1997

Wavelet packets in image compressioon : Spatial and frequency optimization

Lester Gong
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Gong, Lester, "Wavelet packets in image compressioon : Spatial and frequency optimization" (1997).
Master's Theses. 1445.
DOI: <https://doi.org/10.31979/etd.bj8p-jqx5>
https://scholarworks.sjsu.edu/etd_theses/1445

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

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. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

WAVELET PACKETS IN IMAGE COMPRESSION: SPATIAL AND FREQUENCY OPTIMIZATION

A Thesis

Presented to

The Faculty of the Department of Electrical Engineering
San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Lester Gong

May, 1997

UMI Number: 1384694

**Copyright 1997 by
Gong, Lester Jan Ming**

All rights reserved.

**UMI Microform 1384694
Copyright 1997, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
**300 North Zeeb Road
Ann Arbor, MI 48103**

APPROVED FOR THE DEPARTMENT OF
ELECTRICAL ENGINEERING

Dr. Essam Marouf

Dr. Richard Duda

Dr. Chang Choo

APPROVED FOR THE UNIVERSITY

M. Lou Lewandowski

© 1997

Lester Gong

ALL RIGHTS RESERVED

Abstract

WAVELET PACKETS IN IMAGE COMPRESSION: SPATIAL AND FREQUENCY OPTIMIZATION

by Lester Gong

Application of adaptive wavelet packets currently dominate the literature in transform-based image compression. Many image compression systems are variations of searching for the optimum wavelet packet basis and for the most efficient coefficient energy compaction. A conceptually promising idea that takes advantage of spatial segmentation to isolate local similarities in an image has been described in recent papers as a state-of-the-art technique. This so-called “Double-Tree” (DT) algorithm [9][14] adapts wavelet packets to local spatial phenomena, thereby optimizing image representation in both the spatial and frequency domains. This thesis examines general image compression theory and explains the motivation for using wavelet packets as a transform method of choice. It explains the theory behind the Double-Tree algorithm and its advantages. An implementation of the Double-Tree algorithm in C++ code is developed, leveraging the core modules in the “Baseline Wavelet Transform Coding Construction Kit” [7]. Comparisons of the Double-Tree coder with a wavelet packet coder and two other wavelet transform coders are made with quantitative and qualitative measures using various standard target images.

Acknowledgments

My gratitude goes out to the Faculty of the Electrical Engineering Department at San Jose State University, who over 7 years of my life, chipped away at my ignorance and brought out the engineer in me.

Many thanks goes to Professor Essam Marouf who introduced me to the topic of image compression and wavelets, and whose teaching has led me to a career in the field. He did much to focus my efforts in order to make this thesis a reality, especially down the “home stretch.”

Thanks also to Professor Dick Duda, who was always happy to help and was instrumental in shaping my academic endeavors. I greatly value his thoughtful e-mail answers to my many e-mail questions.

And finally, special thanks goes to my family, who frequently had to endure the worst in me, brought about by too much thinking and not enough sleep. Lucky for them, my work kept me in my room most of the time.

Table of Contents

1. INTRODUCTION	1
1.1 Thesis Organization	2
2. IMAGE COMPRESSION BUILDING BLOCKS	3
2.1 Compression Using Linear Transformations	4
2.2 Transformation	4
2.3 Quantization	5
2.4 Entropy Coding	8
3. JPEG	10
3.1 Transform	10
3.2 Quantization	11
3.3 Coding	12
3.4 Drawback of JPEG	12
4. WAVELETS	14
4.1 Wavelets and Filter Banks	17
4.2 Wavelet Packets	20
5. WAVELETS IN IMAGE COMPRESSION	23
6. DEVELOPING THE IMAGE COMPRESSION SYSTEM	27
6.1 The Single-Tree Algorithm	27
6.2 The Double Tree Algorithm	29
6.2.1 Conceptual Discussion	30

6.2.2 The Algorithm	36
6.3 Program Components	37
6.3.1 Wavelet Transform	37
6.3.2 Quantizer	37
6.3.3 Entropy Coding	38
7. COMPRESSION EXPERIMENTS	39
7.1 Original Test Images	40
7.2 Quantitative Results	48
7.3 Subjective Evaluations	57
7.3.1 "Lena" Image	57
7.3.2 "Barbara" Image	58
7.3.3 "Goldhill" Image	59
7.3.4 "Boat" Image	59
7.3.5 "Peppers" Image	60
7.3.6 "Building" Image	60
7.3.7 "Camera" Image	61
7.3.8 Summary of Subjective Evaluations	61
8. CONCLUSIONS	64
REFERENCES	67
APPENDIX A - Sample Images from Compression Tests	70
APPENDIX B - Image Compression Program	90
Software Description	90
Program Flow Description	93
C++ Code	99

Chapter 1

Introduction

Image coding is a subject that has in recent years been a topic of high interest. A current industry standard like JPEG that incorporates a standardized decoder (fixed quantization table) constrains the encoding effort with a potentially far less than optimum scheme. Its advantage is, though, that it is simple to implement. This solution, however, begs the question: “Can we do better by compressing an image adaptively?” And a follow-on question: “How do we cope with the trade-offs among the key factors in compression such as image quality, coding efficiency, and computational complexity?” A natural place to begin the search for optimization is to exploit the intuitive idea of coding accuracy at different resolutions. Multiresolution decomposition and wavelet analysis stands to fare well in solving this problem. Wavelet transforms in particular have growing appeal and have been the transform of choice for image processing as revealed in the recent literature [9][14][28][31]. In addition, optimization techniques for design and coding in image compression systems using wavelets have been developed [14][28][29].

1.1 Thesis Organization

This thesis first explores the basics of transform coding image compression, breaking down the process into common functional blocks or “building” blocks. The JPEG compression standard is used to illustrate how the building blocks are implemented in a working compression scheme. Wavelets and wavelet packets are then introduced as a way to use ideas of multiresolution and subband coding in compression optimization. Finally, a working image compression system is assembled using the ideas put forth. The compression system implementing the Double-Tree algorithm, is compared to other compression methods using standard high-quality grayscale images.

Chapter 2

Image Compression Building Blocks

Many image compression schemes take advantage of the sparsity of data in the transform domain. In orthogonal transformations, the image is projected onto orthogonal bases; there is no redundancy in the resulting coefficients representing the original image. Transforming the image is just the beginning. After we calculate the coefficients that represent the image, there is still much we can do to further shrink our data, namely coefficient quantization and entropy coding. There are a variety of quantization schemes to choose from, and yet another variety of ways to efficiently code the coefficients we save.

Figure 1: Block diagram of a basic image compression system.

2.1 Compression Using Linear Transformations

Compression systems based on linear transformations can be broken down into three basic building blocks: 1) Transform, 2) Quantizer, and 3) Entropy Coder (Figure 1). Their corresponding functions can be roughly described as: 1) decorrelating the pixel values of the image or subimages of the original, 2) discretizing resulting coefficients to meet a desired coding budget, and 3) efficiently coding the quantized output for storage in a file or for transmission. Although there are various techniques currently in use within each of the three functional blocks and variations in layout of the blocks, we restrict our discussion to the open-loop system shown in Figure 1, and to common techniques of implementation as used in the compression system described in Chapter 6.

2.2 Transformation

In practice, the removal of inter-pixel dependencies through linear transformation produces a more compact representation of the original pixels. There is no assurance, though, that the transformed representation is worth the computational effort [25, p. 372]. In the extreme example of an image consisting of random values, the transform produces no compaction benefits. For most natural images, however, the information packing benefits of the linear transform is evident as many resulting coefficients may be coarsely quantized and insignificant values discarded entirely without appreciable effect on the reconstructed image [8, p. 377]. In decompressing an image, the inverse transform is performed and the image is reconstructed from the set of coefficients. For computational simplicity, the image may be divided into smaller blocks of fixed size. The JPEG

compression standard uses this subdivision method, however annoying artifacts are often introduced at the division boundaries (see Chapter 3 on JPEG).

The Karhunen-Loeve transform (KLT), of all unitary transforms,¹ is recognized as giving optimal energy compaction in the first k coefficients in the transform domain. It is also very computationally intensive because of its dependence on the autocovariance matrix of the image. An acceptable approximation to the KLT is the discrete cosine transform (DCT) used in the JPEG standard. Other transforms having promising characteristics are the wavelet transform and wavelet packets. The characteristics of wavelets ease the development of compression schemes with advantages such as implementation with filter banks and exploitation of the idea of multiresolution in images, discussed in Chapter 5 on wavelets in image compression.

2.3 Quantization²

After transformation, the quantizer maps resulting coefficients into a finite set of symbols. In scalar quantization, as used in our compression scheme, the values are mapped one-to-one. Other methods include vector quantization, which maps a series of coefficients (instead of one) to a symbol, and predictive quantization, which attempts to minimize quantization error by using linear prediction of coefficients. Resulting symbols of the

¹ Unitary transforms have some desired characteristics for quantization and subband coding or multi-resolution analysis. See sections covering these topics.

² For an overview of various quantization methods in relation to subband coding, refer to [21, Appendix C].

quantizer represents discretized values of the original coefficients, and as a result, usually introduces non-linear distortion into the system. The degree of quantization depends on the given budget of bits for coding and the bit allocation process used in the compression scheme. The simplest quantization method, uniform quantization, is the assignment of a set of continuous values to a set of uniformly spaced bins. Other non-linear quantization schemes have been developed for different applications such as A-law or μ -law for audio and speech signals [21, p. 566]. A related method used to adjust uniform quantization values of an image by adding random values to them is called *improved gray-scale quantization* (IGS) [8, p.318]. This method prevents the formation of noticeable contour lines in quantized images.

In uniform quantization, statistics of the coefficients to be quantized are taken, and upper and lower limits are assigned (usually some multiple of the standard deviation). Within these limits, a coefficient value will fall into one of the bins, and the symbol associated with the bin (usually just the bin number in binary), is the result (Figure 2). In image decompression, the set of symbols is read and decoded into individual values, the result being a set of discretized coefficients, ready to be inverse transformed. The simplicity of this method makes it attractive for use in compression.

The quantization block in an image compression scheme is generally most critical to maintaining perceptual quality in compressed images since most errors are introduced in

Figure 2: Midtread uniform scalar quantization to 7 levels with stepsize of 1.

this block. Allocation of bits from a given budget is determined by a cost evaluation algorithm. Since the squared error metric is additive over the set of transform coefficients, a property of unitary transforms, it serves as a convenient and conceptually appropriate parameter to minimize. The squared error distortion measure and rate (number of bits used for quantization) have an inverse proportional relation as shown in Figure 3 and is commonly referred to as the “R-D curve.”

Figure 3: Typical rate-distortion relationship.

The two measures can be combined into a cost function by the introduction of a Lagrange multiplier, λ .

$$J(\lambda) = d + \lambda r \quad (1)$$

where $J(\lambda)$ is the Lagrangian cost function, d is squared error (distortion), and r is the bit rate.

Optimum bit allocation can be obtained by algorithmic search over different values of λ , ultimately converging on the best rate-distortion trade-off. The method is described in [18]. In addition, techniques that allow individual weighting of the costs involved in deciding bit allocation have been used in compression programs, but in the associated literature, the assignment of actual weight values has generally been avoided.³

2.4 Entropy Coding

The final step in the compression process is the reversible (or lossless) coding of symbols obtained from the quantization block. The function of the entropy coder is to efficiently map symbols into bits actually put on file or sent over a transmission line. Data compression takes place when symbols with high probability are mapped to relatively few bits while symbols with low probability are mapped with more bits. This way, the average number of bits used to code each symbol is optimized. Morse code was developed using

³ See brief discussion on Psychovisual Redundancy in [8, section 6.1.3].

the same idea, assigning short code sequences to the letters used more frequently in the English language. The classic entropy measure illustrates this concept mathematically:

$$H = -\sum_{j=1}^J P(a_j) \log_2 P(a_j) \quad (2)$$

where H is entropy in bits/symbol and $P(a)$ is the probability of a . Lower probability values contribute more to the average information output, H .

Huffman coding is widely known and is used in the JPEG standard. It is a common method of entropy coding using symbol probabilities and makes use of a binary tree to assign codes.⁴ Huffman coding, however, is only optimal if all the symbol values have a probability with integral powers of $\frac{1}{2}$ [27, p. 521]. Another method which gives coding compression superior to Huffman coding is Arithmetic Coding which achieves the theoretical entropy of any source [27, p. 521].⁵ Its adaptive version is well-suited for non-stationary sources.

The three building blocks of Figure 1 is the typical architecture of data compression systems. The model is illustrated further with a real-world compression system, JPEG, in the following section.

⁴ See Gonzales and Woods, pp. 324-341 for a treatment on entropy and information theory.

⁵ Reference includes C source code and comparisons with Huffman Coding.

Chapter 3

JPEG

The Joint Photographic Experts Group (JPEG) standard is a common example of transform coding and is simple enough to use as an illustration of the above mentioned building blocks in a basic image compression system. Each block as related to the JPEG standard is briefly described below.

3.1 Transform

JPEG uses the discrete cosine transform (DCT) which closely approximates the Karhunen-Loeve optimal decorrelating transform. Its main advantages are that, contrary to the KLT, it is not signal dependent and it is relatively easy to implement. The original image is divided into blocks of 8 x 8 pixels and the forward DCT is performed on each, resulting in corresponding 8 x 8 blocks of transform coefficients. Each block is independently quantized and coded.

3.2 Quantization

Each transform coefficient in the 8 x 8 block is quantized using a stepsize specific to its relative location on the 8 x 8 matrix using a standard quantization table. The values shown in Table 1 are (relative) stepsize values.

Table 1: JPEG Standard Quantization Table

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

As one can ascertain from the table, the lower frequency coefficient in the upper left corner of the block will be quantized with smaller steps, thereby introducing less quantization error at lower frequencies. Conversely, the higher frequency components at the lower right corner are subject to higher quantization errors. The trade-off is designed to emphasize the importance of information stored in lower frequency components and is based on tests done on the human visual system and how it perceives errors in different frequency bands. In order to constrain compression to a given bit budget, an overall scale factor is assigned to each block and is used in conjunction with the rate-distortion optimization, mentioned in Chapter 2. Finally, a symbol⁶ corresponding to the quantized coefficient is sent on to the next functional block.

⁶ A *symbol* is also known as a *label* in the literature.

3.3 Coding

JPEG makes use of a combination of the Huffman coding scheme and zero run-length coding scheme, both of which are too complicated to explain adequately here (refer to [7][18]). The main point is that the coder uses the shortest stream of bits possible to provide lossless coding of symbols. This is based on the statistics of the bit stream.

Often, the probability distribution of the symbols is not uniform. The coder will take advantage of this by using short bit streams to represent frequent symbols and longer bit streams to represent infrequent symbols. In JPEG, the coder is also optimized to take advantage of long runs of symbols representing zero, and of correlation of the DC values between adjacent 8 x 8 blocks.

3.4 Drawback of JPEG

Similar to the sensitivity of the human ear to particular ranges of frequencies, our eyes are particularly sensitive to certain characteristics such as edges, especially in the vertical and horizontal direction [25, p. 411]. Since JPEG codes each 8 x 8 block independently, mismatches between DC values in adjacent blocks produce annoying vertical and horizontal "blocking effects."

Figure 4: Illustration of blocking effects in JPEG compressed images.⁷ On the left is an enlargement of a segment of a 512x512 grayscale image. On the right is a JPEG compressed version showing signs of blocking along the edges of 8x8 pixel blocks. The arrows point to noticeable horizontal edges of pixel blocks.

Certain transforms are designed to reduce or eliminate the effects of blocking such as the Lapped Orthogonal Transform (LOT).⁸ The LOT is taken over overlapping blocks of pixels, reducing overall blocking effects. Other transforms like the wavelet transform avoids the effects of blocking altogether. Wavelets and wavelet packets are discussed in the next chapter.

⁷ Original raw images were JPEG compressed using ImageTool v. 1.27 developed by the University of Texas Health and Science Center, San Antonio. See <http://ddsdx.uthscsa.edu/dig/itdesc.html>.

⁸ See T. Nguyen's web-site on LOT (includes software): <http://saigon.ece.wisc.edu/~waveweb/QMF/software.html>.

Chapter 4

Wavelets

A *wavelet* is a function that is used in a method of signal analysis called *wavelet analysis*. Much like the sinusoids in Fourier analysis, wavelets form a set of basis functions to which a signal may be decomposed. If the signal to be analyzed is non-stationary (as most are) the periodic nature of the sinusoid places Fourier analysis at a distinct disadvantage as an analysis tool. To perform time-frequency analysis on a non-stationary signal, it must be artificially manipulated by windowing in order to use Fourier analysis (in the form of a short-time Fourier transform). Wavelet analysis requires no such manipulation of the signal. Wavelets can have finite length and therefore are ideal basis functions to represent real-world signals. Contracted and dilated copies of a basic wavelet (called the "mother" wavelet) are shifted along the time axis in order to provide the functions needed for analysis of finite signals. Superposition of wavelets make a convenient (and hopefully, compact) way to represent the original signal. It was discovered in recent years that wavelets are related to filter banks--filters that separate signals into multiple frequency

bands [10][20][22]. When filter banks are arranged in a cascading fashion, the resulting structure performs the wavelet transform [23].

It helps to review the characteristics of a relative of the wavelet transform, the Short-time Fourier Transform (STFT). A Fourier signal expansion when used on non-periodic signals give results that are not localized in time. To remedy this, the STFT creates localized Fourier bases by windowing the signal in time and applying a localized Fourier expansion. Windows must be long enough to give good frequency resolution, but at the same time, must be short enough to resolve events happening close together in time. Resolution in time and frequency is bounded by the uncertainty principle. The time-bandwidth product is bounded:

$$\Delta t \Delta f \geq \frac{1}{4\pi} \quad (3)$$

Once a window is chosen for the STFT it becomes fixed over the time-frequency plane. Thus, its resolution characteristics may not be suitable over the entire bandwidth of the signal. In wavelet transforms, the concept of scale answers the shortcomings of the STFT directly. Lower frequencies may be analyzed on a large time scale to obtain better frequency resolution, and conversely, higher frequencies may be analyzed on a smaller time scale to provide better localization in time. Figure 5 shows how the time-frequency resolution bins for the STFT and the wavelet transform differ. The STFT resolves frequency and time in fixed intervals while the wavelet transform compromises frequency

resolution at high frequencies for better time resolution, and at low frequencies, time resolution is sacrificed to better resolve frequency. Thus, while the STFT provides constant bandwidth analysis, wavelets provide constant time-bandwidth product--constant “Q”--analysis.

In contrast to the STFT which takes the inner product of an artificially windowed target signal with amplitude modulated infinite length sinusoids, the wavelet transform is done by the shifting and scaling of a single wavelet function of finite length⁹ and taking inner products with the unmodified target signal.

Figure 5: A general representation of time-frequency resolution bins for the STFT and wavelet transform. Resolution cells are of fixed size for the STFT over the entire bandwidth. Wavelet cells, on the other hand, resolve high-frequencies better in time, and low frequencies better in frequency.

⁹ Not all wavelet functions have finite length. The “Morlet” wavelet, for example, has infinite length, but it is common to analyze finite length signals with finite length wavelet functions.

In continuous time, the scaling and shifting of wavelets is illustrated in the following equations where ψ is the “mother wavelet,” the function that is scaled and shifted:

$$\psi_{kn}(t) = a^{-k/2} \psi(a^{-k}t - nT) \quad (4)$$

where a is a scale factor, k and n , integers, and T , the amount of shift.

If the wavelet is finite in length, it is said to have “compact support,” and therefore applies well to analysis of finite length, real-world signals--in our case, images. If the shifts of the wavelet function have orthogonal behavior then there are no redundancies in the coefficients of the transform, and if complete, they form an orthogonal set of basis functions. Orthogonal behavior is exploited in application of wavelet transforms to problems such as image compression.

4.1 Wavelets and Filter Banks

Wavelets were discovered to have a fundamental relationship with non-uniform filter banks [9]. Filter banks iterated in a particular way using special types of FIR filters perform the discrete-time wavelet transform [10]. Firstly, it is important that “perfect reconstruction” filters are used. Decimated filter banks using non-ideal filters typically suffer from aliasing, amplitude distortion, and phase distortion. These negative effects can be canceled by careful design of analysis and synthesis filters yielding perfect reconstruction. In a two-channel system, perfect reconstruction is achieved using Quadrature Mirror Filters (QMF) in the analysis and synthesis banks [23]. QMFs are also

referred to as “half-band” filters because the high and low pass filter pair both have cutoffs at half the usable bandwidth ($\pi/4$ radians), as illustrated in Figure 6. Secondly, the QMF FIR filter bank must be implemented in a particular way--cascaded in the manner shown in Figure 7.

Figure 6: Frequency response of a Quadrature Mirror Filter pair.¹⁰

¹⁰ Typically, QMFs have symmetric magnitude response. This pair of filters have the characteristics of linear phase which accounts for its non-symmetric appearance. The filters still exhibit perfect reconstruction.

Figure 7: A 3-level binary tree perfect reconstruction analysis bank. The cascade structure is continued until the level of desired resolution is obtained.

Many different digital filters implementing presently well-known wavelet functions have been developed.¹¹ The corresponding wavelet and scaling function for a given set of digital filter coefficients $g(n)$ -low-pass FIR and $h(n)$ -high-pass FIR is given by iterating the so-called “dilation” equations:

$$\phi(t) = \sqrt{2} \sum_{n=-\infty}^{\infty} g(n) \phi(2t - n) \quad (5)$$

$$\psi(t) = \sqrt{2} \sum_{n=-\infty}^{\infty} h(n) \phi(2t - n) \quad (6)$$

where $\phi(t)$ is the so called “scaling function” corresponding to the “mother” wavelet function $\psi(t)$.

¹¹ See [26] for the evaluation of some wavelet filters for image compression.

In applications where linear phase filter characteristics are desired, the filter bank structure given above can be used if the filter set, $H(z), G(z)$, are biorthogonal. Mathematically, orthogonality and biorthogonality may be explained as follows. For orthogonal bases the filtering effect may be described as the matrix transformation, $v = Ax$, where x is the input, A is a normalized orthogonal matrix implementing the wavelet transform, and v is the resulting coefficients. The forward transform (analysis) is the same as the inverse transform (synthesis) because of the property of orthonormal matrices:

$S = A^{-1} = A^T = A$, and $x = Sv$, where S is the inverse transform matrix. The rows of A are orthogonal to each other *and* to the columns of A ($A = A^T$). In the case of biorthogonality, $S = A^{-1} \neq A$. That is, the rows of A are *not* orthogonal to each other, but they are orthogonal to the columns of S . Therefore, in the biorthogonal case, we have a synthesis filter bank, S , that matches an analysis filter bank, A , but they are not equal to each other. A biorthogonal analysis filter set, $H(z), G(z)$, when combined with its dual synthesis filters, $\tilde{H}(z), \tilde{G}(z)$, still result in perfect reconstruction.

4.2 Wavelet Packets

Scaling is not only limited to logarithmic divisions as are those obtained by using filter banks with the binary tree structure in Figure 7. Arbitrary time scalings known as “wavelet packets” may be formed using arbitrary tree structures, suggesting that a search for the “best” tree representation for a particular signal may be possible.

In typical compression applications using wavelet packets, a full tree is calculated (Figure 9 and Figure 10) and then “trimmed” down based on some cost criterion. If the sum of the costs associated with the lower branches of any node (also called “children” or “leaves” of the node) is greater than the cost associated with the “parent” node, then the subordinate branches are trimmed and the parent node remains as part of the desired basis. If the wavelets used in producing the wavelet packets are orthogonal, the remaining combination of connected subtrees form an orthogonal basis. It is this idea of finding an optimum basis for compression of images that we pursue in detail here.

Figure 8: Example of time-frequency cells for some unspecified wavelet packet basis. Compare with Figure 5.

Figure 9: A full 3-level wavelet packet tree and commonly used terminology to describe basis tree structures.

Figure 10: Filter bank configuration that implements a fully populated 3-level binary tree.

Chapter 5

Wavelets in Image Compression

Multiresolution representation of images is intuitively related to human perceptions of images. We examine and understand images at different resolutions. We see a picture of an urban scene, then focus in on the coffee shop in the scene. Next we zoom in and notice the lady inside having a bagel . . . no cream cheese. It therefore seems like it would be beneficial to try optimization at different resolutions. Since digital images can be decomposed into successive approximations, from coarse averages to fine details, it lends itself nicely to quantization at the different resolutions. Wavelet transforms are straightforwardly extended into two dimensions of the image plane [10]:

The transform is performed on each individual row of the image, and then performed again on the resulting columns of wavelet coefficients. The resulting 2-D grid of transform coefficients are typically plotted like an image with the layout of subbands as depicted in Figure 12.

Successive wavelet transforms of each LL subband creates the layout shown in Figure 13 for representation of the wavelet coefficients. Wavelet packets are created when splitting is done in all four subbands, not just the LL band as in the wavelet case, as illustrated in Figure 14.

Analysis/synthesis filter banks provide the computational platform for image decomposition for coding, and for image reconstruction, respectively. We take the filter bank structure in Figure 10 and extend it into 2 dimensions as shown in Figure 11. We now have the tools available to create the image compression system we set to accomplish.

Figure 11: Wavelet transform extended to 2 dimensions.

Figure 12: Common layout for subbands of images.

Figure 13: Subband layout for wavelet transformed images (three levels).

Figure 14: Subband layout for wavelet packet transformed images. Each different line characteristic (i.e., thin, dotted, or bold) delineates the boundaries of a particular resolution level (bold is coarsest, dotted is finest).

Chapter 6

Developing the Image Compression System

We now have all the building blocks and techniques available to design an image compression scheme. State-of-the-art compression systems are all variations of the basic architecture given in Chapter 2 on image compression building blocks. A recent variation, the Double-Tree algorithm, discussed in [14], is conceptually promising, and is discussed below.

6.1 The Single-Tree Algorithm

To understand the Double-Tree (DT) algorithm, we first introduce the heart of it--the Single-Tree (ST) algorithm [13]. In simple terms, it is the adaptive wavelet packet that searches for the optimum basis function (in a rate-distortion sense) among all wavelet packet bases. The choice of a best basis is made among a very large number of available bases. For example, in a depth-5 2-D wavelet packet expansion, the number of available bases (all combinations of nodes forming bases) is 5.60×10^{78} [14, p. 1200], of which the wavelet basis function is only one. The ST algorithm makes use of the ideas of optimal bit

allocation in [19] and best basis search in [6] to result in the best wavelet packet basis (or in a graphical sense as in Figure 14, the best time-frequency tiling). If indeed the best basis for a particular signal or image is the wavelet representation, then the best basis algorithm will find it. The main steps of the ST algorithm are as follows:

1. Calculate the wavelet packet coefficients for the entire frequency decomposition tree.
2. For a given set of quantization grades determined in the design of the compression system, calculate the bit rates corresponding to each of the levels of quantization, and the resulting distortion (squared error) over each set of wavelet packet coefficients. This amounts to plotting discrete points on the R-D curve of Figure 3.
3. It is shown in [19] that for R-D optimality, all nodes must “operate” at a single optimal value of λ , λ^* . The optimal value of λ can be viewed as the slope at the optimum “operating” point of the node’s R-D curve. Search for the optimum slopes on the R-D curves, λ^* . The method of [19] is a convex-hull (here, the R-D curve) optimization, whereby λ^* is bracketed by high and low values of λ . The high and low values of λ are adjusted by the algorithm to converge upon λ^* . Use our choice of λ ’s to calculate the best wavelet packet basis. When λ converges to λ^* , the best wavelet packet basis (the leaves of the pruned tree after completing step 5 below) is found.
4. For a given λ chosen in step 3, calculate the Lagrangian cost, J , (equation 1), for each set of R-D points of each node and find the minimum cost. The quantizer grade

associated with the minimum cost and the R-D values for that point are optimal for the current value of λ . Populate all nodes of the tree using this method.

5. Using the pruning algorithm with Lagrangian cost replacing entropy in [6], march up the tree from the lowest branch toward the root node, pruning as determined from the following conditional statement:

If the sum of the Lagrangian costs, J , of all children nodes is less than the cost of the current node (parent node), keep the children nodes, otherwise, prune the children nodes.

The set of nodes at the lowest branches of the remaining tree is the optimal tree for this iteration. Repeat from step 3 until λ converges to a λ^* .

6.2 The Double Tree Algorithm

A novel idea related to finding the best basis in a table of wavelet packets is to split the image along *spatial* lines and search for the best basis for each image segment. First, we illustrate the idea behind spatial splitting (or spatial segmentation) with 1-D signals, then continue the discussion with a 2-D example. Following the discussion of the concepts behind the DT algorithm, we present the algorithm itself and its relation to the ST algorithm above.

6.2.1 Conceptual Discussion

Spatial Segmentation in One-Dimension

A normal speech signal by its nature is highly non-stationary. In normal speech, voiced sounds like the vowels “a” in “arch” or “e” in “elephant” have significantly different harmonic content than unvoiced sounds like the “s” in “stew” or “ch” in “cheese”.

Conceptually, it would be advantageous to separate a speech signal into sections of voiced sounds and sections of unvoiced sounds. That way we can adapt our analysis of each segment according to its characteristics. This is the essential motivation for spatial segmentation of signals in a compression application. Figure 15 and Figure 16 illustrate how best bases may be different for different segments of a speech signal. Best wavelet packet bases are calculated for the segments A and B individually, resulting in an optimal basis function for each segment.

Figure 15: A speech signal that has been spatially divided into two segments: A and B. Each segment has unique characteristics.

Figure 16: Possible best bases for segments A and B of the speech signal in Figure 15.

Spatial Segmentation in Two-Dimensions

Extending the segmentation discussed above to 2-D images is intuitively simple. Recall the frequency splitting tree in the discussion on wavelet packets in Chapter 4 and the discussion on subbands in Chapter 5. An image repeatedly divided into equal segments in space is analogous to the division of an image into a tree of frequency subbands. The original image divided into quarters would comprise the 1st level of the tree, and each quarter image divided into quarters again would comprise the 2nd level, and so on, as Figure 17 illustrates. The resulting tree structure is known as a *quadtree* data structure [14, p.1200]. To simplify the terminology hereafter, we will refer to the image splitting tree as the spatial tree, and the frequency decomposition tree as the frequency tree. Also, the images that result from splitting the original image into quarters, sixteenths, etc. will be

referred to as subimages of the spatial tree, or simply, subimages. The black lines in Figure 17 show one possible outcome of a best basis search of a spatial tree. The algorithm divides the image into sections, fills the spatial tree, then searches for the best wavelet packet basis within each section or subimage. Hopefully, using a best basis for each section of the image will cost less bits than using a single wavelet packet basis that tries to optimize the entire image. If the image contains distinct sections with like characteristics (e.g., homogeneously smooth, exhibits like patterns), then splitting the image along those boundaries should increase the likelihood of finding a very good wavelet packet basis specifically for that part of the image.

Consider the example of analyzing a tropical beach image. One wavelet packet basis which may be optimal in one area of the image consisting of, say clear sky, probably would not be optimal for another area of the same image consisting of palm trees. The Double-Tree algorithm finds the optimum basis function for both. The DT essentially is a generalization of the ST. The option exists for the spatial tree to degenerate to one node--the node containing the original image--on which the ST algorithm is applied. From a strictly algorithmic point-of-view, we already have the framework necessary to optimize spatially from our discussion of the ST algorithm. A best basis search over the spatial tree, comparing the overall costs of optimum wavelet packet bases for the subimages, would give us the optimum spatial tree. And we already have the optimum frequency trees to go with the surviving branches of the spatial tree. Of course, this requires many

more calculations since the best wavelet packet basis must be found for each subimage in the spatial tree. In the compression program used to obtain results in the following chapter, a depth-3 spatial tree is formed and searched. To populate all the nodes of the depth-3 spatial tree would require 85 iterations of the ST algorithm. Using the recursion equation found in [14], the number of possible spatial segmentation combinations is 83522, of which, only one is optimum.

Note for example, in Figure 18 we see that the subimage in the upper left corner (outlined in black) is the 2nd iteration of spatial segmentation (the first iteration being cutting the image into equal quarters. Within that subimage, the white lines show the subimage was divided into 4 subbands, that is, one iteration of frequency decomposition.

Figure 17: Example of splitting in the 2-D spatial tree. The black lines indicate the boundaries of the subimages.

Figure 18: One possible outcome of optimal wavelet packet bases for the image as divided up in Figure 17. The white lines show layout for frequency subbands (see Chapter 5 on wavelet packets).

6.2.2 The Algorithm

In terms of the ST algorithm, the DT algorithm is simply the repeated application of the ST over dyadic splittings of the original image. The steps are as follows:

1. Calculate the R-D characteristics for the wavelet packet tree, as was done in the ST algorithm, for each subimage of the spatial tree.¹²
2. Generate the wavelet packet coefficients of the frequency tree for each subimage.
3. Perform the ST algorithm on each subimage of the spatial tree. The result is the optimal wavelet packet basis for each subimage.
4. Using the optimum rate and distortion measures for each optimum wavelet packet, fill in each node of the spatial tree. Prune the spatial tree in the same manner the frequency tree was pruned in the ST algorithm, comparing the Lagrangian costs for each node of the spatial tree. In this step, we fix λ and compute the Lagrangian costs with the rate and distortion measures we stored in the spatial tree nodes.

The resulting pruned tree is the best spatial segmentation of the original image, that is, the best set of subimages, and optimum wavelet packet bases to go along with each subimage.

¹² In our implementation of the DT algorithm, the spatial tree will be of depth-3, and each subimage will be decomposed in frequency to depth-4, except the deepest level of the spatial tree, which will be decomposed in frequency to depth-3. For a 256x256 image, the smallest subimage will be of size 64x64, and the frequency decomposition to depth-3 will result in the smallest coefficient block size of 8x8. This is the limit of decomposition used in the experiments of [14]. Also, by limiting the frequency decomposition depth, we limit ourselves to practical time and memory limitations in computer calculations.

6.3 Program Components

The image compression program investigated for this thesis implements the Double-Tree algorithm of [14]. The new compression program developed for this thesis leverages software in the “Baseline Wavelet Transform Coder Construction Kit” [7] designed to assist researchers in their study of image compression with wavelets. The kit was found to be very helpful. Good use was made out of the modules (C++ classes) of the distribution for system-level input/output and quantization and entropy coding. Nevertheless, significant effort was required to develop the framework necessary to implement wavelet packets and best-basis searching, which amounted to the development of approximately 2100 new lines of code. See Appendix B for further details on the code contribution of this thesis. The following subsections describe the key image compression components used in the compression program developed in this thesis.

6.3.1 Wavelet Transform

“Antonini 7/9” biorthogonal wavelets [1][7] were used in implementing the wavelet transform. Biorthogonal filters have the advantages of having linear phase, which is desired to minimize “ringing” effect [11] in images, and at the same time are very nearly orthogonal in behavior.

6.3.2 Quantizer

A simple uniform quantizer is used, mapping each coefficient to its own symbol, which is later entropy coded. Another quantizer used in the software distribution is the “multi-

layer embedded” quantizer which is a type of predictive quantizer [7, “quantizer.cc”]. It is memory intensive as it requires the formation of histograms and therefore is only used in the original wavelet transform coder given in [7]. This original program configuration is referred to as W (Layer), and for the purposes of the comparison tests in Chapter 7, is considered best among wavelet transform image coders. Ten different quantization grades (or stepsizes) are calculated for the R-D curve population portion of the ST algorithm.

The method of [19] to optimize the rate-distortion trade-off in bit allocation (finding λ^* of section 6.1) is used, supplemented by the “greedy augmentation” algorithm [25, p. 386]. This addition allocates bits left over from the first bit allocation process to where they will do the most good, in terms of further lowering distortion.

6.3.3 Entropy Coding

Arithmetic coding, as presented in [27] is used for symbol coding.¹³

¹³ The Arithmetic Coding module by John Danskin is available separate from the “Baseline Wavelet Transform Coder Construction Kit” at <http://www.cs.dartmouth.edu/~jmd>.

Chapter 7

Compression Experiments

Seven images were used to conduct image compression quality and comparison tests. A wavelet transform based compression system was tested against an adaptive wavelet packet compression system with and without the Double-Tree algorithm. JPEG compressed images were used as a baseline. Both quantitative measurements (peak signal-to-noise ratio) and qualitative measures (based on the author's perceptions) were taken.

Note that familiar quality measures such as the statistical mean squared error or peak signal to noise ratio measures, as used here, may be misleading since the human visual system has higher sensitivity to certain image characteristics over others. A better measure would be to develop a distortion model that gives weights to distortion figures based on perceptual criteria. Some examples of the disparity between perceptual quality and squared error based measures can be seen in results obtained below.

A representative set of images are given in Appendix A. All data plots and discussion of the results are given in this section. The results obtained are among the best coding results available for these images.¹⁴ No attempt was made to make direct comparison of the results obtained here with those achieved in the literature. The sensitivity of the algorithm to the configuration used in the different parts of the compression scheme (i.e., quantization method, entropy coding method, filters used in filter banks, etc.) precludes meaningful comparisons [9, p. 3356]. Also, our interest lies in the performance of the algorithm and not in the comparative performance of the whole compression scheme.

7.1 Original Test Images

All test images are 256-level grayscale, standard test images. They were chosen because of their high quality and variety, ranging from fine details to regular patterns to smooth features. The test images and a brief description of their characteristics follow.

¹⁴ See the table of coding results at UCLA's Image Processing Lab.
http://synergy.icsl.ucla.edu/~ipl/psnr_results.html.

Figure 19: "Lena" image (512x512, 256-level grayscale).

This is the ubiquitous test image common to almost all literature on image compression.

It has the full range of characteristics desired for comparison testing: detailed lines in the feather and hair, patterning on the hat, and gradual grayscale contour around the shoulder.

Figure 20: "Barbara" image (512x512, 256-level grayscale).

This image is a bit busier than the previous. Fine lines in patterning on the trousers, table cloth, headdress, and wicker chair make this a tough challenge for any image compression program.

Figure 21: "Goldhill" image (512x512, 256-level grayscale).

On the high detail side of the spectrum, this image showcases highly contrasted faces and regular lines of the houses, fading into the hazier countryside and sky.

Figure 22: "Boat" image (512x512, 256-level grayscale).

A complicated image that has open sky in the top two corners. This image may benefit from spatial splitting.

Figure 23: "Peppers" image (512x512, 256-level grayscale).

This image has more gradual grayscale variations than it has high contrasting edges. The wavelet transforms coders should do well on images like this one.

Figure 24: "Building" image (256x256, 256-level grayscale).

DT spatial splitting may optimize performance in the largely white area above the building but the noise in the image may cause problems.

Figure 25: "Camera" image (256x256, 256-level grayscale).

Another image with large areas of plain sky. This image may also stand to benefit from spatial splitting.

7.2 Quantitative Results

The set of standard images above were compressed using 5 different compression programs--two using the straight wavelet transform, one using adaptive wavelet packets, one using wavelet packets and the Double-Tree algorithm, and finally, JPEG as a control data set. Results given here were found to be in sync with results found in recent literature [14][28][29].

Table 2: Approximate times of one compression-decompression cycle of a 512x512, 256-level grayscale image.¹⁵

Compression Method	Abbr.	Time
Wavelet (Layer Coder)	W (Layer)	9 min.
Wavelet (Mono Coder)	W (Mono)	40 sec.
Wavelet Packet (Level 5)	WP	7 min. 15 sec.
WP w/ Double-Tree	WPDT	33 min.

Table 2 shows the wide range of times needed by each program. As we increase the depth of the wavelet packet tree, the times grow exponentially as expected. A wavelet packet tree with a depth of 2 has 21 nodes; depth of 3 has 85 nodes; depth of 4 has 341 nodes; and a depth of 5 has 1365 nodes.

In the following graphs, results of the four wavelet transform based compression schemes and JPEG (see "Notes on Compression Tests" below) are plotted for each test image.

¹⁵ Based on the program detailed in Appendix B. The program compiled with Microsoft Visual C++ v.4.0 was run on a 100 MHz Pentium, with 64 MB RAM.

The graphs below plot the target bit-rate for the compression in bits per pixel versus peak signal-to-noise ratio (PSNR), the common quantitative measure of compression quality.

Four compression ratios are used: 8:1, 16:1, 32:1, and 64:1. Abbreviations for the compression schemes are as given in Table 2 and are used throughout this chapter.

Notes on Compression Tests

- JPEG bit rates are approximate. The JPEG program used does not allow specification of exact bit budgets, therefore, JPEG bit rates are rounded to the nearest of the standard rates used in these experiments. The JPEG plot is only meant to be for rough comparison.
- The Layer Coder (See Appendix B) requires intense calculations. It was not activated in the wavelet packet coders because of memory and time constraints.

Figure 26: Results of compression for "Lena" image.

The adaptive wavelet packet based compression schemes are clear winners at low compression ratios. The wavelet transform as determined from tests in [14] is near optimal for the "Lena" image but the wavelet packets adapted well. Agreement between the wavelet packet coder (WP) and the wavelet packet coder with the Double-Tree algorithm (WPDT) suggest that there was no clear advantage by splitting in the spatial domain. It must be noted, though, the WP coder searches to depth 5 while the WPDT coder searches only to depth-4 in spatial levels 0 - 2, and depth-3 on the lowest branches of the spatial tree, depth-3.¹⁶

¹⁶ See Chapter 6 and the footnote for that section.

Figure 27: Results of compression for "Barbara" image.

Of the five 512x512 images, the "Barbara" image, having the lowest average PSNR, gives the most trouble in terms of compression quality. Fine striped patterns throughout the image proved difficult to manage for all compression programs, however, the adaptive wavelet packets did very well, out-doing even the calculation intensive Layer Coder by a good margin. Again, the wavelet packet based programs converged to approximately the same basis; their results are almost identical over the range of bit values. Because of the similarity of results, it seems the Double-Tree spatial segmentation again had no effect on this image--no particular spatial segmentation (given restraints of the dividing algorithm) was advantageous over optimizing the image as a whole.

Figure 28: Results of compression for "Goldhill" image

"Goldhill" being the most "busy" of the test images served as a fairly equal challenge to each of the coders. The results of the wavelet packet based coders converged to identical solutions, and all the coders are similar in trend. Although WP and WPDT has a slight edge over the wavelet coders, W (Layer) and W (Mono), no real clear best compression scheme falls out from this example.

Figure 29: Results of compression for "Boat" image.

Here, the WP and WPDT have a slight edge over the other coders at low compression ratios. The open skies in the top half of the image apparently gave the Double-Tree coder no significant advantage over its non-Double-Tree counterpart.

Figure 30: Results of compression of "Peppers" image.

Surprisingly, the wavelet coders performed well in this image. The gradual contours over the peppers in the image seem to have favored the schemes that concentrate on low frequency resolution such as the two wavelet coders.

Figure 31: Results of compression of "Building" image.

This image caused the most erratic behavior among the different coders. It would seem that the image would be a good target for spatial optimization but WPDT only has the advantage at high compression ratios. Apparently the noise in the image is enough to counteract any advantage spatial segmentation provides. Even at the lowest bit rates, the WP coder fares better than WPDT. Although from the graph it seems JPEG performs as well as the other coders at 1 bpp and 0.25 bpp, the JPEG coder actually performed worse. For example, the JPEG compression rates plotted at 1, 0.5, and 0.25 bpp above were actually 1.16, 0.52, and 0.35 bpp, respectively. See "Notes on Compression Tests" above.

Figure 32: Results of compression of "Camera" image.

Only in this image and the previous did WP and WPDT converge with significantly different solutions. WPDT wins over WP but doesn't keep up with the best wavelet coder W (Layer). Again, where it seems JPEG performs as well as the other coders, it is actually performing worse. Actual JPEG compression rates plotted at 1, 0.5, and 0.25 bpp above were 1.16, 0.50, and 0.33 bpp, respectively.

7.3 Subjective Evaluations

No check off list is used in this evaluation. Only brief comments will be made to point out major degradation or annoying artifacts. An attempt is made to pick out one compression method that beats the others. The subjective results are compared with quantitative results given above and correlation or non-correlation with the data is pointed out.

Subjective bests are indicated in **boldface** type. The different compression methods are referred to by their abbreviation as listed in Table 2 above. A brief summary of observations is given at the end of the section.

7.3.1 “Lena” Image

1.0 bpp (8:1 compression)

W (Layer) - Some effects of false contouring around the shoulder is noticeable but very good overall.

W (Mono) - Starting to show some graininess around the face but very good overall.

WP, WPD - Very good.

0.5 bpp (16:1 compression)

W (Layer), W (Mono) - False contouring all over the image.

WP, WPD - False contouring around shoulders but not as extensive as the wavelet transform coders.

0.25 bpp (32:1 compression)

W (Layer) - Facial features still looks good, i.e., they are smoother than on the other coders. Details in the hat are lost.

W (Mono) - Facial features start to muddy.

WP, WPD - Some patchy ripples all around the image. Facial features look good but show evidence of rippling.

0.125 bpp (64:1 compression)

W (Layer) - Facial features still look fine. Details around the eyes are still well-defined.

W (Mono), WP, WPDT - Blurring around edges of eyes with uniform blotchiness all around the image.

7.3.2 “Barbara” Image

1.0 bpp (8:1 compression)

Remarkably, all images look equally excellent. All patterns on the tablecloth, trousers and head scarf are sharp and unblemished. Images look as good as original, which is not expected as this image has the lowest PSNR for all compression routines among the three test images. It could be because the original was a bit out of focus in the first place.

0.5 bpp (16:1 compression)

W (Mono) - Has least amount of blending between arms and trousers.

W (Layer), WP, WPDT - Some ringing apparent along edges of table legs. Slight overall blotchiness. Trousers are starting to lose the clarity in their patterning.

0.25 bpp (32:1 compression)

W (Layer), W (Mono) - Table cloth has been washed of its pattern as has the trousers and head scarf. The whole image is badly blurred.

WP, WPDT - Images are very similar. A small part of the tablecloth is washed out but the stripes on the head scarf are still noticeable. The images are slightly sharper than the wavelet compressed images.

0.125 bpp (64:1 compression)

W (Layer) - Image is severely degraded. The face is badly muddled. The trousers and arm has merged together. The tablecloth and the toy on the table are badly blurred.

W (Mono) - Trousers and arm are blurred as in the layer coder but face has best definition. This image, surprisingly, looks the best.

WP, WPDT - Images exhibit all-around muddiness.

7.3.3 “Goldhill” Image

1.0 bpp (8:1 compression)

W (Layer), W (Mono) - On close inspection, slight blurring of details on the roofs is noticeable.

WP, WPDT - Excellent image.

0.5 bpp (16:1 compression)

W (Layer), W (Mono) - Roofs have lost their definition.

WP, WPDT - Window edges are still excellent. Can still make out shingles on roofs.

0.25 bpp (32:1 compression)

W (Layer), W (Mono) - Windows are starting to get washed out.

WP, WPDT - Some ringing in window panes. Roof patterns are still holding out.

0.125 bpp (64:1 compression)

W (Layer) - Poor image but the sides of the houses are less blotchy than the others. Image best among the batch of very poor representations.

W (Mono) - Very poor overall.

WP, WPDT - Vertical streaking effect along the faces of the houses are evident.

7.3.4 “Boat” Image

1.0 bpp (8:1 compression)

All images are of very good quality. It is hard to tell them apart.

0.5 bpp (16:1 compression)

W (Layer), W (Mono) - Boat rigging is starting to blur and blend into the sky.

WP, WPDT - Very good results. Boat rigging is still well-defined.

0.25 bpp (32:1 compression)

W (Layer), W (Mono) - Boat masts are starting to blur. Straight lines of the mast are starting to break.

WP, WPDT - Definition is becoming lost in the masts but overall, these images are crisper than the wavelet coders.

0.125 bpp (64:1 compression)

W (Layer), W (Mono), WP, WPDT - All images are similar but the wavelet packet coders have a slight edge in crispness.

7.3.5 “Peppers” Image

1.0 bpp (8:1 compression)

All images are of very good quality. Lack of sharp definition in the original image makes this image a relatively easy one to compress. Although the PSNR averages are not particularly high, the final reconstructed image does not suffer from many apparent artifacts.

0.5 bpp (16:1 compression)

W (Layer), W (Mono) - False contouring is evident on the large dark pepper in the center of the image.

WP, WPDT - Contouring in these images look more natural than in the wavelet coded ones.

0.25 bpp (32:1 compression)

W (Layer) - This image looks best because it has a natural smoothness in the artifacts.

W (Mono), WP, WPDT - Images are very similar.

0.125 bpp (64:1 compression)

No one image stands out as best. Rippling effect usually appears earlier for the wavelet packet images, but is not evident except in this highly compressed image.

7.3.6 “Building” Image

1.0 bpp (8:1 compression)

W (Layer) - The original image was very grainy. This compression method appears to smooth out the graininess faster than the other methods and therefore looks better.

W (Mono), WP, WPDT - Images are similar but have more detail than W (Layer). The detail, here however, is noise.

0.5 bpp (16:1 compression)

W (Layer), W (Mono) - Sharp edges of the windows are smudging.

WP, WPDT - Brick portion of image is still defined.

0.25 bpp (32:1 compression)

W (Layer), W (Mono) - Mono coder has slight edge in definition over the Layer coder.

WP - Ringing is evident along the windows of the building.

WPDT - Overall, gives the sharpest image.

0.125 bpp (64:1 compression)

W (Layer) - Best overall for the wavelet transform based coders. The blurring effect is less objectionable than artifacts in the other images

W (Mono) - Blurry and not quite as good as W (Layer).

WP, WPDT - Bad ringing effect along the windows.

7.3.7 “Camera” Image

1.0 bpp (8:1 compression)

All have similar artifacts of false contouring in the background and “halo” effect around the man’s coat.

0.5 bpp (16:1 compression)

W (Layer), W (Mono) - Losing detail in the background. The buildings in the background are all blurred.

WP, WPDT - The background looks crisper. The control tower is still well-defined.

0.25 bpp (32:1 compression)

W (Layer), W (Mono), WP, WPDT - More of the same problems as in 16:1 compression.

0.125 bpp (64:1 compression)

W (Mono) - Poor image, but the man’s face is better defined.

W (Layer), WP, WPDT - All very poor.

7.3 8 Summary of Subjective Evaluations

Subjective image evaluation stayed roughly close with the plotted error values. Similarity among images confirmed where the quantitative results tracked closely together. Some interesting deviations from this trend of correlation were noted, however:

1. Absolute value of the PSNR was not a good measure of quality. Relative values conveyed more information. As an example, even though the “Lena” image PSNR numbers were a good 3 dB higher than “Barbara” and “Goldhill,” flaws were apparent

in its 8:1 images that were not as noticeable in “Barbara” and “Goldhill” 8:1 images. The “Peppers” image, as another example, retained very good quality down near 29 dB PSNR, while the “Lena” image degraded very noticeably at above 30 dB.

2. At extreme compression ratios where the image is severely degraded, even clear relative differences in PSNR become useless measures of perceptual quality. In two examples above, one compression method that was “beating” the others at lower compression ratios was suddenly “overtaken” by another method. At extreme compression, it seems a level of randomness in the distortion actually serves to clarify certain image features, at least in the perceptual sense.
3. The wavelet and wavelet packet coders optimize different aspects of images. The wavelet packet coders are better at preserving high frequency in the image since they allow optimization at high frequencies as well as low frequencies. On the other hand, the two wavelet-based coders tend to smooth high frequency content. These contrasting effects have different qualitative results depending on the image content. For example, in a noisy image like “Building,” the smoothing effect of the wavelet coders are beneficial to the overall aesthetics of the image after compression by ridding the noise, while the wavelet packet coders, in trying to preserve details, only work to keep what was deemed annoying in the first place. In addition, inaccuracies in reconstructing the noise (wavelet packet coders) could actually serve to increase distortion over what would have been obtained by averaging (wavelet coders).

It is clear that quantitative measurements have their limitations in evaluating compression effectiveness. Overall, the adapted wavelet packet coders WP and WPDТ performed best quantitatively, and in subjective tests at moderate compression levels. The high frequency analysis at different resolutions left out by the wavelet transform proved to be more detrimental in subjective tests at moderate compression levels.

Chapter 8

Conclusions

Results are shown to be highly dependent on the image content, as expected. Although no one image compression system evaluated here could be labeled as having outstanding performance for each of the images, the adaptive wavelet packet coders have a slight edge over the wavelet coders overall. Indeed, the published results in comparisons of wavelet transform-based compression systems vary only slightly in quantitative measure among the all the systems tested. As an example, the results in [14] show rather marginal improvements in the Double-Tree over the Single-Tree and of the Single-Tree over wavelet coders. The more apparent and more important measures of improvement of the wavelet packet coders--subjective results--is unfortunately hard to quantify fairly. We can conclude, though, that our results provide an argument for adaptive compression schemes. The rigidity of the wavelet coders leave them short on performance.

The Double-Tree algorithm proved to be effective in some images. Although a blocky image like "Building" would seem ideal for the DT, the algorithm only performed fairly.

However, in the “Lena” and “Barbara” images, the algorithm performed quite well. It is very difficult to tell which image may benefit more from spatial segmentation. Because of its restrictiveness in its ability to segment images and its processing requirements, it is probably not a good candidate for a general image compression scheme but would do quite well in specialized cases. The authors of the original paper on the DT algorithm [14] have sought to refine a quantization technique combining ideas from R-D optimization and zero-trees called space-frequency quantization (SFQ) [29][30][31]. In their recent papers, the authors of SFQ briefly described using SFQ in the DT scheme. The adaptive wavelet packet coder without DT, though, is the best all-purpose coder. It is reasonably fast while obtaining very good results on all the images up through to moderate compression levels.

JPEG is inferior to any of the wavelet transform compression schemes for our selection of large (512x512) images presented here. Subjectively, JPEG blocking effects may be masked by regular structures in the image (this is probably more true in the “Building” image than any other) but in most natural images blocking is prominent at moderate to high compression. Overall superiority of the wavelet transform based coders in compressing any image (detailed or not) in addition to having results free of blocking effects, put wavelets on top as the better performer.

Of late, the literature has been concentrating on the area of optimizing quantization. As mentioned above, it's probably the most important piece of the image compression system. Regardless of the need to emphasize quantizer technology, however, further study should be done in the area of developing alternatives to the PSNR measure of quality. Perhaps a composite measure of PSNR and subjective measures would be more appropriate.

Another possibility is to categorize the images themselves into characteristics that affect the compression process. Each compression scheme could then be evaluated depending on its performance on each of the sets of images in a particular category, and an overall performance factor may be calculated. An additional topic requiring further study is the trimming wavelet packet trees without calculating each node and comparing costs of the parent node and children nodes. Success in finding a efficient trimming method would save calculations by eliminating the need to populating the entire wavelet packet tree.

Also, the area of perceptual weighting of distortion values should be explored in greater detail. As shown in the subjective test section, the mean squared error measure has limited value in terms of visual perception of quality. Finally, the spatial and frequency optimization afforded by the Double-Tree algorithm should be explored in greater detail, starting with categorization of images which the Double-Tree may have significant advantages.

In spite of all the inconsistencies found here, the future of image compression seems well entrenched in the world of wavelets and wavelet packet analysis. From the subjective

results gotten here, and regardless of the PSNR data, we conclude that wavelet packets is still the most promising field of study for image compression.

References

- [1] M. Antonini, M. Barlaud, P. Mathieu, I. Daubechies, "Image Coding Using Wavelet Transform," *IEEE Transactions on Image Processing*, vol. 1, pp. 205-220, April 1992.
- [2] V. Bhaskaran, K. Konstantinides, *Image and Video Compression Standards*, Kluwer Academic Publishers, 1995.
- [3] A. Bruce, D. Donoho, H. Gao, "Wavelet Analysis," *IEEE Spectrum Magazine*, pp. 26-35, October 1996.
- [4] K. Castleman, *Digital Image Processing*, Prentice Hall, 1996.
- [5] Y. T. Chan, *Wavelet Basics*, Kluwer Academic Publishers, 1995.
- [6] R. Coifman, V. Wickerhauser, "Entropy-based Algorithms for Best Basis Selection," *IEEE Transactions on Information Theory*, vol. 38, pp. 713-718, March 1992.
- [7] G. Davis, "Baseline Wavelet Transform Coder Construction Kit," version 0.3, C++ source code distribution, <http://www.cs.dartmouth.edu/~gdavis>, 11 September 1996.
- [8] R. Gonzales, R. Woods, *Digital Image Processing*, Addison-Wesley, 1992.
- [9] C. Herley, J. Kovacevic, K. Ramchandran, M. Vetterli, "Tilings of the Time-Frequency Plane: Construction of Arbitrary Orthogonal Bases and Fast Tiling Algorithms," *IEEE Transactions on Signal Processing*, vol. 41, pp. 3341-3359, December 1993.
- [10] S. Mallat, "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674-693, July 1989.
- [11] T. Nguyen, P.P. Vaidyanathan, "Two-Channel Perfect-Reconstruction FIR QMF Structures Which Yield Linear-Phase Analysis and Synthesis Filters," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, pp. 676-690, May 1989.
- [12] R. de Queiroz, K. Rao, "Time-Varying Lapped Transforms and Wavelet Packets," *IEEE Transactions on Signal Processing*, vol. 41, pp. 3293-3305, December 1993.
- [13] K. Ramchandran, M. Vetterli, "Best Wavelet Packet Bases in a Rate-Distortion Sense," *IEEE Transactions on Image Processing*, vol. 2, pp. 160-176, April 1993.

- [14] K. Ramchandran, Z. Xiong, K. Asai, M. Vetterli, "Adaptive Transforms for Image Coding Using Spatially Varying Wavelet Packets," *IEEE Transactions on Image Processing*, vol. 5, pp. 1197-1204, July 1996.
- [15] L.-M. Reissell, "Multiresolution and Wavelets," from "Wavelets and their Applications in Computer Graphics," *Siggraph '95 Course Notes: #26 Wavelets*.
- [16] O. Rioul, "A Discrete-Time Multiresolution Theory," *IEEE Transactions on Signal Processing*, vol. 41, pp. 2591-2606, August 1993.
- [17] O. Rioul, M. Vetterli, "Wavelets and Signal Processing," *IEEE Signal Processing Magazine*, pp. 14-38, October 1991.
- [18] K. Sayood, *Introduction to Data Compression*, Morgan Kaufmann, 1996.
- [19] Y. Shoham, A. Gersho, "Efficient Bit Allocation for an Arbitrary Set of Quantizers," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, pp. 1445-1453, September 1988.
- [20] G. Strang, T. Nguyen, *Wavelets and Filter Banks*, Wellesley-Cambridge Press, 1996.
- [21] F. Stremler, *Introduction to Communication Systems*, Addison-Wesley, 2nd ed., 1982.
- [22] P.P. Vaidyanathan, "Quadrature Mirror Filter Banks, M-Band Extensions and Perfect- Reconstruction Techniques," *IEEE ASSP Magazine*, pp. 4-20, July 1987.
- [23] P.P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, 1993.
- [24] M. Vetterli, D. Le Gall, "Perfect Reconstruction FIR Filter Banks: Some Properties and Factorizations," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, pp. 1057-1071, July 1989.
- [25] M. Vetterli, J. Kovacevic, *Wavelets and Subband Coding*, Prentice Hall, 1995.
- [26] J. Villasenor, B. Belzer, J. Liao, "Wavelet Filter Evaluation for Image Compression," *IEEE Transactions on Image Processing*, August 1995.
- [27] I. Witten, R. Neal, J. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, June 1987.
- [28] Z. Xiong, K. Ramchandran, M. Orchard, "Joint Optimization of Scalar and Tree-structured Quantization of Wavelet Image Decomposition," *Proceedings of the 27th Annual Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, CA, November 1993.
- [29] Z. Xiong, K. Ramchandran, M. Orchard, "Space-frequency Methods in Image Compression," *SPIE*, Denver, pp. 332-343, 1996.

- [30] Z. Xiong, K. Ramchandran, M. Orchard, "Space-frequency Quantization for Wavelet Image Coding," *SPIE*, Denver, pp. 388-399, 1996.
- [31] Z. Xiong, K. Ramchandran, M. Orchard, K. Asai, "Wavelet Packets-Based Image Coding Using Joint Space-Frequency Quantization," *Proceedings ICIP '94*, Austin, TX, vol. III, p324-328, November 1994.

Appendix A

Sample Images from Compression Tests

Images follow this page.

Compression Ratio: 32:1 (0.25 bpp)

Compression Method: WP 5

Comments: Even at moderate compression rates, the wavelet packet transform coder gives very good results. The image has not lost much detail. Subjectively, the W (Layer) coder wins out because the facial features in that compressed image are smoother than in this image.

Compression Ratio: 64:1 (0.125 bpp)

Compression Method: W (Layer)

Comments: The wavelet representation is purported to be optimal for the “Lena” image. As a result, the high-quality wavelet coder does best. The wavelet coder seems to preserve smooth feature like the face here better than the other coders.

Compression Ratio: 64:1 (0.125 bpp)

Compression Method: WPDT

Comments: This image is compressed at the same rate as the previous but emphasizes more detail. The smoothness of the facial features are sacrificed, however, and since the lighter color of the face dominates the image, the overall subjective evaluation favors the wavelet coder W (Layer).

Compression Ratio: 33:1 (~0.24 bpp)

Compression Method: JPEG

Comments: Annoying artifacts caused by blocking is apparent even at moderate compression ratios.

Compression Ratio: 32:1 (0.25 bpp)

Compression Method: WPDT

Comments: The WPDT coder does the best in quantitative measures. The striped pattern which is washed out in the wavelet coders are still visible here.

Compression Ratio: 64:1 (0.125 bpp)

Compression Method: W (Mono)

Comments: At extreme compression ratios, error measurements become less useful when comparing images at the same compression ratio. This one looks the best despite not “winning” at lower compression.

Compression Ratio: 24:1 (~0.33 bpp)

Compression Method: JPEG

Comments: Stripe pattern in trousers and head scarf are lost due to blocking.

Compression Ratio: 32:1 (0.25 bpp)

Compression Method: WPDT

Comments: Details in the roof are still apparent but the faces of the houses are starting to develop vertical streaks. The countryside is starting to smudge. This image shows how well the wavelet packet coders do in preserving details.

Compression Ratio: 64:1 (0.125 bpp)

Compression Method: W (Layer)

Comments: All images are similar at this compression ratio but this one has best appearance as the sides of the houses here suffers the least of the rippling effect.

Compression Ratio: 30:1 (~0.27 bpp)

Compression Method: JPEG

Comments: Blocking is not as noticeable on the houses because of the regularity of the original image there, but the countryside is chopped up by the effect.

Compression Ratio: 16:1 (0.5 bpp)
Compression Method: W (Layer)

Comments: The rigging of the boat is starting to blur and blend into the background sky.

Compression Ratio: 32:1 (0.25 bpp)

Compression Method: WPD

Comments: For the DT coder, the rigging of the boat is still visible though a ringing effect is becoming very noticeable.

Compression Ratio: 64:1 (0.125 bpp)

Compression Method: WPDT

Comments: Though this coder does especially well with details, even the smooth surfaces and gradual graduations in grayscale is well preserved in this image. The PSNR is only slightly above 29 dB yet the overall quality of the image is good. The artifacts introduce in this context are not very objectionable.

Compression Ratio: 32:1 (0.25 bpp)

Compression Method: WPD

Comments: One would expect spatial segmentation to be beneficial here, but apparently, the noise present in the image posed a tough challenge to the coder. There were, however, some subimages that were coded with very high PSNR, but overall, the accuracy of the coding is only average.

Compression Ratio: 32:1 (0.25 bpp)
Compression Method: W (Mono)

Comments: Oddly, the wavelet coders served to clear up the image by smearing out the grainy noise. In doing so, though, some pertinent detail in the brick part of the building is lost.

Compression Ratio: 40.9:1 (~0.20 bpp)

Compression Method: JPEG

Comments: The regularity and blockiness of the original image allows JPEG to add its characteristic artifacts without appreciable effect on the image.

Compression Ratio: 32:1 (0.25 bpp)

Compression Method: WPDT

Comments: Some of the detail in the background building is still evident. The step from the gray sky to the black coat causes the ringing effect all around the outline of the man in the picture.

Compression Ratio: 32:1 (0.25 bpp)

Compression Method: W (Mono)

Comments: The control tower in the background is completely washed out as is the grassy part of the image at the bottom. Results here are not as crisp as for the wavelet packet coders.

Compression Ratio: 42.5:1 (0.19 bpp)

Compression Method: JPEG

Comments: The artifacts from the JPEG compression are not very apparent in the small image. JPEG gets most of its coding gain in the dark coat and the gray sky.

Appendix B

Image Compression Program

Software Description

This program leverages many of the program modules in the "Baseline Wavelet Transform Coder Construction Kit" (BWTC) version 0.3 by Geoff Davis.¹⁷ Some of the modules were incorporated into the body of the author's program. New C++ classes were developed by the author to implement the core wavelet packets compression program. A description of each of the author's modules and source code follow.

BasisTree2D Class

This data class maintains a 2D basis tree. It provides functions to track the status of a 2D basis tree, controls indexing of tree nodes, and trims or fills the tree as required. Extensive use is made of this class to step through tree nodes for cost comparisons, and to store the optimum spatial and frequency basis trees.

¹⁷ <http://www.cs.dartmouth.edu/~gdavis>

WPTransform Class

This class performs the compression and reconstruction of the image (or subimage) given to it. It uses the `BasisTree2D` data class to keep track of its optimum tree. The class uses the quantizer, wavelet transform, entropy coder, coefficient storage, bit allocation, image storage, and I/O classes of BWTC and has its own facilities to calculate costs and best basis, and to implement the wavelet packet transform. The class relies heavily on linked-list data structures to store packet tree information.

dtencode

This main program calls the key functions of the `WPTransform` Class to do compression and reconstruction. It implements the Double-Tree algorithm by maintaining an overall spatial tree and does the cutting and pasting necessary on each subimage so that after each are processed by `WPTransform`, the final result will have each subimage in the right place.

Notes on program development:

All of the author's development was done with Microsoft Visual C++ v. 4.0 running on Windows NT 4.0, 100Mhz Intel Pentium processor with 64 MB RAM. The code is still in the beta stage. Although the core program works well, there are still some bugs in the `BasisTree2D` class node counting algorithm. Many "bells and whistles" have yet to be

added. When all bugs are gotten rid of, a complete program will be provided to the University for research purposes.

Program Flow Description

The process is explained for the Single-Tree operation. Extension to the Double-Tree operation involves higher level manipulations of the image, but Double-Tree essentially calls the Single-Tree operation as a subroutine.

1. Input images are taken in PGM format. File names are passed to the program via command line arguments. Compression programs included in the companion disk to this thesis are compiled with the options delineated in section 6.3 explaining the program components. Different options and parameters may be specified by editing the main program file `dtcoder.cpp`, and recompiling the program. The options are passed to the program modules (classes) as function parameters. Available options are explained in the modules in the “Baseline Wavelet Transform Coder Construction Kit” (BWTC) package.

Encoding

2. The image is read into the class `WPTransform`. The constructor of the class calls functions to create the data structure, transform the data, and perform the necessary bit allocation and best basis searches.

The operation is broken down as follows:

- a. On the first transform iteration, the program reads in the image and stores the pixel values in a variable. A data structure called a `PacketNode` is created (function `addNode()`) to store the data for a single wavelet packet tree node. Overall node structure is maintained in another data structure, `BasisTree2D`. The program keeps track of the depth of the tree at the current iteration and extracts the appropriate data for the current node. Each `PacketNode` stores the appropriate node data and sets up BWTC classes associated with that set of data, i.e. the quantizer. As shown in Figure B-1, the `PacketNode` holds pointers to other `PacketNodes` to implement the tree structure in a linked list fashion. Each call to add a node does the following:
 - 1) Create all the necessary pointers and status variable to maintain the node and links to other nodes.
 - 2) Create a BWTC `CoeffSet` class to store node data.
 - 3) Create a BWTC quantizer and interface with the entropy coder.
 - 4) Calculate the R-D curve points for the current set of data.
- b. The contents of the variable storing the image/coefficients corresponding to the current node are then transformed by calling the wavelet transform function `wavelet->transform2d()` and written back to the image variable. The image variable now contains the coefficient values after one pass of the wavelet

transform. The first pass give the coefficients in depth-1 of the tree, the second pass gives the coefficients in depth-2, etc.

c. Step 2a - 2b is repeated until the entire wavelet packet tree is populated.

3. Data in the tree structure now undergoes a bit allocation procedure

(`allocateBits()`). The core bit allocation procedure in the compression program was taken from BWTC. The procedure performs the optimum search for lambda by calculating a value of lambda in the current iteration and performing the following:

a. For the current lambda, the Lagrangian costs are found over the points on the R-D curve for each node in the tree. The minimum (optimum) cost, rate, and distortion values are found and stored in variables. Data on the best quantizer grade (stepsize) associated with the optimum cost is stored.

b. A search for the best basis is done (`findBestBasis()`) on the tree by comparing the values stored in the optimum cost variables of each node. A temporary `BasisTree2D` variable is kept to store the intermediate results of the tree search. When the search through the entire tree is complete, the contents of the temporary `BasisTree2D` variable is stored in the permanent `BasisTree2D`. We have the best basis for the current value of lambda.

c. Revise lambda and repeat steps 3a - 3b until an optimum lambda and best basis is found.

- d. Perform “greedy” augmentation. This portion of code, as was the portion on bit allocation was taken from BWTC.
4. Based on the information on the best basis tree contained in a `BasisTree2D` data structure, the program prunes the branch nodes (`pruneTree()`) that are not in the best basis to free up some memory.
 5. A call to `encode()` then takes the data from the best basis nodes and encodes the quantized values using Arithmetic Coding into the output file.

Decoding

6. A link to the compressed output file is made and data is read from the file with a call to `decode()`. The BWTC decoder class is used to perform reverse Arithmetic Coding and retrieve the quantized data and quantizer information from the data file.
7. The data structure in the encoding process was not destroyed so that data read from the compressed file can be entered directly into the wavelet packet tree. No re-creation of the tree is necessary.

8. Using the tree information, the program performs the inverse wavelet transforms on the appropriate packet nodes (`invert()`) and the uncompressed image is written section by section until all the packets in the basis have been transformed. The result is the uncompressed image.

WPTransform Tools

FIGURE B-1

```

// Wavelet Packet Transform Class
// basistree2d.h
//
// Lester Gong
// Version: 0.01a
// Date: 07 March 1997
//
// To accompany:
// Baseline Wavelet Transform Coder Construction Kit
//
// Geoff Davis
// gdavis@cs.dartmouth.edu
// http://www.cs.dartmouth.edu/~gdavis
//
// Permission is granted to use this software for research purposes as
// long as this notice stays attached to this software.

// Modifications for Wavelet Packets by Lester Gong 3/7/97

// WPTransform class performs transform operations and holds all data for a
// wavelet packet table.

#ifndef _BASISTREE2D_
#define _BASISTREE2D_

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

//////////////////////////////////////
//
// Class Definitions

// Array representation of a wavelet packet basis tree
// May want to put this inside class and ref by BasisTree2D::Reset, etc.
enum TraverseParam
{
    Reset, All_Nodes, Basis_Nodes, Bottom_to_Top, Top_to_Bottom
};

enum TreeStruct
{
    Full_2D, Zero_2D, Wavelet_2D
};

class BasisTree2D
{
public:
    BasisTree2D(const int& depth); // default constructor
    BasisTree2D(const BasisTree2D& btree); // copy constructor
    ~BasisTree2D();
    BasisTree2D& operator=(const BasisTree2D& btree);

    // Check if parameters are within range of tree
    char validRange(const int& d, const int& x, const int& y) const;

    // Get value at Node
    char Node(const int& d, const int& x, const int& y) const;
    // char Node() const; // get status of current node
    // set value at Node
    int setNode(const int& d, const int& x, const int& y, const int& val);
    // int setNode(); // set current node

```

```

void setIndex(const int& value);
//
int getIndex() const;
int getIndex(const int& d_idx, const int& x_idx, const int& y_idx)
    const;
int traverseBasisTree(TraverseParam param);
int setTreeStruct(TreeStruct structure);

// Returns array of BasisTree Coefs
char* getBasis() const { return basisArray; }
unsigned int numTreeNodees() const { return nTreeNodees; }
unsigned int numBasisNodes() const { return nBasisNodes; }
unsigned int numAllNodes() const { return nAllNodes; }
char resetFlag() const { return rFlag; }
int mode() const { return tMode; }
int maxDepth() const { return mDepth; }
int curDepth() const { return cDepth; }
int* curNodeParams(int* paramPtr) const;

private:
char* basisArray; // array representing nodes of basis tree
int* indexArray; // index for user's use associated with node
unsigned int nTreeNodees; // number of nodes in current tree
unsigned int nBasisNodes; // number of nodes in basis (leaves only)
unsigned int nAllNodes; // total number of nodes
char rFlag; // status of reset state
int tMode; // holds current traversing mode
int (* stack)[3]; // stack
int cnt; // stack counter
int mDepth; // max depth of tree
int cDepth; // current depth of deepest leaf nodes
int d; // depth of node currently indexed
int x; // x position of node currently indexed
int y; // y position of node " "
};

#endif

```



```

// Wavelet Packet Classes
// basistree2d.cpp
//
// Lester Gong
// Version: 0.01a
// Copyright (c) 1997 Lester Gong
//
// to accompany:
// Baseline Wavelet Transform Coder Construction Kit
// by Geoff Davis
// gdavis@cs.dartmouth.edu

/*-----*/
#include "basistree2d.h"
/*-----*/

// Constructor
BasisTree2D::BasisTree2D(const int& depth):
    nAllNodes(0), // root node
    rFlag(1),
    tMode(Reset),
    mDepth(depth),
    cDepth(0),
    cnt(0),
    d(0),
    x(0),
    y(0)
{
    if ((depth < 0) || (depth > 7)) cerr << "Invalid depth specified" << endl;
    if (depth == 0)
    {
        basisArray = new char(0);
        indexArray = new int(0);
        nAllNodes = nTreeNodees = nBasisNodes = 1;
    }
    else
    {
        // since all leafs in the last depth of the tree must be zero, use
        // depth-1 to allot basisArray to save some memory
        int length_b = (int)(pow(4, (depth-1))*1.5);
        basisArray = new char [length_b];
        // Full length needed here since all indices may be used
        int length_idx = (int)(pow(4, depth)*1.5);
        indexArray = new int [length_idx];
        // initialize all nodes to char 0
        for (int i = 0; i < length_b; i++)
        {
            basisArray[i] = 1;
        }
        for (int j = 0; j < length_idx; j++)
        {
            indexArray[j] = 0;
        }
        for (int k = 1; k < depth+1; k++)
        {
            nAllNodes += (int)pow(4,k);
        }
        nAllNodes++; // for root node
        nTreeNodees = nAllNodes;
        nBasisNodes = (int)pow(4,depth);
    }
}
// Statics default inits to zero.

```

```

        stack = new int [4*mDepth + 1][3];
        stack[0][0] = 0;
        stack[0][1] = 0;
        stack[0][2] = 0;
    }

    // Copy constructor
    BasisTree2D::BasisTree2D(const BasisTree2D& btree)
    {
        nAllNodes = btree.nAllNodes;
        nTreeNodees = btree.nTreeNodees;
        nBasisNodes = btree.nBasisNodes;
        rFlag = 1;
        tMode = Reset;
        mDepth = btree.mDepth;
        cDepth = btree.cDepth;
        cnt = 0;
        d = x = y = 0;
        basisArray = new char [btree.mDepth];
        for (int i = 0; i < (int) (pow(4, (mDepth-1))*1.5); i++)
        {
            basisArray[i] = btree.basisArray[i];
        }
        for (int j = 0; j < (int) (pow(4, (mDepth))*1.5); j++)
        {
            indexArray[j] = btree.indexArray[j];
        }
        stack = new int [4*mDepth + 1][3];
        stack[0][0] = 0;
        stack[0][1] = 0;
        stack[0][2] = 0;
    }

    // Destructor
    BasisTree2D::~BasisTree2D()
    {
        delete [] basisArray;
        delete [] indexArray;
        delete [] stack; // deleting a null pointer is ok
    }

    // Overload =
    BasisTree2D&
    BasisTree2D::operator=(const BasisTree2D& btree)
    {
        if (basisArray == btree.basisArray)
        {
            cerr << "Attempted Self-assignment in operator=" << endl;
        }
        else
        {
            nAllNodes = btree.nAllNodes;
            for (int i = 0; i < (int) (pow(4, (mDepth-1))*1.5); i++)
            {
                basisArray[i] = btree.basisArray[i];
            }
            for (int j = 0; j < (int) (pow(4, mDepth)*1.5); j++)
            {
                indexArray[j] = btree.indexArray[j];
            }
            nTreeNodees = btree.nTreeNodees;
            nBasisNodes = btree.nBasisNodes;
            rFlag = 1;
        }
    }

```

```

        tMode = Reset;
        stack[0][0] = 0;
        stack[0][1] = 0;
        stack[0][2] = 0;
        mDepth = btree.mDepth;
        cDepth = btree.cDepth;
        cnt = 0;
        d = x = y = 0;
    }
    return *this;
}

char
BasisTree2D::validRange(const int& d, const int& x, const int& y) const
{
    if ((d < 0) || (d > mDepth) || (x < 0) || (x > pow(2,d)-1) ||
        (y < 0) || (y > pow(2,d)-1))
    {
        cerr << "Parameters are out of range of maximum tree or are invalid ";
        cerr << d << ',' << x << ',' << y << endl;
        return 0;
    }
    else
    {
        return 1;
    }
}

char
BasisTree2D::Node(const int& d, const int& x, const int& y) const
{
    if (d == mDepth)
    {
        return 0; // last leaves always 0
    }
    else
    {
        return basisArray[(int)pow(4,d)/2+x*(int)pow(2,d)+y];
    }
}

// setNode()
int
BasisTree2D::setNode(const int& d, const int& x, const int& y, const int& val)
{
    char temp;
    temp = val ? 1 : 0; // regard 0 as 0 and all other values as 1
    if (d == mDepth)
    {
        if (temp)
        {
            cerr << "Can't add node to leaves at maximum depth" << endl;
            return -1;
        }
        else // if setting leaf at max depth to zero, nothing needs to be done
        {
            return 0;
        }
    }
    // Check if assignment makes a change to tree
    if (basisArray[(int)pow(4,d)/2 + x*(int)pow(2,d) + y] != temp)
    {
        if (temp) // adding nodes(s)

```

```

{
    basisArray[(int)pow(4,d)/2 + x*(int)pow(2,d) + y] = 1;
    nBasisNodes += 4;
    nTreeNodes += 4;
    if (d == 0) // adding the root node only
    {
        basisArray[0] = 1;
        nBasisNodes = 4;
        nTreeNodes = 5;
    }
    else
    {
        int i = (d-1); // we never have d == 0 here
        int tx;
        int ty;
        while (i >= 0)
        {
            tx = x/2;
            ty = y/2;
            if (Node(i,tx,ty) == 0) // if parent not set
            {
                cout << "Warning: Adding multiple node" << endl;
                basisArray[(int)pow(4,i)/2 + tx*(int)pow(2,i) + ty] = 1;
                nBasisNodes += 3;
                nTreeNodes += 4;
            }
            else
            {
                nBasisNodes--; // subtr one if parent node was set
                break;
            }
            i--;
        }
        nBasisNodes--; // subtr final one
    }
    if ((d+1) > cDepth) cDepth = d+1; // update current depth
}
else // killing node(s)
{
    // If any children are set
    if (Node(d+1,2*x,2*y) || Node(d+1,2*x,2*y+1) || Node(d+1,2*x+1,2*y)
        || Node(d+1,2*x+1,2*y+1))
    {
        //
        cout << "Warning: Killing multiple nodes" << endl;
        // Use algorithm in traverseBasis(All_Nodes)
        int (* tstack)[3] = new int [4*(mDepth-d) + 1][3];
        int k = 0;
        int td, tx, ty;
        tstack[k][0] = d;
        tstack[k][1] = x;
        tstack[k][2] = y;
        do
        {
            td = tstack[k][0];
            tx = tstack[k][1];
            ty = tstack[k][2];
            k--;
            if (Node(td, tx, ty) == 1)
            {
                basisArray[(int)pow(4,td)/2 + tx*(int)pow(2,td) + ty] = 0;
                nTreeNodes -= 4;
                if ((td + 1) == mDepth)
                {

```

```

        nBasisNodes -= 4; // consider opt. w/ multiple --'s
    }
    else
    {
        k++;
        tstack[k][0] = td + 1;
        tstack[k][1] = 2 * tx;
        tstack[k][2] = 2 * ty;
        k++;
        tstack[k][0] = td + 1;
        tstack[k][1] = 2 * tx + 1;
        tstack[k][2] = 2 * ty;
        k++;
        tstack[k][0] = td + 1;
        tstack[k][1] = 2 * tx;
        tstack[k][2] = 2 * ty + 1;
        k++;
        tstack[k][0] = td + 1;
        tstack[k][1] = 2 * tx + 1;
        tstack[k][2] = 2 * ty + 1;
        k++;
    }
    else
    {
        nBasisNodes--;
    }
    nBasisNodes++; // add back a final one
} while (k >= 0);
delete [] tstack;
}
else // no children are set
{
    basisArray[(int)pow(4,d)/2 + x*(int)pow(2,d) + y] = 0;
    nTreeNode -= 4;
    nBasisNodes -= 3;
}
if ((d+1) == cDepth) // update current depth
{
    char flag = 1;
    for (int i = 0; i < (int)pow(2,d); i++)
    {
        if (flag == 0) break; // no need to continue loop
        for (int j = 0; j < (int)pow(2,d); j++)
        {
            if (Node(d,i,j) == 1)
            {
                flag = 0; // no change to cDepth needed
                break; // no need to finish loop
            }
        }
    }
    if (flag) cDepth--;
}
} // end killing nodes
return 1; // a change was made
}
return 0; // no change required
}

// setIndex()
void
BasisTree2D::setIndex(const int& value)
{

```

```

    indexArray[(int)pow(4,d)/2 + x*(int)pow(2,d) + y] = value;
}

// getIndex()
// Get stored index value at current location
int
BasisTree2D::getIndex() const
{
    return indexArray[(int)pow(4,d)/2 + x*(int)pow(2,d) + y];
}

// getIndex()
int
BasisTree2D::getIndex(const int& d_idx,
                      const int& x_idx, const int& y_idx) const
{
    return indexArray[(int)pow(4,d_idx)/2 + x_idx*(int)pow(2,d_idx) + y_idx];
}

// setTreeStruct()
// Set basis to a specific structure
int
BasisTree2D::setTreeStruct(TreeStruct structure)
{
    switch (structure)
    {
        case 1: // Zeroed Tree
        {
            for (int i = 0; i < (int)(pow(4, (mDepth-1))*1.5); i++)
            {
                *(basisArray + i) = 0;
            }
            cDepth = 0;
            rFlag = 1;
            nTreeNodees = 1;
            nBasisNodes = 1;
            tMode = Reset;
            d = x = y = 0;
            break;
        }
        case 2: // Wavelet Tree
        {
            // first zero out the tree
            for (int i = 0; i < (int)(pow(4, (mDepth-1))*1.5); i++)
            {
                *(basisArray + i) = 0;
            }
            // set LL band for each level to 1
            for (int j = 0; j < mDepth; j++)
            {
                basisArray[(int)pow(4,j)/2 - 1] = 1;
            }
            cDepth = mDepth;
            rFlag = 1;
            nTreeNodees = (int)pow(4,mDepth) + 1;
            nBasisNodes = (int)pow(3, (mDepth-1)) + 1;
            tMode = Reset;
            d = x = y = 0;
            break;
        }
        case 0: // Full Tree
        default:
    }
}

```

```

    {
        for (int i = 0; i < (int)(pow(4,(mDepth-1))*1.5); i++)
        {
            *(basisArray + i) = 1;
        }
        cDepth = mDepth;
        rFlag = 1;
        nTreeNodees = nAllNodes;
        nBasisNodes = (int)pow(4,mDepth);
        tMode = Reset;
        d = x = y = 0;
        break;
    }
}
return 0;
}

// traverseBasisTree()
// Called to step thru basis tree
int
BasisTree2D::traverseBasisTree(TraverseParam param)
{
    // static int stkcount= 0;
    if (cnt < 0) param = Reset; // traversed whole tree
    if (param != Reset)
    {
        if ((tMode != Reset) && (tMode != param))
        {
            cerr << "Changing modes without resetting" << endl;
            return 1;
        }
        else if (mDepth == 0)
        {
            param = Reset;
        }
        else
        {
            tMode = param;
        }
    }
    if (param == Reset)
    {
        // Reset routine
        tMode = Reset;
        cnt = 0;
        stack[cnt][0] = 0;
        stack[cnt][1] = 0;
        stack[cnt][2] = 0;
        d = x = y = 0;
        rFlag = 1;
        // cout << "reset ";
        // cout << "(" << d << ',' << x << ',' << y << ")" << endl;
        return 0;
    }
    if (param == Bottom_to_Top)
    {
        if (d == 0)
        {
            d = mDepth;
            y = -1; // catches node (mDepth, 0, 0) on first y++
        }
        rFlag = 0;
        y++;
        if (y > (pow(2,d)-1)) // if at max y

```

```

    {
        y = 0;
        x++;
        if (x > (pow(2,d)-1)) // if at max x
        {
            x = 0;
            d -= 1;
            if (d == 0) rFlag = 1;
        }
    }
    // cout << "Bot-top ";
    // cout << "(" << d << ',' << x << ',' << y << ")" << endl;
    return 0;
}
else if (param == Top_to_Bottom)
{
    if (rFlag == 1) y = -1;
    if (d > mDepth)
    {
        traverseBasisTree(Reset);
    }
    else
    {
        rFlag = 0;
    }
    y++;
    if (y > (pow(2,d)-1)) // if at max y
    {
        y = 0;
        x++;
        if (x > (pow(2,d)-1)) // if at max x
        {
            x = 0;
            d += 1;
            if (d > mDepth) rFlag = 1;
        }
    }
    // cout << "Top_bot ";
    // cout << "(" << d << ',' << x << ',' << y << ")" << endl;
    return 0;
}
else
{
    rFlag = 0;
    do
    {
        d = stack[cnt][0];
        x = stack[cnt][1];
        y = stack[cnt][2];
        cnt -= 1;
        if (Node(d, x, y) == 1)
        {
            cnt++;
            stack[cnt][0] = d + 1;
            stack[cnt][1] = 2 * x;
            stack[cnt][2] = 2 * y;
            cnt++;
            stack[cnt][0] = d + 1;
            stack[cnt][1] = 2 * x + 1;
            stack[cnt][2] = 2 * y;
            cnt++;
            stack[cnt][0] = d + 1;
            stack[cnt][1] = 2 * x;

```



```

        stack[cnt][2] = 2 * y + 1;
        cnt++;
        stack[cnt][0] = d + 1;
        stack[cnt][1] = 2 * x + 1;
        stack[cnt][2] = 2 * y + 1;
        if (param == All_Nodes)
        {
            cout << "All ";
            cout << "(" << d << ',' << x << ',' << y << ")" << endl;
            return 0;
        }
    } while ((param == Basis_Nodes) && (Node(d, x, y) == 1));
    cout << "Basis ";
    cout << "(" << d << ',' << x << ',' << y << ")" << endl;
    return 0;
}

// curNodeParams()
int*
BasisTree2D::curNodeParams(int* paramPtr) const
{
    if (paramPtr == 0)
    {
        cerr << "curNodeParams() was passed a null pointer" << endl;
        return 0;
    }
    paramPtr[0] = d;
    paramPtr[1] = x;
    paramPtr[2] = y;
    return paramPtr;
}

```

```

/////////////////////////////////////////////////////////////////
//
// Wavelet Packets and Double Tree Algorithm Image Compression Program
// dtcoder.cpp
// Main program v.0.01a
//
// Uses WPTransform and BasisTree2D classes of the Wavelet Packet Classes
// by Lester Gong
//
// to accompany:
// Baseline Wavelet Transform Coder Construction Kit
// by Geoff Davis (gdavis@cs.dartmouth.edu)
//
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <iostream.h>
#include <fstream.h>
#include "wptransform.h"
#include "basistree2d.h"
#include "coeffset.h"
#include "quantizer.h"
#include "coder.h"

/////////////////////////////////////////////////////////////////
//
// Function Definitions
//
/////////////////////////////////////////////////////////////////
void
singleTreeCompression(char* compfile, ostream& stats,
                      Image* output_img, Image* image,
                      Wavelet* wavelet, int nStages, int budget,
                      int paramPrecision, int capacity, int nQuant,
                      Real minStepSize, int monolayer, int symmetric);

void
doubleTreeCompression(BasisTree2D* spatialtree, WPTransform** wpackets,
                      char* compfile, Image* output_img);

void
buildSpatialTree(BasisTree2D* stree, WPTransform** wpackets, Wavelet* wavelet,
                 Image* image, int budget, int paramPrecision, int capacity,
                 int nQuant, Real minStepSize, int monolayer,
                 ostream& stats, int symmetric,
                 int nStages);

void
findBestSpatialTree(BasisTree2D* stree, WPTransform** wpackets,
                    int maxLevel, ostream& stats);

int*
findQuad(const int& d_idx, const int& x_idx, const int& y_idx,
          const int& length, int* quadbounds);

void
compressWP(WPTransform* packets, char* compfile);

Image*
decompressWP(WPTransform* packets, char* compfile, Image* image);

```

```

void
writeImageQuad(Image* image, Image* quad, const int* quadbounds);

/////////////////////////////////////////////////////////////////
//
// Main Program
//
/////////////////////////////////////////////////////////////////
int
main (int argc, char** argv)
{
    char* program = argv[0];

    if (argc < 4)
    {
        cerr << "Usage: " << program << " [image][output][ratio]";
        cerr << "[restored][data]" << endl;
        cerr << "image: image to be compressed (in PGM format)" << endl;
        cerr << "output: name of compressed image" << endl;
        cerr << "ratio: compression ratio" << endl;
        cerr << "restored: name of file for restored image" << endl;
        cerr << "data: data file for program status output" << endl;
        exit(0);
    }
    char* input = argv[1];
    char* compfile = argv[2];
    Real ratio = atof(argv[3]);
    char* decompfile = argv[4];
    char* statfile = argv[5];

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

    // assume 8 bit pixels
    int budget = (int)((Real)(image->hsize*image->vsize)/ratio);
    cout << "Reading " << image->hsize << " x " << image->vsize;
    cout << " image, " << input << " writing " << compfile << endl;
    cout << "Compression ratio " << ratio << ":1" << endl;

    // Coding parameters
    Real p = 2.0; // exponent for L^p error metric
    int nStages = 5; // # of stages in the wavelet transform
    int capacity = 512; // capacity of histogram for arithmetic coder
    int paramPrecision = 4; // precision for stored quantizer parameters
    int nQuant = 10; // # of quantizers to examine for allocation
    Real minStepSize = 0.05; // smallest quantizer step to consider
    int monolayer = TRUE; // TRUE for non-embedded uniform quantizer
    // FALSE for multilayer quantizer
    int symmetric = -1; // default
    int maxLevel = 3; // max of spatial sublevel + transform depth
    int doubleTree = FALSE; // do doubletree algorithm

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

    // Image to be reconstructed from compressed file
    Image* output_img = new Image(image->hsize, image->vsize);

    ofstream stats(statfile, ios::out | ios::trunc);
    if (doubleTree)
    {
        BasisTree2D* spatialtree = new BasisTree2D(maxLevel); // spatial tree
    }

```

```

    WPTransform** wpackets = new WPTransform* [spatialtree->numAllNodes()];
    buildSpatialTree(spatialtree, wpackets, wavelet, image, budget,
                    paramPrecision, capacity, nQuant, minStepSize,
                    monolayer, stats, symmetric, nStages);
    findBestSpatialTree(spatialtree, wpackets, maxLevel, stats);
    doubleTreeCompression(spatialtree, wpackets, compfile, output_img);
    delete [] wpackets;
    delete spatialtree;
}
else // do normal single wavelet packet compression
{
    singleTreeCompression(compfile, stats,
                          output_img, image, wavelet, nStages,
                          budget, paramPrecision, capacity, nQuant,
                          minStepSize, monolayer, symmetric);
}

output_img->savePGM(decompfile); // save file in PGM format

stats.close();
delete output_img;
delete wavelet;
delete image;
return 0;
} // end main()

// singleTreeCompression
//
void
singleTreeCompression(char* compfile, ostream& stats,
                      Image* output_img, Image* image,
                      Wavelet* wavelet, int nStages, int budget,
                      int paramPrecision, int capacity, int nQuant,
                      Real minStepSize, int monolayer, int symmetric)
{
    WPTransform *transform =
        new WPTransform (wavelet, image, nStages, budget, paramPrecision,
                        capacity, nQuant, minStepSize, monolayer,
                        stats, symmetric);

    transform->allocateBits();
    transform->pruneTree(); // deallocate memory for unused nodes

    // Open output file
    ofstream outfile(compfile, ios::out | ios::trunc | ios::binary);
    if (!outfile)
    {
        error ("Unable to open file %s", compfile);
    }

    // Create I/O interface object for arithmetic coder
    Encoder* encoder = new Encoder (outfile);

    // Write image size to output file
    encoder->writePositive (image->hsize);
    encoder->writePositive (image->vsize);

    transform->encode(encoder);

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

```

```

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

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

// Read image dimensions from file.  These values aren't needed.
int hs = decoder->readPositive ();
int vs = decoder->readPositive ();

transform->decode(decoder);

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

// Allocate image and invert transform
transform->invert(output_img);
delete transform;
}

void
doubleTreeCompression(BasisTree2D* spatialtree, WPTransform** wpackets,
                      char* compfile, Image* output_img)
{
    spatialtree->traverseBasisTree(Reset);
    while (1)
    {
        int indices[3];
        spatialtree->traverseBasisTree(Basis_Nodes);
        if (spatialtree->resetFlag()) break;
        spatialtree->curNodeParams(indices);
        int idx = spatialtree->getIndex(indices[0], indices[1], indices[2]);
        int quadbounds[4]; // boundaries for each sub-image
        findQuad(indices[0], indices[1], indices[2], output_img->hsize, quadbounds);
        compressWP(wpackets[idx], compfile);
        Image* temp_img = new Image(quadbounds[2] - quadbounds[0] + 1,
                                   quadbounds[3] - quadbounds[1] + 1);
        decompressWP(wpackets[idx], compfile, temp_img);
        writeImageQuad(output_img, temp_img, quadbounds);
        delete temp_img;
    }
}

void
buildSpatialTree(BasisTree2D* stree, WPTransform** wpackets, Wavelet* wavelet,
                 Image* image, int budget, int paramPrecision, int capacity,
                 int nQuant, Real minStepSize, int monolayer,
                 ostream& stats, int symmetric,
                 int nStages)
{
    int indices[3]; // holds indices d, x, y of spatial tree
    int idx = 0;    // index of wavelet packet basis for each spatial sublvl
    for (int m = 0; m < stree->numAllNodes(); m++)
    {
        stree->traverseBasisTree(Top_to_Bottom);
        stree->curNodeParams(indices);
        int d = indices[0];

```

```

    int x = indices[1];
    int y = indices[2];
    // dissect input image into sub-images to be processed
    int bounds[4]; // holds boundaries of sub-image
    findQuad(d, x, y, image->hsize, bounds); // assume square quad
    int h, v;
    h = *(bounds+2) - *(bounds) + 1; // horizontal size of quad
    v = *(bounds+3) - *(bounds+1) + 1; // vertical size of quad
    Image* temp_img = new Image(h, v);
    // Get section of image framed by quad
    for (int i = *(bounds+1); i < *(bounds+3)+1; i++)
    {
        for (int j = *(bounds); j < *(bounds+2)+1; j++)
        {
            *(temp_img->value + h*(i-*(bounds+1)) + (j-*(bounds))) =
                *(image->value + (image->hsize*i+j));
        }
    }
    cout << "Spatial Lvl: " << d << ", " << x << ", " << y << endl;
    cout << flush;
    int adj_budget = budget/(int)pow(4,d); // adjust budget
    int stages = nStages;
    // if (d == 2) stages--; // limit last splits
    if (d == 3) stages--;
    wpackets[idx] = new WPTransform (wavelet, temp_img, stages,
                                     adj_budget, paramPrecision, capacity,
                                     nQuant, minStepSize, monolayer,
                                     stats, symmetric);

    wpackets[idx]->allocateBits();
    wpackets[idx]->pruneTree(); // deallocate memory for unused nodes
    stree->setIndex(idx); // save packet index into current node
    idx++;
}
}

// find best spatial tree basis
void
findBestSpatialTree(BasisTree2D* stree, WPTransform** wpackets,
                    int maxLevel, ostream& stats)
{
    // create new spatial tree to record best
    stats << "Finding Best Spatial Basis" << endl;
    BasisTree2D* newSTree = new BasisTree2D(maxLevel);
    newSTree->setTreeStruct(Zero_2D); // zero out new spatial tree
    int indices[3];
    stree->traverseBasisTree(Reset);
    // loop until out of leaves
    while (1)
    {
        stree->traverseBasisTree(Bottom_to_Top);
        stree->curNodeParams(indices);
        if (indices[0] < maxLevel) break;
    }
    Real costChildren = 0.0;
    int i = 0; // Flag to signal last iteration. Used for loop control.
    do
    {
        stree->curNodeParams(indices);
        int d = indices[0];
        int x = indices[1];
        int y = indices[2];
        costChildren =
            wpackets[stree->getIndex(d+1,2*x,2*y)]->optDist +

```

```

        wpackets[stree->getIndex(d+1,2*x+1,2*y)]->optDist +
        wpackets[stree->getIndex(d+1,2*x,2*y+1)]->optDist +
        wpackets[stree->getIndex(d+1,2*x+1,2*y+1)]->optDist;
    if (costChildren < wpackets[stree->getIndex(d,x,y)]->optDist)
    {
        wpackets[stree->getIndex(d,x,y)]->optDist = costChildren;
        wpackets[stree->getIndex(d,x,y)]->optRate =
            wpackets[stree->getIndex(d+1,2*x,2*y)]->optRate +
            wpackets[stree->getIndex(d+1,2*x+1,2*y)]->optRate +
            wpackets[stree->getIndex(d+1,2*x,2*y+1)]->optRate +
            wpackets[stree->getIndex(d+1,2*x+1,2*y+1)]->optRate;
        wpackets[stree->getIndex(d,x,y)]->optCost =
            wpackets[stree->getIndex(d+1,2*x,2*y)]->optCost +
            wpackets[stree->getIndex(d+1,2*x+1,2*y)]->optCost +
            wpackets[stree->getIndex(d+1,2*x,2*y+1)]->optCost +
            wpackets[stree->getIndex(d+1,2*x+1,2*y+1)]->optCost;
        newSTree->setNode(d, x, y, 1);
        stats << " (" << d << ", " << x << ", " << y << ") 1" << endl;
    }
    else
    {
        newSTree->setNode(d, x, y, 0);
        stats << " (" << d << ", " << x << ", " << y << ") 0" << endl;
    }
    stree->traverseBasisTree(Bottom_to_Top);
    if (maxLevel == 1) break; // only one iteration needed
    if (stree->resetFlag()) i = 1; // go for one last iter at (0,0,0)
} while (stree->resetFlag() || i == 0);
stree->traverseBasisTree(Reset);
// copy over index values to new tree
newSTree->traverseBasisTree(Reset);
while (1)
{
    newSTree->traverseBasisTree(Basis_Nodes);
    if (newSTree->resetFlag()) break;
    newSTree->curNodeParams(indices);
    newSTree->setIndex(stree->getIndex(indices[0],indices[1],indices[2]));
}
*(stree) = *(newSTree);
delete newSTree;
}

int*
findQuad(const int& d_idx, const int& x_idx,
        const int& y_idx, const int& length, int* quadbounds)
{
    int quadlength = length/(int)pow(2,d_idx);
    quadbounds[0] = x_idx*quadlength;
    quadbounds[1] = y_idx*quadlength;
    quadbounds[2] = (x_idx+1)*quadlength - 1;
    quadbounds[3] = (y_idx+1)*quadlength - 1;
    return quadbounds;
}

void
compressWP(WPTransform* packets, char* compfile)
{
    ofstream outfile (compfile, ios::out | ios::trunc | ios::binary);
    if (!outfile)
    {
        error("Unable to open file %s", compfile);
    }
    Encoder* encoder = new Encoder (outfile);

```

```

        encoder->writePositive(packets->getHsize());
        encoder->writePositive(packets->getVsize());
        packets->encode(encoder);
        encoder->flush();
        delete encoder;
        outfile.close();
    }

Image*
decompressWP(WPTransform* packets, char* compfile, Image* image)
{
    ifstream infile (compfile, ios::in | ios::nocreate | ios::binary);
    if (!infile)
    {
        error ("Unable to open file %s", compfile);
    }
    Decoder *decoder = new Decoder (infile);

    int h = decoder->readPositive();
    int v = decoder->readPositive();
    packets->decode(decoder);
    delete decoder;
    infile.close();
    packets->invert(image);
    return image;
}

void
writeImageQuad(Image* image, Image* quad, const int* quadbounds)
{
    int lox = quadbounds[0];
    int loy = quadbounds[1];
    int hix = quadbounds[2];
    int hiy = quadbounds[3];
    int quadh = hix - lox + 1;

    for (int i = 0; i < hiy-loy+1; i++)
    {
        for (int j = 0; j < hix-lox+1; j++)
        {
            *(image->value + image->hsize*(i+loy) + (j+lox)) =
                *(quad->value + quadh*i + j);
        }
    }
}

```



```

// Wavelet Packet Transform Class
// wpttransform.h
//
// Lester Gong
// Version: 0.01a
// Date: 07 March 1997
// Copyright (c) 1997 Lester Gong
//
// to accompany:
// Baseline Wavelet Transform Coder Construction Kit
// Geoff Davis
// gdavis@cs.dartmouth.edu
// http://www.cs.dartmouth.edu/~gdavis
//
// Copyright 1996 Geoff Davis 9/11/96
//
// Permission is granted to use this software for research purposes as
// long as this notice stays attached to this software.

// WPTransform class performs transform operations and holds all data for a
// wavelet packet table.

#ifndef _WPTRANSFORM_
#define _WPTRANSFORM_
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "wavelet.h"
#include "image.h"
#include "coeffset.h"
#include "quantizer.h"
#include "coder.h"
#include "entropy.h"
#include "basistree2d.h"

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/
// Class Definitions

// Class PacketNode
class PacketNode
{
public:
    PacketNode();
    ~PacketNode();
    PacketNode* getPacketNode(int d, int x, int y);
    PacketNode* child_LL;
    PacketNode* child_LH;
    PacketNode* child_HL;
    PacketNode* child_HH;
    CoeffSet* data;
    ErrorMetric* err;
    EntropyCoder* entropy;
    Quantizer* quant;
    int index;
    Real optRate;
    Real optDist;
    Real optCost;

    friend class PacketTable;
    friend class WPTransform;
    friend class PacketIndexer;

```

```

    private:
        int size;
        int depth;
};

// Class PacketTable
class PacketTable
{
    public:
        // Constructor
        PacketTable();
        ~PacketTable();
        int addNode(Real* data, int index, int size, int d, int x, int y,
                    int monolayer, int capacity, int paramPrecision, int nQuant,
                    Real minStepSize, PacketIndexer*& indexer);
        int getDepth(); // get depth of packet table

        PacketNode* firstNode;
        friend class PacketIndexer;
        friend class WPTransform;
    private:
        int totalDepth;
        int numberNodes;
};

//Class PacketIndexer
class PacketIndexer
{
    public:
        // Constructor
        PacketIndexer(const PacketTable*& pt, BasisTree2D*& basis):
            firstNode(pt->firstNode),
            basis(basis){ };
        PacketIndexer():firstNode(0),basis(0){ int g = 10; g++;}
        PacketIndexer(const PacketIndexer*& idx);

        int traversePacket();
        PacketNode* getPacketNode(const int& d, const int& x, const int& y);
        PacketNode* firstNode;
        BasisTree2D* basis;
        friend class WPTransform;
    private:
};

class WPTransform
{
    public:
        // Constructors
        WPTransform(Wavelet* wavelet, Image* image, const int& depth,
                    int budget, int paramPrecision, int capacity, int nQuant,
                    Real minStepSize, int monolayer,
                    ostream& statfile, int symmetric = -1);
        WPTransform(Wavelet* wavelet, int hsize, int vsize, BasisTree2D* basis,
                    int symmetric = -1);
        // Copy Constructor
        WPTransform(const WPTransform &WPTransform);

        // Destructor
        ~WPTransform();

        // Overloaded = for
        WPTransform& operator= (const WPTransform &WPTransform);
};

```

```

friend class PacketNode;
friend class PacketTable;
friend class PacketIndexer;
// Do wavelet pkt transform w/ current basis tree
int transform();
void invert(Image* transformedImage); //, BasisTree2D* btree);
void findBestBasis(const Real*& optRDJ, BasisTree2D* tempBasis,
                  Real& optimumRate, Real& optimumDist);
void findBasisRD(const Real*& optRDJ,
                 Real& optimumRate, Real& optimumDist);
void pruneTree();
void allocateBits(); // Run bit allocator
void allocateLambda(CoeffSet** coef, int nSets, Real lambda,
                   Real*& optRDJ);
// following overloaded function does basis nodes only
void allocateLambda(CoeffSet** coef, Real lambda,
                   Real*& optRDJ);
void encode(Encoder* encoder);
void decode(Decoder* decoder);
int* quad(const int& d_idx, const int& x_idx, const int& y_idx,
          const int& length, int* quadbounds);
Real optRate;
Real optDist;
Real optCost;
int getHsize() const { return hsize; }
int getVsize() const { return vsize; }

private:
    int hsize, vsize; // size of transformed image
    int d, x, y;
    Image* image;
    Real* value; // array of transform coefficients
    Wavelet* wavelet; // wavelet used for transform
    int packetDepth; // depth of packet table
    int symmetric; // TRUE for symmetric extensions, FALSE for periodic
    BasisTree2D* basis;
    int* precision; // saves quant precision for each coeffSet
    PacketTable* packetTbl;
    PacketIndexer* indexer;
    int budget;
    int paramPrecision;
    int capacity;
    int nQuant;
    Real minStepSize;
    int monolayer;
    ostream& statfile; // output text file
};
#endif

```

```

// Wavelet Packet Classes
// wptransform.cpp
//
// Lester Gong
// version 0.01a
// Copyright (c) 1997 Lester Gong
//
// to accompany
// Baseline Wavelet Transform Coder Construction Kit
// by Geoff Davis
// gdavis@cs.dartmouth.edu

// The bit allocation routine contained in the function allocateBits() is
// largely adapted from the "Construction Kit"
/*-----*/
#include "wptransform.h"
/*-----*/

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/
//
// WPTransform class implementation
//

// Constructor
WPTransform::WPTransform (Wavelet *wavelet, Image *image, const int& depth,
                          int budget, int paramPrecision, int capacity,
                          int nQuant, Real minStepSize, int monolayer,
                          ostream& statfile, int symmetric):
    wavelet(wavelet),
    packetDepth(depth),
    d(0), x(0), y(0),
    image(image),
    optRate(0), optDist(0), optCost(0),
    budget(budget),
    paramPrecision(paramPrecision),
    capacity(capacity),
    nQuant(nQuant),
    minStepSize(minStepSize),
    monolayer(monolayer),
    statfile(statfile),
    symmetric(symmetric)
{
    basis = new BasisTree2D(depth);
    packetTbl = new PacketTable();
    indexer = new PacketIndexer(packetTbl, basis);
    value = NULL;
    if (image != NULL)
    {
        hsize = image->hsize;
        vsize = image->vsize;
        value = image->value; // copy value of image
        transform();
    }
    else
    {
        hsize = vsize = 0;
    }
}

// Destructor
WPTransform::~WPTransform()
{

```

```

    delete indexer;
    delete packetTbl;
    delete basis;
    delete [] precision;
}

// transform()
int
WPTransform::transform()
{
    int index = 0;
    basis->traverseBasisTree(Reset);
    cout << "Adding Nodes:" << endl;
    while (1) // until resetFlag is reset
    {
        basis->traverseBasisTree(Top_to_Bottom);
        if (basis->resetFlag()) break; // break out if resetFlag is set
        int indices[3];
        basis->curNodeParams(indices);
        d = indices[0];
        x = indices[1];
        y = indices[2];
        int bounds[4];
        quad(d, x, y, hsize, bounds); // assume square quad
        int h, v;
        h = *(bounds+2) - *(bounds) + 1;
        v = *(bounds+3) - *(bounds+1) + 1;
        Real* temp = new Real [h * v];
        // Get section of image framed by quad
        for (int i = *(bounds+1); i < *(bounds+3)+1; i++)
        {
            for (int j = *(bounds); j < *(bounds+2)+1; j++)
            {
                *(temp + h*(i-*(bounds+1)) + (j-*(bounds))) =
                    *(image->value + (hsize*i+j));
            }
        }
        packetTbl->addNode(temp, index, h*v, d, x, y, monolayer, capacity,
                           paramPrecision, nQuant, minStepSize, indexer);
        index++; // unique tag for packetNode
        if (d != packetDepth)
        {
            Real* temp_out = new Real [h * v];
            wavelet->transform2d (temp, temp_out, h, v, 1, symmetric);
            // Write back results for next iteration
            for (int m = *(bounds+1); m < *(bounds+3)+1; m++)
            {
                for (int n = *(bounds); n < *(bounds+2)+1; n++)
                {
                    *(image->value + (hsize*m+n)) =
                        *(temp_out + h*(m-*(bounds+1)) + (n-*(bounds)));
                }
            }
            delete [] temp_out;
        }
    }
}

if (basis->maxDepth() == 0) // special case
{
    Real* temp = new Real [hsize*vsize];
    for (int k = 0; k < hsize*vsize; k++)
    {
        *(temp + k) = *(image->value + k);
    }
}

```

```

        packetTbl->addNode(temp, 0, hsize*vsize, 0, 0, 0, monolayer, capacity,
                           paramPrecision, nQuant, minStepSize, indexer);
    }
    cout << endl; // ends line for node count output in addNode()
    return 0;
}

// quad()
int*
WPTransform::quad(const int& d_idx, const int& x_idx,
                  const int& y_idx, const int& length, int* quadbounds)
{
    int quadlength = length/(int)pow(2,d_idx);
    quadbounds[0] = x_idx*quadlength;
    quadbounds[1] = y_idx*quadlength;
    quadbounds[2] = (x_idx+1)*quadlength - 1;
    quadbounds[3] = (y_idx+1)*quadlength - 1;
    return quadbounds;
}

// allocateBits()
// This bit allocation routine is largely adapted from the "Baseline Wavelet
// Transform Coder Construction Kit"
void
WPTransform::allocateBits()
{
    CoeffSet** coeff = new CoeffSet* [packetTbl->numberNodes];
    basis->traverseBasisTree(Reset);
    int indices[3];
    for (int i = 0; i < packetTbl->numberNodes; i++)
    {
        basis->traverseBasisTree(All_Nodes);
        basis->curNodeParams(indices);
        d = indices[0];
        x = indices[1];
        y = indices[2];
        coeff[indexer->getPacketNode(d, x, y)->index] =
            indexer->getPacketNode(d, x, y)->data;
    }
    Real* optRDJ = new Real [3*(packetTbl->numberNodes)];
    // Assuming resulting best basis has same no. nodes as a wavelet xform.
    budget -= (basis->maxDepth()*3 + 1)*4; // subtr. off approx. header info

    // optimum Allocate
    Real bitBudget = 8*budget;
    Real lambda, lambdaLow, lambdaHigh;
    Real rateLow, rateHigh, curRate;
    Real distLow, distHigh, curDist;

    precision = new int [packetTbl->numberNodes];
    for (int j = 0; j < packetTbl->numberNodes; j++) precision[j] = 0;

    // make a BasisTree2D to store best basis at each iteration
    BasisTree2D* tempBasis = new BasisTree2D(packetDepth);

    // Find optimal lambda based on Rate and Dist for current basis
    lambdaLow = 0.0;
    statfile << "Allocating Lambda for Basis" << endl;
    allocateLambda(coeff, packetTbl->numberNodes, lambda, optRDJ);
    tempBasis->setTreeStruct(Full_2D);
    findBestBasis(optRDJ, tempBasis, rateLow, distLow);
    if (rateLow >= bitBudget) // test if max bits is within budget
    {

```

```

lambdaHigh = 1000000.0;
Real lastRateHigh = -1;
do
{
    // added this line to orig code
    lastRateHigh = rateHigh;
    // try to use the smallest possible # of bits
    allocateLambda(coeff, packetTbl->numberNodes, lambdaHigh, optRDJ);
    tempBasis->setTreeStruct(Full_2D);
    findBestBasis(optRDJ, tempBasis, rateHigh, distHigh);
    // if this is still > bitBudget, try again w/ larger lambda
    if (rateHigh > bitBudget && lastRateHigh != rateHigh)
    {
        lambdaLow = lambdaHigh;
        rateLow = rateHigh;
        distLow = distHigh;
        lambdaHigh *= 10.0;
    }
} while (rateHigh > bitBudget && lastRateHigh != rateHigh);
// debug
if (rateHigh > bitBudget) cout << "rateHigh > bitBudget!" << endl;

// give up when changing lambda has no effect on things
if (lastRateHigh != rateHigh)
{
    // Note rateLow will be > rateHigh
    if (rateLow < bitBudget)
    {
        cerr << "Failed to bracket bit budget = " << budget;
        cerr << "rateLow = " << rateLow << "rateHigh = " << rateHigh;
        cerr << endl;
    }
    while (lambdaHigh - lambdaLow > 0.01)
    {
        lambda = (lambdaLow + lambdaHigh)/2.0;
        cout << "lambda: " << lambda << endl;
        allocateLambda(coeff, packetTbl->numberNodes, lambda, optRDJ);
        tempBasis->setTreeStruct(Full_2D);
        findBestBasis(optRDJ, tempBasis, curRate, curDist);
        // record info into status file
        statfile << " lambda: " << lambda;
        statfile << ", RDJ: " << optRate << " " << optDist;
        statfile << " " << optCost << endl;
        if (curRate > bitBudget)
        {
            lambdaLow = lambda;
        }
        else
        {
            lambdaHigh = lambda;
        }
        if (curRate > bitBudget)
        {
            lambda = lambdaHigh;
            allocateLambda(coeff, packetTbl->numberNodes, lambda, optRDJ);
            tempBasis->setTreeStruct(Full_2D);
            findBestBasis(optRDJ, tempBasis, rateLow, distLow);
            statfile << " lambda: " << lambda;
            statfile << ", RDJ: " << optRate << " " << optDist;
            statfile << " " << optCost << endl;
        }
    }
}
}

```

```

}
delete [] optRDJ;
*(basis) = *(tempBasis);
delete tempBasis;
basis->traverseBasisTree(Reset);

// Greedy augmentation
int bestSet, newPrecision = -1;
Real delta, maxDelta, bestDeltaDist, bestDeltaRate = 0;
int bitsLeft = (bitBudget - curRate);
do
{
    bestSet = -1;
    maxDelta = 0;
    // Find best coeff set to augment
    while (1)
    {
        basis->traverseBasisTree(Basis_Nodes);
        if (basis->resetFlag()) break;
        basis->curNodeParams(indices);
        d = indices[0];
        x = indices[1];
        y = indices[2];
        int idx = indexer->getPacketNode(d,x,y)->index;
        for (int j = precision[idx]+1; j < coeff[idx]->nQuant; j++)
        {
            Real deltaRate = coeff[idx]->rate[j] -
                coeff[idx]->rate[precision[idx]];
            Real deltaDist = -(coeff[idx]->dist[j] -
                coeff[idx]->dist[precision[idx]]);
            if (deltaRate != 0 && deltaRate <= (bitsLeft))
            {
                delta = deltaDist / deltaRate;
                if (delta > maxDelta)
                {
                    maxDelta = delta;
                    bestDeltaRate = deltaRate;
                    bestDeltaDist = deltaDist;
                    bestSet = idx;
                    newPrecision = j;
                }
            }
        }
    } // end while
    if (bestSet != -1)
    {
        precision[bestSet] = newPrecision;
        bitsLeft -= bestDeltaRate;
    }
} while (bestSet != -1);

// Print out stats
Real totalRate = 0, totalDist = 0;
int totalData = 0;
printf (" Set   Basis      Precision          Rate              Distortion\n");
while (1)
{
    int idx;
    basis->traverseBasisTree(Basis_Nodes);
    if (basis->resetFlag()) break;
    basis->curNodeParams(indices);
    d = indices[0];
    x = indices[1];

```



```

    y = indices[2];
    idx = indexer->getPacketNode(d,x,y)->index;
    printf ("%3d: (%1d,%2d,%2d)    %2d    %9.2f    %5.2f    %11.2f\n",
        %7.2f\n",
        idx, d, x, y, precision[idx],
        coeff[idx]->rate[precision[idx]],
        coeff[idx]->rate[precision[idx]]/(Real)coeff[idx]->nData,
        coeff[idx]->dist[precision[idx]],
        coeff[idx]->dist[precision[idx]]/(Real)coeff[idx]->nData);
    totalRate += coeff[idx]->rate[precision[idx]];
    totalDist += coeff[idx]->dist[precision[idx]];
    totalData += coeff[idx]->nData;
}
// special case
if (packetDepth == 0)
{
    int idx = indexer->getPacketNode(d,x,y)->index;
    printf ("%3d: (%1d,%2d,%2d)    %2d    %9.2f    %5.2f    %11.2f\n",
        %7.2f\n",
        idx, d, x, y, precision[idx],
        coeff[idx]->rate[precision[idx]],
        coeff[idx]->rate[precision[idx]]/(Real)coeff[idx]->nData,
        coeff[idx]->dist[precision[idx]],
        coeff[idx]->dist[precision[idx]]/(Real)coeff[idx]->nData);
    totalRate += coeff[idx]->rate[precision[idx]];
    totalDist += coeff[idx]->dist[precision[idx]];
    totalData += coeff[idx]->nData;
}

// Precisions may have changed because of greedy augmentation,
// so optimum rate/dist/cost must be updated.
optRate = totalRate;
optDist = totalDist;
optCost = optDist + lambda*optRate;

Real rms = sqrt(totalDist/(Real)totalData);
Real psnr = 20.0 * log(255.0/rms)/log(10.0);
printf ("\n");
printf ("total rate = %g\n", totalRate/8.0);
printf ("total dist = %g\n", totalDist);
printf ("total coeffs = %d\n", totalData);
printf ("RMS error = %g\n", rms);
printf ("PSNR (transform domain) = %g\n", psnr);
cout << endl;
}

// allocateLambda()
//
void
WPTransform::allocateLambda(CoeffSet** coeff, int nSet,
    Real lambda, Real*& optRDJ)
{
    int i, j;
    Real G, minG, minRate, minDist;
    // want to minimize G = distortion + lambda * rate
    // loop through all rate-distortion curves
    for (i = 0; i < nSet; i++)
    {
        minG = minRate = minDist = MaxReal;
        for (j = 0; j < coeff[i]->nQuant; j++)
        {
            G = coeff[i]->dist[j] + lambda*coeff[i]->rate[j];
            if (G < minG)

```

```

    {
        minG = G;
        minRate = coeff[i]->rate[j];
        minDist = coeff[i]->dist[j];
        precision[i] = j;
        //debug
        if (minG < 0 || minRate < 0 || minDist < 0)
        {
            break;
            cout << "Negative values in RDJ" << endl;
        }
    }
}
*(optRDJ+3*i) = minRate;
*(optRDJ+3*i+1) = minDist;
*(optRDJ+3*i+2) = minG;
}

// overloaded function
void
WPTransform::allocateLambda(CoeffSet** coeff, Real lambda,
                           Real*& optRDJ)
{
    // statfile << "Allocating Lambda" << endl;
    int j;
    int indices[3];
    Real G, minG, minRate, minDist;
    // want to minimize  $G = \text{distortion} + \lambda * \text{rate}$ 
    // loop through all rate-distortion curves
    while (1)
    {
        basis->traverseBasisTree(Basis_Nodes);
        if (basis->resetFlag()) break;
        basis->curNodeParams(indices);
        d = indices[0];
        x = indices[1];
        y = indices[2];
        int idx = indexer->getPacketNode(d,x,y)->index;
        minG = minRate = minDist = MaxReal;
        for (j = 0; j < coeff[idx]->nQuant; j++)
        {
            G = coeff[idx]->dist[j] + lambda*coeff[idx]->rate[j];
            if (G < minG)
            {
                minG = G;
                minRate = coeff[idx]->rate[j];
                minDist = coeff[idx]->dist[j];
                precision[idx] = j;
                //debug
                if (minG < 0 || minRate < 0 || minDist < 0)
                {
                    break;
                    cout << "Negative values in RDJ" << endl;
                }
            }
        }
        *(optRDJ+3*idx) = minRate;
        *(optRDJ+3*idx+1) = minDist;
        *(optRDJ+3*idx+2) = minG;
    }
    // special case
    if (packetDepth == 0)

```

```

{
    int idx = indexer->getPacketNode(d,x,y)->index;
    minG = minRate = minDist = MaxReal;
    for (j = 0; j < coeff[idx]->nQuant; j++)
    {
        G = coeff[idx]->dist[j] + lambda*coeff[idx]->rate[j];
        if (G < minG)
        {
            minG = G;
            minRate = coeff[idx]->rate[j];
            minDist = coeff[idx]->dist[j];
            precision[idx] = j;
        }
    }
    *(optRDJ) = minRate;
    *(optRDJ + 1) = minDist;
    *(optRDJ + 2) = minG;
}
}

void
WPTransform::findBestBasis(const Real*& optRDJ, BasisTree2D* tempBasis,
                          Real& optimumRate, Real& optimumDist)
{
    optimumRate = optimumDist = 0.0;
    // Populate optimal rate/distortion/cost values in each node
    tempBasis->traverseBasisTree(Reset);
    int indices[3];
    int idx = 0;
    while (1)
    {
        tempBasis->traverseBasisTree(All_Nodes);
        if (tempBasis->resetFlag()) break;
        tempBasis->curNodeParams(indices);
        d = indices[0];
        x = indices[1];
        y = indices[2];
        idx = indexer->getPacketNode(d, x, y)->index;
        indexer->getPacketNode(d, x, y)->optRate = *(optRDJ + 3*idx);
        indexer->getPacketNode(d, x, y)->optDist = *(optRDJ + 3*idx + 1);
        indexer->getPacketNode(d, x, y)->optCost = *(optRDJ + 3*idx + 2);
    }
    if (packetDepth == 0) // special case
    {
        optRate = indexer->getPacketNode(0, 0, 0)->optRate = *(optRDJ);
        optDist = indexer->getPacketNode(0, 0, 0)->optDist = *(optRDJ + 1);
        optCost = indexer->getPacketNode(0, 0, 0)->optCost = *(optRDJ + 2);
        optimumRate = optRate;
        optimumDist = optDist;
        return; // there are no children
    }
    BasisTree2D* newBasis = new BasisTree2D(packetDepth);
    newBasis->setTreeStruct(Zero_2D);
    tempBasis->traverseBasisTree(Reset);
    // loop until out of leaves
    while (1)
    {
        tempBasis->traverseBasisTree(Bottom_to_Top);
        tempBasis->curNodeParams(indices);
        if (indices[0] < packetDepth || basis->resetFlag()) break;
    }
    Real costChildren = 0.0;
    int i = 0;

```

```

do
{
    tempBasis->curNodeParams(indices);
    d = indices[0];
    x = indices[1];
    y = indices[2];
    costChildren =
        indexer->getPacketNode(d, x, y)->child_LL->optCost +
        indexer->getPacketNode(d, x, y)->child_LH->optCost +
        indexer->getPacketNode(d, x, y)->child_HL->optCost +
        indexer->getPacketNode(d, x, y)->child_HH->optCost;
    if (costChildren < indexer->getPacketNode(d, x, y)->optCost)
    {
        indexer->getPacketNode(d, x, y)->optCost = costChildren;
        indexer->getPacketNode(d, x, y)->optRate =
            indexer->getPacketNode(d, x, y)->child_LL->optRate +
            indexer->getPacketNode(d, x, y)->child_LH->optRate +
            indexer->getPacketNode(d, x, y)->child_HL->optRate +
            indexer->getPacketNode(d, x, y)->child_HH->optRate;
        indexer->getPacketNode(d, x, y)->optDist =
            indexer->getPacketNode(d, x, y)->child_LL->optDist +
            indexer->getPacketNode(d, x, y)->child_LH->optDist +
            indexer->getPacketNode(d, x, y)->child_HL->optDist +
            indexer->getPacketNode(d, x, y)->child_HH->optDist;
        newBasis->setNode(d, x, y, 1);
    }
    else
    {
        newBasis->setNode(d, x, y, 0);
    }
    if (packetDepth == 1) break; // only one iteration needed
    tempBasis->traverseBasisTree(Bottom_to_Top);
    if (tempBasis->resetFlag())
    {
        i = 1; // go for one last iteration at (0,0,0)
    }
} while (tempBasis->resetFlag() || i == 0);
tempBasis->traverseBasisTree(Reset); // for neatness
// value at top node is cumulative Rate/Dist for best basis
optimumRate = optRate = indexer->getPacketNode(0, 0, 0)->optRate;
optimumDist = optDist = indexer->getPacketNode(0, 0, 0)->optDist;
optCost = indexer->getPacketNode(0, 0, 0)->optCost;
*(tempBasis) = *(newBasis);
delete newBasis;
tempBasis->traverseBasisTree(Reset);
}

// pruneTree()
//
void
WPTransform::pruneTree()
{
    int indices[3];
    basis->traverseBasisTree(Reset);
    // statfile << "Pruning children" << endl;
    // Prune nodes. Nodes are deleted recursively due to linked list struct.
    while (1)
    {
        basis->traverseBasisTree(Basis_Nodes);
        if (basis->resetFlag()) break;
        basis->curNodeParams(indices);
        d = indices[0];
        x = indices[1];

```

```

y = indices[2];
// Deletes linked list recursively until it reaches NULL
// therefore it is important to set leaf nodes to NULL so that when
// destructor for PacketTable is called, there will be no memory
// collision.
delete indexer->getPacketNode(d, x, y)->child_LL;
indexer->getPacketNode(d, x, y)->child_LL = NULL;
delete indexer->getPacketNode(d, x, y)->child_LH;
indexer->getPacketNode(d, x, y)->child_LH = NULL;
delete indexer->getPacketNode(d, x, y)->child_HL;
indexer->getPacketNode(d, x, y)->child_HL = NULL;
delete indexer->getPacketNode(d, x, y)->child_HH;
indexer->getPacketNode(d, x, y)->child_HH = NULL;
// statfile << " (" << d << ", " << x << ", " << y << ")" << endl;
}
// statfile << "End Pruning" << endl;
}

// invert()
void
WPTransform::invert(Image* transformedImage)
{
    int indices[3];
    int bounds[4];
    int h, v;
    basis->traverseBasisTree(Reset);
    while(1)
    {
        basis->traverseBasisTree(Basis_Nodes);
        if (basis->resetFlag()) break;
        basis->curNodeParams(indices); // transfer basis indices to obj
        d = indices[0];
        x = indices[1];
        y = indices[2];
        quad(d, x, y, hsize, bounds); // assume square quad
        h = *(bounds+2) - *(bounds) + 1;
        v = *(bounds+3) - *(bounds+1) + 1;
        for (int i = *(bounds+1); i < (*(bounds+3)+1); i++)
        {
            for (int j = *(bounds); j < (*(bounds+2)+1); j++)
            {
                // getPacketNode points to a Coefficient* ("data") which points to
                // ("data") a Real* which points to the actual data contents
                *(value + hsize*i + j) = *(indexer->getPacketNode(d,x,y)->
                    data->data + h*(i-*(bounds+1))+(j-*(bounds)));
            }
        }
    }
    basis->traverseBasisTree(Reset);
    // Now do inverse transform
    if (basis->Node(0,0,0) != 0) // inverting needed only if there are branches
    {
        do
        {
            do
            {
                basis->traverseBasisTree(Bottom_to_Top);
                basis->curNodeParams(indices);
                d = indices[0];
                x = indices[1];
                y = indices[2];
            } while (basis->Node(d,x,y) == 0);
            quad(d, x, y, hsize, bounds);
        }
    }
}

```

```

    h = *(bounds+2) - *(bounds) + 1;
    v = *(bounds+3) - *(bounds+1) + 1;
    Real* temp = new Real [h*v];
    for (int i = *(bounds+1); i < (*(bounds+3)+1); i++)
    {
        for (int j = *(bounds); j < (*(bounds+2)+1); j++)
        {
            *(temp + h*(i-*(bounds+1)) + (j-*(bounds))) =
                *(value + hsize*i + j);
        }
    }
    Real* temp_out = new Real [h*v];
    if (temp_out == 0) cout << "didn't allocate"; // check for error
    wavelet->invert2d(temp, temp_out, h, v, 1, symmetric);
    delete [] temp;
    // write back to image
    for (int k = 0; k < v; k++)
    {
        for (int j = 0; j < h; j++)
        {
            *(value+hsize*(k*(bounds+1))+(j*(bounds))) =
                *(temp_out+h*k+j);
        }
    }
    delete [] temp_out;
} while (basis->resetFlag() == 0);
}
else // just copy values directly to output image
{
    for (int j = 0; j < hsize*vsize; j++)
    {
        *(transformedImage->value + j) = *(indexer->getPacketNode(d,x,y)->
                                            data->data + j);
    }
    return;
}
for (int i = 0; i < hsize*vsize; i++)
{
    *(transformedImage->value + i) = *(value + i);
}
}

// decode()
void
WPTransform::decode(Decoder* decoder)
{
    basis->traverseBasisTree(Reset);
    int indices[3];
    int idx = 0;
    int* precis = new int [basis->numAllNodes()];
    while (1)
    {
        basis->traverseBasisTree(Basis_Nodes);
        if (basis->resetFlag()) break;
        basis->curNodeParams(indices);
        d = indices[0];
        x = indices[1];
        y = indices[2];
        idx = indexer->getPacketNode(d, x ,y)->index;
        indexer->getPacketNode(d, x, y)->data->
            readHeader(decoder, precis[idx]);
    }
    basis->traverseBasisTree(Reset);
}

```

```

while (1)
{
    basis->traverseBasisTree(Basis_Nodes);
    if (basis->resetFlag()) break;
    basis->curNodeParams(indices);
    d = indices[0];
    x = indices[1];
    y = indices[2];
    idx = indexer->getPacketNode(d, x, y)->index;
    indexer->getPacketNode(d, x, y)->data->
        decode(decoder, precis[idx]);
}
if (packetDepth == 0) // special case
{
    indexer->getPacketNode(0, 0, 0)->data->
        readHeader(decoder, precis[0]);
    indexer->getPacketNode(0, 0, 0)->data->
        decode(decoder, precis[0]);
}
delete [] precis;
}

// encode()
void
WPTTransform::encode(Encoder* encoder)
{
    basis->traverseBasisTree(Reset);
    int indices[3];
    int idx = 0;
    while (1)
    {
        basis->traverseBasisTree(Basis_Nodes);
        if (basis->resetFlag()) break;
        basis->curNodeParams(indices);
        d = indices[0];
        x = indices[1];
        y = indices[2];
        idx = indexer->getPacketNode(d, x, y)->index;
        indexer->getPacketNode(d, x, y)->data->
            writeHeader(encoder, precision[idx]);
    }

    basis->traverseBasisTree(Reset);
    statfile << "Encoding Basis Node" << endl;
    while (1)
    {
        basis->traverseBasisTree(Basis_Nodes);
        if (basis->resetFlag()) break;
        basis->curNodeParams(indices);
        d = indices[0];
        x = indices[1];
        y = indices[2];
        idx = indexer->getPacketNode(d, x, y)->index;
        indexer->getPacketNode(d, x, y)->data->
            encode(encoder, precision[idx]);
        statfile << " (" << d << ", " << x << ", " << y << ") ";
        // statfile << "node cost " << indexer->getPacketNode(d, x, y)->optCost;
        // statfile << endl;
    }
    if (packetDepth == 0)
    {
        indexer->getPacketNode(0, 0, 0)->data->
            writeHeader(encoder, precision[0]);
    }
}

```

```

        indexer->getPacketNode(0, 0, 0)->data->
            encode(encoder, precision[0]);
    }
    statfile << "End Encoding" << endl;
//  statfile << "Total Cost: " << optCost << endl;
}

/////////////////////////////////////////////////////////////////

// Constructor
PacketNode::PacketNode():
    optRate(0), optDist(0), optCost(0),
    index(0),
    size(0),
    depth(0)
{
    child_LL = NULL;
    child_LH = NULL;
    child_HL = NULL;
    child_HH = NULL;
    data = NULL;
    err = NULL;
    entropy = NULL;
    quant = NULL;
}

// Destructor
PacketNode::~~PacketNode()
{
    if (child_LL != 0) delete child_LL;
    if (child_LH != 0) delete child_LH;
    if (child_HL != 0) delete child_HL;
    if (child_HH != 0) delete child_HH;
    if (data != 0) delete data;
    if (err != 0) delete err;
    if (entropy != 0) delete entropy;
    if (quant != 0) delete quant;
}

/////////////////////////////////////////////////////////////////

// Copy Constructor
PacketIndexer::PacketIndexer(const PacketIndexer*& idx)
{
    firstNode = idx->firstNode;
    basis = idx->basis;
}

// getPacketNode
//
PacketNode*
PacketIndexer::getPacketNode(const int& d, const int& x, const int& y)
{
    //  if (d < 0)
    //  {
    //      cerr << "Tried to get node out of current range";
    //  }
    int (* reg)[2] = new int[d][2];
    int tmp[2] = {x, y};
    for (int i = 0; i < d; i++)
    {
        reg[i][0] = tmp[0]*2;
    }

```



```

        reg[i][1] = tmp[1]*2;
        tmp[0] = tmp[0]/2;
        tmp[1] = tmp[1]/2;
    }
    PacketNode* ptr;
    ptr = firstNode;
    for (int j = d-1; j >= 0; j--)
    {
        switch ((reg[j][0])*2 + reg[j][1])
        {
            case 0:
                ptr = ptr->child_LL;
                break;
            case 1:
                ptr = ptr->child_LH;
                break;
            case 2:
                ptr = ptr->child_HL;
                break;
            case 3:
                ptr = ptr->child_HH;
                break;
        }
    }
    delete [] reg;
    return ptr;
}

////////////////////////////////////
// Constructor
PacketTable::PacketTable():
    totalDepth(0),
    numberNodes(0)
{
    firstNode = new PacketNode;
}

PacketTable::~~PacketTable()
{
    delete firstNode;
}

// addNode()
int
PacketTable::addNode(Real* data, int index, int size, int d, int x, int y,
                    int monolayer, int capacity, int paramPrecision,
                    int nQuant, Real minStepSize, PacketIndexer*& indexer)
{
    if (d > totalDepth+1)
    {
        cerr << "Cannot add node beyond layer just below current depth";
        return 0;
    }
    int band;
    band = (2*(x%2) + (y%2)); // calculate if LL, LH, HL, or HH
    if (numberNodes == 0) // if adding first node
    {
        firstNode->index = index;
        firstNode->size = size;
        firstNode->depth = d;
        LpError* tmp = new LpError(2.0);
        firstNode->err = tmp;
    }

```

```

if (monolayer)
{
    EntropyCoder* tmp_entropy = new EscapeCoder(capacity);
    firstNode->entropy = tmp_entropy;
    UniformQuant* tmp_quant =
        new UniformQuant((MonoLayerCoder*)firstNode->entropy,
                        paramPrecision, d || x || y,
                        firstNode->err);
    firstNode->quant = tmp_quant;
}
else
{
    EntropyCoder* tmp_entropy =
        new LayerCoder(nQuant, d || x || y, capacity);
    firstNode->entropy = tmp_entropy;
    LayerQuant* tmp_quant =
        new LayerQuant((MultiLayerCoder*)firstNode->entropy,
                        paramPrecision, d || x || y, nQuant,
                        firstNode->err);
    firstNode->quant = tmp_quant;
}
CoeffSet* tmp_data = new CoeffSet(data, size, firstNode->quant);
firstNode->data = tmp_data;
firstNode->data->getRateDist(nQuant, minStepSize);
}
else // adding after first node
{
    PacketNode* parentNodePtr = indexer->getPacketNode(d-1, x/2, y/2);
    PacketNode* pn_ptr;
    switch (band)
    {
        case 0:
            parentNodePtr->child_LL = new PacketNode;
            pn_ptr = parentNodePtr->child_LL;
            break;
        case 1:
            parentNodePtr->child_LH = new PacketNode;
            pn_ptr = parentNodePtr->child_LH;
            break;
        case 2:
            parentNodePtr->child_HL = new PacketNode;
            pn_ptr = parentNodePtr->child_HL;
            break;
        case 3:
            parentNodePtr->child_HH = new PacketNode;
            pn_ptr = parentNodePtr->child_HH;
            break;
    }
    pn_ptr->index = index;
    pn_ptr->size = size;
    pn_ptr->depth = d;
    LpError* tmp_err = new LpError(2.0);
    pn_ptr->err = tmp_err;
}
if (monolayer)
{
    EntropyCoder* tmp_entropy = new EscapeCoder(capacity);
    pn_ptr->entropy = tmp_entropy;
    UniformQuant* tmp_quant =
        new UniformQuant((MonoLayerCoder*)pn_ptr->
                        entropy,
                        paramPrecision, d || x || y,
                        pn_ptr->err);
}

```

```

        pn_ptr->quant = tmp_quant;
    }
    else
    {
        EntropyCoder* tmp_entropy = new LayerCoder(nQuant, d || x || y,
                                                    capacity);
        pn_ptr->entropy = tmp_entropy;
        LayerQuant* tmp_quant =
            new LayerQuant((MultiLayerCoder*)pn_ptr->
                           entropy,
                           paramPrecision, d || x || y, nQuant,
                           pn_ptr->err);
        pn_ptr->quant = tmp_quant;
    }
    CoeffSet* tmp_data = new CoeffSet(data, size, pn_ptr->quant);
    pn_ptr->data = tmp_data;
    pn_ptr->data->getRateDist(nQuant, minStepSize);
}
if (d > totalDepth) totalDepth = d;
numberNodes++;
cout << numberNodes << " " << flush;
return 0;
}

```