DEVELOPMENT OF AN ALGORITHM FOR WAVELET TRANSFORM IN A JPEG LIKE IMAGE CODER

A Dissertation Submitted in partial fulfillment of the requirements
for the award of the degree of

MASTER OF ENGINEERING
(Electronics and Communication)

By

Puneet Sabbarwal
College Roll no. 06/E&C/04
Delhi University Roll No. 8733

Under the guidance of

Mrs. S. Indu

Department of Electronics & Communication Engineering

Delhi College of Engineering

Bawana Road, Delhi-110042

(University of Delhi)

June-2006

CERTIFICATE

This is to certify that dissertation entitled **"DEVELOPMENT OF AN ALGORITHM FOR WAVELET TRANSFORM IN A JPEG LIKE IMAGE CODER"** which is submitted by **Puneet Sabbarwal** in partial fulfillment of the requirement for the award of degree M.E. in Electronics and Communication Engineering to Delhi College of Engineering, Delhi is a record of the candidate own work carried out by him under my supervision. The matter embodied in this thesis is original and has not been submitted for the award of any other degree

(Mrs. S.Indu)
Lecturer
Dept. of Electronics and Comm. Engg.
Delhi College of Engineering, Delhi

ACKNOWLEDGEMENTS

It is a great pleasure to have the opportunity to extent my heartiest felt gratitude to everybody who helped me throughout the course of this project.

I would like to express my heartiest felt regards to **Mrs.S.Indu,** Lecturer, Department of Electronics and Communication Engineering for the constant motivation and support during the duration of this project. It is my privilege and owner to have worked under her supervision. Her invaluable guidance and helpful discussions in every stage of this project really helped me in materializing this project. It is indeed difficult to put her contribution in few words.

I would also like to express my deep sense of gratitude towards my teacher **Prof. Asok Bhattacharyya,** Head of Department, Electronics and Comm. Engineering, Delhi College of Engineering He was always kind, cooperative & helped me whenever I needed his guidance. Inspite of his busy schedule, he always find time to provide precious guidance and encouragement. He gladly accepted all pains in going through my work and participated in enlightening and motivating discussions which are extremely useful.

I would also like to take this opportunity to present my sincere regards to my teachers viz. **Mrs. R.Pandey**, **Mr. R.Rohilla**, **Dr. M.Kulkarni** , **Mr. J.Panda** and **Mr. A.K.Singh** for their support and encouragement.

I am thankful to my friends and classmates for their unconditional support and motivation during this project.

Puneet Sabbarwal

M.E. (Electronics and Communication Engineering)

College Roll No. 06/E&C/04

Delhi University Roll No. 8733

ABSTRACT

Image compression is now essential for applications such as transmission of large sized images and their storage in databases. The fundamental goal of image compression is to reduce the number of bits for transmission or storage while maintaining an acceptable image quality. Compression can be achieved by transforming the data, projecting it on a basis function, and encoding this transform.

A new Discrete Wavelet Transform (DWT) based image compression algorithm is presented in this thesis whose structure is modeled on the lines of the "Baseline JPEG (Joint Picture Expert Group) standard". The JPEG algorithm that uses the Discrete Cosine Transform (DCT) yields good results for compression ratios till 10-15:1. As the compression ratio increases, the coarse quantization on the image blocks causes blocking artifacts in the decompressed image which is very annoying to the Human Visual System (HVS). The wavelet transform that has better decorrleating properties than the DCT operates on the complete image and hence it does not create any blocking artifacts and provides significant improvement in achieving compression.

Lifting scheme which is an easy, efficient and the latest advancement in the field of implementing conventional wavelets by the use of simple lifting steps has been used to implement the wavelet transform and the multiresolution decomposition of the images at different scales. This is followed by the quantization of the transformed coefficients. The transformed coefficients which are now at different resolution levels are quantized using block thresholding. The block thresholding levels are fixed such that the coefficients in the lowest resolution range are least quantized whereas the coefficients in the higher resolution range are quantized more severely. The final 3 blocks which contain the highest frequency components are completely quantized to a zero-level mark. Finally entropy encoding (Run length encoding and Huffman encoding) is done to achieve final compression. This algorithm produces compression results competitive with the DCT base Baseline JPEG model and requires no training,

no pre-stored tables or codebooks and requires no prior knowledge of the image source.

Results of this new coder are compared to JPEG standard, which shows a gain of 1dB to 5dB over the standard baseline JPEG algorithm and a better performance in the visual perceptibility of the reconstructed images.

Contents

Chapter 1

INTRODUCTION

An overview of Image Compression Concepts is presented in subsections 1.1 – 1.7 below. Thereafter the general introduction of the work done is presented in 1.8. The organization of the thesis is presented in 1.9.

1.1 Digital Image

An image may be defined as a two dimensional function, ***f (x, y)***, where x and y are spatial (plane) coordinates, and the amplitude of f at any pair of coordinates *(x, y)* is called the *intensity or gray level* of the image at that point. When x, y and the amplitude of ***f*** are all finite discrete quantities then we call the image a digital image.

1.2 Need of Compression

The figures in Table 1.1 show the qualitative transition from simple text to full-motion video data and the disk space, transmission bandwidth, and transmission time needed to store and transmit such uncompressed data. It clearly illustrates the need for sufficient storage space, large transmission bandwidth, and long transmission time for image, audio, and video data. At the present state of technology, the only solution is to compress multimedia data before its storage and transmission, and decompress it at the receiver for play back. For example, with compression ratio of 32:1, the space, bandwidth, and transmission time requirements can be reduced by a factor of 32, with acceptable quality.

Multimedia Data	Size/Duration	Bits/Pixel or Bits/Sample	Uncompressed Size
A page of text	11" x 8.5"	Varying resolution	16-32 Kbits
Telephone quality speech	1 sec	8 bps	64 Kbits
Grayscale Image	512 x 512	8 bpp	2.1 Mbits
Color Image	512 x 512	24 bpp	6.29 Mbits
Medical Image	2048 x 1680	12 bpp	41.3 Mbits
Full-motion Video	640 x 480, 10 sec	24 bpp	2.21 Gbits

TABLE 1.1 Multimedia data types and uncompressed storage space, transmission bandwidth, and transmission time required. The prefix kilo- denotes a factor of 1000 rather than 1024. [1]

1.3 Principles of Image Compression

A common characteristic of most images is that the neighboring pixels are correlated and therefore contain redundant information. The foremost task then is to find less correlated representation of the image. Two fundamental components of compression are redundancy and irrelevancy reduction. **Redundancy reduction** aims at removing duplication from the signal source (image/video). **Irrelevancy reduction** omits parts of the signal that will not be noticed by the signal receiver, namely the Human Visual System (HVS). In general, three types of redundancy can be identified:

- **Interpixel (Spatial) Redundancy** or correlation between neighboring pixel values.
- **Coding Redundancy** usage of more code symbols than needed.
- **Psychovisual redundancy** more information than HVS can process.

1.4 Performance Ratios

The performance of Compression achieved is measured by two significant ratios[2]. These ratios serve as an important tool for direct comparison with other coders.

i) Mean Square Error (MSE)

It is the Cumulative squared error between the compressed and the original image. It is usually represented by σ_d and is defined as

$$RMSE = \sqrt{\frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N}[x(n,m) - y(n,m)]^2} \qquad (1\text{-}1)$$

x_n is the source output, y_n is the reconstructed sequence.

MSE lower the better, means lesser error.

ii) Peak Signal to Noise Ratio (PSNR)

It is a measure of the peak error.

$$PSNR(dB) = 10\log_{10}\frac{x_{peak}^2}{\sigma_d^2} \qquad (1\text{-}2)$$

1.5 Typical Environment For Image Compression

Image compression research aims at reducing the number of bits needed to represent an image by removing the spatial and spectral redundancies as much as possible while maintaining and acceptable quality and intelligibility.

A typical environment for image compression is shown in Figure 1.1. The digital image is encoded by an image coder. The output of the image coder is a string of bits that represents the source image. The channel coder transforms string of bits to a form suitable for transmission over a communication channel. At the receiver, the received signal is transformed back into a string of bits by a channel decoder. The image decoder reconstructs the image from the string of bits. In contrast to the communication environment in Figure 1.1, no communication channel is involved in application of image coding for purpose of storage.

The image coder in Figure 1.1 consists of three closely connected components viz. (a) Source Encoder or Linear Transforms, (b) Quantizer, and (c) Entropy Encoder, shown in Figure 1.2. Compression is accomplished by applying a linear transform to decorrelate the image data, quantizing the resulting transform coefficients and entropy coding the quantized values.

1.5.1 source encoder

The source encoder is responsible for reducing or eliminating any coding, interpixel, or psychovisual redundancies in the input image. A variety of linear transforms are available like Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), Discrete Wavelet Transform (DWT) for this purpose. This operation is reversible and may or may not reduce directly the amount of data required to represent the image.

1.5.2 quantizer

A Quantizer reduces the precision of the values generated from the encoder and therefore reduces the number of bits required to save the transform co-coefficients. This process is lossy. Quantization can be performed on each

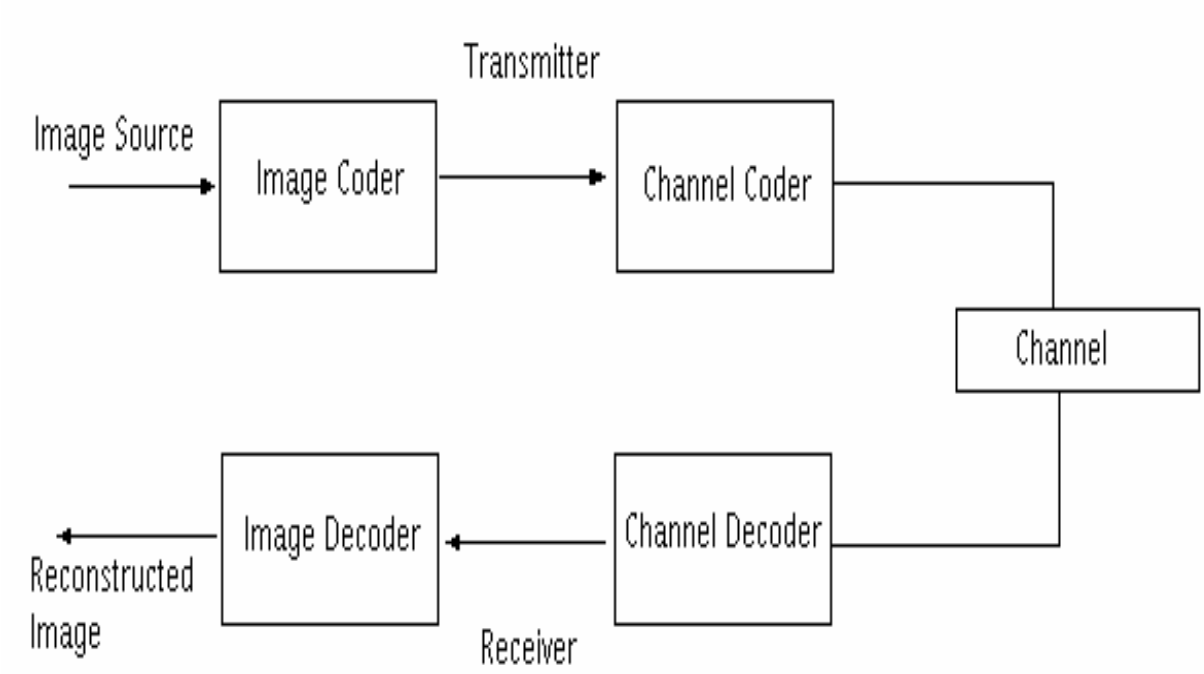


FIG. 1.1 Typical Environment for Image Coding

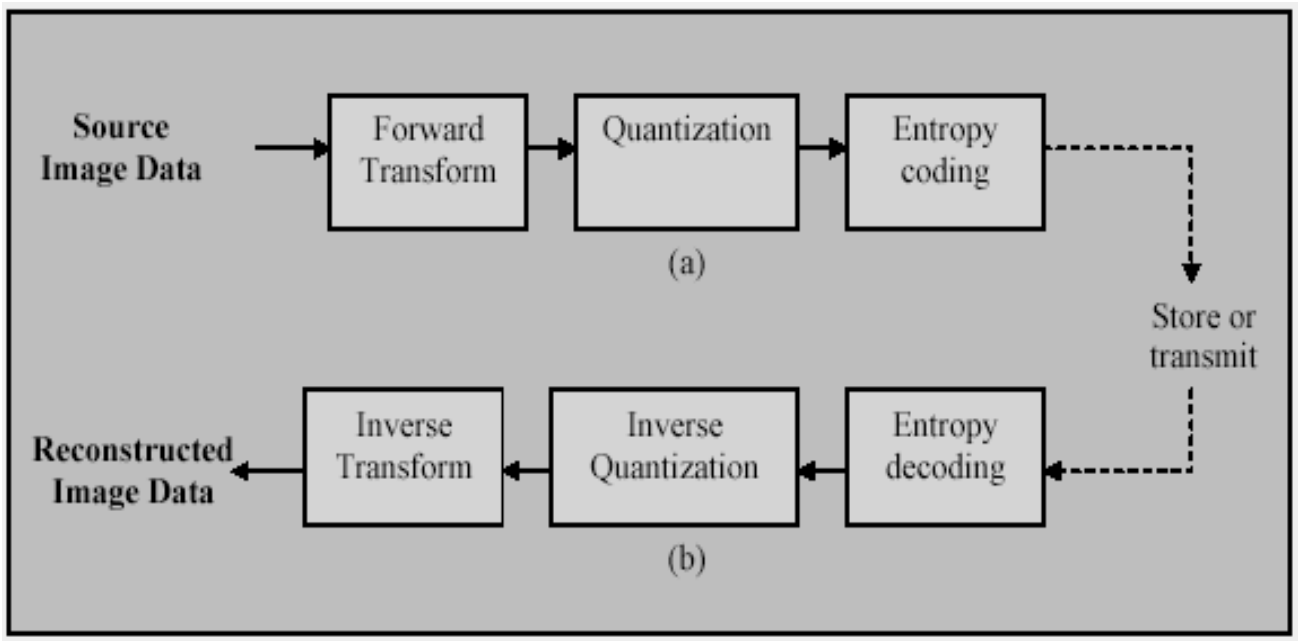


FIG. 1.2 Typical Structured Image Compression System

individual coefficient i.e. Scalar Quantization (SQ) or it can be performed on a group of coefficients together i.e. Vector Quantization (VQ).

1.5.3 Entropy Encoder

An entropy encoder does further compress the quantized values. This is done to achieve even better overall compression. The various commonly used entropy encoders are the Huffman encoder, arithmetic encoder, and simple run-length encoder. For better performance with compression, it's important to have the best of all the three components.

1.6 Image Compression Techniques

There are different schemes for classifying compression techniques. Two of these schemes, described in this report, are:

1.6.1 lossless Vs. lossy compression: The first categorization is based on the information content of the reconstructed image. They are 'lossless compression' and 'lossy compression' schemes. In lossless compression, the

reconstructed image after compression is numerically identical to the original image on a pixel-by-pixel basis. However, only a modest amount of compression is achievable in this technique. In lossy compression on the other hand, the reconstructed image contains degradation relative to the original, because redundant information is discarded during compression. As a result, much higher compression is achievable, and under normal viewing conditions, no visible loss is perceived (visually lossless). Performance analysis of these schemes is shown in Fig. 1.3 below.

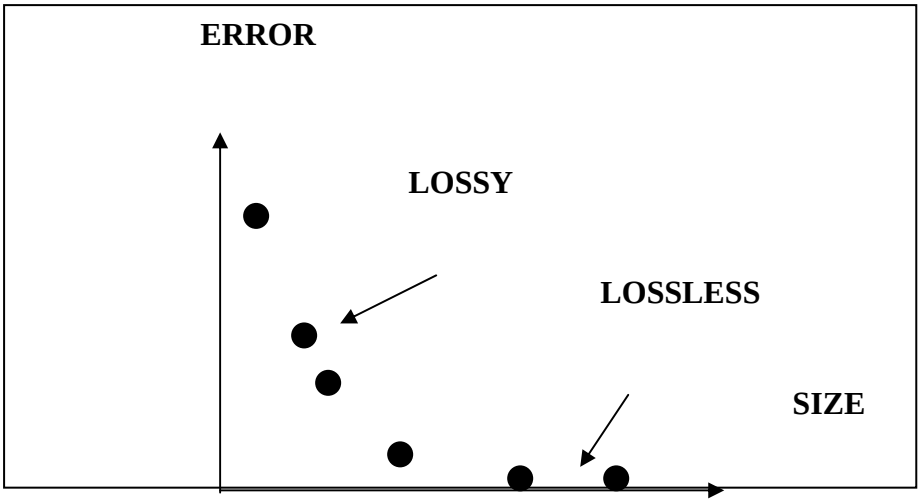


FIG. 1.3 Performance Analysis

1.6.2 predictive Vs. transform coding: The second categorization of various coding schemes is based on the 'space' where the compression method is applied. These are 'predictive coding' and 'transform coding'. In predictive coding, information already sent or available is used to predict future values, and the difference is coded. Since this is done in the image or spatial domain, it is relatively simple to implement and is readily adapted to local image characteristics.

Transform coding, on the other hand, first transforms the image from its spatial domain representation to a different type of representation using some well-known transforms mentioned later, and then codes the transformed values (coefficients). Some of these are: Karhunen-Loeve Transform (KLT), Discrete Fourier Transform (DFT), Discrete Cosine

Transform (DCT), Discrete Wavelet Transform (DWT). The primary advantage is that, it provides greater data compression compared to predictive methods, although at the expense of greater computations.

1.7 Image Compression Algorithm

After reviewing the various techniques used for development of image compression algorithms, a scheme is presented for the work done in this thesis.

Visual communication is becoming more and more important in applications such as digital television transmission, teleconferencing, multimedia communications, transmission and archival of remote sensing and medical images, video databases, educational and business documents, etc. Since digital images and video data are voluminous, efficient compression techniques are essential for their storage, retrieval and transmission.

Transform based image compression schemes first involve the transformation of spatial information in to another domain. For example, the Discrete Cosine Transform (DCT) transforms an image into the frequency domain. The goal of the transformation is a compact, complete representation of the image. The transform should decorrelate the spatially distributed energy into fewer data samples such that no information is lost. Orthogonal transforms have the feature of eliminating redundancy in the transformed image. Compression occurs in the second step when the transformed image is quantized (i.e. when some data samples usually those with insignificant energy-level are discarded). The inverse transform reconstructs the compressed image in the spatial domain. Since the quantization process is not invertible, the reconstruction cannot perfectly recreate the original image[3]. This type of compression is called lossy; Figure 1.2 shows a block diagram of a lossy compression scheme.

In transform based image compression, entropy coding typically follows the quantization stage. Entropy coding minimizes the redundancy in the bit stream and is fully invertible at the decoding end. So, it is lossless and usually gives about 0.4-0.6 dB gain in the PSNR [3].

Recently, discrete wavelet transform (DWT) has proven itself as a powerful and promising technique for image coding applications. Energy compaction is one of the most important properties for low bit rate transfer coding. Wavelets have better energy compaction property than the Fourier transform (DCT based JPEG) for signal with discontinuities. A pedagogically oriented image compression system using the modified version of the integer-lifting scheme by Cohen-Daubechies-Feauveau[4,5] is used to implement the wavelet transform and the multiresolution analysis. Lifting scheme is one of the latest advancements made in the field of wavelet research and is popularly known as the Second Generation Wavelets.The lifting scheme, developed by Sweldens, started as a method to improve a given Discrete Wavelet Transform (DWT) to obtain specific properties [4,6]. Later it became an efficient algorithm to calculate any wavelet transform as a sequence of simple lifting steps [5]. Digital signals are usually a sequence of integer numbers, while wavelet transforms result in floating point numbers. For an efficient reversible implementation it is of great importance to have a transform algorithm that converts integers to integers. Fortunately a lifting step can be modified to operate on integers, while preserving the reversibility [6]. Thus the lifting scheme became a method to implement reversible integer wavelet transforms. The advantages of lifting are numerous [4]:

1. It allows a faster ($\boldsymbol{O}$(n)) implementation of the wavelet transform. Traditionally, the fast wavelet transform is calculated with a two-band subband transform scheme. In each step the signal is split into a high pass and low pass band and then subsampled. Recursion occurs on the low pass band. The lifting scheme makes optimal use of similarities between the high and low pass filters to speed up the calculation. In some cases the number of operations can be reduced by a factor of two.
2. The lifting scheme allows a fully in-place calculation of the wavelet transform. In other words, no auxiliary memory is needed and the original signal (image) can be replaced with its wavelet transform.
3. In the classical case, it is not immediately clear that the inverse wavelet transform actually is the inverse of the forward transform. Only with the Fourier transform one can convince oneself of the perfect reconstruction property. With the lifting scheme, the inverse wavelet transform can immediately be found by undoing the operation of the forward transform. In

practice, this comes down to simply reversing the order of the operations and changing each + into a – and vice versa[4,6,7].

Based on this a software has been developed on the lines of baseline JPEG that uses Discrete Wavelet Transform instead of the Discrete Cosine Transform, quantization and entropy coding and hence provides image compression of two-dimensional grayscale images of size 512 X 512 with a gain in the PSNR values in the range of 1dB to 5dB over the standard baseline JPEG algorithm.

1.8 Organization of the Thesis

The present thesis is distributed in 5 chapters. Chapter 1 deals with the general introduction to the concepts of image compression and the algorithm presented. Chapter 2 describes the compression techniques using the different transformation schemes available. The popular DCT based JPEG, the lifting scheme for the wavelets have been detailed here. Chapter 3 describes the implementation of the image compression system developed in this thesis. The results are presented , analyzed and compared with some of the well known techniques of image compression in Chapter 4. Conclusions and issues identified for further research are presented in Chapter 5. References cited in the text of the thesis are presented at the end.

Chapter 2

TRANSFORM BASED IMAGE COMPRESSION

In transform image coding, an image is transformed into a domain significantly different form the image intensity domain. Transform coding technique attempt to reduce the correlation that exists among image pixel intensities. When the correlation is reduced, the redundant information does not have to be coded repeatedly. Transform coding techniques also exploit the observation that for typical images a large amount of energy is concentrated in a small fraction of the transform coefficients. This is called the energy compaction property. Because of this property, it is possible to code only a fraction of the transform coefficients without seriously affecting the image quality and intelligibility.

Many different transforms have been considered for transform image coding. They differ in energy compaction and in computational requirements. Some of them are:

1. Karhunen-Loeve Transform (KLT)
1. Discrete Fourier Transform (DFT)
2. Discrete Cosine Transform (DCT)
3. Discrete Wavelet Transform (DWT)

2.1 Karhunen-Loeve Transform

Karhunen-Loeve Transform (KLT) is the best of all linear transforms for energy compaction. In the KLT, the basis functions $a(n_1,n_2; k_1,k_2)$ are obtained within a scaling factor by solving

$$\lambda(k_1,k_2)\,a(n_1,n_2;k_1,k_2) =$$

$$\sum_{l_1}^{n_1-1}\sum_{l_1}^{n_1-1} k_f(n_1,n_2;l_1,l_2)\;a(l_1,l_2;k_1,k_2) \qquad (2\text{-}1)$$

where

$$k_f(n_1,n_2;l_1,l_2) =$$
$$E[f(n_1,n_2)-E[f(n_1,n_2)])(f(l_1,l_2)-E[f(l_1,l_2)])] \qquad (2\text{-}2)$$

KLT of $f(n_1,n_2)$ is given by

$$T_f(k_1,k_2)=\sum_{k_1=0}^{N_1-1}\sum_{k_2=0}^{N_2-1} f(n_1,n_2)\,a(n_1,n_2;k_1,k_2) \qquad (2\text{-}3\ a)$$

Inverse KLT is given by

$$f(n_1,n_2)=\sum_{k_1=0}^{N_1-1}\sum_{k_2=0}^{N_2-1} T_f(k_1,k_2)\,b(n_1,n_2;k_1,k_2) \qquad (2\text{-}3\ b)$$

where $a(n_1,n_2; k_1,k_2)$ and $b(n_1,n_2; k_1,k_2)$ are basis functions.

The KLT is interesting theoretically, but it has serious practical difficulties. Since KLT is data dependent, obtaining the KLT basis images for each sub image, in general is a nontrivial computational task. For this reason, the KLT is seldom used in practice for image compression[2,4].

2.2 Discrete Fourier Transform

Discrete Fourier Transform(DFT) has fixed set of basis functions, and is an efficient algorithm for its computation and good energy compaction.

The DFT of a sequence $f(n_1,n_2)$ for $n_1,n_2 = 0,1,\ldots\ldots\ldots\ldots N-1$ is defined as[3]

$$F(k_1,k_2)=\sum_{n_1=0}^{N-1}\sum_{n_2=0}^{N-1} f(n_1,n_2)e^{-j2\Pi k_1 n_1}e^{-j2\Pi k_2 n_2} \qquad (2\text{-}4)$$

for $k_1,k_2 = 0,2\ldots\ldots\ldots N-1$

and the discrete inverse transform is given by

$$f(n_1,n_2) = \frac{1}{N^2}\sum_{k_1=0}^{N-1}\sum_{k_2=0}^{N-1} F(k_1,k_2)e^{j2\Pi k_1 n_1} e^{j2\Pi k_2 n_2} \quad (2\text{-}5)$$

for $n_1,n_2 = 0,1,2,\ldots\ldots..N-1$.

When transform image coding techniques were first developed in the late 1960's the DFT was the first that was considered. DFT has very good energy compaction property for typical images, but it is suboptimal as the transform coefficients $F(k_1,k_2)$ are not correlated. Moreover the DFT gives rise to boundary discontinuities due to Gibbs phenomenon resulting as blocking artifacts in an image[4].

2.3 Discrete Cosine Transform

Compared to other input independent transforms, Discrete Cosine Transform (DCT) has the advantages of packing the most information into the fewest coefficients, and minimizing the block like appearance, called the blocking artifact,[5] that results when the boundaries between subimages become visible. The DCT of sequence $f(n_1,n_2)$ for $n_1=0,1,\ldots\ldots\ldots N_1-1$ and $n_2=0,1,\ldots\ldots\ldots..N_2-1$, is defined as[4]

$$F(k_1,k_2) = \begin{cases} \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} 4f(n_1,n_2)\cos\frac{\Pi}{2N_1}k_1(2n_1+1)\cos\frac{\Pi}{2N_2}k_2(2n_2+1) \\ for \\ 0 \le k_1 \le N_1-1, \\ 0 \le k_2 \le N_2-1 \\ \\ 0, otherwise \end{cases} \quad (2.6)$$

2.3.1 JPEG : DCT-based image coding standard

The idea of compressing an image is not new. However, the discovery of Discrete Cosine Transform (DCT) in 1974, was an important achievement for the scientific, engineering, and research communities working on image compression. Joint Picture Experts Group (JPEG) is one of the most popular and comprehensive still image compression standards brought out in 1992. It defines three different coding systems[4]: (1) a lossy *baseline coding system*, which is based on the DCT and is adequate for most compression algorithms; (2) an extended coding system for greater compression; and (3) a lossless independent coding system for reversible compression. The 'baseline JPEG coder' (Figure 2.1), which is the sequential encoding in its simplest form, will be briefly discussed here. Fig. 2.2(a) and 2.2(b) show the key processing steps in such an encoder and decoder for gray scale images.

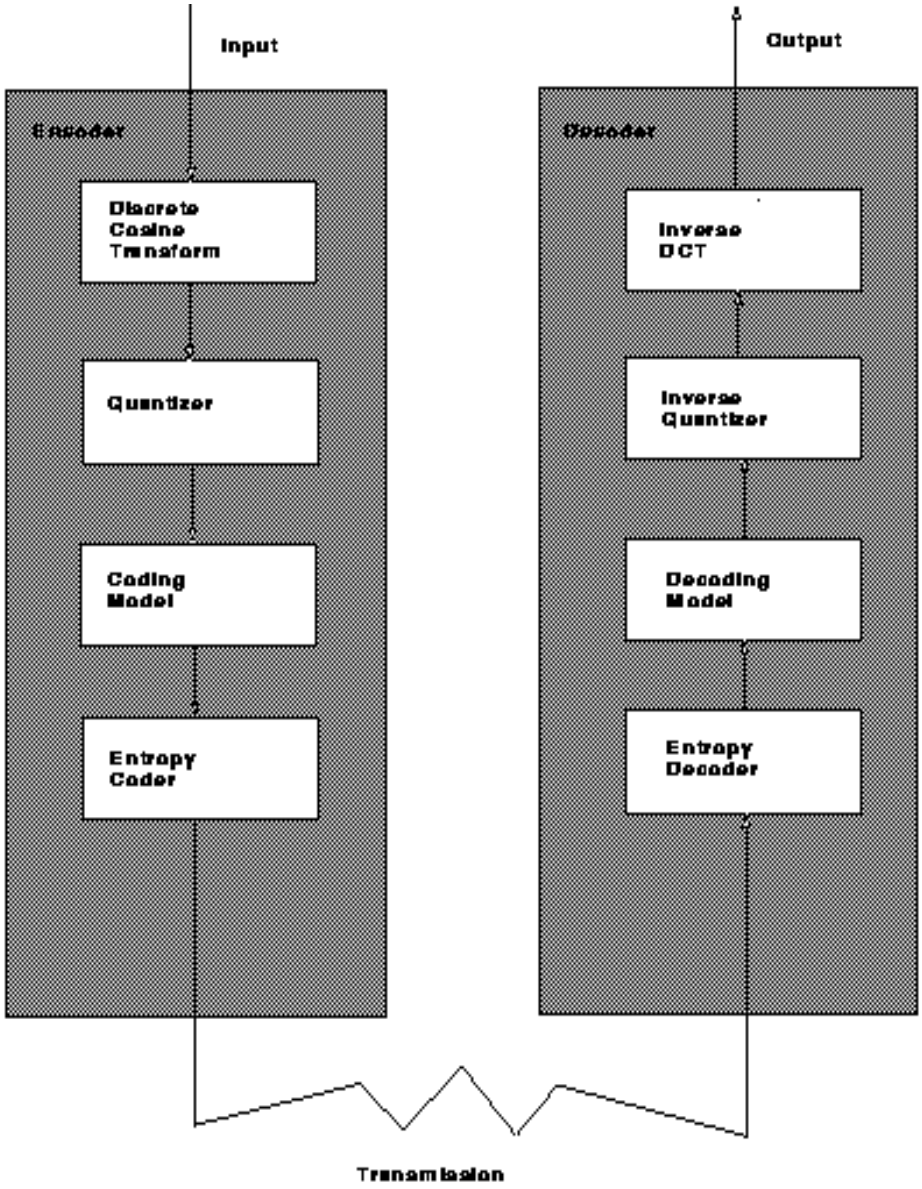


FIG. 2.1 Flow of information from encoder to decoder for JPEG baseline system [5]

The DCT-based encoder can be thought of as essentially compressing a stream of 8x8 blocks of image samples. Each 8x8 block makes its way through each processing step, and yields output in compressed form into the data stream. Because adjacent image pixels are highly correlated, the forward DCT (FDCT) processing step lays the foundation for achieving data compression by concentrating most of the signal in the lower spatial frequencies. For a typical 8x8 sample block from a typical source image, most of the spatial frequencies have zero or near-zero amplitude and need not be encoded. In principle, the DCT introduces no loss to the source image samples and

merely transforms them to a domain in which they can be more efficiently encoded.

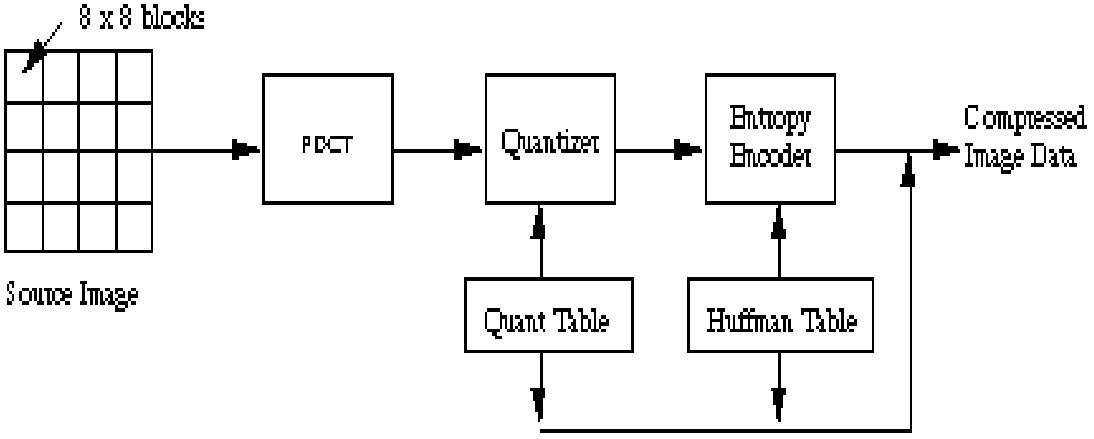


FIG. 2.2(a) JPEG Encoder Block Diagram

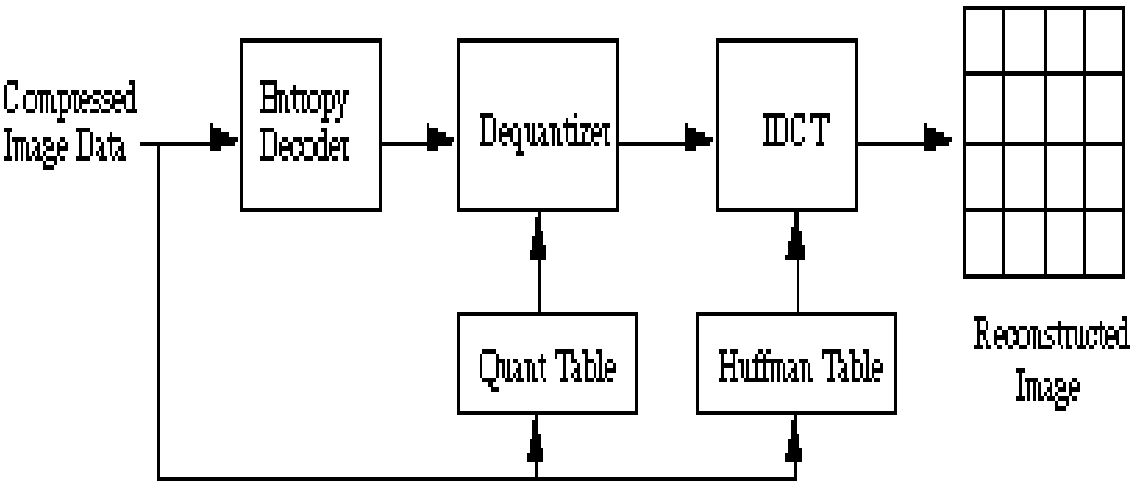


FIG. 2.2(b) JPEG Decoder Block Diagram

After output from the FDCT, each of the 64 DCT coefficients is uniformly quantized in conjunction with a carefully designed 64-element Quantization Table (QT). At the decoder, the quantized values are multiplied by the corresponding QT elements to recover the original unquantized values. After quantization, all of the quantized coefficients are ordered into the "zig-zag" sequence as shown in Figure 2.3. This ordering helps to facilitate entropy coding by placing low-frequency non-zero coefficients before high-frequency coefficients. The DC coefficient, which contains a significant fraction of the total image energy, is differentially encoded[4,5].

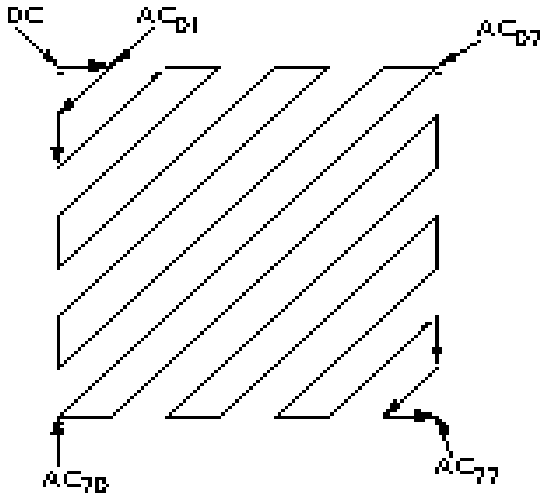


FIG. 2.3 Zig-Zag sequence

Entropy Coding (EC) achieves additional compression losslessly by encoding the quantized DCT coefficients more compactly based on their statistical characteristics. The JPEG proposal specifies both Huffman coding and Arithmetic coding. The Baseline sequential codec uses Huffman coding, but codecs with both methods are specified for all modes of operation. Arithmetic coding, though more complex, normally achieves 5-10% better compression than Huffman coding. The basic steps in JPEG implementation are[4]:-

1. The image is first subdivided into pixel blocks of size 8x8, which are processed left to right, top to bottom.
2. Level shift the 64 pixel by subtracting the quantity 2^{n-1}.
3. The 2-D DCT of the block is then computed, quantized and reordered, using the zigzag pattern of Figure. 2.3 to form a 1-D sequence of quantized coefficients.

2.4 Discrete Wavelet Transform

2.4.1 wavelets

Wavelets (small waves) are functions defined over a finite interval and having an average value of zero. The main idea behind wavelet analysis is to decompose a signal *f* into a basis of functions Ψ*i*:

$$f = \sum_i a_i \psi_i \qquad (2\text{-}7)$$

To have an efficient representation of the signal *f* using only a few coefficients a_i, it is very important to use a suitable family of functions Ψ*i*. The functions Ψ*i* should match the features of the data we want to represent.

Real-world signals usually have the following features: they are both limited in time (time-limited) and limited in frequency (band-limited). What we need is a compromise between the pure time-limited and band-limited basis functions, a compromise that combines the best of both worlds: wavelets ("small waves")[6].

2.4.2 wavelet based compression

Digital image is represented as a two-dimensional array of coefficients, each coefficient representing the brightness level in that point. We can differentiate between coefficients as more important ones, and lesser important ones. Most natural images have smooth color variation, with the fine details being represented as sharp edges in between the smooth variations. Technically, the smooth variations in color can be termed as low frequency variations and the sharp variations as high frequency variations.

The low frequency components constitute the base of an image, and the high frequency components (the edges with give the detail) add upon them to refine the image, thereby giving a detailed image. Hence, the smooth variations are demanding more importance that the details.

Separating the smooth variations and details of the image can be done in many ways. One such way is the decomposition of the image using the Discrete Wavelet Transform (DWT). Digital image compression is based on the ideas of sub band

decomposition or Multi Resolution Analysis (MRA) which can be achieved using DWT[7].

2.4.3 advantages of wavelet based compression

Wavelet Transforms have several advantages with regards to image compression and processing[11]. Some of them are mentioned here:

- One of the main features of Wavelet Transform, which is important for data compression and image processing applications, is its decorrelating behavior.
- Wavelets are localized in both the space (time) and scale (frequency) domains. Hence they can easily detect local features in a signal.
- Wavelets are based on multi-resolution analysis (MRA). Wavelet decomposition allows to analyze a signal at different resolution levels (scales).
- There exist fast and stable algorithms to calculate the discrete wavelet transform and its inverse.

2.4.4 one-dimensional wavelet transform

The one-dimensional discrete wavelet transform can be described in terms of a filter band as shown in Figure 2.4. An input signal *x[n]* is applied to the low pass filter *l[n]* and to the analysis high-pass filter *h[n]*. The odd samples of the outputs of these filters are then discarded, corresponding to a decimation factor of two. The decimated outputs of these filters constitute the reference signal *r[k]* and the detail signal *d[k]* for a new-level of decomposition. During reconstruction, interpolation by a factor of two is performed, followed by filtering using the lowpass and high-pass synthesis filters *l[n]* and *h[n]*. Finally, the outputs of the two synthesis filters are added together[7,8,9,10].

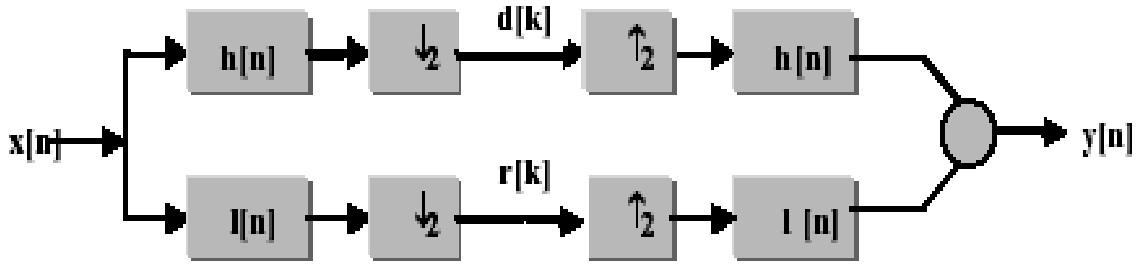


FIG. 2.4 One-dimensional wavelet transform.

The above procedure can be expressed mathematically as the following equations[9].

$$d[k] = \sum_n x[n] \bullet h[2k - n] \qquad (2\text{-}8)$$

$$r[k] = \sum_n x[n] \bullet l[2k - n] \qquad (2\text{-}9)$$

$$x[n] = \sum_n (d[k] \bullet g[2k - n]) + (r[k] \bullet l[2k - n]) \qquad (2\text{-}10)$$

2.4.5 multilevel decomposition by wavelet transform

For a multilevel decomposition, the above process is repeated. The previous level's lower resolution reference signal *ri[n]* becomes the next level sub-sampling input, and its associated detail signal *di[n]* is obtained after each level filtering. Figure 2.5 illustrates this procedure. The original signal *x[n]* is input into the low-pass filter *h[n]* and the high-pass filter *g[n]*[8,10].

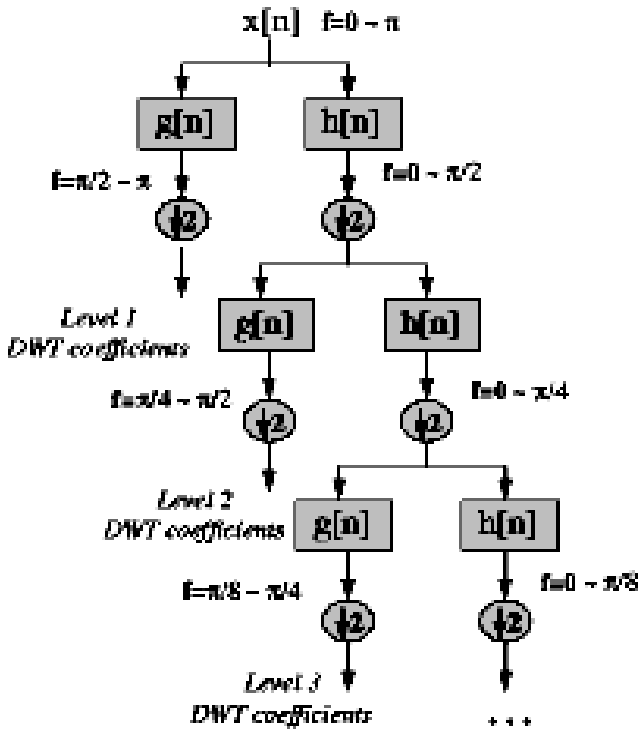


FIG. 2.5 Computing DWT by MRA

2.4.6 two dimensional wavelet transform

To apply wavelet transform to a 2D array, such as an image, it is a common practice to apply the wavelet transform in the horizontal and vertical direction separately. The approach is called *2D-separable* wavelet transform. The 2D data array of the image is first filtered in the horizontal direction, which results in two subbands, a horizontal low and a horizontal high pass subband. Each subband then passes through a vertical wavelet filter. The image is thus decomposed into four subbands, - subband LL (low pass horizontal and vertical filter), LH (low pass vertical and high pass horizontal filter), HL (high pass vertical and low pass horizontal filter) and HH (high pass horizontal and vertical filter). Because the wavelet transform is a linear transform, we may switch the order of the horizontal and vertical wavelet filter, and still reach the same effect.

A multi-scale dyadic wavelet pyramid shown in Figure 2.6 can be obtained by further decomposing the subband LL with another 2D wavelet. Recursive application of

wavelet transform in spatial domain corresponds to partition of data in the frequency domain[7].

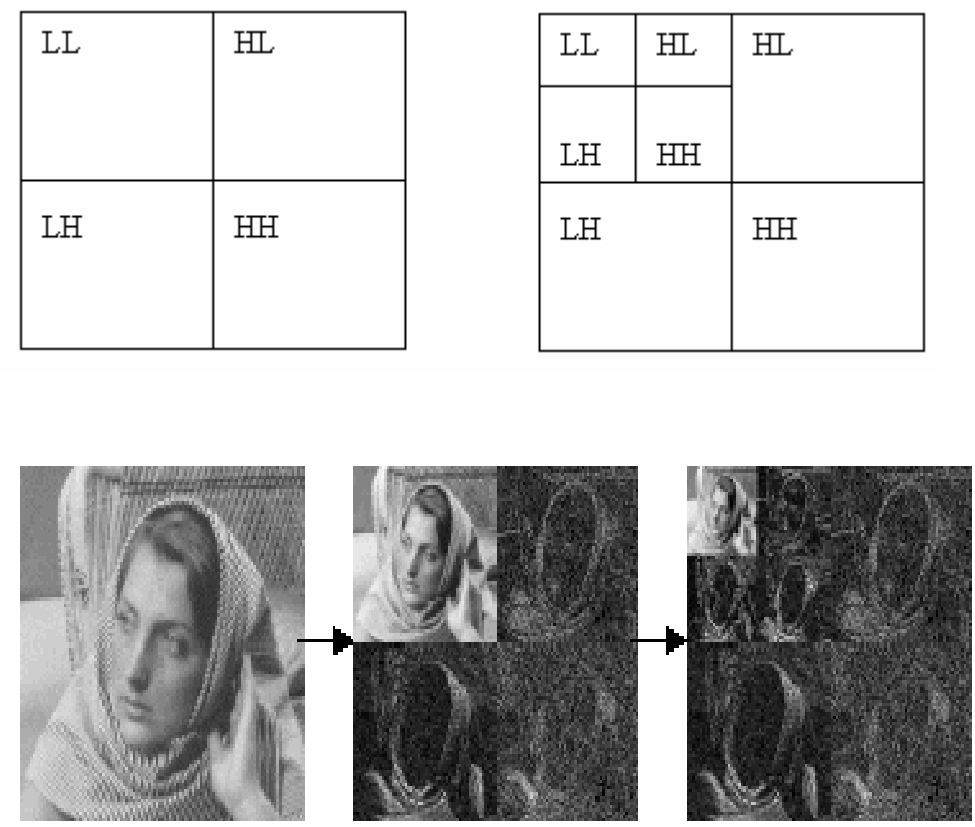


FIG. 2.6 Example of Two level Dyadic Decomposition of an Image

Low-pass samples represent a smaller low-resolution version of the original signal. The high-pass samples represent a smaller residual version of the original; this is needed for a perfect reconstruction of the original set from the low-pass set. This is the basis for multiresolution analysis. In MRA, a scaling function is used to create a series of approximations of a function or image, each differing by a factor of 2 from its nearest neighboring approximations. Additions functions called wavelets are then used to encode the difference in information between adjacent approximations.

A wavelet transform discussed in sections 2.4.3 and 2.4.4 is called the **convolution-based** implementation. The easiest implementation of the filtering process that will work for all wavelet filters (and even non-wavelet filters) is a convolution algorithm.

Here, the signal is first convoluted with the filter bank and then subsampled. In other words, only half the samples are kept, with the other half of the filtered samples thrown away. Clearly this is not efficient, and it would be better to do the sub sampling before the filtering operation. This leads to an alternative implementation of the wavelet transform termed **lifting** approach. It turns out that all FIR wavelet filters can be factored into lifting step[11].

2.4.7 the lifting scheme

The lifting scheme is an algorithm to calculate wavelet transforms in an efficient way. It was developed first as a method to improve a given wavelet transform to obtain some specific properties. Wavelets based on dilations and translations of a mother wavelet are referred to as first generation wavelets or classical wavelets. Second generation wavelets are wavelets that are not necessarily translations and dilations of one function. Second generation wavelets retain the powerful properties of first generation wavelets, like fast transform, localization and good approximation. Lifting scheme is a rather new method for constructing wavelets. The main difference with the classical constructions is that it is does not rely on the Fourier transform. In this way, Lifting can be used to construct second generation wavelets Lifting scheme [15] can in addition, efficiently implement classical wavelet transforms. Existing classical wavelets can be implemented with Lifting Scheme by factorization them into Lifting steps [14].

The basic idea behind the Lifting Scheme is very simple; we try to use the correlation in the data to remove redundancy. To this end, we first split the data into two sets (*Split phase*): the odd samples and the even samples (see Figure 2.7). If the samples are indexed beginning with 0 (the first sample is the 0th sample), the even set comprises all the samples with an even index and the odd set contains all the samples with an odd index. Because of the assumed smoothness of the data, we predict that the odd samples have a value that is closely related to their neighboring even samples. We use N even samples to predict the value of a neighboring odd value (*Predict phase*). With a good prediction method, the chance is high that the original odd sample is in the same range as its prediction. We calculate the difference between the odd sample and its prediction and replace the odd sample with this difference. As long as the signal is highly correlated, the newly calculated odd samples will be on the

average smaller than the original one and can be represented with fewer bits. The odd half of the signal is now transformed. To transform the other half, we will have to apply the predict step on the even half as well. Because the even half is merely a sub-sampled version of the original signal, it has lost some properties that we might want to preserve. In case of images for instance, we would like to keep the intensity (mean of the samples) constant throughout different levels. The third step (*Update phase*) updates the even samples using the newly calculated odd samples such that the desired property is preserved. Now the circle is round and we can move to the next level; we apply these three steps repeatedly on the even samples and transform each time half of the even samples, until all samples are transformed. These three steps are explained below in more detail[12].

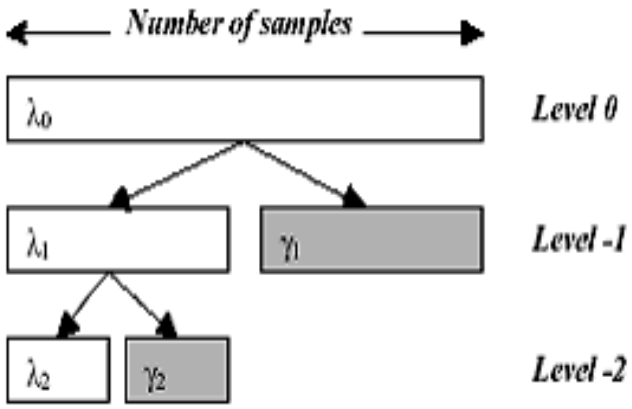


FIG. 2.7 Number of samples in different levels

2.4.7.1 split phase (lazy wavelet transform)

Assume that the scheme starts at level 0. We denote the data set as $\lambda_{0,k}$ where *k* represents the data element and 0 signifies the iteration level 0. In the first stage, the data set is split into two other sets: the even samples $\lambda_{-1,k}$ and the odd samples $\gamma_{-1,k}$ (see Figure 2-8). This is also referred to as the Lazy Wavelet transform because it does not decorrelate the data, but just sub-samples the signal into even and odd samples.

$$\lambda_{-1,k} = \lambda_{0,2k} \qquad (2\text{-}11)$$

$$\gamma_{-1,k} = \gamma_{0,2k+1} \qquad (2\text{-}12)$$

The negative indices are used according to the convention that the smaller the data set, the smaller the index.

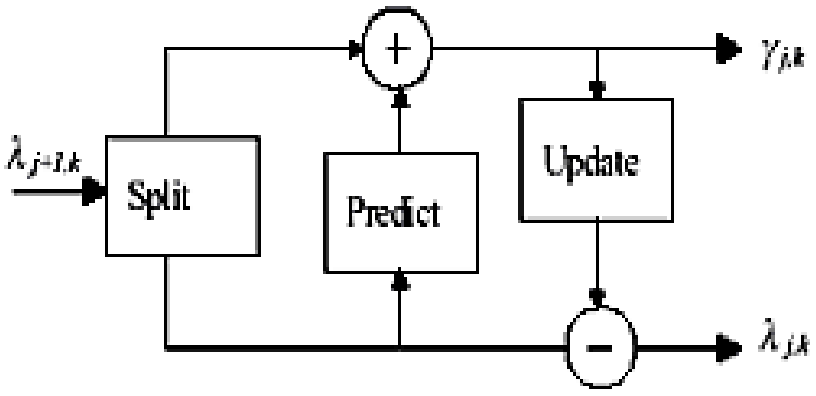


FIG. 2.8 The Lifting Scheme, forward transform: Split, Predict and Update phases

2.4.7.2 *predict phase or dual lifting*

The next step is to use the even set $\lambda_{-1,k}$ to predict the odd set $\gamma_{-1,k}$ using some prediction function *P*, which is independent of the data. The prediction will be

$$Prediction = P(\lambda_{-1,k}) \qquad (2\text{-}13)$$

The more correlation present in the original data, the closer the predicted value will be to the original $\gamma_{-1,k}$. Now, the odd set $\gamma_{-1,k}$ will be replaced by the difference between itself and its predicted value. Thus,

$$\gamma_{-1,k} = \gamma_{-1,k} - P(\lambda_{-1,k}) \qquad (2\text{-}14)$$

Different functions can be used for prediction of odd samples. The easiest choice is to predict that an odd sample is just equal to its neighboring even sample. This prediction method is result to the *Haar* wavelet. Obviously this is an easy but not realistic choice, as there is no reason why the odd samples should be the equal to the even ones.

Another option is to predict that an odd sample $\gamma_{-1,k}$ is equal to the average of the neighboring even samples at its left end right side $\lambda_{-1,k}$, $\lambda_{0,k+1}$.

$$\gamma_{-1,k} = \gamma_{-1,k} - \tfrac{1}{2}\,(\lambda_{-1,k} + \lambda_{0,\,k+1}) \qquad (2\text{-}15)$$

In other words, we assume that the data has a *piecewise linear* behavior over intervals of length 2. If the original signal complies with this model, all wavelet coefficients ($\gamma_{-1,k}$, $\forall k$) will be zero. In other words, the wavelet coefficients measure to which extent the original signal fails to be linear. In terms of frequency content, the wavelet coefficients capture the high frequencies present in the original signal.

The prediction does not necessarily have to be linear. We could try to find the failure to be cubic and any other higher order. This introduces the concept of interpolating subdivision. We use some value N to denote the order of the subdivision (interpolation) scheme. For instance, to find a piecewise linear approximation, we use N equal to 2. To find a cubic approximation N should be equal to 4. It can be seen that N is important because it sets the smoothness of the interpolating function used to find the wavelet coefficients (high frequencies). This function is referred to as the dual wavelet. Figure 2.9-left illustrates linear interpolation (N=2), where one even sample at each side is used for predicting the odd sample, while Figure 2.9-right depicts cubic interpolation (N=4), where two even samples at each side are used for prediction.

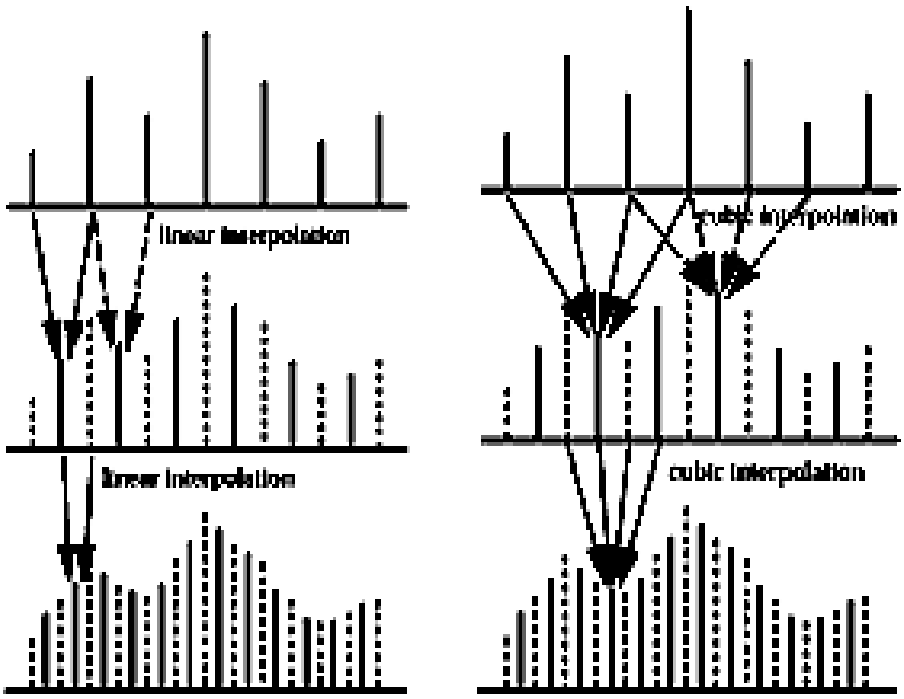


FIG. 2.9 Examples of different prediction functions, Left: Piecewise linear prediction, right: Cubic polynomial interpolation.

2.4.7.3 update phase or primal lifting

By iterating the predict step on the $\lambda_{j,k}$ outputs of each level for lets say n times, we can convert all the samples, except N coarsest level coefficients $\lambda_{-n,k}$, to their corresponding wavelet coefficients. These last N coarsest coefficients are N samples from the original data and form the smallest version of the original signal. This introduces considerable aliasing. We would like some global properties of the original data set to be maintained in the smaller versions $\lambda_{j,k}$**.** for example in the case of images we would like the smaller images to have the same overall brightness, i.e. the same average pixel value. Therefore, we would the last values to be the average of all the pixel values in the original image. This problem can be solved by introducing a third stage: the update stage.

In this stage the coefficients $\lambda_{-1,k}$ are lifted with the help of the neighboring wavelet coefficients γs, so that a certain scalar quantity Q, e.g. the mean, is preserved. A new operator *U* is introduced that ensures the preservation of this quality. Operator *U* uses a wavelet coefficient of the current level $(\lambda_{j,k})$ to update *N* even samples of the same level.

$$\lambda_{-1,k} = \lambda_{-1,k} + U(\gamma_{-1,k}) \qquad (2\text{-}16)$$

This is referred to as *primal lifting.*

2.4.7.4 the inverse transform

One of the great advantages of the lifting scheme realization of a wavelet transform is that it decomposes the wavelet filters into extremely simple elementary steps, and each of these steps is easily invertible. As a result, the inverse wavelet transform can always be obtained immediately from the forward transform. The inversion rules are trivial: revert the order of the operations, invert the signs in the lifting steps, and replace the splitting step by a merging step. Here follows a summary and the steps to be taken for both forward and inverse transform for the popular Cohen, Daubechies and Feauveau classical biorthogonal wavelets whose modified scheme has been implemented in this dissertation for the calculation of wavelet transform (CDF(2,2)) [12,14], (see Figure 2.11).

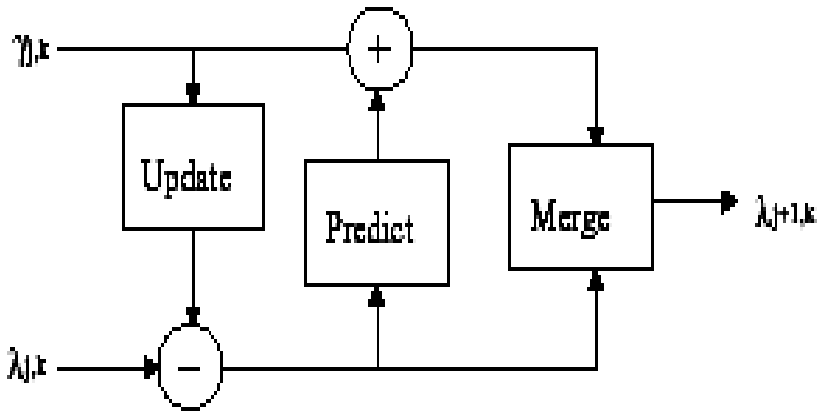


FIG. 2.10 The Lifting Scheme, inverse transform : Update, Predict and Merge stages

Forward transform		Inverse transform	
Splitting	$s_l \leftarrow x_{2l}$ $d_l \leftarrow x_{2l+1}$	Inverse primal lifting	$s_l \leftarrow s_l - \frac{1}{4}(d_{l-1} + d_l)$
Dual lifting	$d_l \leftarrow d_l - \frac{1}{2}(s_l + s_{l+1})$	Inverse dual lifting	$d_l \leftarrow d_l + \frac{1}{2}(s_l + s_{l+1})$
Primal lifting	$s_l \leftarrow s_l + \frac{1}{4}(d_{l-1} + d_l)$	Merging	$x_{2l} \leftarrow s_l$ $x_{2l+1} \leftarrow d_l$

FIG. 2.11 Lifting steps for CDF(2,2) Wavelets

2.4.7.5 integer-to-integer transform

In many applications (e.g. image compression and processing, video, audio, . . .) the input data consist of integer samples. In addition the storage and encoding of integer numbers is easier, compared to floating point numbers. Unfortunately all of the above transforms assume that the input samples are floating point values. Even if the input values actually are integer, doing the filtering operations on these numbers will transform them in rational or real numbers because the filter coefficients need not be integers. To obtain an efficient implementation of the discrete wavelet transform, it is of great practical importance that the wavelet transform is represented by a set of integers as well (possibly up to some scaling factor). Of course, this can always be obtained by scaling and rounding the real numbers, but this rounding process will loose information, and after rounding, the original signal can not be reconstructed from its transform without an error. Especially from a compression point of view, it

looks a bit silly to first expand the integer input data from e.g. 8 bits per sample to 32 bit wide floating point numbers, while the goal is to reduce the total amount of data.

Fortunately the lifting scheme can be modified easily to a transform that maps integers to integers and that is reversible, and thus allows a perfect reconstruction [6]. This is done by adding some rounding operations, at the expense of introducing a non-linearity in the transform. Since even the most elementary lifting steps involve for example divisions by 2, we obtain in general prediction values $P(\lambda_j)$ and update values $U(\gamma_j)$ which are not integer. We shall round these numbers to integers (for example the nearest integer) and indicate this operation by curly braces. Thus, we actually compute rounded values:

$$\gamma_j \leftarrow \gamma_j - \{P(\lambda_j)\} \qquad (2\text{-}17)$$

$$\lambda_j \leftarrow \lambda_j + \{U(\gamma_j)\} \qquad (2\text{-}18)$$

To see that perfect reconstruction is always possible, we note the following. The last computation in a forward transform step is to compute $\lambda_j \leftarrow \lambda_j + \{U(\gamma_j)\}$

Thus the first computation in the inverse transform step is to recompute $\{U(\gamma_j)$ and this will give exactly the same result as before if γ_j has been perfectly reconstructed so far. Therefore $\lambda_j \leftarrow \lambda_j - \{U(\gamma_j)\}$ reconstructs exactly the original λ_j. Thus also $\{P(\lambda_j)\}$ can be reproduced in the reconstruction step, exactly as in the forward step, and thus also $\gamma_j \leftarrow \gamma_j - \{P(\lambda_j)\}$ will give the original γ_j back. This shows that each step of the lifting scheme with rounding, is perfectly invertible and thus the whole signal is perfectly reconstructible. This is one of the most amazing features of integer lifting: whatever deterministic rounding operation is used, the lifting operation is *always* reversible[12].

Chapter 3

ALGORITHM DEVELOPMENT & SOFTWARE IMPLEMENTATION

The entire software is developed using "C" programming language and complied in the Microsoft Visual C++ environment. Total image compression scheme that is developed in this dissertation is methodically explained below along with its software implementation.

3.1 Deviation from the "Baseline JPEG" Model

The main difference between the "Baseline JPEG" (see Figure 3.1) and the present implementation (see Figure 3.2) scheme is that the JPEG is DCT based, whereas this implementation is based on the wavelet transform because of its numerous advantages mentioned earlier in Chapter 2.

JPEG uses a fixed quantization table of size 8 X 8, whereas in the wavelet based implementation, once the image is scaled into the blocks of different resolution, different block threshold levels are used for quantization purpose. The Run Length Coding and the Huffman Entropy coding has been implemented using separate functions, whereas JPEG uses the standard tables.

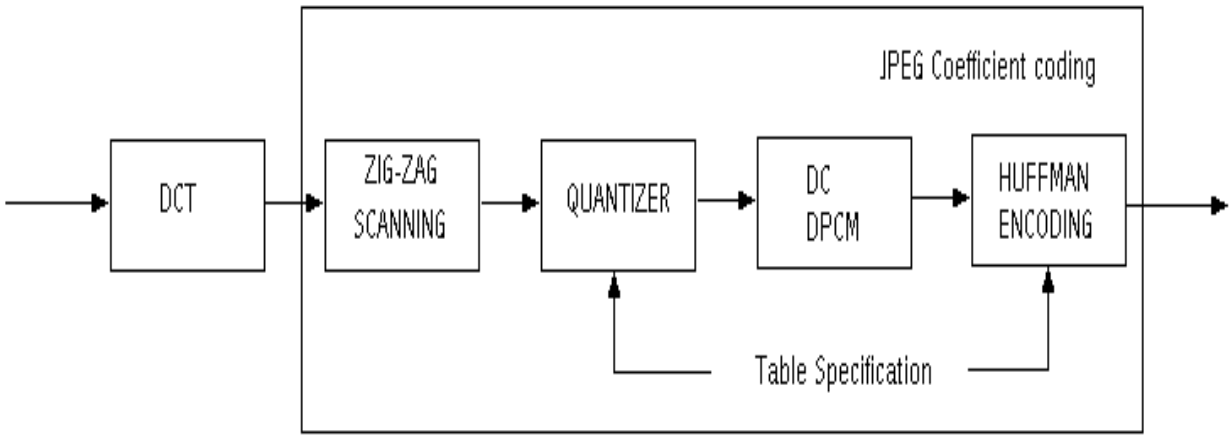


FIG. 3.1 Baseline JPEG basic encoding flow diagram.

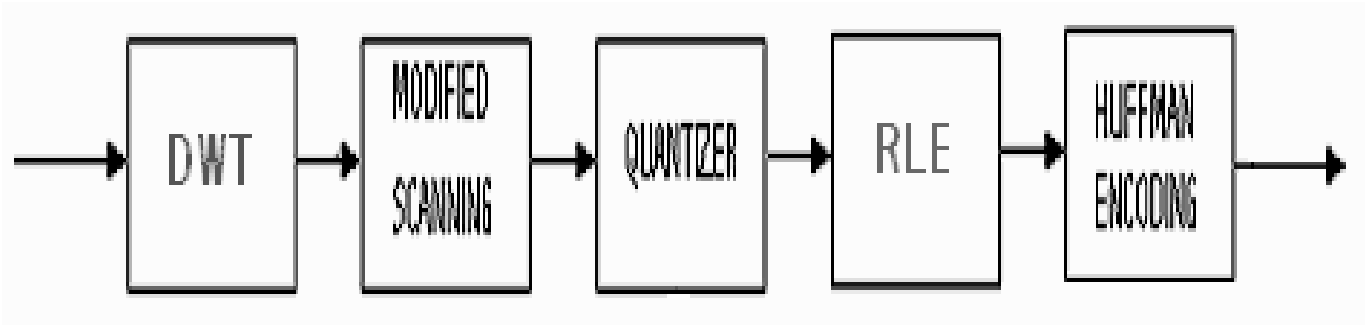


FIG. 3.2 DWT-JPEG based on the JPEG structure

3.2 The Compression step

The "`compress.c`" contains all the routines required for a simple wavelet-based image compression of a 512 X 512 8-bits per pixel image in PGM format. The compression process consists of four basic steps: *wavelet transform, quantization, run-length encoding, and entropy coding.* This is shown in Figure 3.3.

The program accepts an optional compression-factor parameter specifying how aggressively the image should be compressed. A compression factor of 0 indicates minimal compression and maximum image quality. The compression factor of 255

indicates maximum compression with higher degradation to the image quality. If the compression factor is not indicated, a default compression rate of 128 will be used. The compressed output image file format is specific to this program, and has a "`.cmp`" extension.

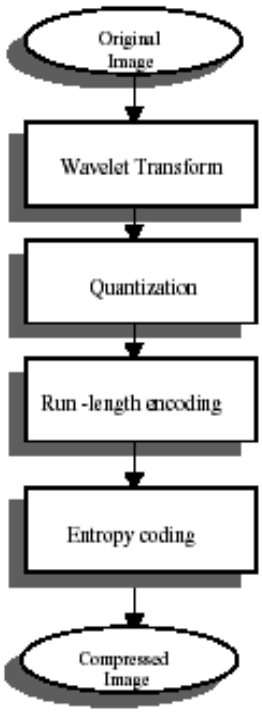


FIG. 3.3 Image Compression routines.

3.2.1 reading the original image

The original image, which is unprocessed, is stripped off with its header information. Only the necessary information is retained while reading the image header. These are: num_cols, num_rows *:* number of columns, rows in the image vector.

The image size(img_size) is then calculated which is equal to the product of number of columns and rows. The information after the header inside the image are the pixel intensities. These values are then read in an array of integers (int_data). The image data is ready for further processing.

3.2.2 wavelet transform routine

The first step, the wavelet transform routine process, is a modified version of the biorthogonal Cohen-Daubechies–Feauveau wavelet[12,14]. The basic concept behind wavelet transform is to hierarchically decompose an input signal into a series of successively lower resolution reference signals and their associated detail signals. At each level, the reference signal and their associated detail signal contain the information needed to reconstruct the reference signal at the next higher resolution level.

The wavelet transform routine employs a lifting scheme to simplify the wavelet transform implementation. Therefore, it only requires integer adds and shifts. The computation of the wavelet filter is performed according to the following equations.

$$D_0 = D_0 + D_0 - S_0 - S_0 \qquad (3\text{-}1)$$

$$S_0 = S_0 + (2*D_0/8) \qquad (3\text{-}2)$$

$$D_i = D_i + D_i - S_i - S_{i+1} \qquad (3\text{-}3)$$

$$S_i = S_i + ((D_{i-1}+D_i)/8) \qquad (3\text{-}4)$$

In the above equations, D_i and S_i are odd and even pixels taken from one row or column, respectively. In image compression, one row or column of an image is regarded as a signal.

Calculation of the wavelet transform requires the pixels to be taken from one row or column at a time. In Equations (1) – (4). D_i should be calculated first before processing S_i. Therefore, the odd pixel should be processed first, then the even pixel due to the data dependency. There are a total of three levels based on the 3-level decomposition wavelet transform algorithm. In each level, the rows are processed first and then the columns. Each level's signal length (amount of each row/column pixels) is half of the previous level. Equations (1) – (4) are grouped into a function called "`forward-wavelet` " & " `fcdf22`" in the C code. Figure 3.4 illustrate the three levels of wavelet transform implementation.

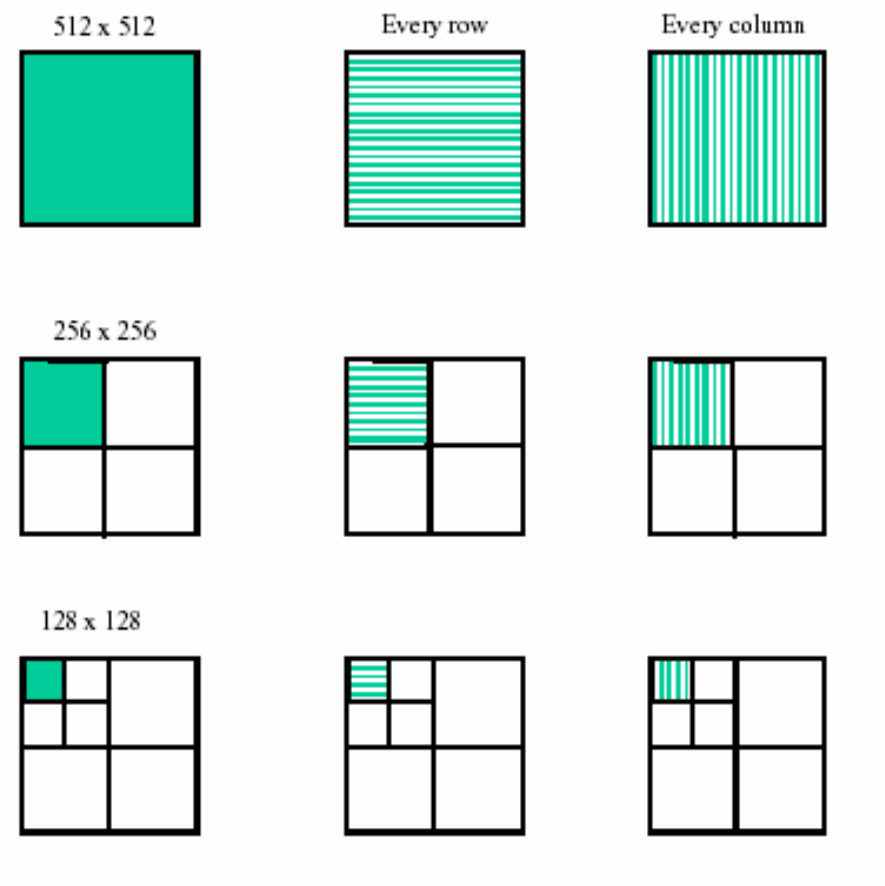


FIG. 3.4 Wavelet transform implementation.

3.2.3 quantization routine

After obtaining the three levels of the wavelet transform, the quantization routine follows. During the quantization routine, the image is divided into 10 blocks; the first four will be 64 x 64 pixels (4096 pixels), then three will be 128 x 128 (16384 pixels), and the remaining three of 256 x 256 pixels (65536 pixels). Every block executes the same quantization process. Figure 3.5 illustrates this as a block diagram. Before processing each block, some parameters should be prepared. First is the blockthresh, which is a threshold value below which all the intensities in the transformed image are given the zero mark. An array is used to hold these 10 block blockthreshes *values*:

Blockthresh[10] = { 0, 39, 27, 104, 79, 51, 191, 99999, 99999, 99999}

For example: Since we want to retain the values in lower numbered blocks, Block 1 ‘s *blockthresh* is 0, Block 10’s *blockthresh* is 99999. The next values that need to be calculated are the sixteen thresholds for each block, $thresh_1$~$thresh_{10}$. The formula to calculate these values is given in equation (5).

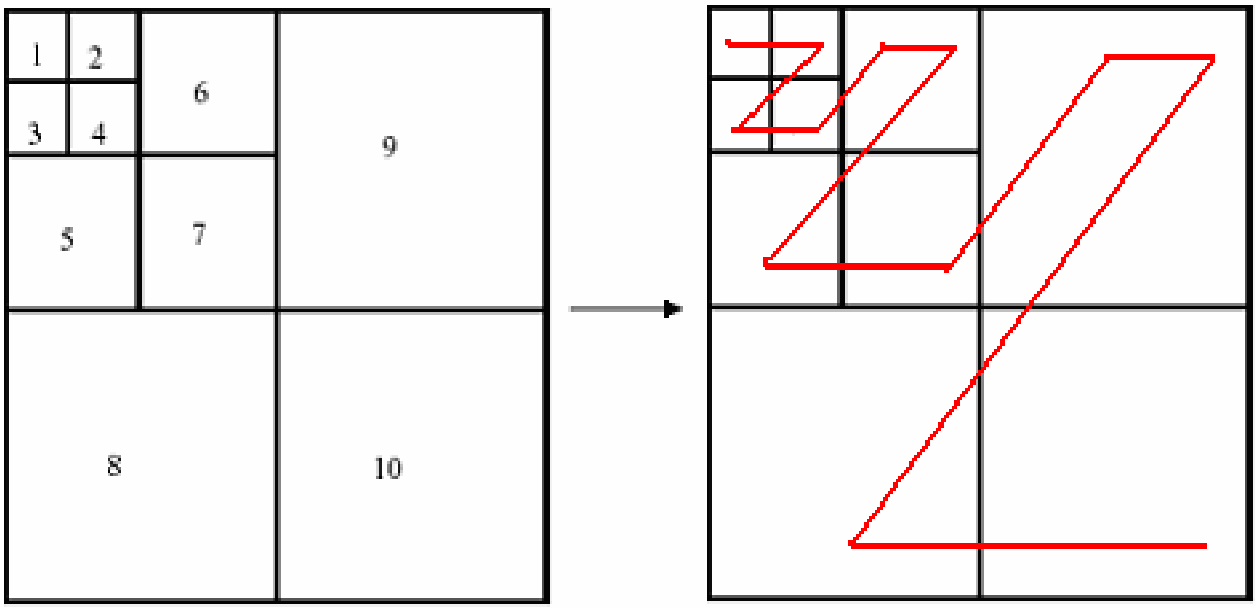


FIG. 3.5 Quantization and Run-Length encoding block diagram.

$$thresh_n = \min + \frac{(\max - \min)\cdot n + 8}{2^4}$$
$$n : 1 \sim 16 \qquad (3\text{-}5)$$

The $thresh_n$ is the *n*th threshold value in a block, and *n* value ranges from *1* to *16*. Values *min* and *max* are the minimal and maximum pixel values within this block. After these numbers are computed, each block can run the quantization process. First, each input pixel's absolute value is compared with its corresponding *blockthresh[n]*, if it is smaller than the *blockthresh* value, the original pixel value is assigned to a constant value *ZERO_MARK*, which should be defined by the user. In this dissertation, this is assigned a value of *16*. If the `abs` *(pixel)* value is not smaller then the *blockthresh* value, the pixel will be passed to a subroutine called "`classify()`" in the program. The original pixel value will then be changed into its corresponding *threshn* value after this call. Listing 3-1 is the pseudo code to represent the above calculation.

```
if(abs(val)< value of that block’s threshold)
quant_buf[num][qsize++]=ZERO_MARK;
else
quant_buf[num][qsize]=classify(val);

/*********************************************************
Routine: classify() to find the quantized value in the
range of 0 to 16.
*********************************************************/

int classify(int val)
{
if(val>thresh8) {
if(val>thresh12) {
if(val>thresh14) {
if(val>thresh15) return 15;
else return 14;
} else {
if(val>thresh13) return 13;
else return 12;
} } else {
if(val>thresh10) {
if(val>thresh11) return 11;
else return 10;
} else {
if(val>thresh9 ) return 9;
else return 8;
} } } else {
if(val>thresh4) {
if(val>thresh6 ) {
if(val>thresh7 )
 return 7;
else
return 6;
```

```
} else {
if(val>thresh5 )
return 5;
else
return 4;
} } else {
if(val>thresh2 ) {
if(val>thresh3 )
return 3;
else return 2;
} else {
if(val>thresh1 )
return 1;
```

LISTING 3-1: Pseudo code for quantization algorithm.

3.2.4 run-length encoding routine (RLE)

Run-length encoding is the next routine following the quantization process. The basic concept is to code each contiguous group of 0's encountered in a scan of a row by its length and to establish a convention for determining the value of the run. This is shown as an example in Figure 3.6

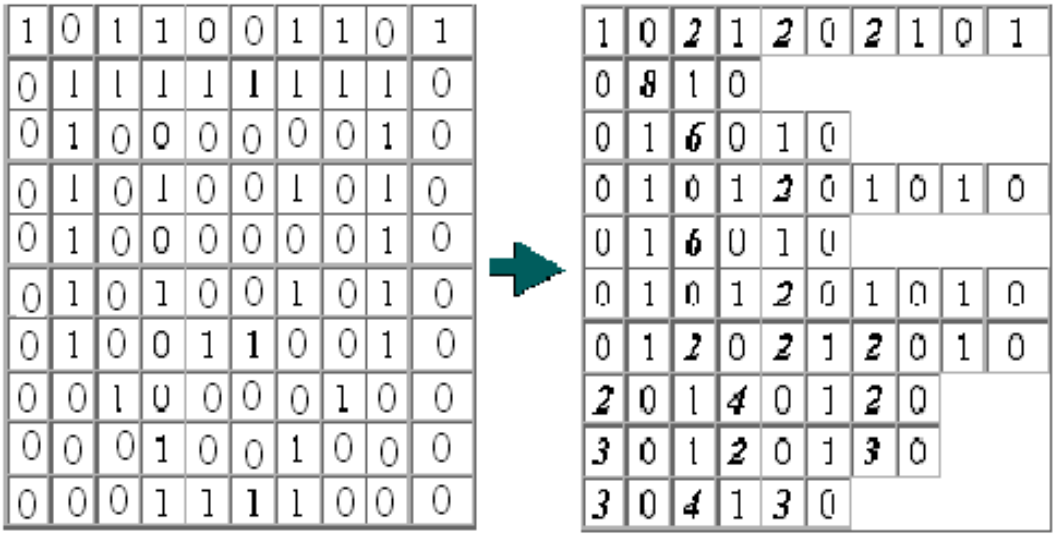


FIG. 3.6 An example of Run Length Encoding

The purpose of this step is to compress the image size based on the pixel values from quantization, which are between integer values *0* to *16*. The image can be compressed

as much as up to 10% of the original after the run length encoding. As in the quantization routine, the image is also divided into 10 blocks. Each block will run the same run-length encoding algorithm.

Because values in blocks 8, 9, and 10 after quantization are all equal to 16, these three blocks will not be processed. The process pseudo code for each block is shown in Listing 3-2.

```
While (pixel is in this block)
{
if ( pixel !=ZERO_MARK (16) )
saving pixel value in rle-buffer;
else {
count = 0;
while( pixel ==ZERO_MARK(16))
{
count++;
if((count == 256-ZERO_MARK)||(no more pixels in this
block))
break;
read the next pixel;
}
save (count+ZERO_MARK-1) in rle-buffer;
}
read the next pixel;
}
```

LISTING 3-2: Pseudo code for each block run-length encoding algorithm.

3.2.5 entropy coding routine

The last routine in the Image Wavelet Compression algorithm is entropy coding. This process is based on the calculation results from run-length encoding. Using the Huffman encoding algorithm for the entropy coding, the resulting image file can be compressed as much as up to 2-3% of the original image size. In the algorithm, two 256 size integer arrays are used to hold the parameters for a fixed Huffman encoding tree. These parameters are expected to work reasonably well with most images. The definition of these parameters is as follows.

Int HufSize[256] = { 9, 7, 6, ….}
Int HufVal[256] = {0x00097, 0x00066 …}

The Huffman coding call in the C code refers to the following listing 3-3.

```
/*********************************************************
 Routine: void hufenc(unsigned char ich, int *nb)
*********************************************************/
void hufenc(unsigned char ich, int *nb)
{
nbits=HufSize[ich];
val=HufVal[ich];
for(i=0; i<nbits; i++)
{
bytn=(*nb)>>3;
bitn=(*nb)&7;
if((val&(1<<i))>0) codep[bytn]|=(1<<bitn);
(*nb)++;
}
}
```

LISTING 3-3: Pseudo code for entropy coding algorithm.

The whole process includes checking values from the Huffman tree parameters, shifting, multiplication, addition, and comparison to get the final compressed value.

3.2.6 writing the compressed file

The original image, which has been processed using the above steps, is written to a "`.cmp`" format file with name "`Original_Image.cmp`". A header is attached with this file, which contains the maximum and minimum values of pixel intensities, run length, number of bytes inside every block of the decomposed image and the compressed signal.

3.3 The Decompression step

The file "`decompress.c`" consists of all routines for decompression of a "`.cmp`" format file. (This is a program specific format). There are four distinct steps for decompression of the "`.cmp`" format file. These are: *reverse wavelet transform, dequantization, run length decoding and entropy decoding*.

Each of these steps is just the opposite of steps applied in the compression routine. The output of this program will generate a PGM format file. The expected input is the name of the file to be decompressed. The name of the output "`.pgm`" file is an optional input.

3.4 Comparison of Original and the Reconstructed Image

The program "`compare.c`" will be used to perform the acceptance test of the image quality. It will compare the original "`.pgm`" image file with the decompressed "`.pgm`" image. It will report the root mean square error (RMSE) and the peak signal to noise ratio (PSNR). Higher image quality is reflected by a low RMSE, and therefore high PSNR. The acceptance test is based solely on the signal to noise ratio (PSNR). This program expects the name of both "`.pgm`" image files as its input.

3.5 Implementation of DCT based Image Coder

A DCT based image compression system was also developed for the comparison purpose. The only difference that lies in the standard baseline JPEG model is that the coefficients are coded using variable length coding instead of using the standard Huffman tables[4]. The coding used is explained below.

The code for each coefficient consists of two components, a bit count, and an encoded value. The bit count is encoded as a prefix code with the following binary values:

Number of Bits	Binary Code
0	00
1	010
2	011
3	1000
4	1001
5	1010
6	1011
7	1100
8	1101
9	1110
10	1111

A four-bit number telling how many zeros are in the encoded run follows a bit count of zero. A value of 1 through ten indicates a code value follows, which takes up that many bits. The encoding of values into this system has the following scheme:

Bit Count	Amplitudes
1	-1, 1
2	-3 to -2, 2 to 3
3	-7 to -4, 4 to 7
4	-15 to -8, 8 to 15
5	-31 to -16, 16 to 31
6	-63 to -32, 32 to 64
7	-127 to -64, 64 to 127
8	-255 to -128, 128 to 255
9	-511 to -256, 256 to 511
10	-1023 to -512, 512 to 1023

The compression is performed in three sequential steps. First is the computation of the DCT coeffiecients, then the quantization of the transformed coefficients and finally

the variable-length code assignment. The image is first subdivided into pixel blocks of size 8X8, which are processed left to right and top to bottom. As each 8X8 block or subimage is encountered, its 64 pixels are left shifted by subtracting the quantity 2^{n-1}, where 2^n is the maximum number of gray levels in a grayscale image. The 2-D DCT of the block is then computed, quantized, and reordered using the zig-zag scanning pattern to form 1-D sequence of quantized coefficients. All the coefficients are then coded using the variable length coding and written to a "`.dat`" file which is the compressed file size.

Chapter 4

RESULTS

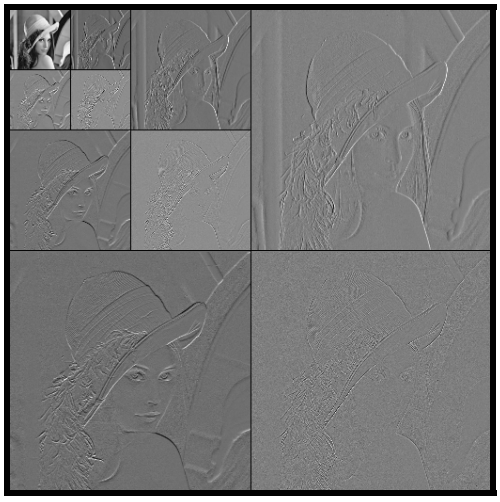
This section discusses the experimental results obtained using the developed algorithm for the Image Codec. The results are presented according to their qualitative evaluation, the relative size of the compressed file formats and the intelligibility of the reconstructed medical image.

4.1 Bitrate(bpp) Vs. PSNR(db)

For the qualitative investigation of the reconstructed images, PSNR values of the standard images "LENA" and "BARBARA" were taken for comparison with the well-known techniques and formats of Image compression. The standard images have been compared with the JPEG, JPEG2000, and the DCT based Image Coder. A practical image "BRAIN" has been compared with the Embedded Zerotree Wavelet algorithm [16]. Figures 4.1, 4.2, 4.3 show the original images, the 3 scale decomposition of the images using the wavelet transform, reconstructed images at different bits per pixel (bpp) using the DCT based and the present DWT based JPEG algorithm. Figures 4.4, 4.5, 4.6 present the graphs of PSNR Vs. the bitrate for these images corresponding to the values in Tables 4.1,4.2,4.3.

It can be observed from Figures 4.1, 4.2 and 4.3 that the DCT based algorithm produces blocking artifacts which appear in the image as small squares. This is due to the fact that the DCT is applied to the blocks of size 8X8, on the entire image. The wavelet based technique does not produce these blocking artifacts even at a reduced bit rate (0.30 bpp) (see Figure 4.1(e),4.2(e),4.3(e)). From Table 4.1, for image LENA 512X512, there does not appear much difference in the PSNR values at higher bit rates(0.75 bpp). As the bit rate is reduced, the JPEG and the DCT base implementation show a degraded PSNR response to the developed new coder. The PSNR values of JPEG2000 outperform the developed coder at all bit rates because of its complex quantizer and coding system. Similar behavior is observed for the image BARBARA 512X512, Table 4.2, which shows a peculiarity at 0.25 bpp where the PSNR of the developed coder shows a moderate gain of around 0.5 dB over the

JPEG2000 standard. For the practical test image BRAIN 512X512, Table 4.3, the new coder shows a gain of 1dB over the embedded zerotree coder at 0.25bpp.



(a)

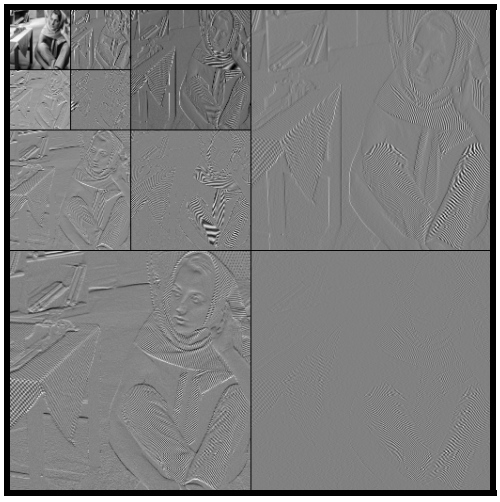
(b)

(c)

(d)

(e)

FIG. 4.1 (a) Lena 512X512- original image; (b) three level decomposition of Lena; (c) reconstructed image using DCT based image codec at 0.42 bpp; (d),(e) reconstructed image using DWT based JPEG at 0.88 and 0.30 bpp respectively.



(a)

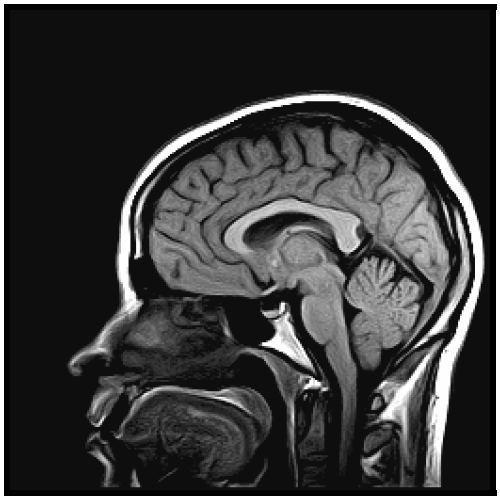
(b)

(c)

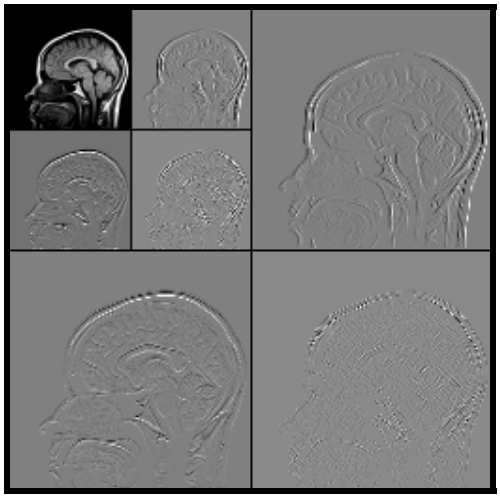
(d)

(e)

FIG. 4.2 (a) Barbara 512X512 - original image; (b) three level decomposition of Lena; (c) reconstructed image using DCT based image codec at 0.5 bpp; (d), (e) reconstructed image using DWT based JPEG at 0.9 and 0.20 bpp respectively.



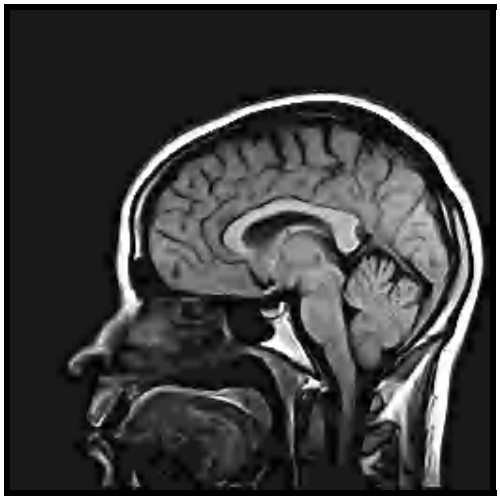
(b)



(a)

(c)

(e)



(d)

FIG. 4.3 (a) Brain 512X512 - original image; (b) three level decomposition of Lena; (c) reconstructed image using DCT based image codec at 0.5 bpp; (d), (e) reconstructed image using DWT based JPEG at 0.96 and 0.18 bpp respectively.

Table 4.1 Coding results for 512X512 LENA showing PSNR

O/p Bytes	Rate (bpp)	Ratio	JPEG	DWT-JPEG	JPEG2000	DCT
				PSNR (dB)	PSNR (dB)	PSNR (dB)
29492	0.9	8.9:1	--	36.54	47.50	34.43
24576	0.75	10.6:1	36.60	36.93	45.72	32.59
19661	0.60	13.3:1	35.6	37.24	43.66	29.87
16384	0.50	16:1	34.90	37.36	42.20	26.93
8192	0.25	32:1	31.6	36.69	37.40	-

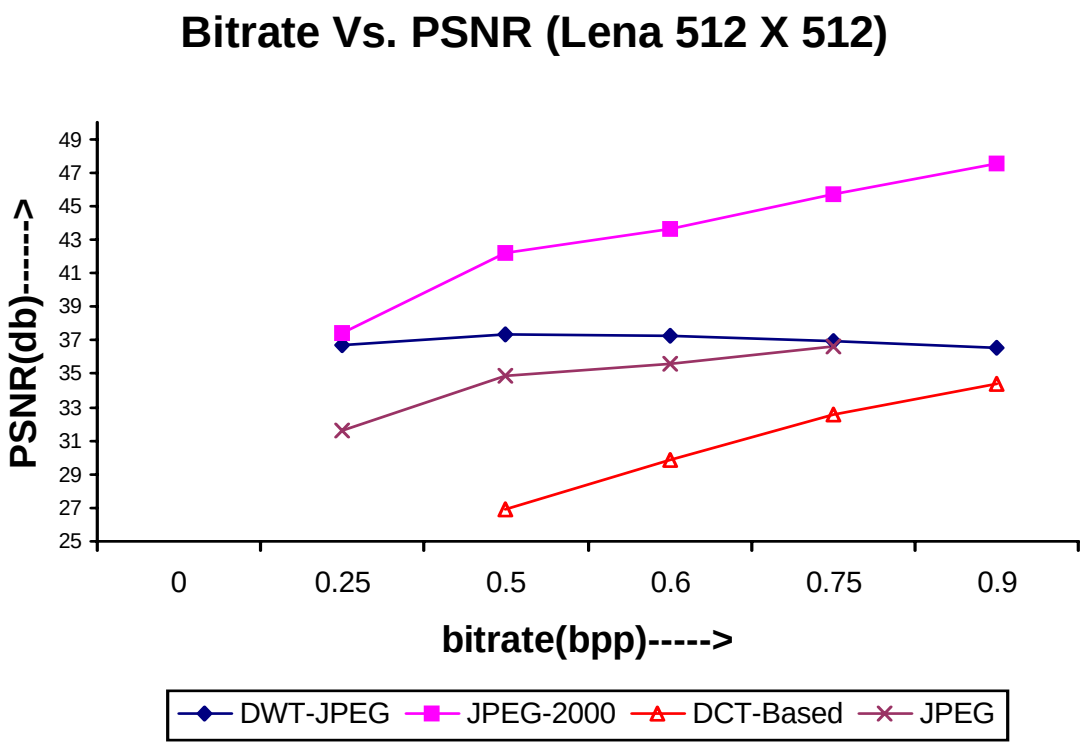


FIG. 4.4 PSNR as a function of bit rate for the Lena image

TABLE 4.2 Coding results for 512X512 BARBARA showing PSNR

O/p Bytes	Rate (bpp)	Ratio	JPEG	DWT-JPEG	JPEG2000	DCT
				PSNR (dB)	PSNR (dB)	PSNR (dB)
29492	0.9	8.9:1	--	29.40	36.27	28.85
24576	0.75	10.6:1	31.00	29.60	34.88	26.85
19661	0.60	13.3:1	29.4	29.66	33.40	24.49
16384	0.50	16:1	28.30	29.61	32.30	22.64
8192	0.25	32:1	25.2	28.82	28.36	-

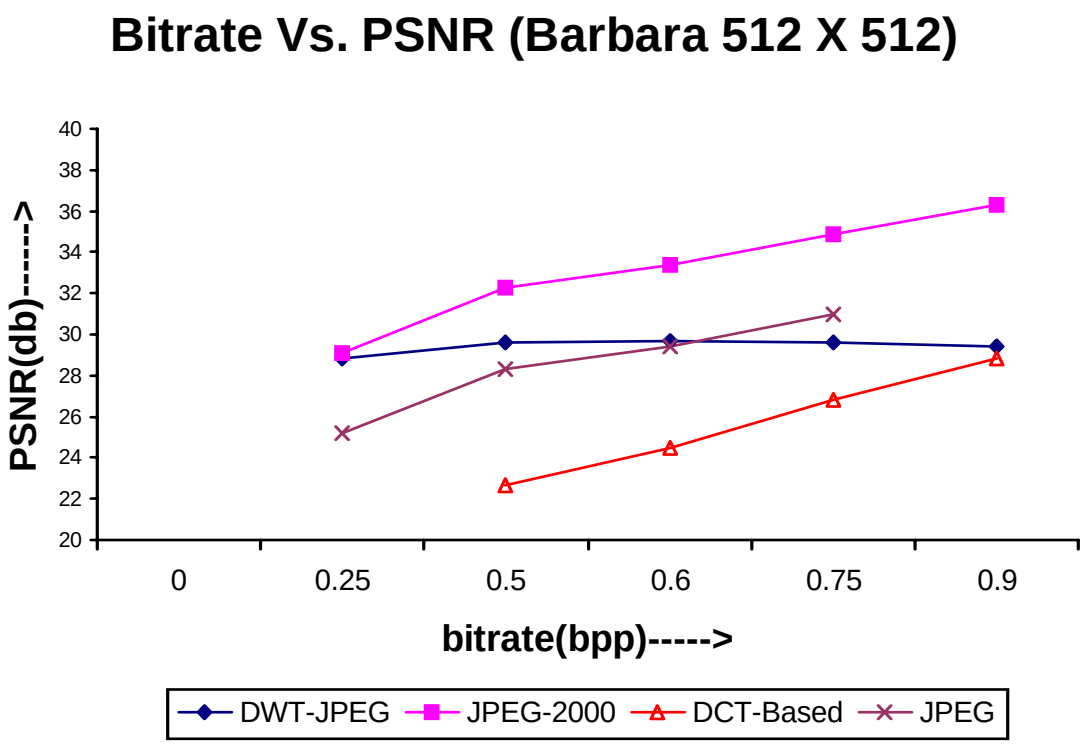


FIG. 4.5 PSNR as a function of bit rate for the Barbara image

TABLE 4.3 Coding results for 512X512 BRAIN showing PSNR

O/p Bytes	Rate (bpp)	Ratio	EZW	DWT-JPEG	JPEG2000	DCT
			PSNR (dB)	PSNR (dB)	PSNR (dB)	PSNR (dB)
29492	0.9	8.9:1	36.41	25.30	38.28	31.23
24576	0.75	10.6:1	35.53	29.29	36.42	28.78
19661	0.60	13.3:1	33.31	25.38	34.78	25.71
16384	0.50	16:1	32.00	29.70	33.51	22.26
8192	0.25	32:1	28.4	29.39	29.61	-

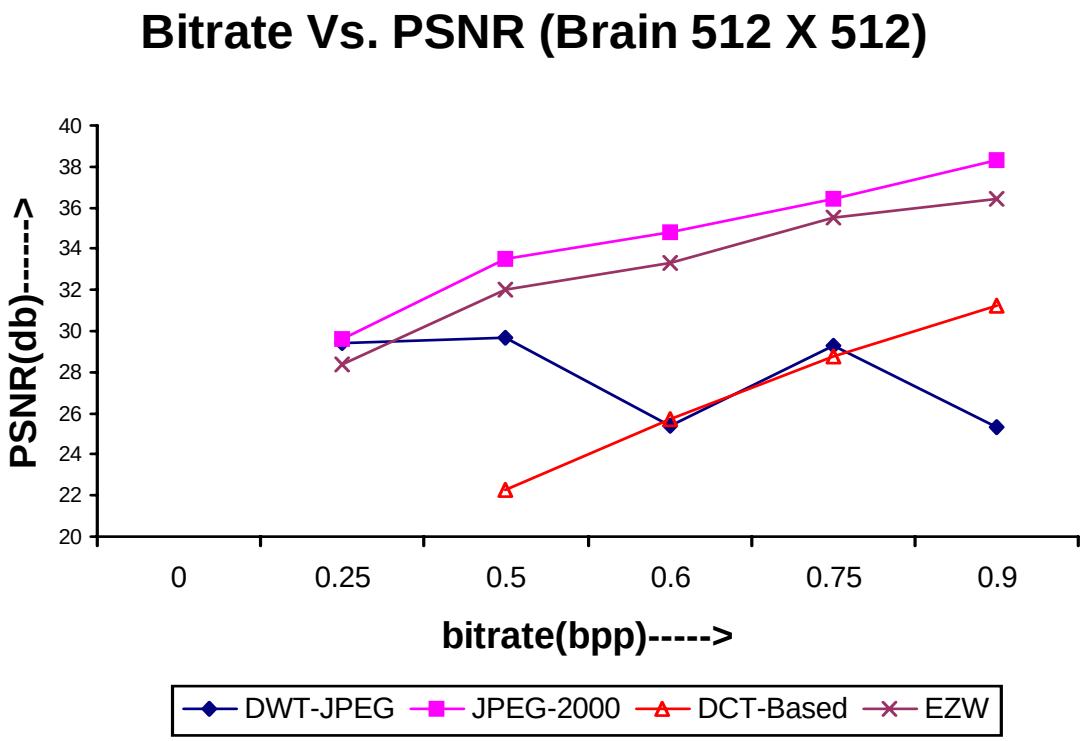


FIG. 4.6 PSNR as a function of bit rate for the Brain image

4.2 File Size Vs. File Format of Storage

This section presents the comparative file sizes of different images compressed using different techniques. These are the GIF, JPEG and the present DWT-JPEG model. For the DWT based JPEG, quality factor was fixed by setting the compression factor to: low (255), medium (128), high (0). For compressing images in the GIF method, Compuserve GIF (interlaced) codec in Irfanview has been used which gives a constant file size.

Adobe Photoshop version 5.00 was used to encode images in the JPEG format. The quality settings were 1 (low), 5 (medium), 10 (high) with baseline standard option.

This is shown in charts in Figures 4.7, 4.8, 4.9. The GIF format produces the compressed files of a fixed size for all three quality factors, whereas the developed new coder compresses the files to a size that is much lesser to the baseling JPEG format at all quality factors.

TABLE 4.4 Coding results showing file size at low quality.

	GIF	JPEG	DWT-JPEG
LENA	146,359	25,179	3,767
BARBARA	291,581	37,068	4,915
BRAIN	44,046	25,057	4,965
GOLDHILL	255,943	21,796	3,636

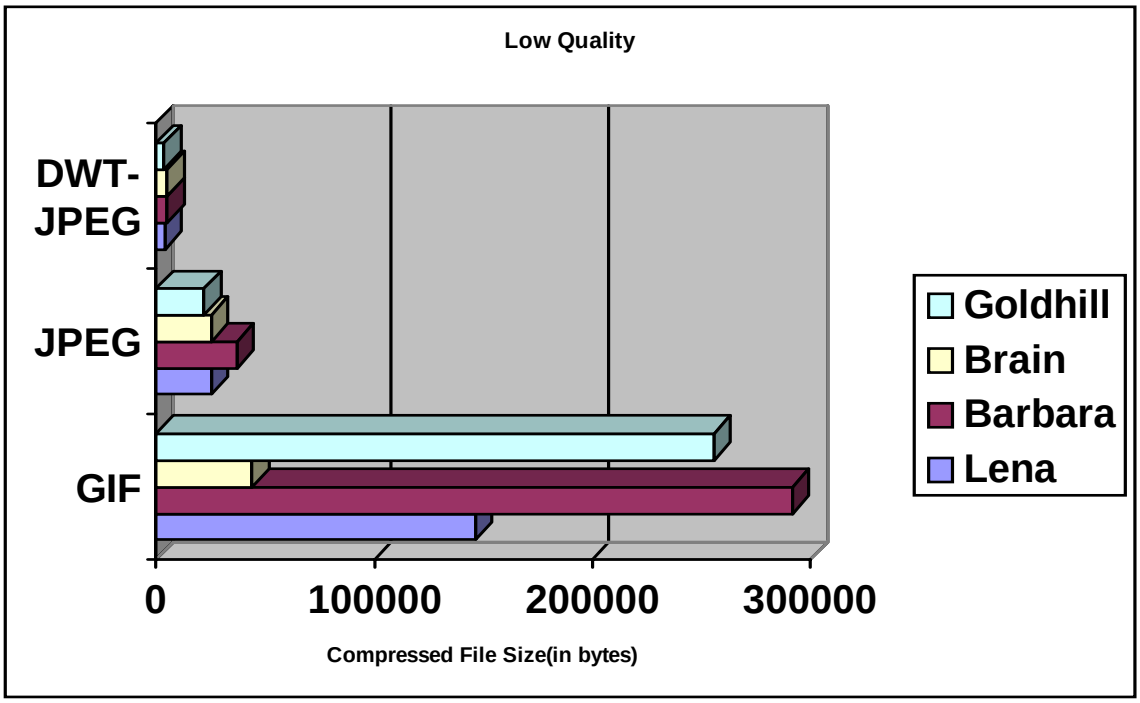


FIG. 4.7 Compressed file size Vs. Compressed file format (low quality)

TABLE 4.5 Coding results showing file size at medium quality.

	GIF	JPEG	DWT-JPEG
LENA	146,359	40,743	5,556
BARBARA	291,581	51,144	8,187
BRAIN	44,046	43,356	7,271
GOLDHILL	255,943	47,144	6.070

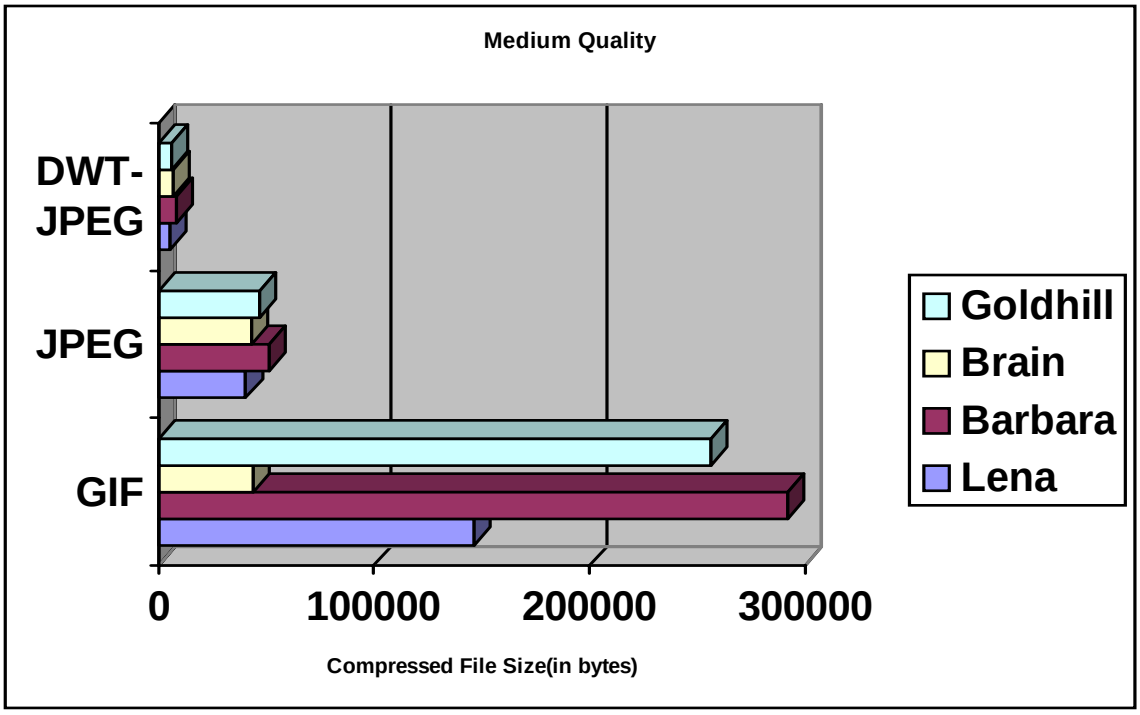


FIG. 4.8 Compressed file size Vs. Compressed file format (medium quality)

TABLE 4.6 Coding results showing file size at high quality.

	GIF	JPEG	DWT-JPEG
LENA	146,359	90,558	28,775
BARBARA	291,581	166,713	29,708
BRAIN	44,046	114,765	31,444
GOLDHILL	255,943	170,630	30,024

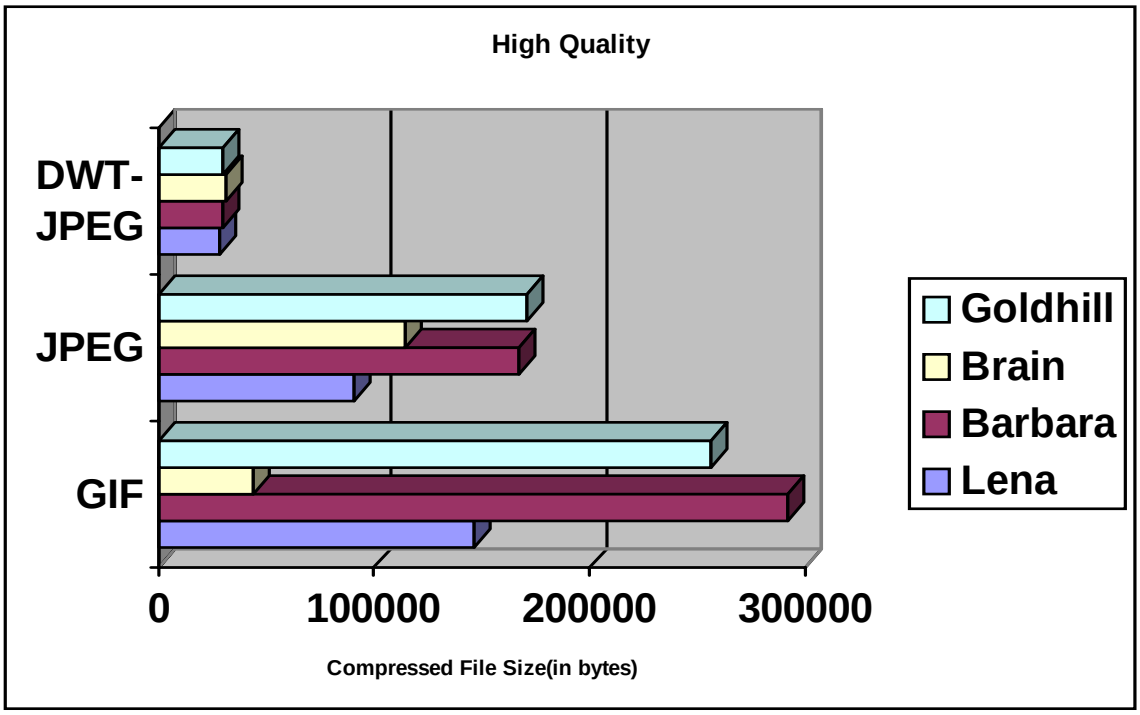
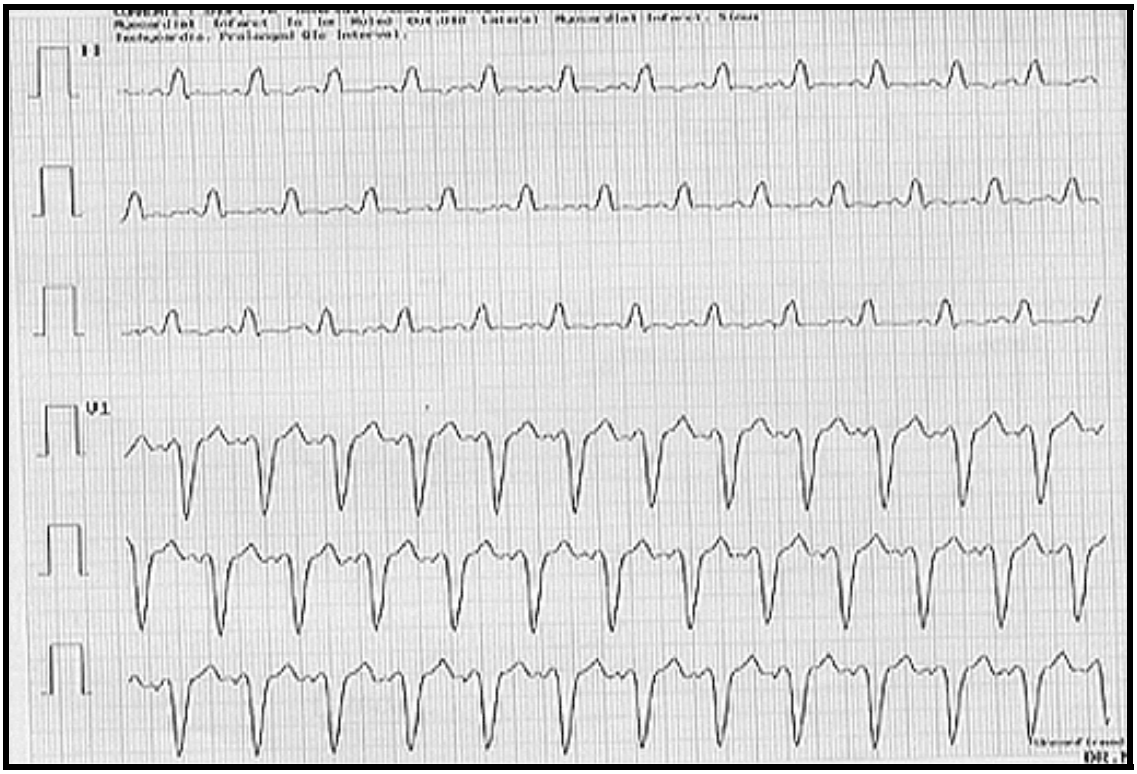


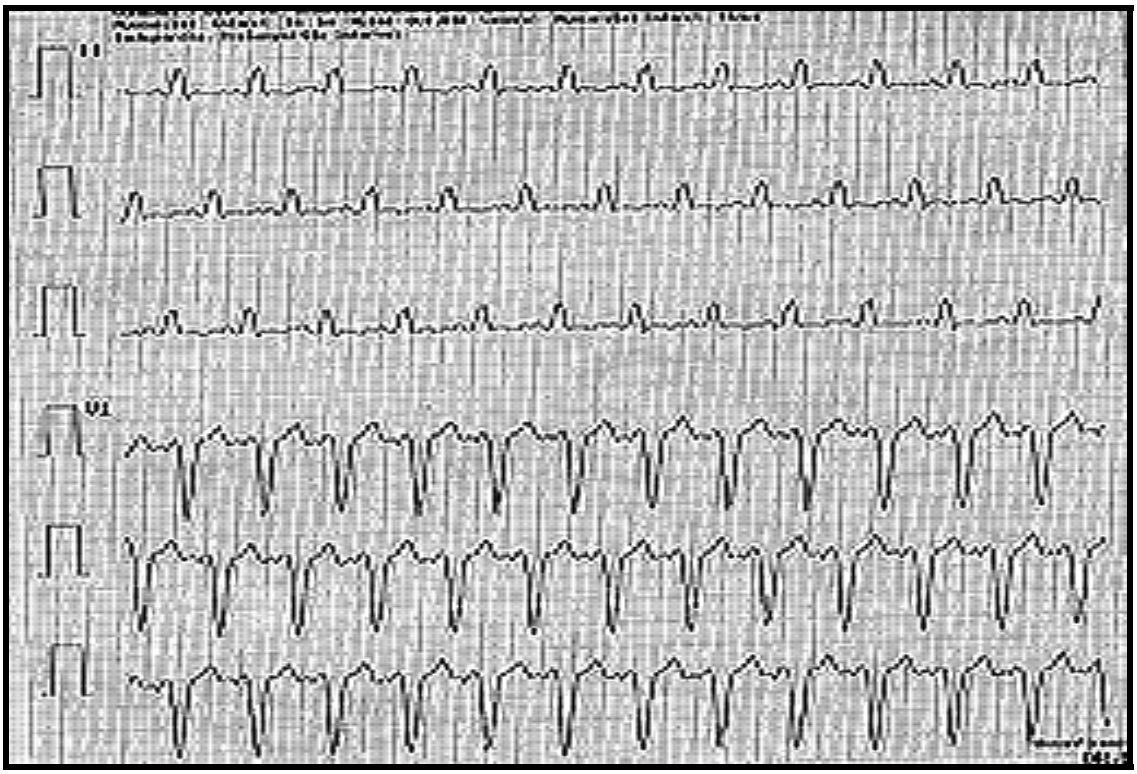
FIG. 4.9 Compressed file size Vs. Compressed file format (high quality)

4.3 Perceptual Validation

An ECG signal of a cardiac patient was scanned and compressed using the present compression algorithm. The original image and the reconstructed result (at Compression Ratio of 88.8:1) is shown in Figure 4.10. Perceptual validation was done by showing the coding results to a few subjects. The results were found to be "perceptible and slightly annoyin" to the Human Visual System (HVS) of the subjects. The result shows that the reconstructed image is well within the perceptibility of the corrected HVS. Several other medical images were tested and the perceptibility of the reconstructed images was found to be of good quality at even higher compression ratios.



(a)



(b)

FIG. 4.10 (a) Original ECG image (512X512), (b) Reconstructed image at compression ratio of 88.7:1

Chapter 5

CONCLUSION AND SUGGESTIONS FOR FURTHER RESEARCH

5.1 Conclusions

The DWT based JPEG was shown to outperform the baseline JPEG, the DCT based compression algorithm and approaching the performance of more sophisticated and complex DWT based coders. The lifting scheme is an efficient implementation of the conventional wavelet transform technique and it is not computationally intensive and does not require large amount of data and memory and is therefore optimal for the application.

The images that have been used for the testing of algorithm are the standard images for image compression research. Results show that the image quality (PSNR) does not degrade even at low bit rates when compared to the other compression methods. For a standard test image LENA 512 X 512, at higher bit rates (0.75 bpp) there is not much difference in PSNR values, whereas for low bit rates (0.25 bpp) there is a difference of 5 dB in the PSNR values of the new coder and the baseline JPEG coder and hence the poor performance of JPEG at lower bit rates. The DCT based image coder has an overall degraded performance at all bit rates. JPEG-2000 outperforms this new coder at all bit rates because of its complex quantizer system. Similar behavior is observed for BARBARA 512 X 512 image with the exception that (0.25 bpp) the developed coder shows a slight improvement in PSNR over the JPEG2000. For a practical test image BRAIN 512 X 512, developed coder shows a gain of 1dB to 5dB over the DCT based implementation. Moreover the comparison of compressed file sizes using different compression techniques illustrate that the new image coder provides better compression ratio for all quality of images. Perceptual validation of a practical medical ECG signal was done by showing the coding results to a few subjects. The results are “perceptible and slightly annoying” to the Human Visual System of the subjects.

5.2 Suggestions for Further Research

The present work can be extended in the following areas:

- Both quantizer and Huffman tables can be optimized and optimal coefficient thresholding can be applied,
- Embedded bit coding of the quantized coefficients to achieve progressive transmission of the images,
- Currently the software operates for grayscale images of size 512 X 512 in the PGM format only. A more realistic version could be implemented for images of all types and of any size incorporating boundary treatment by using the classical extension methods such as periodic extension.

References

[1] Saha S, “Image Compression – from DCT to Wavelets: A review,” [URL http://www.acm.org/crossroads/xrds6-3/sahaimgcoding.htm].

[2] Sayood Khalid, “*Introduction to Data Compression,*” 2nd Edition, Morgan Kaufmann, Harcourt India Pvt. Ltd. (2000).

[3] Gonzalez, R.C. and Woods, R.E, “*Digital Image Processing*”. 2nd Edition, Pearson Education Asia. (2002).

[4] W. Sweldens, “The lifting scheme: A new philosophy in biorthogonal wavelet constructions,” In A. F. Laine and M. Unser, editors, *Wavelet Applications in Signal and Image Processing III*, pp. 68–79. Proc. SPIE 2569, 1995.

[5] Ingrid Daubechies and Wim Sweldens, "Factoring Wavelet Transforms into Lifting Steps," *J. Fourier Anal. Appl.*, 4 (no. 3), pp. 247-269, 1998.

[6] Geert Uytterhoven, *"Wavelets: software and applications*,” PhD thesis, April 1999, Catholic University of Leuven.

[7] W. Sweldens, “The lifting scheme: A custom-design construction of biorthogonal wavelets,” *Appl. Comput. Harmon. Anal.*, 3(2); pp. 186–200, 1996.

[8] Alan V. Oppenheim, Ronald Schaofer, “*Discrete-Time Signal Processing*”. PHI (1992).

[9] ISO IS-10918- Understanding JPEG Image Compression.

[10] Rao, R.M. and Bopardikar, A.S., “*Wavelet Transform*”. Pearson Education Asia, (2000).

[11] Application of Wavelets to image compression, [URL http://ouray,cudenver.edu/~mbhatia/Project%20Report_Web.htm].

[12] Polikar, Robi, "The Wavelet Tutorial Part I, II, III, IV," [URL http://www.public.iastate.edu/~rpolikar/wavelet/Wttutorial.htm].

[13] Polikar R., "Story of Wavelets," *Proc IEEE CSCC*, pp 5481-5486, 1999.

[14] *M. Antonini, M. Barlaud, P. Mathieu and I. Daubechie*s,"Image coding using Wavelet Transform," Image Processing, IEEE Transactions on, Volume: 1, Issue: 2, April 1992, Page(s): 205-220.

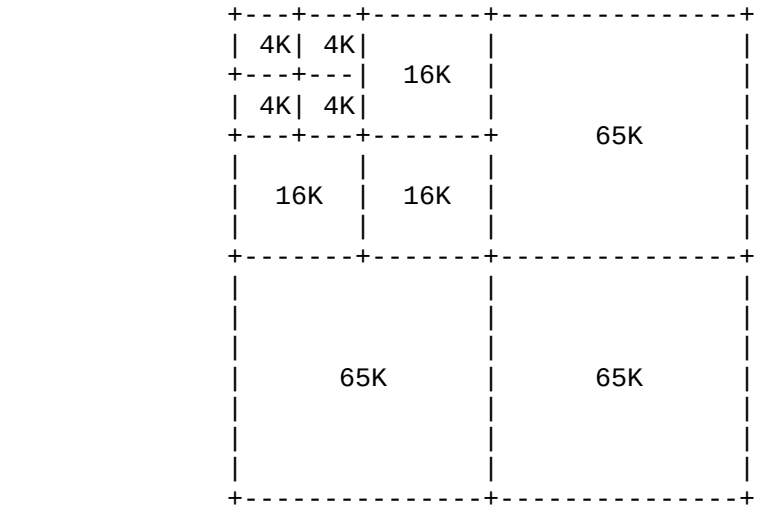
[15] W. Sweldens, "The lifting scheme: A construction of second generation wavelets," *SIAM J. Math. Anal*., 29(2); pp. 511–546, 1997.

[16] J.M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients", *IEEE Trans. Signal Processing,* vol 41, pp. 3445-3463, Dec 1993.

APPENDIX A

**********CompressImage**********

```
/*****************************************************************/
/*This program provides the routines required for a wavelet based  */
/* image compression of a 512 X 512 grayscale(8bit) image in the   */
/* PGM format.The structure of the program is similar to baseline  */
/* JPEG standard which is implemented in 4 basic steps: Wavelet-   */
/* -Transform, quantization, run length coding and entropy coding  */
/*Each of these steps is implemented as a function and is described*/
/* in the program. This program will expect the name of the PGM    */
/* image to be compressed as a required input. It will also accept */
/* an optional compression-factor (range 0-255)parameter specifying*/
/* how the image should be compressed : A compression factor of 0  */
/* indicates minimal compression and maximum image quality.        */
/* Conversely, a compression factor of 255 indicates maximum       */
/* compression with higher degradation to the image quality. If the*/
/* compression factor is not indicated, a default compression rate */
/* of 128 will be used. The compressed image file format is        */
/* specified in this program which is indicated by “InputImage.cmp”*/
/* file.                                                           */
/*****************************************************************/


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

#define NBLKS 10
#define BLKS_SAVED 7

#define IMG_SIZE 262146

#define BLOCK_SIZE 128
#define ZERO_MARK 16

#define min(a,b) ((a<b)?(a):(b))
#define max(a,b) ((a>b)?(a):(b))

/* I/O Variables */
#define DEFAULT_CRATIO 128         /* Default Compression Level */
char in_name[80],out_name[80];
int img_size, nbytes, cratio;

/* Wavelet variables */
#define ROW 1
#define COL 512
int int_data[IMG_SIZE];

/* Quantization variables */
/* The image will be divided into 10 blocks, the first four will be
of
   size 64x64 pixels (4096 pixels), then three of size 128x128 (16384
   pixels)   and the remaining three of size 256x256 pixels (65536
   pixels). These blocks will be arranged as represented below:
```




```
*/
int
blocksize[NBLKS]={4096,4096,4096,4096,16384,16384,16384,65536,65536,6
5536};
int blockthresh[NBLKS]= {0, 39, 27, 104, 79, 50,191, 99999, 99999,
99999};
int blockminval[NBLKS], blockmaxval[NBLKS];
int quant_buf[NBLKS][IMG_SIZE];
int thresh1 ,thresh2 ,thresh3 ,thresh4 ,thresh5 ,thresh6 ,thresh7
,thresh8;
int thresh9
,thresh10,thresh11,thresh12,thresh13,thresh14,thresh15,thresh16;

/* RLE variables */
unsigned char rle_buf[NBLKS][IMG_SIZE];
unsigned int rle_size[NBLKS];


/* Huffman Encoding variables */
unsigned char codep[IMG_SIZE];

/* Parameters for a fixed huffman encoding tree. These parameters are
*/
/* expected to work reasonably well with most images.
*/
int HufSize[256]={ };

int HufVal[256]={ };

/* Function Declarations */
void forward_wavelet();
void quantization();
void RLE_encode();
void entropy_encode();
void hufenc(unsigned char, int *);
int write_compressed_file(char *);
int read_image(char *);
```

```
/*****************************************************************
 * Routine: main(int argc, char **argv)
 * Expects image file name and the scale factor for compression as
 * input parameters. Calls all the required routines for compression.
 * The compressed file is the output of write_compressed_file().
 * This file format is specific to this program and includes all
image
 * information required for decompression.

*****************************************************************/


main(int argc, char **argv)
{
  initialize(argc, argv);
  read_image(in_name);
  forward_wavelet();
  quantization();
  RLE_encode();
  entropy_encode();
  write_compressed_file(in_name);
} /* End of Main() */


initialize(argc, argv)
int argc;
char **argv;
{
  int i;

  cratio = DEFAULT_CRATIO;
  if(argc==3) cratio = atoi(argv[2]);
  if(argc> 1) strcpy(in_name,argv[1]);

  if (argc < 2 || argc > 3 || cratio < 0 || cratio > 255 ){
    printf("Syntax: %s image.pgm [compression rate]\n",argv[0]);
    printf("        Where image.pgm is the name of the image to
be\n");
    printf("        compressed and the optional compression ratio
parameter\n");
    printf("        specifies how aggressively to compress the
image.\n");
    printf("        (O = least compression and  255 = highest
compression)\n");
    printf("        Examples: %s lena.pgm OR %s xyz.pgm 100\n"

,argv[0],argv[0]);
    exit(-1);
  }

  for(i=0; i<9; i++) blockthresh[i]=(blockthresh[i]*cratio)>>7;
}
```

```
/*********************************************************************
 * Routine: read_image(in_name)
 * read_image opens the specified image file, calculates the size
 * of image the imagE to be compressed (excluding the .pgm header),
and
 * makes a copy of the image as integers in array int_data.
*********************************************************************/
int read_image(in_name)
char *in_name;
{
  FILE *fp_in_image;
  unsigned char char_in[IMG_SIZE];
  int num_cols,num_rows,num_bits;
  int i;
  char p5[10];

  /* Open input image file */
  if ((fp_in_image = fopen (in_name, "rb")) == NULL) {
      fprintf (stderr, "\n Could not open file \"%s\"\n\n", in_name);
      return(1);
  }

  /* Skip .pgm image file header */
  fscanf(fp_in_image,"%s\n %d %d\n
%d\n",p5,&num_rows,&num_cols,&num_bits);

  /* Copy image bytes following the header into char array char_in */
  img_size = num_cols * num_rows;
  fread(char_in, img_size, 1, fp_in_image);
  fclose(fp_in_image);

  /* Copy into integer arrays int_data */
  for (i = 0; i < img_size; i++) int_data[i]  = char_in[i];
}



/*********************************************************************
 * Routine: void quantization()
*********************************************************************/
void quantization()
{
  int bl,num;

  num=0;
  block_quantize(0,0,BLOCK_SIZE/2,num++);
  for(bl=BLOCK_SIZE/2; bl<512; bl<<=1)
  {
    block_quantize(0,bl,bl,num++);
    block_quantize(bl,0,bl,num++);
    block_quantize(bl,bl,bl,num++);
  }
}
```

```
/*********************************************************************
 * Routine: block_quantize(int x, int y, int size, int num)
*********************************************************************/
block_quantize(int x, int y, int size, int num)
{
  int i,j,val;
  int minval,maxval;
  int qsize;

  minval = 10000; maxval =-10000;
  for(i=x; i<x+size; i++)
    for(j=y; j<y+size; j++)
    {
      val=int_data[i+512*j];
      minval = min(val,minval);
      maxval = max(val,maxval);
    }

  thresh1= minval+((1*(maxval-minval)+8)>>4);
  thresh2= minval+((2*(maxval-minval)+8)>>4);
  thresh3= minval+((3*(maxval-minval)+8)>>4);
  thresh4= minval+((4*(maxval-minval)+8)>>4);
  thresh5= minval+((5*(maxval-minval)+8)>>4);
  thresh6= minval+((6*(maxval-minval)+8)>>4);
  thresh7= minval+((7*(maxval-minval)+8)>>4);
  thresh8= minval+((8*(maxval-minval)+8)>>4);
  thresh9= minval+((9*(maxval-minval)+8)>>4);
  thresh10=minval+((10*(maxval-minval)+8)>>4);
  thresh11=minval+((11*(maxval-minval)+8)>>4);
  thresh12=minval+((12*(maxval-minval)+8)>>4);
  thresh13=minval+((13*(maxval-minval)+8)>>4);
  thresh14=minval+((14*(maxval-minval)+8)>>4);
  thresh15=minval+((15*(maxval-minval)+8)>>4);
  thresh16=minval+((16*(maxval-minval)+8)>>4);

  qsize=0;
  for(i=x; i<x+size; i++)
    for(j=y; j<y+size; j++)
    {
      val=int_data[i+512*j];
      if(abs(val)<blockthresh[num])
quant_buf[num][qsize++]=ZERO_MARK;
      else
      {
        quant_buf[num][qsize]=classify(val);
        qsize++;
      }
    }

  blockminval[num]=minval;
  blockmaxval[num]=maxval;
}
```

```
/*****************************************************************
 * Routine: int classify(int val)
*****************************************************************/
int classify(int val)
{
  if(val>thresh8) {
    if(val>thresh12) {
      if(val>thresh14) {
        if(val>thresh15) return 15;
                    else return 14;
      } else {
        if(val>thresh13) return 13;
                    else return 12;
      } } else {
      if(val>thresh10) {
        if(val>thresh11) return 11;
                    else return 10;
      } else {
        if(val>thresh9 ) return 9;
                    else return 8;
      } } } else {
    if(val>thresh4) {
      if(val>thresh6 ) {
        if(val>thresh7 ) return 7;
                    else return 6;
      } else {
        if(val>thresh5 ) return 5;
                    else return 4;
      } } else {
      if(val>thresh2 ) {
        if(val>thresh3 ) return 3;
                    else return 2;
      } else {
        if(val>thresh1 ) return 1;
                    else return 0;
      } } }
}


/*****************************************************************
 * Routine: void  RLE_encode()
*****************************************************************/
void  RLE_encode()
{
  unsigned int bl;

  for (bl = 0; bl < NBLKS; bl++)
    block_RLE_encode(bl);
}


/*****************************************************************
 * Routine: block_RLE_encode(int num)
*****************************************************************/
block_RLE_encode(int num)
{
  int i,j,count;

  i = j = 0;
  while(i<blocksize[num])
  {
```

```
    if(quant_buf[num][i]!=ZERO_MARK)
rle_buf[num][j]=quant_buf[num][i];
    else
    {
      count=0;
      while(quant_buf[num][i]==ZERO_MARK)
      {
        i++;
        count++;
        if((count==256-ZERO_MARK)||(i==img_size)) break;
      }
      i--;
      rle_buf[num][j]=count+ZERO_MARK-1;
    }
    j++;
    i++;
  }
  rle_size[num]=j;
}

/*********************************************************************
 * Routine: void entropy_encode()
*********************************************************************/
void entropy_encode()
{
  int i,j,nb;
  int rlecount;

  for(i=0; i<IMG_SIZE; i++) codep[i]=0;

  rlecount=0;
  nb=0;
  for (i = 0; i < BLKS_SAVED; i++)
  {
    for(j = 0; j < rle_size[i]; j++)
      hufenc(rle_buf[i][j],&nb);
    rlecount+=rle_size[i];
  }

  printf("Compressed size is %d bytes\n",(nb>>3)+88);
  nbytes = (nb/8);
}


/*********************************************************************
 * Routine: void hufenc(unsigned char ich, int *nb)
*********************************************************************/
void hufenc(unsigned char ich, int *nb)
{
  int i,nbits,val;
  int bytn,bitn;

  nbits=HufSize[ich];
  val=HufVal[ich];
  for(i=0; i<nbits; i++)
  {
    bytn=(*nb)>>3;
    bitn=(*nb)&7;
    if((val&(1<<i))>0) codep[bytn]|=(1<<bitn);
    (*nb)++;
  }
```

```
}


/*********************************************************************
 * Routine: int write_compressed_file()
*********************************************************************/
int write_compressed_file(in_name)
char *in_name;
{
  FILE *fp_out_file;

  if(strlen(in_name)>4)
    strncat(out_name,in_name,strlen(in_name)-4);
  else
    strcat(out_name,in_name);
  strcat(out_name,".cmp");


  if ((fp_out_file= fopen (out_name, "wb")) == NULL) {
      fprintf (stderr, "\n Could not open file %s \n\n",out_name);
      return(1);
  }

  fwrite(blockminval, sizeof(int) ,7     , fp_out_file);
  fwrite(blockmaxval, sizeof(int) ,7     , fp_out_file);
  fwrite(rle_size    , sizeof(int) ,7     , fp_out_file);
  fwrite(&nbytes     , sizeof(int) ,1     , fp_out_file);
  fwrite(codep       , sizeof(char),nbytes, fp_out_file);

  fclose(fp_out_file);
}

/*********************************************************************
 * Routine: void fcdf22(int x[], int n, int st)
*********************************************************************/
void fcdf22(int x[], int n, int st)
{
  int s[256],d[256];
  int mid, i;

  mid=(n/2)-1;

  for (i=0;i<=mid;i++)
  {
    s[i]=x[2*i*st];
    d[i]=x[2*i*st+st];
  }

  d[0]=d[0]+d[0]-s[0]-s[1];
  s[0]=s[0]+((d[0]+d[0])>>3);
  for (i=1;i<mid;i++)
  {
    d[i]=d[i]+d[i]-s[i]-s[i+1];
    s[i]=s[i]+((d[i-1]+d[i])>>3);
  }
  d[mid]=d[mid]+d[mid]-s[mid]-s[mid];
  s[mid]=s[mid]+((d[mid-1]+d[mid])>>3);

  for(i=0; i<=mid; i++)
  {
    x[i*st]=s[i];
```

```
    x[(i+mid+1)*st]=d[i];
  }
}

/*********************************************************************
 * Routine: void forward_wavelet()
 *********************************************************************/
void forward_wavelet()
{
  int i,nt;

  for (nt=512;nt>=BLOCK_SIZE;nt>>=1)
  {
    for (i=0;i<nt*512;i+=512) fcdf22(&int_data[i],nt,ROW);
    for (i=0;i<nt;i++) fcdf22(&int_data[i],nt,COL);
  }
}
```

**********Decompress Image**********

```
/*****************************************************************/
/* This program provides the routines required for a wavelet based */
/* image compression of a 512 X 512 grayscale(8bit) image in the   */
/* PGM format. The structure of the program is similar to baseline */
/* JPEG standard which is implemented in 4 basic steps: reverse-   */
/* -Wavelet Transform, dequantization, run length decoding and     */
/* entropy decoding. Each of these steps is implemented as function*/
/* and is described in the program. This program will expect thname*/
/* of the ".cmp" file to be decompressed and the output PGM file aa*/
/* required input.                                                 */
/*****************************************************************/


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

#define NUMROWS 512
#define NUMCOLS 512

#define BLOCKS_SAVED 7
#define GL_INC 4.0

#define IMG_SIZE 262146

#define BLOCK_SIZE 128
#define ZERO_MARK 16

/* I/O Variables */
int img_size,num_cols,num_rows,nbytes;
char in_name[80], out_name[80];

/* Wavelet variables */
int int_data[IMG_SIZE];



/* Quantization variables */
/*
The image will be divided into 10 blocks, the first four will be of
size 64x64 pixels (4096 pixels), then three of size 128x128 (16384
pixels)and the remaining three of size 256x256 pixels (65536 pixels).
These blocks will be arranged as represented below
```

```
   +---+---+-------+---------------+
   | 4K| 4K|       |               |
   +---+---|  16K  |               |
   | 4K| 4K|       |               |
   +---+---+-------+      65K      |
   |       |       |               |
   |  16K  |  16K  |               |
   |       |       |               |
   +-------+-------+---------------+
   |               |               |
   |               |               |
   |               |               |
   |      65K      |      65K      |
   |               |               |
   |               |               |
   |               |               |
   +---------------+---------------+
*/
int quant_buf[10][IMG_SIZE];
int block_minval[10],block_maxval[10];

/* RLE variables */
unsigned char rle_buf[20][IMG_SIZE];
unsigned int rle_size[20];

/* Huffman variables */
#define HUFTABSIZE 2000000
int htable[HUFTABSIZE];
unsigned char codep[IMG_SIZE];

int
HufSize[256]={ };

int
HufVal[256]={ };


void reverse_wavelet();
void dequantization();
void RLE_decode();
void hufinit();
void entropy_decode();
unsigned char hufdec(unsigned long *nb);
int read_format_file(void);
int write_image(int argc, char **argv);

/*****************************************************************
 * Routine: main(int argc, char **argv)
 * The expected input is the name of the file to be decompressed. The
 * name of the output ".pgm" file is an optional input. It will calls
 * all routines required for decompression.
*****************************************************************/
main(int argc, char **argv)
{
  initialize(argc, argv);
  read_compressed_file();
  entropy_decode();
  RLE_decode();
  dequantization();
  reverse_wavelet();
```

```
  write_image(argc, argv);
}


/*************************************************************************
 * Routine: write_image(int argc, char **argv)
 * write_image first writes the PGM format file header to the user
 * specified file and then appends the decompressed image from array
 * data to output file.
*************************************************************************/
int write_image(int argc, char **argv)
{
  FILE *fp_out_image;
  unsigned char data[IMG_SIZE];
  int i;
  char p5[10];

 /* Open user specified file for writing */
  if ((fp_out_image = fopen (out_name, "wb")) == NULL) {
      fprintf (stderr, "\n Could not open file \"%s\"\n\n",
               argv[argc-1]);
      return(1);
  }

 /* Write PGM format header in output file */
  fprintf(fp_out_image,"P5\n");
  fprintf(fp_out_image,"%d %d\n", num_cols, num_rows);
  fprintf(fp_out_image,"%d\n", 255);

 /* Convert image from integer bytes to char bytes */
  for (i = 0; i < img_size; i++)
  {
     if(int_data[i]<0.0) data[i]=0;
     else if(int_data[i]> 255.0) data[i]=255;
     else data[i] = int_data[i];
  }

  /* Write image in output file */
  fwrite(data, sizeof(unsigned char), img_size, fp_out_image);

  fclose(fp_out_image);
}

initialize(argc, argv)
int argc;
char **argv;
{
  if(argc<2 || argc >3)
  {
    printf("Syntax: %s image.cmp [image.pgm]\n",argv[0]);
    printf("        Where image.pgm is the name of the compressed");
    printf(" image file to be\n        expanded. The optional second
argument");
    printf(" specifies the name of the\n        output pgm file that
will be");
    printf(" written by this program.\n");
    printf("        Examples: %s lena.cmp OR %s lena.cmp new.pgm\n",

argv[0],argv[0]);
    exit(-1);
  }
```

```
  strcpy(in_name,argv[1]);
  if(argc==3) strcpy(out_name,argv[2]);
  else
  {
    strncat(out_name,in_name,strlen(in_name)-3);
    strcat(out_name,"pgm");
  }

}



/*****************************************************************
 * Routine: block_dequantize(int x, int y, int size, int num)
*****************************************************************/
block_dequantize(int x, int y, int size, int num)
{
  int i,j,ind;
  int minval,maxval,qsize;
  float diff,findex;
  int thresh[17];

  minval=block_minval[num];
  maxval=block_maxval[num];

  diff=(float)(maxval-minval);
  findex=0.5;
  for(i=0; i<16; i++)
  {
    thresh[i]=minval+(int)(GL_INC+(findex*diff)/16.0);
    findex+=1.0;
  }

  if(num>=BLOCKS_SAVED)
  {
    for(i=x; i<x+size; i++)
      for(j=y; j<y+size; j++)
      {
        ind=i+512*j;
        int_data[ind]=0;
      }
  }
  else
  {
    qsize=0;
    for(i=x; i<x+size; i++)
      for(j=y; j<y+size; j++)
      {
        ind=i+512*j;
        if(quant_buf[num][qsize]==ZERO_MARK) int_data[ind]=0;
        else int_data[ind]=thresh[quant_buf[num][qsize]];
        qsize++;
      }
  }
}
```

```
/*********************************************************************
 * Routine: void dequantization()
*********************************************************************/
void dequantization()
{
  int bl,num,i;

  num=0;
  for(i=0; i<img_size; i++) int_data[i]=0;
  block_dequantize(0,0,BLOCK_SIZE/2,num++);
  for(bl=BLOCK_SIZE/2; bl<512; bl<<=1)
  {
    block_dequantize(0,bl,bl,num++);
    block_dequantize(bl,0,bl,num++);
    block_dequantize(bl,bl,bl,num++);
  }
}

/*********************************************************************
 * Routine: void entropy_decode()
*********************************************************************/
void entropy_decode()
{
  unsigned long nb;
  int i,j;

  hufinit();

  nb=0;
  for (i = 0; i < BLOCKS_SAVED; i++)
    for(j = 0; j < rle_size[i]; j++)
    {
      rle_buf[i][j]=hufdec(&nb);
    }
}

/*********************************************************************
 * Routine: void  RLE_decode()
*********************************************************************/
void  RLE_decode()
{
  unsigned int bl;

  for (bl = 0; bl < 10; bl++)
    block_RLE_decode(bl);
}


/*********************************************************************
 * Routine: block_RLE_decode(int num)
*********************************************************************/
block_RLE_decode(int num)
{
  int i,j,value,count;

  j=0;
  for(i=0; i<rle_size[num]; i++)
  {
    value=rle_buf[num][i];
    if(value<ZERO_MARK) quant_buf[num][j++]=value;
    else for(count=0; count<=(value-ZERO_MARK); count++)
```

```
                                        quant_buf[num][j++]=ZERO_MARK;
  }
}

/*********************************************************************
 * Routine: unsigned char hufdec(unsigned long *nb)
*********************************************************************/
unsigned char hufdec(unsigned long *nb)
{
  int val,bytn,bitn,bitv;

  val=1;
  while(htable[val]<0)
  {
    bytn=(*nb)>>3;
    bitn=(*nb)&7;

    bitv=((codep[bytn]&(1<<bitn))>0);
    val=(val<<1)+bitv;

    (*nb)++;
  }

  return(htable[val]);
}


/*********************************************************************
 * Routine: void hufinit()
*********************************************************************/
void hufinit()
{
  int i,j,val,huf;

  for(i=0; i<HUFTABSIZE; i++) htable[i]=-1;

  for(i=0; i<256; i++)
  {
    val=1;
    huf=HufVal[i];
    for(j=0; j<HufSize[i]; j++)
    {
      val=(val<<1)+(huf&1);
      huf>>=1;
    }
    htable[val]=i;
  }
}

/*********************************************************************
 *  Routine: int read_compressed_file(void)
*********************************************************************/
int read_compressed_file(void)
{
FILE *fp_fmt_file;
int i;

  if ((fp_fmt_file= fopen (in_name, "rb")) == NULL) {
      fprintf (stderr, "\n Could not open file file.fmt \n\n");
      return(1);
  }
```

```
    num_rows=NUMROWS;
    num_cols=NUMCOLS;
    img_size=num_rows*num_cols;

    fread(block_minval, sizeof(int),7, fp_fmt_file);
    fread(block_maxval, sizeof(int),7, fp_fmt_file);
    fread(rle_size, sizeof(int), 7, fp_fmt_file);
    fread(&nbytes, sizeof(int), 1, fp_fmt_file);
    fread(codep, sizeof(char), nbytes, fp_fmt_file);

    fclose(fp_fmt_file);
}


/*********************************************************************
 *  Routine: void bcdf22(int x[], int n, int st)
*********************************************************************/
void bcdf22(int x[], int n, int st)
{
  int s[256],d[256], mid, i;

  mid=(n/2)-1;

  for (i=0;i<=mid;i++)
  {
    s[i]=x[i*st];
    d[i]=x[(i+mid+1)*st];
  }

  s[mid]=s[mid]-((d[mid-1]+d[mid])>>3);
  d[mid]=d[mid]+s[mid]+s[mid];
  for (i=mid-1; i>0; i--)
  {
    s[i]=s[i]-((d[i-1]+d[i])>>3);
    d[i]=d[i]+s[i]+s[i+1];
  }
  s[0]=s[0]-((d[0]+d[0])>>3);
  d[0]=d[0]+s[0]+s[1];

  for(i=0; i<=mid; i++)
  {
    x[2*st*i]=s[i];
    x[2*st*i+st]=d[i]/2;
  }
}

/*********************************************************************
 *  Routine: void reverse_wavelet()
*********************************************************************/
#define ROW 1
#define COL 512
void reverse_wavelet()
{
  int i,nt;
  for (nt=BLOCK_SIZE;nt<=512;nt<<=1)
  {
    for (i=0;i<nt;i++) bcdf22(&int_data[i],nt,COL);
    for (i=0;i<nt*512;i+=512) bcdf22(&int_data[i],nt,ROW);
  }
}
```

**********Compare Image**********

```
/*****************************************************************/
/* This program is used to perform the quality test of the       */
/* reconstructed image. It will compare the original ".pgm image  */
/* file with the decompressed ".pgm" image file. It will report the*/
/* PSNR and the maximum pixel value difference.                  */
/*****************************************************************/

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

#define IMG_SIZE 262146

#define max(a,b) ((a>b)?(a):(b))

int orig_data[IMG_SIZE], int_data[IMG_SIZE];
int img_size_orig,img_size_decomp;
int num_cols,num_rows;

/*****************************************************************
 * Routine: main(int argc, char **argv)
 * Rounite main expects the image files as inputs for the acceptance
 * tests. The Root Mean Square Eror and PSNR is defined in chapter 1.
*****************************************************************/
main(int argc, char **argv)
{
  FILE *fp_in_image,*fp_out_image;
  unsigned char char_in[IMG_SIZE];
  char p5[10];

  int i;
  int num_bits;
  double j,square,sum,mean,max_square,max_diff;
  double root, psnr;

  if (argc != 3 || argc > 3 ){
    printf("Please specify image files to be compared.\n");
    printf("For example: %s original.pgm decompressed.pgm
\n",argv[0]);
    exit(0);
  }

  /* Open original .pgm file */
  if ((fp_in_image = fopen (argv[argc-2], "rb")) == NULL) {
      fprintf (stderr, "\n Could not open file \"%s\"\n\n",
argv[argc-2]);
      return(1);
  }

  /* Open compressed and then decompressed .pgm file */
  if ((fp_out_image = fopen (argv[argc-1], "rb")) == NULL) {
      fprintf (stderr, "\n Could not open file \"%s\"\n\n",
argv[argc-1]);
      return(1);
  }

  /* Skip pgm file header of the original file*/
```

```
  fscanf(fp_in_image,"%s\n %d %d\n
%d\n",p5,&num_rows,&num_cols,&num_bits);

  img_size_orig = num_cols * num_rows;
  fread(char_in, img_size_orig, 1, fp_in_image);
  fclose(fp_in_image);

  /* Copy image into integer array orig_data */
  for (i = 0; i < img_size_orig; i++)
  {
    orig_data[i] = char_in[i];
  }

  /* Skip pgm file header for compressed and then decompressed file*/
  fscanf(fp_out_image,"%s\n %d %d\n
%d\n",p5,&num_rows,&num_cols,&num_bits);

  img_size_decomp = num_cols * num_rows;
  fread(char_in, img_size_decomp, 1, fp_out_image);
  fclose(fp_out_image);


/* Copy image into integer array int_data */
  for (i = 0; i < img_size_decomp; i++)
  {
    int_data[i] = char_in[i];
  }


/* Compute Root Mean Square Error (RMSE) between original
     image in orig_data and decompressed image in int_data */

  sum = max_square = max_diff = 0;

  for (i = 0; i < img_size_orig; i++)
  {
    j = (double) (orig_data[i]-int_data[i]) ;
    square = j * j ;
    sum += square;
    max_square = max(square,max_square);
    max_diff= max(fabs(orig_data[i]-int_data[i]),max_diff);
  }
  mean = sum / img_size_orig;
  root = sqrt(mean);

 /* Compute Peak Signal-To-Noise Ratio (PSNR) */
  psnr = consta * (log10(255.0/root));
    printf("Signal to Noise Ratio (PSNR) = %f\n", psnr);

}
```

**********HUFFMAN ARRAYS**********

```
Int Hufsize[256]=

        { 9 , 7 , 6 , 5 , 4 , 4 , 3 , 4 , 3 , 3 , 4 , 5 , 6 , 6 , 7 , 8 ,
          4 , 5 , 5 , 6 , 6 , 7 , 7 , 7 , 8 , 8 , 8 , 8 , 9 , 9 , 9 , 9 ,
          9 , 9 , 9 ,10 , 9 ,10 ,10 ,10 ,10 ,10 ,10 ,10 ,10 ,10 ,10 ,11 ,
         10 ,11 ,11 ,10 ,10 ,11 ,11 ,11 ,11 ,11 ,11 ,12 ,12 ,12 ,12 ,11 ,
         11 ,12 ,11 ,12 ,12 ,12 ,12 ,11 ,13 ,12 ,12 ,12 ,12 ,13 ,12 ,13 ,
         13 ,13 ,14 ,13 ,13 ,14 ,13 ,13 ,13 ,13 ,14 ,13 ,13 ,13 ,13 ,13 ,
         13 ,13 ,13 ,13 ,14 ,13 ,14 ,13 ,13 ,14 ,15 ,13 ,13 ,14 ,13 ,14 ,
         13 ,14 ,13 ,14 ,14 ,14 ,14 ,13 ,13 ,13 ,13 ,14 ,13 ,13 ,13 ,13 ,
         14 ,13 ,15 ,14 ,15 ,14 ,14 ,13 ,14 ,16 ,14 ,14 ,13 ,12 ,12 ,13 ,
         15 ,15 ,17 ,18 ,18 ,18 ,17 ,17 ,15 ,17 ,18 ,16 ,17 ,17 ,16 ,16 ,
         18 ,18 ,16 ,18 ,16 ,18 ,18 ,18 ,16 ,16 ,16 ,18 ,15 ,16 ,18 ,15 ,
         16 ,18 ,16 ,16 ,18 ,18 ,18 ,18 ,16 ,18 ,16 ,18 ,18 ,18 ,16 ,18 ,
         18 ,18 ,18 ,18 ,18 ,18 ,18 ,18 ,16 ,16 ,18 ,18 ,17 ,18 ,18 ,18 ,
         18 ,17 ,15 ,18 ,18 ,15 ,15 ,18 ,17 ,16 ,16 ,15 ,16 ,16 ,18 ,18 ,
         16 ,17 ,18 ,18 ,15 ,18 ,16 ,16 ,18 ,18 ,18 ,18 ,16 ,18 ,18 ,16 ,
         18 ,16 ,18 ,18 ,18 ,18 ,18 ,18 ,18 ,16 ,18 ,18 ,17 ,18 ,18 , 9 };

int
HufVal[256]=
{0x00097,0x00066,0x00007,0x00016,0x00000,0x0000B,0x00001,0x0000F,
0x00002,0x00004,0x0000E,0x00015,0x00037,0x00025,0x00038,0x000ED,
0x00003,0x0001D,0x00008,0x0000D,0x00018,0x00067,0x00005,0x00006,
0x000D7,0x0002D,0x000C6,0x000F8,0x001A7,0x00197,0x0016D,0x00127,
0x001C5,0x001A6,0x00146,0x00357,0x00178,0x00257,0x000AD,0x0006D,
0x000A7,0x003AD,0x001AD,0x00017,0x000C5,0x002C5,0x00145,0x00117,
0x000A6,0x00427,0x002A7,0x00246,0x00226,0x002AD,0x00217,0x00027,
0x006A6,0x002A6,0x00445,0x00A27,0x00A6D,0x00717,0x00D57,0x00345,
0x00446,0x00057,0x00726,0x006A7,0x00026,0x00EA7,0x00E17,0x00126,
0x0026D,0x00D26,0x006AD,0x00B17,0x00845,0x00F17,0x00317,0x00617,
0x01457,0x01326,0x02526,0x00B26,0x01D17,0x01F17,0x00627,0x01C57,
0x0166D,0x00E6D,0x00957,0x01E6D,0x01E45,0x01526,0x00517,0x01846,
0x00745,0x01617,0x00157,0x01627,0x00E27,0x01EAD,0x00227,0x01745,
0x00245,0x01957,0x06957,0x01517,0x01157,0x0326D,0x01645,0x02557,
0x00A45,0x03957,0x01046,0x03045,0x00457,0x02227,0x00557,0x01A45,
0x00046,0x00C57,0x00326,0x02457,0x00EAD,0x01B26,0x00645,0x01227,
0x01426,0x0066D,0x02E45,0x01557,0x03F17,0x02E27,0x03426,0x01245,
0x00D17,0x02957,0x03557,0x02D17,0x00045,0x00857,0x00F45,0x00846,
0x0526D,0x00426,0x1A957,0x06C26,0x26C26,0x14426,0x1D045,0x09E27,
0x00526,0x19E27,0x04426,0x06E45,0x05E27,0x15E27,0x02C26,0x04526,
0x24426,0x0CC26,0x0EE45,0x2CC26,0x0C526,0x16C26,0x36C26,0x0AC26,
0x07E27,0x01826,0x09826,0x2AC26,0x00C26,0x05826,0x0A826,0x01045,
0x0D826,0x2A826,0x04C26,0x0CE45,0x1A826,0x3A826,0x1AC26,0x3AC26,
0x0C426,0x1CC26,0x0126D,0x1EC26,0x3EC26,0x1DC26,0x05C26,0x3DC26,
0x0DC26,0x2DC26,0x1C826,0x3C826,0x04826,0x24826,0x0C826,0x2C826,
0x03826,0x0B826,0x16426,0x36426,0x0DE27,0x06426,0x0E826,0x2E826,
0x14826,0x1DE27,0x03E27,0x34826,0x0A426,0x07826,0x00826,0x2A426,
0x0FE27,0x0BC26,0x02826,0x00E45,0x07C26,0x0926D,0x05045,0x25045,
0x0FC26,0x1FE27,0x15045,0x35045,0x07F17,0x1A426,0x06826,0x0E426,
0x3A426,0x19C26,0x39C26,0x09C26,0x02426,0x29C26,0x26426,0x01C26,
0x13C26,0x01E27,0x33C26,0x0D045,0x2D045,0x03C26,0x3CC26,0x23C26,
0x1E826,0x04E45,0x3E826,0x0EC26,0x0A957,0x2EC26,0x34426,0x00078};
```

LIST OF FIGURES

LIST OF TABLES