & C_2 \\ C_1 & & & C_3 \\ C_2 & C_o & & \\ C_3 & C_1 & & \\ & C_2 & C_o & \\ & C_3 & C_1 & \\ & & C_2 & C_o \\ & & C_3 & C_1 \end{bmatrix} \begin{bmatrix} C_0 & C_2 \\ C_1 & C_3 \\ C_2 & C_0 \\ C_3 & C_1 \end{bmatrix} \begin{bmatrix} C_0 + C_2 \\ C_1 + C_3 \end{bmatrix} [a_o] \tag{3.8}$$

and this generates the eight ordinates in the interval $0 \leq x < 1$ for the wrap-around scaling function. If we revise the definitions of the M matrices of the wavelet coefficients so that

$$M_1 = \begin{bmatrix} C_o + C_2 \\ C_1 + C_3 \end{bmatrix} \qquad \text{of order } 2 \times 1 \tag{3.9}$$

$$M_2 = \begin{bmatrix} C_o & C_2 \\ C_1 & C_3 \\ C_2 & C_0 \\ C_3 & C_1 \end{bmatrix} \qquad \text{of order } 2^2 \times 2 \tag{3.10}$$

$$M_3 = \begin{bmatrix} C_o & & & C_2 \\ C_1 & & & C_3 \\ C_2 & C_o & & \\ C_3 & C_1 & & \\ & C_2 & C_o & \\ & C_3 & C_1 & \\ & & C_2 & C_o \\ & & C_3 & C_1 \end{bmatrix} \quad \text{of order } 2^3 \times 2^2 \tag{3.11}$$

then the algorithm for generating the contribution of $a_0\phi(x)$ to f(x) is

$$f^{\phi}(x) = M_3\ M_2\ M_1\ a_0 \tag{3.12}$$

or, in diagrammatic form,

$$a_0 = f^{\phi}(1) \xrightarrow{M_1} f^{\phi}(1:2) \xrightarrow{M_2} f^{\phi}(1:4) \xrightarrow{M_3} f^{\phi}(1:8) \tag{3.13}$$

where $f^{\phi}(1:8)$ means an array of eight elements that represents the contribution of $a_0\phi(x)$ to f(x) at x = 0, 1/8, 1/4, ..., 7/8.

Returning to the sequence (2), consider the second term, a_1. This is the amplitude of the wavelet function W(x) which is generated from a unit box by iteration as shown in figure (3.5). The matrix operations for doing this are the same as for generating the scaling function $\phi(x)$ except that the first step involves replacing

$$\begin{bmatrix} C_o \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} \quad by \quad \begin{bmatrix} -C_3 \\ C_2 \\ -C_1 \\ C_0 \end{bmatrix} \tag{3.14}$$

according to (3.4). The procedure for allowing for wrap-around is exactly the same as for the scaling function.

Defining

$$G_1 = \begin{bmatrix} -C_3 - C_1 \\ C_2 + C_0 \end{bmatrix} \tag{3.15}$$

The algorithm for generating the contribution of $a_1 W(x)$ to f(x) is

$$f^{(0)}(1:8) = M_3\ M_2\ G_1\ a_1 \tag{3.16}$$

or in diagrammatic form,

$$a_1 = f^{(0)}(1) \xrightarrow{G_1} f^{(0)}(1:2) \xrightarrow{M_2} f^{(0)}(1:4) \xrightarrow{M_3} f^{(0)}(1:8) \tag{3.17}$$

The third term in (3.5), a_2, is the amplitude of W(2x). Instead of the second iteration being reached by

$$\begin{bmatrix} C_o & C_2 \\ C_1 & C_3 \\ C_2 & C_0 \\ C_3 & C_1 \end{bmatrix} \begin{bmatrix} -C_3 - C_1 \\ C_2 + C_0 \end{bmatrix} [a_2] \tag{3.18}$$

the first operation is omitted and we go straight to

$$\begin{bmatrix} -C_3 \\ C_2 \\ -C_1 \\ C_0 \end{bmatrix} [a_2] \tag{3.19}$$

Thereafter the iteration proceeds as before to get

$$f^{(1,1)}(1:8) = M_3 \begin{bmatrix} -C_3 \\ C_2 \\ -C_1 \\ C_0 \end{bmatrix} [a_2] \tag{3.20}$$

The fourth term in (3.5), a_3, is the amplitude of the translated wavelet W(2x - 1). Allowing for wrap-around, the procedure for computing the contribution that this makes to f(x) is exactly the same as for a_2 except that the elements in the first matrix are arranged in the order

$$\begin{bmatrix} -C_1 \\ C_0 \\ -C_3 \\ C_2 \end{bmatrix} \tag{3.21}$$

so that (3.20) becomes

$$f^{(1,2)}(1:8) = M_3 \begin{bmatrix} -C_1 \\ C_0 \\ -C_3 \\ C_2 \end{bmatrix} [a_3] \tag{3.22}$$

Combining (3.20) and (3.22) then gives

$$f^{(1,2)}(1:8) = M_3 \begin{bmatrix} -C_3 & -C_1 \\ C_2 & C_0 \\ -C_1 & -C_3 \\ C_0 & C_2 \end{bmatrix} \begin{bmatrix} a_2 \\ a_3 \end{bmatrix} \tag{3.23}$$

or putting

$$G_2 = \begin{bmatrix} -C_3 & \\ C_2 & \\ -C_1 & -C_3 \\ C_0 & C_2 \end{bmatrix} = \begin{bmatrix} -C_3 & -C_1 \\ C_2 & C_0 \\ -C_1 & -C_3 \\ C_0 & C_2 \end{bmatrix}$$

we get

$$f^{(1)}(1:8) = M_3\ G_2 \begin{bmatrix} a_2 \\ a_3 \end{bmatrix} \tag{3.24}$$

and, in a diagram,

$$\begin{bmatrix} a_2 \\ a_3 \end{bmatrix} \overset{G_2}{\to} f^{(1)}(1:4) \overset{M_3}{\to} f^{(1)}(1:8) \tag{3.25}$$

The remaining four elements of (3.5), a_4, a_5, a_6, a_7 are the amplitudes of wavelets $\Psi(4x)$, $\Psi(4x-1)$, $\Psi(4x-2)$, $\Psi(4x-3)$. Each wavelet has the elements $[-c_3 \quad c_2 \quad -c_1 \quad c_0]^t$

and so the single stage of calculation is

$$f^{(2)}(1:8)=\begin{bmatrix} -C_3 & & & -C_1 \\ C_2 & & & C_0 \\ -C_1 & -C_3 & & \\ C_0 & C_2 & & \\ & -C_1 & -C_3 & \\ & C_0 & C_2 & \\ & & -C_1 & -C_3 \\ & & C_0 & C_2 \end{bmatrix}\begin{bmatrix} a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} \tag{3.26}$$

or, diagrammatically,

$$\begin{bmatrix} a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} \xrightarrow{G_3} f^{(2)}(1:8) \tag{3.27}$$

combining the results we get

$$f(1:8)=f^{(\phi)}(1:8)+f^{(0)}(1:8)+f^{(1)}(1:8)+f^{(2)}(1:8) \tag{3.28}$$

we have the final diagram below:

$$f'(1) \xrightarrow{M_1} f'(1:2) \xrightarrow{M_2} f'(1:4) \xrightarrow{M_3} f'(1:8)=f \tag{3.29}$$

$$\uparrow \quad G_1 \uparrow \qquad G_2 \uparrow \qquad G_3 \uparrow$$

$$a=[a(1) \qquad a(2) \qquad a(3:4) \qquad a(5:8)]$$

where a(1) = a_0, a(2) = a_1, a(3,4) = $[a_2\ a_3]^t$ a(5,8) = $[\ a_4\ a_5\ a_6\ a_7]^t$, This is the inverse of Mallat's tree algorithm [1.2]. Now consider how to break down an arbitrary function $f(1:2^n)$ into its wavelet transform $a(1:2^n)$. The matrix M and G follow the orthogonality conditions, so we have

$$\frac{1}{2}\, M_r^t\, M_r = I$$

$$M_r^t\ G_r = 0$$

$$G_r^t\ M_r = 0$$

$$\frac{1}{2}\ G_r^t\ G_r = I$$

Reversing the tree and taking transpose of M and G we get the Mallat's tree algorithm.

We can decompose and reconstruct a discrete function in the same by defining a filter L and H (low pass and high pass filter).

3.4.1 The filter matrices L & H

For a given vector f(x) = [f_1, f_2, f_3, ... f_n] where n = 2^j they may be equally spaced values of a function f(x) on a unit interval. The goal is to decompose this vector at different scales. At each new level the entries cut into half. The decomposition is

$$f = f^{\phi} + f^{(0)} + \cdot\cdot\cdot\cdot\cdot f^{j-1}$$

the details f^j is the combination of 2^j wavelets and f^{ϕ} is the multiple of the scaling function ϕ.

The matrix L is the fine to coarse filter and it produces a vector with half as many entries. The entries (L_{ij} = C_{2i-j}) of this filter are the recursion coefficient for the scaling function. Rows 1, 2 and columns -1, 0, 1, 2 are displayed with N = 3

$$L = \frac{1}{2}\begin{bmatrix} C_3 & C_2 & C_1 & C_0 & & \\ & & C_3 & C_2 & C_1 & C_0 \end{bmatrix}$$

The beautiful thing is that the high pass filter H uses the same coefficient and in the same way this filter is associated with wavelets Ψ just as L is associated with the scaling function the filter H is defined as follow

$$H_{ij} = (-1)^{j+1}\ C_{j+1-2i}$$

for i = 1, 2 and j = 1, 2....4

$$H = \frac{1}{2}\begin{bmatrix} C_0 & -C_1 & C_2 & -C_3 & & \\ & & C_0 & -C_1 & C_2 & -C_3 \end{bmatrix}$$

These indices are used to match the Haar different are also possible. The important points are

$$H \quad L^* = 0$$

$$L\ L^* = I \qquad \& \quad H\ H^* = I$$

$$and \qquad L^*\ L + H^*\ H = I$$

3.4.2 Decomposition

The decomposition of a discrete function f in to Mallat's Pyramid algorithm is as follow starting from

$$a^j = f \qquad for \quad i = J, \cdots\cdots 1$$

$$a^{j-1} = La^j \qquad and \qquad b^{j-1} = Ha^j$$

the full decomposition is represented by a tree of filters

$$a^J \xrightarrow{L} a^{J-1} \xrightarrow{L} a^{J-2} \cdots\cdots \xrightarrow{L} a^0$$

$$\searrow b^{J-1} \searrow b^{J-2} \cdots\cdots \searrow b^0$$

3.4.3 Reconstruction

The decomposition of discrete function at different scale can be recovered by starting from a^0 and $b^0, \cdots\cdots b^{J-1}$ for j = 1, ...J

$$a^j = La^{j-1} + H^* b^{j-1}$$

The reconstruction goes from the branches of the tree back to the root

$$a^0 \xrightarrow{L^*} a_1 \xrightarrow{L^*} a^2 \cdots\cdots \xrightarrow{L^*} a^J = f$$

$$b^0 \nearrow_{H^*} b_1 \nearrow_{H^*} \cdots \nearrow_{H^*}$$

3.5 Wavelet and Multiresolution Analysis

Multiresolution will be described first for subspaces V_j and W_j. The scaling spaces V_j are increasing. The wavelet space W_j is the difference between V_j and V_{j+1}. The sum of V_j and W_j is V_{j+1}. Then these extra conditions involving dilation to 2_t and translation to $t-k$ define a genuine multiresolution:

If $f\ (t)$ is in V_j then $f\ (t)$ and $f\ (2t)$ and all $f\ (t-k)$ and $f\ (2t-k)$ are in V_{j+1}.

In the end, one wavelet generates a whole basis. The functions $w(2^j\, t-k)$ come by dilation and translation (all j and all k). There are six steps toward this goal, and we take them one at a time:

1. An increasing sequence of subspaces V_j (complete in L^2) .
2. The wavelet subspace W_j that gives $V_j + W_j = V_{j+1}$
3. The dilation requirement from $f\ (t)$ in V_j to $f\ (2t)$ in V_{j+1}
4. The basis $\phi\ (t-k)$ for V_0 and $w(t-k)$ for W_0
5. The basis $\phi\ (2^j t-k)$ for V_j and $w(2^j t-k)$ for W_j
6. The basis of all wavelets $w(2^j t-k)$ for the whole space L^2

3.5.1 A Scale Of Subspaces

Each V_j is contained in the next subspace V_{j+1}. A function in one subspace is in all the higher (finer) subspaces:

$$V_0 \subset V_1 \ldots \subset V_j \subset V_{j+1} \subset \ldots.$$

A function $f\ (t)$ in the whole space has a piece in each subspace. Those pieces contain more and more of the full information in $f\ (t)$. The piece in $V_j$is $f_j\ (t)$. One requirement on the sequence of subspaces is completeness.

$$f_j\ (t) \rightarrow f\ (t) \qquad \text{as} \quad j \rightarrow \infty$$

3.5.2 The Dilation Requirement

So for we have an increasing and complete scalar of spaces. Each is V_j contained in the next V_{j+1}.For multiresolution, the crucial word *scale* carries an additional meaning. V_{j+1} consists of all re scaled functions in V_{j+1}

$$Dilation: \ f\ (t) \text{ is in } V_j \iff f\ (2t) \text{ is in } V_{j+1}.$$

In addition to completeness as $j \rightarrow \infty$, we require emptiness as $j \rightarrow -\infty$:

$$\cap\, V_j = \{0\} \quad \text{and} \quad \cup\, V_j = \text{whole space.}$$

Emptiness means that $\parallel f_j(t) \parallel \rightarrow 0$ as $j \rightarrow -\infty$.Completeness means that $f_j(t) \rightarrow f(t)$ as $j \rightarrow \infty$.The detail:

$\Delta\ f_j = f_{j+1} - f_j$ belongs to W_j and we still have

$$V_j \otimes\ W_j\ =\ V_{j+1}$$

This can be orthogonal sum, with Δf_j orthogonal to f_j. It must be a direct sum, with $V_j \cap W_j = \{0\}$. The construction of $f(t)$ from its details Δf_j can start at $j = 0$ as before, or it can start at $j = -\infty$:

3.5.3 The Translation Requirement And The Basis

Instead of rescaling $f(t)$,we now shift its graph. This is *translation*, and it leads to the fundamental requirement of time-invariance in signal processing. The subspaces are $shift - invariant$:

$$If\ f_j(t)\ is\ in\ V_j\ then\ so\ are\ its\ translates\ f_j(t-k).$$

Suppose $f(t)$ is in V_0.Then $f(2t)$ is in V_1 and so is $f(2t-k)$. By induction, $f(2^j t)$ is in V_j and so is $f(2^j t - k)$.Dilation and translation are now built in.

Chapter 4

Image Compression Using Wavelet Transform

The main obstacle for many application of digital images, i.e,acquisition, data storage, printing, and display, is the huge amount of data required to represent an image directly. Such an image needs to be compressed for storage or transmissions. The actual compression ratio can vary from 100:1 to 2:1 depending on the specific application and encoder/decoder complexity. State-of-the art techniques can compressed typical images by a factor of 10 to 50 without significantly effecting the image quality, depending on the technique applied. There are many techniques for image coding, we use the most successful, i.e,wavelet based subband image coding that avoid blocking artifact (which is the main disadvantage of JPEG standard) at medium bit rate, because the variable length of its basis functions.

4.1 Subband Coding

Subband coding (SBC) is another form of frequency decomposition. In SBC, a signal is decomposed into a number of equal- or unequal-frequency bands using filter banks that have been developed recently [13, 14, 15,]. In fact by using perfect reconstruction filter banks [5], the original signal going through the frequency decomposition-subsampling-interpolation-synthesis process can be fully recovered. The philosophy behind SBC is that coding techniques compatible with the frequency bands can be applied. signal components in high-frequency bands can be either dropped out or coarsely quantized.

Subband coding is also ideally suited for progressive image transmission (PIT) as bits related to lower bands can be transmitted first, followed by those related to the upper bands.

4.2 Subband Image Coding

A popular approach to subband image coding [16] is to map the image into four equal subbands in the 2D frequency domain as shown in figure 4.1. In general, a transform (such as DCT) is applied to the lowest subband, followed by quantization and VLC. The remaining subbands are coarsely quantized.

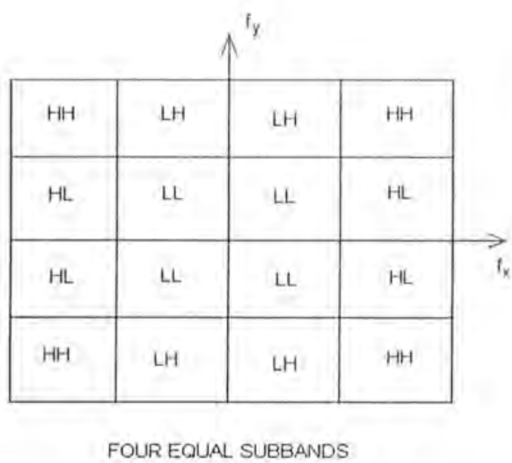

Figure 4.1: Mapping of an image into four equal sbbands in the 2-D frequency domain (L : low frequency, H: high frequency).

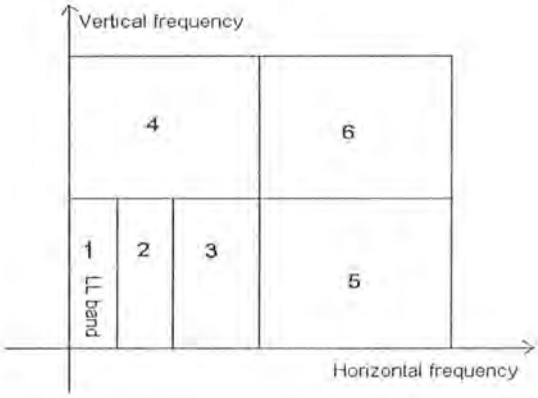

Figure 4.2: Unequal subband decomposition proposed by Bellcore for the ATM / SCONET / H-4-ISDN HDTV project [17]

4.3 DWT And Subband Coding

The DWT is known to be generated by a cascade of filter banks and the DWT is essentially the well-known subband decomposition [18]. The advantages of the DWT comes from the trade-off between spatial and frequency resolution, as the DWT has shorter basis functions for the higher frequencies as shown in the fig. Therefore, DWT has a versatile time-frequency localization due to a pyramid like multiresolution decomposition.

The 2-D wavelet transform is implemented independently, first in the horizontal direction and then in the vertical direction [12]. This method of applying the DWT to a two dimensional signal that is sampled on a rectangular lattice is called *the dyadic* 2-D DWT. The dyadic 2-D DWT decomposes the original image into four subbands, as shown in figure 4.3(a).

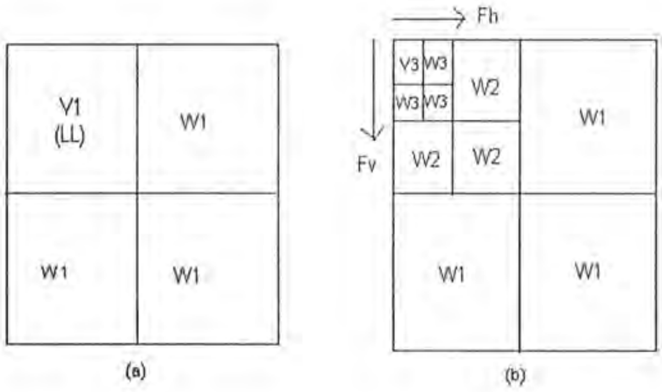

Figure 4.3: Image decomposition (a) first stage of the 2-D dyadic DWT (b) third stage of the 2-D dyadic DWT. V and W mean subspaces and difference subspaces, respectively.

One of the subband contains all the low-pass information (LL) and another all the high-pass information (HH). The other two subbands are a horizontal high-pass band containing vertical low-pass information (LH) and a vertical high-pass containing horizontal low-pass information (HL). For a full DWT the decomposition is repeated several times on the LL part [19]. Fig.4.3(b) shows the decomposition of the image after three iterations. There are a total of ten sub-images, which are used to encode the original image. V and W were defined as subspaces and spaces, respectively. This decomposition provides sub-images corresponding to different resolution levels and orientations.

Figure 4.4 shows the analysis section of a two-dimensional (2D) separable filter bank, where the first image rows are passed through the 2-channel filter bank and then, the columns are processed. The analysis section can be viewed as a 2×2 transform applied to the image (note that each subband has one fourth of the samples in the original signal). Also, the synthesis section can be viewed as a 2×2 inverse transform. Note that only the low-pass subband is connected to next stage transform. The inverse transform is, of course, accomplished by reversing the paths and the transforms.

To reconstruct the image from the subbands, one may choose the set of low-pass and high-pass filters shown in figure 4.4, so as to provide perfect reconstruction.

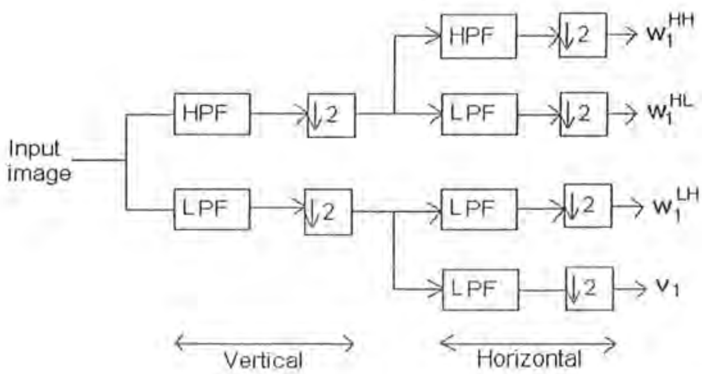

Figure 4.4: One step (first level) of a 2-D wavelet decomposition.

4.3.1 Entropy Coding:

After bit allocation and quantization, we have subimages with discrete levels represented by integers. How do we store or transmit these subimages? Many high pass coefficients are zero after quantization. These coefficients should be grouped so that the entropy coder can take full advantage of long strings of zeros. This is accomplished by *scanning*.

Run-length coding or Huffman coding or a combination should be used to reduce the redundancy of the images. We will discuss the baseline entropy coder which is a combination. JPEG also uses the baseline coding method.

4.3.2 Scanning Of The Discrete Wavelet Coefficients:

To demonstrate scanning, consider the three level wavelet transform in Figure 4.5. Subband 2,5 and 8 are highly correlated since 2 is the coarse approximation of 5 and 5 is the coarse approximation of 8. Suppose the pixel at the upper left corner of subband 2 is zero. *Then it is very likely that the pixels in a* 2×2 *shaded square of subband* 5 *are zero.* Similarly, the pixels in a 4×4 shaded square of subband 8 are probably zero. One can these group pixels into an "AC sequence" of length $21(=1+4+16)$ by vertically scanning the shaded squares. Figure also shows the scanning patterns of subbands 3, 6 and 9 (horizontal) and for subbands 4, 7 and 10 (diagonal). When the original image has size 32, the 16 pixels in each subband2, 3, and 4 give 48 AC sequences. The low frequency band is scanned horizontally and grouped into the DC sequence of length 16. This scanning method is similar to the *zero-tree coder* proposed by [Shapiro].

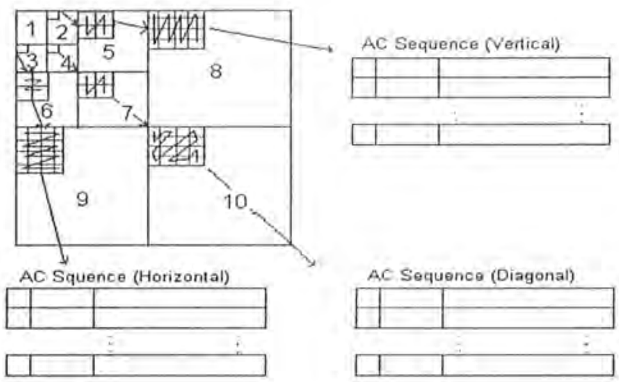

Figure 4.5: Scaning method used in the discrete wavelet decomposition.

4.3.3 Sequential Baseline Coding:

After scanning the quantized subimages, we have a set of DC and AC sequences to be stored. The baseline coder takes advantage of the correlation in the AC sequences. This algorithm combines Sequential Baseline Coding and Huffman entropy coding. The basic principle is similar to the JPEG coder, but does not restrict to the DCT.

Coding Of The AC Sequence:

The sequence 9 0 0 2 0 1 0 0 0 - 3 0 0 3 0 - 1 - 1 has strings of zeros interlacing with nonzero's. An efficient representation remembers the number of zeros before each nonzero. *Symbol-1* is the pair *(runlength, size)* where *runlength* specifies the number of preceding zeros. *Size* determines the number of bits to encode the current nonzero. *Symbol-2* gives the *(amplitude)* of the nonzero: *Size* $= n$ corresponds to *amplitude* less than 2^n(but not less than 2^{n-1}).

This representation of the example gives a string of *Symbol-1* and Symbol-2:

(0, 1)1 (2, 2)2 (1, 1)1 (3, 2)-3 (2, 2)3 (1, 1)-1 (0, 1)-1 (0, 0):

Note the terminal *symbol-1* (0, 0) at the end. Also (1, 1) and (0, 1) and (2, 2) occur twice in the string. Huffman entropy coding as shown in figure 4.6 can exploit this redundancy in *symbol-1*.

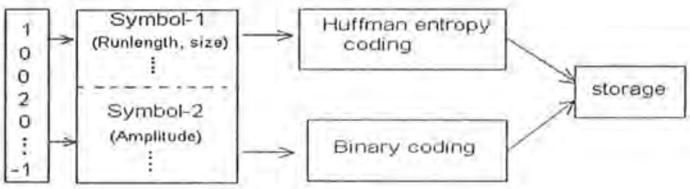

Figure 4.6: Steps in a sequential baseline coder.

Coding Of The DC Sequence:

DC coefficients measure the average energy of the input signal. There are usually a strong correlation between neighbouring coefficients. For efficiency, we use *differential coding:* save the first coefficient and then the *differences between successive coefficients* . These are coded as for AC coefficients. Since one would not expect long zero strings, *runlength* is not used. *Symbol-1* only gives *size*.

Three error measures are often used to compare coders and perceptual quality:

Mean Square Error $MSE = \frac{1}{mn} \sum_{m=0}^{m-1} \sum_{n=0}^{n-1} \mid x(m,n) - \widehat{x}(m,n) \mid^2$

Peak Signal Noise Ratio $PSNR = 10 \, \log_{10}(\frac{255^2}{MSE})$

	Bit Rate	0.2	0.25	0.3	0.4	0.5	0.6	0.6	0.8	0.9	1.0
Lena	PSNR	32.6	33.5	34.1	35.5	36.4	37.0	37.8	38.5	39.0	39.6
Barbara	PSNR	25.8	26.9	27.8	29.1	29.9	31.6	32.1	33.1	33.7	34.8
Goldhill	PSNR	29.8	30.4	31.0	32.0	32.9	33.8	34.1	34.9	35.6	36.1

Table 4.1: PSNR and Bit rate of Lena, Barbara and Goldhill

Maximum Error $\qquad MaxError = \text{Max}|\ x(m,n) - \widehat{x}(m,n)\ |$

The image is $M \times N$. The MSE and PSNR are directly related, and one normally uses PSNR to measure the coder's objective performance. At high rate, images with PSNR above 32 db are considered to be perceptually lossless [5]. At medium and low rates, the PSNR does not agree with the quality of the image.

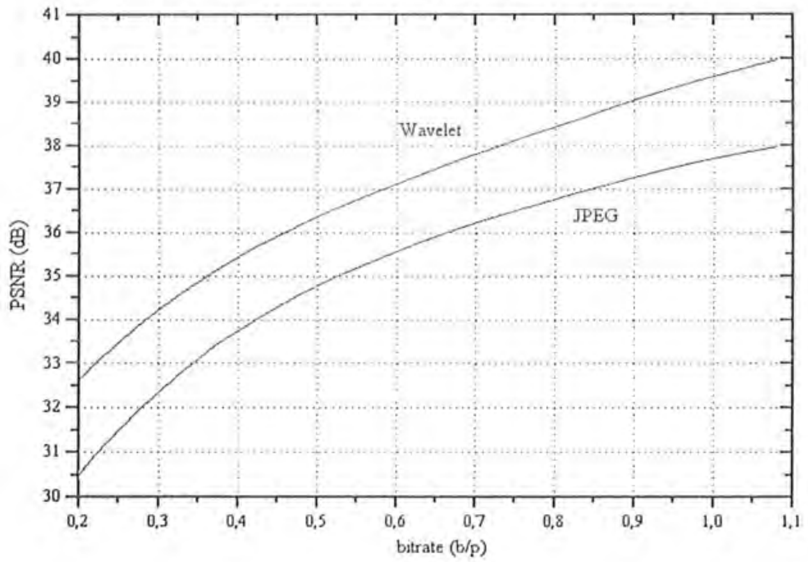

Figure 4.10: Comparisons of wavelet based image coding and JPEG, in bitrate and PSNR for the 512 x 512 Lena image.

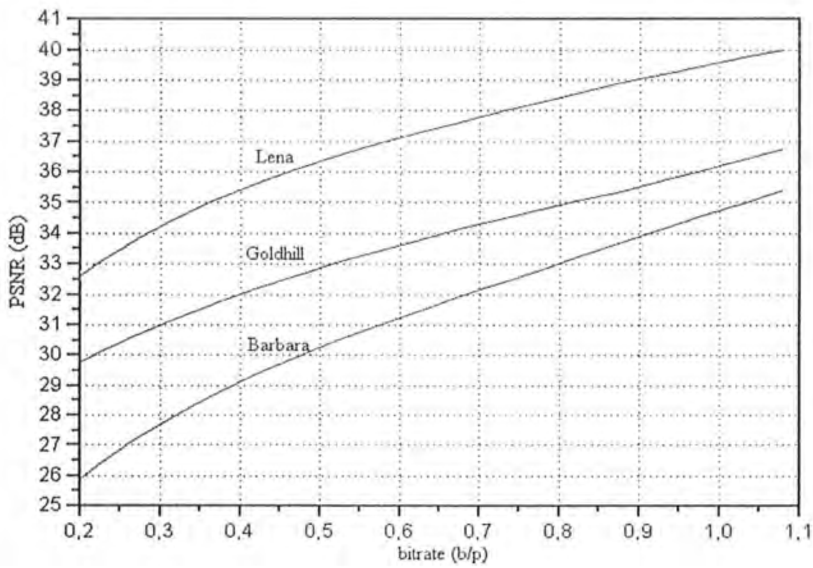

Figure 4.11: Comparisons of wavelet based image coding, in bitrate and PSNR for the 512 x 512 Lena, Barbara and Goldhill images.

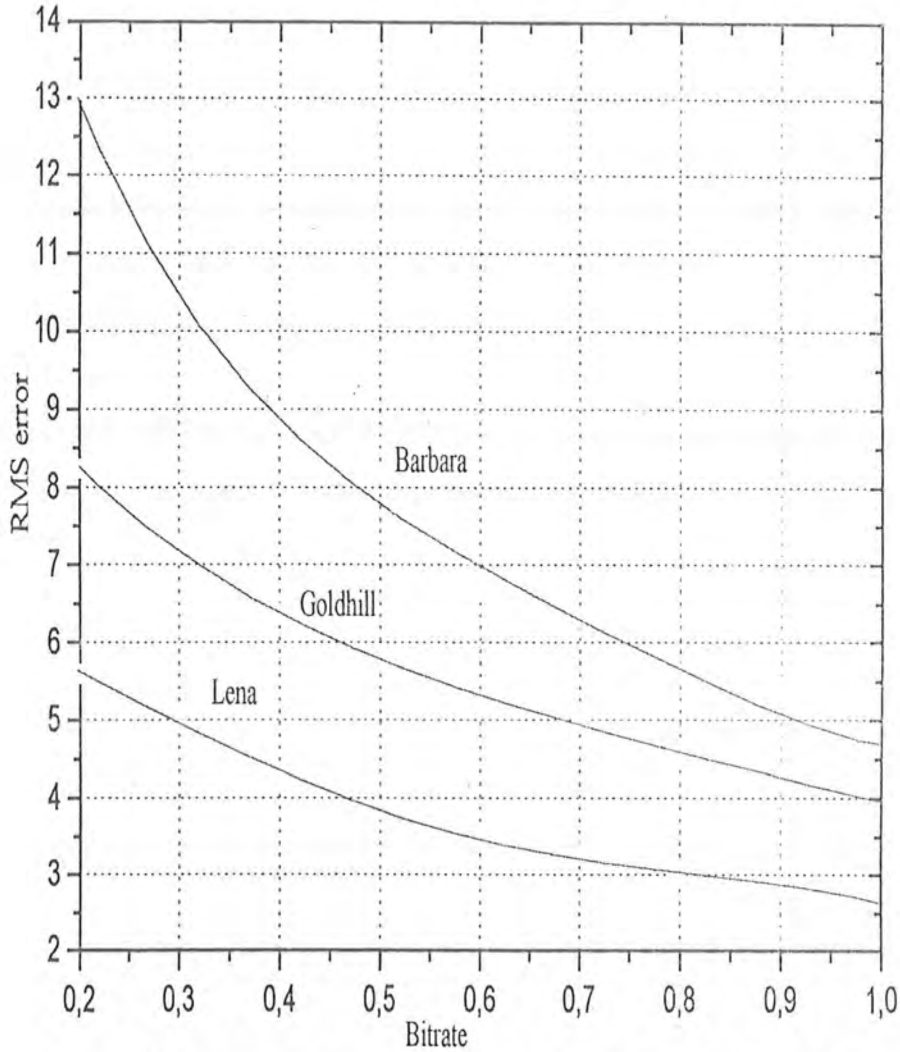

Figure 4.12: Comparisons of wavelet based image coding, in bitrate, RMS error, for the 512 x 512, Barbara, Lena, and Goldhill images

4.5 References

1. ISO / IEC CD 10918-1 Digital Compression and Coding of Continuous-tone still Images, Part 1: Requirements and Guidelines, March 15, 1991.

2. ISO / IEC DIS 11172, Coding of Moving Pictures and associated Audio for Digital Storage Media up to about 1.5 Mbits/s, 1992.

3. K. R. Rao and P.Yip, Discrete Cosine Transform. New York: Academic Press, 1990.

4. Y. Huh, J. J. Hwang, and K. R. Rao, Block wavelet transform coding of images using classified vector quantization, IEEE Trans. on circuits and system for Video Technology, Vol. 5, pp. 63-67, Feb. 1995.

5. G. Strang, T. Nguyen, Wavelets and filter banks, Welle-sley-Cambridge Press, 1996.

6. Y. Meyer, Wavelets: Algorithms and applications: SIAM,1993.

7. M. Vetterli and J. Kovacevic, Wavelets and Subband Coding, Prentice-Hall,1995.

8. G. W. Wornell, Signal Processing with Fractals: A Wavelet-Based Approach, Prentice-Hall,1995.

9. C. K. Chui, ed., Wavelets: A Tutorial in Theory and Applications, Academic Press,1992.

10. I. Daubechies, Ten Lectures on Wavelets, SIAM,1992.

11. K. R. Rao and J. J. Hwang, Techniques and Standard for Image, Video and Audio Coding, Prentice-Hall,1996.

12. S. G. Mallat, "A theory of multiresolution signal decomposition: The wavelet representation, " IEEE Trans. on Pattern Anal. and Mach. Intel., Vol. 11, pp. 674-693, July. 1989.

13. P. P. Vaidyanathan, "Multirate digital filters, filter banks, Polyphase networks, and applications: a tutorial " Proc. IEEE, Vol. 78, pp. 56-93, Jan. 1990.

14. H. Gharavi and A. Tabatabai, "Subband coding of monochrome and color images, " IEEE Trans. Circuits Systems, Vol. 35, pp. 207-214, Feb. 1988.

15. M. Vetterli "A theory of Multirate filter banks, " IEEE Trans. Acoust., Speech and signal Process., Vol. ASSP-35, pp. 356-372, Mar. 1987.

16. J. W. Woods (Editor), Subband image coding, Norwell, MA: Kluwer Academic, 1991.

17. P. E. Fleischer, C. Lan, and M. Lucas, "Digital transport of HDTV on optical fiber, " IEEE Commun. Magazine, Vol. 29, pp. 36-41, Aug. 1991.

18. M. Vetterli and C. Herley, "Wavelets and filter banks: Theory and Design, " IEEE Trans. on Signal Processing., Vol. 40, pp. 2207-2232, Sep. 1992.

19. Yong Huh and J. J. Hwang , "The New Extended JPEG Coder with Variable Quantizer Using Block Wavelet Transform," IEEE Transactions on consumer Electronics, Vol. 43, No. 3, August 1997.

20. Zixiang Xiong, Ph.D, Thesis University of Illinnois at Urbana-Champaign,1996.

Appendix

```
/*---------------------------------------------------------------------------*/
//ENCODE.cpp
/*---------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <iostream.h>
#include <fstream.h>
#include "trans.hh"
#include "coeffset.hh"
#include "allocr.hh"
#include "quant.hh"
/*---------------------------------------------------------------------------*/
void compress       (Image *image, Wavelet *wavelet, int nStages,
                     int capacity, Real p, Real *weight,
                     int paramPrecision, int budget, int nQuant,
                     Real minStepSize, char *filename, int monolayer);
/*---------------------------------------------------------------------------*/

int
    main (
void)
{

  char input_name[50], output_name[50];
  Real Ratio;

#ifdef PGM


  printf("Please enter the Input file name\n"),
  scanf("%s",&input_name);
  printf("Please enter the Output file name\n"),
  scanf("%s",&output_name);
  printf("Please enter the Compression Ratio\n"),
  scanf("%f",&Ratio);

  char *infile_name = input_name;
  char *outfile_name = output_name;
  Real ratio = Ratio;

  // Load the image to be coded
  Image *image = new Image (Infile_name);

  int budget = (int)((Real)(image->hsize*image->vsize)/ratio);  // (assumes 8 bit pix
els)
  printf ("Reading %d x %d image %s, writing %s\nCompression ratio %g:1\n",
          image->hsize, image->vsize, infile_name, outfile_name, ratio);

#else
  if (argc < 6)  {
    fprintf (stderr,
             "Usage: %s [image][width][height][output][ratio]\n",
             program);
    fprintf (stderr,
             "image: image to be compressed (in RAW format)\n");
    fprintf (stderr,
             "width, height: width and height of image to be compressed\n");
    fprintf (stderr,
             "output: name of compressed image\n");
    fprintf (stderr,
             "ratio: compression ratio\n");
    exit(0);
  }
  char *infile_name = argv[1];
  int hsize = atoi(argv[2]);
  int vsize = atoi(argv[3]);
  char *outfile_name = argv[4];
  Real ratio = atof(argv[5]);
  int budget = (int)((Real)(hsize*vsize)/ratio);  // (assumes 8 bit pixels)
  printf ("Reading %d x %d image %s, writing %s\nCompression ratio %g:1\n",
          hsize, vsize, infile_name, outfile_name, ratio);

  // Load the image to be coded
   Image *image = new Image (infile_name, hsize, vsize);
#endif

  // Create a new wavelet from the 7/9 Antonini filterset
  Wavelet *wavelet = new Wavelet (&Antonini);

  // Coding parameters
  Real p = 2.0;             // exponent for L^p error metric
  int nStages = 5;          // # of stages in the wavelet transform
  int capacity = 512;       // capacity of histogram for arithmetic coder
  int paramPrecision = 4;   // precision for stored quantizer parameters
  int nQuant = 10;          // # of quantizers to examine for allocation
  Real minStepSize = 0.05;  // smallest quantizer step to consider
  int monolayer = FALSE;    // TRUE for non-embedded uniform quantizer
                            // FALSE for multilayer quantizer

  Real *weight =            // perceptual weights for coefficient sets
    new Real[3*nStages+1];
  // for now give all sets equal weight
  for (int i = 0; i < 3*nStages+1; i++)
    weight[i] = 1.0;

  compress (image, wavelet, nStages, capacity, p, weight,
            paramPrecision, budget, nQuant, minStepSize, outfile_name,
            monolayer);

  delete [] weight;
  delete image;
  delete wavelet;
  return 0;
}


/*---------------------------------------------------------------------------*/
void compress (Image *image, Wavelet *wavelet, int nStages,
               int capacity, Real p, Real *weight,
               int paramPrecision, int budget, int nQuant,
               Real minStepSize, char *filename, int monolayer)
{
  int i;
  // Compute the wavelet transform of the given image
  WaveletTransform *transform =
    new WaveletTransform (wavelet, image, nStages);

  // For each subband allocate a CoeffSet, an error metric, an
  // EntropyCoder, and a Quantizer
  int nSets = transform->nSubbands;
  CoeffSet **coeff = new CoeffSet* [nSets];
  ErrorMetric **err = new ErrorMetric* [nSets];
  EntropyCoder **entropy = new EntropyCoder* [nSets];
```

```
Quantizer **quant = new Quantizer* [nSets];

for (i = 0; i < nSets; i++) {

  err[i] = new LpError (p);

  if (monolayer) {
    // Use uniform quantizer and single layer escape coder for each
    //   subband
    entropy[i] = new EscapeCoder  (capacity);
    // Assume all subbands have pdf's centered around 0 except the
    //   low pass subband 0
    quant[i]   = new UniformQuant ((MonoLayerCoder *)entropy[i],
                                   paramPrecision, i != 0, err[i]);
  } else {
    // Use a layered quantizer with dead zones and a layered entropy
    //   coder for each subband
    entropy [i] = new LayerCoder (nQuant, i != 0, capacity);
    // Assume all subbands have pdf's centered around 0 except the
    //   low pass subband 0
    quant [i] = new LayerQuant ((MultiLayerCoder *)entropy[i],
                                paramPrecision, i != 0, nQuant, err[i]);
  }
  // Partition the wavelet transformed coefficients into subbands --
  //   each subband will have a different quantizer
  coeff[i]   = new CoeffSet (transform->subband(i),
                             transform->subbandSize[i], quant[i]);

  // For each subband determine the rate and distortion for each of
  //    the possible nQuant quantizers
  coeff[i]->getRateDist (nQuant, minStepSize);
}

Allocator *allocator = new Allocator ();
// Use rate/distortion information for each subband to find bit
//    allocation that minimizes total (weighted) distortion subject
//    to a byte budget
budget -= nSets * 4;  // subtract off approximate size of header info
allocator->optimalAllocate (coeff, nSets, budget, TRUE, weight);
printf ("Target rate = %d bytes\n", budget);
// Display the resulting allocation
allocator->print (coeff, nSets);

// Open output file
ofstream outfile (filename, ios::out | ios::trunc | ios::binary);
if (!outfile) {
  error ("Unable to open file %s", filename);
}
// Create I/O interface object for arithmetic coder
Encoder *encoder = new Encoder (outfile);

// Write image size to output file
encoder->writePositive (image->hsize);
encoder->writePositive (image->vsize);
//  printf ("hsize = %d, vsize = %d\n", image->hsize, image->vsize);

for (i = 0; i < nSets; i++) {
  // Write quantizer parameters for each subband to file
  coeff[i]->writeHeader (encoder, allocator->precision[i]);


}
for (i = 0; i < nSets; i++) {
    // Quantize and write entropy coded coefficients for each subband
    coeff[i]->encode (encoder, allocator->precision[i]);
  }

  // Flush bits from arithmetic coder and close file
  encoder->flush ();
  delete encoder;
  outfile.close ();

  // Clean up
  for (i = 0; i < nSets; i++) {
    delete err[i];
    delete entropy[i];
    delete quant[i];
    delete coeff[i];
  }
  delete [] err;
  delete [] entropy;
  delete [] quant;
  delete [] coeff;
  delete allocator;
  delete transform;
}

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

```
/*--------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <iostream.h>
#include <fstream.h>
#include "trans.hh"
#include "coeffset.hh"
#include "allocr.hh"
#include "quant.hh"
/*--------------------------------------------------------------------------*/
void decompress     (Image **image, Wavelet *wavelet, int nStages,
                     int capacity, int paramPrecision, char *filename,
                     int nQuant, int monolayer);
/*--------------------------------------------------------------------------*/

int
    main (
void)
{

  char input_file[50], output_file[50];


  printf("Please Enter The Coded File\n");
  scanf("%s",&input_file);
  printf("Please Enter The Output File\n");
  scanf("%s",&output_file);

  char *infile_name = input_file;
  char *outfile_name = output_file;

  printf ("Reading compressed image %s, writing %s\n\n",
          infile_name, outfile_name);

  // Create a new wavelet from the 7/9 Antonini filterset
  Wavelet *wavelet = new Wavelet (&Antonini);

  // Coding parameters
  int nStages = 5;           // # of stages in the wavelet transform
  int capacity = 512;        // capacity of histogram for arithmetic coder
  int paramPrecision = 4;    // precision for stored quantizer parameters
  int nQuant = 10;           // # of quantizers to examine for allocation
  int monolayer = FALSE;     // TRUE for non-embedded uniform quantizer
                             // FALSE for multilayer quantizer

  Image *reconstruct;
  decompress (&reconstruct, wavelet, nStages, capacity, paramPrecision,
              infile_name, nQuant, monolayer);
#ifdef PGM
  reconstruct->savePGM (outfile_name);
#else
  reconstruct->saveRaw (outfile_name);
#endif
  delete reconstruct;
  delete wavelet;

  return 0;
}
```

```
/*--------------------------------------------------------------------------*/
void decompress (Image **image, Wavelet *wavelet, int nStages,
                 int capacity, int paramPrecision, char *filename,
                 int maxQuant, int monolayer)
{
  int i;

  // open compressed image file
  ifstream infile (filename, ios::in | ios::nocreate | ios::binary);
  if (!infile) {
    error ("Unable to open file %s", filename);
  }

  // Create I/O interface object for arithmetic decoder
  Decoder *decoder = new Decoder (infile);

  // Read image dimensions from file
  int hsize = decoder->readPositive ();
  int vsize = decoder->readPositive ();
  //  printf ("hsize = %d, vsize = %d\n", hsize, vsize);


  WaveletTransform *transform =
    new WaveletTransform (wavelet, hsize, vsize, nStages);

  // For each subband allocate a CoeffSet, an EntropyCoder, and a
  //    Quantizer (don't need to know anything about errors here)
  int nSets = transform->nSubbands;
  CoeffSet **coeff = new CoeffSet* [nSets];
  EntropyCoder **entropy = new EntropyCoder* [nSets];
  Quantizer **quant = new Quantizer* [nSets];
  // Quantizer precision for each subband
  int *precision = new int [nSets];

  for (i = 0; i < nSets; i++) {
    if (monolayer) {
      // Use uniform quantizer and single layer escape coder for each
      //    subband
      entropy[i] = new EscapeCoder (capacity);
      // Assume all subbands have pdf's centered around 0 except the
      //    low pass subband 0
      quant[i]   = new UniformQuant
        ((MonoLayerCoder *)entropy[i], paramPrecision, i != 0);
    } else {

      entropy [i] = new LayerCoder (maxQuant, i != 0, capacity);
      // Assume all subbands have pdf's centered around 0 except the
      //    low pass subband 0
      quant [i] = new LayerQuant
        ((MultiLayerCoder *)entropy[i], paramPrecision, i != 0, maxQuant);
    }
    // Indicate that each set of coefficients to be read corresponds
    //    to a subband
    coeff[i]   = new CoeffSet (transform->subband(i),
                               transform->subbandSize[i], quant[i]);
  }

  for (i = 0; i < nSets; i++) {
    // Read quantizer parameters for each subband
    coeff[i]->readHeader (decoder, precision[i]);

  }
  for (i = 0; i < nSets; i++) {
```

```
    // Read, decode, and dequantize coefficients for each subband
    coeff[i]->decode (decoder, precision[i]);
  }

  // Close file
  delete decoder;
  infile.close ();

  // Clean up
  for (i = 0; i < nSets; i++) {
    delete entropy[i];
    delete quant[i];
    delete coeff[i];
  }
  delete [] entropy;
  delete [] quant;
  delete [] coeff;
  delete [] precision;

  // Allocate image and invert transform
  *image = new Image (hsize, vsize);
  transform->invert (*image);
  delete transform;
}

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

```
/*      Allocato.cpp
/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include "global.hh"
#include "entropy.hh"
#include "quant.hh"
#include "coder.hh"
#include "allocr.hh"
/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/

Allocator::Allocator ()
{
  precision = NULL;
}

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

Allocator::~Allocator ()
{
  if (precision != NULL)
    delete [] precision;
}

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

void Allocator::resetPrecision (int nSets)
{
  if (precision != NULL)
    delete [] precision;

  precision = new int [nSets];
  for (int i = 0; i < nSets; i++)
    precision[i] = 0;
}

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

void Allocator::optimalAllocate (CoeffSet **coeff, int nSets,
                                 int budget, int augment, Real *weight)
{
  Real bitBudget = 8*budget;

  Real lambda, lambdaLow, lambdaHigh;
  Real rateLow, rateHigh, currentRate;
  Real distLow, distHigh, currentDist;

  resetPrecision (nSets);

  lambdaLow = 0.0;
  allocateLambda (coeff, nSets, lambdaLow, rateLow, distLow, weight);
  if (rateLow < bitBudget) // this uses the largest possible # of bits
    return;                //   -- if this is within the budget, do it

  lambdaHigh = 1000000.0;
  Real lastRateHigh = -1;
  do {
    // try to use the smallest possible # of bits
    allocateLambda (coeff, nSets, lambdaHigh, rateHigh, distHigh, weight);

    // if this is still > bitBudget, try again w/ larger lambda
    if (rateHigh > bitBudget && lastRateHigh != rateHigh) {
      lambdaLow = lambdaHigh;
      rateLow = rateHigh;
      distLow = distHigh;
      lambdaHigh *= 10.0;
    }
  } while (rateHigh > bitBudget && lastRateHigh != rateHigh);

  // give up when changing lambda has no effect on things
  if (lastRateHigh == rateHigh)
    return;

  // Note rateLow will be > rateHigh
  if (rateLow < bitBudget)
    error ("Failed to bracket bit budget = %d: rateLow = %g rateHigh = %g\n",
           budget, rateLow, rateHigh);

  while (lambdaHigh - lambdaLow > 0.01)  {
    lambda = (lambdaLow + lambdaHigh)/2.0;

    allocateLambda (coeff, nSets, lambda, currentRate, currentDist, weight);

    if (currentRate > bitBudget)
      lambdaLow = lambda;
    else
      lambdaHigh = lambda;
  }

  if (currentRate > bitBudget)  {
    lambda = lambdaHigh;
    allocateLambda (coeff, nSets, lambda, currentRate, currentDist, weight);
  }

  if (augment)
    greedyAugment (coeff, nSets, bitBudget-currentRate, weight);
}

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

void Allocator::allocateLambda  (CoeffSet **coeff, int nSets,
                                 Real lambda, Real &optimalRate,
                                 Real &optimalDist, Real *weight)
{
  int i, j;
  Real G, minG, minRate, minDist;

  optimalRate = optimalDist = 0.0;

  // want to minimize G = distortion + lambda * rate

  // loop through all rate-distortion curves
  for (i = 0; i < nSets; i++)  {
    minG = minRate = minDist = MaxReal;

    for (j = 0; j < coeff[i]->nQuant; j++) {
      G = weight[i]*coeff[i]->dist[j] + lambda * coeff[i]->rate[j];
      if (G < minG)  {
        minG = G;
        minRate = coeff[i]->rate[j];
        minDist = weight[i]*coeff[i]->dist[j];
        precision[i] = j;
      }
```

```
    }

    optimalRate += minRate;
    optimalDist += minDist;
  }
}

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

void Allocator::greedyAugment (CoeffSet **coeff, int nSets,
                               Real bitsLeft, Real *weight)
{
  int bestSet, newPrecision = -1;
  Real delta, maxDelta, bestDeltaDist, bestDeltaRate = 0;

  do {
    bestSet = -1;
    maxDelta = 0;

    // Find best coeff set to augment
    for (int i = 0; i < nSets; i++) {
      for (int j = precision[i]+1; j < coeff[i]->nQuant; j++) {
        Real deltaRate = coeff[i]->rate[j] -
          coeff[i]->rate[precision[i]];
        Real deltaDist = -weight[i]*(coeff[i]->dist[j] -
          coeff[i]->dist[precision[i]]);

        if (deltaRate != 0 && deltaRate <= bitsLeft) {
          delta = deltaDist / deltaRate;

          if (delta > maxDelta) {
            maxDelta = delta;
            bestDeltaRate = deltaRate;
            bestDeltaDist = deltaDist;
            bestSet = i;
            newPrecision = j;
          }
        }
      }
    }

    if (bestSet != -1) {
      precision[bestSet] = newPrecision;
      bitsLeft -= bestDeltaRate;
    }
  } while (bestSet != -1);
}

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

void Allocator::print (CoeffSet **coeff, int nSets)
{
  Real totalRate = 0, totalDist = 0;
  int totalData = 0;

   for (int i = 0; i < nSets; i++) {

   totalRate += coeff[i]->rate[precision[i]];
   totalDist += coeff[i]->dist[precision[i]];
   totalData += coeff[i]->nData;
   }
  Real rms = sqrt(totalDist/(Real)totalData);
   Real psnr = 20.0 * log(255.0/rms)/log(10.0);

   printf ("\n");
  // printf ("total rate = %g\n", totalRate/8.0);
  // printf ("total dist = %g\n", totalDist);
  //printf ("total coeffs = %d\n", totalData);
   printf ("RMS error = %g\n", rms);
   printf ("PSNR (transform domain) = %g\n", psnr);
}

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

```
//
#include <iostream.h>
#include <iomanip.h>
#include <assert.h>
#include <math.h>
#include "global.hh"
#include "BitIO.h"
#include "iHisto.h"
#include "Arith.h"

const int CodingValues::CodeValueBits = 16;
const long CodingValues::MaxFreq =  ((long)1 << (CodeValueBits - 2)) - 1;
const long CodingValues::One = ((long)1 << CodeValueBits) - 1;
const long CodingValues::Qtr = One / 4 + 1;
const long CodingValues::Half = 2 * Qtr;
const long CodingValues::ThreeQtr = 3 * Qtr;


ArithEncoder::ArithEncoder(BitOut &bo) : output(bo)
{
    low = 0;
    high = One;
    bitsToFollow = 0;
}


void
ArithEncoder::flush(void)
{
    for (int i = 0; i < CodeValueBits; i++) {
        if (low >= Half) {
            bpf(1);
            low -= Half;
        } else {
            bpf(0);
        }
        low = 2 * low;
    }
    output.flush();
}

void ArithEncoder::Encode(int count, int countLeft, int countTot)
{
    //    cerr << "Encode(" << count << ", " << countLeft << ", " << countTot << ")\n
";


    assert(count);

    long range = high - low + 1;
    high = low + (range * (countLeft + count)) / countTot - 1;
    low = low + (range * countLeft) / countTot;

    while (1) {
        if (high < Half) {
            bpf(0);
        } else if (low >= Half) {
            bpf(1);
            low -= Half;
            high -= Half;
        } else if (low >= Qtr && high < ThreeQtr) {
            bitsToFollow++;
            low -= Qtr;
```

```
            high -= Qtr;
        } else {
            break;
        }
        low = 2 * low;
        high = 2 * high + 1;
    }
}

ArithDecoder::ArithDecoder(BitIn &bi) : input(bi)
{
    low = 0;
    high = One;
    value = 0;
    for (int i = 0; i < CodeValueBits; i++) {
        value = (value << 1) + input.input_bit();
    }

}

int
ArithDecoder::Decode(iHistogram &h)
{
    long range;
    int cum;
    int answer;
    int ct, ctLeft, ctTotal;

    ctTotal = h.TotalCount();
    range = high - low + 1;
    cum = (((long)(value - low) + 1) * ctTotal - 1) / range;
    answer = h.Symbol(cum);

    ct =  h.Count(answer); ctLeft = h.LeftCount(answer);

    //    cerr << "Decoder : cum= " << cum <<  "-> " << answer << "\n";
    //    cerr << "          ct = " << ct << " ctLeft = " << ctLeft << " ctTotal = " <<
 ctTotal << "\n";

    high = low + (range * (ctLeft + ct)) / ctTotal - 1;
    low = low + (range * ctLeft) / ctTotal;
    while (1) {
        if (high < Half) {
            ;
        } else if (low >= Half) {
            value -= Half;
            low -= Half;
            high -= Half;
        }
        else if (low >= Qtr
                 && high < ThreeQtr) {
            value -= Qtr;
            low -= Qtr;
            high -= Qtr;
        }
        else break;
        low = 2 * low;
        high = 2 * high + 1;
        value = 2 * value + input.input_bit();
    }
    return answer;
}
```

```
/*-----------------------------------------------------------------------*/
//coder.cpp
/*-----------------------------------------------------------------------*/
#include <iostream.h>
#include <stdio.h>
#include <math.h>
#include "global.hh"
#include "coder.hh"
/*-----------------------------------------------------------------------*/

Encoder::Encoder (ostream &out, ostream &log)
{
   bitout   = new BitOut (out, log);
   arith    = new ArithEncoder (*bitout);
   intcoder = new CdeltaEncode (bitout);
}

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

Encoder::~Encoder ()
{
   delete intcoder;
   delete arith;
   delete bitout;
}

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

Decoder::Decoder (istream &in, ostream &log)
{
  bitin    = new BitIn (in, log);
  intcoder = new CdeltaDecode (bitin);
  arith    = NULL;
}

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

Decoder::~Decoder ()
{
   delete intcoder;
   if (arith != NULL) delete arith;
   delete bitin;
}

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

```
/*---------------------------------------------------------------------------*/
// Filter.cpp
/*---------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include "wavelet.hh"
/*---------------------------------------------------------------------------*/
Real HaarCoeffs [] = { 1.0/Sqrt2, 1.0/Sqrt2 };

// A few Daubechies filters

Real Daub4Coeffs [] = { 0.4829629131445341,  0.8365163037378077,
                         0.2241438680420134, -0.1294095225512603 };

Real Daub6Coeffs [] = { 0.3326705529500825,  0.8068915093110924,
                         0.4598775021184914, -0.1350110200102546,
                        -0.0854412738820267,  0.0352262918857095 };

Real Daub8Coeffs [] = { 0.2303778133088964,  0.7148465705529154,
                         0.6308807679398587, -0.0279837694168599,
                        -0.1870348117190931,  0.0308413818355607,
                         0.0328830116668852, -0.0105974017850690 };

// Filter from Eero Simoncelli's PhD thesis -- used in Edward Adelson's EPIC wavelet
coder
// These are probably the filter coefficients used in Shapiro's EZW paper
Real AdelsonCoeffs[] = { 0.028220367, -0.060394127, -0.07388188,
                        0.41394752,   0.7984298,    0.41394752,
                        -0.07388188, -0.060394127, 0.028220367 };

// 7/9 Filter from M. Antonini, M. Barlaud, P. Mathieu, and
// I. Daubechies, "Image coding using wavelet transform", IEEE
// Transactions on Image Processing", Vol. pp. 205-220, 1992.

Real AntoniniSynthesis [] = { -6.453888262893856e-02,
                              -4.068941760955867e-02,
                               4.180922732222124e-01,
                               7.884856164056651e-01,
                               4.180922732222124e-01,
                              -4.068941760955867e-02,
                              -6.453888262893856e-02 };
Real AntoniniAnalysis[] =  {  3.782845550699535e-02,
                             -2.384946501937986e-02,
                             -1.106244044184226e-01,
                              3.774028556126536e-01,
                              8.526986790094022e-01,
                              3.774028556126537e-01,
                             -1.106244044184226e-01,
                             -2.384946501937986e-02,
                              3.782845550699535e-02 };

// Unpublished 18/10 filter from Villasenor's group

Real Villa1810Synthesis [] = { 9.544158682436510e-04,
                         -2.727196296995984e-06,
                         -9.452462998353147e-03,
                         -2.528037293949898e-03,
                          3.083373438534281e-02,
                         -1.376513483818621e-02,
                         -8.566118833165798e-02,
                          1.633685405569902e-01,
                          6.233596410344172e-01,
                          6.233596410344158e-01,
                          1.633685405569888e-01,
                         -8.566118833165885e-02,
                         -1.376513483818652e-02,
                          3.083373438534267e-02,
                         -2.528037293949898e-03,
                         -9.452462998353147e-03,
                         -2.727196296995984e-06,
                          9.544158682436510e-04};
Real Villa1810Analysis [] = {  2.885256501123136e-02,
                          8.244478227504624e-05,
                         -1.575264469076351e-01,
                          7.679048884691438e-02,
                          7.589077294537618e-01,
                          7.589077294537619e-01,
                          7.679048884691436e-02,
                         -1.575264469076351e-01,
                          8.244478227504624e-05,
                          2.885256501123136e-02};

// Filters from Chris Brislawn's tutorial code

Real BrislawnAnalysis [] = {  0.037828455506995,  -0.023849465019380,
                             -0.110624404418423,   0.377402855612654,
                              0.852698679009403,
                              0.377402855612654,  -0.110624404418423,
                             -0.023849465019380,   0.037828455506995 };
Real BrislawnSynthesis [] = { -0.064538882628938, -0.040689417609558,
                               0.418092273222212,
                               0.788485616405664,
                               0.418092273222212,
                              -0.040689417609558, -0.064538882628938};

Real Brislawn2Analysis [] = {  0.026913419, -0.032303352,
                              -0.241109818,  0.054100420,
                               0.899506092,  0.899506092,
                               0.054100420, -0.241109818,
                              -0.032303352,  0.026913419};
Real Brislawn2Synthesis [] = { 0.019843545,  0.023817599,
                              -0.023257840,  0.145570740,
                               0.541132748,  0.541132748,
                               0.145570740, -0.023257840,
                               0.023817599, 0.019843545 };

// Filters from J. Villasenor, B. Belzer, J. Liao, "Wavelet Filter
// Evaluation for Image Compression." IEEE Transactions on Image
// Processing, Vol. 2, pp. 1053-1060, August 1995.

Real Villa1Analysis [] = {
  3.782845550699535e-02,
  -2.384946501937986e-02,
  -1.106244044184226e-01,
  3.774028556126536e-01,
  8.526986790094022e-01,
  3.774028556126537e-01,
  -1.106244044184226e-01,
  -2.384946501937986e-02,
  3.782845550699535e-02
};

Real Villa1Synthesis [] = {
```

```
  -6.453888262893856e-02,
  -4.068941760955867e-02,
  4.180922732222124e-01,
  7.884856164056651e-01,
  4.180922732222124e-01,
  -4.068941760955867e-02,
  -6.453888262893856e-02
};

Real Villa2Analysis [] = {
  -8.472827741318157e-03,
  3.759210316686883e-03,
  4.728175282882753e-02,
  -3.347508104780150e-02,
  -6.887811419061032e-02,
  3.832692613243884e-01,
  7.672451593927493e-01,
  3.832692613243889e-01,
  -6.887811419061045e-02,
  -3.347508104780156e-02,
  4.728175282882753e-02,
  3.759210316686883e-03,
  -8.472827741318157e-03
};
Real Villa2Synthesis [] = {
  1.418215589126359e-02,
  6.292315666859828e-03,
  -1.087373652243805e-01,
  -6.916271012030040e-02,
  4.481085999263908e-01,
  8.328475700934288e-01,
  4.481085999263908e-01,
  -6.916271012030040e-02,
  -1.087373652243805e-01,
  6.292315666859828e-03,
  1.418215589126359e-02
};

Real Villa3Analysis [] = {
  -1.290777652578771e-01,
  4.769893003875977e-02,
  7.884856164056651e-01,
  7.884856164056651e-01,
  4.769893003875977e-02,
  -1.290777652578771e-01
};
Real Villa3Synthesis [] = {
  1.891422775349768e-02,
  6.989495243807747e-03,
  -6.723693471890128e-02,
  1.333892255971154e-01,
  6.150507673110278e-01,
  6.150507673110278e-01,
  1.333892255971154e-01,
  -6.723693471890128e-02,
  6.989495243807747e-03,
  1.891422775349768e-02
};

Real Villa4Analysis [] = {
  -1.767766952966369e-01,
  3.535533905932738e-01,
  1.060660171779821e+00,
  3.535533905932738e-01,
  -1.767766952966369e-01
};
Real Villa4Synthesis [] = {
  3.535533905932738e-01,
  7.071067811865476e-01,
  3.535533905932738e-01
};

Real Villa5Analysis [] = {
  7.071067811865476e-01,
  7.071067811865476e-01
};
Real Villa5Synthesis [] = {
  -8.838834764831845e-02,
  8.838834764831845e-02,
  7.071067811865476e-01,
  7.071067811865476e-01,
  8.838834764831845e-02,
  -8.838834764831845e-02
};

Real Villa6Analysis [] = {
  3.314563036811943e-02,
  -6.629126073623885e-02,
  -1.767766952966369e-01,
  4.198446513295127e-01,
  9.943689110435828e-01,
  4.198446513295127e-01,
  -1.767766952966369e-01,
  -6.629126073623885e-02,
  3.314563036811943e-02

};
Real Villa6Synthesis [] = {
  3.535533905932738e-01,
  7.071067811865476e-01,
  3.535533905932738e-01
};

// Filter

Real OdegardAnalysis[] = {
   5.2865768532960523e-02,
  -3.3418473279346828e-02,
  -9.3069263703582719e-02,
   3.8697186387262039e-01,
   7.8751377152779212e-01,
   3.8697186387262039e-01,
  -9.3069263703582719e-02,
  -3.3418473279346828e-02,
   5.2865768532960523e-02
};
Real OdegardSynthesis[] = {
  -8.6748316131711606e-02,
  -5.4836926902779436e-02,
   4.4030170672498536e-01,
   8.1678063499210640e-01,
   4.4030170672498536e-01,
  -5.4836926902779436e-02,
  -8.6748316131711606e-02
};
```

```
FilterSet Haar      (FALSE, HaarCoeffs,          2, 0);
FilterSet Daub4     (FALSE, Daub4Coeffs,         4, 0);
FilterSet Daub6     (FALSE, Daub6Coeffs,         6, 0);
FilterSet Daub8     (FALSE, Daub8Coeffs,         8, 0);
FilterSet Antonini  (TRUE,  AntoniniAnalysis,    9, -4,
                            AntoniniSynthesis,   7, -3);
FilterSet Villa1810 (TRUE,  Villa1810Analysis,  10, -4,
                            Villa1810Synthesis, 18, -8);
FilterSet Adelson   (TRUE,  AdelsonCoeffs,       9, -4);
FilterSet Brislawn  (TRUE,  BrislawnAnalysis,    9, -4,
                            BrislawnSynthesis,   7, -3);
FilterSet Brislawn2 (TRUE,  Brislawn2Analysis,  10, -4,
                            Brislawn2Synthesis, 10, -4);

FilterSet Villa1 (TRUE, Villa1Analysis,  9, -4, Villa1Synthesis,  7, -3);
FilterSet Villa2 (TRUE, Villa2Analysis, 13, -6, Villa2Synthesis, 11, -5);
FilterSet Villa3 (TRUE, Villa3Analysis,  6, -2, Villa3Synthesis, 10, -4);
FilterSet Villa4 (TRUE, Villa4Analysis,  5, -2, Villa4Synthesis,  3, -1);
FilterSet Villa5 (TRUE, Villa5Analysis,  2,  0, Villa5Synthesis,  6, -2);
FilterSet Villa6 (TRUE, Villa6Analysis,  9, -4, Villa6Synthesis,  3, -1);


FilterSet Odegard (TRUE, OdegardAnalysis, 9, -4, OdegardSynthesis, 7, -3);
/*---------------------------------------------------------------------------*/
// Destructor

Filter::~Filter ()
{
  if (coeff != NULL)
    delete [] coeff;
}

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

void Filter::init (int filterSize, int filterFirst, Real *data)
{
   size = filterSize;
   firstIndex = filterFirst;
   center = -firstIndex;

   coeff = new Real [size];
   if (data != NULL) {
     for (int i = 0; i < size; i++)
       coeff[i] = data[i];
   } else {
     for (int i = 0; i < size; i++)
       coeff[i] = 0;
   }
}

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

void Filter::copy (const Filter& filter)
{
  if (coeff != NULL)
    delete [] coeff;
  init (filter.size, filter.firstIndex, filter.coeff);
}

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

FilterSet::FilterSet (int symmetric,
          Real *anLow, int anLowSize, int anLowFirst,
          Real *synLow, int synLowSize, int synLowFirst) :
          symmetric(symmetric)
{
  int i, sign;

  analysisLow = new Filter (anLowSize, anLowFirst, anLow);

  // If no synthesis coeffs are given, assume wavelet is orthogonal
  if (synLow == NULL)  {
    synthesisLow = new Filter (*analysisLow);

    // For orthogonal wavelets, compute the high pass filter using
    // the relation g_n = (-1)^n h_(1-n)^*
    // (or equivalently g_(1-n) = (-1)^(1-n) h_n^*)

    analysisHigh = new Filter (analysisLow->size, 2 - analysisLow->size -
                               analysisLow->firstIndex);

    // Compute (-1)^(1-n) for first n
    if (analysisLow->firstIndex % 2)
      sign = 1;
    else sign = -1;

    for (i = 0; i < analysisLow->size; i++)  {
      analysisHigh->coeff[1 - i - analysisLow->firstIndex -
                          analysisHigh->firstIndex] =
        sign * analysisLow->coeff[i];
      assert (1 - i - analysisLow->firstIndex -
                          analysisHigh->firstIndex >= 0);
      assert (1 - i - analysisLow->firstIndex -
                          analysisHigh->firstIndex < analysisHigh->size);
      sign *= -1;
    }

    // Copy the high pass analysis filter to the synthesis filter
    synthesisHigh = new Filter (*analysisHigh);

  } else {
    // If separate synthesis coeffs given, assume biorthogonal

    synthesisLow = new Filter (synLowSize, synLowFirst, synLow);

    // For orthogonal wavelets, compute the high frequency filter using
    // the relation g_n = (-1)^n complement (h~_(1-n)) and
    //              g~_n = (-1)^n complement (h_(1-n))
    // (or equivalently g_(1-n) = (-1)^(1-n) complement (h~_n))

    analysisHigh = new Filter (synthesisLow->size, 2 - synthesisLow->size -
                         synthesisLow->firstIndex);

    // Compute (-1)^(1-n) for first n
    if (synthesisLow->firstIndex % 2)
      sign = 1;
    else sign = -1;

    for (i = 0; i < synthesisLow->size; i++)  {
      analysisHigh->coeff[1 - i - synthesisLow->firstIndex -
                         analysisHigh->firstIndex] =
        sign * synthesisLow->coeff[i];
      assert (1 - i - synthesisLow->firstIndex -
                          analysisHigh->firstIndex >= 0);
      assert (1 - i - synthesisLow->firstIndex -
```

```
                          analysisHigh->firstIndex < analysisHigh->size);
      sign *= -1;
    }

    synthesisHigh = new Filter
                          (analysisLow->size, 2 - analysisLow->size -
                           analysisLow->firstIndex);

    // Compute (-1)^(1-n) for first n
    if (analysisLow->firstIndex % 2)
      sign = 1;
    else sign = -1;

    for (i = 0; i < analysisLow->size; i++)  {
      synthesisHigh->coeff[1 - i - analysisLow->firstIndex -
                          synthesisHigh->firstIndex] =
        sign * analysisLow->coeff[i];
      assert (1 - i - analysisLow->firstIndex -
                          synthesisHigh->firstIndex >= 0);
      assert (1 - i - analysisLow->firstIndex -
                          synthesisHigh->firstIndex < synthesisHigh->size);
      sign *= -1;
    }
  }
}

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

FilterSet::FilterSet (const FilterSet& filterset)
{
  copy (filterset);
}

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

FilterSet::~FilterSet ()
{
  delete analysisLow;
  delete analysisHigh;
  delete synthesisLow;
  delete synthesisHigh;
}

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

FilterSet& FilterSet::operator= (const FilterSet filterset)
{
  delete analysisLow;
  delete analysisHigh;
  delete synthesisLow;
  delete synthesisHigh;
  copy (filterset);
  return *this;
}

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

void FilterSet::copy (const FilterSet& filterset)
{
  symmetric = filterset.symmetric;
  analysisLow = new Filter (*(filterset.analysisLow));
  analysisHigh = new Filter (*(filterset.analysisHigh));
  synthesisLow = new Filter (*(filterset.synthesisLow));
  synthesisHigh = new Filter (*(filterset.synthesisHigh));
}

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

```
/*------------------------------------------------------------------------*/
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <new.h>
#include <math.h>
#include <assert.h>
#include "global.hh"
/*------------------------------------------------------------------------*/
#ifdef DEBUG
static FILE *debug_file;
static int debug_file_open = FALSE;
#endif

/*------------------------------------------------------------------------*/
// function called when out of memory -- put a debugger breakpoint here
//   if trying to locate cause of Out of memory error

void no_more_memory ()
{
   error ("Out of memory");
}

/*------------------------------------------------------------------------*/
// Initialize system-level things

void init()
{
  // Call no_more_memory when unable to malloc
  set_new_handler (no_more_memory);

#ifdef DEBUG
  debug_file = fopen ("debug.log", "w+");
  debug_file_open = (debug_file != NULL);
#endif
}

/*------------------------------------------------------------------------*/
// Close down system-level stuff

void shut_down ()
{
#ifdef DEBUG
   fclose (debug_file);
   debug_file_open = FALSE;
#endif
}

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

volatile void error (char *format, ...)
{
   va_list list;

   va_start (list, format);

   printf ("Error: ");
   vprintf (format, list);
   va_end (list);
   printf ("\n");

#ifdef DEBUG
   if (debug_file_open) {
     fprintf (debug_file, "Error: ");
     vfprintf (debug_file, format, list);
     fprintf (debug_file, "\n");
     fflush (debug_file);
   }
#endif

   assert(0);
}

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

void warning (char *format, ...)
{
   va_list list;

   va_start (list, format);

#ifdef DEBUG
   if (debug_file_open) {
     fprintf (debug_file, "Warning: ");
     vfprintf (debug_file, format, list);
     fprintf (debug_file, "\n");
     fflush (debug_file);
   }
#endif

   printf ("Warning: ");
   vprintf (format, list);
   va_end (list);
   printf ("\n");
}

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

```
/*-------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <iostream.h>
#include "global.hh"
#include "image.hh"
/*-------------------------------------------------------------------------*/

int main (int argc, char **argv)
{
   Image *image;

   char *program = argv[0];

   if (argc != 3)  {
      fprintf (stderr,
               "Convert an image in pbm/pgm format to raw pixel format\n");
      fprintf (stderr,
               "Usage: %s [pgm image name][raw image name]\n", program);

      return 1;
   }

   image = new Image (argv[1]);
   image->saveRaw (argv[2]);
   return 0;
}

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

```
/*-----------------------------------------------------------------------*/
//Quantize.cpp
/*-----------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream.h>
#include <iomanip.h>
#include "global.hh"
#include "quant.hh"
/*-----------------------------------------------------------------------*/
Quantizer::Quantizer (ErrorMetric *err) : err (err)
{
  data = NULL;
  nData = 0;
  max = min = sum = sumSq = mean = var = 0;
  initialDist = 0;
}

/*-----------------------------------------------------------------------*/
void Quantizer::getStats ()
{
  max = -MaxReal;
  min = MaxReal;
  sum = sumSq = 0;

  for (int i = 0; i < nData; i++) {
    if (data[i] < min)
      min = data[i];
    if (data[i] > max)
      max = data[i];
    sum += data[i];
    sumSq += square(data[i]);
  }
  mean = sum / (Real)nData;
  var = sumSq / (Real)nData - square (mean);
}

/*-----------------------------------------------------------------------*/
/*-----------------------------------------------------------------------*/
UniformQuant::UniformQuant (MonoLayerCoder *entropy, int paramPrecision,
                            int zeroCenter, ErrorMetric *err) :
  Quantizer (err), entropy (entropy), paramPrecision (paramPrecision),
  zeroCenter (zeroCenter)
{
}

/*-----------------------------------------------------------------------*/
void UniformQuant::setDataEncode (Real *newData, int newNData)
{
  data = newData;
  nData = newNData;

  getStats ();

  if (zeroCenter) {
    max = fabs(max) > fabs(min) ? fabs(max) : fabs(min);
    min = -max;
  }
  imax = realToInt (max, paramPrecision);
  qmax = intToReal (imax, paramPrecision);
  // Make sure qmax >= max and -qmax <= min
  while (qmax < max)
    qmax = intToReal (++imax, paramPrecision);

  if (zeroCenter) {
    imin = -imax;
    qmin = -qmax;
  } else {
    imin = realToInt (min, paramPrecision);
    qmin = intToReal (imin, paramPrecision);
    // Make sure qmin <= min
    while (qmin > min)
      qmin = intToReal (--imin, paramPrecision);
  }

  imean = realToInt (mean, paramPrecision);
  qmean = intToReal (imean, paramPrecision);

  initialDist = 0;
  if (zeroCenter)
    for (int i = 0; i < nData; i++)
      initialDist += (*err)(data[i]);
  else
    for (int i = 0; i < nData; i++)
      initialDist += (*err)(data[i] - qmean);
}

/*-----------------------------------------------------------------------*/
void UniformQuant::setDataDecode (Real *newData, int newNData,
                                  int imax, int imin, int imean)
{
  data = newData;
  nData = newNData;

  if (imin < imax) {
    qmax = intToReal (imin, paramPrecision);

    if (zeroCenter) {
      qmin = -qmax;
    } else {
      qmin = intToReal (imin, paramPrecision);
    }
    qmean = intToReal (imean, paramPrecision);
  }
}

/*-----------------------------------------------------------------------*/
void UniformQuant::getRateDist (int precision, Real minStepSize,
                                Real &rate, Real &dist)
{
  if (precision > 0) {
    const int  nSteps = (1<<precision)-1;
    const Real stepSize = (qmax-qmin)/(Real)nSteps;
    const Real recipStepSize = 1.0/stepSize;

    if (stepSize < minStepSize) {
      rate = MaxReal;
      dist = MaxReal;
      return;
    }

    entropy->setNSym (nSteps);
    rate = dist = 0;

    for (int i = 0; i < nData; i++) {
```

```
      int symbol = (int)((data[i]-qmin)*recipStepSize);
      assert (symbol < nSteps && symbol >= 0);
      rate += entropy->cost (symbol, TRUE);
      Real reconstruct = qmin + ((Real)symbol + 0.5) * stepSize;
      dist += (*err) (data[i]-reconstruct);
    }
  } else {
    rate = dist = 0;
    if (zeroCenter) {
      for (int i = 0; i < nData; i++) {
        dist += (*err) (data[i]);
      }
    } else {
      for (int i = 0; i < nData; i++) {
        dist += (*err) (data[i] - qmean);
      }
    }
  }
}

/*---------------------------------------------------------------------------*/
void UniformQuant::quantize (Encoder *encoder, int precision)
{
  if (precision > 0) {
    const int  nSteps = (1<<precision)-1;
    const Real stepSize = (qmax-qmin)/(Real)nSteps;
    const Real recipStepSize = 1.0/stepSize;

    entropy->setNSym (nSteps);

    for (int i = 0; i < nData; i++) {
      int symbol = (int)((data[i]-qmin)*recipStepSize);
      assert (symbol < nSteps && symbol >= 0);
      entropy->write (encoder, symbol, TRUE);
    }
  }
}

/*---------------------------------------------------------------------------*/
void UniformQuant::dequantize (Decoder *decoder, int precision)
{
  if (precision > 0) {
    const int  nSteps = (1<<precision)-1;
    const Real stepSize = (qmax-qmin)/(Real)nSteps;
    int symbol;

    entropy->setNSym (nSteps);

    for (int i = 0; i < nData; i++) {
      symbol = entropy->read (decoder, TRUE);
      assert (symbol < nSteps && symbol >= 0);
      data[i] = qmin + ((Real)symbol + 0.5) * stepSize;
    }
  } else {
    for (int i = 0; i < nData; i++) {
      data[i] = qmean;
    }
  }
}

/*---------------------------------------------------------------------------*/
void UniformQuant::writeHeader (Encoder *encoder, int precision)
{
  encoder->writeNonneg (precision);

  if (precision > 0) {
    encoder->writeInt (imax);
    if (!zeroCenter)
      encoder->writeInt (imin);
  } else {
    if (!zeroCenter)
      encoder->writeInt (imean);
  }
}

/*---------------------------------------------------------------------------*/
void UniformQuant::readHeader  (Decoder *decoder, int &precision)
{
  precision = decoder->readNonneg ();

  if (precision > 0) {
    imax = decoder->readInt ();
    qmax = intToReal (imax, paramPrecision);

    if (zeroCenter) {
      qmin = -qmax;
    } else {
      imin = decoder->readInt ();
      qmin = intToReal (imin, paramPrecision);
    }
    qmean = 0;
  } else {
    if (!zeroCenter) {
      imean = decoder->readInt ();
      qmean = intToReal (imean, paramPrecision);
    } else {
      qmean = 0;
      imean = realToInt (qmean, paramPrecision);
    }
    qmax = qmin = qmean;
  }
}

/*---------------------------------------------------------------------------*/
void UniformQuant::setParams (int newParamPrecision, Real newMax,
                              Real newMin, Real newMean)
{
  paramPrecision = newParamPrecision;
  qmax = newMax;
  qmin = newMin;
  qmean = newMean;
}

/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/
LayerQuant::LayerQuant  (MultiLayerCoder *entropy, int paramPrecision,
                         int signedSym, int nLayers, ErrorMetric *err) :
  Quantizer (err), entropy (entropy), paramPrecision (paramPrecision),
  signedSym (signedSym), nLayers (nLayers)
{
  currentLayer = -1;
  layerRate = new Real [nLayers];
  layerDist = new Real [nLayers];
  context = NULL;
  residual = NULL;
}
```

```
/*---------------------------------------------------------------------------*/
LayerQuant::~LayerQuant ()
{
  delete [] layerRate;
  delete [] layerDist;
  if (context != NULL)
    delete [] context;
  if (residual != NULL)
    delete [] residual;
}

/*---------------------------------------------------------------------------*/
void LayerQuant::setDataEncode  (Real *newData, int newNData)
{
  data = newData;
  nData = newNData;

  getStats ();

  if (signedSym) {
    max = fabs(max) > fabs(min) ? fabs(max) : fabs(min);
    min = -max;
  }
  imax = realToInt (max, paramPrecision);
  qmax = intToReal (imax, paramPrecision);
  // Make sure qmax >= max and -qmax <= min
  while (qmax < max)
    qmax = intToReal (++imax, paramPrecision);

  if (signedSym) {
    imin = -imax;
    qmin = -qmax;
  } else {
    imin = realToInt (min, paramPrecision);
    qmin = intToReal (imin, paramPrecision);
    // Make sure qmin <= min
    while (qmin > min)
      qmin = intToReal (--imin, paramPrecision);
  }
  if (signedSym)
    threshold = qmax/2.0;
  else
    threshold = (qmax - qmin)/2.0;

  currentLayer = -1;
  if (context != NULL)
    delete [] context;
  if (residual != NULL)
    delete [] residual;
  context = new int [nData];
  residual = new Real [nData];

  resetLayer ();

  initialDist = 0;
  for (int i = 0; i < nData; i++)
    initialDist += (*err)(residual[i]);

  entropy->reset ();
}

/*---------------------------------------------------------------------------*/
void LayerQuant::setDataDecode  (Real *newData, int newNData, int imax,
                                 int imin, int imean)
{
  data = newData;
  nData = newNData;

  if (imin < imax) {
    qmax = intToReal (imin, paramPrecision);
    qmean = intToReal (imean, paramPrecision);

    if (signedSym) {
      qmin = -qmax;
      threshold = qmax/2.0;
    } else {
      qmin = intToReal (imin, paramPrecision);
      threshold = (qmax - qmin)/2.0;
    }

  }

  currentLayer = -1;
  if (context != NULL)
    delete [] context;
  context = new int [nData];
  if (signedSym) {
    for (int i = 0; i < nData; i++) {
      data[i] = 0;
      context[i] = 0;
    }
  } else {
    for (int i = 0; i < nData; i++) {
      data[i] = qmean;
      context[i] = 0;
    }
  }

  resetLayer ();
  entropy->reset ();
}

/*---------------------------------------------------------------------------*/
void LayerQuant::getRateDist (int precision, Real minStepSize,
                              Real &rate, Real &dist)
{
  assert (precision <= nLayers);

  Real currentRate = 0;
  Real currentDist = initialDist;

  for (int i = 0; i < precision; i++) {
    // rates & distortions have been computed for layers up to currentLayer
    if (i <= currentLayer) {
      currentRate += layerRate[i];
      currentDist += layerDist[i];
    } else {
      if (threshold > minStepSize) {
        quantizeLayer (NULL);
        currentRate += layerRate[i];
        currentDist += layerDist[i];
      } else {
        layerRate[i] = MaxReal;
        layerDist[i] = -MaxReal;
        currentRate = MaxReal;
```

```
        currentDist = MaxReal;
      }
    }
  }
  rate = currentRate;
  dist = currentDist;
}

/*---------------------------------------------------------------------------*/
void LayerQuant::quantize (Encoder *encoder, int precision)
{
  resetLayer ();
  entropy->reset ();

  for (int i = 0; i < precision; i++) {
    quantizeLayer (encoder);
  }
}

/*---------------------------------------------------------------------------*/
void LayerQuant::resetLayer ()
{
  if (residual != NULL) {
    if (signedSym) {
      for (int i = 0 ;i < nData; i++) {
        residual[i] = data[i];
        context[i] = 0;
      }
    } else {
      // on the first layer we remove the mean
      for (int i = 0 ;i < nData; i++) {
        residual[i] = data[i] - 0.5*(qmax+qmin);
        context[i] = 0;
      }
    }
  } else {
    if (signedSym) {
      for (int i = 0 ;i < nData; i++) {
        context[i] = 0;
      }
    } else {
      // on the first layer we remove the mean
      for (int i = 0 ;i < nData; i++) {
        context[i] = 0;
      }
    }
  }
  currentLayer = -1;

  if (signedSym)
    threshold = qmax/2.0;
  else
    threshold = (qmax - qmin)/2.0;
}

/*---------------------------------------------------------------------------*/
// deltaRate = bits required to code current layer
// deltaDist = reduction in distortion from current layer

void LayerQuant::quantizeLayer (Encoder *encoder)
{
  const Real halfThreshold = 0.5 * threshold;
  const Real threeHalvesThreshold = 1.5 * threshold;
  Real deltaRate = 0, deltaDist = 0;
  int symbol;

  currentLayer++;

  //  printf ("current layer = %d, threshold = %g\n", currentLayer, threshold);
  if (signedSym) {
    for (int i = 0; i < nData; i++) {
      deltaDist -= (*err)(residual[i]);  // subtract off old error
      //      Real oldResid = residual[i];
      if (context[i] == 0) {
        if (residual[i] > threshold) {
          symbol = 1;
          residual[i] -= threeHalvesThreshold;
        } else if (residual[i] < -threshold) {
          symbol = -1;
          residual[i] += threeHalvesThreshold;
        } else {
          symbol = 0;
        }
      } else {
        if (residual[i] > 0) {
          symbol = 1;
          residual[i] -= halfThreshold;
        } else {
          symbol = 0;
          residual[i] += halfThreshold;
        }
      }


      deltaDist += (*err)(residual[i]);  // add in new error
      deltaRate += entropy->write (encoder, symbol, TRUE, currentLayer,
                                   context[i]);
      context[i] = 2*context[i] + symbol;
    }

  } else {
    for (int i = 0; i < nData; i++) {
      deltaDist -= (*err)(residual[i]);  // subtract off old error

      if (residual[i] > 0) {
        symbol = 1;
        residual[i] -= halfThreshold;
      } else {
        symbol = 0;
        residual[i] += halfThreshold;
      }

      deltaDist += (*err)(residual[i]);  // add in new error
      deltaRate += entropy->write (encoder, symbol, TRUE, currentLayer,
                                   context[i]);
      context[i] = 2*context[i] + symbol;
    }
  }

  threshold *= 0.5;

  layerRate[currentLayer] = deltaRate;
  layerDist[currentLayer] = deltaDist;
}

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

```
void LayerQuant::dequantize   (Decoder *decoder, int precision)
{
  resetLayer ();

  for (int i = 0; i < precision; i++) {
    dequantizeLayer (decoder);
  }
}

/*---------------------------------------------------------------------------*/
void LayerQuant::dequantizeLayer (Decoder *decoder)
{
  int symbol;

  currentLayer++;


  if (signedSym) {
    for (int i = 0; i < nData; i++) {
      symbol = entropy->read (decoder, TRUE, currentLayer,
                              context[i]);

      //      int oldData = data[i];
      if (context[i] == 0)
        data[i] += 1.5*threshold * symbol;
      else
        data[i] += (symbol - 0.5) * threshold;


      context[i] = 2*context[i] + symbol;
    }
  } else {
    for (int i = 0; i < nData; i++) {
      symbol = entropy->read (decoder, TRUE, currentLayer, context[i]);

      data[i] += (symbol - 0.5) * threshold;
      context[i] = 2*context[i] + symbol;
    }
  }

  threshold *= 0.5;
}

/*---------------------------------------------------------------------------*/
void LayerQuant::writeHeader (Encoder *encoder, int precision)
{
  encoder->writeNonneg (precision);

  if (precision > 0) {
    encoder->writeInt (imax);
    if (!signedSym)
      encoder->writeInt (imin);
  } else {
    if (!signedSym)
      encoder->writeInt (imean);
  }



/*---------------------------------------------------------------------------*/
void LayerQuant::readHeader  (Decoder *decoder, int &precision)
{
  precision = decoder->readNonneg ();


  if (precision > 0) {
    imax = decoder->readInt ();
    qmax = intToReal (imax, paramPrecision);

    if (signedSym) {
      qmin = -qmax;
      qmean = 0;
    } else {
      imin = decoder->readInt ();
      qmin = intToReal (imin, paramPrecision);
      qmean = 0.5 * (qmax + qmin);
    }
  } else {
    if (!signedSym) {
      imean = decoder->readInt ();
      qmean = intToReal (imean, paramPrecision);
    } else {
      qmean = 0;
      imean = realToInt (qmean, paramPrecision);
    }
    qmax = qmin = qmean;
  }

}

/*---------------------------------------------------------------------------*/
void LayerQuant::setParams (int newParamPrecision, Real newMax,
                            Real newMin, Real newMean)
{
  paramPrecision = newParamPrecision;
  qmax = newMax;
  qmin = newMin;
  qmean = newMean;
}

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

```
/*------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <iostream.h>
#include "global.hh"
#include "image.hh"
/*------------------------------------------------------------------------*/

int main (int argc, char **argv)
{
   Image *image;

   char *program = argv[0];

   if (argc != 5)  {
      fprintf (stderr,
               "Convert an image in pbm/pgm format to raw pixel format\n");
      fprintf (stderr,
               "Usage: %s [raw image name][height][width][pgm image name]\n",
               program);

      return 1;
   }

   int hsize = atoi(argv[3]);
   int vsize = atoi(argv[2]);
   image = new Image (argv[1], hsize, vsize);

   image->savePGM (argv[4]);
   return 0;
}

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

```
/*-----------------------------------------------------------------------*/
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include "global.hh"
#include "image.hh"
#include "wavelet.hh"
/*-----------------------------------------------------------------------*/
/*-----------------------------------------------------------------------*/

Wavelet::Wavelet (FilterSet *filterset)
{
  analysisLow = filterset->analysisLow;
  analysisHigh = filterset->analysisHigh;
  synthesisLow = filterset->synthesisLow;
  synthesisHigh = filterset->synthesisHigh;
  symmetric = filterset->symmetric;

  // amount of space to leave for padding vectors for symmetric extensions
  npad = max(analysisLow->size, analysisHigh->size);
}

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

Wavelet::~Wavelet ()
{
}

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

void Wavelet::transform1d (Real *input, Real *output, int size,
                           int nsteps, int sym_ext)
{
   int i;
   int currentIndex = 0;
   Real *data[2];
   int lowSize = size, highSize;


   if (sym_ext == -1)
     sym_ext = symmetric;


   data [0] = new Real [2*npad+size];
   data [1] = new Real [2*npad+size];

   for (i = 0; i < size; i++)
     data[currentIndex][npad+i] = input[i];

   while (nsteps--)  {
     if (lowSize <= 2 && symmetric == 1) {
       warning ("Reduce # of transform steps or increase signal size");
       warning ("  or switch to periodic extension");
       error ("Low pass subband is too small");
     }

     // Transform
     printf ("transforming, size = %d\n", lowSize);
     transform_step (data[currentIndex], data[1-currentIndex],
                     lowSize, sym_ext);

     highSize = lowSize/2;
     lowSize = (lowSize+1)/2;

     // Copy high-pass data to output signal
     copy (data[1-currentIndex] + npad + lowSize, output +
           lowSize, highSize);

     for (i = 0; i < lowSize+highSize; i++)
       printf ("%5.2f ", data[1-currentIndex][npad+i]);
     printf ("\n\n");

     // Now pass low-pass data (first 1/2 of signal) back to
     // transform routine
     currentIndex = 1 - currentIndex;
   }

   // Copy low-pass data to output signal
   copy (data[currentIndex] + npad, output, lowSize);

   delete [] data [1];
   delete [] data [0];
}

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

void Wavelet::invert1d (Real *input, Real *output, int size,
                        int nsteps, int sym_ext)
{
   int i;
   int currentIndex = 0;
   Real *data[2];


   if (sym_ext == -1)
     sym_ext = symmetric;

   int *lowSize = new int [nsteps];
   int *highSize = new int [nsteps];

   lowSize[0] = (size+1)/2;
   highSize[0] = size/2;

   for (i = 1; i < nsteps; i++) {
     lowSize[i] = (lowSize[i-1]+1)/2;
     highSize[i] = lowSize[i-1]/2;
   }

   data [0] = new Real [2*npad+size];
   data [1] = new Real [2*npad+size];

   copy (input, data[currentIndex]+npad, lowSize[nsteps-1]);

   while (nsteps--)  {

     // grab the next high-pass component
     copy (input + lowSize[nsteps],
           data[currentIndex]+npad+lowSize[nsteps], highSize[nsteps]);

     // Combine low-pass data (first 1/2^n of signal) with high-pass
     // data (next 1/2^n of signal) to get higher resolution low-pass data
     invert_step (data[currentIndex], data[1-currentIndex],
                  lowSize[nsteps]+highSize[nsteps], sym_ext);
```

```
      // Now pass low-pass data (first 1/2 of signal) back to
      // transform routine
      currentIndex = 1 - currentIndex;
    }

    // Copy inverted signal to output signal
    copy (data[currentIndex]+npad, output, size);

    delete [] highSize;
    delete [] lowSize;

    delete [] data [1];
    delete [] data [0];
}

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

void Wavelet::transform2d (Real *input, Real *output, int hsize, int vsize,
                           int nsteps, int sym_ext)
{
    int j;
    int hLowSize = hsize, hHighSize;
    int vLowSize = vsize, vHighSize;


    if (sym_ext == -1)
      sym_ext = symmetric;

    Real *temp_in = new Real [2*npad+max(hsize,vsize)];
    Real *temp_out = new Real [2*npad+max(hsize,vsize)];

    copy (input, output, hsize*vsize);

    while (nsteps--)  {
       if ((hLowSize <= 2 || vLowSize <= 2) && sym_ext == 1) {
         warning ("Reduce # of transform steps or increase signal size");
         warning ("  or switch to periodic extension");
         error ("Low pass subband is too small");
       }


       for (j = 0; j < vLowSize; j++)  {
          // Copy row j to data array
          copy (output+(j*hsize), temp_in+npad, hLowSize);


          transform_step (temp_in, temp_out, hLowSize, sym_ext);

          // Copy back to image
          copy (temp_out+npad, output+(j*hsize), hLowSize);
       }


       for (j = 0; j < hLowSize; j++)  {
          // Copy column j to data array
          copy (output+j, hsize, temp_in+npad, vLowSize);

          // Convolve with low and high pass filters
          transform_step (temp_in, temp_out, vLowSize, sym_ext);

          // Copy back to image
          copy (temp_out+npad, output+j, hsize, vLowSize);
       }

       // Now convolve low-pass portion again
       hHighSize = hLowSize/2;
       hLowSize = (hLowSize+1)/2;
       vHighSize = vLowSize/2;
       vLowSize = (vLowSize+1)/2;
    }

    delete [] temp_out;
    delete [] temp_in;
}

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

void Wavelet::invert2d (Real *input, Real *output, int hsize, int vsize,
                        int nsteps, int sym_ext)
{
    int i, j;


    if (sym_ext == -1)
      sym_ext = symmetric;

    int *hLowSize = new int [nsteps],
        *hHighSize = new int [nsteps];
    int *vLowSize = new int [nsteps],
        *vHighSize = new int [nsteps];

    hLowSize[0] = (hsize+1)/2;
    hHighSize[0] = hsize/2;
    vLowSize[0] = (vsize+1)/2;
    vHighSize[0] = vsize/2;

    for (i = 1; i < nsteps; i++) {
      hLowSize[i] = (hLowSize[i-1]+1)/2;
      hHighSize[i] = hLowSize[i-1]/2;
      vLowSize[i] = (vLowSize[i-1]+1)/2;
      vHighSize[i] = vLowSize[i-1]/2;
    }

    Real *temp_in = new Real [2*npad+max(hsize,vsize)];
    Real *temp_out = new Real [2*npad+max(hsize,vsize)];

    copy (input, output, hsize*vsize);

    while (nsteps--)  {
       // Do a reconstruction for each of the columns
       for (j = 0; j < hLowSize[nsteps]+hHighSize[nsteps]; j++)  {
          // Copy column j to data array
          copy (output+j, hsize, temp_in+npad,
                vLowSize[nsteps]+vHighSize[nsteps]);

          // Combine low-pass data (first 1/2^n of signal) with high-pass
          // data (next 1/2^n of signal) to get higher resolution low-pass data
          invert_step (temp_in, temp_out,
                       vLowSize[nsteps]+vHighSize[nsteps], sym_ext);

          // Copy back to image
          copy (temp_out+npad, output+j, hsize,
                vLowSize[nsteps]+vHighSize[nsteps]);
       }
```

```
      // Now do a reconstruction pass for each row
      for (j = 0; j < vLowSize[nsteps]+vHighSize[nsteps]; j++)  {
         // Copy row j to data array
         copy (output + (j*hsize), temp_in+npad,
               hLowSize[nsteps]+hHighSize[nsteps]);

         // Combine low-pass data (first 1/2^n of signal) with high-pass
         // data (next 1/2^n of signal) to get higher resolution low-pass data
         invert_step (temp_in, temp_out,
                      hLowSize[nsteps]+hHighSize[nsteps], sym_ext);

         // Copy back to image
         copy (temp_out+npad, output + (j*hsize),
               hLowSize[nsteps]+hHighSize[nsteps]);
      }
   }

   delete [] hLowSize;
   delete [] hHighSize;
   delete [] vLowSize;
   delete [] vHighSize;

   delete [] temp_in;
   delete [] temp_out;
}

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

void Wavelet::transform_step (Real *input, Real *output, int size,
                              int sym_ext)
{
  int i, j;

  int lowSize = (size+1)/2;
  int left_ext, right_ext;

  if (analysisLow->size %2) {
    // odd filter length
    left_ext = right_ext = 1;
  } else {
    left_ext = right_ext = 2;
  }

  if (sym_ext)
    symmetric_extension (input, size, left_ext, right_ext, 1);
  else
    periodic_extension (input, size);

  //                      coarse  detail
  // xxxxxxxxxxxxxxxx --> HHHHHHHHGGGGGGGG
  for (i = 0; i < lowSize; i++)  {
    output[npad+i] = 0.0;
    for (j = 0; j < analysisLow->size; j++)  {
      output [npad+i] +=
        input[npad + 2*i + analysisLow->firstIndex + j] *
        analysisLow->coeff[j];
    }
  }

  for (i = lowSize; i < size; i++)  {
    output[npad+i] = 0.0;
    for (j = 0; j < analysisHigh->size; j++)  {
      output [npad+i] +=
        input[npad + 2*(i-lowSize) + analysisHigh->firstIndex + j] *
        analysisHigh->coeff[j];
    }
  }
}

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

void Wavelet::invert_step (Real *input, Real *output, int size, int sym_ext)
{
   int i, j;
   int left_ext, right_ext, symmetry;
   // amount of low and high pass -- if odd # of values, extra will be
   //   low pass
   int lowSize = (size+1)/2, highSize = size/2;

   symmetry = 1;
   if (analysisLow->size % 2 == 0) {
     // even length filter -- do (2, X) extension
     left_ext = 2;
   } else {
     // odd length filter -- do (1, X) extension
     left_ext = 1;
   }

   if (size % 2 == 0) {
     // even length signal -- do (X, 2) extension
     right_ext = 2;
   } else {
     // odd length signal -- do (X, 1) extension
     right_ext = 1;
   }

   Real *temp = new Real [2*npad+lowSize];
   for (i = 0; i < lowSize; i++) {
     temp[npad+i] = input[npad+i];
   }

   if (sym_ext)
     symmetric_extension (temp, lowSize, left_ext, right_ext, symmetry);
   else
     periodic_extension (temp, lowSize);

   // coarse  detail
   // HHHHHHHHGGGGGGGG --> xxxxxxxxxxxxxxxx
   for (i = 0; i < 2*npad+size; i++)
     output[i] = 0.0;

   int firstIndex = synthesisLow->firstIndex;
   int lastIndex = synthesisLow->size - 1 + firstIndex;

   for (i = -lastIndex/2; i <= (size-1-firstIndex)/2; i++)  {
      for (j = 0; j < synthesisLow->size; j++)  {
        output[npad + 2*i + firstIndex + j] +=
          temp[npad+i] * synthesisLow->coeff[j];
      }
   }

   left_ext = 2;

   if (analysisLow->size % 2 == 0) {
```

```
    // even length filters
    right_ext = (size % 2 == 0) ? 2 : 1;
    symmetry = -1;
  } else {
    // odd length filters
    right_ext = (size % 2 == 0) ? 1 : 2;
    symmetry = 1;
  }

  for (i = 0; i < highSize; i++) {
    temp[npad+i] = input[npad+lowSize+i];
  }
  if (sym_ext)
    symmetric_extension (temp, highSize, left_ext, right_ext,
                         symmetry);
  else
    periodic_extension (temp, highSize);


  firstIndex = synthesisHigh->firstIndex;
  lastIndex = synthesisHigh->size - 1 + firstIndex;

  for (i = -lastIndex/2; i <= (size-1-firstIndex)/2; i++)  {
     for (j = 0; j < synthesisHigh->size; j++)  {
       output[npad + 2*i + firstIndex + j] +=
         temp[npad+i] * synthesisHigh->coeff[j];
     }
  }

  delete [] temp;
}

/

void Wavelet::symmetric_extension (Real *output, int size, int left_ext, int
                                   right_ext, int symmetry)
{
  int i;
  int first = npad, last = npad + size-1;

  if (symmetry == -1) {
    if (left_ext == 1)
      output[--first] = 0;
    if (right_ext == 1)
      output[++last] = 0;
  }
  int originalFirst = first;
  int originalLast = last;
  int originalSize = originalLast-originalFirst+1;

  int period = 2 * (last - first + 1) - (left_ext == 1) - (right_ext == 1);

  if (left_ext == 2)
    output[--first] = symmetry*output[originalFirst];
  if (right_ext == 2)
    output[++last] = symmetry*output[originalLast];


  int nextend = min (originalSize-2, first);
  for (i = 0; i < nextend; i++) {
    output[--first] = symmetry*output[originalFirst+1+i];
  }

  while (first > 0) {
    first--;
    output[first] = output[first+period];
  }

  nextend = min (originalSize-2, 2*npad+size-1 - last);
  for (i = 0; i < nextend; i++) {
    output[++last] = symmetry*output[originalLast-1-i];
  }


  while (last < 2*npad+size-1) {
    last++;
    output[last] = output[last-period];
  }
}


void Wavelet::periodic_extension (Real *output, int size)
{
  int first = npad, last = npad + size-1;


  while (first > 0) {
    first--;
    output[first] = output[first+size];
  }


  while (last < 2*npad+size-1) {
    last++;
    output[last] = output[last-size];
  }
}

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

```
/*---------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "trans.hh"
/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/

WaveletTransform::WaveletTransform  (Wavelet *wavelet, Image *image,
                                     int nsteps, int symmetric) :
  wavelet (wavelet), nsteps(nsteps), symmetric(symmetric)
{
  value = NULL;

  if (image != NULL) {
    hsize = image->hsize;
    vsize = image->vsize;
    transform (image, wavelet, nsteps, symmetric);
  } else {
    hsize = vsize = 0;
  }
}

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

WaveletTransform::WaveletTransform (Wavelet *wavelet, int hsize, int
                                    vsize, int nsteps, int symmetric):
                         hsize(hsize), vsize(vsize), wavelet
                         (wavelet), nsteps(nsteps), symmetric(symmetric)
{
  nsteps = 0;
  symmetric = -1;
  init ();
  for (int i = 0; i < hsize*vsize; i++)
    value[i] = 0;
}

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

WaveletTransform::WaveletTransform (const WaveletTransform &t)
{
  wavelet = t.wavelet;
  hsize = t.hsize;
  vsize = t.vsize;
  nsteps = t.nsteps;
  symmetric = t.symmetric;

  if (t.value == NULL) {
    value = NULL;
  } else {
    init ();
    for (int i = 0; i < hsize*vsize; i++)
      value[i] = t.value[i];
  }
}

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

WaveletTransform::~WaveletTransform ()
{
   freeAll ();
}
/*---------------------------------------------------------------------------*/

void WaveletTransform::init ()
{
   int i;

   value = new Real [hsize*vsize];

   nSubbands = 3 * nsteps + 1;
   subbandSize = new int [nSubbands];
   subbandHsize = new int [nSubbands];
   subbandVsize = new int [nSubbands];
   subbandPtr = new Real* [nSubbands];

   int *lowHsize = new int [nsteps];
   int *lowVsize = new int [nsteps];
   int *highHsize = new int [nsteps];
   int *highVsize = new int [nsteps];

   lowHsize[nsteps-1] = (hsize+1)/2;
   lowVsize[nsteps-1] = (vsize+1)/2;
   highHsize[nsteps-1] = hsize/2;
   highVsize[nsteps-1] = vsize/2;

   for (i = nsteps-2; i >= 0; i--) {
     lowHsize[i] = (lowHsize[i+1]+1)/2;
     lowVsize[i] = (lowVsize[i+1]+1)/2;
     highHsize[i] = lowHsize[i+1]/2;
     highVsize[i] = lowVsize[i+1]/2;
   }

   subbandPtr[0] = value;
   subbandHsize[0] = lowHsize[0];
   subbandVsize[0] = lowVsize[0];
   subbandSize[0] = subbandHsize[0]*subbandVsize[0];

   for (i = 0; i < nsteps; i++) {
     subbandHsize[3*i+1] = highHsize[i];
     subbandVsize[3*i+1] = lowVsize[i];
     subbandHsize[3*i+2] = lowHsize[i];
     subbandVsize[3*i+2] = highVsize[i];
     subbandHsize[3*i+3] = highHsize[i];
     subbandVsize[3*i+3] = highVsize[i];
   }

   for (i = 1; i < nSubbands; i++) {
     subbandSize[i] = subbandHsize[i]*subbandVsize[i];
     subbandPtr[i] = subbandPtr[i-1] + subbandSize[i-1];
   }

   delete [] lowHsize;
   delete [] lowVsize;
   delete [] highHsize;
   delete [] highVsize;
}

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

void WaveletTransform::freeAll ()
{
  if (value != NULL) {
    delete [] value;
    delete [] subbandSize;
```

```
    delete [] subbandHsize;
    delete [] subbandVsize;
    delete [] subbandPtr;
  }
}

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

void WaveletTransform::transform (Image *image, Wavelet *newWavelet,
                                  int steps, int isSymmetric)
{
  // clear out old info and set up subband pointers
  freeAll ();
  hsize = image->hsize;
  vsize = image->vsize;
  wavelet = newWavelet;
  nsteps = steps;
  symmetric = isSymmetric;
  init ();

  Real *temp = new Real [hsize*vsize];
  wavelet->transform2d (image->value, temp, hsize, vsize, nsteps,
                        symmetric);

  // linearize data
  mallatToLinear (temp);
  delete [] temp;
}

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

void WaveletTransform::invert (Image *invertedImage)
{
  Real *temp = new Real [hsize*vsize];

  // put data in Mallat format
  linearToMallat (temp);

  wavelet->invert2d (temp, invertedImage->value, hsize, vsize,
                     nsteps, symmetric);

  delete [] temp;
}

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

void WaveletTransform::mallatToLinear (Real *mallat)
{
  int i, j, k;

  int *lowHsize = new int [nsteps];
  int *lowVsize = new int [nsteps];

  lowHsize[nsteps-1] = (hsize+1)/2;
  lowVsize[nsteps-1] = (vsize+1)/2;

  for (i = nsteps-2; i >= 0; i--) {
    lowHsize[i] = (lowHsize[i+1]+1)/2;
    lowVsize[i] = (lowVsize[i+1]+1)/2;
  }

  // move transformed image (in Mallat order) into linear array structure
  // special case for LL subband
  for (j = 0; j < subbandVsize[0]; j++)
    for (i = 0; i < subbandHsize[0]; i++)
      subbandPtr[0][j*subbandHsize[0]+i] =
        mallat[j*hsize+i];

  for (k = 0; k < nsteps; k++) {
    for (j = 0; j < subbandVsize[k*3+1]; j++)
      for (i = 0; i < subbandHsize[k*3+1]; i++)
        subbandPtr[k*3+1][j*subbandHsize[k*3+1]+i] =
          mallat[j*hsize+(lowHsize[k]+i)];

    for (j = 0; j < subbandVsize[k*3+2]; j++)
      for (i = 0; i < subbandHsize[k*3+2]; i++)
        subbandPtr[k*3+2][j*subbandHsize[k*3+2]+i] =
          mallat[(lowVsize[k]+j)*hsize+i];

    for (j = 0; j < subbandVsize[k*3+3]; j++)
      for (i = 0; i < subbandHsize[k*3+3]; i++)
        subbandPtr[k*3+3][j*subbandHsize[k*3+3]+i] =
          mallat[(lowVsize[k]+j)*hsize+(lowHsize[k]+i)];
  }

  delete [] lowHsize;
  delete [] lowVsize;
}

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

void WaveletTransform::linearToMallat (Real *mallat)
{
  int i, j, k;

  int *lowHsize = new int [nsteps];
  int *lowVsize = new int [nsteps];

  lowHsize[nsteps-1] = (hsize+1)/2;
  lowVsize[nsteps-1] = (vsize+1)/2;

  for (i = nsteps-2; i >= 0; i--) {
    lowHsize[i] = (lowHsize[i+1]+1)/2;
    lowVsize[i] = (lowVsize[i+1]+1)/2;
  }

  // put linearized image in Mallat format
  // special case for LL subband
  for (j = 0; j < subbandVsize[0]; j++)
    for (i = 0; i < subbandHsize[0]; i++)
      mallat[j*hsize+i] = subbandPtr[0][j*subbandHsize[0]+i];

  for (k = 0; k < nsteps; k++) {
    for (j = 0; j < subbandVsize[k*3+1]; j++)
      for (i = 0; i < subbandHsize[k*3+1]; i++)
        mallat[j*hsize+(lowHsize[k]+i)] =
          subbandPtr[k*3+1][j*subbandHsize[k*3+1]+i];

    for (j = 0; j < subbandVsize[k*3+2]; j++)
      for (i = 0; i < subbandHsize[k*3+2]; i++)
        mallat[(lowVsize[k]+j)*hsize+i] =
          subbandPtr[k*3+2][j*subbandHsize[k*3+2]+i];

    for (j = 0; j < subbandVsize[k*3+3]; j++)
      for (i = 0; i < subbandHsize[k*3+3]; i++)
        mallat[(lowVsize[k]+j)*hsize+(lowHsize[k]+i)] =
```

```
        subbandPtr[k*3+3][j*subbandHsize[k*3+3]+i];
  }

  delete [] lowHsize;
  delete [] lowVsize;
}

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

```
/*--------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "global.hh"
#include "image.hh"
/*--------------------------------------------------------------------------*/
// Create a blank image with width=hsize, height=vsize
//    If hsize is unspecified, creates an image with width=0, height=0
//    If vsize is unspecified, creates a square image with width =
//       height = hsize

Image::Image  (int new_hsize, int new_vsize) : hsize(new_hsize),
    vsize(new_vsize)
{
  if (hsize == -1)
    hsize = vsize = 0;
  if (vsize == -1)
    vsize = hsize;

  value = new Real [hsize*vsize];
  if (value == NULL)
    error ("Can't allocate memory for image of size %d by %d\n",
           hsize, vsize);
}

/*--------------------------------------------------------------------------*/
// Copy constructor

Image::Image (const Image& image)
{
   int i;

   hsize = image.hsize;
   vsize = image.vsize;
   value = new Real [hsize*vsize];
   if (value == NULL)
     error ("Can't allocate memory for image of size %d by %d\n",
            hsize, vsize);

   for (i = 0; i < hsize*vsize; i++)
     value[i] = image.value[i];
}

/*--------------------------------------------------------------------------*/
// Loads a raw image of size hsize by vsize from the specified file
//    The file is assumed to contain an image in raw byte format

Image::Image (const char *filename, int new_hsize, int new_vsize) :
  hsize(new_hsize), vsize(new_vsize)
{
  if (hsize == -1)
    hsize = vsize = 0;
  if (vsize == -1)
    vsize = hsize;

  value = new Real [hsize*vsize];
  if (value == NULL)
    error ("Can't allocate memory for image of size %d by %d\n",
           hsize, vsize);

   loadRaw (filename);
}

/*--------------------------------------------------------------------------*/
// Loads a PGM image from a file. Sets hsize, vsize.

Image::Image (const char *filename)
{
  vsize = hsize = 0;
  value = NULL;
  loadPGM (filename);
}

/*--------------------------------------------------------------------------*/
// Destructor

Image::~Image ()
{
   hsize = vsize = -1;
   delete [] value;
}

/*--------------------------------------------------------------------------*/
// Assignment operator

Image &Image::operator= (const Image& image)
{
  delete [] value;
  hsize = image.hsize;
  vsize = image.vsize;
  value = new Real [hsize*vsize];

  for (int i = 0; i < hsize*vsize; i++)
    value[i] = image.value[i];

  return *this;
}

/*--------------------------------------------------------------------------*/
// Loads an image from the specified file.  The file is assumed to
//   contain an image in raw byte format of size hsize by vsize

void Image::loadRaw (const char *filename)
{
   FILE *infile;
   unsigned char *buffer;
   int i;

   infile = fopen (filename, "rb");
   if (infile == NULL)
      error ("Unable to open file %s\n", filename);

   buffer = new unsigned char [hsize * vsize];

   if (fread (buffer, hsize*vsize, sizeof(unsigned char), infile) != 1)
        error ("Read < %d chars when loading file %s\n", hsize*vsize, filename);

   for (i = 0; i < hsize*vsize; i++)
        value[i] = (Real)buffer[i];

   delete [] buffer;
   fclose (infile);
}

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

```
// Saves an image to the specified file.  The image is written in
//   raw byte format.

void Image::saveRaw (const char *filename)
{
  FILE *outfile;
  unsigned char *buffer;
  int i;

  outfile = fopen (filename, "wb+");
  if (outfile == NULL)
    error ("Unable to open file %s\n", filename);

  buffer = new unsigned char [hsize*vsize];

  for (i = 0; i < hsize*vsize; i++)
    buffer[i] = realToChar(value[i]);

  fwrite (buffer, hsize*vsize, 1, outfile);

  delete [] buffer;
  fclose (outfile);
}

/*---------------------------------------------------------------------------*/
// Private stuff to load pgms/ppms

void Image::PGMSkipComments (FILE* infile, unsigned char* ch)
{
  while ((*ch == '#')) {
    while (*ch != '\n') { *ch = fgetc(infile); }
    while (*ch <  ' ' ) { *ch = fgetc(infile); }
  }
} // Comment(s)

/*---------------------------------------------------------------------------*/
// Get a number from a pgm file header, skipping comments etc.

unsigned int Image::PGMGetVal (FILE* infile)
{
  unsigned int tmp;
  unsigned char ch;
  do { ch = fgetc(infile); } while ((ch <= ' ') && (ch != '#'));
  PGMSkipComments(infile, &ch);
  ungetc(ch, infile);
  if (fscanf(infile,"%u",&tmp) != 1) {
    printf("%s\n","Error parsing file!");
    exit(1);
  }
  return(tmp);
}

/*---------------------------------------------------------------------------*/
// Loads a binary (P5) PGM image from the specified file. Sets hsize and
// vsize to the correct values for the file.

void Image::loadPGM (const char *filename)
{
  FILE* infile;
  unsigned char ch = ' ';

  infile = fopen (filename, "rb");
  if (infile == NULL)
    error ("Unable to open file %s\n", filename);

  // Look for type indicator
  while ((ch != 'P') && (ch != '#')) { ch = fgetc(infile); }
  PGMSkipComments(infile, &ch);
  char ftype = fgetc(infile); // get type, 5 or 6

  // Look for x size, y size, max grey level
  int xsize = (int)PGMGetVal(infile);
  int ysize = (int)PGMGetVal(infile);
  int maxg = (int)PGMGetVal(infile);

  // Do some consistency checks

  if ( (hsize <= 0) && (vsize <= 0) ) {
    resize (xsize, ysize);
    if (value == NULL)
         error ("Can't allocate memory for image of size %d by %d\n",
             hsize, vsize);
  } else {
    if ((xsize != hsize) || (ysize != vsize)) {
      error ("File dimensions conflict with image settings\n");
    }
  }

  if (ftype == '5') {
    printf("File %s is of type PGM, is %d x %d with max gray level %d\n",
           filename, hsize, vsize, maxg);
    PGMLoadData(infile, filename);
  }
  if (ftype == '6') {
    printf("File %s is of type PPM, is %d x %d with max gray level %d\n",
           filename, hsize, vsize, maxg);
    error("Attempt to load a PPM as a PGM\n");
  }

  fclose(infile);
}

/*---------------------------------------------------------------------------*/
// Loads the data segment of a PGM image from the specified file.

void Image::PGMLoadData (FILE *infile, const char *filename)
{
  unsigned char *buffer;
  int i;

  buffer = new unsigned char [hsize * vsize];

  long fp = -1*hsize*vsize;
  fseek(infile, fp, SEEK_END);

  if (fread (buffer, hsize*vsize, sizeof(unsigned char), infile) != 1)
       error ("Read < %d chars when loading file %s\n", hsize*vsize, filename);

  for (i = 0; i < hsize*vsize; i++)
       value[i] = (Real)buffer[i];

  delete [] buffer;
}

/*---------------------------------------------------------------------------*/
// Saves an image to the specified file.  The image is written in
```