INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Performance Evaluation of Filter Bank Algorithms in Image Compression

by

Shixin Wei

**A Thesis Submitted to the
Faculty of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of**

MASTER OF SCIENCE

**Department of System and Computer Engineering
Faculty of Engineering
Carleton University,
Ottawa, Ontario, Canada**

July 23, 2001

© 2001, Shixin Wei

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-68327-3

Canada

The undersigned recommended to
the Faculty of Graduate Studies and Research
acceptance of the thesis

"Performance Evaluation of Filter Bank Algorithms
in Image Compression"

Submitted by Shixin Wei, B.Sc. M.Sc.
in partial fulfillment of the requirement for
the Degree of Master of Science

Department Chair

Thesis Supervisor

Abstract

In this thesis, image coding using multirate filter banks and wavelet systems are evaluated and compared. The coding systems considered are based on the M -channel general lapped biorthogonal transforms (GLBT) and the wavelet-based subband coding schemes, as well as the baseline JPEG algorithm. The principles behind each algorithm are briefly described. The thesis concentrates on both coding efficiency and complexity. The tradeoff between efficiency and complexity of each coding system has also been analyzed.

From the analysis of lossy image coding results, we find that the GLBT algorithm provides a compromise solution between the JPEG algorithm and the wavelet-based scheme due to its characteristics of moderate efficiency and complexity. Therefore, the GLBT algorithm is an alternative solution to current image coding standards. On the other hand, JPEG is the fastest algorithm and therefore presumably the least complex. The performance of several wavelet-based transforms is also evaluated on the same metrics. It reveals that the SPIHT algorithm which is the basis of the JPEG-2000 standard provides the most effective and flexible solution. Through the implementation of our software package, which is based on the EZW algorithm, it is worth noting that the main factors in image coding are the quantizer and entropy coder rather than the transforms only. The high performance of the wavelet algorithm is offset by its computing complexity. The coding results also show that the choice of an image coding scheme for a particular application depends strongly on its requirements.

Acknowledgments

I would like to express my deepest gratitude to Professor Howard M. Schwartz for his invaluable advice and inspirational supervision during the course of this research. His guidance has been instrumental in success of this work. I am also indebted to him for his support, patience and friendship. Next, I would like to express my gratitude to my co-supervisor Dr. Delphine Rossille for her invaluable suggestions and comments during the research. She has been a constant source of inspiration. Sincere thanks are also extended to Prof. Eric Dubois, Prof. David C. Coll and Dr. Victor Aitken for their interests and suggestions and serving as committee member.

I would like to thank my family and friends for their moral supports. In particular, I would like to thank my wife, Xinan, for her support and desire to help, my daughter, Jane, for her smile every morning which always boosted my spirits, and my parent, for their constant encouragement. To the extended, sincere gratitude is also expressed to my friends Shuchun Du, Ping Fu, Shudao Ni and Jiren Zhang for their moral supports.

I would also like to thank all the people at Carleton University I have had the pleasure to meet. I especially thank Dr. Trevor W. Pearce for his excellent courses on real-time systems. I also like to thank Yang Wang for being a good friend and constructive discussion. Thanks are also extended to Dongyang Zhang, Zhaohuan Liu, Xuqin Wu, Gang Ji and Zhiying Guo for their help during my studies.

I am indebted to the Department of System and Computer Engineering and Carleton University for providing necessary facilities and fund for this research, which makes this research possible.

"Nothing in nature is random ... A thing appears random only through the incompleteness of our knowledge."

_____ Spinoza, *Ethics I*

"That which shrinks must first expand."

_____ Lao Tzu, *Tao Te Ching*

Table of Contents

Abstract	ii
Acknowledgments	iii
Table of Contents	v
List of Figures	ix
List of Tables	xii
List of Abbreviations	xiii
 Chapter 1 Introduction	
1.1 Problem Statement.....	1
1.2 Overview and Contribution of the Thesis.....	3
1.3 Notation.....	5
 Chapter 2 Principles behind Compression	
2.1 Introduction	7
2.2 Principles of Image Compression.....	8
2.3 Classification of Compression Techniques	8
2.4 Image Coding Strategies	9
2.4.1 Source Encoder/Decoder	10
2.4.2 Quantization	11
2.4.3 Entropy Encoder	13
2.4.4 Bit Allocation	14
2.5 Comparison Methodology.....	15
2.6 DCT-based JPEG Image Coding	16

2.7	JPEG-2000: New Image Compression Standard	20
-----	---	----

Chapter 3 Multirate and Wavelet Signal Processing

3.1	Introduction	21
3.2	Multirate System and Basic Building Blocks	21
3.2.1	Downsampling and Upsampling	22
3.2.2	Noble Identities	27
3.3	The Polyphase Representation	28
3.4	Multirate Filter Banks and Subband Coding	32
3.4.1	Filter Banks	32
3.4.2	M -Channel QMF Banks	35
3.4.3	Perfect Reconstruction of QMF Banks	36
3.5	Implementation of QMF Banks.....	37
3.5.1	Polyphase Form of QMF Banks.....	37
3.5.2	Conditions for PR System	40
3.5.3	QMF Bank Implementation	43
3.6	QMF Banks and Wavelet Analysis.....	44
3.6.1	QMF Banks and Series Expansions of Signals.....	44
3.6.2	Octave-band Filter Banks and Wavelet Transform.....	48
3.6.3	Filter Bank Implementation of Wavelet Transform	50
3.7	Design Issues of Filter Bank System.....	52
3.7.1	Basis Selection	52
3.7.2	Multidimensional Signals	53
3.7.3	Finite Length Signals	54

Chapter 4 Multirate Filter Bank System in Image Coding

4.1	Introduction	58
4.2	Block Transforms	59
4.2.1	Linear-phase Perfect Reconstruction	59
4.2.2	Definition of Block Transform	60

4.2.3	JPEG-based DCT	63
4.3	General Lapped Transform	65
4.3.1	Problem Formulation	65
4.3.2	Lapped Transform	68
4.3.3	Lattice Structure Implementation	70
4.4	Linear-phase Biorthogonal Filter Banks	74
4.4.1	Bases of Coefficient Matrix	74
4.4.2	Design Issues in Image Coding	75
4.5	Coding Results and Analysis	77
4.5.1	Coding and Implementation	77
4.5.2	Results	78
4.5.3	Discussion	81

Chapter 5 Wavelet-based Image Compression

5.1.	Introduction	82
5.2.	Review of the Wavelet Transform	83
5.2.1	Definition of Wavelet Transform	83
5.2.2	Discrete Wavelet Transform	85
5.3	The DWT for Image Compression	87
5.3.1	Basic Concept of Wavelet Transform in Image Coding	87
5.3.2	The Inverse DWT of an Image	90
5.4	Advanced Wavelet Coding Schemes	90
5.4.1	Embedded Zerotree Wavelet (EZW) Algorithm	91
5.4.2	Set Partitioning in Hierarchical Trees (SPIHT) Algorithm	93
5.4.3	Scaleable Image Compression with EBCOT	96
5.5	Results and Analysis	97
5.5.1	Coding and Implementation	97
5.5.2	Results	99
5.5.3	Discussion	101

Chapter 6 Conclusions and Future Research

6.1 Summary	104
6.2 Future Research	107

References	108
-------------------------	-----

Appendix A Proof

A.1 Proof of general PR condition.....	115
---	-----

Appendix B Test Images

B.1 Overview	117
-----------------------	-----

Appendix C Source Codes

C.1 Overview	118
C.2 Source Codes.....	119

List of Figures

Chapter 2

2.1	A typical lossy image encoder.....	10
2.2	A uniform scalar quantizer	12
2.3	The distortion function vs. the effect of adding a bit.....	15
2.4	JPEG encoder block diagram	17
2.5	JPEG decoder block diagram	18
2.6	Quantization table	19
2.7	Zig-Zag sequence	19

Chapter 3

3.1	M -fold downsampler	22
3.2	Demonstration of downsampling for $M=2$	23
3.3	Transform domain effects of the upsampler and downsampler	24
3.4	M -fold upsampler	25
3.5	Demonstration of upsampling for $M=2$	25
3.6	(a) Decimation filter, and (b) Interpolation filter	26
3.7	Multirate identities	27
3.8	Schematic of relationship between $h[n]$ and its l -th polyphase components	30
3.9	Polyphase realization of a filter (a) Type 1, (b) Type 2	31
3.10	Analysis and synthesis filter banks	33
3.11	Three types of frequency band partitions in filter banks	34
3.12	M -channel QMF bank	35
3.13	Type 1/2 polyphase form of an M -channel QMF bank	40
3.14	Filter bank equivalent circuit.....	42
3.15	Revised M -channel QMF bank	45
3.16	An octave-band filter bank with J stages.....	48

3.17	Octave-band synthesis filter bank with Haar filters and 3 stages.....	49
3.18	Splitting the signal spectrum with an octave-band filter bank.....	50
3.19	Types of signal symmetries	57
3.20	(1,1) – Symmetric extension example	57

Chapter 4

4.1	M-channel maximally-decimated QMF bank	59
4.2	Signal sampling and block transform	62
4.3	Flow graph representation of fast 1-D DCT for $M=8$	65
4.4	Basic functions of the LOT overlap across block boundaries	66
4.5	Example of lapped orthogonal transform (LOT).....	68
4.6	Implementation of a more general version of the LOT for $M=8$	72
4.7	GLBT-based encoder/decoder processing steps	77
4.8	Comparative evaluation of GLBT in compression	79
4.9	Comparative evaluation of GLBT in coding time	79
4.10	Original Lena (512x512x8)	80
4.11	JPEG-DCT: CR = 32, PSNR = 24.4	80
4.12	GLBT: CR = 15.6, PSNR = 35.8	80
4.13	GLBT: CR = 28.8, PSNR = 32.7	80
4.14	GLBT: CR = 55.2, PSNR = 29.1	80
4.15	GLBT: CR = 106.6, PSNR = 25.0	80

Chapter 5

5.1	The wavelet transform implemented with a bank of filters	85
5.2	Octave-based discrete wavelet transform	88
5.3	(a) Two level octave-based decomposition of Lena image, and (b) Spectral decomposition and ordering	89
5.4	(a) Structure of zerotrees, and (b) Scanning order of subbands for encoding	92
5.5	Comparative evaluation of coding results for image 'Lena'	100
5.6	Comparative evaluation of coding results for image 'Barbara'	100

5.7	Performance of different wavelet-based coders operating on 'Lena'	102
5.8	Performance of different wavelet-based coders operating on 'Barbara'	103

Appendix A

A1.	The equivalent circuit for QMF filter bank.....	115
-----	---	-----

Appendix B

B1.	Image "Lena"	117
B2.	Image "Barbara"	117

List of Tables

Chapter 4

4.1	DCT bases example for $M=8$. The even basis vectors are symmetric while the odd ones are anti-symmetric	64
4.2	Forward GLBT bases example for $M=8$ and $N=2$. The even basis vectors are symmetric while the odd ones are anti-symmetric.....	76
4.3	Inverse GLBT bases example for $M=8$ and $N=2$. The even basis vectors are symmetric while the odd ones are anti-symmetric.....	76
4.4	Coding results for 512x512x8 bit image "Lena".....	79

Chapter 5

5.1	The coefficients of the 9/7 biorthogonal wavelet transform.....	90
5.2	Results of different coding methods (512x512, 8-bpp grayscale 'Lena' image).....	100
5.3	Results of different coding methods (512x512, 8-bpp grayscale 'Barbara' image)...	100

List of Abbreviations

bpp	bits per pixel
BR	Bit Rate
CR	Compression Ratio
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DWT	Discrete Wavelet Transform
EZW	Embedded Zero-Tree Wavelet (Coding)
FFT	Fast Fourier Transform
FIR	Finite-length Impulse Response
GLBT	General Lapped Biorthogonal Transform
HVS	Human Visual System
IIR	Infinite-length Impulse Response
ISO	International Standards Organization
JPEG	Joint Photographic Experts Group
LRPRFB	Linear-phase Perfect Reconstruction Filter Bank
LOT	Lapped Orthogonal Transform
MRA	MultiResolution Analysis
PE	Periodic Extension
PR	Perfect Reconstruction
PSNR	Peak-Signal-to-Noise Ratio
QMF	Quadrature Mirror-image Filter
SBC	SubBand Coding
SE	Symmetric Extension
SPIHT	Set Partitioning In Hierarchical Trees
SOT	Spatial Orientation Tree

Chapter 1

Introduction

1.1 Problem Statement

Over the last two decades, there has been a steady growth of interest in the study of image and video compression. This growth was driven by the wide range of applications in multimedia communication, medical imaging, remote sensing, and industrial inspection, to name a few. For still image compression, JPEG standards have been established by the ISO[26]. The performance of these coders generally degrades at low bit-rates mainly because of the underlying block-based Discrete Cosine Transform (DCT)[29] scheme. Different techniques have been explored to overcome this deficiency. Wavelet-based systems and multirate filter bank-based systems are two main but interrelated developments.

Wavelet transforms have been proven to be extremely useful for image coding as has been shown in many publications (e.g., [14], [18], [43], [46]). Wavelet-based coding [2] provides substantial improvements in picture quality at higher compression ratios. Its theoretical basis was founded mainly by Mallat [22] and Daubechies [7] in the early 1990's. Over the past few years, a variety of powerful and sophisticated wavelet-based schemes for image compression have been developed. In 1993, Shapiro [31] first introduced the notion of embedded zerotree coding. His coding scheme, called Embedded

Zerotree Wavelet (EZW) coding, was based on wavelet decomposition. In 1996, Said and Pearlman [30] introduced a new coding scheme known as Set Partitioning in Hierarchical Trees (SPIHT) which is similar in spirit to EZW. Taubman [35] developed the concept of scalable image compression with EBCOT in 1999. Due to the obvious advantages of the embedding property in many applications, wavelet-based embedded coding quickly grew in popularity --- the new JPEG-2000 standard [15] adopts wavelet-based algorithms.

Other developments in image coding research include multirate analysis-synthesis systems pioneered by Malvar [20], Vaidyanathan [38], Vetterli and Kovacevic [40], Strang and Nguyen [33], Fliege [12], Suter [34], and so on, commonly called quadrature mirror filter (QMF) banks. In such systems the input signal is divided into channels using band-pass filters, and the individual channels are downsampled to rates appropriate to the individual channel bandwidths. Research in this area has been motivated largely by the interest in compressing and coding speech, image, and video signals for fast transmission or storage. Subband coders are considered among the best methods available and have been adopted in many new international coding standards [3].

Although there are many important linkages between block transform-based filter bank theory and wavelet-based subband systems, the two techniques were developed independently for a long time. Digital filter bank methods have a somewhat shorter history than wavelet theory. The linkage between filter bank theory and wavelet theory was not established until the early 1990s [27]. Although a lot of research has been carried out on different methods, a comprehensive performance evaluation of the two systems on the framework of coding efficiency and complexity is absent. This is the motivation of this research.

1.2 Overview and Contribution of the Thesis

This thesis evaluates the performance of multirate filter bank systems and wavelet systems for image compression. The filter bank systems provide a very useful framework for building fast image compression systems, while wavelet systems are very powerful means for image coding and network transmission. This thesis draws on ideas from all of the previously described developments, but the two of most direct importance are the M -channel filter bank based method for the design of coding system proposed by de Queiroz and Tran [9], de Queiroz, Nguyen and Rao [10] and Tran et. al. [36], and the combination techniques of biorthogonal wavelet systems [7] and the embedded coding scheme proposed by Shapiro [31] and Said and Pearman [30]. Many of the results presented are closely related to the ideas in these relevant papers. Some of the topics addressed in detail by this thesis include:

- methods for constructing M -channel filter bank systems;
- practical issues associated with the implementation of M -channel filter bank transforms such as computational efficiency and complexity;
- methods and implementations of wavelet transforms on image compression;
- effectiveness of DWT-related coding schemes on image compression; and
- modifications and enhancements to the DCT algorithm to handle the suppression of block effects, M -channel transforms, boundary extension, etc.

The thesis is divided into six chapters. The two subsequent chapters (Chapter 2 and 3) are intended to give the reader most of the background which is needed for the remainder of the thesis. In particular, Chapter 2 introduces the background of image compression and highlights DCT-based image coding schemes. JPEG-2000 standards are outlined at the end of this chapter. In Chapter 3, the theories of multirate filter banks and wavelet systems are introduced. The chapter begins by presenting the fundamentals of multirate systems. Then, quadrature mirror image filter (QMF) banks are examined in some detail. Finally, the link between QMF banks and wavelet systems is established. The remaining chapters present a mix of research results and other fundamental concepts required to understand these results.

In Chapter 4, multirate systems are discussed in depth. After introduction of block transformation, the discrete-time maximally decimated M -channel uniform filter bank is addressed at length. The practical issues associated with the implementation of M -channel filter bank transforms are discussed. Finally, the performance comparisons of M -channel filter bank transforms in image coding are carried out.

In Chapter 5, both theoretical and practical aspects of wavelet-based image compression are discussed. The chapter begins by introducing the basic ideas behind wavelet-based image compression systems. Measures for compression performance are discussed. Then embedded coding is defined and its benefits explained. Next, the EZW and SPIHT coding schemes are introduced. The chapter also evaluates the performance of those transforms and identifies the classes of images for which the various transforms are most effective. Based on these observations, an approach for a particular application is proposed in which the coding scheme is selected on a per image basis using image

specific characteristics. Modifications and enhancements to the EZW-based Wavekit coding package are also discussed.

Finally, Chapter 6 summarizes some of the more important results presented in this thesis along with the contributions it makes. The chapter concludes by suggesting directions for future research.

1.3 Notation

Some of the notation and conventions used in this thesis are as follows:

- The symbols $\mathbf{Z}$, $\mathbf{R}$, $\mathbf{C}$ denote the sets of integers, real numbers, and complex numbers, respectively.
- The symbol j denotes the quantity $\sqrt{-1}$.
- Function of a continuous variable is indicated with a round brackets, for example $f(t)$.
- Function of a discrete variable is indicated with a square brackets, for example $x[n]$.
- Boldface type is used to denote matrix and vector quantities.
- The symbol $\mathbf{I}_k$ denotes the $k \times k$ identity matrix.
- The symbol $\mathbf{J}_k$ denotes the $k \times k$ counter-identity or exchange matrix.
- The quantity $\mathbf{A}^T$ denotes the transpose of $\mathbf{A}$.
- The quantity $\mathbf{A}^*$ denotes the conjugate of $\mathbf{A}$.
- The quantity $\mathbf{A}^H$ denotes the conjugate transpose of $\mathbf{A}$.
- The z -transform of a sequence $x[n]$ is denoted as $X(z)$ and is defined as

$$Z\{x[n]\} = X(z) = \sum_n x[n]z^{-n}$$

- The inner product of two real-valued continuous functions is defined in the continuous variable case as

$$\langle f(t)g(t) \rangle = \int_{-\infty}^{\infty} f(t)g(t)dt$$

and in the discrete variable case as

$$\langle f[n]g[n] \rangle = \sum_n f[n]g[n]$$

- The notation $f \cdot g$ stands for the convolution of f and g .
- Filter coefficients are assumed to be real valued unless explicitly stated.
- The sampling period for all discrete sequences is assumed to be one. In other words, the sampling period is always normalized to be one. Thus, if we are given a sequence $x[n]$ with z -transform $X(z)$, the frequency spectrum associated with the signal is always obtained by evaluating $X(z)$ at $z = e^{j\omega}$.
- A filter H has the transfer function $H(z)$. That is, roman text is used to name a particular filter while italics are used to indicate the transfer function associated with the filter.
- The notation $\lfloor x \rfloor$ denotes the greatest integer not more than x (or equivalently, x rounded to the nearest integer in the direction of $-\infty$).

Chapter 2

Principles behind Compression

2.1 Introduction

Uncompressed multimedia (graphics, audio and video) data requires considerable storage capacity and transmission bandwidth. Despite rapid progress in mass-storage density, processor speeds, and digital communication system performance, demand for data storage capacity and data-transmission bandwidth continues to outstrip the capabilities of available technologies. The recent growth of data intensive multimedia-based web applications have not only sustained the need for more efficient ways to encode signals and images but have made compression of such signals central to storage and communication technology.

For still image compression, the 'Joint Photographic Experts Group' or JPEG [26] standard has been established by ISO (International Standards Organization) and IEC (International Electro-Technical Commission). The performance of these coders generally degrades at low bit-rates mainly because of the underlying block-based Discrete Cosine Transform (DCT)[29] scheme. More recently, the wavelet transforms and block-based lapped transforms have emerged as the cutting-edge technologies within the field of image compression. Wavelet-based coding schemes [40] allow progressive transmission with good picture quality at higher compression ratios, while block-based

lapped transforms [20] provide substantial improvements in picture quality and easy VLSI implementation.

2.2. 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 a 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 [47]:

- **Spatial Redundancy** or correlation between neighboring pixel values.
- **Spectral Redundancy** or correlation between different color planes or spectral bands.
- **Temporal Redundancy** or correlation between adjacent frames in a sequence of images (in video applications only, exclusive of this study).

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. Since we will focus only on still image compression, we will not deal with temporal redundancy in this thesis.

2.3. Classification of Compression Techniques

There are two ways of classifying compression techniques,

- (a) **Lossless vs. Lossy compression:** In lossless compression schemes, the reconstructed image, after compression, is numerically identical to the original image. However lossless compression can only achieve a modest amount of compression. An image reconstructed following lossy compression contains degradation relative to the original. Often this is because the compression scheme completely discards redundant information. However, lossy schemes are capable of achieving much higher compression. At sufficiently high bit-rate, no visible loss is perceived (visually lossless). The primary focus of this study is research on lossy compression.
- (b) **Predictive vs. 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. Differential Pulse Code Modulation (DPCM) is one particular example of predictive coding. 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 transform and then codes the transformed values (coefficients). This method provides greater data compression compared to predictive methods, although at the expense of greater computation.

2.4. Image Coding Strategies

A typical lossy image compression system is shown in Figure 2.1. It consists of three closely connected components namely (a) *Source Encoder* (b) *Quantizer*, and (c) *Entropy*

Encoder. Compression is accomplished by applying a linear transform to decorrelate the image data, quantizing the resulting transform coefficients, and entropy coding the quantized values.

Figure 2.1. A typical lossy image encoder

In this thesis we discuss the issues on compressing gray scale images only. The algorithms explained can be easily extended to colour images, either by processing each of the colour planes separately, or by transforming the image from RGB representation to other convenient representations like YUV in which the processing is much easier [47].

2.4.1. Source Encoder/Decoder

An image is represented as a two-dimensional array of pixel values, each value representing the brightness level at that point. It is possible to differentiate among coefficients as more important ones and lesser important ones. Most natural images have smooth colour variations, with the fine details being represented as sharp edges in between the smooth variations. Technically, the smooth variations in colour 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 which give the detail) is added upon to refine the image, thereby giving a detailed image. Hence, the smooth variations or low frequency components are demanding more importance than the details.

Separating the smooth variations and details of the image can be done in many ways. A variety of linear transforms have been developed which include Discrete Fourier Transform (DFT) [16], Discrete Cosine Transform (DCT) [29], Discrete Wavelet Transform (DWT) [13] and many more, each with its own advantages and disadvantages.

2.4.2. Quantization

Quantization refers to the process of approximating the continuous set of values in the image data with a finite (preferably small) set of values. The input to a quantizer is the original data, and the output is always one among a finite number of levels. Since this is a many-to-one mapping, it is a lossy process and is the main source of compression in an encoder. A good quantizer is one which represents the original signal with minimum loss or distortion.

There are two types of quantization - *Scalar Quantization* (SQ) and *Vector Quantization* (VQ). In scalar quantization, each input symbol is treated separately in producing the output, while in vector quantization the input symbols are clustered together in groups called vectors, and processed to give the output [16]. Both uniform and non-uniform quantizers can be used depending on the problem at hand.

A scalar quantizer is the simplest quantizer. It maps a continuous variable r to a finite set. If the input range is divided into levels of equal spacing, then the quantizer is termed as a Uniform Quantizer, and if not, it is termed as a Non-Uniform Quantizer. A uniform quantizer can be easily specified by its lower bound, upper bound and the step size. Also, implementing a uniform quantizer is easier than a non-uniform quantizer. Take a look at

the uniform quantizer shown in Figure 2.2. If the input falls between $n \cdot r$ and $(n+1) \cdot r$, the quantizer outputs the symbol n .

Figure 2.2. A uniform scalar quantizer

In contrast to scalar quantization, vector quantization is the process of mapping a continuous N -dimensional vector $\mathbf{x}$ to a discrete-valued N -dimensional vector. This clustering of data and treating them as a single unit increases the optimality of the vector quantizer, but at the cost of increased computational complexity.

Just the same way a quantizer partitions its input and outputs discrete levels, a *Dequantizer* is one which receives the output levels of a quantizer and converts them into near-original data, by translating each level into a 'reproduction point' in the actual range of data. It can be seen from the literature [16], that the optimum quantizer (encoder) and optimum dequantizer (decoder) must satisfy the following conditions.

- Given the output levels or partitions of the encoder, the best decoder is one that puts the reproduction points $\mathbf{x}'$ on the centers of mass of the partitions. This is known as the *centroid condition*.
- Given the reproduction points of the decoder, the best encoder is one that puts the partition boundaries exactly in the middle of the reproduction points, i.e. each $\mathbf{x}$ is translated to its nearest reproduction point. This is known as the *nearest neighbour condition*.

The quantization error ($x-x'$) is used as a measure of the optimality of the quantizer and dequantizer.

2.4.3. Entropy Encoder

After the data has been quantized into a finite set of values, it can be encoded using an Entropy Coder to give additional compression. The entropy encoder uses a model to accurately determine the probabilities for each quantized value and produces an appropriate code based on these probabilities so that the resultant output code stream will be smaller than the input stream. Two of the most popular entropy coding schemes are Huffman coding and Arithmetic coding [23][42].

Huffman coding is a classical data compression technique. It utilizes the statistical property of alphabets in the source stream, and then produces respective codes for these symbols. These codes are of variable code length using an integral number of bits. The codes for symbols having higher probability of occurrence are shorter than those codes for symbols having lower probability. This simple idea causes a reduction in the average code length, and thus the overall size of compressed data is smaller than the original.

Arithmetic coding is also a kind of statistical coding algorithm similar to Huffman coding. However, it uses a different approach to utilize symbol probabilities, and it in general performs better than Huffman coding. In Huffman coding, optimal codeword length is obtained when the symbol probabilities are of the form $(1/2)^x$, where x is an integer. This is because Huffman coding assigns codes with an integral number of bits. Obviously, this form of symbol probabilities is rare in practice. Arithmetic coding is a statistical coding method solving this problem. The main procedure of arithmetic coding

is to try to find an interval number between 0 and 1 to represent an alphabet in the source. According to the actual order of these alphabets in the stream, these intervals are combined until the last alphabet is accounted for, then a final interval of numbers is obtained. Although arithmetic coding is more powerful than Huffman coding in terms of compression ratio, arithmetic coding is more complex and requires more computational power and memory. Huffman coding is more attractive compared with arithmetic coding when simplicity is the major concern.

2.4.4. Bit Allocation

The first step in compressing an image is to segregate the image data into different classes. Depending on the importance of the data it contains, each class is allocated a portion of the total bit budget, such that the compressed image has the minimum possible distortion. This procedure is known as bit allocation [25].

The Rate-Distortion theory is often used for solving the problem of allocating bits to a set of classes, or for bit-rate control in general. The theory aims at reducing the distortion for a given target bit-rate, by optimally allocating bits to the various classes of data. One approach to solve the problem of Optimal Bit Allocation using the Rate-Distortion theory is given in [28], which is explained below (see also Figure 2.3).

1. Initially, all classes are allocated a predefined maximum number of bits.
2. For each class, one bit is reduced from its quota of allocated bits, and the distortion due to the reduction of that 1 bit is calculated.
3. Of all the classes, the class with minimum distortion for a reduction of 1 bit is noted, and 1 bit is reduced from its quota of bits.
4. The total distortion D for all classes is calculated.

5. The total rate for all the classes is calculated as $R = \sum p(i) * B(i)$, where $p(i)$ is the probability of the i -th class and $B(i)$ is the bit allocation for each class.
6. Compare the target bit rate and distortion specifications with the values obtained above. If not optimal, go to step 2.

In the approach explained above, we keep on reducing one bit at a time until we achieve optimality either in distortion or target rate, or both. An alternate approach which is also mentioned in [28] is to initially start with zero bits allocated for all classes, and to find the class which is most 'benefited' by getting an additional bit. The 'benefit' of a class is defined as the decrease in distortion for that class. The benefit of a bit is a decreasing function of the number of bits allocated previously to the same class. Both approaches can be used to the bit allocation problem [19].

Figure 2.3. The distortion function vs. the effect of adding a bit.

2.5. Comparison Methodology

One of the major concerns in coding techniques has been compression efficiency, but this is not the only factor that determines the choice of a particular algorithm for an application. In this thesis, we concentrate in both compression efficiency and complexity.

The error metric used to evaluate the compression efficiency of the various image compression techniques is the *Mean Square Error (MSE)* and its corresponding *Peak Signal to Noise Ratio (PSNR)* [32]. The MSE is the cumulative squared error between the compressed and the original image, whereas PSNR is a measure of the distortion. The mathematical formulae for the two errors are

$$MSE = \frac{1}{MN} \sum_{n=1}^N \sum_{m=1}^M [I(m, n) - I'(m, n)]^2 \quad (2.1)$$

$$PSNR = -10 * \log_{10} \frac{MSE}{(2^b - 1)^2} \quad (2.2)$$

where $I(m, n)$ is the original image with dimensions $M \times N$ and having b bits/pixel, and $I'(m, n)$ is the reconstructed image. Logically, a higher value of $PSNR$ is good because it means that the ratio of Signal to Noise is higher. Here, the 'signal' is the original image, and the 'noise' is the error in reconstruction.

Evaluating complexity is a difficult issue, with no well-defined measure. In this thesis, we use the execution times of the different algorithms on a PC-computer as a rough indication of complexity. This only gives an appreciation of the involved complexity because of many influence factors: total working memory, bandwidth, number of CPU cycles, number of hardware gates, etc.

2.6. DCT-based JPEG Image Coding

The discrete cosine transform (DCT) can be regarded as a discrete-time version of the Fourier-cosine series. It is a close relative of the discrete Fourier transform (DFT), a technique for converting a signal into elementary frequency components. Thus DCT can be computed with a Fast Fourier Transform (FFT) like algorithm in $O(n \log n)$ operations

[37]. Unlike the DFT, the DCT is a real-valued transform and provides a better approximation of a signal with fewer coefficients. The DCT coefficient $X[u]$ of a discrete signal $x[n]$, $n=0, 1, \dots, N-1$ is defined as:

$$X[u] = \sqrt{\frac{2}{N}} C(u) \sum_{n=0}^{N-1} x[n] \cos\left\{\frac{(2n+1)u\pi}{2N}\right\} \quad (2.3)$$

$$\text{where, } C(u) = 1/\sqrt{2} \quad \text{for } u = 0 \text{ and} \\ = 1 \quad \text{otherwise.}$$

In 1992, JPEG established the first international standard for still image compression where the encoders and decoders are DCT-based [26]. The JPEG standard specifies three modes namely sequential, progressive, and hierarchical for lossy encoding, and one mode of lossless encoding. The 'baseline JPEG coder' [26][41] which is the sequential encoding in its simplest form, will be briefly discussed here. Figures 2.4. and 2.5 show the key processing steps in such an encoder and decoder for grayscale images. Color image compression can be approximately regarded as compression of multiple grayscale images, which are either compressed entirely one at a time, or are compressed by alternately interleaving 8x8 sample blocks from each in turn. In this thesis, we focus on grayscale images only.

Figure 2.4 JPEG Encoder Block Diagram

Figure 2.5 JPEG Decoder Block Diagram

The DCT-based encoder can be thought of as essentially compression of a stream of 8×8 blocks of image samples. Each 8×8 block makes its way through each processing step, and yields sequential 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 8×8 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; it merely transforms them to a domain in which they can be more efficiently encoded.

From the output of the FDCT, each of the 64 DCT coefficients is uniformly quantized in conjunction with a carefully designed 64-element Quantization Table (QT) [26] as shown in Figure 2.6. After quantization, all of the quantized coefficients are ordered into the "zig-zag" sequence as shown in Figure 2.7. This ordering helps to facilitate entropy encoding 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.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	35	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Fig. 2.6 Quantization Table

Fig. 2.7 Zig-Zag sequence.

Run-length coding has been conducted on zig-zag data sequence, which reduces the quantity of data remarkably. Entropy Coding (EC) achieves additional compression, this time 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 [23].

The decoding is a reverse procedure of the encoding (Figure 2.5). At first, the compressed data is reassembled into a two-dimensional array by the entropy decoder. The values in the array are then multiplied by the corresponding QT elements to recover the original coefficient values. The image is reconstructed by the inverse DCT (IDCT) processing of the coefficient values.

2.7. JPEG-2000: New Image Compression Standard

Extensive research has been conducted on still image compression since the establishment of the JPEG standard in 1992. To bring these research efforts into a focus, a new standard called JPEG-2000 [15] for coding of still images has been established. The JPEG-2000 standard is based on the Discrete Wavelet transform (DWT) algorithms instead of DCT algorithms. This standard is intended to advance standardized image coding systems to serve applications into the 21st century. It provides a set of features vital to many high-end and emerging image applications by taking advantage of new modern technologies. Specifically, this new standard addresses area where current standards fail to produce the best quality or performance. It also provides capabilities to markets that currently do not use compression. It is intended to complement, not replace, the baseline JPEG standards.

Comparing with previous standards, JPEG-2000 standard includes many new features such as improved low bit-rate compression performance, lossless and lossy compression, continuous-tone and bi-level compression, compression of large images, single decompression architecture, transmission in noisy environments including robustness to bit-errors, progressive transmission by pixel accuracy and resolution, content-based description, protective image security, and so on [21]. This allows an application to manipulate or transmit only the essential information for any target device from any JPEG 2000 compressed source image.

Chapter 3

Multirate and Wavelet Signal Processing

3.1 Introduction

The purpose of this chapter is to provide most of the preliminaries needed for the study of multirate filter banks system. Understanding the fundamentals of multirate systems is the first step to study quadrature mirror-image filter (QMF) banks, which is especially significant in image compression. This chapter begins by introducing some basic multirate system concepts, and then uses these concepts to establish a general framework for the study of QMF banks. Lastly, the link between QMF banks and wavelet systems is established.

The material presented in this chapter is standard in the multirate digital signal processing literature and is taken from [20], [38], [12], [40], [33], [24] and [34].

3.2 Multirate System and Basic Building Blocks

Depending on the number of different sampling rates it employs, a discrete-time system can be classified as being either unirate or multirate. A system that processes signals at a single sampling rate is referred to as unirate. In contrast, a system that process signals at more than one sampling rate is referred to as multirate.

Multirate systems are extremely useful for many signal processing applications. Often, a multirate system can be used to perform a task more easily or efficiently than is possible with a unirate system. Other times, a signal processing task inherently requires using multiple sampling rates, like speech or image coding. In such cases, a multirate system must be used.

The basic building blocks of unirate systems are familiar to us all, namely, the unit delay/advance, adder, and multiplier. Multirate systems have these same building blocks plus two new blocks: the downsampler and upsampler. In this section the basic multirate building blocks are introduced, along with their frequency domain characterization and interconnection behavior.

3.2.1 Downsampling and Upsampling

(a). Downsampler

One of the basic building blocks of multirate systems is the downsampler. The downsampler is also called a decimator, subsampler, sampling rate compressor, or merely a compressor. Figure 3.1 shows the block diagram of the M -fold downsampler which is characterized by the input-output relation in Equation (3.1).

Figure 3.1 M -fold downsampler

$$y[n] = x[Mn] \quad (3.1)$$

That is, the output at time n is equal to the input at time Mn . The constant M is referred to as the downsampling factor. In simple terms, the downsampling operation keeps every

M^{th} sample and discards the others. As a consequence, only the input sample numbers equal to a multiple of M are retained. This operation is demonstrated in Figure 3.2 for the case of $M=2$.

Figure 3.2 Demonstration of downsampling for $M=2$.

From Equation (3.1), it follows that downsampling is a linear time-varying operation. The relationship between the input and output of the downsampler in the z -domain is given in [38] as following

$$Y(z) = \frac{1}{M} \sum_{k=0}^{M-1} X(z^{1/M} e^{-j2\pi k/M}) = \frac{1}{M} \sum_{k=0}^{M-1} X(z^{1/M} W_M^k) \quad (3.2)$$

Assuming that the sampling period before and after downsampling is normalized to one, this leads directly to the frequency domain relation

$$Y(e^{j\omega}) = \frac{1}{M} \sum_{k=0}^{M-1} X(e^{j(\omega - 2\pi k)/M}) \quad (3.3)$$

Downsampling has a very simple frequency domain interpretation. The term with $k=0$ in Equation (3.3), i.e. $X(e^{j\omega/M})$, is indeed the M -fold stretched version of $X(e^{j\omega})$. The other $M-1$ terms with $k > 0$ are uniformly shifted versions of this stretched version. These M terms together make up a function with period 2π in ω , which is the basic property of the

Discrete Fourier Transform (DFT) of any sequence [38]. Figure 3.3 demonstrates the effect of downsampling for $M=2$, and we see that the stretched version $X(e^{j\omega/2})$ can in general overlap with its shifted replicas. If this happens, $x[n]$ cannot be recovered after downsampling. This overlap effect is called *aliasing*. If aliasing occurs, it is not possible to recover the original signal from its downsampled version. Aliasing can be avoided if $x[n]$ is a lowpass signal bandlimited to $|\omega| < \pi/M$. This is equivalent to saying that Nyquist criterion must not be violated if aliasing is to be avoided.

Figure 3.3 Transform domain effects of the upsampler and downsampler (aliasing case). (a).the Fourier transforms of the input signal $x[n]$, (b) the two stretched and shifted versions of (a), and (c) the downsampled signal.

(b). Upsampler

Figure 3.4 shows a building block which is commonly called an upsampler. Other names for the upsampler are: expander, interpolator, and sampling rate expander. It is characterized by the following input-output relation:

Figure 3.4 M -fold upsampler

$$y[n] = \begin{cases} x[n/M], & \text{if } n \text{ is an integer multiple of } M \\ 0, & \text{otherwise} \end{cases} \quad (3.4)$$

That is, the output $y[n]$ is obtained by inserting $M-1$ zeros between the adjacent samples of input signal $x[n]$, as demonstrated in Figure 3.5 for $M = 2$. It is evident that the upsampling does not cause loss of information. We can recover the input $x[n]$ from $y[n]$ by M -fold downsampling.

Figure 3.5 Demonstration of upsampling for $M = 2$.

From Equation (3.4), it follows that upsampling is a linear time-varying operation. The relationship between the input and output of the upsampler in the z -domain is given by

$$Y(z) = X(z^M) \quad (3.5)$$

Assuming that the sampling period before and after upsampling is normalized to one, this directly yields the frequency domain relation

$$Y(e^{j\omega}) = X(e^{j\omega M}) \quad (3.6)$$

Upsampling has a very straightforward interpretation in the frequency domain. The upsampling process simply serves to move the location of the sampling frequency on the frequency axis. Due to the sampling period renormalization, the spectrum appears compressed. It is worth noting, however, that this compression effect is only a consequence of the sampling period renormalization and is not caused by upsampling itself.

Although multiple copies of the original baseband spectrum appear by upsampling, each copy has the same shape as the original input spectrum. As a result, the original signal can always be recovered from the upsampled signal with the appropriate use of filtering. There is no information loss.

(c) Decimation Filters and Interpolation Filters

In most applications, the downsampler is preceded by a frequency selective decimation filter (e.g., low-pass), as in Fig. 3.6(a), to prevent or reduce the aliasing caused by downsampling. The filter ensures that the signal being downsampled is approximately bandlimited. The exact band-edges of the filter depend on how much aliasing is permitted. Normally, a certain degree of aliasing is permitted because this can eventually be canceled off in a filter bank [38].

Figure 3.6 (a) Decimation filter, and (b) Interpolation filter.

Next, an *interpolation filter* is a digital filter that follows an upsampler. The typical purpose is to suppress all the images, in which case the interpolation filter would be a lowpass filter with cut-off frequency π/M .

3.2.2 Noble Identities

The *noble identities* are very useful in the theory and implementation of multirate systems. Often a downsampler or upsampler appears in cascade with a filter. Although it is not always possible to interchange the order of upsampling/downsampling and filtering without changing the system behavior, it is sometimes possible to find an equivalent system with the order of these operations reversed, through the use of two very important relationships called the *noble identities*. The first identity allows one to replace a filtering operation on one side of a downsampler with an equivalent filtering operation on the other side of the downsampler. This identity is illustrated in Figure 3.7(a) where the transfer function $H(z)$ is a rational polynomial expression.

Figure 3.7 Multirate identities (a) Interchange of downsampling and filtering, and (b) Interchange of filtering and upsampling

Using Equation (3.2), it follows that downsampling the filtered signal with the z -transform $H(z^M)X(z)$ results in [34],

$$Y(z) = \frac{1}{M} \sum_{k=0}^{M-1} X(W_M^k z^{1/M}) H((W_M^k z^{1/M})^M) = H(z) \frac{1}{M} \sum_{k=0}^{M-1} X(W_M^k z^{1/M}),$$

which is equal to filtering a downsampled version of $X(z)$.

The second identity allows us to replace a filtering operation on one side of an upsampler with an equivalent filtering operation on the other side of the upsampler. This identity is shown in Figure 3.7(b). It is important to emphasize that these identities hold only if the transfer function $H(z)$ is a rational polynomial expression. Using Equation (3.5), it is immediate that both systems lead to an output with the z -transform $H(z^M)X(z^M)$ when the input is $X(z)$.

In short, these two properties (identities) simply say that filtering in the downsampled (lower sampling rate) domain can always be realized by filtering in the upsampled (higher sampling rate) domain, but then with the upsampled filter. Using the noble identities, we can move filtering operations across upsamplers (or downsamplers) and achieve improved computational efficiency.

3.3 The Polyphase Representation

An important advancement in multirate signal processing is the invention of the polyphase representation [38]. The utility of polyphase methods is twofold. First, these methods provide a mathematically convenient representation for filtering operations in a multirate framework. This representation, known as the polyphase representation, is very useful as it greatly simplifies many theoretical results allowing easier analysis of multirate systems. Second, polyphase methods offer a desirable means for implementing

filtering operations in a multirate framework. This implementation strategy, known as the polyphase realization, leads to computationally efficient structures for many multirate systems.

To explain the basic idea, consider a filter $H(z)$ with impulse response $h(n)$. By separating the even numbered coefficients of $h(n)$ from the odd numbered ones, we can write

$$H(z) = \sum_{n=-\infty}^{\infty} h[2n]z^{-2n} + z^{-1} \sum_{n=-\infty}^{\infty} h[2n+1]z^{-2n}$$

Defining

$$E_0(z) = \sum_{n=-\infty}^{\infty} h[2n]z^{-n}, \quad E_1(z) = \sum_{n=-\infty}^{\infty} h[2n+1]z^{-n}$$

we can, therefore, write $H(z)$ as

$$H(z) = E_0(z^2) + z^{-1}E_1(z^2) \quad (3.7)$$

Note that this representation holds whether $H(z)$ is FIR or IIR; causal or noncausal. The functions $E_0(z)$, $E_1(z)$ are called the *type 1 polyphase components* of $H(z)$.

Example 3.1 : if $H(z)$ is an FIR, let $H(z) = 1 + 2z^{-1} + 3z^{-2} + 4z^{-3}$ then

$$E_0(z) = 1 + 3z^{-1}, \quad E_1(z) = 2 + 4z^{-1}$$

For an IIR example, let $H(z) = 1/(1-\alpha z^{-1})$. By using the identity $(1-x) = (1-x^2)/(1+x)$ we can write

$$H(z) = \frac{1}{1-\alpha z^{-1}} = \frac{1}{1-\alpha^2 z^{-2}} + \frac{\alpha z^{-1}}{1-\alpha^2 z^{-2}},$$

So that $E_0(z) = 1/(1-\alpha^2 z^{-1})$ and $E_1(z) = \alpha/(1-\alpha^2 z^{-1})$.

Extending this idea further, suppose we are given any positive integer M . We can always decompose $H(z)$ as

$$\begin{aligned} H(z) &= \sum_{n=-\infty}^{\infty} h[Mn]z^{-Mn} \\ &+ z^{-1} \sum_{n=-\infty}^{\infty} h[Mn+1]z^{-Mn} \\ &\vdots \\ &+ z^{-(M-1)} \sum_{n=-\infty}^{\infty} h[Mn+M-1]z^{-Mn} \end{aligned}$$

This can be compactly written as

$$H(z) = \sum_{l=0}^{M-1} z^{-l} E_l(z^M) \quad (3.8)$$

Where

$$E_l(z) = \sum_{n=-\infty}^{\infty} e_l[n]z^{-n} \quad (3.9)$$

with

$$e_l[n] = h[Mn+l], \quad 0 \leq l \leq M-1 \quad (3.10)$$

Equation (3.8) is called the *type 1 polyphase representation* (with respect to M) and $E_l(z)$ the polyphase components of $H(z)$. Figure 3.8 summarizes the generation of $e_l[n]$ from $h[n]$:

Fig. 3.8 Schematic of the relation between $h[n]$ and its l -th polyphase component.

A variation of Equation (3.8) is

$$H(z) = \sum_{l=0}^{M-1} z^{-(M-1-l)} R_l(z^M) \quad (3.11)$$

where $R_l(z)$, called *type 2 polyphase components* of $H(z)$, are permutations of $E_l(z)$, that is, $R_l(z) = E_{M-1-l}(z)$. Type 1 and 2 polyphases are very useful for efficient implementation of decimation (interpolation) filters.

When realized in its type 1 or 2 M -fold polyphase form, a filter is implemented using a delay/advance chain, adders, and M filters each having a transfer function that is a rational polynomial expression in z^M . The structures corresponding to the two types of polyphase representations are presented in Figure 3.9.

Figure 3.9 Polyphase realization of a filter $H(z)$ (a) Type 1 (b) Type 2

In multirate systems, filters are often connected in cascade with upsamplers/downsamplers. For reasons of computational efficiency, it is usually preferable to perform any filtering on the side of the upsampler/downsampler with the lower sampling rate. The polyphase representation provides us with a means to express any filter as a set of filters with transfer functions in z^M so that filtering can be moved to the more desirable side of an upsampler/downsampler.

Suppose a filter realized in its type 1 polyphase form is immediately followed by an M -fold downsampler. In such a case, the first noble identity can be used to move the downsampling operation before the polyphase filtering operations. Likewise, suppose a filter realized in its type 2 polyphase form is immediately preceded by an M -fold upsampler. In such a case, the second noble identity can be used to move the upsampling operation after the polyphase filtering operations. In this way, we can use the polyphase implementation of filters to obtain efficient multirate structures.

3.4 Multirate Filter Banks and Subband Coding

A common application of multirate techniques is encountered in the processing of signals using filter banks. In this section we briefly review multirate filter banks ([34], [38] and [12]) along with one of their popular applications, i.e., subband coding.

3.4.1 Filter Banks

A filter bank is a set of filters operating in parallel, either in an analysis mode with a single input and multiple outputs or in a synthesis mode with multiple inputs and a single output. An analysis filter bank is shown in Figure 3.10 side (a) (filters $H_0(z)$, $H_1(z)$, ..., $H_{N-1}(z)$) and is used to decompose a given signal into N components, typically referred to as subband signals. In most cases, these subband signals are bandlimited, therefore downsampled before subsequent use (transmission over a communication channel or storage). The downsampling factor in general may differ from channel to channel based on the characteristics of the analysis filter employed in that channel. A synthesis filter bank, shown on the (b) side of Figure 3.10 (filters $F_0(z)$, $F_1(z)$, ..., $F_{N-1}(z)$), enables the

merging of several input signals into a higher rate signal. In principle, this operation consists of upsampling N signals to a common higher sampling rate and applying the upsampling signals to suitable filters followed by an addition of the filtered signals.

Figure 3.10 Analysis and synthesis filter banks

Consider connecting an analysis bank and a synthesis bank as shown in Figure 3.10. The resulting system is commonly used in compressing and coding speech, image and video signals. There are several features to be considered in the design of this system depending on the application. These include the choice of filters in the analysis and synthesis banks, the widths and spacings of the frequency bands, the extent of allowed overlap of frequency bands, down and upsampling factors, and the nature of reconstruction required. One key consideration is the choice of the pass-band widths and stopband attenuations of the analysis filters, which is dictated by the application. Three types of typical magnitude frequency responses that are used in practice are shown in Figure 3.11(a)-(c). These could be classified based on the width and spacing between different bands. If frequency bands have equal width and equal spacing as in Figures 3.11(a) and 3.11(b), then the filtered signals can all be downsampled by the same factor. This type of filter bank is

referred to as a uniform filter bank. In some applications, the partition of the frequency band is non-uniform, leading to a non-uniform filter bank, as illustrated in Figure 3.11(c).

Figure 3.11 Three types of frequency band partitions in filter banks (a). non-overlapping uniform; (b). overlapped uniform; and (c). overlapping non-uniform.

Another feature to be considered is the choice of the sampling rate for the outputs of the different filters in the analysis bank. As shown in Figure 3.10, the outputs of a bank of N filters are downsampled by factors $M_0, M_1, \dots, M_{N-1}$. If the total sampling rate of the outputs of the analysis filters is equal to the input sampling rate, that is $\sum_{k=0}^{N-1} (1/M_k) = 1$, then the filter bank is referred to as *critically sampled* or *maximally decimated*. Maximal decimation is often encountered in subband systems where an analysis-synthesis filter bank pair is designed to provide an output signal that very closely approximates or exactly reconstructs an arbitrary input signal.

The filter banks that are used in a large number of applications, like that in MUSICAM and JPEG [12], are uniform and maximally decimated, partly because they can be efficiently implemented using polyphase decomposition. Note that in uniform

filter banks the downsampling factors are equal ($M_k = M$, $0 \leq k \leq N-1$). Therefore in maximally decimated uniform filter banks, the number of channels equals the decimation factor, that is, $N = M$ since $\sum(1/M)=1$. In this thesis, we will focus on M -channel maximally decimated uniform filter banks and octave-band filter banks. Such systems are linear periodically time varying (LPTV) with period M . Conversely, as shown in [38] any M -periodic LPTV system can be realized by a multirate filter bank.

3.4.2 M -Channel QMF Banks

Although many filter bank configurations exist, an extremely useful one is the so-called M -channel quadrature mirror-image filter (QMF) bank. Such a system is more correctly referred to as a uniformly maximally decimated filter bank, but the term QMF bank is often used for historical reasons. The general structure of such a system is shown in Figure 3.12. The input signal $x[n]$ is split into M channels, processed by the analysis filters H_k , downsampled by M , upsampled by M , transformed by the synthesis filters F_k , and then summed to produce the output signal $y[n]$.

Figure 3.12 M -channel QMF bank

3.4.3 Perfect Reconstruction of QMF Banks

For the *QMF* bank of Figure 3.12 to be of practical use, the output $y[n]$ is usually required to be an accurate reproduction of the input $x[n]$. If the system is such that $y[n] = cx[n - n_0]$ for all $x[n]$ and for some real c and integer n_0 , the system is said to have the perfect reconstruction (PR) property. In other words, a PR system can reproduce the input signal exactly except for a time shift. Generally, there are three reasons that the reconstructed signal $y[n]$ can differ from $x[n]$: *aliasing distortion*, *amplitude distortion*, and *phase distortion*. The analysis and synthesis filters can be designed in such a way so as to eliminate some or all of these distortions depending on what is required by the application.

Example 3.2: Two-channel QMF filter bank

Using the results developed in Sec. 3.4, we can find an expression for output $Y(z)$. For a special case two channel QMF filter bank ($M = 2$) in Figure 3.12, we have

$$X_k(z) = H_k(z)X(z), \quad k = 0, 1.$$

The z -transforms of the decimated signals $v_k[n]$ are [from Equation (3.2) with $M = 2$]

$$V_k(z) = \frac{1}{2}[X_k(z^{1/2}) + X_k(-z^{1/2})], \quad k = 0, 1.$$

The second term above represents aliasing. The z -transform of $Y_k(z)$ is $V_k(z^2)$ so that

$$\begin{aligned} Y_k(z) = V_k(z^2) &= \frac{1}{2}[X_k(z) + X_k(-z)] \\ &= \frac{1}{2}[H_k(z)X(z) + H_k(-z)X(z)], \quad k = 0, 1. \end{aligned}$$

The reconstructed signal is

$$\begin{aligned} Y(z) &= F_0(z)Y_0(z) + F_1(z)Y_1(z) \\ &= \frac{1}{2}[H_0(z)F_0(z) + H_1(z)F_1(z)]X(z) + \frac{1}{2}[H_0(-z)F_0(z) + H_1(-z)F_1(z)]X(-z) \end{aligned}$$

Or, it can be expressed in matrix-vector notation,

$$Y(z) = \frac{1}{2} \begin{bmatrix} X(z) & X(-z) \end{bmatrix} \begin{bmatrix} H_0(z) & H_1(z) \\ H_0(-z) & H_1(-z) \end{bmatrix} \begin{bmatrix} F_0(z) \\ F_1(z) \end{bmatrix} = \frac{1}{2} \mathbf{X}^T(z) \mathbf{H}(z) \mathbf{F}(z)$$

The matrix $\mathbf{H}(z)$ is the alias component matrix. The term that contains $X(-z)$ originated from the decimation. So, we can cancel aliasing by choosing the filters such that the quantity $H_0(-z)F_0(z) + H_1(-z)F_1(z)$ is zero. The following assignments cancel aliasing:

$$F_0(z) = H_1(-z), \quad F_1(z) = -H_0(-z).$$

Given analysis filter bank $H_0(z)$ and $H_1(z)$, we can completely cancel aliasing and recover the original signal by choosing a synthesis filter bank using the above conditions.

3.5. Implementation of QMF Bank

3.5.1 Polyphase Form of a QMF Bank

The polyphase form of a QMF bank is based on the polyphase representation discussed earlier in the context of individual filters. Although many different variations on the polyphase form of a QMF bank exist, one particular variant is most commonly used. It involves representing the filters of the analysis bank in its type 1 polyphase form and the filters of the synthesis bank in its type 2 polyphase form.

Suppose we have an M -channel QMF bank with analysis filters H_k and synthesis filters F_k . For notational convenience, we define the equations

$$\mathbf{h}(z) = \begin{bmatrix} H_0(z) \\ H_1(z) \\ \dots \\ H_{M-1}(z) \end{bmatrix}, \quad \mathbf{f}(z) = \begin{bmatrix} F_0(z) \\ F_1(z) \\ \dots \\ F_{M-1}(z) \end{bmatrix}, \quad \mathbf{e}(z) = \begin{bmatrix} 1 \\ z^{-1} \\ \dots \\ z^{-(M-1)} \end{bmatrix}$$

First, let us consider the analysis filters of the QMF bank. The transfer functions $H_k(z)$ can be expressed in polyphase form as

$$H_k(z) = \sum_{l=0}^{M-1} z^{-l} E_{k,l}(z^M) \quad (\text{Type 1 polyphase}) \quad (3.12)$$

This equation can be rewritten in matrix form as

$$\begin{bmatrix} H_0(z) \\ H_1(z) \\ \vdots \\ H_{M-1}(z) \end{bmatrix} = \begin{bmatrix} E_{0,0}(z^M) & E_{0,1}(z^M) & \dots & E_{0,M-1}(z^M) \\ E_{1,0}(z^M) & E_{1,1}(z^M) & \dots & E_{1,M-1}(z^M) \\ \vdots & \vdots & \ddots & \vdots \\ E_{M-1,0}(z^M) & E_{M-1,1}(z^M) & \dots & E_{M-1,M-1}(z^M) \end{bmatrix} \begin{bmatrix} 1 \\ z^{-1} \\ \vdots \\ z^{-(M-1)} \end{bmatrix}$$

or more compactly as

$$\mathbf{h}(z) = \mathbf{E}(z^M) \mathbf{e}(z) \quad (3.13)$$

Equation (3.13) completely characterizes the analysis bank and is called the polyphase representation of the analysis bank. Matrix $\mathbf{E}(z)$ is the $M \times M$ Type 1 polyphase component matrix for the analysis bank.

Now, let us consider the synthesis filters of the QMF bank. The transfer functions $F_k(z)$ can be expressed in polyphase form as

$$F_k(z) = \sum_{l=0}^{M-1} z^{-(M-1-l)} R_{k,l}(z^M) \quad (\text{Type 2 polyphase}) \quad (3.14)$$

Using matrix notations, this becomes

$$\begin{bmatrix} F_0(z) & F_1(z) & \cdots & F_{M-1}(z) \end{bmatrix} = \begin{bmatrix} z^{-(M-1)} & z^{-(M-2)} & \cdots & 1 \end{bmatrix} \begin{bmatrix} R_{0,0}(z^M) & R_{0,1}(z^M) & \cdots & R_{0,M-1}(z^M) \\ R_{1,0}(z^M) & R_{1,1}(z^M) & \cdots & R_{1,M-1}(z^M) \\ \vdots & \vdots & \ddots & \vdots \\ R_{M-1,0}(z^M) & R_{M-1,1}(z^M) & \cdots & R_{M-1,M-1}(z^M) \end{bmatrix}$$

This can be written more concisely as

$$\mathbf{f}^T(z) = z^{-(M-1)} \tilde{\mathbf{e}}(z) \mathbf{R}(z^M) \quad (3.15)$$

Equation (3.15) completely characterizes the synthesis bank and is called the polyphase representation of the synthesis bank. Matrix $\mathbf{R}(z)$ is the Type 2 polyphase matrix for the synthesis bank.

Notice that Equations (3.13) and (3.15) provide an alternative way in which to express the analysis and synthesis banks of the QMF bank. Suppose now that we have a QMF bank where the analysis and synthesis banks have been decomposed using type 1 and type 2 polyphase representations, respectively. In this case, these equations give us the transformed, but mathematically equivalent, system shown in Figure 3.13 (a). Using the noble identities, however, we can move the analysis polyphase filtering to the right of the downsamplers and the synthesis polyphase filtering to the left of the upsamplers. This gives us the polyphase form of the filter bank shown in Figure 3.13 (b).

Finally, we can combine the matrices $\mathbf{E}(z)$ and $\mathbf{R}(z)$ as a matrix $\mathbf{P}(z)$, which is defined as

$$\mathbf{P}(z) = \mathbf{R}(z)\mathbf{E}(z)$$

These equivalent circuits are very useful for analytical study as well as in the design and efficient implementation of QMF banks.

(a) Polyphase representation of an M-channel filter bank

(b) After rearrangement using the noble identities

Figure 3.13. Type 1/2 polyphase form of an M-channel QMF bank.

3.5.2 Conditions for PR System

A filter bank with PR property is often desirable in practice. It is only natural to wonder what conditions the filter bank must satisfy in order to have PR. The answer lies in the

polyphase matrices $\mathbf{E}(z)$ and $\mathbf{R}(z)$ introduced in the last section. Intuitively, an M -channel QMF bank with analysis polyphase matrix $\mathbf{E}(z)$ and synthesis polyphase matrix $\mathbf{R}(z)$ has the PR property if the product $\mathbf{P}(z) \equiv \mathbf{R}(z)\mathbf{E}(z)$ has the following form

$$\mathbf{P}(z) = \mathbf{R}(z)\mathbf{E}(z) = \mathbf{I}$$

Then the relationship between the input signal $x[n]$ and the reconstructed signal $y[n]$ is given by $y[n] = x[n-n_0]$, where n_0 is an integer. This condition holds regardless of whether the analysis/synthesis filters are of the FIR or IIR type.

The above condition is sufficient for perfect reconstruction. More generally it can be shown that, the system has perfect reconstruction if and only if the $\mathbf{P}(z)$ has the form

$$\mathbf{P}(z) = \mathbf{R}(z)\mathbf{E}(z) = z^{-m} \begin{bmatrix} \mathbf{0} & \mathbf{I}_{M-r} \\ z^{-1}\mathbf{I}_r & \mathbf{0} \end{bmatrix} \quad (\text{most general PR condition}),$$

for some integer r with $0 \leq r \leq M-1$, some integer, m , and some non-zero constant, c . If these conditions hold, then $y[n] = cx[n-n_0]$, where $n_0 = Mm+r+M-1$.

The proof for this condition can be found in Appendix A.

Example 3.3:

As an example, let us consider the two channel QMF bank. The matrices $\mathbf{E}(z)$, $\mathbf{R}(z)$ and $\mathbf{P}(z)$ are 2×2 now. This system has perfect reconstruction if and only if $\mathbf{P}(z)$ has the form

$$\mathbf{P}(z) = \mathbf{R}(z)\mathbf{E}(z) = z^{-n_0} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Suppose the analysis filters are $H_0(z) = 1$ and $H_1(z) = 1-z^{-1}$. From equation (3.13) the z -transform domain expression of $\mathbf{E}(z)$ satisfies following equation

$$\mathbf{h}(z) = \mathbf{E}(z^2)\mathbf{e}(z)$$

Substituting the appropriate entries into the matrices results in

$$\mathbf{E}^T(z^2) = \begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix}. \quad \text{Hence} \quad \mathbf{E}(z) = \begin{bmatrix} 1 & 0 \\ 1 & -1 \end{bmatrix}$$

Now, choose $\mathbf{R}(z) = \mathbf{E}^{-1}(z)$ to insure perfect reconstruction, i.e. $y[n] = x[n-1]$. Then,

$$\mathbf{R}(z) = \begin{bmatrix} 1 & 0 \\ 1 & -1 \end{bmatrix}$$

Since the entries of $\mathbf{R}(z)$ are not functions of z , then $\mathbf{R}(z^2) = \mathbf{R}(z)$. Therefore, we can compute the synthesis filter using

$$\mathbf{f}^T(z) = z^{-1} \tilde{\mathbf{e}}(z) \mathbf{R}(z^2),$$

and substituting entries into the matrices results in

$$[F_0(z) \ F_1(z)] = z^{-1} [1 \ z] \begin{bmatrix} 1 & 0 \\ 1 & -1 \end{bmatrix},$$

or equivalently,

$$[F_0(z) \ F_1(z)] = [1+z^{-1} \ -1].$$

Hence, we get synthesis filter bank as following

$$F_0(z) = 1+z^{-1} \quad \text{and} \quad F_1(z) = -1$$

This filter bank system is depicted in Figure 3.14.

Figure 3.14. Filter bank equivalent circuit

3.5.3. QMF Bank Implementation

In principle, the design of a PR QMF bank amounts to decomposing the identity matrix into two factors with desired properties. These two factors are simply the polyphase matrices of the filter bank. Once we have the analysis and synthesis polyphase matrices $\mathbf{E}(z)$ and $\mathbf{R}(z)$, respectively, we are ready to proceed to the implementation of the filter bank. Although the filter bank could be realized by directly implementing the filtering operations in each of the polyphase matrices, it is often beneficial to break the filtering process into a number of smaller and simpler cascaded stages.

Let us consider the analysis side of the filter bank first. Instead of implementing $\mathbf{E}(z)$ directly, we further decompose $\mathbf{E}(z)$ as follows

$$\mathbf{E}(z) = \mathbf{E}_{n-1}(z) \cdots \mathbf{E}_1(z) \mathbf{E}_0(z) \quad (3.16)$$

Each of the $\mathbf{E}_i(z)$ can then be taken to represent a single filtering stage in the final implementation. Similarly, $\mathbf{R}(z)$ can be decomposed to produce

$$\mathbf{R}(z) = \mathbf{R}_{n-1}(z) \cdots \mathbf{R}_1(z) \mathbf{R}_0(z) \quad (3.17)$$

This corresponds to the cascade realization of the synthesis polyphase matrix. In the event that PR condition $\mathbf{R}(z)\mathbf{E}(z)=\mathbf{I}$, we have $\mathbf{R}(z)=\mathbf{E}^{-1}(z)$. Thus, we can choose $\mathbf{R}(z)$ as

$$\mathbf{R}(z) = \mathbf{E}_0^{-1}(z) \mathbf{E}_1^{-1}(z) \cdots \mathbf{E}_{n-1}^{-1}(z) \quad (3.18)$$

This factorization results in a certain symmetry between the analysis and synthesis sides of the filter bank which can often be advantageous for implementation.

3.6 QMF Bank and Wavelet Analysis

3.6.1 QMF Banks and Series Expansions of Signals

Recall the structure of the M -channel QMF bank as shown in Figure 3.15. Assume that the QMF filter bank has PR and, therefore, performs an invertible transformation. Mathematically, there are two different ways to interpret the action of such a QMF bank:

1. First, the QMF bank can be interpreted as calculating the coefficients for a series expansion of the discrete-time input signal $x[n]$. The subband signals $s_i[n]$ then represent the coefficients for this series expansion of $x[n]$.
2. Second, the QMF bank can be viewed as computing the coefficients for a series expansion of a continuous-time signal $g(t)$ from the coefficients of a related series expansion of the same signal. In this case, the input sequence $x[n]$ is interpreted as expansion coefficients for the original series, and the subband signals $s_i[n]$ represent the expansion coefficients for the new series expansion.

From the application of image compression, the first interpretation is probably more realistic. In this application, the input sequence $x[n]$ is typically the sampled values of some continuous-time signal. Relatively, the second interpretation is somewhat artificial although some useful insights can be gained by considering it in some cases.

Figure 3.15 M -channel QMF bank (revisited).

Consider the M -channel QMF bank shown in Figure 3.15. Denote the input to the QMF bank $x[n]$, the subband signals $s_i[n]$, and let $u_i[n]$, $v_i[n]$, and $t_i[n]$ denote the various intermediate signals as shown in the diagram. Further, denote the analysis and synthesis filter impulse responses as $h_i[n]$ and $f_i[n]$, respectively.

The QMF bank takes the input sequence $x[n]$, and transforms it into M new sequences $s_i[n]$ using the analysis filters H_k . Although the number of sequences increases, there is no net growth in the number of samples due to downsampling. Actually, the subband signals $s_i[n]$ can be viewed as the coefficients of a series expansion of $x[n]$. This can be explained below. First, we express the output of the analysis filters as

$$\begin{aligned} u_i[m] &= x[m] * h_i[m] \\ &= \sum_n x[n] h_i[m-n] \\ &= \langle x[n], h_i[m-n] \rangle \end{aligned} \quad (3.19)$$

for $i = 0, 1, \dots, M-1$. From this, we can express the subband signal as

$$\begin{aligned} s_i[m] &= u_i[Mm] \\ &= \langle x[n], h_i[Mm-n] \rangle \\ &= \langle x[n], \phi_k[n] \rangle \end{aligned} \quad (3.20)$$

for $i = 0, 1, \dots, M-1$. Thus, the analysis side of the QMF bank computes the inner products of the input signal $x[n]$ with the analysis basis functions $\phi_k[n]$

$$\phi_k[n] = h_i[Mm-n] \quad \text{for } i = 0, 1, \dots, M-1, m \in \mathbb{Z}$$

where $k = Mm+i$. These basis functions can be obtained by shifted M -fold downsampled versions of the analysis filter impulse responses. Now, considering the synthesis side of the QMF bank, we can express the outputs of the upsamplers as

$$v_i[n] = \begin{cases} s_i[n/M] & \text{if } n \text{ is integer multiple of } M \\ 0 & \text{otherwise} \end{cases} \quad (3.21)$$

for $i = 0, 1, \dots, M-1$. From this, the outputs of the synthesis filters are given by

$$\begin{aligned} t_i[n] &= v_i[n] * f_i[n] \\ &= \sum_m v_i[m] f_i[n-m] \\ &= \sum_m s_i[m/M] f_i[n-m] \\ &= \sum_m s_i[m] f_i[n-Mm] \\ &= \langle s_i[m], f_i[n-Mm] \rangle \end{aligned} \quad (3.22)$$

for $i = 0, 1, \dots, M-1$. Using (3.20) and (3.22), we can write

$$\begin{aligned} y[n] &= \sum_{i=0}^{M-1} t_i[n] \\ &= \sum_{i=0}^{M-1} \sum_m s_i[m] f_i[n-Mm] \\ &= \sum_{i=0}^{M-1} \sum_m \langle x[m], h_i[Mm-n] \rangle f_i[n-Mm] \\ &= \sum_k \langle x[n], \phi_k[n] \rangle \varphi_k[n] \end{aligned} \quad (3.23)$$

where $k = Mm+i$ and $\varphi_k[n] = f_i[n-Mm]$. Thus, we can see that the QMF bank computes a series expansion of $x[n]$ where the synthesis basis functions are $\varphi_k[n]$ and the analysis basis functions are $\phi_k[n]$. The synthesis basis functions are simply M -shifts of the synthesis filter impulse response.

Since the QMF bank has the PR property, it follows that the analysis and synthesis bases are biorthogonal. The dual bases in $\phi_k[n]$ and $\varphi_k[n]$ satisfy the biorthogonal constraint

$$\langle \phi_k[n], \varphi_l[n] \rangle = \delta[k-l] \quad (3.24)$$

In the more general case, we simply have that shifted versions of the analysis basis functions are biorthogonal to the synthesis basis functions. A QMF bank can be viewed as a structure that computes a series expansion of a discrete-time signal using basis functions of a particular form. In other words, such a system simply performs a change of basis. This amounts to nothing more than a linear transform. Thus, the PR QMF bank provides a computationally efficient way to implement a particular class of linear transforms.

Example 3.4: Haar Expansion of Signals

Consider the Haar expansion of a discrete-time signal to show the above theorem of series expansions of signals. The Haar basis functions [40] are given by

$$\varphi_{2k}[n] = \begin{cases} \frac{1}{\sqrt{2}} & n = 2k, 2k+1, \\ 0 & \text{otherwise,} \end{cases} \quad \varphi_{2k+1}[n] = \begin{cases} \frac{1}{\sqrt{2}} & n = 2k, \\ -\frac{1}{\sqrt{2}} & n = 2k+1, \\ 0 & \text{otherwise.} \end{cases}$$

It follows that the even-indexed basis functions are translates of each other, and so are the odd-indexed ones, or

$$\varphi_{2k}[n] = \varphi_0[n-2k], \quad \varphi_{2k+1}[n] = \varphi_1[n-2k].$$

The transform is

$$\begin{cases} X[2k] = \langle \varphi_{2k}, x \rangle = \frac{1}{\sqrt{2}} (x[2k] + x[2k+1]), \\ X[2k+1] = \langle \varphi_{2k+1}, x \rangle = \frac{1}{\sqrt{2}} (x[2k] - x[2k+1]). \end{cases}$$

The reconstruction is obtained from

$$x[n] = \sum_{k \in \mathbb{Z}} X[k] \varphi_k[n].$$

3.6.2 Octave-Band Filter Banks and Wavelet Transform

An easy way to construct multi-channel filter banks is to cascade two-channel banks appropriately. For a 2-channel QMF system, the analysis side of the QMF bank decomposes the input signal $x[n]$ into two subband signals $v_k[n]$, $k = 0, 1$. The synthesis side then recombines these two subband signals to obtain $y[n]$, the reconstructed version of the original signal. It is clear that some or all of the resulting subband signals can again be decomposed with more QMF banks. In other words, this idea can be applied recursively, and the final result is a tree structure filter bank.

Consider the filter bank given in Figure 3.16. We see that the signal is decomposed first via a two-channel filter bank, then the lowpass subband is decomposed again using the same filter bank, and so on. Assume that the two-channel filter banks are perfect reconstruction. If the two-channel filter bank is orthonormal, then it implements a discrete-time wavelet transform associated with the input signal $x[n]$. Such a tree-structured filter bank is called an *octave-band filter bank*. The analysis side of the octave-band filter bank calculates the *forward wavelet transform* and the synthesis side calculates the *inverse wavelet transform*.

Figure 3.16 An octave-band filter bank with J stages

Example 3.5: Octave-band filter bank [40]

Assuming the synthesis filters $g_i[n]$ in Figure 3.16 are Haar filters, which are defined in the z -transform domain as

$$G_0(z) = \frac{1}{\sqrt{2}}(1 + z^{-1}), \quad G_1(z) = \frac{1}{\sqrt{2}}(1 - z^{-1}),$$

In this example, we take $J = 3$. That means that we have three two-channel filter banks. Using the multirate noble identity introduced in Sec.3.2.2, $G(z)$ followed by upsampling 2 is equivalent to upsampling by 2 followed by $G(z^2)$. We can transform this filter bank into a four-channel one as shown in Figure 3.17. The equivalent filters are

$$G_1^{(1)}(z) = G_1(z) = \frac{1}{\sqrt{2}}(1 - z^{-1}),$$

$$G_1^{(2)}(z) = G_0(z)G_1(z^2) = \frac{1}{2}(1 + z^{-1} - z^{-2} - z^{-3}),$$

$$G_1^{(3)}(z) = G_0(z)G_0(z^2)G_1(z^4) = \frac{1}{2\sqrt{2}}(1 + z^{-1} + z^{-2} + z^{-3} - z^{-4} - z^{-5} - z^{-6} - z^{-7}),$$

$$G_0^{(3)}(z) = G_0(z)G_0(z^2)G_0(z^4) = \frac{1}{2\sqrt{2}}(1 + z^{-1} + z^{-2} + z^{-3} + z^{-4} + z^{-5} + z^{-6} + z^{-7}),$$

Figure 3.17 Equivalent four-channel synthesis filter bank with Haar filters. It is obtained by transforming the filter bank from Figure 3.16 using noble identity.

3.6.3 Filter Bank Implementation of Wavelet Transform

Having seen the relationship between PR QMF banks and signal expansions in last two sections, we can now consider how to implement a wavelet transform using octave-band filter banks.

The analysis side of Figure 3.16 can be redrawn as Figure 3.18. After the repeated spectrum splitting, the result is a series of band-pass bands with doubling bandwidth and one low-pass band. In other words, we can perform the same subband analysis by feeding the signal into a band-pass filter bank of which each filter has a bandwidth twice as wide as its left neighbor (the frequency axis runs to the right here) and a low-pass filter. This kind of analysis is generally referred to as multiresolution analysis (MRA) [22]. According to the research by Mallat [22], a wavelet transform is nothing but a subband coding scheme with some constraint. In fact, if we implement the wavelet transform as an octave-band filter bank, we do not even have to specify the wavelet explicitly.

Figure 3.18 Splitting the signal spectrum with an octave-band filter bank

An octave-band filter bank can be viewed as a tree-structure system that computes a series expansion of a continuous-time signal $f(t)$ recursively. This structure is suitable to implement the wavelet transform. In this section, we consider how to implement the wavelet transform as an octave-band filter bank. The definition of wavelet and its relevant properties will be discussed in depth in Chapter 5.

A continuous-time signal $f(t)$ can be expressed in terms of dilated and translated wavelet base function in the following equation [40]:

$$f(t) = \sum_k \lambda_{j-1}(k) \phi(2^{j-1}t - k) + \sum_k \gamma_{j-1}(k) \varphi(2^{j-1}t - k) \quad (3.25)$$

If the scaling function $\phi_{j,k}(t)$ and the wavelets $\varphi_{j,k}(t)$ are orthonormal, then the coefficients $\lambda_{j-1}(k)$ and $\gamma_{j-1}(k)$ are found by taking the inner products:

$$\lambda_{j-1}(k) = \langle f(t), \phi_{j,k}(t) \rangle \quad (3.26)$$

$$\gamma_{j-1}(k) = \langle f(t), \varphi_{j,k}(t) \rangle \quad (3.27)$$

If we choose the scaling function $\phi_{j,k}(t)$ and the wavelets $\varphi_{j,k}(t)$ appropriately, we arrive at the following important results [40]:

$$\lambda_{j-1}(k) = \sum_m h(m-2k) \lambda_j(m) \quad (3.28)$$

$$\gamma_{j-1}(k) = \sum_m g(m-2k) \gamma_j(m) \quad (3.29)$$

These two equations state that the wavelet and scaling function coefficients on a certain scale can be found by calculating a weighted sum of the scaling function coefficients from the previous scale. Since we know that the coefficients $\lambda_{j-1}(k)$ come from the low-

pass side of the split signal spectrum, the weight factor $h(k)$ in (3.28) must be a low-pass filter. Similarly, the weight factor $g(k)$ in (3.29) must be a high-pass filter. This means that (3.28) and (3.29) together form one stage of an octave-band filter bank. The coefficients $h(k)$ can be referred to as the scaling filter and the coefficients $g(k)$ can be referred to as the wavelet filter.

In practical cases, the tree depth of the filter bank need not be very large in order to obtain basis functions that closely resemble sampled versions of the scaling and wavelet functions. Typically, a depth not more than six is sufficient for many sets of filters used in practice. In this sense, the scaling and wavelet functions are very important as they provide a good indication of the shape of the basis functions used in the series expansion of the input signal $f(t)$.

An octave-band filter bank can be also viewed as computing a series expansion of a discrete-time signal $x[n]$. In particular, such a filter bank computes a discrete wavelet transform. Each time the lowpass coefficients are successively decomposed, some of the basis functions in the series expansion are replaced with rescaled versions. As we descend deeper into the tree, the corresponding basis functions become increasingly stretched.

3.7 Design Issues of Filter Bank System

3.7.1 Basis Selection

The series expansion interpretations of filter banks are very important. Typically, we use linear transforms to obtain alternative representations of signals that are more convenient for the application at hand. In image compression applications, we seek more compact

representations of signals. To obtain more compact representations, however, it is necessary to choose a set of basis functions which can efficiently represent a signal. That means that we seek a basis that allows us to accurately represent a signal with as few expansion coefficients as possible. As one might expect, better results are achieved when the synthesis basis functions are similar in shape to the function being represented. For example, if a signal is very smooth, it is advantageous to employ very smooth synthesis basis functions. Similarly, if a signal has many sharp transitions, the synthesis basis functions should also have a similar behavior.

When octave-band filter banks are employed as in the case of wavelet transforms, the shape of the scaling and wavelet functions is of interest. These functions provide a good indication of the type of signal which can be most efficiently represented with a particular transform. As we have shown, the choice of basis functions is determined by the QMF bank filter coefficients. For this reason, the filter coefficients must be chosen appropriately in order to obtain effective transforms. In the other words, the selection of basis function should be adapted to the specific application.

3.7.2 Multidimensional Signals

The signals of interest in this thesis are images. Because images are two-dimensional signals, clearly two-dimensional transforms are needed. Normally, two-dimensional transforms can be constructed from one-dimensional ones. In the case of images, we can simply apply a one-dimensional transform first to the rows and then to the columns of an image (or vice versa). This amounts to nothing more than forming a two-dimensional transform using tensor products of the basis functions of the one-dimensional transform.

To express this in mathematical terms, suppose that we have a one-dimensional transform with analysis and synthesis basis functions ϕ_i and φ_i , respectively. If we apply this one-dimensional transform to each dimension of a two-dimensional signal separately, this is equivalent to expressing the signal $g(x, y)$ as

$$g(x, y) = \sum_i \sum_k \langle g(x, y), \phi_{i,k}(x, y) \rangle \varphi_{i,k}(x, y) \quad (3.30)$$

where the basis functions $\phi_{i,k}(x, y)$ and $\varphi_{i,k}(x, y)$ are given by

$$\phi_{i,k}(x, y) = \phi_i(x) \phi_k(y) \quad (3.31)$$

$$\varphi_{i,k}(x, y) = \varphi_i(x) \varphi_k(y) \quad (3.32)$$

By construction, a transform of this form can always be expressed as two separate one-dimensional transforms. Such a transform is said to be separable [40]. Although separable transforms have axial dependencies that can sometimes be undesirable, they are more efficient to compute than their non-separable counterparts. Because of this reason, separable transforms are most commonly used today, and this thesis considers only such transforms.

3.7.3 Finite Length Signals

The images are usually defined on some finite region. That is that such signals are not infinite in extent. These finite-length signals introduce problems when being processed by a filter bank. The problem is how to handle filtering at the boundaries of the signal. Since the outputs of the filters (in the filter bank) depend on past and/or future sample

values, once we get close enough to a signal edge the filters need sample values that are not defined. There are two general approaches that can be used to address this problem. The first approach is to extend the signal so that it is defined for all possible sample indices. Such an approach is referred to as an extension method. The second approach is to change the filters at the boundaries of the signal in order to avoid needing undefined sample values. This type of approach is referred to as boundary filtering. In the following discussion, we consider two commonly used extension methods (i.e., periodic and symmetric extension) only.

Periodic extension is the simplest approach to handle finite-length signals. With this method, the original finite-length signal is repeated indefinitely to yield a periodic signal. If the original signal is defined over N sample indices, periodic extension results in an infinite-length signal of period N . Assuming we process the extended signal with an M -channel QMF bank. Provided N is evenly divisible by M , the subband signals will all have period N/M . Therefore, only N/M samples are required to completely characterize each subband signal, and only N samples in total are required to completely characterize all of the subband signals. In this way, we have a transform that maps N values to N new values. There is no net growth in the number of non-redundant samples. This property is desirable in many applications especially image compression. However, periodic extension has the potential disadvantage that the extended signal might possess very large jumps at the splice points between periods. Moreover, if N is not evenly divisible by M , periodic extension cannot be employed as described above. Although we can always pad the signal with extra samples so that divisibility is achieved, doing this is often not desirable.

Symmetric extension is another relatively simple approach to handling finite-length signals. With this approach, the original signal is extended so that it is both symmetric and periodic. A signal can either be symmetric about one of its samples or about a point midway between two samples. These two cases are referred to as whole sample symmetry (WS) and half sample symmetry (HS), respectively. The example of both cases is given in Figure 3.19. The centers of symmetry of periodic signals always appear in pairs. If the period of the signal is even, both symmetry centers will be WS. If the period is odd, one will be WS and the other HS.

There is more than one way in which to symmetrically extend a signal. The main difference among them is in the number of times that the first and last sample of the original signal is repeated in the extended signal. Here, we will consider (1,1)-symmetric extension only. In this case, the first and last sample of the original signal appear once (i.e., they are not repeated). An example of this type of extension is illustrated in Figure 3.20. It is worth noting that in the extended signal, the first and last sample appear only once in each period. If a signal has length N , (1,1)-symmetric extension will yield a WS signal of period $2N-2$. If such an extended signal is used as the input to a 2-channel QMF bank with certain properties, the subband signals will be symmetric and $(N-1)$ -periodic. Regardless of whether N is an even or odd number, it is always true that only N samples are required in total to characterize the subband signals completely. Therefore, such a transform maps N values to N new values. In other words, the transform is not expansive. There is no net growth in the number of samples required to represent the original signal. Consequently, in the case of 2-channel QMF banks, symmetric extension yields non-expansive transforms for input signals of arbitrary length. We have seen before that the

periodic extension does not always yield non-expansive transforms. Moreover, symmetric extension has the additional advantage that it does not introduce any jumps in the extended signal.

Figure 3.19. Types of signal symmetries.

(a) Wholesample symmetry. (b) Halfsample symmetry.

Figure 3.20. (1,1)-symmetric extension example.

(a) Original signal. (b) Extended signal.

Chapter 4

Multirate Filter Banks System in Image Coding

4.1 Introduction

Digital filter banks have found wide applications and are particularly useful in subband coding and multiple carrier data transmission. In both applications, critically decimated M -channel analysis and synthesis filter banks are used. Among a great number of different filter-bank approaches, cosine-modulated filter banks have become very popular and have been developed to provide perfect reconstruction (PR) [12][33][38]. The Discrete Fourier transform (DFT) has applications in complex input sequences [12][17].

In this chapter, we consider the discrete-time maximally decimated M -channel uniform filter bank as depicted in Figure 4.1. At the analysis stage, the input signal $x[n]$ is passed through a bank of M analysis filters $H_i(z)$, each of which preserves a frequency band of uniform bandwidth π/M . These M filtered signals are then decimated by M to preserve the system's overall sampling rate. The resulting subband signals can be coded, processed, and/or transmitted independently or jointly. At the synthesis stage, the subbands are combined by a set of upsamplers and M synthesis filters $F_i(z)$ to form the reconstructed signal $y[n]$. Assuming that there is no information loss at the processing

stage, filter banks that yield the output $y[n]$ as a purely delayed version of the input $x[n]$, i.e. $y[n] = x[n-n_0]$, are called perfect reconstruction (PR) filter banks.

Figure 4.1 M -channel maximally-decimated QMF bank

In numerous applications, especially image processing, it is also crucial that all analysis and synthesis filters have linear phase. Besides the elimination of the phase distortion, linear phase systems allow us to use simple symmetric extension methods to accurately handle the boundaries of finite-length signals. Furthermore, the linear phase property can be exploited, leading to faster and more efficient filter bank implementation. From this point on, all of the Filter Banks (FB) discussed in this chapter are Linear-phase Perfect Reconstruction Filter Bank (LRPRFB). For practical purposes, we consider only causal, FIR, and real-coefficient systems.

4.2 Block Transforms

4.2.1 Linear-Phase Perfect Reconstruction

Most multi-band block transforms are M -channel maximally-decimated filter bank as shown in the Figure 4.1. The M -channel filter bank can also be represented in terms of its polyphase matrices as shown in Figure 3.13, where $E(z)$ is the analysis bank's polyphase

matrix and $\mathbf{R}(z)$ is the synthesis bank's polyphase matrix. Both $\mathbf{E}(z)$ and $\mathbf{R}(z)$ are $M \times M$ matrices whose elements are polynomials in z -transform [38]. Now, if $\mathbf{E}(z)$ is invertible with minimum-phase determinant, we define the filter bank as Perfect Reconstruction (PR), or biorthogonal, by simply choosing $\mathbf{R}(z) = \mathbf{E}^{-1}(z)$. In other words, any choice of $\mathbf{R}(z)$ and $\mathbf{E}(z)$ that satisfies

$$\mathbf{R}(z)\mathbf{E}(z) = z^{-l}\mathbf{I} \quad l \geq 0 \quad (4.1)$$

yields PR. A popular choice of $\mathbf{R}(z)$ in several previous works [10] [36] is

$$\mathbf{R}(z) = \mathbf{E}^{-1}(z) = z^{-K}\mathbf{E}^T(z^{-1}), \quad (4.2)$$

where K is the order of a properly-designed $\mathbf{E}(z)$. $\mathbf{R}(z)$ is chosen to be the conjugate transpose of $\mathbf{E}(z)$, which leads to *paraunitary* or *orthogonal* systems. In the case where $\mathbf{E}(z)$ may not be paraunitary but Equation (4.1) still holds, the FB is said to be *biorthogonal*. A paraunitary filter bank is a special case of biorthogonal filter bank.

According to equation (4.2), we can draw the relationship of the synthesis filters $F_k(z)$ to analysis filters $H_k(z)$ (of length $N+1$) as:

$$F_k(z) = z^{-N} H_k(z^{-1}), \quad f_k(n) = h_k(N-n) \quad (4.3)$$

When $\mathbf{E}(z)$ are constant matrices, then both $\mathbf{E}(z)$ and $\mathbf{R}(z)$ are unitary matrices. Examples are the DFT, DCT and orthogonal matrices.

4.2.2 Definition of Block Transform

We assume a one-dimensional input sequence $x[n]$ which is transformed into several coefficients $y_i[n]$, where $y_i[n]$ would belong to the i -th subband. In block transform

processing, the signal is divided into blocks of M samples, and each block is processed independently [20][26]. Let the samples in the m -th block be denoted as

$$\mathbf{x}_m^T = [x_0(m), x_1(m), \dots, x_{M-1}(m)] \quad (4.4)$$

with $x_k(m) = x(m.M+k)$, and let the corresponding transform vector be

$$\mathbf{y}_m^T = [y_0(m), y_1(m), \dots, y_{M-1}(m)] \quad (4.5)$$

For a real unitary transform $\mathbf{A}$, satisfies $\mathbf{A}^T = \mathbf{A}^{-1}$. The forward and inverse transforms for the m -th block are

$$\mathbf{y}_m = \mathbf{A} \mathbf{x}_m \quad \text{and} \quad \mathbf{x}_m = \mathbf{A}^T \mathbf{y}_m \quad (4.6)$$

The rows of $\mathbf{A}$, denoted $\mathbf{a}_n^T$ ($0 \leq n \leq M-1$), are called the basis vectors because they form an orthogonal basis for the M -channel output. The coefficients $[y_0(m), y_1(m), \dots, y_{M-1}(m)]$ represent the corresponding weights of vector $\mathbf{x}_m$ with respect to the above basis.

If the input signal is represented by vector $\mathbf{x}$ while the subbands are grouped into blocks in vector $\mathbf{y}$, we can represent the transform $\mathbf{H}$ which operates over the entire signal as a block diagonal matrix:

$$\mathbf{H} = \text{diag}\{\dots \mathbf{A}, \mathbf{A}, \mathbf{A}, \dots\} \quad (4.7)$$

Where, $\mathbf{H}$ is an orthogonal matrix if $\mathbf{A}$ is also an orthogonal matrix [10]. In summary, a signal is transformed by block segmentation followed by block transformation, which amounts to transforming the signal with a sparse matrix.

The sequences $x_i(m)$ in equation (4.4) are called polyphase components of the input signal $x[n]$. The sequence $y_i(m)$ in equation (4.5) are the subbands resulting from the transform process. In an alternative view of the transformation process, the signal

samples are "blocked" or parallelized into polyphase components through a sequence of delays and decimators as shown in Figure 4.2. Each block is transformed by system A into M subband samples (transformed samples). The inverse transform (for orthogonal transforms) is accomplished by system A^T whose outputs are polyphase components of the reconstructed signal, which are then serialized by a sequence of upsamplers and delays. In this system, blocks are processed independently. Therefore, the transform can be viewed as a multi-input multi-output (MIMO) system.

Figure 4.2 The signal samples are parallelized into polyphase components through a sequence of delays and decimators. Effectively the signal is "blocked" and each block is transformed by system A into M subband samples (transformed samples). Inverse transform (or orthogonal transforms) is accomplished by system A^T whose output are polyphase components of the reconstructed signal, which are then serialized by a sequence of upsamplers and delays.

Example 4.1: M -channel Block Transform

Consider a M -channel block transform with analysis basis function $h_i(m)$, $i, m = 0, 1, \dots, M-1$, and synthesis basis function $g_i(m)$, $i, m = 0, 1, \dots, M-1$. Then the block A is equal to

$$\mathbf{A} = \begin{bmatrix} h_0[M-1] & \cdots & h_0[0] \\ \vdots & \ddots & \vdots \\ h_{M-1}[M-1] & \cdots & h_{M-1}[0] \end{bmatrix} = \begin{bmatrix} g_0[0] & \cdots & g_0[M-1] \\ \vdots & \ddots & \vdots \\ g_{M-1}[0] & \cdots & g_{M-1}[M-1] \end{bmatrix}$$

Because $\mathbf{A}$ is a unitary matrix, it implies that

$$\langle h_i[kM - n], h_j[lM - n] \rangle = \langle g_i[n - kM], g_j[n - lM] \rangle = \delta[i-j] \delta[k-l]$$

Thus, we obtain the orthonormality relations for this case. Denoting $\varphi_{kM-i}[n] = g_i[n - kM]$, we have the set of basis functions $\{\varphi_{kM-i}[n]\} = (g_0[n - kM], g_1[n - kM], \dots, g_{M-1}[n - kM])$, with $i = 0, 1, \dots, M-1$, and $k \in \mathbb{Z}$.

4.2.3 JPEG-Based DCT

The JPEG-based discrete cosine transform (DCT) is a typical blocking transform. It is performed on a 2-D image by acting upon subblocks of adjacent pixels within the image. For example, a 512x512 pixel image may be broken up into an array of 8x8 pixel blocks and a DCT may be performed on each block. It is a special case of an 8-channel filter bank system.

The basic principle of DCT has been discussed in Chapter 2. A one dimensional discrete cosine transform (DCT) can be written in matrix form:

$$\mathbf{Y} = \mathbf{A}\mathbf{x}$$

where $\mathbf{A}$ is an $M \times M$ matrix of coefficients based on the cosine function and $\mathbf{x}$ and $\mathbf{Y}$ are M -point input and output vectors.

For a specific instance, the matrices and vectors are written out for $M = 8$:

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \end{bmatrix} = \begin{bmatrix} c_4 & c_4 & c_4 & c_4 & c_4 & c_4 & c_4 & c_4 \\ c_1 & c_3 & c_5 & c_7 & -c_7 & -c_5 & -c_3 & -c_1 \\ c_2 & c_6 & -c_6 & -c_2 & -c_2 & -c_6 & c_6 & c_2 \\ c_3 & -c_7 & -c_1 & -c_5 & c_5 & c_1 & c_7 & -c_3 \\ c_4 & -c_4 & -c_4 & c_4 & c_4 & -c_4 & -c_4 & c_4 \\ c_5 & -c_1 & c_7 & c_3 & -c_3 & -c_7 & c_1 & -c_5 \\ c_6 & -c_2 & c_2 & -c_6 & -c_6 & c_2 & -c_2 & c_6 \\ c_7 & -c_5 & c_3 & -c_1 & c_1 & -c_3 & c_5 & -c_7 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \quad (4.8)$$

where $c_i = a_i \cdot \cos(\pi i / 2M)$, $M = 8$, $a_i = 1/\sqrt{2}$ for $i = 0$ and $a_i = 1$ for $i \neq 0$. Table 4.1 is the example of basis for 8-channel DCT transform computed with this formula.

Table 4.1. DCT bases example for $M = 8$, based on equation (4.8).
The even basis vectors are symmetric while the odd ones are anti-symmetric.

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
3.535534E-01	4.903926E-01	4.619398E-01	4.157348E-01	3.535534E-01	2.777851E-01	1.913417E-01	9.754516E-02
3.535534E-01	4.157348E-01	1.913417E-01	-9.754516E-02	-3.535534E-01	-4.903926E-01	-4.619398E-01	-2.777851E-01
3.535534E-01	2.777851E-01	-1.913417E-01	-4.903926E-01	-3.535534E-01	9.754516E-02	4.619398E-01	4.157348E-01
3.535534E-01	9.754516E-02	-4.619398E-01	-2.777851E-01	3.535534E-01	4.157348E-01	-1.913417E-01	-4.903926E-01
3.535534E-01	-9.754516E-02	-4.619398E-01	2.777851E-01	3.535534E-01	-4.157348E-01	-1.913417E-01	4.903926E-01
3.535534E-01	-2.777851E-01	-1.913417E-01	4.903926E-01	-3.535534E-01	-9.754517E-02	4.619398E-01	-4.157348E-01
3.535534E-01	-4.157348E-01	1.913417E-01	9.754517E-02	-3.535534E-01	4.903926E-01	-4.619398E-01	2.777851E-01
3.535534E-01	-4.903926E-01	4.619398E-01	-4.157348E-01	3.535534E-01	-2.777851E-01	1.913417E-01	-9.754515E-02

The computation of Equation (4.8) requires $M^2 = 64$ multiplications and 64 additions. The matrices and vectors can be decomposed into two $M/2 \times M/2$ matrix and vector products to save computation:

$$\begin{bmatrix} Y_0 \\ Y_2 \\ Y_4 \\ Y_6 \\ Y_1 \\ Y_3 \\ Y_5 \\ Y_7 \end{bmatrix} = \begin{bmatrix} c_4 & c_4 & c_4 & c_4 \\ c_2 & c_6 & -c_6 & -c_2 \\ c_4 & -c_4 & -c_4 & c_4 \\ c_6 & -c_2 & -c_2 & c_6 \\ c_1 & c_3 & c_5 & c_7 \\ c_3 & c_7 & -c_1 & -c_5 \\ c_5 & -c_1 & -c_7 & c_3 \\ c_7 & -c_5 & c_3 & -c_1 \end{bmatrix} \begin{bmatrix} x_0 + x_7 \\ x_1 + x_6 \\ x_2 + x_5 \\ x_3 + x_4 \\ x_0 - x_7 \\ x_1 - x_6 \\ x_2 - x_5 \\ x_3 - x_4 \end{bmatrix} \quad (4.9)$$

This requires $2(M/2)^2 = 32$ multiplications and $2*2(M/2)^2 = 64$ additions for computation. A flowgraph that results from successive decomposition provides a fast version of this algorithm (see Figure 4.3), as proposed in [1].

Figure 4.3 Flow graph representation of fast 1-D DCT for $M = 8$.

4.3 General Lapped Transform

4.3.1 Problem Formulation

The lapped orthogonal transform (LOT) is a large improvement over DCT for image coding. A lapped transform can be generally defined as any transform whose basis vectors have length L , such that $L > M$, extending across traditional block boundaries. Thus, the transform matrix is no longer square. In this thesis, we will concentrate on the orthogonal lapped transform and consider $L = NM$, where N is the overlap factor. Note that N , M , and L are all integers. As in the case of block transforms, we define the transform matrix as containing the orthogonal basis vectors as its rows. A lapped transform matrix P of dimensions $M \times L$ can be divided into square $M \times M$ submatrices P_i ($i = 0, 1, \dots, N-1$) as follows

$$P = [P_0 \ P_1 \ \dots \ P_{N-1}] \quad (4.10)$$

The orthogonality property does not hold because $\mathbf{P}$ is no longer a square matrix and it is replaced by the perfect reconstruction (PR) property [20], defined by

$$\sum_{i=0}^{N-1-l} \mathbf{P}_i \mathbf{P}_{i+l}^T = \sum_{i=0}^{N-1-l} \mathbf{P}_{i+l}^T \mathbf{P}_i = \delta(l) \mathbf{I}_M, \quad (4.11)$$

for $l = 0, 1, \dots, N-1$, where $\delta(l)$ is the Kronecker delta, i.e. $\delta(0) = 1$ and $\delta(l) = 0$ for $l \neq 0$. It can be proved that equation (4.11) states the PR conditions and orthogonality of the transform operating over the entire signal [9][20].

If we divided the signal into blocks, each of size M , we would have vectors $\mathbf{x}_m$ and $\mathbf{y}_m$ such as in Equations (4.3) and (4.4). These blocks are not used by the matrix lapped transforms in a straightforward manner. The actual vector, which is transformed by the matrix $\mathbf{P}$, has to have L samples and, at block number m , it is composed of the samples of $\mathbf{x}_m$ plus $L-M$ samples from the neighboring blocks. These samples are chosen by picking $(L-M)/2$ samples at each side of the block $\mathbf{x}_m$, as shown in Figure 4.4, for $N = 2$. However, the number of transform coefficients at each step is M , and in this respect, there is no change in the way we represent the transform-domain blocks $\mathbf{y}_m$.

Figure 4.4 Basic functions of the LOT overlap across block boundaries so that each basis function has the length of two blocks of length M .

The input vector of length L is denoted as $\mathbf{z}_m$, which is centered around block $\mathbf{x}_m$, and can be represented as

$$\mathbf{z}_m^T = [\mathbf{x}_{m-N+1}^T, \dots, \mathbf{x}_{m-1}^T, \mathbf{x}_m^T] \quad (4.12)$$

Then, we have

$$\mathbf{y}_m = \mathbf{P} \mathbf{z}_m = \sum_{i=0}^{N-1} \mathbf{P}_{N-1-i} \mathbf{x}_{m-i} \quad (4.13)$$

If we are not concerned with the localization of the transform, $\mathbf{z}'_m$ can be reconstructed from $\mathbf{y}_m$, as:

$$\mathbf{z}'_m = \mathbf{P}^T \mathbf{y}_m \quad (4.14)$$

Example 4.2: Block Transform with Overlap.

Transforms with overlap is shown in Figure 4.5(a), where the matrix $\mathbf{P}$ is $M \times L$ rather than $M \times M$, with $L \geq M$. This means that the input is partitioned into overlapping blocks

$$\mathbf{x}[n] = [x[nM] \ x[nM-1] \ \dots \ x[nM-L+1]]^T.$$

of length L , and each block transformed into a block of (smaller) length M . Let $L = 2M$, Figure 4.5(b) demonstrates this overlap for $L = 6$, and $M = 3$.

As shown in Figure 4.5(a), $\mathbf{P}$ is an $M \times L$ transform matrix. By partitioning $\mathbf{P}$ as

$$\mathbf{P} = [\mathbf{P}_0 \ \mathbf{P}_1],$$

where $\mathbf{P}_0$ and $\mathbf{P}_1$ are $M \times M$. The analysis filter vector $\mathbf{h}(z)$ can be written as

$$\mathbf{h}(z) = \mathbf{P} \mathbf{e}(z) = [\mathbf{P}_0 + z^{-M} \mathbf{P}_1] \mathbf{e}(z)$$

where $\mathbf{e}(z)$ is the delay chain vector $[1 \ z^{-1} \ \dots \ z^{-(M-1)}]^T$. By comparing with equation (3.13),

we see that the analysis bank has $M \times M$ polyphase component matrix

$$\mathbf{E}(z) = \mathbf{P}_0 + z^{-M} \mathbf{P}_1.$$

We know that if $\mathbf{E}(z)$ is paraunitary, we can obtain perfect reconstruction (in absence of quantization) by taking the synthesis filter to be

$$f_k[n] = h_k^*[2M-1-n].$$

Figure 4.5 Example of Lapped Orthogonal Transform (LOT). (a) The transform coder/decoder with blocks overlapping. (b) Demonstration of overlap, with $L=6$ and $M=3$.

4.3.2 Lapped Transform

Factorization of LOT is based on the DCT because of its fast implementation and near-optimal performance for block transform coding [20]. The DCT matrix $\mathbf{D}$ is defined as having entries.

$$d_{ij} = \sqrt{\frac{2}{M}} k_i \cos\left(\frac{(2j+1)i\pi}{2M}\right) \quad (4.15)$$

where $k_0=1$ and $k_i = 1/\sqrt{2}$, for $1 \leq i \leq M-1$.

The resulting transform matrix for the LOT, assuming blocks of M samples, is nonsquare and given [9] by

$$\mathbf{P} = \begin{bmatrix} \mathbf{U} & \mathbf{0} \\ \mathbf{0} & \mathbf{V} \end{bmatrix} \begin{bmatrix} \mathbf{D}_e - \mathbf{D}_o & \mathbf{J}_{M/2}(\mathbf{D}_e - \mathbf{D}_o) \\ \mathbf{D}_e - \mathbf{D}_o & -\mathbf{J}_{M/2}(\mathbf{D}_e - \mathbf{D}_o) \end{bmatrix} \quad (4.16)$$

where $\mathbf{D}_e$ is the $M/2 \times M$ matrix with even-symmetric basis functions of the DCT, and $\mathbf{D}_o$ is the matrix with even-symmetric ones. $\mathbf{U}$ and $\mathbf{V}$ are $M/2 \times M/2$ orthogonal matrices. $\mathbf{J}_{M/2}$ is a $M/2 \times M/2$ counter-identity matrix.

Example 4.3: Lapped Transform

Consider a simple 4-channel example to illustrate LT. Take the DCT matrix $\mathbf{D}$ as below:

$$\mathbf{D} = \begin{bmatrix} d_2 & d_2 & d_2 & d_2 \\ d_1 & d_3 & -d_3 & -d_1 \\ d_2 & -d_2 & -d_2 & d_2 \\ d_3 & -d_1 & d_1 & -d_3 \end{bmatrix}$$

where the coefficients can be computed with Equation (4.15).

$$d_1 = \cos 22.5^\circ = 0.9239, d_2 = \cos 45.0^\circ = 0.7071, d_3 = \cos 67.5^\circ = 0.3827.$$

Obviously, $\mathbf{D}$ is an invertible matrix. For simplicity, we can assume $\mathbf{U} = -\mathbf{V} = \mathbf{I}$ then

$$\begin{aligned} \mathbf{P} &= \begin{bmatrix} \mathbf{I} & 0 \\ 0 & -\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{D}_o - \mathbf{D}_e & \mathbf{J}_{M/2}(\mathbf{D}_o - \mathbf{D}_e) \\ \mathbf{D}_o - \mathbf{D}_e & -\mathbf{J}_{M/2}(\mathbf{D}_o - \mathbf{D}_e) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 \\ 0 & 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 2d_2 & 2d_2 & 0 & d_1 - d_3 & d_1 + d_3 & -d_1 - d_3 & -d_1 + d_3 \\ d_1 - d_3 & d_1 + d_3 & -d_1 - d_3 & -d_1 + d_3 & 0 & 2d_2 & 2d_2 & 0 \\ 0 & 2d_2 & 2d_2 & 0 & -d_1 + d_3 & -d_1 + d_3 & d_1 + d_3 & d_1 - d_3 \\ d_1 - d_3 & d_1 + d_3 & -d_1 - d_3 & -d_1 + d_3 & 0 & 2d_2 & 2d_2 & 0 \end{bmatrix} \\ &= \begin{bmatrix} d_1 - d_3 & d_1 + 2d_2 + d_3 & -d_1 + 2d_2 - d_3 & -d_1 + d_3 & d_1 - d_3 & d_1 + 2d_2 + d_3 & -d_1 + 2d_2 - d_3 & -d_1 - d_3 \\ d_1 - d_3 & d_1 + 2d_2 - d_3 & -d_1 + 2d_2 - d_3 & -d_1 + d_3 & d_1 - d_3 & d_1 + 2d_2 + d_3 & -d_1 + 2d_2 - d_3 & -d_1 - d_3 \\ -d_1 + d_3 & -d_1 - 2d_2 - d_3 & d_1 - 2d_2 + d_3 & d_1 - d_3 & d_1 - d_3 & d_1 - 2d_2 - d_3 & -d_1 - 2d_2 - d_3 & -d_1 + d_3 \\ -d_1 + d_3 & -d_1 - 2d_2 - d_3 & d_1 - 2d_2 + d_3 & d_1 - d_3 & d_1 - d_3 & d_1 - 2d_2 - d_3 & -d_1 - 2d_2 - d_3 & -d_1 + d_3 \end{bmatrix} \end{aligned}$$

From equation (3.12), we can get the analysis filter bank as following

$$\begin{aligned} H_0(z) &= (d_1 - d_3) + (d_1 + 2d_2 - d_3)z^{-1} - (d_1 - 2d_2 + d_3)z^{-2} - (d_1 - d_3)z^{-3} + (d_1 - d_3)z^{-4} + (d_1 + 2d_2 + d_3)z^{-5} - (d_1 - 2d_2 + d_3)z^{-6} - (d_1 - d_3)z^{-7} \\ H_1(z) &= (d_1 - d_3) + (d_1 + 2d_2 - d_3)z^{-1} - (d_1 - 2d_2 + d_3)z^{-2} - (d_1 - d_3)z^{-3} + (d_1 - d_3)z^{-4} + (d_1 + 2d_2 + d_3)z^{-5} - (d_1 - 2d_2 + d_3)z^{-6} - (d_1 - d_3)z^{-7} \\ H_2(z) &= (-d_1 + d_3) - (d_1 + 2d_2 + d_3)z^{-1} + (d_1 - 2d_2 + d_3)z^{-2} + (d_1 - d_3)z^{-3} + (d_1 - d_3)z^{-4} + (d_1 - 2d_2 - d_3)z^{-5} - (d_1 + 2d_2 + d_3)z^{-6} - (d_1 - d_3)z^{-7} \\ H_3(z) &= (-d_1 + d_3) - (d_1 + 2d_2 + d_3)z^{-1} + (d_1 - 2d_2 + d_3)z^{-2} + (d_1 - d_3)z^{-3} + (d_1 - d_3)z^{-4} + (d_1 - 2d_2 - d_3)z^{-5} - (d_1 + 2d_2 + d_3)z^{-6} - (d_1 - d_3)z^{-7} \end{aligned}$$

It can be found that the length of the 4-channel filter bank is 8, or $L = 2M = 2 \times 4 = 8$.

4.3.3 Lattice Structure Implementation

In Equation (4.16), as long as $\mathbf{U}$ and $\mathbf{V}$ remain as orthogonal matrices, the LT is orthogonal [9]. In terms of the polyphase transfer matrix, as shown in Figure 4.2, let $\mathbf{A} = \mathbf{E}(z)$, we have

$$\mathbf{W} = \frac{1}{\sqrt{2}} \begin{bmatrix} \mathbf{I}_{M/2} & \mathbf{I}_{M/2} \\ \mathbf{I}_{M/2} & -\mathbf{I}_{M/2} \end{bmatrix} \quad (4.17)$$

$$\Phi_i = \begin{bmatrix} \mathbf{U}_i & \mathbf{0}_{M/2} \\ \mathbf{0}_{M/2} & \mathbf{V}_i \end{bmatrix} \quad (4.18)$$

$$\Lambda(z) = \begin{bmatrix} \mathbf{I}_{M/2} & \mathbf{0}_{M/2} \\ \mathbf{0}_{M/2} & z^{-1}\mathbf{I}_{M/2} \end{bmatrix} \quad (4.19)$$

and let $\mathbf{D}$ be the $M \times M$ DCT matrix. Then, for the general LOT,

$$\mathbf{E}(z) = \Phi_1 \mathbf{W} \Lambda(z) \mathbf{W} \mathbf{D} \quad (4.20)$$

where $\mathbf{U}_1 = \mathbf{U}$ and $\mathbf{V}_1 = -\mathbf{V}$. It is well-known that any orthogonal matrix $\mathbf{U}_i$ can always be factorized into a series of planar (Givens) rotations [10] as illustrated in Figure 4.6(c).

Where

$$\mathbf{U}_{\theta_i} = \begin{bmatrix} \cos \theta_i & \sin \theta_i \\ -\sin \theta_i & \cos \theta_i \end{bmatrix}$$

From this formulation along with other results it was realized that all orthogonal systematic lapped transforms can be expressed as:

$$\mathbf{E}(z) = \mathbf{G}_{N-1}(z) \mathbf{G}_{N-2}(z) \cdots \mathbf{G}_2(z) \mathbf{G}_1(z) \mathbf{E}_0 = \prod_{i=N-1}^1 \mathbf{G}_i(z) \mathbf{E}_0 \quad (4.21)$$

where

$$\begin{aligned} \mathbf{G}_i(z) &= \frac{1}{2} \begin{bmatrix} \mathbf{U}_i & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_i \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{I} \\ \mathbf{I} & -\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & z^{-1}\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{I} \\ \mathbf{I} & -\mathbf{I} \end{bmatrix} \\ &= \frac{1}{2} \Phi_i \mathbf{W} \Lambda(z) \mathbf{W} \end{aligned} \quad (4.22)$$

All of the delays are now contained in $\Lambda(z)$, while $\mathbf{W}$ resembles the famous "butterfly" matrix in the FFT implementation. Since $\mathbf{W}$ and $\Lambda(z)$ are paraunitary, $\mathbf{G}(z)$ is invertible if and only if Φ_i is invertible, i.e. $\mathbf{U}$ and $\mathbf{V}$ are invertible. A cascade of $(K-1)$ blocks $\mathbf{G}_i = \frac{1}{2} \Phi_i \mathbf{W} \Lambda(z) \mathbf{W}$ and a zero-order initial block $\mathbf{E}_0$ generates the polyphase matrix of an even-channel LPPRFB with filter length $L = KM$. The starting block $\mathbf{E}_0$ has no delay element, represents an LPPRFB of length M , and was often chosen to be the DCT [9]. The most general $\mathbf{E}_0$ that satisfies Equation (4.21) has the following form:

$$\mathbf{E}_0 = \frac{1}{\sqrt{2}} \begin{bmatrix} \mathbf{U}_0 & \mathbf{U}_0 \mathbf{J} \\ \mathbf{V}_0 \mathbf{J} & -\mathbf{V}_0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} \mathbf{U}_0 & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_0 \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{J} \\ \mathbf{J} & -\mathbf{I} \end{bmatrix} \quad (4.23)$$

For $\mathbf{E}_0$ to have PR, $\mathbf{U}_0$ and $\mathbf{V}_0$ again have to be invertible. The polyphase matrix of the corresponding causal synthesis bank is then

$$\mathbf{R}(z) = z^{-K-1} \mathbf{E}_0^{-1}(z) \mathbf{G}_1^{-1}(z) \mathbf{G}_2^{-1}(z) \cdots \mathbf{G}_{K-2}^{-1}(z) \mathbf{G}_{K-1}^{-1}(z) = \mathbf{E}_0^{-1} \prod_{i=1}^{K-1} z^{-1} \mathbf{G}_i(z) \quad (4.24)$$

The complete lattice structure for both analysis and synthesis bank is depicted in Figure 4.6(a) and (b).

Figure 4.6. Implementation of a more general version of the LOT for $M=8$

Example 4.4:

The following simple example is used to show how to implement the lapped transform.

Let us consider a simple case with $M = 4$. For simplicity, we take $\theta_0 = \theta_1 = \pi/4 = 45^\circ$, $U_1 = V_1$, $U_0 = V_0$. Then, from Figure 4.6 (c), we have following matrices:

$$U_1 = V_1 = U_0 = V_0 = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$$

Next, E_0 and $G_1(z)$ have following forms based on Equations (4.22) and (4.23):

$$\begin{aligned} G(z) &= \frac{1}{\sqrt{2}} \begin{bmatrix} U_1 & 0 \\ 0 & V_1 \end{bmatrix} \begin{bmatrix} I & I \\ I & -I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & z^{-1}I \end{bmatrix} \begin{bmatrix} I & I \\ I & -I \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} U_1 + z^{-1}U_1 & U_1 - z^{-1}U_1 \\ V_1 - z^{-1}V_1 & V_1 + z^{-1}V_1 \end{bmatrix} \\ &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 + z^{-1} & 1 + z^{-1} & 1 - z^{-1} & 1 - z^{-1} \\ -(1 + z^{-1}) & 1 + z^{-1} & -(1 - z^{-1}) & 1 - z^{-1} \\ 1 - z^{-1} & 1 - z^{-1} & 1 + z^{-1} & 1 + z^{-1} \\ -(1 - z^{-1}) & 1 - z^{-1} & -(1 + z^{-1}) & 1 + z^{-1} \end{bmatrix} \end{aligned}$$

$$E_0 = \begin{bmatrix} U_0 & 0 \\ 0 & V_0 \end{bmatrix} \begin{bmatrix} I & J \\ J & -I \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} U_0 & U_0 J \\ V_0 J & -V_0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$

So, lapped transform $E(z)$ can be expressed as follows:

$$\begin{aligned} E(z) &= G_1(z)E_0 \\ &= \frac{1}{2} \begin{bmatrix} 1 + z^{-1} & 1 + z^{-1} & 1 - z^{-1} & 1 - z^{-1} \\ -(1 + z^{-1}) & 1 + z^{-1} & -(1 - z^{-1}) & 1 - z^{-1} \\ 1 - z^{-1} & 1 - z^{-1} & 1 + z^{-1} & 1 + z^{-1} \\ -(1 - z^{-1}) & 1 - z^{-1} & -(1 + z^{-1}) & 1 + z^{-1} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \\ &= \frac{1}{2} \begin{bmatrix} 1 - z^{-1} & 1 + z^{-1} & 1 + z^{-1} & -(1 - z^{-1}) \\ -(1 + z^{-1}) & -(1 - z^{-1}) & 1 - z^{-1} & -(1 + z^{-1}) \\ 1 + z^{-1} & 1 - z^{-1} & 1 - z^{-1} & -(1 + z^{-1}) \\ -(1 - z^{-1}) & -(1 + z^{-1}) & 1 + z^{-1} & -(1 - z^{-1}) \end{bmatrix} \end{aligned}$$

In LPPRFB, every transform can be characterized by a modular lattice structure. The key idea behind the lattice structure is the factorization of the Filter Bank's analysis and synthesis polyphase matrices into cascades of lower-order modules. The lattice offers the following advantages:

- fast and efficient implementation
- robust: both LP and PR properties are retained regardless of lattice coefficient quantization
- perfect for hardware implementation
- powerful filter bank design tool using unconstrained optimization

4.4 Linear-phase Biorthogonal Lapped Transform

4.4.1 Bases of Coefficient Matrix

In this chapter, we implement a general linear-phase biorthogonal lapped transform (GLBT). Its corresponding polyphase matrix is given by

$$\mathbf{E}(z) = \mathbf{G}_1(z)\mathbf{E}_0 \quad (4.25)$$

where,

$$\mathbf{G}_1(z) = \Phi_1 \mathbf{W} \Lambda(z) \mathbf{W} \quad (4.26)$$

Where $\mathbf{W}$ and $\Lambda(z)$ are given as in Equations (4.17) and (4.19), $\mathbf{E}_0$ in Equation (4.23), and Φ_1 is given as in Equation (4.18), and factors $\mathbf{U}_1$ and $\mathbf{V}_1$ are only required to be invertible. The inverse transform can be implemented based on Equation (4.24). The diagram for the implementation of the linear-phase biorthogonal lapped transform is shown in Figure 4.6. Tables 4.2 and 4.3 are an 8-channel LPPRFB with 16-tap filters (8x16 GLBT) designed

by de Querioz and Tran [9]. It is designed to maximize coding gain, minimize the DC leakage, and maximize the stopband attenuation in low frequency bands [9][36].

The example of bases for the linear-phase biorthogonal lapped transform of particular interest to image compression [9] is given in Table 4.2 and 4.3.

4.4.2 Design Issues in Image Coding

Coding gain is a measure that indicates the effectiveness of a system for data compression. To a certain extent, this quantity is correlated with the visual appearance of coded images. It is well known that the maximum coding gain increases with the number of channels as well as the filter order. Here we define σ_x^2 as the variance of the input signal $x[n]$, $\sigma_{x_i}^2$ as the variance of the i -th subband, and $\|f_i\|^2$ as the norm of the i -th synthesis filter. With several assumptions including scalar quantization and a sufficiently large bit rate, the generalized coding gain can be formulated as follows [9]:

$$C_{\text{coding gain}} = 10 \log_{10} \frac{\sigma_x^2}{\left(\prod_{i=0}^{M-1} \sigma_{x_i}^2 \|f_i\|^2 \right)^{\frac{1}{M}}} \quad (4.27)$$

In addition to the coding gain, we can incorporate other desirable properties in the design process. For example, it is possible to reduce the blocking effects by minimizing the stopband energies of the synthesis lowpass filter $g_0(n)$ at frequencies (i/M) , $i = 1, \dots, M/2$ [6]. We can also force the synthesis filters to assume small values at the ends of the impulse responses to improve their smoothness [6].

Table 4.2. Forward GLBT bases example [9] for $M = 8$ and $N = 2$.

The even basis vectors are symmetric while the odd ones are anti-symmetric.

P_{0n}	P_{1n}	P_{2n}	P_{3n}	P_{4n}	P_{5n}	P_{6n}	P_{7n}
-2.1191909e-01	-1.8196802e-01	1.0914440e-04	-9.4264236e-02	3.8595789e-02	-3.4930837e-02	4.9970195e-02	1.9559361e-02
-1.3961840e-01	-1.9661598e-01	1.6037001e-01	5.3338393e-02	9.2327086e-02	1.2467833e-01	-9.2404582e-02	-3.1344144e-02
-3.3872708e-02	-9.5395778e-02	1.7972542e-01	2.5597816e-01	-2.4357678e-01	-1.2311300e-01	1.0673994e-02	-1.9906380e-02
9.3595665e-02	1.0868480e-01	-6.3474352e-02	-1.3317786e-02	-5.6128734e-02	-1.0218159e-01	1.6422597e-01	1.1626840e-01
2.3114228e-01	3.4100854e-01	-3.6292787e-01	-3.9497880e-01	4.2911613e-01	3.6083514e-01	-3.5631454e-01	-2.2434022e-01
3.5831589e-01	4.6362301e-01	-3.5055566e-01	-1.6415283e-01	-1.3163058e-01	-3.1279511e-01	4.7723341e-01	3.1906754e-01
4.6619482e-01	4.2905620e-01	7.3131597e-03	4.2662185e-01	-4.5465048e-01	-7.4344824e-02	-4.0584883e-01	-3.8322186e-01
5.3812793e-01	2.2604475e-01	4.2944014e-01	3.6069631e-01	3.2594757e-01	4.3221619e-01	1.5246437e-01	3.9834371e-01
5.3812793e-01	-2.2604475e-01	4.2944014e-01	-3.6069631e-01	3.2594757e-01	-4.3221619e-01	1.5246437e-01	-3.9834371e-01
4.6619482e-01	-4.2905620e-01	7.3131597e-03	-4.2662185e-01	-4.5465048e-01	7.4344824e-02	-4.0584883e-01	3.8322186e-01
3.5831589e-01	-4.6362301e-01	-3.5055566e-01	1.6415283e-01	-1.3163058e-01	3.1279511e-01	4.7723341e-01	-3.1906754e-01
2.3114228e-01	-3.4100854e-01	-3.6292787e-01	3.9497880e-01	4.2911613e-01	-3.6083514e-01	-3.5631454e-01	2.2434022e-01
9.3595665e-02	-1.0868480e-01	-6.3474352e-02	1.3317786e-02	-5.6128734e-02	1.0218159e-01	1.6422597e-01	-1.1626840e-01
-3.3872708e-02	9.5395778e-02	1.7972542e-01	-2.5597816e-01	-2.4357678e-01	1.2311300e-01	1.0673994e-02	1.9906380e-02
-1.3961840e-01	1.9661598e-01	1.6037001e-01	-5.3338393e-02	9.2327086e-02	-1.2467833e-01	-9.2404582e-02	3.1344144e-02
-2.1191909e-01	1.8196802e-01	1.0914440e-04	9.4264236e-02	3.8595789e-02	3.4930837e-02	4.9970195e-02	-1.9559361e-02

Table 4.3. Inverse GLBT bases example [9] for $M = 8$ and $N = 2$.

The even basis vectors are symmetric while the odd ones are anti-symmetric.

P_{0n}	P_{1n}	P_{2n}	P_{3n}	P_{4n}	P_{5n}	P_{6n}	P_{7n}
1.7863920e-02	-1.4411799e-02	6.1315639e-02	1.9517156e-02	5.2426546e-02	5.3408655e-02	4.6076341e-02	8.3324115e-02
5.6920246e-02	-1.6813024e-02	1.6037108e-01	1.2407306e-01	4.8882116e-02	1.6064616e-01	-9.0423936e-02	-2.1939189e-02
1.0664692e-01	6.5754854e-02	1.2462013e-01	2.4092241e-01	-2.1792622e-01	-1.3556294e-01	2.1084469e-02	-2.0631974e-04
1.6255815e-01	2.0555316e-01	-1.2304453e-01	-3.5598739e-02	-2.1812290e-02	-8.4323166e-02	1.3397362e-01	1.2746610e-01
2.2147634e-01	3.4661124e-01	-3.8106502e-01	-3.5546845e-01	3.6530403e-01	3.9610216e-01	-3.0169785e-01	-2.3277757e-01
2.7738757e-01	4.0526043e-01	-3.2843219e-01	-1.2297514e-01	-1.2623361e-01	-3.5462422e-01	4.1230527e-01	3.4133429e-01
3.2711425e-01	3.3120028e-01	3.9391208e-02	3.8507205e-01	-3.8248451e-01	-8.3609953e-02	-3.5154928e-01	-4.0905814e-01
3.6617057e-01	1.3189886e-01	4.4324237e-01	2.9999633e-01	2.8190585e-01	4.5454663e-01	1.3231892e-01	4.1413507e-01
3.6617057e-01	-1.3189886e-01	4.4324237e-01	-2.9999633e-01	2.8190585e-01	-4.5454663e-01	1.3231892e-01	-4.1413507e-01
3.2711425e-01	-3.3120028e-01	3.9391208e-02	-3.8507205e-01	-3.8248451e-01	8.3609953e-02	-3.5154928e-01	4.0905814e-01
2.7738757e-01	-4.0526043e-01	-3.2843219e-01	1.2297514e-01	-1.2623361e-01	3.5462422e-01	4.1230527e-01	-3.4133429e-01
2.2147634e-01	-3.4661124e-01	-3.8106502e-01	3.5546845e-01	3.6530403e-01	-3.9610216e-01	-3.0169785e-01	2.3277757e-01
1.6255815e-01	-2.0555316e-01	-1.2304453e-01	3.5598739e-02	-2.1812290e-02	8.4323166e-02	1.3397362e-01	-1.2746610e-01
1.0664692e-01	-6.5754854e-02	1.2462013e-01	-2.4092241e-01	-2.1792622e-01	1.3556294e-01	2.1084469e-02	2.0631974e-04
5.6920246e-02	1.6813024e-02	1.6037108e-01	-1.2407306e-01	4.8882116e-02	-1.6064616e-01	-9.0423936e-02	2.1939189e-02
1.7863920e-02	1.4411799e-02	6.1315639e-02	-1.9517156e-02	5.2426546e-02	-5.3408655e-02	4.6076341e-02	-8.3324115e-02

4.5 Coding Results and Analysis

4.5.1 Coding and Implementation

An image compression program package based on GLBT algorithms has been developed in this thesis. This algorithm is based on the principle of lapped transform and JPEG, but it does not suffer from the block-effect, and its performance is superior to JPEG and competitive with EZW. The code has been designed for experimentation. The quantizer is the same as that of JPEG coders. The entropy coder is based on the arithmetic algorithms. The simplified processing steps are illustrated as below:

Figure 4.7 GLBT-based Encoder/Decoder Processing Steps

Uniform quantization on all transform coefficients has been performed. A standard 512×512 with 8-bit gray scale image "Lena" has been used for testing (Figure 4.10). The objective coding results (PSNR in *dB* and time in *second* versus the bit rate) are tabulated in Table 4.4. Figure 4.8 and Figure 4.9 present the results of the GLBT coder compared with the DCT-based coder and wavelet-based EZW coding scheme in compression and coding time. All computed PSNR quotes in *dB* are obtained from a real compressed bit

stream with all overheads included. Coding time is mainly devoted to the quantizer, the decoding procedure is very fast. The encoding and decoding are carried out in a Pentium III 600-MHz PC computer.

4.5.2. Results

A 512 x 512 x 8-bpp Lena image has been used for evaluation. Experimental results are satisfactory both in terms of the visual appearance of the reconstructed image and in terms of PSNR versus bit rate. From the results, we find that the block transform scheme performs better over the entire range of bit rates. The PSNR quoted from Table 4.4 and the rate-PSNR curves in Figure 4.8 clearly demonstrate the superiority of the well-optimized lapped transform over the baseline JPEG algorithm. The reconstructed images are shown in Figure 4.12-15, respectively. It is seen that, at compression ratios less than 20:1 or so, the JPEG performs almost same numerically with that of the GLBT and the simple wavelet coder EZW. The coding time of JPEG coder is shortest. At compression ratios above 30:1, the performance of the JPEG deteriorates rapidly (Figure 4.11), while the performance of the GLBT coder degrades gracefully well beyond ratios of 50:1. As it may be seen, for the GLBT coder, the blocking effect is not visible in the reconstructed images until the compression ratio beyond 1:100 (Figure 4.15).

Table 4.4 Coding Results for 512 x 512 x 8 bit Lena

Compression Ratio	RMSE	PSNR (dB)	Coding Time (second)
4.4	1.8	42.8	1.6
8.4	2.8	39.0	1.4
15.6	4.1	35.8	1.2
28.8	5.9	32.7	0.1
55.2	8.5	29.1	1.0
106.6	12.7	25.0	1.0

Figure 4.8 Comparative Evaluation of GLBT in Compression

Figure 4.9 Comparative Evaluation of GLBT in Coding Time

Fig.4.10 Original Lena (512×512×8)

Fig. 4.11 JPEG-DCT: CR=64, PSNR=24.4 dB

Fig.4.12 GLBT: CR=15.6 PSNR=35.8 dB

Fig.4.13 GLBT: CR=28.8 PSNR=32.7 dB

Fig.4.14 GLBT CR=55.2 PSNR=29.1 dB

Fig.4.15 GLBT CR=106.6 PSNR=25.0 dB

4.5.3 Discussion

Although the experimental results of the GLBT algorithm demonstrate good performance, there are still some problems with it. The GLBT algorithm is superior to the simple DCT-based JPEG coder in suppressing blocking effects at high compression ratios, but it cannot eliminate them completely. Moreover, the good performance of the GLBT algorithm is at the expense of computing complexity. Its coding time is longer than that of the JPEG algorithm.

Compared with the wavelet-based EZW methods, it is found that the performance of the GLBT algorithm is inferior to the wavelet-based EZW coder by 0.5 dB on average. However the coding time of the GLBT algorithm is much shorter than that of the wavelet-based EZW coder. In our test, the average coding time of the GLBT is only about two-thirds to one-half of that of wavelet-based EZW methods.

In the implementation of the GLBT algorithm, we used an 8-channel 16-tap FIR filter bank. This is a simple filter bank system. It can be expected that if we use a filter bank with longer taps, better performance would be obtained but with the cost of longer coding time. There will be a trade-off between performance and execution time. Moreover, the quantizer we used is a simple uniform quantization method, which would affect the performance.

The choice of an image coding scheme for a particular application will depend on its requirements. In the case where low complexity is of high priority, JPEG stands out as the best scheme. The EZW coder provides the most effective and flexible solution, if its relative higher complexity is acceptable. The GLBT algorithm provides a compromised solution with moderate efficiency and complexity.

Chapter 5

Wavelet-Based Image Compression

5.1. Introduction

Despite all the advantages of the image compression scheme based on the DCT-based JPEG algorithm namely simplicity, satisfactory performance, and availability of special purpose hardware for implementation, there are still some shortcomings. The DCT-based scheme gives rise to noticeable and annoying "blocking artifacts" at low bit rates. To overcome this shortcoming, the lapped orthogonal transform (LOT) was introduced to solve this problem. Although block transform based lapped orthogonal transforms can effectively improve visualization, it also brings about increased computational complexity. It is also difficult for exact bit rate control, which is fatal for some applications.

Over the past several years, the wavelet transform has gained widespread acceptance in signal processing in general, and in image compression research in particular. In many applications wavelet-based schemes outperform other coding schemes like the DCT algorithm [44]. Since there is no need to block the input image and the basis functions of wavelet transform have variable length, wavelet coding schemes at higher compression avoid blocking artifacts. Wavelet-based coding is more robust under transmission and decoding errors, and also facilitates progressive transmission of images. In addition, they are better matched to the human visual system (HVS) characteristic [13][40].

5.2 Review of the Wavelet Transform

5.2.1 Definition of Wavelet Transform

Wavelets are functions defined over a finite interval and having an average value of zero. The basic idea of the wavelet transform is to represent any arbitrary function $g(t)$ as a superposition of a set of such wavelets or basis functions. These basis functions or baby wavelets are obtained from a single prototype wavelet called the mother wavelet, by dilations or contractions (scaling) and translations (shifts) [13][32]. The continuous-time wavelet transform of $g(t)$ with respect to wavelet $\psi(t)$ is defined as:

$$W(a, b) = \frac{1}{\sqrt{|a|}} \int_{-\infty}^{\infty} g(t) \psi^* \left(\frac{t-b}{a} \right) dt \quad (5.1)$$

where, $a \neq 0$ and b are real variables and $*$ denotes complex conjugation. The wavelet is defined as:

$$\psi_{ab}(t) = \frac{1}{\sqrt{a}} \psi \left(\frac{t-b}{a} \right) \quad (5.2)$$

The above equation represents a set of functions that are generated from a single function, $\psi(t)$, by dilations and translations. The variable b represents the time shift and the variable a corresponds to the amount of time-scaling or dilation. The function, $\psi(t)$, is referred to as the mother wavelet and it must satisfy two conditions [13]:

1. The functions integrates to zero:

$$\int_{-\infty}^{\infty} \psi(t) dt = 0 \quad (5.3)$$

2. The function is square integrable, or has finite energy:

$$\int_{-\infty}^{\infty} |\psi(t)|^2 dt < \infty \quad (5.4)$$

The inverse of the continuous wavelet transform is given by

$$g(t) = \frac{1}{C_\psi} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \frac{1}{|a|^2} W(a, b) \psi_{ab}(t) da db \quad (5.5)$$

It can be proved that the mother wavelet must satisfy the admissibility condition which shows that $\psi(t)$ has to oscillate and decay [40]:

$$C_\psi = \int_{-\infty}^{\infty} \frac{|\Psi(f)|^2}{f} df \quad (5.6)$$

For image coding, the wavelet is used to decompose the image data into wavelets. The wavelet transform can be viewed as the cross-correlation of the function $g(t)$ and the wavelets $\psi_{ab}(t)$. Therefore, the wavelet transform is equivalent to finding the output of a bank of bandpass filters specified by the wavelets $\psi_{ab}(t)$ as shown in Figure 5.1 [32]. This process decomposes the input signal into several subbands. Since each subband can be further partitioned, the filter bank implementation of the wavelet transform can be used for multiresolution analysis (MRA). Intuitively, when the analysis is viewed as a filter bank, the time resolution must increase with the central frequency of the analysis filters. This can be exactly obtained by the scaling property of the wavelet transform, where the center frequencies of the bandpass filters increase as the bandwidth become wider. Again, the bandwidth becomes wider by reducing the dilation parameter a .

Figure 5.1 The wavelet transform implemented with a bank of filters

Compared with Fourier transform, wavelet transforms have following advantages:

- *Localization*: Fourier basis functions are localized in frequency only. Wavelets are local in both frequency/scale (via dilations) and in time (via translations).
- *Sparse coding*: Function with discontinuities and functions with sharp spikes usually take substantially fewer wavelet basis functions than sine-cosine basis function to achieve a comparable approximation.
- *Low computational cost*: Number of operations for FWT is proportional to the number of input sequence sample n (i.e. $O(n)$), which is best possible. Recall that FFT requires $O(N \log_2 N)$ [29].

5.2.2 Discrete Wavelet Transform

For digital image compression, it is preferred to represent $g(t)$ as a discrete superposition sum rather than an integral. With this move to the discrete space, the dilation parameter a

takes the values $a = 2^k$ and the translation parameter b takes the values $b = 2^k l$, where both k and l are integers. From Equation 5.2, the discrete version of $\psi_{ab}(t)$ becomes [13]:

$$\psi_{kl}(t) = 2^{-\frac{k}{2}} \psi(2^{-k} t - l) \quad (5.7)$$

Its corresponding wavelet transform can be rewritten as:

$$W(k, l) = \int_{-\infty}^{\infty} g(t) \psi_{kl}^*(t) dt \quad (5.8)$$

and the inverse transform becomes:

$$g(t) = \sum_k \sum_l W_d(k, l) 2^{-\frac{k}{2}} \psi(2^{-k} t - l) \quad (5.9)$$

where $W_d(k, l) = W(k, l)/C_\psi$. The $W_d(k, l)$ represents the coefficients of the discrete wavelet transform of the function $g(t)$ [39]. If the discretization is further applied to the time domain by letting $t = mT$, where m is an integer and T is the sampling interval (without loss of generality, we assume $T = 1$), then the discrete-time wavelet transform is defined as:

$$W_d(k, l) = \sum_{m=-\infty}^{\infty} g(m) \psi_{kl}^*(m) \quad (5.10)$$

The sampling interval has to be chosen according to the Nyquist sampling theorem so that no information is lost in the process of sampling. The inverse discrete-time wavelet transform is:

$$g(m) = \sum_k \sum_l W_d(k, l) 2^{-\frac{k}{2}} \psi(2^{-k} m - l) \quad (5.11)$$

The discrete wavelet transform (DWT) of a finite length signal $x(n)$ having N components, for example, can be expressed by an $N \times N$ matrix [13].

5.3 The DWT for Image Compression

5.3.1 Basic Concept of Image Wavelet Transform Coding

In traditional image transform coding like Fourier transform or discrete cosine transform (DCT), one difficult problem is to choose the block size or window width so that statistics computed within that block provide good models of the image signal behavior. The choice of block size has to be compromised so that it can handle both active and smooth areas. In the active areas, the image data are more localized in the spatial domain, while in the smooth areas the image data are more localized in the frequency domain. With traditional transform coding, it is very hard to reach a good compromise. Whereas with the wavelet transform, it provides an elegant framework in which both statistical behaviors of image data can be analyzed with equal importance. The wavelet transform provides such a signal representation that some of the coefficients represent long data lags corresponding to a narrow band or low frequency range, and some of the coefficients represent short data lags corresponding to a wide band or high frequency range. Therefore, it is possible to obtain a good trade-off between spatial and frequency domain with the wavelet representation of image data [22][32].

There are several ways that the wavelet transforms can decompose a signal into various subbands. These include uniform decomposition, octave-band decomposition, and adaptive or wavelet-packet decomposition [39]. Out of these, octave-band decomposition is the most widely used. As introduced in Chapter 3, octave-band decomposition is a non-uniform band splitting method that decomposes the lower frequency part into narrower bands and the high-pass output at each level is left without any further decomposition. The discrete wavelet transform may be obtained by iterating a

2-channel filter bank on its lowpass output. This dyadic transform is illustrated in Figure 5.2.

In the course of wavelet transform, a low pass filter and a high pass filter are chosen such that they exactly halve the frequency range between themselves. This filter pair is called the Analysis Filter pair. First, the low pass filter is applied for each row of data, thereby getting the low frequency components of the row. But since the low-pass-filter is a half band filter, the output data contains frequencies only in the first half of the original frequency range. So, by Shannon's sampling theorem, they can be subsampled by two, so that the output data now contains only half the original number of samples. Now, the high pass filter is applied for the same row of data, and similarly the high pass components are separated, and placed by the side of the low pass components. This procedure is done for all rows.

Figure 5.2 Octave-based discrete wavelet transform

Next, the filtering is done for each column of the intermediate data. The resulting two-dimensional array of coefficients contains four bands of data, each labeled as LL (low-low), HL (high-low), LH (low-high) and HH (high-high). The low frequencies represent a

bandwidth corresponding to $0 < |\omega| < \pi/2$, while the high frequencies represent the band $\pi/2 < |\omega| < \pi$. The LL band can be decomposed once again in the same manner, thereby producing the next level of four subbands. The low frequencies of the second level decomposition correspond to $0 < |\omega| < \pi/4$, while the high frequencies at the second level correspond to $\pi/4 < |\omega| < \pi/2$. This decomposition can be done up to any level, thereby resulting in a pyramidal decomposition as shown in Figure. 5.3(b). Figure 5.3(a) shows the various subband images of a 2-level octave-band decomposed "Lena" image using the popular Daubechies-9/7 [7] biorthogonal wavelet filters.

Fig. 5.3. (a) Two level octave-band decomposition of Lena image, and
(b) Spectral decomposition and ordering.

In this thesis, we use the popular 9/7 biorthogonal wavelet transform, which was proposed by Daubechies et. al. [2][7]. The coefficients of the biorthogonal wavelet transform are as follows:

Table 5.1 The coefficients of the 9/7 biorthogonal wavelet transform

<i>filters</i>	a_{-4}	a_{-3}	a_{-2}	a_{-1}	a_0	a_1	a_2	a_3	a_4
H	0.037828	-0.023849	-0.110624	0.377402	0.852699	0.377402	-0.110624	-0.023849	0.037828
G		-0.064539	-0.040689	0.418092	0.788486	0.418092	-0.040689	-0.064539	

5.3.2 The Inverse DWT of An Image

Just as a forward transform can be used to separate the image data into various classes of importance, a reverse transform is used to reassemble the various classes of data into a reconstructed image as shown in Figure 5.2. A pair of high pass and low pass filters are used here also. This filter pair is called the Synthesis Filter pair. The filtering procedure is just the opposite - we start from the topmost level, apply the filters column-wise first and then row-wise, and proceed to the next level, until we reach the first level.

5.4 Advanced Wavelet Coding Schemes

The interplay between the three components of any image coder cannot be over-emphasized since a properly designed quantizer and entropy encoder are absolutely necessary along with optimum signal transformation to get the best possible compression. Many enhancements have been made to the standard quantizers and encoders to take advantage of how the wavelet transform works on an image, the properties of the HVS, and the statistics of transformed coefficients. Over the past few years, a variety of novel and sophisticated wavelet-based image coding schemes have been developed. These include EZW[31], SPIHT[30], SFQ[44][46], CREW[4], EPWIC[5], EBCOT[35], SR[37], Image Coding using Wavelet Packets[8], and Wavelet Image Coding using

VQ[32]. This list is by no means exhaustive and many more such innovative techniques are being developed. In this thesis, we will implement several typical algorithms -EZW and SPIHT, and compare their performance with that of the M -channel filter banks system discussed in Chapter 4.

5.4.1. Embedded Zerotree Wavelet (EZW) Algorithm

In octave-band wavelet decomposition, shown in Figure 5.4.(a), each coefficient in the high-pass bands of the wavelet transform has four coefficients corresponding to its spatial position in the octave band above in frequency. Because of this very structure of the decomposition, it probably needed a smarter way of encoding its coefficients to achieve better compression. Shapiro [31] called this structure 'zerotree' of wavelet coefficients, and presented his elegant algorithm for entropy encoding called 'Embedded Zerotree Wavelet' (EZW) algorithm in 1993. The zerotree is based on the hypothesis that if a wavelet coefficient at a coarse scale is insignificant with respect to a given threshold T , then all wavelet coefficients of the same orientation in the same spatial location at a finer scale are likely to be insignificant with respect to T . The idea is to define a tree of zero symbols which starts at a root which is also zero and labeled as end-of-block. Figure.5.4(a) and (b) shows a similar zerotree structure. Many insignificant coefficients at higher frequency subbands (finer resolutions) can be discarded, because the tree grows as powers of four. The EZW algorithm encodes the tree structure so obtained. This results in bits that are generated in order of importance, yielding a fully embedded code. The main advantage of this encoding is that the encoder can terminate the encoding at any point, thereby allowing a target bit rate to be met exactly. Similarly, the decoder can also stop

decoding at any point resulting in the image that would have been produced at the rate of the truncated bit stream. The algorithm produces excellent results without any pre-stored tables or codebooks, training, or prior knowledge of the image source.

Figure 5.4. (a) Structure of zerotrees, and (b) Scanning order of subbands for encoding

Zerotree coding is actually a combination of significance map coding and refinement procedure. Four symbols, *ZTR*, *POS*, *NEG* and *IZ* are used in the zero-tree. A *ZTR*, zerotree root, represents a wavelet coefficient and all of its descendants if they are insignificant, and it is the symbol that gets most of the compression. A *POS* or *NEG* symbol stands for a significant coefficient with a positive or negative sign. An *IZ* represents an insignificant coefficient with at least one significant descendent. It is this symbol that reduces the compression since more symbols are needed for encoding its descendants.

A pseudo-code of EZW algorithm is given as below:

0. **Zerotree coding** = significance map coding + refinement.
1. Set a threshold to $T = 2^{\lceil \log_2 m \rceil}$, where m is the maximum magnitude of the coefficients. Normally, m is equal to DC value.
2. **Significance map coding** = scan all the coefficients in a predefined way and output a symbol when the magnitude is larger than the threshold. The coefficient value is set to $1.5T$.
3. **Refinement** = refine significant coefficients by sending one more bit of its binary representation. The coefficient value is changed by $0.25T$.
4. Let $T = 0.5T$. Goto step 2.

5.4.2 Set Partitioning In Hierarchical Trees (SPIHT) Algorithm

The SPIHT-based algorithm, proposed by Said and Pearlman in 1996 [30], is an enhancement version of the EZW algorithms. This algorithm includes two major core techniques: the set partitioning sorting algorithm and the spatial orientation tree. The set partitioning sorting algorithm is the algorithm that hierarchically divides coefficients into significant and insignificant, from the most significant bit to the least significant bit, by decreasing the threshold value at each hierarchical step for constructing a significance map. At each threshold value, the coding process consists of two passes: the sorting pass and the refinement pass – except for the first threshold that has only the sorting pass. Let $c(i, j)$ represent the wavelet-transformed coefficients and m is an integer. The sorting pass involves selecting the coefficients such that $2^m \leq c(i, j) \leq 2^{m+1}$, with m being decreased at each pass. This process divides the coefficients into subsets and then tests each of these subsets for significant coefficients. The significance map constructed in the procedure is tree-encoded. The significant information is stored in three ordered lists: list of insignificant pixels (*LIP*), list of significant pixels (*LSP*), and list of insignificant sets

(*LIS*). At the end of each sorting pass, the *LSP* contains the coordinates of all significant coefficients with respect to the threshold at that step. The trees in the *LSP* can be one of two types: type *D* – check all descendants for significance, type *L* – check all of the descendants with the exception of the immediate children. The refinement pass involves transmitting the m^{th} -most significant bit of all the coefficients with respect to the threshold, 2^{m-i} .

The idea of a spatial orientation tree (SOT) is used to define the spatial relationship of the coefficients in the hierarchical structure. Three main concepts are proposed by Said and Pearlman [30] to adapt the spatial orientation tree to obtain better performance in image coding: (1) partial ordering of the transformed image by magnitude and transmission of coordinates via a subset partitioning algorithm, (2) ordered bit plane transmission of refinement bits, and (3) exploitation of the self-similarity of the coefficients across the various scales.

For the SOT, only two symbols, *zero* and *one*, are used, and each symbol has a different meaning at a different part of the tree. The symbol *one* may represent (1) a significant coefficient, (2) the negative sign of a significant coefficient, (3) the case that any one of the four children is significant, or (4) the case that any of the grandchildren is significant. The symbol *zero* could indicate (1) an insignificant coefficient, (2) the positive sign of a significant coefficient, (3) the case that all four children are insignificant, or (4) the case that all grandchildren are insignificant.

The implementation of the SPIHT algorithm consists of four steps: *initialization*, *sorting pass*, *refinement pass*, and *quantization scale update*. In the *initialization* step, we find an integer $m = \lfloor \log_2(\max_{(i,j)} \{|c(i,j)|\}) \rfloor$. Here $\lfloor \rfloor$ represent an operation of obtaining

the largest integer less than $|c(i, j)|$. The value of m is used for testing the significance of coefficients and constructing the significant map. The *LIP* is initialized as an empty list. The *LIS* is initialized to contain all the coefficients in the low-pass subbands that have descendants. These coefficients can be used as roots of spatial trees. All these coefficients are assigned to be type *D*. The *LIP* is initialized to contain all the coefficients in the low-pass subbands.

In the *sorting pass*, each entry of the *LIP* is tested for significance with respect to the threshold value 2^m . The significance map is transmitted in the following way. If it is significant, a "1" is transmitted, a sign bit of the coefficient is transmitted, and the coefficient coordinates are moved to the *LSP*. Otherwise, a "0" is transmitted. Then, each entry of the *LIS* is tested for finding the significant descendant. If there are none, a "0" is transmitted. If the entry has at least one significant descendant, then a "1" is transmitted and each of the immediate descendants are tested for significance. The significance map for the immediate descendants is transmitted in such a way that if it is significant, a "1" plus a sign bit are transmitted and the coefficient coordinates are appended to the *LSP*. If it is not significant, a "0" is transmitted and the coefficient coordinates are appended to the *LIP*. If the coefficient has more descendants, then it is moved to the end of the *LIS* as an entry of type *L*. If an entry in the *LIS* is of type *L*, then its descendants are tested for significance. If at least one of them is significant, then this entry is removed from the list, and its immediate descendants are appended to the end of the list of type *D*.

For the *refinement pass*, the m^{th} -most significant bit of the magnitude of each entry of the *LSP* is transmitted except those in the current sorting pass. For the *quantization scale update* step, m is decreased by 1 and the procedure is repeated from sorting pass.

A pseudo-code of SPIHT algorithm is given as below:

1. Compute the threshold, initialize:
 LIP to all root nodes coefficients,
 LIS to all trees (assign type D to them),
 LSP to an empty set.
2. Check the significance of all coefficients in LIP:
 If significant, output 1 followed by a sign bit and move it to the LSP.
 If not significant, output 0.
3. Check the significance of all trees in the LIS according to the type of tree type:
 For a tree of type D:
 If it is significant, output 1, and code its children:
 If a child is significant, output 1, then a sign bit and add it to the LSP
 If a child is insignificant, output 0, and add it to the end of LIP.
 If the children have descendants,
 move the tree to the end of LIS as type L,
 otherwise remove the tree from LIS.
 If it is insignificant output 0.
 For a tree of type L:
 If it is significant,
 output 1, add each of the children to the end of LIS as an entry of type D
 and remove the parent tree from LIS.
 If it is insignificant, output 0.
4. Let the threshold $T=0.5T$. Goto step 2.

5.4.3 Scalable Image Compression with EBCOT

The algorithm is based on independent Embedded Block Coding with Optimized Truncation (EBCOT) of the embedded bit-streams [35] which identifies some of the major contributions of the algorithm. The EBCOT algorithm is related in various degrees to much earlier work on scalable image compression. Noteworthy among its early predecessors are the EZW algorithm, SPIHT algorithm, and Taubman and Zakhor's LZC (Layered Zero Coding) algorithm. Like each of these, the EBCOT algorithm uses a

wavelet transform to generate the subband coefficients which are then quantized and coded.

Scalable compression refers to the generation of a bit-stream which contains embedded subsets, each of which represents an efficient compression of the original image at a reduced resolution or increased distortion. A key advantage of scalable compression is that the target bit-rate or reconstruction resolution need not be known at the time of compression. Another advantage of practical significance is that the image need not be compressed multiple times in order to achieve a target bit-rate, as is common with the existing JPEG compression standard. Rather than focusing on generating a single scalable bit-stream to represent the entire image, EBCOT partitions each subband into relatively small blocks of samples and generates a separate highly scalable bit-stream to represent each so-called code-block. The algorithm exhibits state-of-the-art compression performance while producing a bit-stream with an unprecedented feature set, including resolution and SNR scalability together with a random access property. The algorithm has modest complexity and is extremely well suited to applications involving remote browsing of large compressed images. The JPEG-2000 is actually based on EBCOT algorithm [21].

5.5 Results and Analysis

5.5.1. Coding and Implementation

An image compression encoder/decoder program – Wavekit has been developed. Wavekit has been developed mainly following EZW principles with some simplification. A simple scalar quantization scheme has been used. Entropy encoding is an arithmetic

encoder. The popular Daubechies 9/7 biorthogonal wavelet transform has been used. It includes Encoder and Decoder packages.

The Encoder program includes the following steps in compressing an image,

1. Specifying the Rate (bits available) and Distortion (tolerable error) parameters for the target image.
2. Specifying the level of wavelet transform, which can transform the image data into a wavelet-tree structure. In every level, the coefficients are divided into four classes, based on the property of the coefficients.
3. Allocating the available bit budget among these classes, such that the distortion is a minimum. This can be controlled by Rate-distortion criterion.
4. Quantize each class separately using the bit allocation information derived in step 3.
5. Encode each class separately using an entropy coder and write to the file.

Reconstructing the image from the compressed data is usually a faster process than compression. The Decoder program includes following steps,

1. Read in the quantized data from the file, using an entropy decoder. (reverse of step 5).
2. Dequantize the data. (reverse of step 4).
3. Rebuild the image. (reverse of step 2).

Wavekit has been developed following the above procedures. The coder is not the most sophisticated. New functions and a more sophisticated coder will be added in the future.

Source code documentation can be found in Appendix B.

Included in the Wavekit packages are routines for

1. Classification of image data, using the 2-D DWT. Daubechies - 7/9 biorthogonal wavelet filters [7] are used;

2. Optimal bit allocation for the various classes of image data, using Rate-Distortion optimization methods. The module is an implementation of the paper [28].
3. Quantization of the image data using a uniform quantizer.
4. Entropy coding of the quantized data using an arithmetic coder, described in [42].

The data structure for maintaining the cumulative frequency tables used in the above described algorithm is an implementation of the paper [11].

5.5.2. Results

The peak signal to noise ratios (PSNR) of several compression techniques applied to the $512 \times 512 \times 8$ -bpp Lena image and Barbara image are compared in Table 5.2 and 5.3 as well as Figure 5.5 and 5.6. The result of the EZW algorithm is from literature [31] while the result of the SPIHT algorithm is obtained by executing the coder provided by the original authors (ftp://ipl.rpi.edu/pub/ew_code). The JPEG coder is from the Independent JPEG Group (<http://www.iijg.org>), version 6b. It is seen that, at compression ratios less than 20:1 or so, the JPEG performs better numerically than the simple Wavekit coder. At compression ratios above 30:1, the performance of the JPEG rapidly deteriorates, while the Wavekit coder degrades gracefully well beyond ratios of 100:1. The graphs also show that both the encoding technique and particular wavelet used can make a significant difference in the performance of a compression system: The SPIHT coder performs the best; the performance of the Wavekit is comparable to Shapiro's EZW coder. Wavekit performs better than JPEG at high compression ratio. Figure 5.7 and 5.8 are reconstructed image of Lena and Barbara with different coding algorithms at bit rate 0.25 bpp, respectively.

Table 5.2 Results of Different Coding Methods (512 x 512, 8-bpp grayscale 'Lena' image)

Coding Methods	0.0625bpp	0.125bpp	0.25 bpp	0.50 bpp	0.75 bpp	1.00 bpp
Baseline – JPEG*	15.16	24.40	31.60	34.90	36.91	38.60
Wavekit	25.91	28.67	32.11	34.91	36.87	38.31
EZW ^[31]	26.56	30.43	33.17	36.28	-	39.55
SPIHT **	28.38	31.10	34.11	37.21	39.04	40.40

note: * results are obtained by executing the JPEG coder from <http://www.iig.org>

** results are obtained by executing the SPIHT executable coder from ftp://ipl.rpi.edu/pub/ew_code

Table 5.3 Results of Different Coding Methods (512 x 512, 8-bpp grayscale 'Barbara' image)

Coding Methods	0.0625bpp	0.125bpp	0.25 bpp	0.50 bpp	0.75 bpp	1.00 bpp
Baseline – JPEG*	11.93	18.40	25.17	28.30	31.92	34.90
Wavekit	20.68	22.79	26.24	28.91	31.74	34.61
EZW ^[31]	22.10	24.03	26.77	30.53	-	35.14
SPIHT **	23.35	24.86	27.58	31.39	33.90	36.41

note: * results are obtained by executing the JPEG coder from <http://www.iig.org>

** results are obtained by executing the SPIHT executable coder from ftp://ipl.rpi.edu/pub/ew_code

Figure 5.5 Comparative Evaluation of Coding Results for Image "Lena"

Figure 5.6 Comparative Evaluation of Coding Results for Image "Barbara"

5.5.3 Discussion

Through implementing several wavelet-based image coding algorithms, some findings can be discussed as following:

- ❖ DCT-based image coder performs well at low to moderate bit rates, at higher compression ratios, the reconstructed image quality degrades rapidly.
- ❖ Wavelet-based coding scheme provides substantial improvement in picture quality at low bit rates. It outperforms the DCT-based image coder. However, the wavelet algorithm has a longer coding time.
- ❖ EZW and SPIHT algorithms take advantage of the statistics of the wavelet coefficients and give high SPNR values. The elaborate quantizer is a key point for their high performance.
- ❖ Wavelet-based coders facilitate progressive transmission of images. They can control bit rate exactly. They are more flexible than the block transform based coders.
- ❖ The performance of the Wavekit package is inferior to that of the EZW algorithm by an average of 0.5 dB. This may be attributed to the sub-elegant design of the quantizer and the entropy encoder. Further improvements shall be necessary in the future.

(a) Original 512x512 "Lena",
Image at 8 bpp,

(b) Wavelet compression, 0.25 bpp,
32:1, PSNR=32.11 dB,

(c) EZW compression, 0.25 bpp,
32:1, PSNR=33.17 dB.

(d) SPIHT compression, 0.25 bpp,
32:1, PSNR=34.11 dB.

Figure 5.7 Performance of Different Wavelet-based Coders operating on "Lena"

(a) Original 512x512 "Barbara"
Image at 8 bpp.

(b) Wavelet compression, 0.25 bpp, 32:1,
PSNR=26.24 dB.

(c) EZW compression, 0.25 bpp,
32:1. PSNR=26.77 dB.

(d) SPIHT compression, 0.25 bpp,
32:1, PSNR=27.58 dB.

Figure 5.8 Performance of Different Wavelet-based Coders operating on "Barbara"

Chapter 6

Conclusions and Future Research

This thesis has studied the design and implementation of multirate filter bank systems and wavelet transforms as well as their applications in image compression. In summary, the first part of the thesis focused mainly on the multirate blocking transform, while the second part considered the octave-based embedded wavelet image compression systems. In accordance with this logical partitioning, the contributions made by this thesis can be categorized as either pertaining to bridging two image coding systems or performance evaluation. These contributions along with some directions for the future research are summarized in the following sections.

6.1 Summary

Block transform coding and subband coding are two dominant techniques in existing image compression standards and implementations. In this thesis, we have carried out a comparative study of block transform based filter bank coding and wavelet-based subband coding for image compression. Based on the performance results, the following conclusions have been obtained for block transform based filter bank coding:

- ❖ Block transform coders have been used successfully due to their low complexity in implementation and reasonable performance. The current image compression standard, JPEG, which utilizes 8×8 DCT at its transformation stage, is the most popular block transform coder. DCT-based image coders perform very well at high bit rates, but at low bit rates, annoying blocking artifact shows up because of two reasons: *i.* the DCT bases are short, non-overlapped, and have discontinuities at the ends; *ii.* JPEG processes each image independently without accounting for the inter-block correlation.
- ❖ To overcome the deficiencies of DCT-based imaging coders, the lapped transform has been proposed. The lapped transform is a block transform with overlapping basis function. It produces the transform coefficients of the current block by borrowing pixels from the adjacent blocks, which takes into account inter-block correlation and leads to more efficient entropy coding of the coefficients. Its overlapping bases decaying smoothly to zero at the ends can suppress blocking discontinuities effectively.
- ❖ General lapped biorthogonal transform (GLBT) used in this thesis is an 8-channel uniform-band lapped transform with all 16-tap filters. The GLBT algorithm provides a compromise solution with moderate complexity and efficiency. It can be used to replace the DCT algorithm in some cases.
- ❖ The lapped transforms facilitate hardware implementation with fast, efficient lattice structures which employ a minimal number of delay elements and are robust under the quantization of lattice coefficients. The block-based nature of the transform also increases the parallelism of the computations.

Subband coding is another technique used for image coding. Those methods are mainly wavelet-based algorithms. Wavelet-based algorithms have the following characteristics:

- ❖ The wavelet-based algorithms have been employed by latest image compression standards (JPEG2000 and MPEG-4) thanks to their outstanding compression efficiency and good visual quality. Their inherent multiresolution nature facilitates progressive transmission of the images. This allows for variable bit rates.
- ❖ The wavelet-based algorithms have other attractive characteristics, such as embedded quantization and exact bit rate control. Precise rate control is a distinct advantage. The user can choose a bit rate and encode the image to exactly the desired bit rate.
- ❖ The wavelet-based algorithms provide high subjective and objective performance and outplay the general block transform schemes in terms of their higher energy compaction, progressive transmission, and bit rate control, but the longer coding time also limits their real-time application.
- ❖ Advanced wavelet coding schemes such as the SPIHT and the EZW take into account the advantages of the statistics of the wavelet coefficients and produce high PSNR values. According to our experiments, we found that the SPIHT algorithm provides the highest PSNR value for image coding. This would be due to its elaborate design of the quantizer and entropy coder.
- ❖ Compared with block transforms, wavelet-based transforms have longer coding times, are difficult for hardware implementation and are sensitive to image with complex textures.

6.2 Future Research

The work in this thesis can be extended in a few directions. Here are some:

In this study, we evaluate the performance of different image coding algorithms, solely based on the resulting PSNR. While PSNR is a general performance measure that is widely accepted in the literature, there might be other performance measures that are suitable for specific applications. Further investigation will be needed to examine the performance of the imaging coders with respect to those performance measures.

One limitation of the current study is that we consider lossy compression only. The performance of different filter bank algorithms for lossless compression (integer to integer transform) is worth while to evaluate in the future.

In this study, we did not consider the special features of images. Our results have shown that the performance of coding algorithm is image dependent. The coding algorithm should be chosen adaptively depending on the statistical nature of image being coded. It is highly likely to improve compression performance by obtaining accurate models of images with feature extraction techniques and selecting suitable imaging coder.

Finally, there are many factors to influence the performance of an image coding system, especially the effects of quantization error and entropy coding. Further studies of these techniques may help us to improve the overall performance of an image coding system.

References

- [1] Ackenhusen, J. G., *Real-time Signal Processing: Design and Implementation of Signal Processing System*, Prentice Hall PTR, Upper Saddle River, New Jersey 07458, 1999.
- [2] Antonini, M. Barlaud, M. Mathieu, P. and Daubechies, I. "Image coding Using Wavelet Transform," *IEEE Trans. on Image Processing*, Vol.1, No.2, pp 205-220, April 1992.
- [3] Balasingham, I. et. al., "Performance Evaluation of Different Filter Banks in the JPEG-2000 Baseline System," *ICIP'98 IEEE Intl. Conf. on Image Processing*, Chicago, Ill. USA 1998.
- [4] Boliek, M., Gormish, M. J., Schwartz, E. L., and Keith, A. "Next Generation Image Compression and Manipulation Using CREW," *Proc. IEEE ICIP*, 1997, <http://www.crc.ricoh.com/CREW>.
- [5] Buccigrossi, R., and Simoncelli, E. P. "EPWIC: Embedded Predictive Wavelet Image Coder," *GRASP Laboratory, TR #414*, <http://www.cis.upenn.edu/~butch/EPWIC/index.html>.
- [6] Chan, S. C., et. al., "A class of M-Channel Linear-Phase Biorthogonal Fliter Banks and Their Applications to Subband Coding", *IEEE Trans. on Signal Processing*, Vol. 47, No.2, pp. 564-571, Feb. 1999.

- [7] Cohen, A., Daubechies, I., and Feauveau, J. C. "Biorthogonal Bases of Compactly Supported Wavelets," *Comm. on Pure and Applied Mathematics*, Vol. XLV, pp. 485-560, 1992.
- [8] Coifman, R. R. and Wickerhauser, M. V. "Entropy Based Algorithms for Best Basis Selection", *IEEE Trans. Information Theory*, Vol. 38, No. 2, pp. 713-718, Mar. 1992.
- [9] de Queiroz, R. L., and Tran, T. D., "Lapped Transforms for Image Compression", *The Handbook on Transforms and Data Compression*, edited by Rao, K. R. and Yip, P. CRC Press, pp. 195-265, Oct. 2000.
- [10] de Queiroz, R. L., Nguyen, T. Q., and Rao, K. R., "The GenLOT: General Linear-Phase Lapped Orthogonal Transform," *IEEE Trans. on Signal Processing*, Vol. 44, Vol. 5, pp. 497-507, Mar. 1996.
- [11] Fenwick, P. M. "A New Data Structure for Cumulative Frequency Tables", *Software-Practice and Experience*, Vol.24, No.3, pp. 327-336, March 1994.
- [12] Fliege, N. J., *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*, John Wiley, Chichester, England, 1994.
- [13] Goswami, J. C. and Chan, A. K., *Fundamentals of Wavelets*, John Wiley & Sons. Inc. New York, 1999.
- [14] Hilton, M. L., Jawerth, B. D., and Sengupta, A., "Compressing Still and Moving Images with Wavelets," *Multimedia Systems*, Vol. 2, No. 3, April, 1994, <ftp://ftp.math.scarcolina.edu/pub/wavelet/papers/varia/tutorial/tutorial.ps.Z>.

- [15] ISO/IEC/JTC1/SC29/WG1 N390R, JPEG 2000 Image Coding System, Mar. 1997, <http://www.jpeq.org/public/wg1n505.pdf>.
- [16] Jain, A. K., *Fundamentals of Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ 07632, USA, 1989.
- [17] Karp, T. and Fliege, N. J., "Modified DFT Filter Banks with Perfect Reconstruction", *IEEE Trans. on Circuits and Systems -II*, Vol. 46, No. 11, pp. 1404-1414, Nov. 1999.
- [18] Lewis. A. S. and Knowles, G., "Image Compression Using the 2-D Wavelet Transform." *IEEE Trans. Image Processing*, Vol. 1, No. 2, pp. 244-250, April 1992.
- [19] Lin. L. J. and Ortega, A., "Bit-Rate Control Using Piecewise Approximated Rate-Distortion Characteristics", *IEEE Trans. on Circuits and Systems for Video Technology*, Vol. 8, No. 4, pp. 446-459, Aug. 1998.
- [20] Malvar. H. S., *Signal Processing with Lapped Transforms*, Norwood, MA, Artech House. 1992.
- [21] Marcellin. M.W. et. al. "An Overview of JPEG-2000," *Proc. of IEEE Data Compression Conference*, pp. 523-541, 2000.
- [22] Mallat. S. G. "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation." *IEEE Trans. PAMI*, Vol. 11, No. 7, pp. 674-693, July 1989.
- [23] Nelson. M. "Arithmetic Coding+Statistical Modelling=Data Compression", *Dr. Dobb's Journal*, February, 1991, <http://dogma.net/markn/articles/arith/part1.htm>

- [24] Nguyen, T. Q., "A Tutorial on Filter Banks and Wavelets," presented at the *Intl. Conf. on Digital Signal Processing*, Cypress, June 1995.
- [25] Ortega, A. "Optimal rate allocation under multiple rate constraints", *Data Compression Conference'96*, Snowbird, UT, March 1996
- [26] Pennebaker, W. B. and Mitchell, J. L. *JPEG - Still Image Data Compression Standards*, Van Nostrand Reinhold, 1993.
- [27] Ramstad, T. A., Aase, S. O., and Husoy, J. H., *Subband Compression of Images – Principles and Examples*, Elsevier Science Publishers BV, North Holland, 1995.
- [28] Riskin, E. A. "Optimum bit allocation via the generalized BFOS algorithm," *IEEE Trans. on Information Theory*, Vol. 37, No.3, pp. 400-402, Mar. 1991.
- [29] Rao, K.R. and Yip, P. *Discrete Cosine Transforms – Algorithms, Advantages, Applications*, Academic Press, 1990.
- [30] Said, A. and Pearlman, W. A. "A New, Fast, and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees," *IEEE Trans. CSVT*, Vol. 6, No. 3, pp. 243-250, June 1996. ftp://ipl.rpi.edu/pub/EW_Code/SPIHT.ps.gz.
- [31] Shapiro, J. M. "Embedded Image Coding Using Zerotrees of Wavelet Coefficients", *IEEE Trans. Signal Processing*, Vol. 41, No. 12, pp. 3445-3462, Dec. 1993,
- [32] Shi, Y. Q. and Sun, H. *Image and Video Compression for Multimedia Engineering*, CRC Press, Boca Raton, Florida 33431, 1999.

- [33] Strang, G. and Nguyen, T. Q., *Wavelets and Filter Banks*, Wellesley-Cambridge Press, Wellesley, MA, 1996.
- [34] Suter, B.W., *Multirate and Wavelet Signal Processing*, Academic Press, San Diego, CA 92101, 1998.
- [35] Taubman, D. "High Performance Scalable Image Compression with EBCOT", submitted to *IEEE Trans. Image Processing*, Mar. 1999.
<http://maestro.ee.unsw.edu.au/~taubman/activities/preprints/ebcot.zip>.
- [36] Tran, T. D., de Queiroz, R. L., and Nguyen, T. Q., "Linear Phase Perfect Reconstruction Filter Bank: Lattice Structure, Design, and Application in Image Coding," *IEEE Trans. Signal Processing*, vol. 48, no. 1, Jan. 2000, pp. 133-147.
- [37] Tsai, M. J., Villasenor, J. D., and Chen, F. "Stack-Run Image Coding", *IEEE Trans. CSVT*. Vol. 6, No. 5, pp. 519-521, Oct. 1996,
http://www.icsl.ucla.edu/~ipl/papers/stack_run.html.
- [38] Vaidyanathan, P. P. *Multirate Systems and Filter Banks*, Prentice Hall, Englewood Cliffs, N.J., 1993.
- [39] Vetterli, M. and Herley, C. "Wavelets and Filter Banks: Theory and Design". *IEEE Trans. Signal Processing*, Vol. 40, No. 9, pp. 2207-2232, Sep. 1992,
- [40] Vetterli, M. and Kovacevic, J. *Wavelets and Subband Coding*, Englewood Cliffs, NJ, Prentice Hall, 1995.
- [41] Wallace, G. K. "The JPEG Still Picture Compression Standard," *Communications of the ACM*, Vol. 34, No. 4, pp. 30-44, April 1991,

- [42] Witten, H., Neal, M. and Cleary, C., "Arithmetic Coding For Data Compression", *Communications of the ACM*, Vol. 30, No. 6, Pages 520-540, June 1987.
- [43] Woods, J. W. and O'Neil, S. D. "Subband Coding of Images," *IEEE Trans. ASSP*. Vol.34, No.5, pp.1278-1288, October 1986.
- [44] Xiong, Z., Ramachandran, K. and Orchard, M. T. "Space-Frequency Quantization for Wavelet Image Coding," *IEEE Trans. Image Processing*, Vol. 6, No.5, pp.677-693, May 1997.
http://troi.ifp.uiuc.edu/~kannan/my_papers/xiong_ramchandran_orchard_ip97.ps.gz.
- [45] Xiong, Z., Ramachandran, K. Orchard, M. T., and Zhang, Y. Q., "A Comparative Study of DCT- and Wavelet-Based Image Coding," *IEEE Trans. Circuits and Systems for Video Technology*. Vol. 9, No. 5, pp. 692-695, Aug. 1999.
- [46] Xiong, Z. and Wu, X. "Wavelet Image Coding Using Trellis Coded Space-Frequency Quantization," *IEEE Signal Processing*, Vol.6, No.7, pp.158-161, July 1999.
- [47] Sayood, K. *Introduction to Data Compression*, Second Edition. Morgan Kaufman Publishers, San Francisco, 2000.

Appendix

A. Proof

B. Test Images

C. Source Codes

Appendix A

Proof

A.1 Proof of General PR Condition

Consider Fig. a-1 which is the equivalent circuit of Fig.3.13 for the QMF bank in terms of $\mathbf{P}(z)$. We will express $Y(z)$ in terms of $X(z)$ and the elements $P_{s,l}(z)$ of $\mathbf{P}(z)$.

Figure A1 The equivalent circuit for QMF filter bank

The input and output of $\mathbf{P}(z)$ are given by

$$C_l(z) = \frac{1}{M} \sum_{k=0}^{M-1} (z^{\frac{1}{M}} W^k)^{-l} X(z^{\frac{1}{M}} W^k), \quad 0 \leq l \leq M-1 \quad (\text{A.1})$$

$$B_s(z) = \sum_{k=0}^{M-1} P_{s,l}(z) C_l(z) \quad (\text{A.2})$$

Where $W = e^{j2\pi/M}$. The reconstructed signal is

$$\begin{aligned} Y(z) &= \sum_{s=0}^{M-1} z^{-(M-1-s)} B_s(z^M) \\ &= \sum_{s=0}^{M-1} z^{-(M-1-s)} \sum_{l=0}^{M-1} P_{s,l}(z^M) C_l(z^M) \\ &= \frac{1}{M} \sum_{s=0}^{M-1} z^{-(M-1-s)} \sum_{l=0}^{M-1} P_{s,l}(z^M) \sum_{k=0}^{M-1} (z W^k)^{-l} X(z W^k) \end{aligned} \quad (\text{A.3})$$

This can be rearranged as

$$Y(z) = \frac{1}{M} \sum_{s=0}^{M-1} X(zW^s) \sum_{l=0}^{M-1} W^{-sl} \sum_{k=0}^{M-1} z^{-l} z^{-(M-1-s)} P_{s,l}(z^M) \quad (\text{A.4})$$

The terms of the form $X(zW^k)$, $k \neq 0$ represent aliasing. The above expression is free from these aliasing terms if and only if the elements of $\mathbf{P}(z)$ are zero except the 0^{th} row elements. By definition, $Y(z) = T(z)X(z)$ with $T(z)$ obtained from Equation (A.4) as

$$T(z) = \frac{1}{M} \sum_{s=0}^{M-1} z^{-(M-1-s)} \sum_{l=0}^{M-1} z^{-l} P_{s,l}(z^M) \quad (\text{A.5})$$

Since the elements $P_{s,l}(z)$ are completed by the 0^{th} row elements $P_{0,l}(z)$, the above equation can be rearranged into following form

$$T(z) = z^{-(M-1)} \sum_{m=0}^{M-1} z^{-m} P_m(z^M) \quad (\text{A.6})$$

A PR system is an alias free system with $T(z) = \text{delay}$. This is satisfied only if $P_m(z) = 0$ for all but one value of m in the range $0 \leq m \leq M-1$. And this nonzero $P_m(z)$ must have the form cz^{-m_0} . Summarizing, an alias free system has perfect reconstruction if and only if $\mathbf{P}(z)$ has 0^{th} row equal to

$$[0 \ \dots \ 0 \ \ c z^{-m_0} \ 0 \ \dots \ 0] \quad (\text{A.7})$$

In other words $\mathbf{P}(z)$ has the form

$$\mathbf{P}(z) = \mathbf{R}(z)\mathbf{E}(z) = cz^{-m_0} \begin{bmatrix} \mathbf{0} & \mathbf{I}_{M-r} \\ z^{-1}\mathbf{I}_r & \mathbf{0} \end{bmatrix}, \quad (\text{A.8})$$

for some integer r with $0 \leq r \leq M-1$, some integer m_0 , and some constant $c \neq 0$. This is the proof of the general PR condition stated in Section 3.5.2. Under this condition, the reconstructed signal is $y[n] = cx[n-n_0]$, where Equation (A.6) reduces to

$$T(z) = cz^{-r} z^{-(M-1)} z^{-m_0 M} \quad (\text{A.9})$$

◆

Appendix B

Test Images

B.1 Overview

The test images used in this thesis are standard $512 \times 512 \times 8$ bpp grayscale images 'Lena' and 'Barbara', which were taken from the ISO JPEG test set and other databases. The original images are in *raw* format. A simple MATLAB *M*-file has been developed to show them.

Image 'Lena' has mainly smooth areas with occasional edges and textures, while image 'Barbara' has large detailed areas with high-frequency patterns. 'Lena' is an ideal testing image to compare the general performance of different coding algorithms. 'Barbara' is more suitable for evaluating the functions of different coding algorithms to deal with the highly- textured image.

Appendix C

Source Codes

C.1 Overview

The image codec software consists of the following routines:

- I. Wavekit
- II. Image Processing
- III. Quantizer
- IV. Entropy Coder
- V. Wavelet Transform
- VI. Lapped Transform

The software is developed with Visual C++ 5.0. Implementation environment is on a PC with 550 MHz Pentium III™ processor, 128 MB of RAM under Windows NT.

Some functions are taken from the 'Wavelet Image compression Kit' written and distributed by Goeff Davis <http://www.cs.dartmouth.edu/~davis/wavelet/wavelet.html>.

The quantization, bit allocation and entropy coding are very general-purpose. They will work with a whole variety of transforms, including DCT's, LOT's and Wavelet. BitAllocation is done using a modified version of the algorithm given in "Optimal Bit Allocation via the General BFOS algorithm" by Eve A. Riskin [28]. Entropy coding is an implementation of the algorithm given in "Arithmetic coding for Data compression" by Witten, Neal & Cleary[42]. The implementation is courtesy of John Danskin, <http://www.cs.dartmouth.edu/~jmd>. The source model is based on the algorithm in "A New Data structure for Cumulative Frequency Tables" by Peter M. Fenwick[11].

Lapped transform is done using a modified version of the algorithm given in *Discrete Cosine Transform* by K.R.Rao and P.Yip [29]. The original program is developed by Henrique S. Malvar [20]. Some modification has been done by referring to the 'Uicoder' developed by Nguyen et al. <http://saigon.ece.wisc.edu/uicoder>.

C.2 Source Codes

I. Wavekit Part

```

/*-----*/
//      Wavekit - Image Compression Package version 1.0
//      written by      Shixin Wei
//      Copyright© 2001 S.Wei. Permission is granted to use for research purposes only
/*-----*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "Image.h"
#include "Wavelet.h"
#include "aricoder.h"
#include "quantizer.h"

#define NSTEPS 5                                //changeable based on requirements
#define NBANDS (NSTEPS*3+1)

#define sqrt2 1.414213
#define MAXREAL 1e+100

//Haar filter coefficients
double HarrCoeffs[] = {1.0/sqrt2, 1.0/sqrt2};

//-----Daubechies 9/7 filters coefficients-----//
double AntoniniAnalysis[] = { 3.782845550699535e-02,
                             -2.384946501937986e-02,
                             -1.106244044184226e-01,
                             3.774028556126536e-01,
                             8.526986790094022e-01,
                             3.774028556126537e-01,
                             -1.106244044184226e-01,
                             -2.384946501937986e-02,
                             3.782845550699535e-02 };
double AntoniniSynthesis [] = { -6.453888262893856e-02,
                                -4.068941760955867e-02,
                                4.180922732222124e-01,
                                7.884856164056651e-01,
                                4.180922732222124e-01,
                                -4.068941760955867e-02,
                                -6.453888262893856e-02 };
//-----//

struct stBlock
{
    int x, y;
    int width, height;
    int ncoeffs, nonzerocoeffs;
    unsigned int *symbols;
    unsigned int zeroTreeSymbol;
    double *coeffs;
    double maxcoeff, mincoeff, meancoeff;
    UniformQuant *uq;
};

//-----//
Filter fHarrAnalysis(2, 0, HarrCoeffs);
Wavelet wHarr(&fHarrAnalysis);

//-----//
Filter fAntoniniAnalysis(9, -4, AntoniniAnalysis);
Filter fAntoniniSynthesis(7, -3, AntoniniSynthesis);
Wavelet wAntonini(&fAntoniniAnalysis, &fAntoniniSynthesis);

double nBitsForZero =0;

```



```

//-----//
void AllocSubBands(Image &image, struct stBlock *subband, int nsteps)
{
    int prevlWidth = image.width;
    int prevlHeight = image.height;
    int i;

    for(i=nsteps-1; i>=0; i--)
    {
        int llWidth = (prevlWidth+1)/2;
        int llHeight = (prevlHeight+1)/2;
        int hhWidth = prevlWidth - llWidth;
        int hhHeight = prevlHeight - llHeight;

        // top right block
        subband[i*3+1].x = llWidth;
        subband[i*3+1].y = 0;
        subband[i*3+1].width = hhWidth;
        subband[i*3+1].height = llHeight;

        // bottom left block
        subband[i*3+2].x = 0;
        subband[i*3+2].y = llHeight;
        subband[i*3+2].width = llWidth;
        subband[i*3+2].height = hhHeight;

        // bottom right block
        subband[i*3+3].x = llWidth;
        subband[i*3+3].y = llHeight;
        subband[i*3+3].width = hhWidth;
        subband[i*3+3].height = hhHeight;

        prevlWidth = llWidth;
        prevlHeight = llHeight;
    }
    // LL coefficient band.
    subband[0].x = 0;
    subband[0].y = 0;
    subband[0].width = prevlWidth;
    subband[0].height = prevlHeight;

    // alloc mem & copy the coefficients to individual arrays.
    for(i=0; i<NBANDS; i++)
    {
        subband[i].ncoeffs = subband[i].width * subband[i].height;
        subband[i].coeffs = new double[ subband[i].ncoeffs ];
        subband[i].symbols = new unsigned int[ subband[i].ncoeffs ];
        assert(subband[i].coeffs && subband[i].symbols);
        subband[i].uq = NULL;
    }
}

//-----//
void FreeSubBands(struct stBlock *subband, int nbands)
{
    for(int i=0; i<nbands; i++)
    {
        delete[] subband[i].coeffs;
        delete[] subband[i].symbols;
    }
}

//-----//
void InitSubBands(Image &image, struct stBlock *subband)
{
    int i,j;

    // copy the data to the indiv. subbands find the max & min values in each subband.
    for(i=0; i<NBANDS; i++)
    {

```

```

image.GetData(subband[i].x, subband[i].y, subband[i].width, subband[i].height,
subband[i].coeffs);

// find the max & min in the coefficient array.
double maxcoeff = -MAXREAL;
double mincoeff = MAXREAL;
double *coeffs = subband[i].coeffs;
subband[i].nonzerocoeffs = 0;
for(j=subband[i].ncoeffs; j>0; j--)
{
    if (maxcoeff < *coeffs) maxcoeff = *coeffs;
    else if (mincoeff > *coeffs) mincoeff = *coeffs;
    if (*coeffs != 0.0) subband[i].nonzerocoeffs++;
    coeffs++;
}
subband[i].maxcoeff = maxcoeff;
subband[i].mincoeff = mincoeff;
// except for lowpass subband, assume the pdfs are centered around zero.
if (i > 0)
{
    if (fabs(maxcoeff) > fabs(mincoeff))
        subband[i].mincoeff = -maxcoeff;
    else
        subband[i].maxcoeff = -mincoeff;
}
subband[i].meancoeff = (subband[i].maxcoeff+subband[i].mincoeff)/2;
}
}

//-----//
void CalcRateDistortion(struct stBlock &subband, int nBits, double &Rate, double &Distortion, ArithEncoder &arienc)
{
    int nLevels = (1<<nBits);

    // initialize the quantizer.
    UniformQuant uq(subband.maxcoeff, subband.mincoeff, nLevels);

    // calculate the rate & distortion.
    double *coeffs = subband.coeffs;
    Distortion = 0.0;
    Rate = 0.0;

    for(int ncoeffs = subband.ncoeffs; ncoeffs>0; ncoeffs--)
    {
        unsigned int symbol = uq.Symbol(*coeffs);
        double recons = uq.Value(symbol);

        Distortion += pow((*coeffs - recons),2);
        coeffs++;
        if (nBits)
        {
            double tRate = arienc.EncodingCost(symbol);
            //double tRate=nBits;
            if (*coeffs==0.0) nBitsForZero += tRate;
            Rate += tRate;
        }
    }
    Distortion /= subband.ncoeffs;
    Rate /= subband.ncoeffs;
}

//-----//
void CalcBitAllocation(struct stBlock *subband, int *B, int M, int q, double TargetRate)
{
    // calculate the bitallocation for each subband/class.
    double *p; // the probability of each class
    double *r; // the rate for each class for a given no of bits.
    double *d; // the distortion for each class for a given no. of bits.
    int *allow;
    int i,j;

```

```

p = new double[M];
r = new double[M * (q+1)]; // 2-d array
d = new double[M * (q+1)]; // 2-d array
allow = new int[M];

// initialize all p(i)'s
double totcoeffs = 0;
for(i=0; i<M; i++)
    totcoeffs += subband[i].ncoeffs;
for(i=0; i<M; i++)
    p[i] = ((double)subband[i].ncoeffs)/totcoeffs;

// first calc all di(j)'s
for(i=0; i<M; i++)
{
    printf("For class %d\n", i);
    for(j=0; j<=q; j++)
    {
        // Initialize the arithmetic coder with a dummy bitstream.
        ArithEncoder arienc(NULL);
        arienc.SetNumSyms((1<=j));
        arienc.InitEncode();

        double tRate, tDist;

        CalcRateDistortion(subband[i], j, tRate, tDist, arienc);
        r[i*(q+1)+j] = tRate;
        d[i*(q+1)+j] = tDist;
    }
}

// initialise all B(i)'s
for(i=0; i<M; i++)
    B[i] = q;

double D; // overall average distortion.
double R; // overall average rate.
double *lowestSi = new double[M]; // lowest Si in each class.
int *lowestSiClass_n = new int[M]; // the corresponding n.
double TargetRateInBytes = TargetRate * totcoeffs; // assuming 1 pixel = 1 byte in src.
for(i=0; i<M; i++) allow[i]=1;

do
{
    // find all Si's and the class(es) for which Si is the lowest.
    double lowestSiInAll = MAXREAL;
    for(i=0; i<M; i++)
        lowestSi[i] = MAXREAL;
    for(i=0; i<M; i++)
    {
        for(int n=1; n<=B[i]; n++)
        {
            double Si = (d[i*(q+1) + B[i]-n] - d[i*(q+1) + B[i]])/(double)n;
            if(Si <= lowestSi[i] && allow[i])
            {
                lowestSiClass_n[i] = n;
                lowestSi[i] = Si;
                if (Si <= lowestSiInAll)
                    lowestSiInAll = Si;
            }
        }
    }
    if (lowestSiInAll == MAXREAL) // cannot find one??
        break;
    for(i=0; i<M; i++)
        if (lowestSi[i] == lowestSiInAll)
            B[i] = B[i] - lowestSiClass_n[i]; // remove bits from that class.

    // calculate the rate and distortion of the new code.

```

```

D=0; R=0;
for(i=0; i<M; i++)
{
    D += p[i] * d[i]*(q+1) + B[i];
    R += r[i]*(q+1) + B[i] * subband[i].ncoeffs;
}
if (R<TargetRateInBytes-50)
{
    for(i=0; i<M; i++)
        if (lowestSi[i] == lowestSiInAll)
        {
            B[i] = B[i] + lowestSiClass_n[i];
            allow[i] = 0;
        }
    R = TargetRateInBytes+1;
}
else
    for(i=0; i<M; i++) allow[i]=1;
}while(R>TargetRateInBytes);

// print the statistics.
double totdist = 0, totrate=0;
for(i=0; i<M; i++)
{
    printf("Class % 2d\\%2d bits\\n", i, B[i]);
    totdist += d[i]*(q+1)+B[i] * subband[i].ncoeffs;
    totrate += r[i]*(q+1)+B[i] * subband[i].ncoeffs;
}
double rms = sqrt(totdist/totcoeffs);
double psnr = 20.0 * log(255.0/rms)/log(10.0);

printf("Requested Rate = %d\\n", (int)TargetRateInBytes);
printf("Actual rate = %d\\n", (int)totrate);
printf("RMS error = %11.2f\\n", (float)rms);
printf("PSNR = %11.2f\\n", (float)psnr);

delete[] allow;
delete[] p;
delete[] r;
delete[] d;
}

//-----//
void normal_decode(char *fname)
{
    FILE *fp = fopen(fname, "rb");
    BitStream bs(fp);

    int imwidth = bs.ReadBits(16);
    int imheight = bs.ReadBits(16);
    int nsteps = bs.ReadBits(16);
    int nbands = nsteps*3+1;

    Image image;
    image.InitEmptyImage(imwidth, imheight);

    struct stBlock *subband = new struct stBlock[nbands];
    AllocSubBands(image, subband, nsteps);

    for(int i=0; i<nbands; i++)
    {
        // read in the quantizer data.
        UniformQuant *uq = new UniformQuant();
        uq->ReadHeader(bs);
        subband[i].uq = uq;
        int nlevels = uq->nLevels;

        unsigned int *symbols = subband[i].symbols;
        // only a single level? then no bits will be stored, and the symbol is always 0.
        if (nlevels<=1)

```

```

    {
        for(int j=0; j<subband[i].ncoeffs; j++)
            *symbols++ = 0;
    }
    else // normal case.
    {
        // Setup the arithmetic decoder.
        ArithDecoder adc(&bs);
        adc.SetNumSyms(nlevels);
        adc.InitDecode();

        // now readin the symbols.
        for(int j=0; j<subband[i].ncoeffs; j++)
        {
            *symbols = adc.Decode();
            symbols++;
        }
    }
    // and dequantize them.
    double *coeffs = subband[i].coeffs;
    symbols = subband[i].symbols;

    for(int j=0; j<subband[i].ncoeffs; j++)
    {
        *coeffs++ = uq->Value(*symbols++);
    }

    image.SetData(subband[i].x, subband[i].y, subband[i].width, subband[i].height, subband[i].coeffs);
}

// close the file
fclose(fp);

// take inverse transform.
Wavelet *cw = &wAntonini;
cw->Inverse(image, nsteps);
image.SaveRawFile("wletn.raw");

FreeSubbands(subband, nbands);
for(i=0; i<nbands; i++)
    delete subband[i].uq;
delete[] subband;
}

//-----
void normal_encode(char *fname)
{
    int i,j;
    Image image;
    image.LoadRawFile(fname,512,512);

    // wavelet transform
    Wavelet *cw = &wAntonini;
    cw->Transform(image,NSTEPS);

    //printf("Number of zeros = %d\n",image.SetZeroBelow(2.0));

    // init some subband data & alloc memory for data in each subband
    struct stBlock *subband = new struct stBlock[NBANDS];
    AllocSubBands(image subband, NSTEPS);

    // separate subband data & find max & min values in each subband.
    InitSubBands(image subband);

    // Find the bitallocation for the bands
    int *BitAllocation = new int[NBANDS];
    double TargetRate = 0.32; // BitsPerPixel for the target image.
    CalcBitAllocation(subband, BitAllocation, NBANDS, 12, TargetRate);
    //int BitAllocation[] = {7,7,6,7,6,5,5,5,4,4,4,4,3,2,0,0};

```

```

// quantize all the coeffs into the 'symbols' array.
for(i=0; i<=NBANDS-1; i++)
{
    int nLevels = (1<<BitAllocation[i]);
    UniformQuant *uq = new UniformQuant(subband[i].maxcoeff, subband[i].mincoeff, nLevels);
    subband[i].uq = uq;

    double *coeffs = subband[i].coeffs;
    unsigned int *symbols = subband[i].symbols;

    for(int j=subband[i].ncoeffs; j>0; j--)
        *symbols++ = uq->Symbol(*coeffs++);
}

// open the output file.
FILE *fp = fopen("wleln.cmp", "wb");
BitStream bs(fp);

// write the file header.
bs.WriteBits(image.width, 16);
bs.WriteBits(image.height, 16);
bs.WriteBits(NSTEPS, 16);

// write each band of data.
for(i=0; i<NBANDS; i++)
{
    unsigned int nlevels = (1<<BitAllocation[i]);
    UniformQuant *uq = subband[i].uq;

    // write quantizer header.
    uq->WriteHeader(bs);

    // write the symbols.
    if (nlevels>1) // if nlevels==1, then all symbols are the same
    {
        // hence no need to write them to the file.
        ArithEncoder aec(&bs);
        aec.SetNumSyms(nlevels);
        aec.InitEncode();

        unsigned int *symbols = subband[i].symbols;
        for(j = subband[i].ncoeffs; j>0; j--)
        {
            aec.Encode(*symbols++);
        }

        aec.Flush();
    }
    bs.Flush();
    fclose(fp);

    delete[] BitAllocation;
    FreeSubbands(subband, NBANDS);
    for(i=0; i<NBANDS; i++)
        delete subband[i].uq;
    delete[] subband;
}

//-----//
void zerotree_decode(char *fname)
{
    FILE *fp = fopen(fname, "rb");
    BitStream bs(fp);
    int i;

    //read in the file header.
    int imwidth = bs.ReadBits(16);
    int imheight = bs.ReadBits(16);
    int nsteps = bs.ReadBits(16);
    int nbands = nsteps*3+1;

```

```

// create an empty image.
Image image;
image.InitEmptyImage(imwidth, imheight);

// allocate memory for each subband.
struct stBlock *subband = new struct stBlock[nbands];
AllocSubBands(image, subband, nsteps);

// alloc mem for the sigmap.
unsigned char *SigMap = new unsigned char[imwidth * imheight];
for(i=0; i<imwidth*imheight; i++)
    SigMap[i]=-1;

// read the sigmap.
ArithDecoder SigmapAdc(&bs);
SigmapAdc.SetNumSyms(3);
SigmapAdc.InitDecode();
for(i=0; i<nbands; i++)
{
    // parent subband index.
    int pband = ((i-3)>0)?i-3:0;
    for(int y=0; y<subband[i].height; y++)
    {
        for(int x=0; x<subband[i].width; x++)
        {
            if (i==0)    // no zerotrees for root subband.
            {
                SigMap[(subband[i].y+y)*imwidth+(subband[i].x+x)]=SigmapAdc.Decode();
            }
            else
            {
                // is the parent symbol not a zerotree root?
                if (SigMap[(subband[pband].y+(y/2))*imwidth+(subband[pband].x+(x/2))] != 2)
                {
                    SigMap[(subband[i].y+y)*imwidth+(subband[i].x+x)]=SigmapAdc.Decode();
                }
                else // mark this also as a zerotree root (for its children)
                    SigMap[(subband[i].y+y)*imwidth+(subband[i].x+x)]=2;
            }
        }
    }
}

// readin the symbols.
for(i=0; i<nbands; i++)
{
    // read in the quantizer header.
    UniformQuant *uq = new UniformQuant();
    uq->ReadHeader(bs);
    subband[i].uq = uq;
    int nlevels = uq->nLevels/2;

    unsigned int *symbols = subband[i].symbols;

    unsigned int zeroSymbol = uq->Symbol(0.0);
    if (nlevels <=1)
    {
        for(int y=0; y<subband[i].height; y++)
            for(int x=0; x<subband[i].width; x++)
                *symbols++ = zeroSymbol;
    }
    else
    {
        ArithDecoder adc(&bs);
        adc.SetNumSyms(nlevels);
        adc.InitDecode();

        for(int y=0; y<subband[i].height; y++)
        {

```

```

        for(int x=0; x<subband[i].width; x++)
        {
            unsigned int symbol;
            // insignificant symbol?
            if (i!=0 && SigMap[(subband[i].y+y)*imwidth+(subband[i].x+x)]==2)
                symbol = zeroSymbol;
            else
                symbol = adc.Decode();

            *symbols++ = symbol;
        }
    }
}

// dequantize
double *coeffs = subband[i].coeffs;
symbols = subband[i].symbols;
int xx=0;
for(int y=0; y<subband[i].height; y++)
{
    for(int x=0; x<subband[i].width; x++)
    {
        // reconstruct the MSB
        if (SigMap[(subband[i].y+y)*imwidth+(subband[i].x+x)]==1)
        {
            *symbols = *symbols | nlevels;
        }
        *coeffs = uq->Value(*symbols++);
        coeffs++;
    }
}
image.SetData(subband[i].x, subband[i].y, subband[i].width, subband[i].height, subband[i].coeffs);
}
fclose(fp);

// take inverse transform.
Wavelet *cw = &wAntonini;
cw->Inverse(image, nsteps);
image.SaveRawFile("wletz.raw");

delete[] SigMap;
FreeSubbands(subband, nbands);
for(i=0; i<nbands; i++)
    delete subband[i].uq;
delete[] subband;
}

//-----//
void zerotree_encode(char *fname)
{
    int i;
    Image image;
    image.LoadRawFile(fname,512,512);

    // transform.
    Wavelet *cw = &wAntonini;
    cw->Transform(image,NSTEPS);

    //printf("Number of zeros = %d\n",image.SetZeroBelow(5.0));

    // init some subband data & alloc memory for data in each subband
    struct stBlock *subband = new struct stBlock[NBANDS];
    AllocSubBands(image, subband, NSTEPS);

    // separate subband data & find max & min values in each subband.
    InitSubBands(image, subband);

    // Find the bitallocation for the bands
    int *BitAllocation = new int[NBANDS];
    double TargetRate = 0.32;

    // BitsperPixel for the target image.

```



```

CalcBitAllocation(subband, BitAllocation, NBANDS, 12, TargetRate);

// alloc mem for the significance map.
unsigned char *SigMap = new unsigned char[image.width*image.height];
for(i=0; i<image.width*image.height; i++)
    SigMap[i] = -1;

// quantize all the coeffs into the 'symbols' array.
// also, find if there are any zerotrees, and mark them.
for(i=NBANDS-1; i>=0; i--)
{
    int nLevels = (1<<BitAllocation[i]);
    UniformQuant *uq = new UniformQuant(subband[i].maxcoeff, subband[i].mincoeff, nLevels);
    subband[i].uq = uq;

    double *coeffs = subband[i].coeffs;
    unsigned int *symbols = subband[i].symbols;

    // if this is the last subband in the pyramid, child=this itself.
    int childSubband = ((i+3)>=NBANDS)?i:i+3;

    unsigned int zeroSymbol = uq->Symbol(0.0);
    unsigned int zeroTreeSymbol = nLevels;
    unsigned int meanSymbol = uq->Symbol(subband[i].meancoeff);
    unsigned int *childSymbols = subband[childSubband].symbols;

    int childx=subband[childSubband].x;
    int chily=subband[childSubband].y;

    for(int y=0; y<subband[i].height; y++)
    {
        for(int x=0; x<subband[i].width; x++)
        {
            unsigned int symbol = uq->Symbol((*coeffs));

            // store the MSB in the SigMap.
            if (symbol > meanSymbol)
                SigMap[(subband[i].y+y)*image.width+(subband[i].x+x)] = 1;
            else
                SigMap[(subband[i].y+y)*image.width+(subband[i].x+x)] = 0;

            // not the root band?
            if (i!=0)
            {
                // a zero? then we can look for zerotrees.
                if (symbol == zeroSymbol)
                {
                    // is this the last subband in the pyramid? then mark as zerotree.
                    if (childSubband == i)
                    {
                        symbol = zeroTreeSymbol;

                        // Set SigMap[...] = 2, indicating that this is a zerotree.
                        SigMap[(subband[i].y+y)*image.width+(subband[i].x+x)] = 2;
                    }
                    // are ALL the children also zerotrees?
                    else if (SigMap[(chily+y*2)*image.width+(childx+x*2)] == 2 &&
                        SigMap[(chily+y*2)*image.width+(childx+x*2+1)] == 2 &&
                        SigMap[(chily+y*2+1)*image.width+(childx+x*2)] == 2 &&
                        SigMap[(chily+y*2+1)*image.width+(childx+x*2+1)] == 2)
                    {
                        symbol = zeroTreeSymbol;
                        SigMap[(subband[i].y+y)*image.width+(subband[i].x+x)] = 2;

                        // Set SigMap[...] = 3 for children, indicating to ignore them.
                        SigMap[(chily+y*2)*image.width+(childx+x*2)] = 3;
                        SigMap[(chily+y*2)*image.width+(childx+x*2+1)] = 3;
                        SigMap[(chily+y*2+1)*image.width+(childx+x*2)] = 3;
                        SigMap[(chily+y*2+1)*image.width+(childx+x*2+1)] = 3;
                    }
                }
            }
        }
    }
}

```

```

    }
  }
}
// remove the MSB (which is now encoded in the SigMap)
symbol = (symbol & ((uq->nLevels-1)>>1));
*symbols++ = symbol;

// removing the MSB means removing the sign of the actual coefficient.
*coeffs = fabs(*coeffs);
coeffs++;
}
}

// open the output file.
FILE *fp = fopen("wletz.cmp", "wb");
BitStream bs(fp);

// write the file header.
bs.WriteBits(image.width, 16);
bs.WriteBits(image.height, 16);
bs.WriteBits(NSTEPS, 16);

// write the sigmap.
ArithEncoder SigmapAec(&bs);
SigmapAec.SetNumSyms(3);
SigmapAec.InitEncode();
for(i=0; i<NBANDS; i++)
{
  for(int y=0; y<subband[i].height; y++)
  {
    for(int x=0; x<subband[i].width; x++)
    {
      // write the symbol only if it is not marked as ignorable.
      if (SigMap[(subband[i].y+y)*image.width+(subband[i].x+x)] < 3)
      {
        SigmapAec.Encode(SigMap[(subband[i].y+y)*image.width+(subband[i].x+x)]);
      }
    }
  }
}
SigmapAec.Flush();
printf("Number of bits to code the zerotree = %d\n", bs.numBitsWritten);

// write the symbols into the file.
for(i=0; i<NBANDS; i++)
{
  UniformQuant *uq = subband[i].uq;
  unsigned int nLevels = uq->nLevels/2;

  // write band specific data.
  uq->WriteHeader(bs);

  if (nLevels>1)
  {
    ArithEncoder aec(&bs);
    aec.SetNumSyms(nLevels);
    aec.InitEncode();

    unsigned int *symbols = subband[i].symbols;
    for(int y=0; y<subband[i].height; y++)
    {
      for(int x=0; x<subband[i].width; x++)
      {
        // write the symbol only if it is not a zerotree root or child.
        if (SigMap[(subband[i].y+y)*image.width+(subband[i].x+x)] < 2)
          aec.Encode(*symbols);
        symbols++;
      }
    }
  }
}

```

```

        aec.Flush();
    }
}
bs.Flush();
printf("Total number of bits to code image = %d\n",bs.numBitsWritten);
fclose(fp);

delete[] BitAllocation;
delete[] SigMap;
FreeSubbands(subband, NBANDS);
for(i=0; i<NBANDS; i++)
    delete subband[i].uq;
delete[] subband;
}

//-----*/
int main()
{
    normal_encode("lena.raw");    // writes o/p to file wletn.cmp
    normal_decode("wletn.cmp");    // writes o/p to file wletn.raw

    zerotree_encode("lena.raw");    // writes o/p to file wletz.cmp
    zerotree_decode("wletz.cmp");    // writes o/p to file wletz.raw

    return 0;
}

```

II. Image Processing

```

// image.h
/*-----*/
//      Wavekit - Image Compression Package version 1.0
//      written by      Shixin Wei
//      Copyright© 2001 S.Wei. Permission is granted to use for research purposes only
/*-----*/

#ifndef _IMAGE_H
#define _IMAGE_H

class Image
{
public:
    double *data;    // the image data.
    int width, height;    // statistics of the image.

    Image()
    {
        data = NULL;
        width = height = 0;
    }
    ~Image()
    {
        if (data) delete[] data;
        data = NULL;
        width = height = 0;
    }
    int LoadRawFile(char *fname, int width, int height);
    int SaveRawFile(char *fname, int mustscale = 0);
    int GetData(int x, int y, int xspan, int yspan, double *tdata);
    int SetData(int x, int y, int xspan, int yspan, double *tdata);
    int SetZeroBelow(double threshold);
    void InitEmptyImage(int w, int h);
};

#endif

// image.cpp
/*-----*/
#include <stdio.h>

```

```

#include <malloc.h>
#include <math.h>
#include "Image.h"

typedef unsigned char BYTE;

//-----//
// Implementation of the Image class, a very basic framework for loading/saving .RAW image files. //
//-----//
int Image::LoadRawFile(char *fname, int twidth, int theight)
{
    FILE *fp;

    fp = fopen(fname, "rb");
    if (!fp)
        return 1;

    int imsize = twidth*theight;
    BYTE *bytedata = new BYTE[imsize];
    double *tdata = new double[imsize];
    if (!bytedata || !tdata)
    {
        fclose(fp);
        if (bytedata)
            delete[] bytedata;
        if (tdata)
            delete[] tdata;
        return 1;
    }

    if ((int)fread(bytedata, sizeof(BYTE), imsize, fp) != imsize)
    {
        fclose(fp);
        delete[] bytedata;
        delete[] tdata;
        return 1;
    }

    fclose(fp);
    for(int i=0; i<imsize; i++)
        tdata[i] = (double)bytedata[i];

    if (data)
        delete[] data;

    width = twidth;
    height = theight;
    data = tdata;

    delete[] bytedata;
    return 0;
}

int Image::SaveRawFile(char *fname, int mustscale)
{
    FILE *fp;
    int i;

    fp = fopen(fname, "wb");
    if (!fp)
        return 1;

    int imsize = width * height;
    BYTE *bytedata = new BYTE[imsize];
    if (!bytedata)
    {
        fclose(fp);
        return 1;
    }

```

```

    if (mustscale)
    {
        double scalefactor = 1.0;
        double maxvalue = 0, minvalue = 1000000.0;
        for(i=0; i<imsz; i++)
        {
            if (maxvalue < data[i])
                maxvalue = data[i];
            if (minvalue > data[i])
                minvalue = data[i];
        }
        scalefactor = 256.0/(maxvalue - minvalue);
        for(i=0; i<imsz; i++)
            bytedata[i] = (BYTE)((data[i] - minvalue) * scalefactor);
    }
    else
    {
        for(i=0; i<imsz; i++)
            bytedata[i] = (BYTE)(data[i]);
    }

    fwrite(bytedata, imsz, sizeof(BYTE), fp);

    delete[] bytedata;
    fclose(fp);
    return 0;
}

int Image::GetData(int x, int y, int xspan, int yspan, double *tdata)
{
    int i,j;

    // check for legality of the values.
    if (x>width || y>width)
        return 1;
    if (x+xspan > width)
        return 1;
    if (y+yspan > height)
        return 1;

    double *target = tdata;
    for(i=0; i<yspan; i++)
    {
        // copy each row.
        double *source = (data + (y+i)*width + x);
        j = xspan;
        while(j--)
            *target++ = *source++;
    }
    return 0;
}

int Image::SetData(int x, int y, int xspan, int yspan, double *tdata)
{
    int i,j;

    // check for legality of the values.
    if (x>width || y>width)
        return 1;
    if (x+xspan > width)
        return 1;
    if (y+yspan > height)
        return 1;

    double *source = tdata;
    for(i=0; i<yspan; i++)
    {
        // copy each row.
        double *target = (data + (y+i)*width + x);
        j = xspan;

```

```

        while(j--)
            *target++ = *source++;
    }
    return 0;
}

int Image::SetZeroBelow(double threshold)
{
    int i = width * height;
    int num = 0;
    double *dataptr = data;

    while(i--)
    {
        if (fabs(*dataptr) < threshold)
        {
            *dataptr = 0;
            num++;
        }
        dataptr++;
    }
    return num;
}

void Image::InitEmptyImage(int w, int h)
{
    Image::~Image();

    if (data = new double(w*h))
    {
        width = w;
        height = h;
    }
}

```

III. Quantizer

```

// quantizer.h
/*-----*/
//      Wavekit - Image Compression Package version 1.0
//      written by      Shixin Wei
//      Copyright© 2001 S.Wei. Permission is granted to use for research purposes only
/*-----*/

#ifndef _QUANTIZER_H
#define _QUANTIZER_H

class UniformQuant
{
public:
    double minvalue, stepsize, invstepsize;
    int nLevels;
public:
    UniformQuant()
    {
        minvalue = stepsize = invstepsize = 0.0;
        nLevels = 0;
    }
    UniformQuant(double maxv, double minv, int nlev)
    {
        minvalue = minv;
        nLevels = nlev;
        if (nlev==1)      // only a single level??
        {
            stepsize = (maxv - minv);
            invstepsize = 0.0;
        }
        else
        {
            stepsize = (maxv - minv)/(nlev-1);

```

```

        invstepsize = 1.0/stepsize;
    }
}
UniformQuant(double min, double stsize)
{
    minvalue = min;
    stepsize = stsize;
    invstepsize = 1.0/stepsize;
}
inline unsigned int Symbol(double value)
{
    return ((unsigned int)((value - minvalue) * invstepsize));
}
inline double Value(unsigned int code)
{
    return (minvalue + ((double)code + 0.5)*stepsize);
}
void WriteHeader(BitStream &bs)
{
    bs.WriteBits(nLevels, 32);
    if (nLevels<=1)
    {
        float meanvalue = (float)(minvalue+0.5*stepsize);
        bs.WriteBits*((unsigned int *)&meanvalue),32);
    }
    else
    {
        float minv = (float)minvalue;
        float stsize = (float)stepsize;
        bs.WriteBits*((unsigned int *)&minv), 32);
        bs.WriteBits*((unsigned int *)&stsize), 32);
    }
}
void ReadHeader(BitStream &bs)
{
    nLevels = bs.ReadBits(32);
    if (nLevels==1)
    {
        unsigned int tmeanv = bs.ReadBits(32);
        minvalue = *(float *)&tmeanv;
        stepsize = 0.0;
        invstepsize = 0.0;
    }
    else
    {
        unsigned int tminv = bs.ReadBits(32);
        unsigned int tsteps = bs.ReadBits(32);
        minvalue = *(float *)&tminv;
        stepsize = *(float *)&tsteps;
        invstepsize = 1.0/stepsize;
    }
}
};

#endif

```

IV. Entropy Coder

```

//. aricoder.h
/*-----*/
//      Wavekit - Image Compression Package version 1.0
//      written by      Shixin Wei
//      Copyright© 2001 S.Wei. Permission is granted to use for research purposes only
/*-----*/
//-----// Implementation of the Wavelet class.-----*/
// some functions are taken from the 'Wavelet Image compression Kit'
// written and distributed by Goeff Davis http://www.cs.dartmouth.edu/~gdavis
//-----*/

```

```

#ifndef _ARICODER_H
#define _ARICODER_H

#define onelog2 1.4426950408889634

//----- Bitstream Part.
class BitStream
{
private:
    unsigned char buf;
    int nBits;
    FILE *fp;
public:
    int numBitsWritten;
    int numBitsRead;
    BitStream(FILE *givenfp);
    ~BitStream();
    void WriteBit(unsigned int bit);
    void WriteBits(unsigned int bits,int n);
    void Flush();
    unsigned int ReadBit();
    unsigned int ReadBits(int n);
};

//-----Arithmetic Encoder & Decoder Part.

#define CodeValueBits 16 // Number of bits in a code value
#define TopValue (((long)1<<CodeValueBits)-1) // Largest code value
#define FirstQtr (TopValue/4+1) // Point after first quarter
#define Half (2*FirstQtr) // Point after first half
#define ThirdQtr (3*FirstQtr) // Point after third quarter
#define MaxAllowedFreq 16383 // 2^14-1

class AdaptiveModel
{
    unsigned int *Tree; // the cumulative freq tree.
    unsigned int numSyms; // number of alphabets.
public:
    unsigned int totfreq;

    AdaptiveModel();
    ~AdaptiveModel();
    int SetNumSyms(int n);
    void Scaledown();
    int GetCumul(int lx);
    int GetIndex(unsigned int Freq);
    void PutValue(int Val, int lx);
    int GetFreq(int lx);
};

class ArithEncoder
{
    unsigned int High, Low;
    unsigned int folw; // 'follow with' counter.

    BitStream *bs;
public:
    AdaptiveModel SrcModel;

    ArithEncoder(BitStream *bitstream);
    ~ArithEncoder();
    int SetNumSyms(int n);
    void InitEncode();
    unsigned int Encode(unsigned int symbol,int update=1);
    void Flush();
    inline void WriteBitPlusFollow(int bit)
    {
        bs->WriteBit(bit);
        bit = 1-bit;
        while(folw)

```



```

        {
            bs->WriteBit(bit);
            folw--;
        }
    }
    double EncodingCost(unsigned int symbol);
};

class ArithDecoder
{
    unsigned int High, Low;
    unsigned int folw;          // 'follow with' counter.
    unsigned int Value;         // value.

    AdaptiveModel SrcModel;
    BitStream *bs;
public:
    ArithDecoder(BitStream *bitstream);
    ~ArithDecoder();
    int SetNumSyms(int n);
    void InitDecode();
    unsigned int Decode();
};

#endif

// aricoder.cpp
/*-----*/
#include <stdio.h>
#include <math.h>
#include "aricoder.h"

int writtenBits = 0;
static int eofreached = 0;

// -----the BitStream part.
BitStream::BitStream(FILE *givenfp)
{
    buf = 0;
    nBits = 0;
    numBitsWritten = 0;
    numBitsRead = 0;
    fp = givenfp;
}

BitStream::~BitStream()
{
    buf = 0;
    nBits = 0;
    numBitsWritten = 0;
    fp = NULL;
}

void BitStream::WriteBit(unsigned int bit)
{
    buf = (buf<<1)+bit;
    nBits++;
    if (nBits==8)
    {
        fwrite(&buf,1,1,fp);
        nBits = 0;
        buf = 0;
    }
    numBitsWritten++;
}

void BitStream::Flush()
{
    if (nBits)

```

```

    {
        buf = (buf<<(8-nBits));
        fwrite(&buf,1,1,fp);
        nBits = 0;
        buf = 0;
    }
}

unsigned int BitStream::ReadBit()
{
    int bit;
    if (nBits == 0)
    {
        nBits = 8;
        fread(&buf,1,1,fp);
    }
    bit = (buf & 128)>>7;
    buf = buf<<1;
    nBits--;
    numBitsRead++;
    return bit;
}

void BitStream::WriteBits(unsigned int bits, int n)
{
    unsigned int mask = (1<<(n-1));
    while(n--)
    {
        WriteBit((bits & mask)>0);
        bits = bits<<1;
    }
}

unsigned int BitStream::ReadBits(int n)
{
    int bits=0;
    while(n--)
        bits = (bits<<1)+ReadBit();
    return bits;
}

// -----the Source Model part-----*/
// the source model is an implementation of "A New Data structure for Cumulative Frequency Tables"
// by Peter M. Fenwick, SOFTWARE-PRACTICE AND EXPERIENCE, Vol.24(3), pp 327-336 (3. 1994)
// -----*/
AdaptiveModel::AdaptiveModel()
{
    numSyms = 0;

    Tree = NULL;
    totfreq = 0;
}

AdaptiveModel::~AdaptiveModel()
{
    numSyms = 0;

    if (Tree)
        delete[] Tree;
}

int AdaptiveModel::SetNumSyms(int n)
{
    AdaptiveModel::~AdaptiveModel();

    if (Tree) delete[] Tree;
    Tree = new unsigned int[n+1];
    numSyms = n+1;
    totfreq = 0;
}

```

```

    for(unsigned int i=0; i<numSyms; i++)
    {
        Tree[i] = 0;
    }

    for(i=0; i<numSyms; i++)
        PutValue(1, i);

    return 0;
}

int AdaptiveModel::GetCumul(int lx)
{
    unsigned int sum;
    sum = Tree[0];
    while(lx > 0)
    {
        sum = sum + Tree[lx];
        lx = lx & (lx - 1);
    }
    return sum;
}

void AdaptiveModel::PutValue(int Val, int lx)
{
    lx++;
    while((unsigned int)lx < numSyms)
    {
        Tree[lx] = Tree[lx] + Val;
        lx = lx + (lx & (-lx));
    }
    totfreq += Val;
}

void AdaptiveModel::Scaledown()
{
    for(int i=numSyms-2; i>=0; i--)
        PutValue(-GetFreq(i)/2, i);
}

int AdaptiveModel::GetFreq(int lx)
{
    int val, parent;
    lx++;
    val = Tree[lx];
    if (lx > 0)
    {
        parent = lx & (lx - 1);
        lx = lx - 1;
        while(parent != lx)
        {
            val = val - Tree[lx];
            lx = lx & (lx - 1);
        }
    }
    return val;
}

int AdaptiveModel::GetIndex(unsigned int Freq)
{
    int Index, Testlx, origFreq=Freq;
    int mask = (numSyms+1)/2;
    Index = 0;
    if (Freq > Tree[0])
    {
        while(mask != 0)
        {
            Testlx = Index+mask;
            if (Freq >= Tree[Testlx])
            {

```

```

        Index = TestIdx;
        Freq = Freq - Tree[Index];
    }
    mask = mask / 2;
}
}
else return 0;
return Index;
}

//-----the Arithmetic Encoder part.-----*/
// implementation of "Arithmetic coding for Data compression"
// by Witten, Neal & Cleary, CACM, June 1987 Vol.30
//-----*/
ArithEncoder::ArithEncoder(BitStream *bitstream)
{
    bs = bitstream;
}

ArithEncoder::~ArithEncoder() {}

int ArithEncoder::SetNumSyms(int n)
{
    return SrcModel.SetNumSyms(n);
}

void ArithEncoder::InitEncode()
{
    High = TopValue;
    Low = 0;
    folw = 0;
}

void ArithEncoder::Flush()
{
    for (int i=CodeValueBits; i>0; i--)
    {
        if (Low >= Half){
            WriteBitPlusFollow(1);
            Low -= Half;}
        else
            WriteBitPlusFollow(0);
        Low = 2 * Low;
    }
}

unsigned int ArithEncoder::Encode(unsigned int symbol,int update)
{
    long Range;
    Range = High-Low+1;

    High = Low + (Range*SrcModel.GetCumul(symbol+1))/SrcModel.totfreq - 1;
    Low = Low + (Range*SrcModel.GetCumul(symbol ))/SrcModel.totfreq;

    while(1)
    {
        if (High < Half)
        {
            WriteBitPlusFollow(0);
        }
        else if (Low >= Half)
        {
            WriteBitPlusFollow(1);
            Low -= Half;
            High -= Half;
        }
        else if (Low >= FirstQtr && High < ThirdQtr)
        {
            folw++;
            Low -= FirstQtr;

```

```

        High -= FirstQtr;
    }
    else break;
    Low = Low << 1;
    High = (High << 1) + 1;
}
if (update)
{
    SrcModel.PutValue(1, symbol);
    if (SrcModel.totfreq > MaxAllowedFreq)
        SrcModel.Scaledown();
}
if (High > 65535)
    High = High;
return 0;
}

double ArithEncoder::EncodingCost(unsigned int symbol)
{
    double cost = -log((double)SrcModel.GetFreq(symbol)/SrcModel.totfreq) * onelog2;
    SrcModel.PutValue(1, symbol);
    if (SrcModel.totfreq > MaxAllowedFreq)
        SrcModel.Scaledown();
    return cost;
}

//-----the Arithmetic Decoder part.-----*/
// implementation of "Arithmetic coding for Data compression"
// by Witten, Neal & Cleary, CACM, June 1987 Vol.30
//-----*/
ArithDecoder::ArithDecoder(BitStream *b) { bs = b;}

ArithDecoder::~ArithDecoder() {}

int ArithDecoder::SetNumSyms(int n)
{
    return SrcModel.SetNumSyms(n);
}

void ArithDecoder::InitDecode()
{
    Value = 0;
    // Initialize Value.
    for(int i=0; i<CodeValueBits; i++)
        Value = (Value<<1) + bs->ReadBit();

    Low = 0;
    High = TopValue;
}

unsigned int ArithDecoder::Decode()
{
    long Range;
    unsigned int cum, symbol;

    if (eofreached)
        return EOF;
    Range = High-Low+1;
    cum = ((long)(Value-Low+1))*SrcModel.totfreq - 1)/Range;

    // Find the symbol with this cumulative frequency.
    symbol = SrcModel.GetIndex(cum);

    // Update the range of the code.
    High = Low + (Range*SrcModel.GetCumul(symbol+1))/SrcModel.totfreq - 1;
    Low = Low + (Range*SrcModel.GetCumul(symbol))/SrcModel.totfreq;

    // Now find the bits which can be send off.
    while(1)
    {

```

```

    if (High < Half)
    {
        ; // Nothing to send.
    }
    else if (Low >= Half)
    {
        Value -= Half;
        Low -= Half;
        High -= Half;
    }
    else if (Low >= FirstQtr && High < ThirdQtr)
    {
        Value -= FirstQtr;
        Low -= FirstQtr;
        High -= FirstQtr;
    }
    else break;
    Low = (Low<<1);
    High = (High<<1)+1;
    Value = (Value<<1)+bs->ReadBit();
}
SrcModel.PutValue(1,symbol);
if (SrcModel.totfreq > MaxAllowedFreq)
    SrcModel.Scaledown();

return symbol;
}

```

V. Wavelet Transform

```

// wavelet.h
/*-----*/
//      Wavekit - Image Compression Package version 1.0
//      written by      Shixin Wei
//      Copyright© 2001 S.Wei. Permission is granted to use for research purposes only
/*-----*/
#ifndef _WAVELET_H
#define _WAVELET_H

class Filter
{
public:
    double *coeffs;    // filter coefficients.
    int size;          // length of the filter.
    int startIndex;    // the starting index of the coefficients.

    Filter(int size, int startIndex, double *coeffs = NULL);
};

class Wavelet
{
    Filter *anaLow, *anaHigh;    // low and high pass filters for analysis.
    Filter *synLow, *synHigh;    // low and high pass filters for synthesis.

private:
    void PeriodicExtension(double *output, int size, int npad);
    void SymmetricExtension(double *output, int size, int left_ext, int right_ext,
        int symmetry, int npad);
    void TransformStep(double *data, int datasize, int npad, Filter *lpf,
        Filter *hpf, double *dest);
    void InverseStep(double *data, int datasize, int npad, Filter *lpf,
        Filter *hpf, double *dest);

public:
    Wavelet(Filter *anaLow, Filter *synLow = NULL);
    int Transform(Image &image, int nsteps);
    int Inverse(Image &image, int nsteps);
};

```

```

#endif

// wavelet.cpp
//-----// Implementation of the Wavelet class.-----*/
// some functions are taken from the 'Wavelet Image compression Kit'
// written and distributed by Goeff Davis http://www.cs.dartmouth.edu/~gdavis
//-----*/
//
#include <stdio.h>
#include "Image.h"
#include "Wavelet.h"
#define min(a,b) (((a)<(b))?a):(b))

Filter::Filter(int tsize, int tstartIndex, double *tcoeffs)
{
    coeffs = new double[tsize];
    if (!coeffs)
        return;

    size = tsize;
    startIndex = tstartIndex;

    if (tcoeffs)
        for(int i=0;i<tsize;i++)
            coeffs[i] = tcoeffs[i];
}

/*-----*/
// Do symmetric extension of data using prescribed symmetries
// Original values are in output[npad] through output[npad+size-1]
// New values will be placed in output[0] through output[npad] and in
// output[npad+size] through output[2*npad+size-1] (note: end values may
// not be filled in)
// left_ext = 1 -> extension at left bdry is ... 3 2 1 | 0 1 2 3 ...
// left_ext = 2 -> extension at left bdry is ... 3 2 1 0 | 0 1 2 3 ...
// right_ext = 1 or 2 has similar effects at the right boundary
//
// symmetry = 1 -> extend symmetrically
// symmetry = -1 -> extend antisymmetrically

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

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

    int originalFirst = first;
    int originalLast = last;
    int originalSize = originalLast-originalFirst+1;

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

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

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

```

```

// should have full period now – extend periodically
while (first-- > 0)
    output[first] = output[first+period];

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

// should have full period now – extend periodically
while (last++ < 2*npad+size-1)
    output[last] = output[last-period];
}

/*-----*/
// Do periodic extension of data using prescribed symmetries
// Original values are in output[npad] through output[npad+size-1]
// New values will be placed in output[0] through output[npad] and in output[npad+size]
// through output[2*npad+size-1] (note: end values may not be filled in)
/*-----*/

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

    // extend left periodically
    while (first-- > 0)
        output[first] = output[first+size];

    // extend right periodically
    while (last++ < 2*npad+size-1)
        output[last] = output[last-size];
}

Wavelet::Wavelet(Filter *aLow, Filter *sLow)
{
    int i, sign;
    anaLow = new Filter(aLow->size, aLow->startIndex, aLow->coeffs);

    // If no synthesis coeffs are given, assume wavelet is orthogonal.
    if (sLow == NULL)
    {
        // For orthogonal wavelets, compute the high pass filter using
        // the relation  $g_n = (-1)^n h_{1-n}$ 
        // (or equivalently  $g_{1-n} = (-1)^{1-n} h_n$ ).
        anaHigh = new Filter(anaLow->size, 2 - anaLow->size - anaLow->startIndex);

        // compute  $(-1)^{1-n}$  for  $n=startIndex$ 
        sign = (anaLow->startIndex % 2)?1:-1;

        for(i=0; i<anaLow->size; i++)
        {
            anaHigh->coeffs[1 - i - anaLow->startIndex - anaHigh->startIndex] =
                sign * anaLow->coeffs[i];
            sign *= -1;
        }

        // copy the analysis filters to the synthesis filters.
        synLow = new Filter(anaLow->size, anaLow->startIndex, anaLow->coeffs);
        synHigh = new Filter(anaHigh->size, anaHigh->startIndex, anaHigh->coeffs);
    }
    else // assume wavelet is biorthogonal.
    {
        synLow = new Filter(sLow->size, sLow->startIndex, sLow->coeffs);

        // For orthogonal wavelets, compute the high frequency filter using
        // the relation  $g_n = (-1)^n$  complement ( $h_{1-n}$ ) and
        //  $g_{-n} = (-1)^n$  complement ( $h_{1-n}$ )
        // (or equivalently  $g_{1-n} = (-1)^{1-n}$  complement ( $h_{-n}$ ))
    }
}

```



```

anaHigh = new Filter(synLow->size, 2 - synLow->size - synLow->startIndex);
synHigh = new Filter(anaLow->size, 2 - anaLow->size - anaLow->startIndex);

// fill in coeffs for analysis high.
sign = (synLow->startIndex % 2)?1:-1;
for(i=0; i<synLow->size; i++)
{
    anaHigh->coeffs[1 - i - synLow->startIndex - anaHigh->startIndex] =
        sign * synLow->coeffs[i];
    sign *= -1;
}

// fill in coeffs for synthesis high.
sign = (anaLow->startIndex % 2)?1:-1;
for(i=0; i<anaLow->size; i++)
{
    synHigh->coeffs[1 - i - anaLow->startIndex - synHigh->startIndex] =
        sign * anaLow->coeffs[i];
    sign *= -1;
}
}

int Wavelet::Transform(Image &image, int nsteps)
{
    int i, j, k;
    double *indata, *outdata, *imagedata;
    int *lpwidtharr, *lpheightarr;

    // Find out how much we need to pad (the greater of the two filter lengths-1).
    int npad = anaLow->size;
    if (npad < anaHigh->size)
        npad = anaHigh->size;

    // allocate data for the largest side.
    if (image.width > image.height)
    {
        indata = new double[image.width + npad * 2 + image.width];
        outdata = indata + image.width + npad * 2;
    }
    else
    {
        indata = new double[image.height + npad * 2 + image.height];
        outdata = indata + image.height + npad * 2;
    }
    if (!indata)
        return 1;

    // allocate space to store the size of the lowpass regions in each step.
    lpwidtharr = new int[nsteps * 2];
    if (!lpwidtharr)
    {
        delete[] indata;
        return 1;
    }
    lpheightarr = lpwidtharr + nsteps;

    // and calculate the sizes.
    lpwidtharr[nsteps-1] = image.width;
    lpheightarr[nsteps-1] = image.height;
    for(i=nsteps-2; i>=0; i--)
    {
        lpwidtharr[i] = (lpwidtharr[i+1]+1)/2;
        lpheightarr[i] = (lpheightarr[i+1]+1)/2;
    }

    // now do the transform.
    for(k=nsteps-1; k>=0; k--)
    {

```

```

int lpwidth = lpwidtharr[k];
int lpheight = lpheightarr[k];

// first apply row-wise.
imagedata = image.data;
for(i=0; i<lpheight; i++)
{
    // copy 1 row worth of indata to the buffer.
    for(j=0; j<lpwidth; j++)
        indata[npad+j] = imagedata[j];

    TransformStep(indata, lpwidth, npad, anaLow, anaHigh, outdata);

    // copy back
    for(j=0; j<lpwidth; j++)
        imagedata[j] = outdata[j];

    imagedata += image.width;
}

// then apply column-wise.
imagedata = image.data;
for(i=0; i<lpwidth; i++)
{
    // copy 1 row worth of indata to the buffer.
    for(j=0; j<lpheight; j++)
        indata[npad+j] = imagedata[j*image.width];

    TransformStep(indata, lpheight, npad, anaLow, anaHigh, outdata);

    // copy back
    for(j=0; j<lpheight; j++)
        imagedata[j * image.width] = outdata[j];

    imagedata += 1;
}
}
return 0;
}

void Wavelet: TransformStep(double *data, int datasize, int npad, Filter *lpf, Filter *hpf, double *dest)
{
    int length, start;
    double *coeffs;
    int i,j;

    // fill in the left and right padding samples.
    int lefttext, righttext;
    lefttext = (anaLow->size % 2) ? 1 : 2;
    righttext = lefttext
    SymmetricExtension(data, datasize, lefttext, righttext, 1, npad);

    // now apply the lowpass filter.
    length = lpf->size;
    start = lpf->startindex;
    coeffs = lpf->coeffs;

    int lpsize = (datasize+1)/2;
    for(i=0; i<lpsize; i++)
    {
        dest[i] = 0;
        for(j=0; j<length; j++)
            dest[i] += data[npad + i*2 + start + j] * coeffs[j];
    }

    // now apply the highpass filter.
    length = hpf->size;
    start = hpf->startindex;
    coeffs = hpf->coeffs;

    for(i=lpsize; i<datasize; i++)

```

```

    {
        dest[i] = 0;
        for(j=0; j<length; j++)
            dest[i] += data[npad + (i-lpsize)*2 + start + j] * coeffs[j];
    }
}

int Wavelet::Inverse(Image &image, int nsteps)
{
    int i, j, k;
    double *indata, *outdata, *imagedata;
    int *lpwidtharr, *lpheightarr;

    // Find out how much we need to pad (the greater of the two filter lengths-1).
    int npad = synLow->size;
    if (npad < synHigh->size)
        npad = synHigh->size;

    // allocate data for the largest side.
    if (image.width > image.height)
    {
        indata = new double[(image.width + npad * 2) * 2];
        outdata = indata + image.width + npad * 2;
    }
    else
    {
        indata = new double[(image.height + npad * 2) * 2];
        outdata = indata + image.height + npad * 2;
    }
    if (!indata)
        return 1;

    // allocate space to store the size of the lowpass regions in each step.
    lpwidtharr = new int[nsteps * 2];
    if (!lpwidtharr)
    {
        delete[] indata;
        return 1;
    }
    lpheightarr = lpwidtharr + nsteps;

    // and calculate the sizes.
    lpwidtharr[nsteps-1] = image.width;
    lpheightarr[nsteps-1] = image.height;
    for(i=nsteps-2; i>=0; i--)
    {
        lpwidtharr[i] = (lpwidtharr[i+1]+1)/2;
        lpheightarr[i] = (lpheightarr[i+1]+1)/2;
    }

    // now do the transform.
    for(k=0; k<=nsteps-1; k++)
    {
        int lpwidth = lpwidtharr[k];
        int lpheight = lpheightarr[k];

        // apply column-wise.
        imagedata = image.data;
        for(i=0; i<lpwidth; i++)
        {
            // copy 1 row worth of indata to the buffer.
            for(j=0; j<lpheight; j++)
                indata[npad+j] = imagedata[i*image.width];

            InverseStep(indata, lpheight, npad, synLow, synHigh, outdata);

            // copy back
            for(j=0; j<lpheight; j++)
                imagedata[i*image.width] = outdata[npad + j];
        }
    }
}

```

```

        imagedata += 1;
    }

    // apply row-wise.
    imagedata = image.data;
    for(i=0; i<lpheight; i++)
    {
        // copy 1 row worth of indata to the buffer.
        for(j=0; j<lpwidth; j++)
            indata[npad+j] = imagedata[j];

        InverseStep(indata, lpwidth, npad, synLow, synHigh, outdata);

        // copy back
        for(j=0; j<lpwidth; j++)
            imagedata[j] = outdata[npad + j];

        imagedata += image.width;
    }
}
return 0;
}

void Wavelet::InverseStep(double *data, int datasize, int npad, Filter *lpf, Filter *hpf, double *dest)
{
    int length;
    double *coeffs, *tempin;
    int i,j;

    int lpsize = (datasize+1)/2;
    int hpsize = datasize - lpsize;

    tempin = new double[npad*2+datasize];

    // copy the lowpass samples.
    for(i=0; i<lpsize; i++)
        tempin[npad+i] = data[npad+i];

    // fill in the padding samples.
    int lefttext, righttext;
    lefttext = (anaLow->size % 2) ? 1 : 2;           // odd filter length? lefttext = 1
    righttext = (datasize % 2) ? 1 : 2;             // odd data length? righttext = 1
    SymmetricExtension(tempin, lpsize, lefttext, righttext, 1, npad);

    // initialise the result to zero.
    for(i=0; i<datasize+npad*2; i++)
        dest[i] = 0.0;

    // apply the lowpass filter.
    length = lpf->size;
    coeffs = lpf->coeffs;

    int firstIndex = lpf->startIndex;
    int lastIndex = length - 1 + firstIndex;

    for (i = -lastIndex/2; i <= (datasize-1-firstIndex)/2; i++)
        for (j = 0; j < length; j++)
            dest[npad + 2*i + firstIndex + j] += tempin[npad+i] * coeffs[j];

    // now to the highpass samples.
    for(i=0; i<hpsize; i++)
        tempin[npad+i] = data[npad+lpsize+i];

    // fill in the padding samples to the left and right.
    lefttext = 2;
    int symmetry;
    if (anaLow->size % 2)
    {
        righttext = (datasize % 2) ? 2 : 1;
        symmetry = 1;
    }

```

```

    }
    else
    {
        righttext = (datasize % 2) ? 1 : 2;
        symmetry = -1;
    }
    SymmetricExtension(tempin, hpsize, lefttext, righttext, symmetry, npad);

    // apply the highpass filter.
    length = hpf->size;
    coeffs = hpf->coeffs;

    firstIndex = hpf->startIndex;
    lastIndex = length - 1 + firstIndex;

    for (i = -lastIndex/2; i <= (datasize-1-firstIndex)/2; i++)
        for (j = 0; j < length; j++)
            dest[npad + 2*i + firstIndex + j] += tempin[npad+i] * coeffs[j];
}

```

VI. Lapped Transform

```

// glbt.cpp
// _____*/
//      General Lapped Biorthogonal Transform
//      Written by Shixin Wei
//      Copyright © 2001 S. Wei. Permission is granted to use for research purpose only.
// _____*/

// _____// Implementation of Lapped Transform_____*/
//Lapped transform is done using a modified version of the algorithm given in Discrete Cosine Transform
//by K.R.Rao and P.Yip. The original program is developed by Henrique S. Malvar. Some modification
//has been done by referring to the 'Uicoder' developed by Nguyen et al. http://saigon.ece.wisc.edu/uicoder.
// _____*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "image.h"
#include "aricoder.h"
#include "quantizer.h"

#define sat_uchar(t) ((t < 0.0) ? 0.0 : ((t > 255.0) ? 255.0 : t))
#define free_allocate(X) free(X[0]), free(X);
int **allocate_int();
float **allocate_float();
double **allocate_double();
char **allocate_char();
unsigned char **allocate_uchar(), **alloc_read_image();
short int **allocate_short_int();
FILE *open_get_file(), *open_file();

float LBTa[128]=
{-2.1191909e-01,-1.8196802e-01,1.0914440e-04,-9.4264236e-02,3.8595789e-02,-3.4930837e-02,4.9970195e-02,
1.9559361e-02,-1.3961840e-01,-1.9661598e-01,1.6037001e-01,5.3338393e-02,9.2327086e-02,1.2467833e-01,-
9.2404582e-02,-3.1344144e-02,-3.3872708e-02,-9.5395778e-02,1.7972542e-01,2.5597816e-01,-2.4357678e-01,
-1.2311300e-01,1.0673994e-02,-1.9906380e-02,9.3595665e-02,1.0868480e-01,-6.3474352e-02,-1.3317786e-02,-
5.6128734e-02,-1.0218159e-01,1.6422597e-01,1.1626840e-01,2.3114228e-01,3.4100854e-01,-3.6292787e-01,-
3.9497880e-01,4.2911613e-01,3.6083514e-01,-3.5631454e-01,-2.2434022e-01,3.5831589e-01,4.6362301e-01,-
3.5055566e-01,-1.6415283e-01,-1.3163058e-01,-3.1279511e-01,4.7723341e-01,3.1906754e-01,4.6619482e-01,
4.2905620e-01,7.3131597e-03,4.2662185e-01,-4.5465048e-01,-7.4344824e-02,-4.0584883e-01,-3.8322186e-
01,5.3812793e-01,2.2604475e-01,4.2944014e-01,3.6069631e-01,3.2594757e-01,4.3221619e-01,1.5246437e-01,
3.9834371e-01,5.3812793e-01,-2.2604475e-01,4.2944014e-01,-3.6069631e-01,3.2594757e-01,-4.3221619e-01,
1.5246437e-01,-3.9834371e-01,4.6619482e-01,-4.2905620e-01,7.3131597e-03,-4.2662185e-01,-4.5465048e-01,
7.4344824e-02,-4.0584883e-01,3.8322186e-01,3.5831589e-01,-4.6362301e-01,-3.5055566e-01,1.6415283e-01,-
1.3163058e-01,3.1279511e-01,4.7723341e-01,-3.1906754e-01,2.3114228e-01,-3.4100854e-01,-3.6292787e-01,
3.9497880e-01,4.2911613e-01,-3.6083514e-01,-3.5631454e-01,2.2434022e-01,9.3595665e-02,-1.0868480e-01,-
6.3474352e-02,1.3317786e-02,-5.6128734e-02,1.0218159e-01,1.6422597e-01,-1.1626840e-01,-3.3872708e-02,

```

```
9.5395778e-02, 1.7972542e-01, -2.5597816e-01, -2.4357678e-01, 1.2311300e-01, 1.0673994e-02, 1.9906380e-02,
1.3961840e-01, 1.9661598e-01, 1.6037001e-01, -5.3338393e-02, 9.2327086e-02, -1.2467833e-01, -9.2404582e-02,
3.1344144e-02, -2.1191909e-01, 1.8196802e-01, 1.0914440e-04, 9.4264236e-02, 3.8595789e-02, 3.4930837e-02,
4.9970195e-02, -1.9559361e-02};
```

```
float LBTs[128]=
{1.7863920e-02, -1.4411799e-02, 6.1315639e-02, 1.9517156e-02, 5.2426546e-02, 5.3408655e-02, 4.6076341e-02,
8.3324115e-02, 5.6920246e-02, -1.6813024e-02, 1.6037108e-01, 1.2407306e-01, 4.8882116e-02, 1.6064616e-01,
9.0423936e-02, -2.1939189e-02, 1.0664692e-01, 6.5754854e-02, 1.2462013e-01, 2.4092241e-01, -2.1792622e-01,
1.3556294e-01, 2.1084469e-02, -2.0631974e-04, 1.6255815e-01, 2.0555316e-01, -1.2304453e-01, -3.5598739e-02,
2.1812290e-02, -8.4323166e-02, 1.3397362e-01, 1.2746610e-01, 2.2147634e-01, 3.4661124e-01, -3.8106502e-01,
3.5546845e-01, 3.6530403e-01, 3.9610216e-01, -3.0169785e-01, -2.3277757e-01, 2.7738757e-01, 4.0526043e-01,
3.2843219e-01, -1.2297514e-01, -1.2623361e-01, -3.5462422e-01, 4.1230527e-01, 3.4133429e-01, 3.2711425e-01,
3.3120028e-01, 3.9391208e-02, 3.8507205e-01, -3.8248451e-01, -8.3609953e-02, -3.5154928e-01, -4.0905814e-01,
3.6617057e-01, 1.3189886e-01, 4.4324237e-01, 2.9999633e-01, 2.8190585e-01, 4.5454663e-01, 1.3231892e-01,
4.1413507e-01, 3.6617057e-01, -1.3189886e-01, 4.4324237e-01, -2.9999633e-01, 2.8190585e-01, -4.5454663e-01,
1.3231892e-01, -4.1413507e-01, 3.2711425e-01, -3.3120028e-01, 3.9391208e-02, -3.8507205e-01, -3.8248451e-01,
8.3609953e-02, -3.5154928e-01, 4.0905814e-01, 2.7738757e-01, -4.0526043e-01, -3.2843219e-01, 1.2297514e-01,
1.2623361e-01, 3.5462422e-01, 4.1230527e-01, -3.4133429e-01, 2.2147634e-01, -3.4661124e-01, -3.8106502e-01,
3.5546845e-01, 3.6530403e-01, -3.9610216e-01, -3.0169785e-01, 2.3277757e-01, 1.6255815e-01, -2.0555316e-01,
1.2304453e-01, 3.5598739e-02, -2.1812290e-02, 8.4323166e-02, 1.3397362e-01, -1.2746610e-01, 1.0664692e-01,
6.5754854e-02, 1.2462013e-01, -2.4092241e-01, -2.1792622e-01, 1.3556294e-01, 2.1084469e-02, 2.0631974e-04,
5.6920246e-02, 1.6813024e-02, 1.6037108e-01, -1.2407306e-01, 4.8882116e-02, -1.6064616e-01, -9.0423936e-02,
2.1939189e-02, 1.7863920e-02, 1.4411799e-02, 6.1315639e-02, -1.9517156e-02, 5.2426546e-02, -5.3408655e-02,
4.6076341e-02, -8.3324115e-02};
```

```
/* Forward Lapped Transform */
```

```
/* _____ */
```

```
void glot(float *y, float *GLOTmat, unsigned int m, unsigned int n,
          unsigned int M, unsigned int L, unsigned int N, unsigned int dim)
```

```
{
    unsigned int mm, nn, i, k, l, r, sb, se, t, mstep, nstep, von, ext, extm;
    float *xtmp;
    float tmp1, tmp2;
    if (dim)
    { mstep=m; nstep=1; mm=m; nn=n; }
    else
    { mstep=1; nstep=m; mm=n; nn=m; }
    if ((xtmp=(float *)malloc((mm+(N-1)*M)*sizeof(float))) == NULL) {
        printf("Error allocating memory for xtmp in glot.c\n");
        exit(0);
    }
    ext=(L-M)/2;
    for (t=0; t<nn; t++)
    {
        von=t*mstep;
        for (i=ext, k=von, extm=ext+mm; i<extm; k+=nstep)
            xtmp[i++]=y[k];
        for (i=0, k=2*ext; i<ext;)
            xtmp[i++]=xtmp[-k];
        for (i=2*ext+mm, k=mm, extm=ext+mm; i>extm;)
            xtmp[-i]=xtmp[k++];
        for (i=0, k=von; i<mm; i+=M)
            for (l=0; l<M; l+=2)
            {
                sb=i; se=i+L; r=l;
                tmp1=GLOTmat[r]*(xtmp[sb]+xtmp[-se]);
                tmp2=GLOTmat[r+1]*(xtmp[sb++]-xtmp[se]); r+=M;
                for (; sb<se;)
                {
                    tmp1+=GLOTmat[r]*(xtmp[sb]+xtmp[-se]);
                    tmp2+=GLOTmat[r+1]*(xtmp[sb++]-xtmp[se]);
                    r+=M;
                }
                y[k]=tmp1; k+=nstep; y[k]=tmp2; k+=nstep;
            }
    }
    free(xtmp);
}
```

```

/* Inverse Lapped Transform */
/*-----*/
void iglot(float *x,float *GLOTmat,unsigned int m,unsigned int n,
          unsigned int M,unsigned int L,unsigned int N,unsigned int dim)
{
    unsigned int mm,nn,i,k,lb,le,r,s,ss,extm,t,mstep,nstep,von,bis,ext;
    unsigned int start,end;
    float *xtmp,*ytmp;
    float tmp1,tmp2,coefs,coefa;
    if (dim)
    { mstep=m;nstep=1;mm=m;nn=n;}
    else
    { mstep=1;nstep=m;mm=n;nn=m;}
    ext=N/2*M;
    if ((xtmp=(float *)malloc((mm+2*ext)*sizeof(float))) == NULL) {
        printf("Error allocating memory for xtmp in iglot.c\n");
        exit(0);
    }
    if ((ytmp=(float *)malloc((mm+2*ext+L-M)*sizeof(float))) == NULL) {
        printf("Error allocating memory for ytmp in iglot.c\n");
        exit(0);
    }
    for (t=0;t<nn;t++)
    {
        von=t*mstep;
        /* fill-up input line buffer */
        for (i=ext,k=von,extm=ext+mm;i<extm;k+=nstep)
            xtmp[i++]=x[k];

        /* left symmetric extension */
        for (i=0,k=2*ext;i<ext;i+=M)
            for (r=0,k-=M;r<M;r++)
            {
                xtmp[i+r]=xtmp[k+r++];
                xtmp[i+r]=-xtmp[k+r];
            }

        /* right symmetric extension */
        for (i=2*ext+mm,k=mm,extm=ext+mm;i>extm;k+=M)
            for (r=0,i-=M;r<M;r++)
            {
                xtmp[i+r]=xtmp[k+r++];
                xtmp[i+r]=-xtmp[k+r];
            }

        /* initialize output buffer */
        for (i=0,bis=mm+2*ext+L-M;i<bis;i++)
            ytmp[i]=0.0;

        /* process each block - old */
        for (i=0,bis=mm+2*ext;i<bis;i+=M)
            for (lb=i,le=i+L,r=0,ss=M+i;lb<le;)
            {
                s=i; tmp1=GLOTmat[r++]*xtmp[s++];tmp2=GLOTmat[r++]*xtmp[s++];
                for (;s<ss;)
                { tmp1+=GLOTmat[r++]*xtmp[s++];
                  tmp2+=GLOTmat[r++]*xtmp[s++]; }
                ytmp[lb++]+=tmp1+tmp2;ytmp[-le]+=tmp1-tmp2;
            }

        for (i=L-M+(1-N%2)*M/2,k=von,bis=von+mm*nstep;k<bis;k+=nstep)
            x[k]=ytmp[i++].
    }
    free(xtmp);free(ytmp);
}

/* allocate 2D array */
unsigned char      **allocate_uchar (int c,int r)

```

```

{
    unsigned char **p;
    int i, j;

    if ((p = (unsigned char **) malloc (sizeof (unsigned char *) * r)) == NULL){
        printf (" Error in space allocation : allocate_uchar\n");
        exit (0);
    }
    if ((p[0] = (unsigned char *) malloc (c*r*sizeof(unsigned char))) == NULL){
        printf (" Error in space allocation : allocate_uchar\n");
        exit (0);
    }
    for (i = 1; i < r; i++){
        p[i] = p[i-1] + c;
    }
    return p;
}

/* process pgm header */
void get_image_info(FILE *fd,int *row,int *col,int *color)
{
    char P, Five, str[256];

    fread(&P, 1, 1, fd);
    fread(&Five, 1, 1, fd);
    rewind(fd);
    if ((P == 'P') && (Five == '5')){
        fgets (str, 256, fd);
        do { fgets (str, 256, fd);} while (str[0] == '#');
        sscanf (str, "%d%d", col, row);
        fgets (str, 256, fd);
        *color = 1;
    }
    else if ((P == 'P') && (Five == '6')){
        fgets (str, 256, fd);
        do { fgets (str, 256, fd);} while (str[0] == '#');
        sscanf (str, "%d%d", col, row);
        fgets (str, 256, fd);
        *color = 3;
    }
    else {
        *color = 1;
        fread(col, sizeof(int), 1, fd);
        fread(row, sizeof(int), 1, fd);
    }
}

/* read pgm (raw with simple header) image */
unsigned char **alloc_read_image(char *file,int *row,int *col,int *color)
{
    FILE *fd;
    unsigned char **image;

    if ((fd = fopen(file, "r")) == NULL){
        printf(" Error opening file [%s]\n", file);
        return NULL;
    }
    get_image_info(fd, row, col, color);
    image = allocate_uchar( (*col)*(*color), *row);
    fread(image[0], 1, (*row)*(*col)*(*color), fd);
    fclose(fd);
    return image;
}

/* write pgm (raw with simple header) image */
void save_image(unsigned char **image,char *file,int row,int col,int color)
{
    FILE *fd;
    int i, j;
    int vai;

```



```

    if ((fd = open_file(file,"w")) == NULL){
        printf("Error opening file [%s]\n", file);
        return ;
    }
    if (color == 3)
        fprintf(fd,"P6\n%d %d\n255\n", col, row);
    else {
        fprintf(fd,"P5\n%d %d\n255\n", col, row);
        color = 1;
    }
    fwrite(image[0], 1, color*row*col, fd);
    fclose(fd);
}

FILE *open_get_file (char *message,char *mode,char *dir,char *ext)
{
    char file[256], temp[256], dummy;
    FILE *fd;

    printf ("%s", message);
    do {
        scanf ("%s", temp);
        sprintf (file, "%s/%s%s", dir, temp, ext);
        if ((fd = fopen (file, mode)) == NULL) {
            printf ("\n\t\tError opening file [%s]\n\n", file);
            printf ("%s", message);
        }
    } while (fd == NULL)

    return (fd);
}

FILE *open_file (char *file,char *mode)
{
    FILE *fd;
    if ((fd = fopen (file, mode)) == NULL)
        printf ("Error opening file [%s]\n\n", file);
    return (fd);
}

main (argc, argv)
int argc;
char **argv;
{
    FILE *fd, *fd1, *fd2, *fd3;
    int i, j, k, brow, bcol, br, bc, row, col, color, npt;
    unsigned char **image, **image1, **oimage;
    unsigned int M=8, L=16, N=2, dim;
    float *temp, *temp1;
    float mse=0.0, psnr;
    long start_time, end_time;
    float Q1[8][8] =
    {
        {16.0f, 11.0f, 10.0f, 16.0f, 24.0f, 40.0f, 51.0f, 61.0f},
        {12.0f, 12.0f, 14.0f, 19.0f, 26.0f, 58.0f, 60.0f, 55.0f},
        {14.0f, 13.0f, 16.0f, 24.0f, 40.0f, 57.0f, 69.0f, 56.0f},
        {14.0f, 17.0f, 22.0f, 29.0f, 51.0f, 87.0f, 80.0f, 62.0f},
        {18.0f, 22.0f, 37.0f, 56.0f, 68.0f, 109.0f, 103.0f, 77.0f},
        {24.0f, 35.0f, 55.0f, 64.0f, 81.0f, 104.0f, 113.0f, 92.0f},
        {49.0f, 64.0f, 78.0f, 87.0f, 103.0f, 121.0f, 120.0f, 101.0f},
        {72.0f, 92.0f, 95.0f, 98.0f, 112.0f, 100.0f, 103.0f, 99.0f}
    };

    setbuf( stdout, NULL );
    if (argc < 3 || argc > 3 ) {
        printf ("\n\nForward & Inverse LT – Convolution Version\n");
        printf ("Usage: %s <infile> <outfile>\n\n", *argv);
        printf ("infile = input image file in pgm format\n");
        printf ("outfile = output image file in pgm format\n");
        printf ("Print rate and distortion to standard output\n");
    }
}

```

```

    printf("\n\n");
    exit (0);
}

start_time = clock();
printf("\n");
printf ("Forward & Inverse LT – Convolution Version\n\n");
image = alloc_read_image (argv[1], &row, &col, &color);
image1 = allocate_uchar(col,row);
npt = row*col;

if ((temp = (float *)malloc(npt*sizeof(float))) == NULL){
    printf("Error allocating memory for timage\n");
    exit(0);
}

for(i=0; i<npt; i++)
    temp[i] = (float)image[0][i];

/* forward transform */
glot(temp,LBTa,col,row,M,L,N,0); /* forward LT of rows */
glot(temp,LBTa,col,row,M,L,N,1); /* forward LT of cols */

/* write dec image for debugging purposes */
oimage = allocate_uchar( col, row);
for (k=0; k < row*col; k++)
    oimage[0][k] = (unsigned char)(sat_uchar(fabs(temp[k]))+0.5);
save_image(oimage, "dec.pgm", row, col, 1);
free_allocate(oimage);

/* Quantization & Inverse Quantization */
for(br=0; br<row; br++)
    for(bc=0; bc<col; bc++) {
        j = br*col+bc;
        brow=br%8; bcol=bc%8;
        temp[j] = Q1[brow][bcol]*(float)floor(0.499+temp[j]/Q1[brow][bcol]);
    }

/* Entropy Coding */
// Variable-length zig-zag scanning and entropy coding
// Refer to code in JPEG version 6b, developed by Independent JPEG Group
// http://www.iqg.org

/* inverse transform */
iglot(temp,LBTs,col,row,M,L,N,1); /* inverse LT of cols */
iglot(temp,LBTs,col,row,M,L,N,0); /* forward LT of rows */

for(i=0; i<npt; i++)
    image1[0][i] = (unsigned char)sat_uchar(temp[i]+0.499);

end_time = clock();
printf("\nprocessing time=%2f\n",
    (float)((end_time-start_time)/1000000.0));
save_image (image1, argv[2], row, col, color);
free(image); free(image1);
return 0;
}

//=====

```