Low-complexity Block dividing Coding Method for Image Compression using Wavelets

A thesis presented in partial fulfillment of the requirements for the degree of

Master of Engineering

in

Computer Systems Engineering

at Massey University,

Palmerston North

New Zealand

Jihai Zhu

2007

Abstract

Image coding plays a key role in multimedia signal processing and communications. JPEG2000 is the latest image coding standard, it uses the EBCOT (Embedded Block Coding with Optimal Truncation) algorithm. The EBCOT exhibits excellent compression performance, but with high complexity. The need to reduce this complexity but maintain similar performance to EBCOT has inspired a significant amount of research activity in the image coding community.

Within the development of image compression techniques based on wavelet transforms, the EZW (Embedded Zerotree Wavelet) and the SPIHT (Set Partitioning in Hierarchical Trees) have played an important role. The EZW algorithm was the first breakthrough in wavelet based image coding. The SPIHT algorithm achieves similar performance to EBCOT, but with fewer features. The other very important algorithm is SBHP (Sub-band Block Hierarchical Partitioning), which attracted significant investigation during the JPEG2000 development process.

In this thesis, the history of the development of wavelet transform is reviewed, and a discussion is presented on the implementation issues for wavelet transforms. The above mentioned four main coding methods for image compression using wavelet transforms are studied in detail. More importantly the factors that affect coding efficiency are identified.

The main contribution of this research is the introduction of a new low-complexity coding algorithm for image compression based on wavelet transforms. The algorithm is based on block dividing coding (BDC) with an optimised packet assembly. Our extensive simulation results show that the proposed algorithm outperforms JPEG2000 in lossless coding, even though it still leaves a narrow gap in lossy coding situations

Acknowledgements

This thesis would not have been written without the help and support that I received. I would like to take this opportunity to express my sincerest thanks to every one who helped me.

I thank my principal supervisor Dr Bing Du and associate supervisor Professor Richard Harris and Dr Xiang Gui for providing the research topic that I felt was both interesting from a research perspective and had the potential to produce marketable results. They have always been patient, knowledgeable, and supportive, which were all invaluable in completing this thesis. I also have enjoyed the fact that they always have confidence in my abilities, and me even when I had difficulties.

I would also like to thank Mr. John Hayward, for his patient proof-reading and grammatical corrections.

Finally, I would like to thank my family for their encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

1.1 Overview

Multimedia processing and communication have become more and more pervasive in our daily life. Image coding techniques play a key role in efficient image representation, storage and delivery of images over a telecommunication network.

Currently, two kinds of transforms, DCT (Discrete Cosine Transform), and wavelet transform are widely employed for image compression. The DCT has been adopted in the JPEG standard, while wavelet transforms have been incorporated into the JPEG2000 standard [1]. The DCT based transform techniques are well established and easily implemented. However, the block artefacts inherent with a DCT transform become unacceptable at very low bit rates. Rather than incrementally improving on DCT based techniques, image coding techniques based on wavelet transforms are an entirely new way of performing compression. Although it is more complicated to implement, wavelet based image coding has two advantages over DCT. Firstly, it can overcome the presence of block artefacts in very low bit-rate image coding, and secondly it can be used for both lossy to lossless compression.

In image coding based on wavelet transforms, the discrete wavelet transform is first applied to the source image data. The transform coefficients are then quantized and entropy coded before forming the final output bit stream. One of the beneficial properties of a wavelet transform, relative to data compression, is that it tends to compact the energy of the input into a relatively small number of wavelet coefficients. To represent these coefficients efficiently utilizing the multi-resolution characteristic of the wavelet transform, many algorithms have been developed. The most important three among these algorithms are EZW (Embedded Zero-tree Wavelet) [2], SPIHT (Set Partitioning In Hierarchical Trees) [3] and EBCOT (Embedded Block Coding with Optimized Truncation) [4]. EBCOT has become the basis of the JPEG2000 international standard. The other very important algorithm is SBHP (Sub-band Block Hierarchical Partitioning) [5], which attracted significant levels of attention during the JPEG2000 development process.

This thesis studies coding methods based on wavelets. The wavelet transform has proven to be very useful for image compression as previously mentioned. This thesis

draws on ideas from the above coding methods. The coding algorithm that we propose is motivated by the above mentioned methods and experimental observations.

The contribution of this thesis is twofold:

1. It identifies the factors that affect the performance of image coding;
2. It devises a low-complexity coding method. The output of the designed algorithm is an embedded bit-stream with desirable features.

The performance of the new algorithm has been confirmed by our simulation results, which show that it outperforms JPEC 2000 in lossless coding. This makes it a good candidate to be incorporated into a standard for future development.

1.2 Outline

This thesis is organized into eight chapters and one appendix. Chapters 2 and 3 provide background information about wavelets and previous coding methods. The remaining five chapters present our research results. The appendix provides some source code for our coding method.

More specifically, chapter 2 firstly explores the developing history of wavelet transforms, discusses the implementation issues for wavelet transforms with some examples, and explains why wavelet transformation is very suitable for image compression. Chapter 3 investigates the four main kinds of coding method in detail. Chapter 4 analyses what factors affect the final compressed bit-stream emanating from the coding processes, and sets up a model to represent the affected processes. Some suggestions are also given. Chapter 5 describes the principles of the block dividing coding algorithm and shows how BDC can efficiently encode a significance bit-plane of wavelet coefficients using a simple example. Chapter 6 describes the optimized assembling coding algorithm. Chapter 7 reports the experimental results for both lossy and lossless image compression and discusses the results in various ways. Some reconstructed images are also displayed and some comparison of this work to previous work is given. The conclusions are presented in Chapter 8.

The algorithms resulting from this research project have been implemented using MATLAB (version 7.0) and some of the source code written for this project can be found in the appendix.

1.3 Notation

Some useful and traditional notation used in this thesis is summarized below:

1D	One dimensional
2D	Two dimensional
DCT	Discrete cosine transform
EZW	Embedded zerotree wavelet
SPIHT	Set partition in hierarchical trees
EBCOT	Embedded Block Coding with Optimized Truncation of the Embedded Bit-stream
SBHP	Sub-band hierarchical partitioning
JPEG	Joint Photographic Experts Group
bpp	Bits per pixel
MSE	Mean square error
bs	Block size.
PSNR	Peak-Signal-to-Noise Ratio
BDC	Block dividing coding
LIS	List of insignificant sets
LIP	List of insignificant pixels
LSP	List of significant pixels
ROI	Region of interest

2 Wavelet Transforms

This chapter briefly explores the development history of wavelet and sub-band transforms, discusses the implement issues of discrete wavelet transforms (DWT) on images and associated operations including the convolution method, lifting scheme, extension policies, scalar quantization, and the features of DWT, and explains why DWT is suitable for image compression.

2.1 History of Wavelet Transforms and Applications

2.1.1 Wavelet Transform and Sub-band Transforms

Before the birth of the wavelet transform, there was a famous transformation, that was known as the Fourier transform, was invented by Fourier in 1807. The Fourier transform is very useful in many applications as it breaks down a signal into continuous sinusoids of different frequencies. For many signals, a Fourier transform is particularly suitable because the signal's frequency content is of great importance. So why do we need a wavelet transform? A Fourier transform has a serious drawback, in transforming to the frequency domain, time information is lost. When looking at a Fourier transform of a signal, it is impossible to tell when an event took place. If the signal properties do not change much over time -- that is, if it is what is called a stationary signal -- this drawback is not very important. However, most attractive signals contain numerous nonstationary or transient characteristics: drift, trends, abrupt changes, beginnings and ends of events. These characteristics are often the most important part of the signal, and a Fourier transform is not suitable to detect them.

After almost one hundred years, a different solution was proposed by Haar in 1909. He replaced the sine and cosine functions of the Fourier transform by using another orthonormal basis, now commonly called the Haar basis. The Haar basis is the simplest example to date of a wavelet basis.

The wavelet transform analyses a signal with a windowing technique with variable-sized regions, a long time region is used when we want more precise low-frequency information, and a shorter region is used when we want high-frequency information. A Fourier transform breaks up a signal into sine waves of various frequencies. Similarly, a

wavelet transform involves the breaking up of a signal into shifted and scaled versions of the mother wavelet.

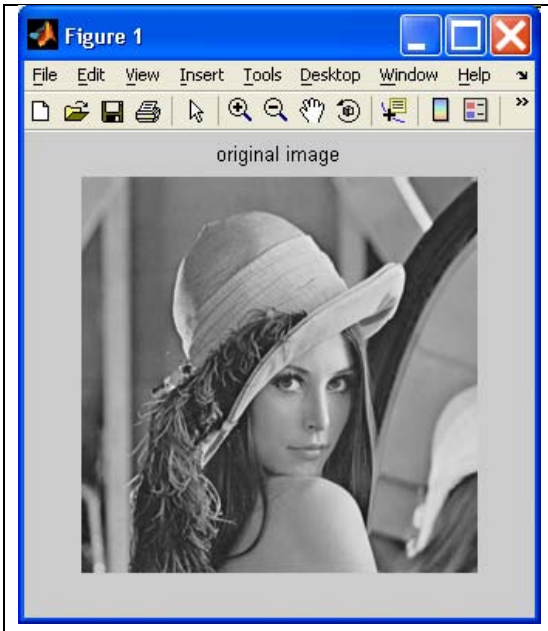
From that time until now, a lot of work has been given to the study of the wavelet basis (mother wavelet). Interested readers can find out more details in reference [6].

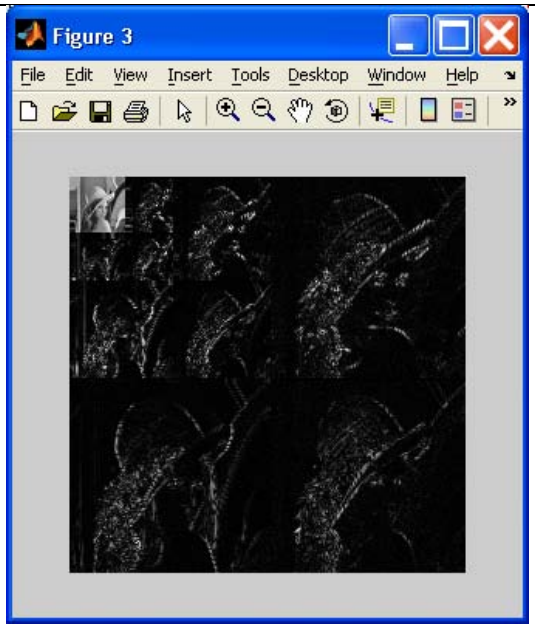
Although the research about the wavelet basis was undertaken many years ago, the term ‘wavelet’ did not come into use until the 1980s. Before that time, wavelet theory was really a disjoint set of ideas from many areas. In the mid-to-late 1980s, a revolution in wavelet theory occurred because of several important discoveries. This revolution served to draw together concepts from many different areas of mathematics and engineering resulting in a unified theory for the study of wavelet systems. In 1984, the term “wavelet” was introduced by Grossman and Morlet [7]. A “wavelet” literally means a *small wave*.

At the same time, another area of research called filter analysis (or sub-band transformation) was developed in the 1970s. The first developments in this area were made by Crochiere, Webber, and Flanagan [8] in 1976 for the digital processing of speech and audio signal with polyphase filters.

Many signals are made up of a large range of frequencies. The low-frequency content is the most important part. It is what gives the signal its characteristics. The high-frequency content, on the other hand, imparts flavour or tone. Consider the human voice, if you remove the high-frequency components, the voice sounds different, but you can still tell what is being said. However, if you eliminate enough of the low-frequency components, you hear gibberish.

When using wavelet transform to image processing, the effect is different from the DCT (Discrete Cosine Transform) transform. The wavelet transform is applied to the whole image unlike DCT applies only to small block (generally 8x8). After the DCT transform, we cannot see any meaning from the coefficients, because the time information has been lost. After the wavelet transform, we can see what the wavelet transform has done. Figure 1 shows the effect of this transformation.





Original image	Wavelet coefficients after transformation

Figure 1 A wavelet example

From the picture on the right, we can see that the most energy was compacted in the lowest sub-band (top-left sub-band). The small picture in every sub-band represents the information of that sub-band. We can see that the information represents what happened and where it happened.

Although the research on sub-band transformation began considerably later than that of the wavelet transform, the development of the sub-band transform and its applications in different areas was very rapid. The QMF filter (Quadrature Mirror Filter) was introduced by Esteban and Galand [9], and the Perfect Reconstruction (PR) filter was first introduced by Smith, Barnwell [10]. In 1986, Wood & O'Neil [11] first used a sub-band transform to image coding. Some of them can be used to reconstruct the signal perfectly. The number of articles relating to this filter is very numerous. If the reader is interested, the book [6] is a very useful reference.

In JPEG2000, there are many filters that can be chosen, the default filter with real-valued coefficients is the Daubechies (9,7) filter for lossy compression, while the default filter for a lossless compression is the Legall (5,3) filter.

2.1.2 Connection between Wavelet Transform and Sub-band Transform

The wavelet transform and sub-band transform had been evolving independently for many years until 1988, when a tremendous breakthrough in wavelet analysis was made by Daubechies [12] and Mallat [13]. In 1989, Mallat presented the theory of multi-resolution analysis, and it later became known as the Mallat algorithm. This work provided a unifying framework for the study of wavelet systems together with many previously disjoint ideas. In 1990, Daubechies pointed out the connection between the wavelet and sub-band transform. Actually a sub-band transform is nothing but a kind of wavelet transform. At this stage, there was an explosion of interest in wavelet transformation because of the connection between pure mathematics and applications in the digital signal processing area. This area was also extended by pure mathematicians working with approximation theory, quantum physicists, numerical analysts, computer graphics developers, statisticians, image and video coding researchers, and workers in many other fields.

2.1.3 Applications of Wavelet Transform

All of the possible applications of wavelet transforms and sub-band transforms are too extensive to be covered here. One might compare possible applications of wavelet transforms with that of Fourier transforms. The Fourier transform has applications in nearly every field of science and engineering. Similarly, wavelet transforms already have found applications in nearly all branches of science and engineering, including mathematics, physics, electrical engineering, geophysics, bioengineering, and computer vision. My major application field is that of the image coding.

2.2 The Implement Issues of Wavelet Transform

The forward discrete wavelet transform (FDWT) is a main function in the encoder side, it gradually compacts the most energy to the lowest sub-band. The transformed signal can be perfectly or almost perfectly reconstructed by using an inverse discrete wavelet transforms (IDWT). This section discusses the implementation issues for wavelet transforms.

2.2.1 One-dimension Wavelet Transform

To apply a wavelet transform to a signal it is necessary to simply pass the signal through a set of digital filters, in which the coefficients of the wavelet transform are simply the output of the convolution between the input signal and the filters. The set of filters usually consists of two parts: a high-pass filter and a low-pass filter, once a signal goes through one filter, for example, the high-pass filter, the output only includes the high-frequency information of that signal. Similarly, the output from the low-pass filter only contains low-frequency information. The fitering operations are followed by a down-sampling using a factor of two. We call those outputs 'wavelet coefficients'. Figure 2 shows this processing of 1-D (one dimension) with 2-band sub-band transformation.

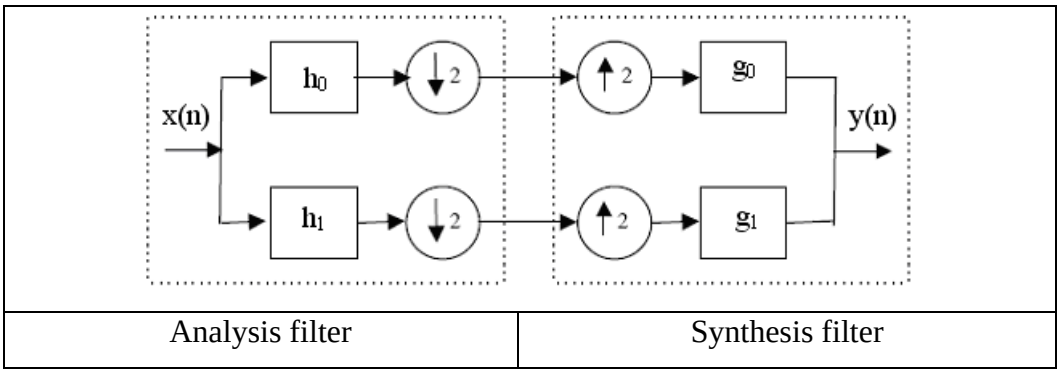


Figure 2 Convolution implementation of the one-dimensional sub-band transform

After the 1-D 2-band decomposition of the signal, the output of the low-pass filter can be subjected to a further stage of two-band decomposition in order to achieve additional decorrelation. In comparison, there is little to be gained by further decomposition with the high-pass output. In most DWT decompositions of an image, only the output of the low-pass filter is further decomposed to produce dyadic or octave decomposition.

2.2.2 Two-dimension Wavelet Transform

Figure 3 shows how to use a wavelet transform to decompose an image. After applying the wavelet transform to every column of the image, we get two outputs including the L-sub-band and H-sub-band. Then we apply the wavelet transform to every row of these two outputs separately. The result contains four parts; they are referred to as LL, HL, LH and HH sub-bands. Such a process is called the one-level two dimensional wavelet transform. If we repeatedly do such transforms to the LL sub-band, then we can

get a multi-level wavelet transform. The left-bottom diagram in Figure 3 shows the final result of a 2-level wavelet transform.

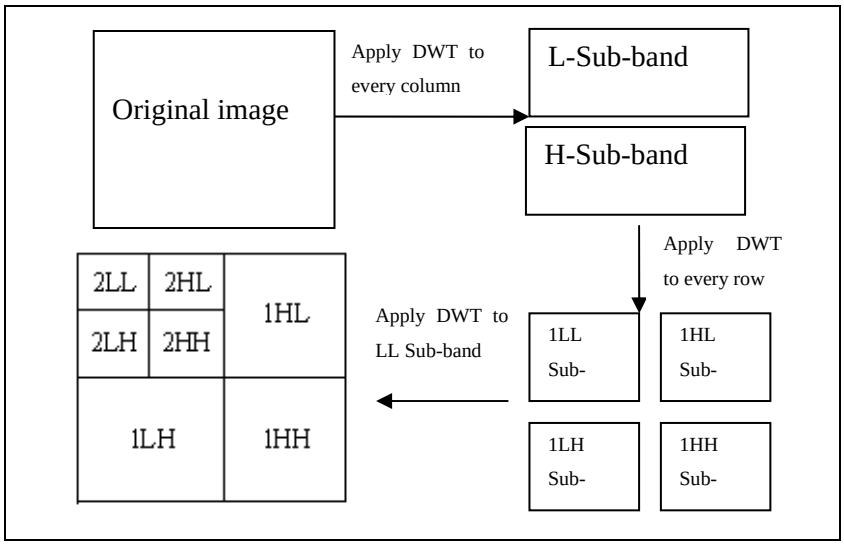


Figure 3 Two-level wavelet transforms

The DWT has several characteristics that make it suitable for image coding. Firstly, after the wavelet transform, most coefficients are small or zero. This characteristic provides the opportunity for image compression. The second feature is that DWT naturally has multi-resolution scalability. Thirdly, no block artefacts occur on the reconstructed image at low bit-rates, because the DWT is applied to the whole image rather than a small block. Finally, DWT can be used for both lossless and lossy compression.

2.2.3 Filter Examples

There are many filters that can be used for the wavelet transform. The reader can find those filters in [14]. Part 1 of the JPEG2000 standard, recommends the Daubechies (9, 7) filter for lossy compression, and Legall I(5,3) filter for lossless compression. The coefficients are listed in Table 1 and Table 2 respectively.

Perfect reconstruction is possible when the 5-3 reversible wavelet transform is used. On the other hand, nearly perfect reconstruction can be gained when the 9-7 irreversible wavelet transform is used to encode the image. Although the reversible wavelet transform seems to be advantageous, the irreversible wavelet transform produces

significantly better results in lossy compression. The reversible wavelet transform can theoretically be used for lossy compression as well. However, its performance is not up to the irreversible filter when used in lossy coding.

i	ho	h1
0	0.6029490182363579	1.115087052456994
1	0.2668641184428723	-0.5912717631142470
2	-0.07822326652898785	-0.05754352622849957
3	-0.01686411844287495	0.09127176311424948
4	0.02674875741080976	

Table 1 Coefficients of the Daubechies (9, 7) Filter

i	ho	h1
0	6/8	1
1	2/8	-1/2
2	-1/8	

Table 2 Coefficients of the Legall (5, 3) Filter

The wavelet filtering operation can be implemented either with convolution or the lifting scheme described in the following sections.

2.2.4 Convolution Method

Let x[n] denote the one-dimensional sequence of input samples and let y[n] denote the one-dimensional sequence of interleaved sub-band outputs, where $0 \le n < N$ and the low-pass sub-band is identified with the even outputs, y[2n], while the high-pass sub-band is identified with the odd outputs, y[2n+1], the boundary extension is symmetric (see the next section). The relevant analysis operation is expressed as:

$$Y[2n] = \sum_{k=-N_h}^{p_h} h_0[k].x[2n+k] \qquad \text{and} \qquad Y[2n+1] = \sum_{k=-Ng}^{p_g} h_1[k].x[2n+1+k]$$

Where $h_0[k]$ and $h_1[k]$ denote the low- and high-pass analysis filters respectively, and N_h, P_h, N_g, and P_g are corresponding negative and positive extents of the finite support kernels.

The synthesis operation is expressed as:

$$X[n]=\sum_{k} y[2k].g_0[n-2k]+y[2k+1].g_1[n-(2k+1)]$$

where $g_0[k]$ and $g_1[k]$ denote the low- and high-pass synthesis filters.

The following example illustrates the calculation procedures in 1D DWT and inverse DWT by the convolution method. The filter is the Daubechies (9, 7). In Figure 4, line 4 contains the input samples (1, 2, 3, 4, 5, and 6); and all other extension data on both sides corresponding to samples obtained by symmetric extension at the signal's border. When an analysis operation is performed, the transformed output in line 5 is from the low-pass filter, while the high-pass coefficients are on line 6. Those outputs should be down by 2, the results are at line 7 and 8. Before doing the inverse sub-band transform, the coefficients obtained by symmetric extension are in lines 12 and 13. When the synthesis operation is performed, the inverse-transformed outputs are in lines 14 (low-pass synthesis) and 15 (high-pass synthesis), which are summed in line 16, the reconstructed results are the same as the original data.

	Forward sub-band transform													
n	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9
	extension				Original samples						extension			
X(n)	5	4	3	2	1	2	3	4	5	6	5	4	3	2
Low coefficients					1.3336	1.9366	3.0198	3.9802	5.0634	5.6664				
High coefficients					-0.865	0.250	0.183	-0.183	-0.25	0.865				
Down low coefficients					1.3336		3.0198		5.0634					
Down high coefficients						0.250		-0.183		0.865				
	Inverse sub-band transform													
n	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9
	extension				wavelet coefficients						extension			
Up low coefficients	5.063	0	3.020	0	1.334	0	3.020	0	5.063	0	5.063	0	3.020	0
Up high coefficients	0	-0.183	0	0.250	0	0.250	0	-0.183	0	0.885	0	-0.183	0	0.250
$Y_1(n)$					1.140	1.836	2.999	4.195	5.181	5.436				
$Y_2(n)$					-0.140	0.164	0.001	-0.195	-0.181	0.584				
Recovered data					1	2	3	4	5	6				

Figure 4 An example of wavelet transformation using a Daubechies (9, 7) filter

2.2.5 Symmetric Boundary Extension

When implementing any wavelet transform using convolution, the image boundaries need to be extended to avoid the data expansion inherent with linear convolution. After extension, circular convolution can be deployed. Generally, two methods can be used for boundary extension: symmetric and periodic. In the case of linear transforms, symmetric extension can yield better compression results than periodic extension due to two principal advantages:

1. It does not cause discontinuities in the extended boundary; it will not introduce disturbing artefacts at the edges of the reconstructed image;
2. It allows for non-expansive transformation of arbitrary length signals, does not significantly degrade compression performance.

2.2.6 Lifting Scheme

A simple implementation of the 2_D DWT decomposition using convolution is quite demanding of computer memory, because it requires a large amount of memory to store the entire set of image data. In practice, an alternative implementation, the lifting scheme, is used; which significantly reduces the requirements for memory and computation. For lossless compression, only reversible integer transforms are implemented using lifting. Lifting is based on three stages: split, prediction, and update:

1. Splitting the input signal x(n) into even and odd indexed sub-sequences $s^0[n]=x[2n]$, $d^0[n]=x[2n+1]$;
2. Predicting each odd sample as a linear combination of the even samples and subtracting it from the odd sample to form the prediction error d_i^1;
3. Updating the even samples by adding to them a linear combination of the already modified odd samples to form the updated sequence s_i^1.

The following formulae are for the (5, 3) filter:

$$d_i^1 = d_i^0 - {}^1/_2 \, (s_i^0 + s_{i+1}^0)$$

$$s_i^1 = s_i^0 + {}^1/_4 \, (d_{i-1}^0 + d_i^0)$$

These mathematical operations are illustrated in the Figure 5. The first column data represents the original signal. The first step computes the sequence d_i^1 that represents

the high-pass sub-band coefficients (second row data) and the second step provides the sequence s_i^1 which are the low-pass coefficients (third row data).

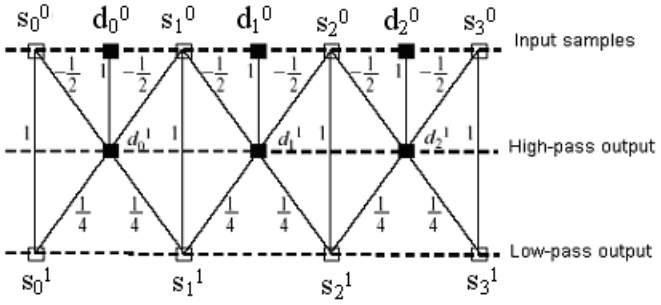
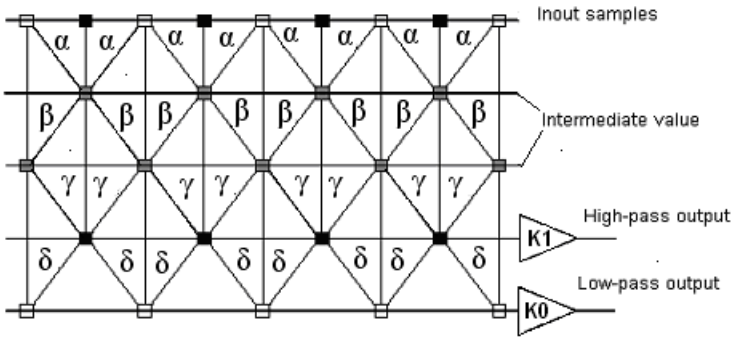


Figure 5 Lifting steps for the (5, 3) filter



α	-1.586134342
β	-0.05298011854
γ	0.8829110762
γ	0.4435068522
K_0	0.8128931
K_1	1.230174105

Figure 6 Lifting step for the (9, 7) filter

Figure 6 shows the lifting step of the (9, 7) filter, it is worth noting that the output from lifting should be normalized by K_1 and K_0 before doing the next step of the transform

2.2.7 Scalar Quantization with Filter Normalization

After wavelet filtering, the results can be directly used for coding purposes. This is usually the case for integer (5, 3) lifting. If the (9, 7) filter is used in conjunction with a full band coding method such as ZEW or SPIHT, the coefficients need to be scaled, otherwise, the importance of coefficients cannot be distinguished, because coefficients with the same magnitude in different sub-bands have a different contribution to the

image quality before scaling. In JPEG2000, the process of scaling the coefficients is combined with quantization, and it is called scalar quantization.

Normalization of the DWT filter is often expressed according to the DC gain of the low-pass analysis filter h0; and the Nyquist gain of the high-pass analysis filter h1. The DC gain and the Nyquist gain of a filter h(n), which is expressed by G_{DC} and $G_{Nyquist}$ respectively, are defined as

$$G_{DC} = \left| \sum_{n} h(n) \right|$$

$$G_{Nyquist} = \left| \sum_{n} (-1)^n h(n) \right|$$

The (9, 7) filter and the (5, 3) filter have been normalized so that the low-pass filter has a DC gain of 1 and the high-pass filter has a Nyquist gain of 2. This is referred to as (1, 2) normalization and has been adopted in Part 1 of the JPEG2000 standard.

Table 3 shows the normalization of every sub-band after a 5-level decomposition with the (9, 7) filter.

sub-band	normalization
5LL	33.924847
5HL or 5LH	17.166698
5HH	8.686717
4HL or 4LH	8.534109
4HH	4.3004827
3HL or 3LH	4.1833673
3HH	2.0792568
2HL or 2LH	1.996813
2HH	0.9672163
1HL or 1LH	1.0112865
1HH	0.52021784

Table 3 Normalization of 5-level wavelet transformation

2.3 The Features of DWT

The discrete wavelet transform provides good energy compaction. Most coefficients are small or zero, that is why the wavelet coefficients can be compressed. Furthermore, the DWT inherently is a multi-resolution image representation. It is very useful to some applications. Integer DWT decomposition can be used for lossless and lossy compression. The scaled coefficients are very suitable for embedded coding. The

embedded bit-stream that results from the bit-plane coding in each sub-band (or small blocks) provides many scalabilities and, many other features, such as SNR, resolution, random access, etc. So DWT is very suitable for image compression, this is why the JPEG2000 standard strongly adopted DWT as the main transform instead of attempting further improvement on the Discrete Cosine Transform.

3 Embedded Image Coding

This chapter investigates the most influential coding methods for determination of wavelet transform coefficients and introduces some important facets of the field relevant to this thesis. For a more detailed understanding of the field, it is suggested that the reader refers to the references provided.

3.1 Introduction

Since Wood & O'Neil [11] first used the sub-band transformation to encode images in 1986, many kinds of coding method [2, 3, 4, 5, 15, 16, 17, 18, 19, 20, 21, and 22] have emerged. However, the four coding methods that now dominate the area of image compression using wavelet are EZW, SPIHT, EBCOT and SBHP. These methods will be investigated in detail in the next several sections.

A coder is embedded if the generated bit-stream is arranged in order of importance. The first famous embedded coding method is EZW (Embedded Zerotree Wavelet) [2], designed by Shapiro in 1993. Since then, the interest in researching embedded image compression has increased and wavelet transformation began to enter many different applications. After 3 years, Said and Pearlman [3] devised a new method called SPIHT (Set Partitioning In Hierarchical Trees) upon generalizing the EZW with the performance improved by 1db on PSNR (Power Signal-to-Noise Ratio). However, these two methods can only be used for full-band and scaled wavelet coefficients. In 1998; Taubman [4] invented another method called EBCOT (Embedded Block Coding with Optimized Truncation of the embedded bit-streams) with a different idea from the EZW and SPIHT methods that was later adopted in the JPEG-2000 [1] international image coding standard. In EBCOT, context-modeling plus arithmetic coding was used, and the performance improved considerably. More importantly, EBCOT can provide many features in the resulting bit stream. The other method is SBHP [5] based on the quad-tree structure, which was a strong competitor to EBCOT during the standard development process, because of its very low-complexity features. In recent years, many variants of SBHP have been proposed in the literature [18], [19], [20], [21] and [22], ZEBC among them (Embedded ZeroBlocks of Sub-band Coding) [21] has attracted great attention with its improved performance.

3.2 EZW (Embedded Zerotree Wavelet)

Embedded zerotree coding of wavelet coefficients was invented by Shapiro [2]. The bit-stream produced by EZW is embedded, which means more bits are added to a bit-stream, the decoded image will contain more detailed information. Shapiro used a progressive encoding process to achieve this.

This technique is based on two key concepts: 1) compression of significance maps using zerotree coding of wavelet coefficients. The zerotree is based on the hypothesis that, if a wavelet coefficient at the lowest sub-band is insignificant with respect to a given threshold T, then all wavelet coefficients of the same orientation in the same spatial location at higher sub-bands are likely to be insignificant with respect to T. 2) Successive approximation quantization, which first sends the most-significant bits and then gradually refines the coefficients of the magnitude.

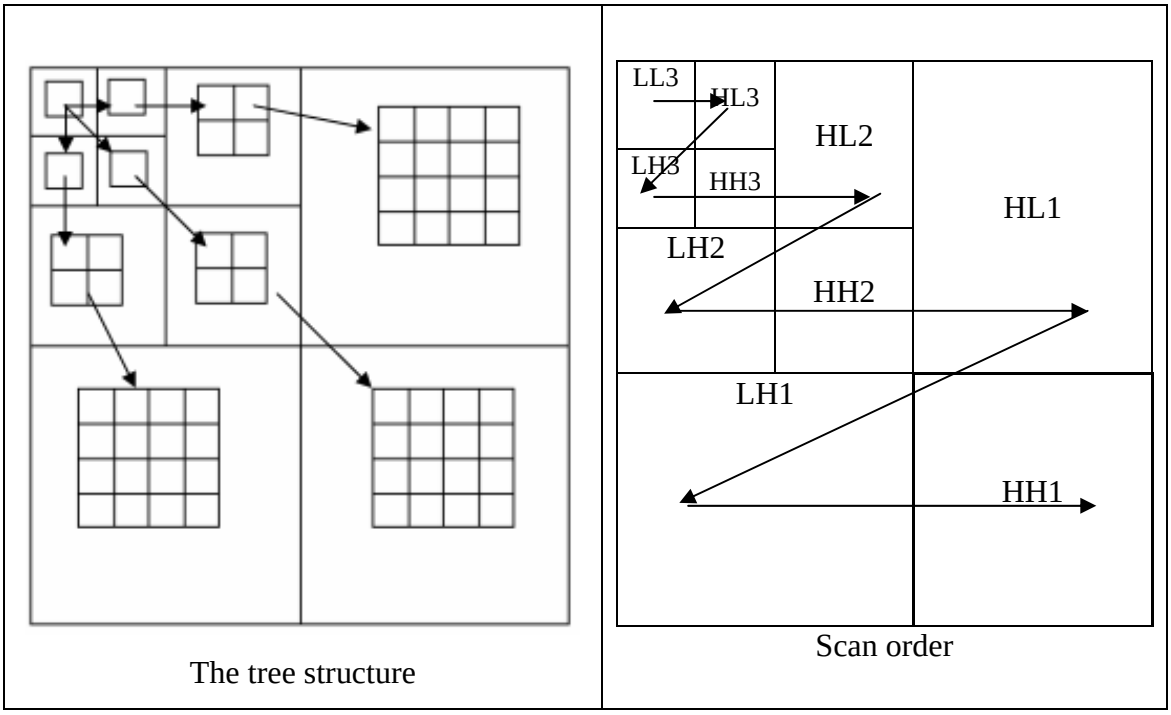


Figure 7 The tree structure and scan order

The tree structure and coefficient raster scan order in the EZW algorithm are shown in Figure 7. The key point to note is that when one coefficient in the lowest sub-band is the root of the zero-tree, all zero coefficients in the higher sub-bands do not need to be encoded. Every bit-plane is encoded in two passes except the top bit-plane; the first pass

encodes all refinement bits, while the second pass encodes the significant bits and sign bits. In the EZW, the refinement pass was called a “Subordinate Pass” while the significant pass was called a “Dominant Pass”.

The algorithm is described as the following:

1. Initialisation
   Place all wavelet coefficients on the dominant list. Set the initial threshold to
   $$2 \text{ ^ floor } (\log_2 X_{max})$$
2. Dominant Pass(raster scan across bands)
   i. Assign the following symbols to each coefficient and the resulting symbol sequence is entropy coded.

   POS – Positive significance

   NEG – Negative significance

   IZ – Isolated zero

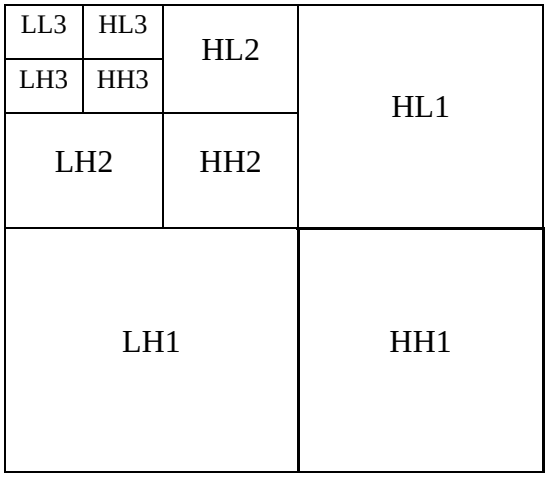
   ZTR – Zero-tree root

   – Move significant coefficients to subordinate list, and put zero in dominant list.
3. Subordinate Pass
   Output one bit (1 or 0) for subordinate list according to whether the coefficients are in upper/lower part of the quantization interval.
4. Loop
   Set $T_i = T_i/2$. Repeat step 2 to 4 until target bit rate.

Four symbols, POS, NEG, IZ, and ZTR, are produced by the dominant pass, where POS stands for positive significant coefficient, NEG for negative coefficient, IZ for isolated zero, and ZTR for zerotree. If a coefficient itself, and all the coefficients below it (its descendants), in the same wavelet tree are insignificant, this coefficient node is called a *zerotree root* and is given the symbol ZTR. If a coefficient itself is insignificant but at least one of its descendants is significant, this coefficient will be assigned the symbol IZ, it is called an isolated zero. Only two binary symbols are required by the subordinate pass, i.e., 1 and 0. All symbols are entropy coded by arithmetic coding using an adaptive model. Figure 8 and Figure 9 provide examples to explain the coding process.

63	-34	49	10	7	13	-12	7
-31	23	14	-13	3	4	6	-1
15	14	3	-12	5	-7	3	9
-9	-7	-14	8	4	-2	3	2
-5	9	-1	47	4	6	-2	2
3	0	-3	2	3	-2	0	4
2	-3	6	-4	3	6	3	6
5	11	5	6	0	3	-4	4

wavelet coefficients



the corresponding sub-band

Figure 8 Three levels Shapiro coefficients

sub-band	coefficients	symbol	Reconstructed value
LL3	63	POS	48
HL3	-34	NEG	-48
LH3	-31	IZ	0
HH3	23	ZTR	0
HL2	49	POS	48
HL2	10	ZTR	0
HL2	14	ZTR	0
HL2	-13	ZTR	0
LH2	15	ZTR	0
LH2	14	IZ	0
LH2	-9	ZTR	0
LH2	-7	ZTR	0
HL1	7	Z	0
HL1	13	Z	0
HL1	3	Z	0
HL1	4	Z	0
LH1	-1	Z	0
LH1	47	POS	48
LH1	-3	Z	0
LH1	-2	Z	0

Dominant pass

Coefficient Magnitude	Symbol	Reconstructed Magnitude
63	1	56
34	0	40
49	1	56
47	0	40

Subordinate pass

Figure 9 The coding process of EZW

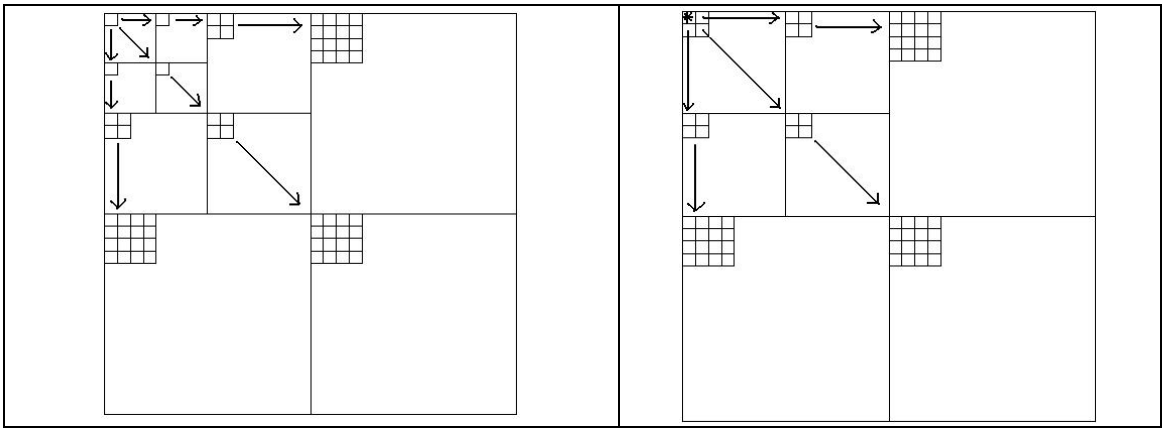
In addition to its high compression performance, which consistently outperforms DCT-based coding algorithms, the EZW has the following features as an embedded coder.

1. SNR(Signal-to-Noise Ratio) scalability achieved by successive approximation is ideal for transmitting a bit-stream over channels of various capacities;
2. Data prioritization, whereby large and small coefficients are naturally coded in order of significance;
3. More efficient entropy coding by arithmetic coding， which automatically adapts to multilevel strings of symbols generated from the coding process and requires no training or pre-stored tables;
4. The precise rate control that allows the coding process to stop anywhere when the desired bit rate budget has been achieved.

The EZW was therefore considered a state of the art image coding algorithm in that time.

3.3 SPIHT (Set partitioning in hierarchical trees)

In 1996, Said and Pearlman [3] devised a method called SPIHT (Set Partitioning In Hierarchical Trees) which was motivated by, and has several features in common with, the EZW, but the performance was improved by up to 1.3dB PSNR (Power Signal-to-Noise Ratio) and there are a number of significant differences between the two coding methods. Firstly, the order of the significance and refinement passes is reversed. Secondly, the parent-child relationship in the lowest sub-band is different. Finally, the output bits are binary rather than symbols as in the EZW, and its output has two versions: one is the binary-uncoded without entropy coding while the other is entropy coded using an arithmetic code.



EZW tree SPIHT tree

Figure 10 Parent–descendent relationships in the tree structure

In the EZW tree, the pixel in the lowest sub-band has three children. Other pixels, except for those in the highest bands, have four children. In the SPIHT tree, one fourth of the pixels in the lowest bands (noted with a "*") do not have a child. Other pixels, except for those in the highest bands, have four children. Figure 10 shows the difference between the two tree structures.

SPIHT defines two types of zero-tree, the first type consists of a single root coefficient having all descendants within a given bit-plane, and this differs from the EZW zero-tree, in that the root itself need not to be zero. The second type of zero-tree is similar but also excludes the four children of the root.

For detailed information about SPIHT, the reader should refer to reference [3]. The main idea is partial ordering by magnitude with a set partitioning sorting algorithm, where three control lists are used to transmit the ordering information. The algorithm an be described as follows.

SPIHT Algorithm:

1. Initialization:
   N: the maximum number of bit-planes,
   LSP: [ ],
   LIP: all coordinates (i, j) in the H,
   LIS: all coordinates (i, j) which have descendants in the H,
2. Sorting Pass
   2.1) for each entry (i, j) in the LIP do:
      2.1.1) output bit value $S_n(i, j)$,
      2.1.2) if $S_n(i, j)=1$, move (I, j) from LIP to the LSP and code the sign bit;
   2.2) for each entry (i, j) in the LIS do:
      2.2.1) if the entry (i, j) is of type A then
      - output bit value $S_n(D(i, j))$;
      - if $S_n(D(i, j))=1$ then
        * for each(k,l) in the O(i, j) do:
          output $S_n(k, l)$;
          if $S_n(k, l)=1$, add (k,l) to the LSP and code the sign bit,
            otherwise, add (k,l) to the LIP;

      *if $L(i, j)\neq 0$ then move (i, j) to the LIS, and goto step 2.2.2;
      otherwise, remove (i, j) from the LIS;

2.2.2) if the entry is of type B then

- output $S_n(L(i, j))$;
- if $S_n(L(i, j))=1$ then
  * add each (k, l) in the O(i, j) to the end of the LIS as an entry of type A;
  * remove (I, j) from the LIS;

3. Refinement Pass: for each (i, j) in the LSP, except those included in the last sorting pass, record the bit value;

4. Quantization-step update: decrease n by 1 and go to step 2.

Some symbols used in the above algorithm are:

- O(i, j): set of coordinates for all off-springs of node (i, j);
- D(i, j): set of coordinate for all descendants of node (i, j);
- H: set of coordinates of all spatial orientation tree roots(nodes in the lowest sub-band);
- L(i, j)=D(i, j)-O(i, j).

SPIHT is a wavelet-based image compression coder that offers a variety of good characteristics. These characteristics include:

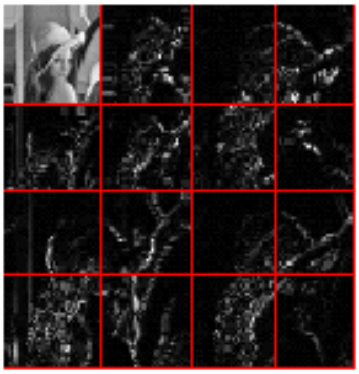
1. Good image quality with a high PSNR;
2. Fast encoding and decoding speed;
3. A fully progressive bit-stream;
4. Can be used for lossless compression;
5. Ability to stop at any target bit rate or PSNR.

The SPIHT algorithm has become a standard benchmark for image compression, because of the above advantages.

3.4 EBCOT (Embedded block coding with optimized truncation)

In 1998, Taubman [4] devised a new image coding method called EBCOT. EBCOT is significantly different from the EZW and SPIHT methods. It uses techniques based on context modelling of sub-band coefficients. This method has been adopted in the JPEG2000 standard. In EBCOT, there are two coding stages, the first is the block bit-plane coding; where the wavelet coefficients associated with each sub-band are

partitioned into fixed-size small blocks (typically 64x64). Figure 11 shows how to partition the wavelet coefficients, then, a set of embedded bit streams for all small blocks is generated using bit-plane coding. The second stage is responsible for efficiently identifying the contribution of each small block to each bit-stream layer. This stage also can be thought of as reassembling the bit-stream from every small block to the final bit-stream.



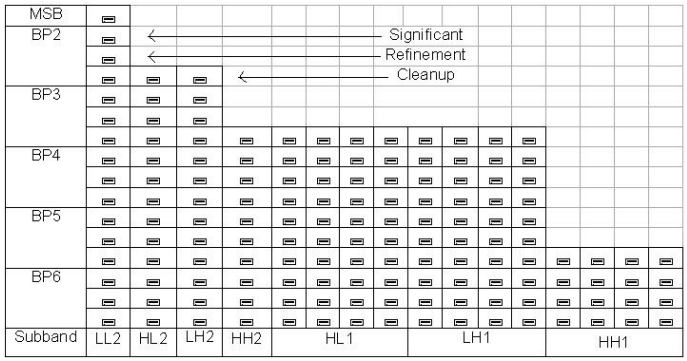


Figure 11 Block partition and compressed data from every small block

The first coding stage is one of bit-plane coding. After all of the sub-bands have been partitioned into small blocks, each of the small blocks is independently encoded using a bit-plane coder. In this stage, there are three scan passes; the pass for significant bits is divided into two passes: a significance pass and a cleanup pass, which are summarised below:

1. Significance propagation pass

   This pass is used to encode those significant bits and their sign information if and only if prior to this pass, at least one of the eight immediate neighbours has significant states. Taubman believed those significant bits were propagated by the neighbour bits. The algorithm is described by the following:

   Significance propagation pass algorithm.

   1: **for** each sample in code block **do**

   2: **if** sample previously insignificant **and** predicted to become significant during current bit plane **then**

   3: code significance of sample /* 1 binary symbol */

4: **if** sample significant **then**
5: code sign of sample /* 1 binary symbol */
6: **end**
7: **end**
8: **end**

2. Refinement pass

The second coding pass for each bit-plane is the refinement pass. If the coefficients were found to be significant in the previous bit plane, the current bit value is represented by using a single binary bit. The algorithm is described by the following:

Refinement pass algorithm.

1: **for** each sample in code block **do**
2: **if** sample found significant in previous bit plane **then**
3: code next magnitude bit in sample /* 1 binary symbol */
4: **end**
5: **end**

3. Cleanup pass

The cleanup pass simply codes the remaining significant bits. The algorithm is described by the following:

Cleanup pass algorithm.

1: **for** each vertical scan in code block **do**
2: **if** four samples in vertical scan **and** all previously insignificant and unvisited **and** none have significant 8-connected neighbour **then**
3: code number of leading insignificant samples via aggregation
4: skip over any samples indicated as insignificant by aggregation
5: **end**
6: **while** more samples to process in vertical scan **do**
7: **if** sample previously insignificant and unvisited **then**
8: code significance of sample if not already implied by run /* 1 binary symbol */
9: **if** sample significant **then**
10: code sign of sample /* 1 binary symbol */

11: **end**

12: **end**

13: **end**

14: **end**

After every small block receives three coding passes in every bit-plane, the bit-stream will be generated. These bit-streams need to be reassembled to form the final bit-stream.

The second coding stage is for packaging the bit-streams from the first coding stage into data units called packets. The resulting packets are then assembled into the final bit-stream. Figure 12 shows the coding structure. Each packet consists of two parts: a head and a body. In this coding stage, many special features are assembled into the final bit-stream including the quality layer, resolution scalability, rate scalability, random access, and region of interest coding.

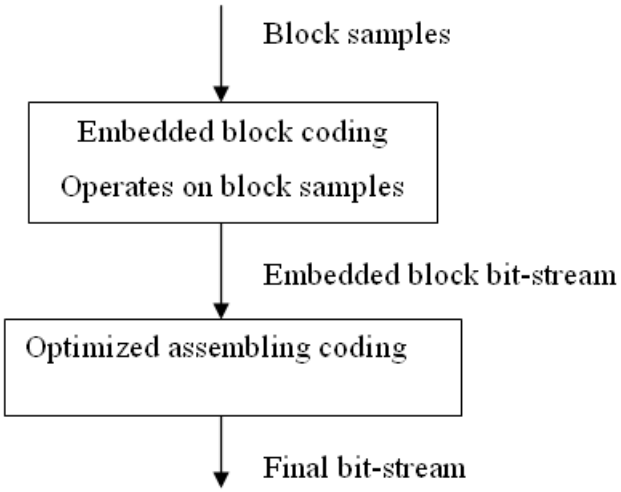


Figure 12 Two stages coding structure of EBCOT

For an appreciation of the performance of EBCOT, some results are extracted from [4]. Table 4 shows the comparison of PSNR between SPIHT and EBCOT. The PSNR results in the third column are from the SPIHT algorithm, and the second column is obtained using the EBCOT algorithm.

More detailed information on EBCOT and JPEG2000 can be found in references [23, 24, 25, 26, 27, and 28]. Software that implements JPEG2000 can be downloaded freely from the following addresses:

1. JJ2000: Java for JPEG2000, http://jj2000.epfl.ch/jj_download/index.html
2. Jasper: C for JPEG2000, http://www.ece.uvic.ca/~mdadams/jasper/
3. Kakadu: C++ for JPEG2000, http://www.ee.unsw.edu.au/~taubman/

Lena (512x512)		
Bit Rate	EBCOT (1 layer)	SPIHT
0.0625	28.30	28.38
0.125	31.22	31.10
0.25	34.28	34.11
0.5	37.43	37.21
1.0	40.61	40.41
Barbar (512x512)		
Bit Rate	EBCOT (1 layer)	SPIHT
0.0625	23.45	23.35
0.125	25.55	24.86
0.25	28.55	27.58
0.5	32.48	31.39
1.0	37.37	36.41

Table 4 Comparison of the results from SPIHT and EBCOT

3.5 SBHP

Another important wavelet coding method is SBHP, which is based on an embedded extension of the quad-tree coding technique. Compared to above three methods, its main advantage is its very low complexity. The encoding speed is about 4 times faster than EBCOT, and the decoder is about 6 to 8 times faster.

The quad-tree provides a simple coding structure for coding the significant information in each bit-plane; Figure 13 shows how to use this simple structure to generate the bit-stream. It recursively divides the non-zero block into four sub blocks until it attains a

2x2 block. It is easy to find that one 4x4 zero block on the top-left and many 2x2 zero blocks are encoded as “0”.

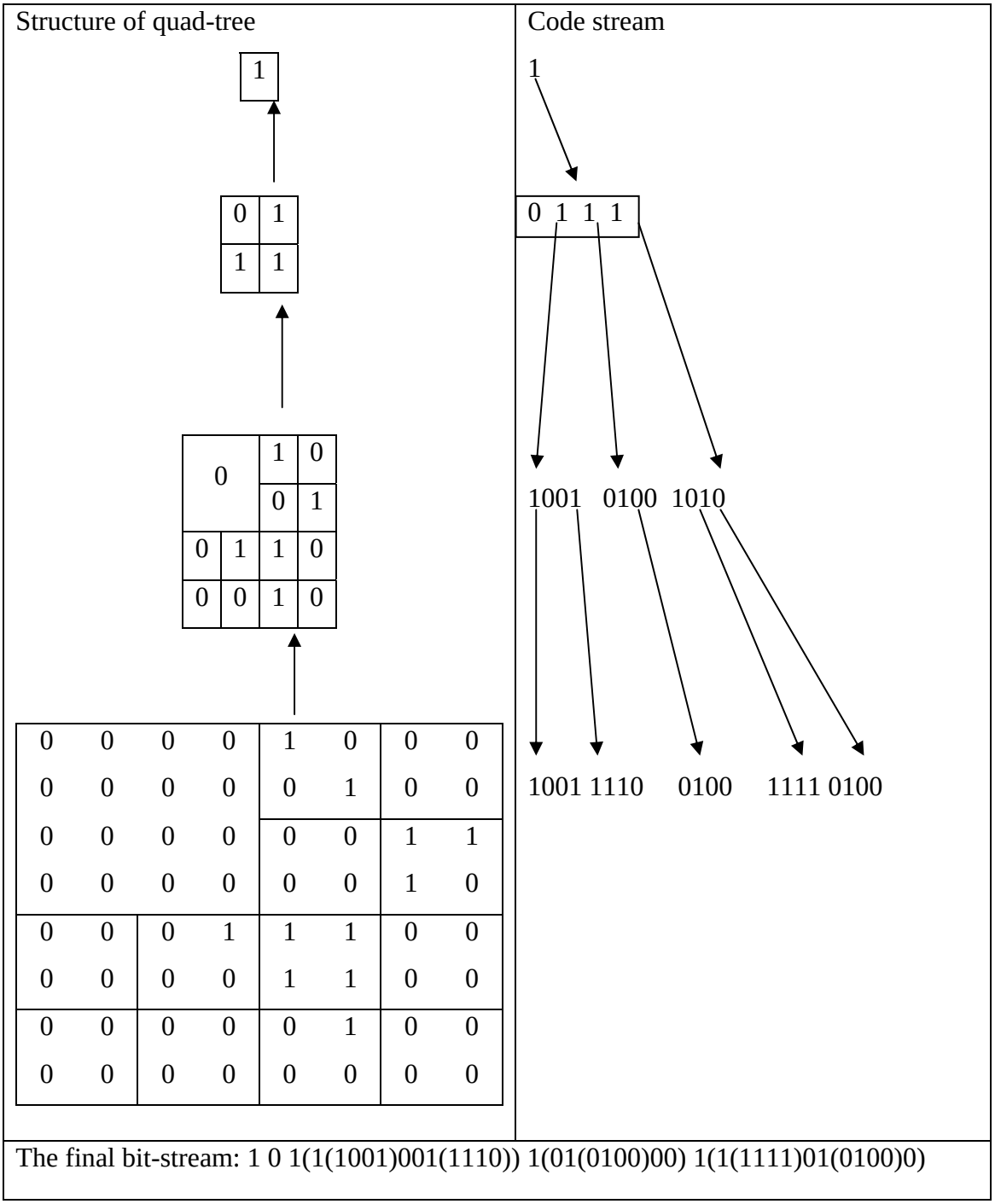


Figure 13 Example of quad-tree coding structure

In order to improve embedded performance, SBHP uses three control lists to minimize the number of tests for a given bit-plane:

1. LIS (list of insignificant sets): it includes those blocks in which all the bits are zero in previous passes;

2. LIP (list of insignificant pixels): it includes the pixels that are insignificant in previous passes;
3. LSP (list of significant pixels): all pixels found to be significant in previous passes; it is used in the refinement pass.

For each new bit-plane, those lists are updated in the order: LIP, LIS, and LSP. The quad-tree coding structure and three control lists can solve two problems in the embedded coding of bit-plane:

1. It compresses the large zero area in an efficient and fast manner;
2. Visiting of the pixels in an optimized order can generate the embedded bit-stream.

Besides SBHP providing faster coding with an optimized bit-stream, it is able to totally satisfy the requirements of JPEG2000.

Due to its simplicity, SBHP has aroused many researchers' interests. Several variants of the quad-tree coding scheme have been reported in the literature. Amplitude and Group Partitioning (AGP) [19], SWEET [18], Set Partitioning Embedded block (SPECK) [20, 22] and Embedded ZeroBlocks of Sub-band Coding (EZBC) [21] are representative examples. Among these coding schemes, EZBC has the best results compared with the other methods. It employs a quad-tree structure to partition the blocks and adaptive arithmetic coding of the context model as adopted in EBCOT. EZBC beats JPEG2000 by 0.25dB on average.

3.6 Comparison of Methods

We compare the described methods under four criteria:

1. **Coding strategy**

   EZW and SPIHT belong to the zerotree coding model. EBCOT uses the block-based adaptive context-modelling bit-plane coding, while SBHP is based on the quad-tree structure in sub-bands.

2. **PSNR performance**

   The gap between EZW and SPIHT is 1 dB averagely, SPIHT is better than EZW, but SPIHT is beaten by EBCOT by 0.1dB and EZBC beats EBCOT by 0.25dB, EBCOT is superior to SBHP by 0.5 to 1.0dB. In particular, the performance gap between SBHP and EBCOT is much larger for artificial images than for natural

images. Table 5 was extracted from [22] and it shows the PSNR performance for each of them.

Coding method	PSNR(dB)						Lossless Rate (bpp)
	0.0625	0.125	0.25	0.5	1.0	2	
Bike(2048x2560)							
JP2k	23.74	26.31	29.56	33.43	37.99	43.95	4.520
SBHP	23.02	25.36	28.53	32.39	37.07	43.04	4.724
EZBC	23.75	26.11	29.58	33.53	38.25	44.33	4.359
SPIHT	23.44	25.89	29.12	33.01	37.70	43.80	4.480
SPECK	23.31	25.59	28.84	32.69	37.33	43.1	4.492
Cafe(2048x2560)							
JP2k	19.03	20.77	23.10	26.76	31.96	39.01	5.384
SBHP	18.76	20.49	22.64	26.01	31.08	38.26	5.466
EZBC	19.11	20.87	23.32	27.00	32.43	39.62	5.125
SPIHT	18.95	20.67	23.03	26.49	31.74	38.91	5.277
SPECK	18.93	20.61	22.87	26.31	31.47	38.7	5.286
Woman(2048x2560)							
JP2k	25.59	27.33	29.95	33.57	38.28	43.97	4.541
SBHP	25.26	27.09	29.59	33.11	37.98	43.69	4.636
EZBC	25.71	27.54	30.31	34.00	38.82	44.48	4.291
SPIHT	25.43	27.33	29.95	33.59	38.28	43.99	4.419
SPECK	25.50	27.34	29.88	33.46	38.07	43.73	4.396
Aerial2(2048x2048)							
JP2k	24.60	26.47	28.54	30.60	33.23	38.05	5.471
SBHP	24.42	26.34	28.15	30.40	33.03	37.70	5.502
EZBC	24.76	26.65	28.70	30.79	33.49	38.51	5.203
SPIHT	24.63	26.52	28.49	30.60	33.32	38.22	5.331
SPECK	24.60	26.49	28.45	30.59	33.25	38.26	5.259

Table 5 Comparison of lossy coding performance for common test images

3. The generated bit-stream

EZW and SPIHT produce a single embedded bit-stream; this kind of bit-stream has one advantage and three disadvantages:

The advantage:

1. The encoding and decoding process can be stopped at the desired bit-rate or PSNR.

The three disadvantages:

1. This method can only be applied to full-band scaled coefficients;
2. The final generated bit-stream cannot satisfy the requirements of some applications (for example, medical applications, only want one part of the information), because it has no resolution scalability and random access ability.
3. The error resistance is poor, because one single error bit in the bit stream will make the bit stream undecodable.

EBCOT and SBHP generate a packet-stream; this kind of packet-stream provides many advantages as well as some disadvantages:

The advantages:

1. Resolution scalability;
2. SNR scalability;
3. Random access;
4. Region of interest coding;
5. Good error resistance.

The disadvantages:

1. The maximum block size of 64x64 restricts the exploitation of the redundancy existing in the every sub-band and the whole coefficients matrix;
2. The encoding and decoding process cannot be stopped at the desired bit-rate or PSNR;
3. Large overhead for packet head and tail;
4. The quality layer may cause part of the image to be degraded, and it loses the original meaning of the embedded bit-stream in the extreme of one layer, because the embedded bit-stream should be ordered according to the importance of the coefficients.

4. The complexities and speed

SBHP is the simplest, while the EBCOT is the most complex with EZW, SPIHT sitting in the middle. The order of speed is SBHP, EBCOT, SPIHT, and EZW.

3.7 Summary

This has been a review of coding methods for image compression using the wavelet transform. The aim was to find the main ideas in those methods and where and how we can improve upon them. EZW and SPIHT discover a hidden secret in wavelet coefficients matrix: Zerotree. Block-based coding with the context modelling in EBCOT can provide many features and excellent error resistance, while SBHP provides a simple coding structure and faster encoding/decoding speed. However, they all have some unpleasant disadvantages. EZW and SPIHT generate a single embedded bit-stream; this kind of bit-stream cannot satisfy the requirements of some applications, it has only SNR scalability. The error resistance is also poor and SPIHT does not use entropy coding to the refinement bits. EBCOT is too complex; the coding efficiency does not pay off the increased complexity in some applications. The encoding and decoding process cannot be stopped at the desired bit-rate; and large overheads for the packet head and tail decrease the coding efficiency, because the maximum block size is restricted to 64x64. SBHP has poor performance, although it has very low complexity. Obviously, finding a new coding method to overcome these advantages is still desirable.

4 Factors affecting the size of the final bit-stream

This chapter identifies the factors, which influence the coding performances, based on the generic codec structure. After analysis of these factors, a new codec structure is proposed.

4.1 General Codec Structure

Figure 14 shows a general codec structure. The encoder consists of six parts: pre-processing, discrete wavelet transforms (DWT), quantization, bit-plane coding, bit-stream assembling, and rate-control. The decoder has four parts; bit-stream decoder, dequantization, inverse discrete wavelet transforms and post-processing.

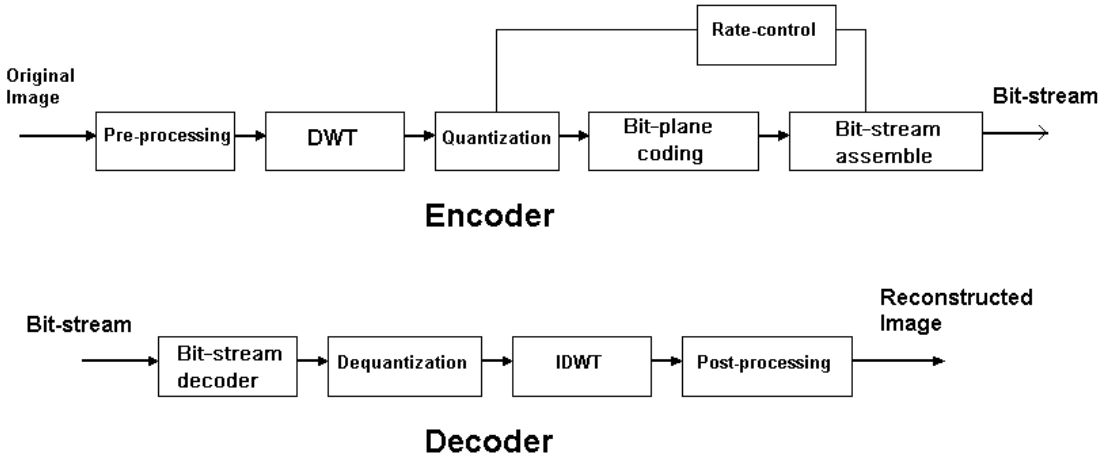


Figure 14 Codec structure

In chapter 2, we described how to implement DWT and the quantization process in some detail. Here we introduce the functions of the other processes and point out how these processes affect the overall coding results.

4.1.1 Pre-processing/post-processing

Pre-processing generally performs three tasks:

1. The first is to partition the large image into rectangular and non-overlapping tiles of equal size, if the image size is too large;

2. The second is for colour transform, there are two kinds of colour transform: ICT (the irreversible colour transform) and RCT (the reversible colour transform);
3. The final task is that the unsigned pixel values are level shifted by subtracting a fixed value of 2^{b-1} from each of the pixel values to make its value symmetric around zero, Figure 15 shows this operation. Signed pixel values are not required for level-shifting. Level-shifting has no effect on the coding efficiency. The post-processing in the decoder side simply undoes the effects of pre-processing in the encoder side.

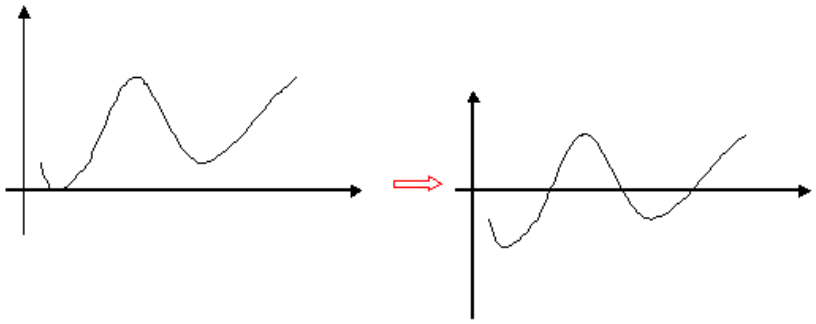


Figure 15 DC-level shifting

In this thesis, we restrict our attention to greyscale images. Therefore, in our implementation, only one task, level shifting, is needed. The first task is also not required in our implementation due to the normal image sizes that we have chosen. Post-processing simply undoes the effects of pre-processing in the encoder.

4.1.2 Discrete wavelet transform

It has already been explained in Chapter 2, that the outputs of the wavelet transform are called wavelet coefficients, those coefficients can be directly passed to bit-plane coding in the reversible model. The scaling of the coefficients has some effects on the coding results. We shall explain it again in more detail in the following sections.

4.1.3 Quantization/dequantization

Quantization allows great compression. Quantization of the transform coefficients is one of the two primary sources of information loss in the coding processes. Transform coefficients are quantized using scalar quantization with a deadzone. Scalar quantization means that the scalar factor should be considered together with the quantize step size. Different quantizers are employed for the coefficients in different sub-bands. This is

different from the traditional quantization. The formulation is presented here:

$$V(x,y)=\text{sign}(U(x,y)) \lfloor |U(x,y)|/D \rfloor$$

Where U is a wavelet coefficient, D is a quantizer step size, and V(x,y) denotes the output for the sub-band.

In the integer model, the quantizer step sizes are always fixed at one; it is equivalent to bypassing a quantization and forcing the quantizer indices and transform coefficients to be the same. In this model, although the scalar factor was not considered, lossy coding is still possible.

In the decoder, the dequantization process tries to undo the effects of quantization. The reconstructed floating point wavelet coefficients U(x,y) are obtained from the quantizer indices V(x,y) for a quantizer step-size of Δ_b. The formulation is:

$$U(x,y) = \begin{cases} (V(x,y)+r)\,\Delta_b & \text{If } V(x,y)>0 \\ (V(x,y)-r)\,\Delta_b & \text{If } V(x,y)<0 \\ 0 & \text{If } V(x,y)=0 \end{cases}$$

The value r is in the range $0 \le r < 1$ and may be chosen to produce the best visual or objective quality for reconstruction (a typical value is r=1/2).

It is worth pointing out that the scalar quantization process is very important in the general codec structure, before the bit-plane coding, the wavelet coefficients should be scaled by using the L_2 norms of each sub-band in order to align the bit-planes of the quantizer indices according to their true contribution to the MSE. Thus, the same magnitude coefficients in each sub-band have the same contribution to MSE. In the EZW, SPIHT and JPEG2000, the ordering of coefficients only applies to the same bit-plane; the scalar quantization can make the optimization easily processed.

Therefore, scalar quantization has two advantages:

1. The sum of the squares of the image samples is approximately equal to the sum of the squares of the transform coefficients, making it easy to calculate the rate distortion;
2. It facilitates the selection of the most important wavelet coefficients in the encoding process.

However, this kind of quantization causes one problem:

1. In the lossless mode, the quantizer step-size is fixed at one, the scalar factor is not taken into consideration in the image coding processes in the existing algorithms, especially for the full-band image coder such as SPIHT, SPECK and EZBC, and this will add difficulty to the design of the coding algorithm.

4.1.4 Bit-plane coding

Bit-plane coding can be applied to wavelet coefficients of full-band, or sub-band and the blocks within a sub-band. EZW and SPIHT are applied to the full-band; EBCOT and SBHP are applied to the small blocks within a sub-band. Bit-plane coding is a key process; it sets out to represent the coefficients in the most efficient way. Generally, this process has two or three passes for different parts of the coefficients, for example, one pass or two passes are for the significant bits, one pass is for refinement bits.

The purpose of bit-plane coding is to generate an embedded bit-stream. After a sub-band or a block is sliced into bit-planes, each bit-plane is encoded independently. The coding order is from the most significant bit-plane to the least significant bit-plane, producing an embedded bit-stream.

As noted above, there are two or three coding passes per bit-plane. In the EZW and SPIHT, the coding passes consist of a significance and a refinement pass while EBCOT and SBHP employ three passes: significance, refinement, and cleanup. Generally, the bit-plane coding pass does not directly generate the binary bit-stream, but rather, a sequence of symbols. Some or all of these symbols need to be entropy coded. In EZW and SPIHT, the context dependent arithmetic coding is used to losslessly compress the sequence of symbols resulting from the coding procedures. EBCOT employs a context-based adaptive binary arithmetic coder, more specifically, the MQ coder from the JBIG2 standard. SBHP does not use any arithmetic coding; it employs two fixed 15-symbol Huffman coders.

The results from bit-plane coding can be directly transmitted to the decoder side in the EZW and SPIHT, because those bit-plane coders are applied to the whole wavelet coefficients matrix. However, the results from EBCOT or SBHP need an extra stage to assemble the bit-stream resulted from the previous stage.

The bit-plane coding process significantly affects the overall coding efficiency. The significant pass of the two coding passes is the main factor. EZW and SPIHT use the zerotree to achieve their best results while EBCOT uses adaptive context modelling.

4.1.5 Bit-stream assembling

Bit-stream assembling generates the final embedded bit-stream. The input to the bit-stream assembler is the set of bit-stream packets generated during the bit-plane coding. The principle is to choose one bit-stream packet with the best contribution to image quality. The process of bit-stream assembling can be performed as follows:

1. Initialization: for all bit-stream packets, set up all contributions c_i =0 (i=1,2,3,…n);
2. Calculate the contribution for each bit-stream packet:

   $$c_i = (MSE - MSE_i) / L_i$$

   where MSE (mean squared error) represents the distortion between the original image and the reconstructed image using the available bit stream. MSE_i is the distortion after the i-th bit-stream pack is added to the fianl bit stream, $MSE - MSE_i$ is the distortion decrease, L_i is the length of the i-th bit-stream packet. Thus we can find out which packet leads to the maximum distortion decrease and this is the packet that we need to send to the final bit stream at this point.
3. Repeat Step 2 until the desired bit rate is reached.

In JPEG2000, the process of bit-stream assembly is much more complex. The interested reader can refer to the relevant material for the standard.

The packet assembly process can increase the overhead to the final bit-stream, but it is necessary.

4.1.6 Rate control

Rate control refers to the process of generating an optimal image for a target bit-rate. In the encoder, rate control can be achieved through two distinct approaches: 1) the choice of quantizer step sizes, and 2) the selection of truncation points in the bit-stream. The first approach may not be suitable in practice. Every time the quantizer step size is changed, the encoding process must be performed again. When the second approach is used, the encoder can find the termination point in the bit-stream, because the encoder can either calculate the distortion reduction of the received bit-stream or count the bit

budget. This approach is very flexible in that different distortion calculation methods can be easily accommodated, for example, MSE, or visually weighted MSE.

4.1.7 Bit-stream decoder

The bit-stream decoder simply undoes the encoded bit-stream. Generally, the decoder speed is faster than the encoder speed, because not so many comparison operations for testing the significant or insignificant pixel or blocks are needed.

4.2 Four Affecting Factors

From the above analysis for every coding process we studied, four factors are found affecting the amount of the compressed bit-stream:

1. The number and magnitude of wavelet coefficients generated from DWT;
2. The efficiency of significant encoding pass in the bit-plane coding;
3. The efficiency of refinement encoding pass in the bit-plane coding;
4. The efficiency of the entropy coding method for exploiting the dependence among the symbols generated from the significant and refinement pass. Figure 16 shows the relationship among these factors.

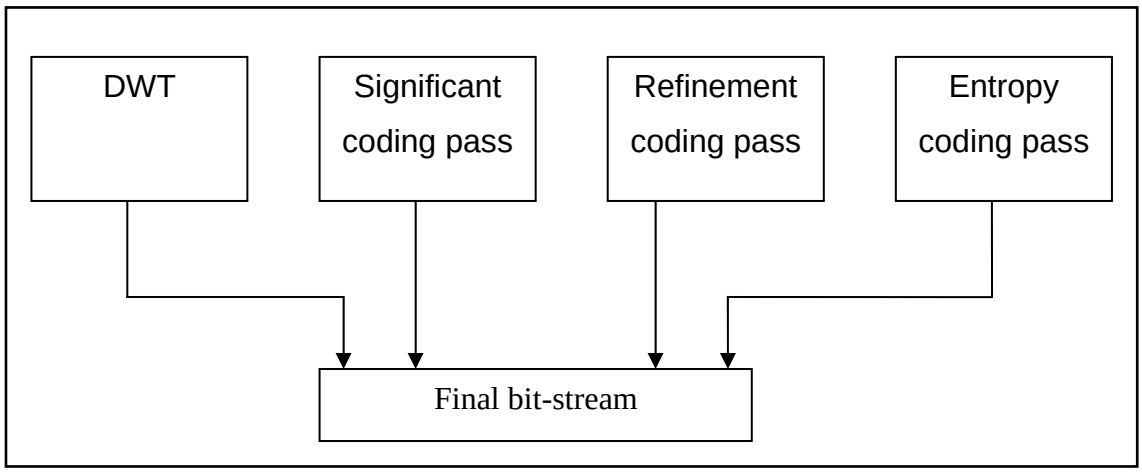


Figure 16 The main factors affecting the amount of final bit-stream

4.2.1 Filters

The first factor is the filters used in the wavelet transformation. Different filters result in different wavelet coefficients, it is not difficult to conclude that the more and bigger the coefficients, the longer the bit-stream. However, wavelet filter design is a branch of mathematics, and is not the focus of this research.

4.2.2 Significant encoding method

The second factor is the method for encoding significant bits and sign bits. The significant bits appear randomly in every bit-plane; how to represent the location of every significant coefficient is a challenging task. The primary purpose of this step is to represent the location of every significant bit in a most efficient way. EZW and SPIHT use the zero-tree to encode the significant bits, while EBCOT uses the context-modeling to encoder those significant bits. SBHP encodes the significant bits by first grouping into small blocks, then using Huffman coding to represent the context of these blocks. This factor is the most important factor.

4.2.3 Refinement encoding method

The third factor is the method for representing the refinement bits. Up till now, most algorithms for encoding the refinement bits simply record the magnitude of the bit value. EBCOT uses context-modeling to encode those refinement bits, but the performance still leaves room to improve. SPIHT and SBHP directly put the refinement bits into the final bit-stream. How to efficiently code those bits is worth further study, in depth, because those bits constitute almost a half of the bit stream in the lossless coding situation.

4.2.4 Entropy coding

The last factor is the method for entropy coding for exploiting the dependence among the symbols generated from significant encoding or refinement encoding pass. Currently, arithmetic coding is used for this task in most of the algorithms. But how to efficiently use the arithmetic coding deserves further consideration, simply using adaptive arithmetic coding is not very effective. In the SPIHT algorithm, the arithmetic coding was used to exploit the dependence between adjacent pixels, while EBCOT uses it for exploiting the dependence among the context modeling in a bit-plane.

Based on the above analysis, we focus our research on finding good strategies to overcome the negative effects of these factors, taking into account the performance, complexity and coding speed.

4.3 New Codec Structure

4.3.1 Strategies to overcome the affecting factors

Generally, after the wavelet transformation, the coefficients in the lowest sub-band are bigger than that in the higher sub-bands, where the wavelet coefficients are mostly small or zero, which is why the wavelet transform is suitable for image compression. Wavelet coefficients are represented by binary data, some take more bits, and some take less. In addition, every non-zero coefficient has its sign code: positive or negative. The bits used for coefficients (including magnitude bit and sign bit) must be transmitted to the decoder. We also need some bits for the representation of location, here; we call those bits for location as overheads or support bits.

Table **6** is an example, where the largest coefficient is 255, the smallest coefficient is 1. The overall number of coefficients is 64, and only 12 coefficients are non-zero.

0	255	0	0	0	0	0	0
156	0	101	0	0	0	0	0
0	0	-42	0	0	0	0	0
0	36	122	0	0	0	0	0
30	0	0	0	63	0	0	0
12	0	0	0	0	0	0	0
0	0	0	0	0	1	-	0
0	0	0	0	23	0	0	0

Table 6 Coefficients in the wavelet array

coefficients		255	156	122	101	63	-42	36	30	23	12	-12	1
Sign bit		0	0	0	0	0	1	0	0	0	0	1	0
MSB	8	1	1										
	7	1	0	1	1								
	6	1	0	1	1	1	1	1					
	5	1	1	1	0	1	0	0	1	1			
	4	1	1	1	0	1	1	0	1	0	1	1	
	3	1	1	0	0	1	0	1	1	1	1	1	
	2	1	0	1	1	1	1	0	1	1	0	0	
LSB	1	1	0	0	1	1	0	0	0	1	0	0	1

Figure 17 The binary representation of coefficients

In the Figure 17, every row represents a bit-plane, the first “1” in each column is called the significant bit, and the bits below the significant bit are called the refinement bits. The top bit is called the sign bit. Generally, the sign bit is encoded with the significant bit at the same time.

Now we calculate how many bits exist in the coefficient matrix. After wavelet transform, the coefficients are represented as decimal data; normally; however, we only code the integer part. The bits of the integer part for every coefficient can be calculated by the following formula:

$$BOC=ceil(log(x(i,j)+1)/log(2))+1 \qquad (x(i,j)\neq 0)$$

BOC is the bits of a coefficient, x(i,j) represents the coefficients data, the last 1 represents the sign bit. Generally, the coding process has two passes, one is for the significance, the other is for refinement, and every coefficient should take two bits for significant coding (1 bit for significant bit, 1 bit for sign bit). Table 7 shows some results generated from well-known Lena image:

Image name: Lena (512x512)	The amount of bits 9/7 filter(1,2), level=5	The amount of bits 5/3 filter(1,2), level=5
Number of non-zero coefficients	138808	228475
overall bit amount	412415	767379
Number of significant bits (including sign bit)	277616	456950
Number of refinement bits	134799	310429

Table 7 The bit amount distribution of Lena image (512x512)

From Table 7, we can see two points: 1) the different filters generate a different number of coefficients; 2) the significant bits take more bits than the refinement bits. These points have already been covered in depth in the literature [29]. Here the bits devoted to the locations of the coefficients are the main reason that the significant bits take a greater proportion of the final bit stream than the refinement bits.

So, the objectives of the coding method really are:

1. Using as few bits for location as possible;
2. The dependence among the generated symbols or binary bits should be exploited effectively.

4.3.2 New codec scheme

Figure 18 is a diagram of a new codec scheme. In this scheme, there are three different processes compared with the traditional codec structure: the block dividing coding process, the optimized assembling coding process, and the layer (or pass) decoding process.

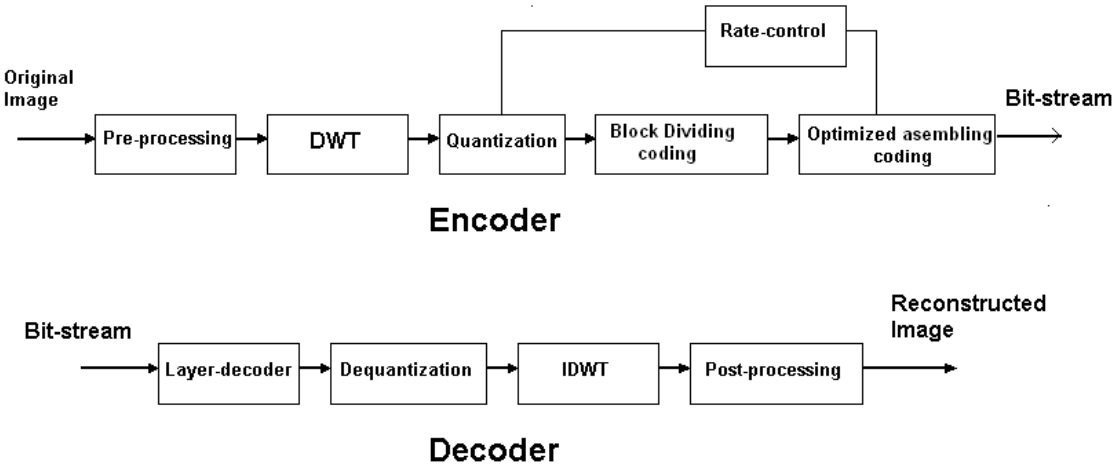


Figure 18 New Codec structure

In the new codec structure, the block dividing coding process is applied to every bit-plane for generating the bit-stream packets. Every bit-plane - except the significant bit-plane - is encoded by two coding passes: a significant pass and a refinement pass. The optimized assembly is responsible for generating the fully embedded bit-stream. The layer decoder is to undo the bit-stream packets for recovering the coefficients in the corresponding sub-bands or blocks. More details about the proposed algorithm are described in the following chapters.

5 Block dividing Coding Algorithm

This chapter describes the block dividing coding method in detail. We provide a simple example to illustrate the spirit of our method. The objective of the coding method is to use less overhead for location of every significant coefficient to achieve the maximum compression of the bit-stream.

5.1 Introduction

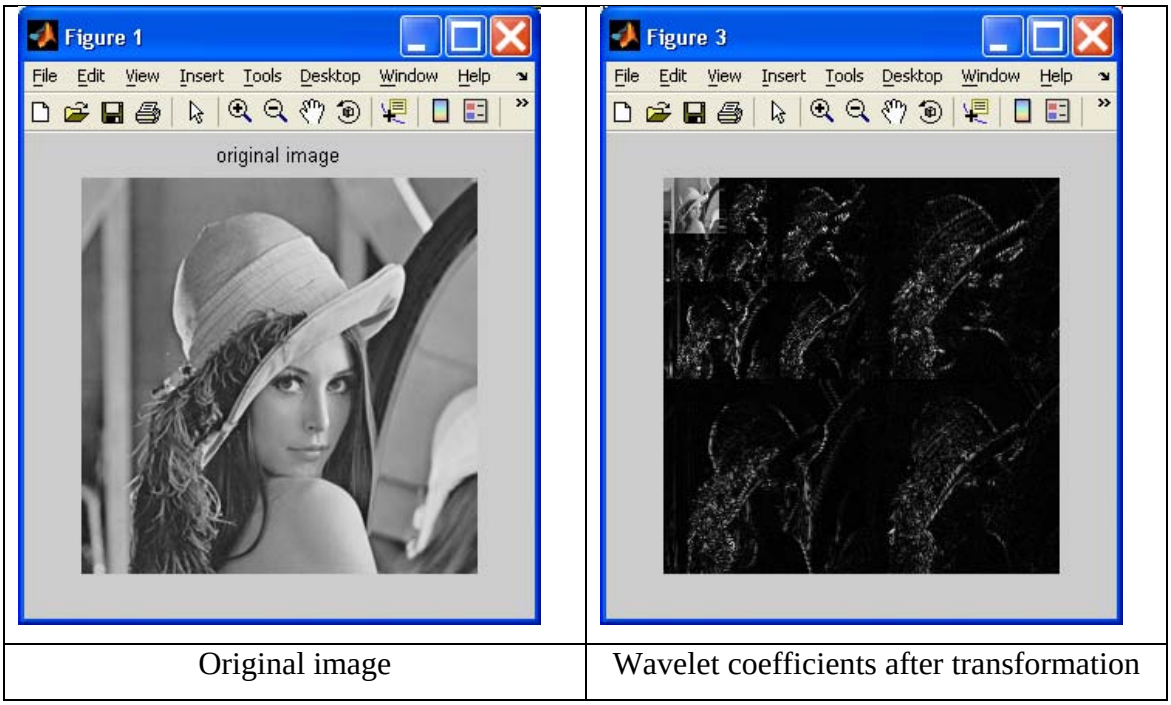
With low-complexity and desirable features such as resolution scalability, SNR scalability, and exact rate control, embedded image coding methods using wavelet transforms have been the main trend in the image coding community in recent years. Due to the high energy compaction nature of the wavelet transform, the bigger coefficients are mainly compacted in the lower sub-band while a large region of coefficients in the higher sub-band are relatively small. Coding those regions with small coefficients in blocks based on a bit-plane is much more efficient than coding the coefficients one-by-one. The order of visiting the coefficients should be optimized for improving embedded performance. The proposed BDC is a good candidate for meeting these requirements. Three block dividing methods can adapt to all kinds of wavelet coefficient blocks, while progressive coding with three control lists can generate the fully embedded bit-stream.

5.2 A New Strategy to Code the Coefficients

5.2.1 Coding the coefficients

In the proposed BDC algorithm, each sub-band of coefficients is encoded independently using bit-plane coding starting from the most significant bit-plane to the least significant bit-plane. Every bit-plane except the most significant bit-plane is encoded by two passes: significant pass and refinement pass. After encoding all sub-bands, the bit stream is reassembled in an optimized way, in which the contributions of the bit-stream packets from each sub-band to the mean square error (MSE) of the image are compared and the bit-stream packet of the sub-band with the biggest contribution to MSE is chosen to be sent out.

Figure 19 shows a 3-level, 2-D dyadic decomposition. From this diagram, we can see the most energy is compacted into the lowest sub-band, and most coefficients in the highest sub-band are small or zero.



Original image	Wavelet coefficients after transformation

Figure 19 An example of the distribution of coefficients

Figure 20 shows the sub-bands of the wavelet coefficients which have been numbered for the future use.

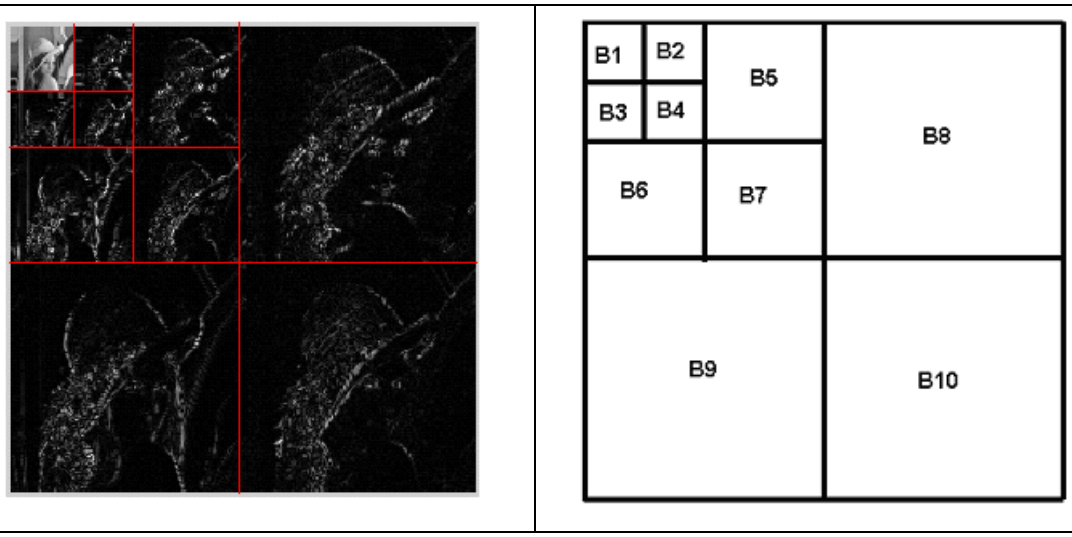


Figure 20 The coding block in the 3-level wavelet decomposition

Figure 21 shows the concept and structure of the bit-plane.

0	5	3	0	-1	-6	-8	-2
0	6	37	12	13	6	-16	7
4	-3	65	34	23	28	-23	5
2	0	45	48	64	7	12	1
0	0	14	22	34	-4	11	0
0	0	-23	12	7	0	-5	0
24	33	-6	-9	7	0	12	0
0	0	0	0	0	0	0	16

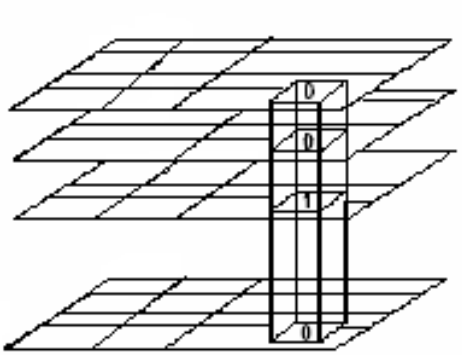


Figure 21 An example of a multi bit-plane

The objective of BDC is to encode large areas of the coefficients in an efficient and fast manner, taking advantage of the fact that those areas have the high probability having zero bits in a particular bit plane. For example, if the size of a zero block is 16x16, one zero output can represent the 256 (16x16) zero bits. Our method is motivated by the previous coding methods: SPIHT, SBHP and EBCOT, but it is different from those algorithms in various respects. We adopt several dividing methods to deal with the small non-zero blocks according to the number and distribution of the significant bits. It features low complexity and high compression performance, and provides many opportunities to rearrange the bit stream efficiently duo to the fact that every sub-band is encoded independently. The principal of the BDC algorithm is summarised as below:

1. Choose one of three dividing methods to divide one block into sub-blocks with the same size, according to the number and distribution of the significant bits in the block;
2. Code every sub-block: every zero sub-block is encoded as 0, every non-zero sub-block is encoded as 1;
3. Recursively apply above block dividing methods to every non-zero block until a 2x2 size block is reached, then use one of 16 symbols to represent the context of the 2x2 block.

During this coding process, we use three lists to track the block dividing information.

5.2.2 Three block dividing methods

In this section, we explain the idea of the three block dividing methods and three control lists. Consider an image X that has been transformed by several level decompositions. The number of bit-planes for every sub-band is determined by the maximum coefficients c(i, j) of that sub-band. We say that this coefficient is significant with respect to n if

$$2^n \le |\, c(i, j) \,| < 2^{n+1}$$

Otherwise, we say the coefficient is insignificant in bit plane n.

Figure 22 is an example of one bit-plane. In this example, there are only three significant bits. Other bits are all zero.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1+	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1-	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1+	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 22 An example of a bit-plane

We use the following step to determine how to divide this bit-plane into sub-blocks. Obviously, it is inefficient to divide this bit-plane into irregular blocks because the overhead for describing the irregular blocks is expensive. The goal of block dividing is to separate the significant bits from insignificant bits efficiently, while maintaining zero blocks as large as possible.

1. If there is only one significant bit in this block, use coordinate representation to code this significant bit, and use quad-tree splitting method to divide this block. Using 10 represents this dividing method;

2. Divide the non-zero block into four sub-blocks if at least one of sub-blocks is zero, using 0 to represent it;
3. Divide the non-zero block into 16 sub-blocks if none of the sub-block is zero, using 11 to represent it.

The statistics of the blocks confirms that the three dividing method are reasonable. During our experiments, we find that 25% of the blocks can be divided using the first method and about half of the blocks can be divided using the second dividing method, while the other 25% of the blocks is divided using the third dividing method.

We now explain the block dividing method in detail with examples and compare the results with SBHP. Figure 23 shows an example of the first dividing method and coding process, where there is only one significant bit in the block in the bit plane. The block can be divided using quad-tree shown in the diagram by the red-bold line. However, the coding process with BDC is different from the quad-tree method. We use the coordinate representation to encode this significant bit. This block is an 8x8 block; the numbers of bits used for representing the x, y coordinate position are three bits. In this example, they are (1, 2), they are encoded as (001,010), and then we add one bit for the sign bit. The total bits consist of:

> 1 (it means this block is one significant block), 10 (it represents this kind of dividing method), 001010 (it represent the x, y coordinate position of the significant bit), 1(it represent the sign bit, we use 0, 1 to represent the positive or negative sign, here we assume its sign is 1). It equal to: 1100010101, the number of bits is 10. But if we use quad-tree method, the result is 1,1000,0100,0010,1(sign bit), the total number of bits is 14. It can save 4 bits. If the block size is more than 8, the more bits are saved. The reader can easily confirm this for themselves.

This dividing method improves the coding efficiency significantly to the several upper bit-planes as the coding result of from the quad-tree method is replaced by the coordinate representation.

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

Bits from our method	1100010101
Bits from quad-tree	11000010000101

Figure 23 An example of first kind of dividing method

The second dividing method is same as the quad-tree. The non-zero block is simply divided into four sub-blocks.

The third kind of dividing method is more efficient than the quad-tree splitting method. The non-zero block is directly divided into 16 sub-blocks. This kind of dividing method can save two bits when compared with the quad-tree splitting method. Figure 24 shows this kind of dividing process.

0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0
0	0	0	0	0	0	0	0

Figure 24 An example of third kind of dividing method

Similar to SPIHT and other hierarchical bit-plane coding methods, three lists are used for tracking the dividing process.

- LIS– list of insignificant sets (or blocks), which includes all tested insignificant blocks;
- LIP– list of insignificant pixels, which includes all tested insignificant coefficients; here pixel and coefficient have used without distinction;

- LSP– list of significant pixels, it is used for the refinement pass, and includes all tested significant pixels.

After the coding process of the one bit-plane, LIS contains insignificant blocks with different sizes. These blocks are reordered by size from the smallest to the largest. This reordering operation can improve the embedded performance, since the smaller blocks are adjacent to the significant pixels or significant blocks during the block dividing process.

As the coding process progresses, the lists are updated for every bit-plane.

5.3 Block dividing Coding Algorithm

The BDC encoder first visits all pixels to gather information about bits in a bit-plane, then recursively chooses one of three block dividing methods to partition the non-zero blocks until 2x2 blocks are reached. The zero blocks with size equal to or greater than 4 is encoded as one bit: “0”, while the non-zero blocks is indicated by bit “1”. Each set of context information on the 2x2 blocks is encoded together as one symbol, the number of contexts for 2x2 blocks is 16. Then the generated symbols are entropy coded.

We firstly give the definition of the symbols used in the algorithm:

1. $c_n(i, j)$ denotes the quantized sub-band coefficient with respect to the n-th bit-plane and its position is at (i,j);
2. (i, j, bs) denotes the insignificant block and its left upper corner position is at (i, j) and the block size is bs;
3. (i, j) denotes the position of the tested significant and insignificant bit;
4. Node, position, point, pixel and coefficient are used inter-changeably without distinction.

The complete algorithm for block dividing coding algorithm is presented as follows:

1. initialization

   Set the initial block in the LIS to the block of a sub-band,

   LIS=[(0,0,bs)];

   LIP=[];

   LSP=[];
2. code LIP

if the number of nodes ≤ T1
for each (i, j) in LIP,
record the value of $c_n(i, j)$;
if $c_n(i, j)$ is significant,
code the sign bit and move (i, j) from LIP to LSP;
else
for each (i, j) in LIP,
if $c_n(i, j)$ is positive significant bit,
output symbol P and move (i, j) from LIP to LSP;
if $c_n(i, j)$ is negative significant bit,
output symbol N and move (i, j) from LIP to LSP;
else
output symbol Z and the node (i, j) remain in LIP;
the generated symbols are entropy coded.

3. code LIS
for each block (i, j, bs) in LIS,
do process_LIS;.

4. code LSP (refinement pass)
for each (i, j) in LSP except those positions added in this bit-plane,
record the value of $c_n(i, j)$;
5. decrement n and go to step 2.

In the algorithm, T1 is a threshold, which determine if or not the arithmetic coding is applied to the symbol stream resulted from coding coefficients in LIP. If the number of coefficients in LIP is too small, applying arithmetic coding does not produce any gain due to the overhead resulting from applying arithmetic coding. In this case, outputting the values of the coefficients directly to the output bit stream is more efficient.

Process_LIS:
for every block (i, j, bs)
if bs=2,
output the symbol corresponding the context of 2x2 block (16 contexts),

if this block have significant bits, output the sign bits;

If bs=4,

If this block is zero block,

output “0”;

else

coding every 2x2 block as a symbol,

if this block have significant bits, output the sign bits;

If bs>4

If this block is zero block,

output “0”;

else

output “1”;

test which kind of block dividing method is suitable this block,

case there is only one significant bit,

encode this block using coordinate representation method and using quad-tree method to divide this block. The insignificant blocks are put into LIS, insignificant pixels are put into LIP, and significant pixels are put into LSP;

case there is one ¼ zero block,

divide this block into four sub blocks, and do Code_sub-block process;

case there is no ¼ zero block,

divide this block into 16 sub blocks, and do Code_sub-block process;

Beside the sign bits, the generated symbols and the dividing route information are entropy coded.

Where the Code_sub-block process is almost same as Process_LIS, only two places are not the same. Firstly, the coding objects are not the same, Code_sub-block codes the sub-blocks. Secondly, no bit is used for last non-zero block if all other blocks are zero blocks. The Code_sub-block process may be recursively applied to sub sub_blocks many times until all points in the LIS are partitioned into suitable lists (LIS, LIP, or LSP). The Code _sub-block process is as follows.

Code_sub-block:

for every sub block (i, j, bs)

if bs=2,

output the symbol corresponding the context of 2x2 block (16 contexts),

if this block have significant bits, output the sign bits;

If bs=4,

If this block is zero block,

output “0”;

else

coding every 2x2 block as a symbol,

if this block have significant bits, output the sign bits;

If bs>4

If this block is zero block,

output “0”;

else

if this block is the last block and all other blocks are zero block,

nothing;

else

output “1”;

end

test which kind of block dividing method is suitable this block,

case there is only one significant bit,

encode this block using coordinate representation method and using quad-tree method to divide this block. The insignificant blocks are put into LIS, insignificant pixels are put into LIP, and significant pixels are put into LSP;

case there is one ¼ zero block,

divide this block into four sub blocks, and do Code_sub-block process;

case there is no ¼ zero block,

divide this block into 16 sub blocks, and do Code_sub-block process;

beside the sign bits, the generated symbols and the dividing route information are entropy coded.

When we apply the algorithm to the most significant bit-plane coding, the lists of LIP and LSP are empty. After the coding process is finished, three lists all have some content. When coding the next bit-plane, the coding order is from LIP, LIS to LSP as illustrated in the algorithm.

5.4 A Simple Example

In this section, a simple example is used to demonstrate the block dividing coding algorithm described in previous section. Table 8 shows the array of coefficients. The numbers in the top line and left column are used for representation of the coordinate position.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	6	18	-5	4	1	0	0	2	0	0	0	0
1	0	0	0	0	0	12	-5	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	-2	0	0	4	0	0	-1	0	0	0
3	0	0	5	0	0	0	0	0	24	0	0	0	0	2	3	1
4	0	2	23	16	0	0	0	0	0	12	-5	0	0	0	-1	0
5	0	0	11	0	0	0	0	23	0	-2	23	-6	0	0	0	0
6	0	0	0	0	0	6	18	0	0	0	-36	22	-8	0	0	0
7	0	0	0	0	23	12	5	0	0	-12	-31	1	8	3	0	0
8	0	0	-6	-12	-56	32	3	0	0	-6	0	4	4	-20	5	0
9	0	0	1	0	-23	33	-6	0	12	48	3	16	63	-36	12	0
10	0	0	0	0	0	20	-4	0	6	50	12	0	14	22	8	0
11	0	1	-1	0	0	6	-16	0	0	21	13	53	37	11	3	0
12	0	0	0	-3	-13	-48	-42	12	0	8	0	12	0	4	0	1
13	0	3	0	0	0	-10	34	8	13	45	51	22	0	12	-11	0
14	0	0	5	0	3	0	7	1	0	12	23	0	-3	-38	-46	6
15	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	8	0

Table 8 Array of coefficients

The maximum absolute value is 63; it means this block has six bit-planes. Every bit-plane except the most significant bit-plane is encoded by two passes: the significant pass and the refinement pass. The significant pass is very important; it provides the location information of the significant coefficients. The refinement pass is very simple, it records the magnitude bit of all coefficients that were encoded in the previous pass. The coding order is from the most significant bit-plane to the least significant bit-plane. Figure 25 shows the distribution of significant bits in the most significant bit-plane. It is extracted from Table 8. The following steps refer to the coding process, the coding outputs and the contents of the three lists are demonstrated:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	1-	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	1-	1+	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	1+	0	0	0	1+	0	0	1+	1-	0	0
10	0	0	0	0	0	0	0	0	0	1+	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1+	1-	0	0	0
12	0	0	0	0	0	1-	1-	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	1+	0	0	1+	1+	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	1-	1-	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 25 The number and distribution of significant bits on the bit-plane

1. First test whether there is one ¼ zero sub-block. In this example, we do have such a sub-block, we divide this block into four sub-blocks (8x8) and we use yellow colour to highlight that sub-block. We output a 0 to represent this kind of dividing method. The first sub-block is encoded as 0, and this sub-block is put into LIS. One bit can represent 64 zero bits. We use the (x, y, sb) format to represent the tested insignificant block, x and y denote the upper left coordinate position and sb is the block size. We use the (x, y) to represent the tested pixels, where x, y are the coordinate position of the pixels. The positive sign is represented as + while the negative sign is represented as -. The first sub-block coding process results in the following output and the contents of the 3 lists are summarised below:.

   Output: [0 0];

   LIS: [(0, 0, 8)];

   LIP: [ ];

   LSP: [ ];

2. The second sub-block is a non-zero block; it is highlighted by the yellow colour as shown in Figure 26. Firstly, it is encoded as 1. It only has one significant bit. We recursively use the quad-tree splitting method to divide non-zero blocks into four sub blocks until the 2x2 block, output “10” is used to represent this dividing method. The position of the significant pixel in the sub-block is (6, 2), and is encoded with sign symbol as 110 010 -. All zero blocks are put into the LIS, because they are insignificant blocks. The positions of insignificant pixels are put into the LIP, while the position of significant pixel is put into the LSP. The output and three lists are as shown below.

   Output: [0 0, 1 1 0 1 1 0 0 1 0 -];
   LIS: [(0,0,8),(0,8,4),(0,12,4),(4,8,2),(4,10,2),(6,8,2),(4,12,4)];
   LIP: [(6,11),(7,10),(7,11)];
   LSP :[(6,10)];

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	1-	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	1-	1+	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	1+	0	0	0	1+	0	0	1+	1-	0	0
10	0	0	0	0	0	0	0	0	0	1+	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1+	1-	0	0	0
12	0	0	0	0	0	1-	1-	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	1+	0	0	1+	1+	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	1-	1-	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 26 Second sub block

3. The third block is also a non-zero block. Firstly, it is encoded as 1. This block has two zero sub-blocks. We divide this block into four 4x4 sub-blocks, and we output 0 to represent this kind of dividing method. The first sub-block is

encoded as 0 and is put into the LIS. The second sub-block is a non-zero block and it is firstly encoded as 1. It has four 2x2 sub blocks. Figure 27 shows the dividing process. Every 2x2 sub block is encoded as one symbol. We use 16 symbols to represent the possible contexts of the 2x2 blocks. The first 2x2 sub-block in the second 4x4 block is encoded as 13, followed by the sign bits:-++. The three remaining 2x2 blocks are encoded as symbols: 0 0 0. The third 4x4 sub-block is encoded as 0. The fourth 4x4 block is encoded as 4 - 10 - + 0 0. All zero sub-blocks are put into the LIS; all tested insignificant pixels are put into the LIP while all tested significant pixels are put into the LSP.

Output: [0 0, 1 1 0 1 1 0 0 1 0 -, 1 0 13 - + + 0 0 0 0 4 - 10 - + 0 0];
LIS: [(0,0,8),(0,8,4),(0,12,4),(4,8,2),(4,10,2),(6,8,2),(4,12,4),(8,0,4),(8,6,2),
(10,4,2),(10,6,2),(12,0,4),(14,4,2),(14,6,2)];
LIP: [(6,11),(7,10),(7,11),(9,4),(12,4),(13,4),(13,5),(12,7),(13,7)];
LSP: [(6,10),(8,4),(8,5),(9,5),(12,5),(12,6),(13,6)];

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	1-	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	1-	1+	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	1+	0	0	0	1+	0	0	1+	1-	0	0
10	0	0	0	0	0	0	0	0	0	1+	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1+	1-	0	0	0
12	0	0	0	0	0	1-	1-	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	1+	0	0	1+	1+	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	1-	1-	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 27 Third block

4. The fourth block is also a non-zero block and is firstly encoded as 1. It has no zero sub-block, we divide this block into 16 2x2 blocks, and we use 11 to represent this dividing method. Every 2x2 block is encoded as one symbol plus

the corresponding sign bits (zero blocks have no sign bits). Every zero 2x2 block is put into the LIS. Other corresponding pixels are put into the corresponding list: LIP and LSP. Figure 28 shows the dividing process. The output and corresponding lists are as follows:

Output: [0 0, 1 1 0 1 1 0 0 1 0 -, 1 0 13 - + + 0 0 0 0 4 - 10 - + 0 0, 1 1 1 1 + 0 3 + - 0 4 + 1 + 2 - 0 1 + 2 + 0 0 0 0 4 - 8 -];

LIS: [(0,0,8),(0,8,4),(0,12,4),(4,8,2),(4,10,2),(6,8,2),(4,12,4),(8,0,4),(8,6,2), (10,4,2),(10,6,2),(12,0,4),(14,4,2),(14,6,2),(8,10,2),(8,14,2),(10,14,2),(12,12,2), (12,14,2),(14,8,2),(14,10,2)];

LIP: [(6,11),(7,10),(7,11),(9,4),(12,4),(13,4),(13,5),(12,7),(13,7),(8,8),(8,9), (9,8),(8,12),(8,13),(10,8),(11,8),(11,9),(10,10),(10,11),(11,10),(10,12), (10,13),(11,13),(12,8),(12,9),(13,8),(12,10),(12,11),(13,11),(14,12),(15,12), (15,13),(14,15),(15,14),(15,15)];

LSP :[6,10),(8,4),(8,5),(9,5),(12,5),(12,6),(13,6),(9,9),(9,12),(9,13),(10,9), (11,11),(11,12),(13,9),(13,10),(14,13),(14,14)];

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	1-	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	1-	1+	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	1+	0	0	0	1+	0	0	1+	1-	0	0
10	0	0	0	0	0	0	0	0	0	1+	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1+	1-	0	0	0
12	0	0	0	0	0	1-	1-	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	1+	0	0	1+	1+	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	1-	1-	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 28 Fourth block

So far, the most significant bit-plane has been encoded completely. Before coding the next bit-plane, the blocks in the LIS list should be reordered, from the smallest block

size to the biggest block size. This reorder operation will improve embedded performance.

Then, we encode the next bit-plane. Coding the next bit-plane is a little bit different from encoding the most significant bit-plane. It has two coding passes: the significant pass and the refinement pass. Now we have three lists: LIS, LIP, and LSP. LIS and LIP are for significant pass, while the LSP is for refinement pass. The coding order is:

1. Firstly, we test every pixel in the LIP, using three symbols [Z(ero), P(ositive), N(egative)] to code it, if the pixel is a significant bit, remove this pixel from LIP, and put it into LSP;
2. Then, we test and code every block in the LIS generating new sub-blocks;
3. Lastly, we do a refinement pass, the bit value of every pixel in the LSP except those pixels added in this bit-plane is recorded as the coding result;

Step 1 and step 2 together are called the significant pass. Step 3 is for the refinement pass. The three lists will be updated during the process of coding. The remaining bit-planes are encoded as described above. Firstly, we do the significant coding pass, and then we do the refinement coding pass.

Up until now, the coding results include a mixture of symbols. Before we put them into the next coding stage: optimized rearrangement coding, it should be further coded by entropy coding. The decoder uses the same rules to reconstruct the wavelet coefficients and create the same lists.

5.5 Entropy Coding

Arithmetic coding plays a key role in image compression using wavelet transforms. It is a well-known method for lossless compression. It can give compression efficiency at or very near entropy. The number of binary bits for a symbol is given by the following formula:

$$- \mathrm{Log}_2 p_i(s)$$

where $p_i(s)$ is the probability of i-th symbol in the message sequence. There are two kinds of model for calculating the probability: a fixed model and an adaptive model. The fixed model is the simplest model in which the symbol's probabilities are fixed. This kind of model is suited for static compression. Adaptive model represents the

changing probabilities seen so far in the sequence of symbols. Specifically, the probabilities are updated as each symbol is seen. Although arithmetic coding can provide the compression performance at entropy level, in principle, the practical efficiencies are always less than the ideal results. There are three factors affecting the efficiency: terminating message, finite precision arithmetic and scaling of the counts.
The decoder can losslessly recover the original sequence of symbols, if the decoding terminating point is properly signalled at the decoder. There are two ways of terminating the decoding: 1) provide the number of symbols in the beginning of the compressed bit-stream; 2) use a unique terminating symbol at the end of sequence of symbols to inform the decoder that it should stop. We use the first method in out implementation. Regardless of which method is used, some overheads in the compressed bit-stream cannot be avoided.

Arithmetic coding is used to further compress the symbols and the binary bits generated from the coding procedures discussed in the above sections. The arithmetic coding we used is based on [30]. In a practical implementation, counters are used to collect the statistics of the symbols and the length of the counter for each symbol in the arithmetic coding is 8 bits; this means that the maximum count value is 255. All counters are initialized to 1, when the sum of all counters reaches the maximum count value, and each counter is incremented by 1 and divided by 2, keeping the relative frequency of each symbols unchanged. In BDC algorithm, the following symbols need to be arithmetic coded.

1. The binary bits from the LSP.
2. The symbols from the LIP;
3. The symbols for the 2x2 blocks from LIS;
4. The binary bits about the three dividing methods, the significance test results and the coding results about the blocks with only one significant bit, those binary bits are all from the LIS.

5.5.1 Entropy coding of the binary bits from the LSP

The binary bits from the LSP are the refinement information. Compression of those binary bits is more difficult. In SPIHT, SPECK and SBHP, the refinement information was simply put into the final bit-stream. It is very difficult to compress this information because the probabilities of 0 and 1 bits in the bit stream are almost the same. For

example, one bit-stream has 1010100110110010, and has the same number of zeros and ones, according to arithmetic coding theory, the probability for zero and one is the same, and therefore, no compression is achievable. We use one combining approach to overcome this difficulty and to achieve a higher compression ratio. For example, we combine two continuous binary bits as one symbol (00—A, 01—B, 10—C, and 11—D), the original bit-stream is now represented by a symbol sequence as CCCBCDAC, which provides each symbol with a different probability facilitating further compression with arithmetic coding.

In our implementation, four different combining lengths are tested. When the length is one, it is equal to binary arithmetic coding. For the example given in the last section, two bits are combined. After compression with arithmetic coding for each combining length, four different bit-streams with different lengths are obtained, and we choose the shortest compressed bit-stream to send out. If the number of bits in the bit stream to be compressed is not enough to be divided by the combined length, zero or zeros are added at the end of the bit stream. For example, when the binary bits are 1010100110110010 and the combined length is 3, the combined symbols become 101,010,011,011,001,0**00**. The final two (bold) 0 bits are the added bits. Although we added two bits to the bit-stream, in the decoder side, we can still correctly recover the original bits, because the decoder knows how many refinement bits there are from the LSP list.

The experimental results show that we can get 3~8% compression. For example, the number of refinement bits of Lena image using the (9, 7) filter is 134799, after compression, the number decreases to 124971; the compression ratio is 7%.
Considering the fact that arithmetic coding itself has some overheads in the compressed bit-stream (e.g. the bits for the length of compressed bit-stream and the combined length), arithmetic coding may not result in any gain, if the original bit-stream is not long enough. In our implementation, for the refinement pass, we set up the shortest length of the original refinement bit-stream to 260, to which arithmetic coding will be applied. If the length of the original bit stream is less than 260, we simply put the bit-stream onto the final bit-stream.

5.5.2 Entropy coding of the symbols from the LIP

The symbols from LIP are: Z, P and N, they correspond to the insignificant bit, positive significant bit and negative significant bit. The contents of the LIP list are the coordinates of the tested insignificant bits in previous passes. In the BDC algorithm we use the arithmetic coding method to those symbols only when the number of symbols is greater than the threshold T1 (in our experiments the T1 value is set to 600), otherwise, we use the direct scanning to generate the binary bits as the algorithm described, that is, the binary bit 0 is for the insignificant bits, the bits 10 are for the positive significant bits, and the bits 11 are for the negative significant bits.

5.5.3 Entropy coding of the symbols of the 2x2 blocks from LIS

Recalling the example in section 5.6, the coding results from the LIS are a mixture of symbols and binary bits, some of them are the binary bits, the others are for 2x2 blocks. In the practical implementation, the mixed symbols are separated into three buffers: the buffer one is for the symbols from 2x2 blocks, the buffer two is for sign bits of the significant bits, the buffer three is for the remain bits, which include the binary bits about the three dividing methods, the significance test results of the blocks and the coding results about the blocks with only one significant bit. The symbols of the buffer one are coded with the adaptive context model and the number of contexts is 16, the sign bits of the buffer two does not need to be entropy coding, since there is no coding gain. How to code the contents of the buffer three is described in the following section. We use 16 symbols instead of 15 symbols that are adopted in other quad-tree methods like SPECK. Our experimental tests confirm that using 16 context symbols is more effective than 15 symbols.

5.5.4 Entropy coding of the binary bits in the buffer three from the LIS

Buffer three includes the binary bits about the three dividing methods, the significance test results of the blocks and the coding results about the blocks with only one significant bit. The entropy coding method for these binary bits is the same as the method described in section 5.6.1. We use the combined bits as inputs to the adaptive arithmetic coding. After four times of compression, we choose the shortest compressed bit-stream as the results.

Up until now, we get three parts for the binary bit-stream, one is from the entropy coding results of buffer one, buffer two is the sign bits, and the last is the entropy coding results of buffer three. We need to combine these three components together to form one binary bit-stream. The algorithm is described below. We use bufb1, bufb3 to represent the entropy coding results from the buffer one and buffer three, bufb2 has the sign bits, out3p gives the combined results. It is worth pointing out that bufb3 may be empty if there is no blocks with size greater than 4 in LIS.

The assembling algorithm:

Initiate out3p as empty;
If bufb3 is empty
    Store the length of bufb1 and bufb2 in out3p;
    Extract 10 bits from bufb1 and 3 bits from bufb2 and
    add them to out3p until finished;
Else
    Store the length of bufb1, bufb2 and bufb3 in the out3p;
    Extract 10 bits from bufb1, 3 bits from bufb2 and 2 bits from bufb3 and
    add them to out3p until finished;
End

On the decoder side, using the same rules to extract the three parts of the binary bits and applying arithmetic decoding to the first and third parts to recover the original symbols or binary bits, with the second part as the sign bits, it does not need arithmetic decoding.

To this point, all coding results are binary bit-stream packed. The results for every significant pass are called the significant packet, while the results from the refinement pass are called the refinement packet. Figure 29 shows the coding results from every sub-band.

Next, we use optimized rearrangement coding to generate the final full bit-stream.

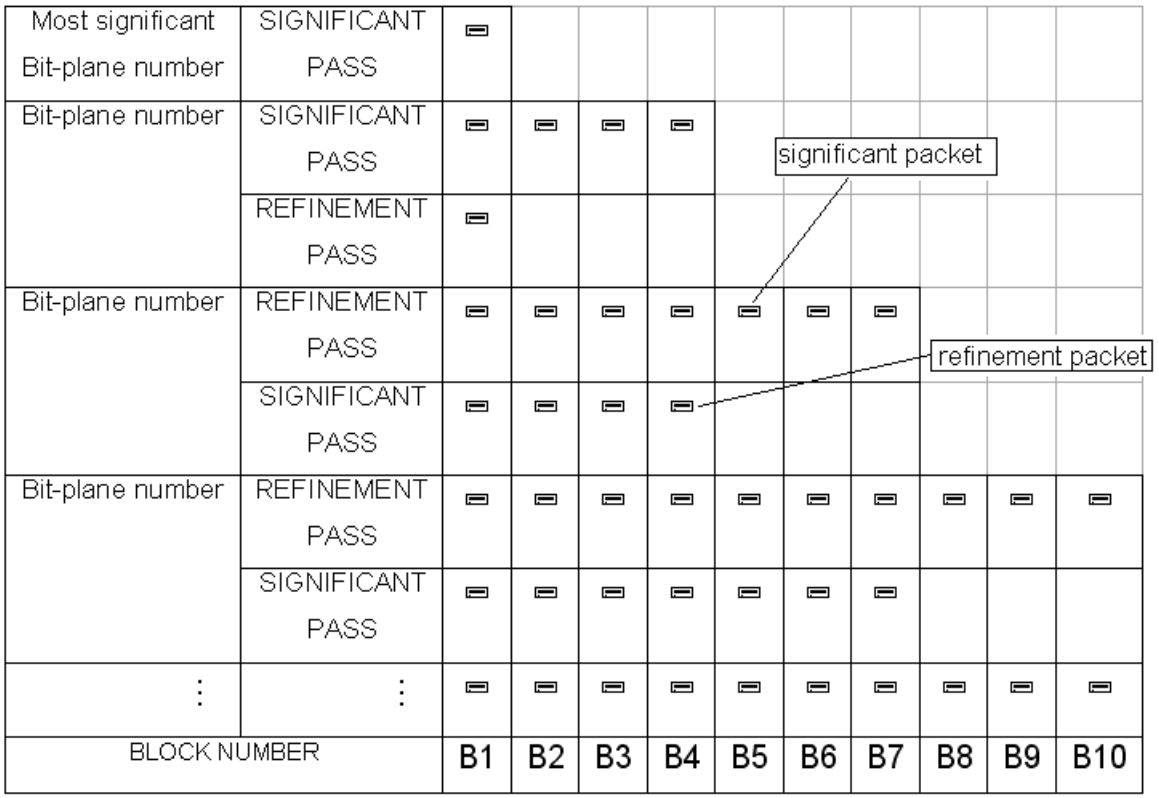


Figure 29 The coding results in every bit-plane

6 Optimized assembling coding

This chapter describes how to assemble the bit-stream packets to form the final fully embedded bit-stream. After the first stage of the block dividing coding, we get the collection of bit-stream packets. The final bit stream needs to be formed by reordering the packets, such that the reconstructed image always has the best possible quality at any decoding termination point.

6.1 Introduction

In the BDC algorithm, every sub-band is encoded independently from the most significant bit-plane to the least significant bit-plane. The bit-stream results from every sub-band itself are embedded. The bit stream obtained from coding each bit-plane of a sub-band is called a bit-stream packet. It is an intermediate packet and needs to be reorganized to form the final bit-stream.

Generally, two assembly methods can be used for generating the final bit-stream:

1. Natural assembling;
2. Optimized assembling.

Natural assembling is the simplest assembling method, which assembles the bit-stream packets from the lowest sub-band to the highest sub-band within the same level bit-plane and starting from the most significant bit-plane to the least significant bit-plane. Different from Natural assembling, optimized assembling is a method that always chooses the bit-stream packet from among the available packets with the best contribution to image quality.

If the wavelet coefficients have been scaled very close to a unitary transform, the natural assembling is very efficient. However, in general the final bit-stream generated from this method is not optimized for the best R-D (rate vs. distortion) performance, due to the following points.

1. Firstly, the coefficients on different bit-planes of different sub-band have different contributions to compression and reconstructed image quality;

2. Secondly, the scaled wavelet coefficients can only be close to the unitary transform, and cannot be really a unitary transform. This is especially the case for the integer wavelet transform.

Obviously, the optimized assembling method is not affected by these two factors, where the contributions of the bit-stream packets from each sub-band to the mean square error (MSE) of the image are compared and the packet of the sub-band with the biggest contribution to MSE is chosen to be sent out.

EZW, SPIHT, and SPECK combine two coding stages as one stage; they adopt the first method to generate the final bit-stream. The coding is ordered from the lowest sub-band to the highest sub-band starting from the most significant bit-plane to the least significant bit-plane, and the loss of the performance is not a big issue, because the wavelet coefficients are scaled. However, those methods cannot be directly used for the integer wavelet transform from the (5, 3) filters. EZBC also uses the first method to rearrange the bit-stream. EBCOT and SBHP use the second method to generate the final optimal bit-stream.

6.2 Optimized Assembling Coding Algorithm

With BDC algorithm the optimised packet assembly algorithm is similar to the one used by EBCOT, but with some differences. We only compare the contribution of the bit-stream packet from each sub-band to the image quality, and then choose the one with the best contribution to image quality to send to the final bit-stream. More specifically, before one bit-stream packet from a sub-band is added to the final bit-stream, we do the inverse wavelet transform to get the reconstructed image, and then we calculate the actual mean-squared error (MSE) between the reconstructed image and the original image, that is:

$$\mathrm{MSE}=\sum_{i} Wi(Xi-Ri)^2$$

where, X_i, R_i are the sample of ith sub-band on the original image and the reconstructed image respectively, W_i is the weighting factor, it is considered as the visual frequency weighting. Part 1 of JPEG2000 lists the recommended frequency weighting factors for three different viewing conditions. In our implementation, W_i is set to one.

We use an example to describe the optimized assembling algorithm. It is assumed that an image has been decomposed using a 3-level, 2-D dyadic wavelet transform, and we get 10 sub-bands as shown in Figure 30. With the optimized assembling algorithm in BDC, the most significant packet from the B1 sub-band is sent first, the next one may be from B2 or other sub-band rather than the B1 sub-band. The details of the optimized assembling algorithm are shown below.

The optimized assembling algorithm:

1. First send out the bit-stream packet on the most significant bit-plane of B1 sub-band (the lowest sub-band);
2. Calculate the first mse using the first packet and we have mse0;
3. Fetch one bit-stream packet from each sub-band (B1, B2, B3 …and B10) and calculate the mse for each packet, we can get ten new mses: mse(1), mse(2), mse(3), … and mse(10). We use L(i) to denote the length of each packet;
4. Calculate (mse0 – mse(i))/L(i), the result is called the average mse per bit, which represents the contribution each bit makes to the decrease of mse.. Now we have a_mse(1), a_mse(2), a_mse(3), … and a_mse(10);
5. Find out the maximum a_mse(i), and send out packet associated with a_mse(i);
6. Update the mse0 with the new mse;
7. Continue steps 3-6 until the target bit-rate is reached.

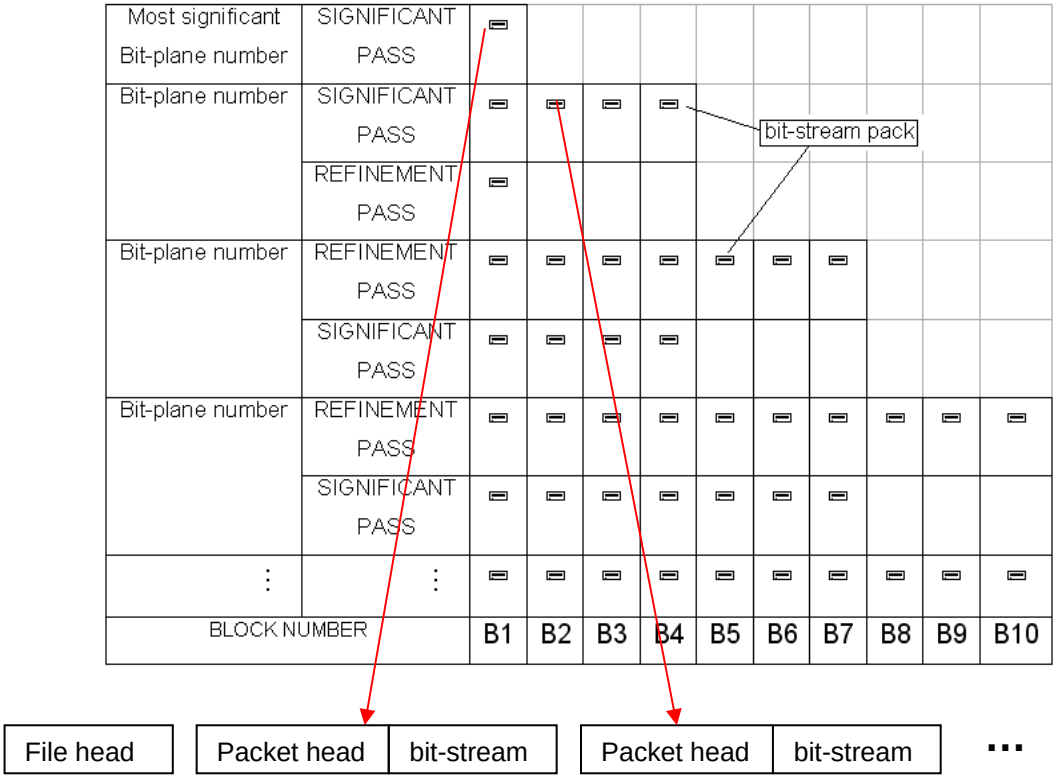


Figure 30 The process of the optimized assembling coding

The algorithm, in principle, is similar to EBCOT. The difference is that EBCOT divides the wavelet coefficients matrix into many small square blocks; this will lead to more accurate block selection.

6.3 Packet Formation

The data structure of the bit-stream packet for every bit-plane of a sub-band (including significant pass and refinement pass) is shown in Table 9

sub-band number	Bit-plane number	Whole Significant bits	Length of bit-stream for significant pass	Bit-stream for significant pass	Length of bit-stream for refinement pass(from second bit-plane)	bit-stream for refinement pass(from second bit-plane)
8bits	4 bit	16 bits	32 bits	Depend on the length	32 bits	Depend on the length

Table 9 Data structure of bit-stream packet

The optimised assembling coding machine reads the bit stream packet to form the final bit-stream. After the optimised assembling coding, the data structure of the final bit-stream is showed in Table 10. It comprises of two parts: file head and bit-plane packet. Every bit-plane packet has two parts: packet head and body.

The final bit stream now includes all the necessary information for the decoder to be able to decode the bit stream. The information includes image size, the number of wavelet transform levels, sub-band number, bit-plane number, the mark for significant pass and refinement pass, bit-stream packets and so on in the bit stream packet.

File head			Bit-plane packet				
			Packet head				Body
Lossy/ Lossless mark	wavelet level	Image Size	sub-band Number	Bit-plane Number	Significant/ Refinement Mark	Length of bit-stream	bit-stream
1 bit	5 bits	12 bits	5 bits	4 bit	1 bit	Depend on the size of the sub-band	

Table 10 Data structure of final bit-stream

Until now, we have described the all-coding processes of BDC. It consists of two stages. The first stage is the coding of every sub-band from the most significant bit-plane to the least significant bit-plane, while the second stage is the optimized assembly coding; which is responsible for generating the final fully embedded bit-stream.

6.4 Key Features of BDC

6.4.1 Precise rate control

The encoding process can be stopped at any target bit-rate.

6.4.2 Resolution scalable

Block dividing coding is based on the sub-band. The final bit-stream consists of packets; every packet has a head descriptor, which includes the sub-band number, bit-

plane number, significant/refinement mark bit. It is easy to decode the bit stream at the desired image resolution.

6.4.3 SNR scalable

The compressed final bit-stream is the embedded bit-stream and can be stopped at any target PSNR or bit rate.

6.4.4 High compression performance

The experimental results are reported in Chapter 7. In a lossless model, our simple method outperforms the complicated JPEG2000.

6.4.5 Error resilience

It is not difficult to see that the block dividing coding provides some abilities to withstand errors; the bit-stream from every pass is only associated with the specific sub-band and bit-plane, and does not affect other parts.

6.4.6 Parallelism

Since every sub-band is encoded independently, this kind of coding method provides the opportunity that all sub-bands can be coded in parallel.

7 Numerical Results

This chapter presents the performances test results of the proposed BDC algorithm with comparisons to other coding methods. Some reconstructed images are also displayed. Some analysis and discussions on the results are given.

7.1 Introduction

In lossy compression, the quality of the reconstructed image is always measured by the PSNR (peak signal-to-reconstructed image measure). For an 8-bit image, the PSNR is defined as:

$$PSNR=10\log_{10}(255^2/MSE)$$

where MSE refers to the mean squared error between the original image and the reconstructed image. The bit rate of the image is always expressed in bpp (bits per pixel). Generally, smaller MSE or larger PSNR values mean lower level distortion. To conduct the tests, images are compressed with bit rates of 0.125, 0.25, 0.5, 1.0, and 2.0 bits/pixel, the resulting bit streams are then used to reconstruct the images using BDC and other different decoding algorithms, the PSNRs are then calculated and compared.

In lossless compression, we use the final bpp (bits per pixel) to measure the compression performance, the bpp of the original grayscale images is 8, and a smaller bpp means the high compression ratio.

7.2 Test Conditions

All the tests with BDC and other algorithms are conducted under the following conditions.

1. The original images are dyadicly transformed using wavelet transform with decomposition levels up to five;
2. The Daubechies (9, 7) filter is used in lossy coding, while a (5, 3) filter is used in both lossy and lossless coding;
3. The three popular test images “Lena”, “Barbara” and “Goldhill” were used in our test. They all are grayscale images and have a bit-depth of 8 bits/pixel. The image sizes all are 512x512. Figure 31 shows those original images.

The Lena image contains many small details like hair and a hat with large smooth areas. The Barbara image contains many objects with sharp edges like a table and books. It also has a lot of fine textures such as the scarf, trousers and a wicker armchair. It contains more information than Lena image. The Goldhill image contains many different textures: for example walls, windows and tiles on a house roof. The main scene contains a couple of houses on a steep hill. The background is distant trees and landscape scenery. It is the most complex image of the three images. Thus it requires more bits to represent it in the coding process.

Lena

Barbara

Goldhill

Figure 31 The three popular test images

7.3 Test Results

7.3.1 Lossless compression performance

The three test images are compressed in a lossless manner using the (5,3) filter. Lossless compression means no distortion is introduced in the encoding and decoding process between the reconstructed and the original image. Table 11 shows the compression performance. The results show that the performance of our BDC algorithm outperforms JPEG2000, while the complexity of BDC is significantly lower than JPEG2000.

		Lena	Barbara	Goldhill
JPEG2000	Total bit amount	1137248	1230408	1276576
	Bits per pixel	4.3382	4.6936	4.8697
BDC	Total bit amount	1108822	1216725	1248642
	Bits per pixel	4.2298	4.6414	4.7632

Table 11 Comparison of lossless compression performance

7.3.2 Lossy compression performance using the (9, 7) filter

In lossy coding, the floating-point wavelet coefficients are scaled to be unitary or nearly unitary so that the distortion in the transform domain can be directly related to the distortion in the pixel domain. Table 12, Table 13 and 14 show the compression performance in the lossy situation. From the results, we can see that the performance of the BDC algorithm is close to other coding methods with only JPEG2000 showing consistent advantage over other algorithm at low bit rate, where the bit rate is smaller than 1. The performance results about SBHP are from [28, p395]. Figure 32, Figure 33 and Figure 34 show the reconstructed images from our method at the 0.0625, 0.125, 0.25, 0.5 and 1.0 bit-rates.

bpp	0.0625	0.125	0.25	0.5	1.0	2.0
EZW	23.10	24.03	26.77	30.53	35.14	-
SPIHT	23.35	24.86	27.58	31.40	36.41	42.65
SBHP	-	24.32	27.50	31.56	36.64	42.81
JPEG2000	23.38	25.28	28.55	32.48	37.37	43.43
BDC	23.60	25.13	27.55	31.86	37.09	43.57

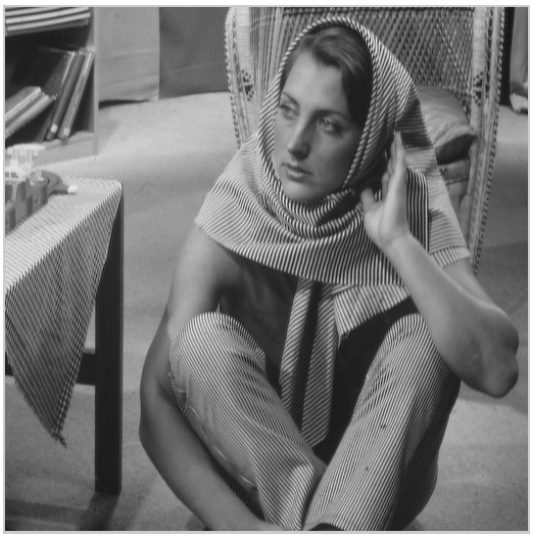
Table 12 The PSNR performance of Barbara using the (9, 7) filter

bpp	0.0625	0.125	0.25	0.5	1.0	2.0
EZW	27.54	30.23	33.17	36.28	39.55	-
SPIHT	28.38	31.10	34.11	37.21	40.41	45.07
SBHP	-	30.26	33.06	36.51	39.68	44.10
JPEG2000	27.99	31.22	34.28	37.43	40.61	44.72
BDC	27.85	30.60	33.60	36.77	40.05	44.76

Table 13 The PSNR performance of Lena using the (9, 7) filter

bpp	0.0625	0.125	0.25	0.5	1.0	2.0
EZW	-	-	30.31	32.87	36.20	-
SPIHT	26.73	28.48	30.56	33.13	36.55	42.02
JPEG2000	26.59	28.52	30.71	33.35	36.72	42.23
BDC	26.49	28.15	30.31	32.80	36.19	41.62

Table 14 The PSNR performance of Goldhill using the (9, 7) filter



Original

Rate=0.0625, PSNR=23.60

Rate=0.125, PSNR=25.13

Rate=0.25, PSNR=27.55

Rate=0.5, PSNR=31.86

Rate=1.0, PSNR=37.09

Figure 32 The reconstructed images of Barbara

Original

Rate=0.0625, PSNR=27.85

Rate=0.125, PSNR=30.60

Rate=0.25, PSNR=33.60

Rate=0.5, PSNR=36.77

Rate=1.0, PSNR=40.05

Figure 33 The reconstructed images of Lena

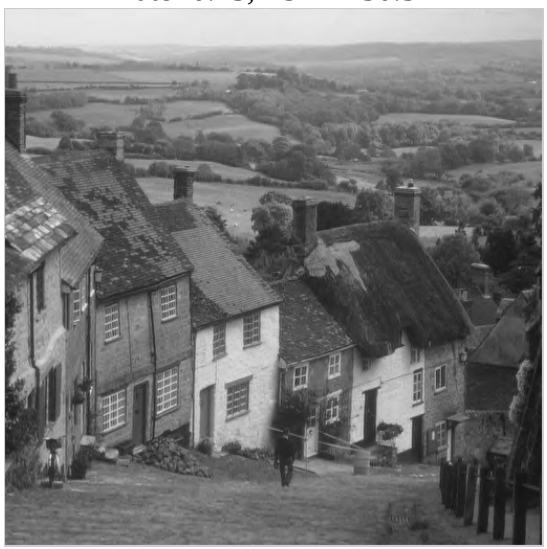
Original

Rate=0.0625, PSNR=26.49

Rate=0.125, PSNR=28.15

Rate=0.25, PSNR=30.31

Rate=0.5, PSNR=32.80



Rate=1.0, PSNR=36.19

Figure 34 The reconstructed images of Goldhill

7.3.3 Lossy compression performance using the (5, 3) filter

One important characteristic of a coder using wavelet transform is that it can be used for progressive lossy-to-lossless compression in a single framework. The reversible wavelet transformation can support such functions and is adopted in the JPEG-2000. We use the (5, 3) filter to test the lossy compression performance of the BDC algorithm.

When the non-reversible transform is used, the wavelet transform coefficients are easily scaled close to unity so that the coefficients in different sub-bands have the same energy weight. However, when the reversible transform is employed, the coefficients all are integer data; it is not easy to scale those coefficients to unity, and the scaling factor may cause bit growth or data expansion, and it is not efficient for lossless data compression. The existing full-band coding algorithms are efficient when applied to the scaled coefficients and inefficient to the unscaled coeffcients since they cannot distinguish the importance of the coefficients. The BDC algorithm solves this problem by using the optimized assembling algorithm without scaling process. Table 15 and 16 show the compression results for the BDC algorithm using both natural assembly and optimized assembly comparing with the performance of JPEG2000. It can be seen that the PSNRs obtained using optimized assembly are much better than PSNRs obtained using natural. From those results, we can see that the optimized assembling coding method can make the lossy-to-lossless compression in a single coding framework and do not need to consider the scalar factor on designing the integer transform filters. Because the optimised assembling coding can choose the best bit-stream packet from sub-bands to form the final bit-stream, the compression performances of the BDC algorithm outperforms JPEG2000 in lossless coding while close to the JPEG2000 in lossy coding.

bpp	0.0625	0.125	0.25	0.5	1.0	2.0
JPEG2000	22.80	24.61	27.26	30.78	35.71	41.30
BDC (Natural)	17.91	21.98	22.75	27.51	32.14	37.40
BDC (optimized)	23.47	24.96	27.04	29.52	35.83	41.72

Table 15 Loss performance of Barbara using the (5, 3) filter (in dB)

bpp	0.0625	0.125	0.25	0.5	1.0	2.0
JPEG2000	27.43	30.11	33.17	36.24	39.23	43.30
BDC (natural)	23.32	27.41	28.37	32.33	36.63	40.75
BDC (optimized)	27.44	30.10	32.99	36.17	39.27	43.34

Table 16 Loss performance of Lena using the (5, 3) filter (in dB)

7.4 Analysis of Experimental Results

Having briefly presented the various experimental results, we now analyze those results in more detail. Recall chapter 4, we pointed out that four factors can affect the final compressed bit-stream, factor 2-4 have been considered in the BDC algorithm. Here we only discuss the influences of the first factor: filters and we examine the performance behavior at low and high bit rates.

7.4.1 Filters

The influence of filters to image compression is significant. Choosing the right filters for an image is very important. Beside the filters itself, the number of wavelet transform levels and the policy for the extension of the boundary of the image to be compressed also have some effects on the final compression result. Table 17 shows the performance difference between different filters. This tells us that the classical floating-point (9,7) filter outperforms the reversible (5, 3) filter for coding the test image at all bit-rates. However, the differences are not significant at relatively low bit rates.

bpp	0.0625	0.125	0.25	0.5	1.0	2.0
(9,7) filter	23.60	25.13	27.55	31.86	37.09	43.57
(5, 3) filter	23.47	24.96	27.04	29.52	35.83	41.72
Difference	-0.13	-0.17	-0.51	-2.34	-1.26	-1.85

Table 17 Loss performance of Barbara using the (5, 3) filter and the (9, 7) filter (in dB)

7.4.2 Bit rate

In lossy coding, the BDC algorithm is comparable with JPEG2000 at low bit rates. However, at extremely low bit rates and high bit rates, the BDC algorithm outperforms JPEG2000. Table 18 shows the performance difference between the BDC and JPEG2000 obtained with the Barbara image. Even though we cannot claim that the

BDC algorithm outperform the JPEG2000 at all low bit-rates, its simplicity still makes it a good candidate to be applied in reality.

bpp	0.0625	0.125	0.25	0.5	1.0	2.0
JPEG2000	23.38	25.28	28.55	32.48	37.37	43.43
BDC	23.60	25.13	27.55	31.86	37.09	43.57
Difference	+0.22	-0.15	-1.0	-0.62	-0.28	+0.14

Table 18 The difference of the PSNR performance of Barbara using the (9, 7) filter

7.5 Discussion

Even though the BDC algorithm is motivated and developed based on some features from EZW, SPIHT, SBHP and JPEG2000 algorithms, it is different from these coding algorithms in various aspects.

EZW and SPIHT is a spatial-orientation-tree-based fully embedded coder, which employs progressive transmission by coding bit planes in decreasing order. It is well known that the coefficients of the wavelet transform exhibits similarities across its sub-bands at the same spatial orientation. This property makes it possible to group the transform coefficients in the form of spatial orientation trees, which can be exploited in efficient coefficient representation to achieve compression. However, there is an increasing demand for some desirable properties for image coders, such as random access, resolution scalability and ROI (region of interesting). Unfortunately, the orientation tree structure makes these two coding algorithms difficult to possess these features.

SBHP is a block-based embedded coder, which employs quad-tree partitioning or grouping technique for exploiting the fact that the blocks of coefficients with high probability of being zero cluster together in a particular bit plane. Coding these coefficients in blocks is much more efficient than coding them one-by-one. But the single partitioning technique does not provide the best performance and adapts to a wide range of images.

EBCOT also is a block-based embedded coder which employs the fractional bit-plane coding idea and adaptive context-modeling technique for achieving the high compression performance and the desirable properties such as SNR, resolution

scalabilities, random access and ROI function. But the method itself is too complicated to be understood and implemented.

The BDC coding algorithm is a fully embedded block-based coder which employs progressive transmission by coding bit planes in decreasing order. Instead of the single grouping technique used in SBHP, It employs three dividing methods to group the coefficients into blocks with variable size. And it uses the adaptive arithmetic coding to achieve further efficiency using an efficient context-modeling technique. The optimized assembly of packets makes the progressive lossy-to-lossless compression possible in a single framework and generate the final fully embedded bit-stream with the desirable properties such SNR, resolution scalabilities and random access. All those functionalities are achieved at low-complexity, which makes the BDC a very efficient block-based embedded image coder.

We now do a complexity analysis. Recalling the BDC algorithm; where we use three lists to indicate the coding order. For the LIP and the LSP, we only visit every pixel once. For the LIS, when the size of a block is less than or equal to 4, the visiting number to the pixel is also one. When the size of a block is greater than 4, the BDC first visits all pixels to determine which kind of block dividing methods to use, then every zero block is encoded as one zero bit, and this block is only visited once. Other non-zero blocks need to be recursively divided into 2x2 blocks, the visiting number is unknown, but can be computed by the entropy of the block. In the EBCOT, every pixel needs to be scanned three times, and for determining the context of a pixel, every pixel plus the surrounding pixels needs to be looked up in a large of table. This process is more complex.

The optimized assembling coding also is low complexity. It only chooses the packet with the best contribution to the image quality for the final bit-stream, there is no complexity calculation about the R-D curve in the EBCOT.

8 Conclusions and Future Research

In this thesis, we proposed a new low-complexity image coding method – BDC. The proposed method utilizes the block dividing coding method plus optimized packet assembly to achieve high compression performance. The high performance can be attributed to the use of the following techniques:

1. Discrete wavelet transform;
2. Block dividing coding method;
3. Optimized assembling coding;
4. Adaptive arithmetic coding.

The final output is a fully embedded bit-stream. The block dividing coding method features a low complexity with high compression performance. The experimental results show that the performance in the lossless coding outperforms JPEG2000, though the performance in the lossy coding close to the performance of JPEG2000 at low bit rates.

8.1 Discrete Wavelet Transform

The wavelet transform provides a compact multi-resolution representation of the image in transform domain. Most of the transform coefficients are small or zero, that is why the wavelet coefficients can be compressed. The integer DWT decomposition can be used for lossless and lossy compression. The embedded bit-stream that results from the bit-plane coding in every sub-band (or small blocks) provides scalabilities that include SNR, resolution, random access, and many other features.

8.2 Block dividing Coding Method

The block dividing coding method is a very effective aprroach with low-complexity. Three block dividing methods quickly group large zero areas into blocks with variable sizes while maintaining the high energy area in small blocks, and it is effective for a wide range of images. Three lists indicate the coding order, and they ensure the generated bit-stream is embedded.

8.3 Optimized Packet Assembling

The optimized packet assembling is a key factor in the BDC algorithm. It always sends the bit-stream packet from sub-bands to the final bit stream, which has the best contribution to the image quality. The final bit-stream is a fully embedded bit-stream.

This proposed assembling method also provides a new opportunity for designing the integer wavelet filters without considering the scalar factor. Different filters provide different results of coefficient compaction, which provide further potential to achieve further improved compression performance.

8.4 Adaptive arithmetic coding

We use the adaptive arithmetic coding to improve the performance. When coding the binary bit-stream (e.g. bit-stream from refinement pass), consecutive bits with different bit lengths are combined to create new symbols that have different statistical characteristics providing further opportunities to improve the coding efficiency. When coding the 2x2 blocks, we use 16 symbols rather than 15 symbols that were adopted by other quad-tree methods.

8.5 Future Research

Future work can be focused in the following directions.

1. ROI (Region of Interest) feature. Region of interest (ROI) coding is important in applications where certain parts of an image are of a higher importance than the rest of the image. In these cases the ROI is decoded with higher quality and/or spatial resolution than the background (BG). In the BDC algorithm the ROI has not been considered. The ROI feature needs to be added to the BDC algorithm in the future.
2. The block partition and grouping in the BDC algorithm are mainly based on a quad-tree structure resulting in rectangular blocks. In future, irregular blocks can be considered. The main concern with using irregular blocks is that the overhead with chain coding on the irregular block boundary will be too expensive. However, if we can define an effective coding algorithm to compress the chain code, the irregular block could be a good candidate for application in image coding.

References

[1] ISO/IEC 15444-1: Information technology—JPEG2000 image coding system—Part 1: Core coding system, 2000.

[2] J. M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. IEEE transactions on signal processing, Vol. 41, No. 12, pp. 3445-3462, Dec 1993.

[3] A. Said and W. A. Pearlman. A new, fast, and efficient image codec based on set partitioning in hierarchical trees. IEEE transactions on circuits and systems for video technology, Vol. 6, No. 3, pp. 243-250, June 1996.

[4] D. Taubman. High performance scalable image compression with EBCOT. IEEE Transactions on Image Processing, Vol. 9, No. 7, pp. 1158 – 1170, July 2000.

[5] C. Chrysafis, A. Said, A. Drukarev, A. Islam and W. A. Pearlman. SBHP—A low complexity wavelet coder. IEEE Int. Conf. Acoust., Speech Signal Processing (ICASSP), Vol. 4, pp. 2035–2038, June 2000.

[6] Agostino Abbate, Casimer M. DeCusatis and Pankaj K. Das. *Wavelets and Sub-bands fundamentals and applications*. Berlin: Birkhauser Boston, 2002.

[7] A. Grossman and J. Morlet. Decomposition of hardy functions into square integrable wavelets of constant shape. SIAM Journal on Mathematical Analysis, 15(4), pp. 723-736, July 1984.

[8] R. E. Crochiere, S. A. Webber, and J. L. Flanagan. Digital coding of speech in sub-bands. Bell System Technical Journal, 55(8), pp. 1069-1085, October 1976.

[9] D. Estaban and C. Galand. Application of quadrature mirror filters to split band voice coding schemes. Proc. International Conference on Acoutsics, Speech and Signal Processing (ICASSP), pp. 191-195, 1977.

[10] M. J. T. Smith and T. P. Barnwell III. A procedure for designing exact reconstruction filter banks for tree-structured sub-band coders. Proc. IEEE Intl. Conf. ASSP, Vol. 9, pp. 27.1.1-27.1.4, March 1984.

[11] J. W. Woods and S. D. O'Neil. Sub-band coding of images. IEEE Trans. On Acoustcs, Speech, and Signal Processing, Vol. 34, No. 5, pp. 1278-1288, Oct, 1986.

[12] I. Daubechies. The wavelet transform, time-frequency localization and signal analysis. IEEE trans. On Inform. Theory, Vol. 36, No. 5, pp. 961-1005, September 1990.

[13] S. G. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. IEEE trans. on Pattern Analys. and Machine Intell., Vol. 11, No. 7, pp. 674-693, July 1989.

[14] JPEG2000 Verification Model 7.0 (Technical description). ISO/IEC JTC 1/SC29/WG1 WG1N1684. April 25, 2000.

[15] E. H. Adelson, E. Simoncelli and R. Hingorai. Orthogonal pyramid transforms for image coding. Visual communications and Image processing II, SPIE vol. 845, 1987.

[16] M. Antonini, M. Barlaud, P. Mathieu and I. Daubechies. Image coding using wavelet transform. IEEE trans. on image processing. Vol. 1. No. 2. April 1992.

[17] R. L. Joshi, V. J. Crump, and T. R. Fisher. Image sub-band coding using arithmetic coded trellis Coded quantization. IEEE Trans. on circuits and systems for video technology, Vol. 5, NO. 6, pp. 515-523, December 1995.

[18] J. Andrew. A simple and efficient hierarchical image coder. Proc. IEEE Int. Conf. Image Processing (ICIP), Vol. 3, pp. 658–661, Oct. 1997.

[19] A. Said and W. A. Pearlman. Low-complexity waveform coding via alphabet and sample-set partitioning. Proc. SPIE Visual Communications and Image Processing, Vol. 3024, pp. 25–37, Feb. 1997.

[20] A. Islam and W. A. Pearlman. An embedded and efficient low-complexity hierarchical image coder. Proc. SPIE Visual Communications and image processing, Vol. 3653, Jan. 1999.

[21] S.-T. Hsiang. Embedded image coding using zeroblocks of sub-band/wavelet coefficients and context modeling. IEEE Int. Conf. Circuits and Systems (ISCAS), vol. 3, pp. 83-92, May 2001.

[22] W. A. Pearlman, A. Islam, N. Nagaraj and A. Said. Efficient, low-complexity image coding with a set-partitioning embedded block coder. IEEE transactions on circuits and systems for video technology, Vol. 14, No. 11, November 2004.

[23] B. E. Usevitch. A tutorial on modern lossy wavelet image compression: Foundation of JPEG2000. IEEE signal processing magazine, Vol. 18, Issue 5, pp. 22-35, September 2001.

[24] D. Taubman, E. Ordentlich, M. Weinberger, and G. Seroussi. Embedded block coding in JPEG2000. Signal procession: Image communication 17, pp. 49-72, 2002.

[25] M. Rabbani, and R. Joshi. An overview of the JPEG2000 still image compression standard. Signal processing: Image communication 17, pp. 3-48, 2002.

[26] M. D. Adams. The JPEG-2000 still image compression standard. ISO/IEC JTC 1/SC 29/WG1 N2412, 2002.

[27] J. Li. Image Compression: The mathematics of JPEG2000. Modern signal processiong, MSRI publications Vol. 46, 2003.

[28] D. S. Taubman, and M. W. Marcellin. *JPEG2000 Image compression fundamentals, standards and practice*. London: Klumer academic publishers group, 2002.

[29] M. D. Adams. Reversible integer-to-integer wavelet transforms for image coding. Ph.D. thesis, The university of British Columbia, September 2002.

[30] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. Communications of the ACM, Vol. 30, pp. 520-540, June 1987.

Appendix

The block dividing coding (BDC) software consists of many functions. Here we only extract four main control functions: BDC_encoder, BDC_sub-band_assemble, BDC_decoder, and BDC_pass_decoding.

A.1 BDC_encoder

The BDC_encoder program is the main control program for generating the fully embedded bit-stream. From this program, we can see six processes: pre-processing, DWT, scalar quantization, rate-control, bit-plane encoder and assembling.

```
function BDC_encoder
% Matlab implementation of BDC
%
% main program
%
% description: variable X is for selection of different source image;
%        variable rate is for rate-control;
%        variable lossy is for selection lossy/lossless compression
%        varibale level is the number of wavelet transform level
%        variable assembling is for selection of the ordering/optimized assembling model
%  the compressed bit-stream: img_s2
%
% Jihai Zhu
% Contact email : jihaizhu1@hotmail.com
% 2006

clear all; close all;

%--------------- reading image ------------------------------------------
X1 = func_ReadRaw('lena512.raw', 512*512, 512, 512);
X2 = func_ReadRaw('barbara_t.raw', 512*512, 512, 512);
X3 = func_ReadRaw('goldhill_t.raw', 512*512, 512, 512);
X4 = func_ReadRaw('baboon_t.raw', 512*512, 512, 512);
X5 = func_ReadRaw('peppers_t.raw', 512*512, 512, 512);
X8=imread('lena128.pgm'); %128X128

%figure,imshow(uint8(X1));

%------------ choose the different data source ----------------------------
X=X8;
% setup the number of wavelet transform
level=3;

%------selection of lossy and lossless compression-----------------------
```

```
% setup the swithch mark lossy/lossless
lossy=1; % 1 is for lossy, 0 is for lossless

%------selection of assembling model ---------------------------------
assembling=0; % o means the ordering assembling model, 1 means optimized

%-------- pre-processing ------------------------------------------------
% change to signed number according to J2000
X=double(X);

% DC-level shifting
if lossy==1
   X=X-128;
else
   %X=X-128;
   X=128-X; % same as JJ2000
end

%---------------- the filter parameters ----------------------------------

%9/7 filter from JPEG2000 verification model 7.0[WG1N1684(p81)]
Lo_D=[0.026748757411,-0.016864118443,-
0.078223266529,0.266864118443,0.602949018236,0.266864118443,-
0.078223266529,-0.016864118443,0.026748757411];
Hi_D=[0.09127176311424948,-0.05754352622849957,-
0.5912717631142470,1.115087052456994,-0.5912717631142470,-
0.05754352622849957,0.09127176311424948];

% calculate synthesis filter from analyse filter
%g0=h1(-Z);
%g1=-h0(-z);
lf0=length(Lo_D);
lf1=length(Hi_D);
for i=1:lf1
   Lo_R(i)=Hi_D(i)*(-1)^((lf1+1)/2-i);
end
for i=1:lf0
   Hi_R(i)=Lo_D(i)*(-1)^((lf0+1)/2-i);
end

%------------------ rate control ------------------------------------------
rate=0; % 0 means the coding all bits, the unit is bpp(bits per pixel)
[r,c] = size(X);
max_bits = floor(rate * r*c);
S=gens(X,level); % S represents the every sub-band size

%-------------- multi_level DWT for lossy----------------------------------
if lossy==1
   %--------------lossy transform using 9/7 filter ----------------------
   [I_W, S] = func_DWT_J2(X, level, Lo_D, Hi_D);
```

```
   %I_W1=I_W; % I_W1 record the results from dwt
else
   %-------lossless transform using 5/3 filter by lifting-----------------
   I_W=dwt53(X,level);
end
% display the picture in wavelet domain
%nbcol = size(X,1);
%cp1_n = uint8(wcodemat(I_W,nbcol));
%figure; imshow(cp1_n);
%title('the wavelet image');

%-------------scalar quantization ----------------------------------------
if lossy==1
   I_W=scalarq(I_W,level,S); % only effective for level:3 or 5
else
   % nothing to do for lossless
   dd=0;
end

% calculate the bit amount of new significant and refinement points
%cx=fix(I_W);
%[nx,ny]=find(abs(cx)>0);% non-zero points
%[y,z]=cal_bits(cx);
%I_W=fix(I_W);
%[r,c]=size(I_W);

%--------- bit-plane encoding and optimized rearrangement coding-----------
%img_s2=N46_ECPL_Enc2s(X,I_W,S,level,max_bits,r*c);
img_s2=BDC_sub-band_assemble(X,I_W,S,level,max_bits,r*c,lossy,assembling);
% place the lossy mark in the first place
if lossy==1
   img_s2=[1,img_s2];
else
   img_s2=[0,img_s2];
end
% transform to single character
csy=transbc(img_s2);
% store the bit-stream in the file
fd2=fopen('bit-stream.txt','w');
for ii=1:length(csy)
   fprintf(fd2,'%c',csy(ii));
end
%fwrite(fd2,csy,'double');
fclose(fd2);

% calculate the compression ratio for lossless
if lossy==0
   [r,c]=size(X);
   cr=length(img_s2)/(r*c); %compression ratio
end
```

```
ed=0; % stop here
% end of file!
```

A.2 BDC_sub-band_assemble

This program is a controlling program, it includes two parts. In the first part there is a controlling program for the bit-plane encoding of sub-bands, the details for the bit-plane encoding were introduced in chapter 5. The second part is for generating the final bit-stream, there are two assembling styles: natural ordering assembling and the optimized assembling.

```
function out=BDC_sub-band_assemble(Y,X,S,level,bitnumber,blocksize,lossy,assembling);
% name:      encoder: fistly coding every sub-band, then assembling
%            into the final bit-stream
% Y:         original image
% X:         the coefficients of matrix;
% S:         the size of every band
% level:     the number of level of wavelet transform;
% bitnumber: the length of coded bit stream; when bitnumber
%            equal zero, coder all coefficients
% out:       the bit stream for transmitting to decoder;

global cobs cop cop12 cop14 cbm14 cbm57;
cobs=0; % whole blocks which greater or equal 4x4
cop=0;  % position number
cop12=0; % 1/2 dividing number
cop14=0; % 1/4 dividing number
cbm14=0; % number of combined bit(1-4)
cbm57=0; % number of combined bit(5-7)

% 1 is for lossy coding, using optimized rearrangement coding,
% 0 is for lossless coding, assembling bit-stream in order
%assembling_mark=1; % 0 means the assembling in order, 1 means the optimized
```

```
%lossy=0; % 1 means lossy compression, 0 means lossless compression

%----------------coding every sub-band ----------------------------------
% extract every layer's coefficients
[r,c]=size(X);
% first ectract LL band size
LL=X(1:S(1,1),1:S(1,2));
pl=ceil(log(r*c*8)/log(2)); % pl bits are used for layer length
bandnumber=1;
out_LL=N928_ECPL_EncLL_ac(LL,bandnumber);  % partition with tracking list
osb(1)={out_LL};
dout=[];
dout=[...
   dout,length(out_LL),out_LL,...
   ];

% coding other three band from top level to lowest level
for sb=1:level
   % extract other three sub-bands and do coding
 HL=X(1:S(sb+1,1),(1+S(sb+1,1)):S(sb+2,1));
 LH=X((1+S(sb+1,1)):S(sb+2,1),1:S(sb+1,1));
 HH=X((1+S(sb+1,1)):S(sb+2,1),(1+S(sb+1,1)):S(sb+2,1));
 bandnumber=bandnumber+1;
 out_HL=N928_ECPL_EncLL_ac(HL,bandnumber);
 bandnumber=bandnumber+1;
 out_LH=N928_ECPL_EncLL_ac(LH,bandnumber);
 bandnumber=bandnumber+1;
 out_HH=N928_ECPL_EncLL_ac(HH,bandnumber);
 osb((sb-1)*3+2)={out_LH};
 osb((sb-1)*3+3)={out_HL};
 osb((sb-1)*3+4)={out_HH};
 dout=[...
    dout,length(out_HL),out_HL,...
    ];
```

```
 dout=[...
     dout,length(out_LH),out_LH,...
     ];
 dout=[...
     dout,length(out_HH),out_HH,...
     ];
end
% save to disk file
fd2=fopen('intermedia.txt','w');
fwrite(fd2,dout,'double');
fclose(fd2);

%---------- begin to assembling ---------------------------------
fd2=fopen('intermedia.txt','r');
%fd2=fopen('lena97m.txt','r');
%fd2=fopen('barbara53m.txt','r');
%fd2=fopen('goldhill53m.txt','r');
dout=fread(fd2,inf,'double');
fclose(fd2);
dout=dout';

if assembling==0
   % assembling bit-stream packs in order
   % assign the bit-stream to every band
   cl=1;
   for sb=1:level*3+1
      len=dout(cl);
      cl=cl+1;
      buf1=dout(cl:cl+len-1); % read bit-stream
      cl=cl+len;
      %buf1=buf1';
      osb(sb)={buf1};
   end
```

```
% ---------- rearrange the bit stream -------------------------
overall_band=level*3+1;
ob=tranb(overall_band,5); % overall_band number
lb=tranb(level,5); %level number
rb=tranb(r,12); % size of image
out=[lb,rb]; %initiate the out: overall band number, row size of image
if bitnumber~=0
   remainbit=bitnumber-17; % 16 bits are used up
end

% set up the index point to the begin of every band
c_osb(1:overall_band)=2;
% initiate every state
for i=1:overall_band
   rs(i)=osb{i}(c_osb(i)); % get the row size
   c_osb(i)=c_osb(i)+1;
   % maximum length of one layer using binary represent
   pl2(i)=ceil(log(rs(i)*rs(i)*2)/log(2))+1;
   r_l2(i)=ceil(log(rs(i))/log(2)); %r_l is the binary length of row number
   pn2=osb{i}(c_osb(i):c_osb(i)+3); % get the maximum layer number of every band
   p_n2=transB(pn2);
   c_layer(i)=p_n2;
   add_l(i)=0; % add_l  means adding which pass, 1: refine, 0:new significant
   topsent(i)=0;
end

% top layer
top_layer=c_layer;

% maximum layer number
pn1=max(c_layer);

% output every pass from top to bottom
for pass=(pn1*2-1):-1:1
```

```
% calculate the corresponding layer
layer=floor((pass+1)/2);
for band=1:overall_band
   % check which pass should be sent out
   if add_l(band)==0 % new significant pass
      if length(out)==434992
         dd=0;
      end
      % try to get the layer number
      if top_layer(band)~=layer
         lb=osb{band}(c_osb(band):c_osb(band)+3);
         c_layer(band)=transB(lb);
      end
      if c_layer(band)<layer
         continue;
      end
      if c_layer(band)==top_layer(band) & topsent(band)==1
         continue;
      end
      c_osb(band)=c_osb(band)+4;
      % get the whole significant points
      nop{band}(c_layer(band))=osb{band}(c_osb(band));
      c_osb(band)=c_osb(band)+1;
      % adjust the length for store of bit-stream for significant pass
      if c_layer(band)==top_layer(band)
         pl3(band)=pl2(band);
      else
         if (rs(band)*rs(band)-nop{band}(c_layer(band)+1))==0
            pl3(band)=5; % at least, there are 5 bits
         else
            pl3(band)=ceil(log((rs(band)*rs(band)-
nop{band}(c_layer(band)+1))*2)/log(2))+1;
         end
      end
```

```
            % get the length of the bit-stream
            length6=osb{band}(c_osb(band):c_osb(band)+pl3(band)-1);
            c_osb(band)=c_osb(band)+pl3(band);
            leng7(band)=transB(length6);
            if leng7(band)==0 % means no new significant
               % change the pointer
               add_l(band)=1;
               c_layer(band)=c_layer(band)-1;
               continue;
            end

            % get the bit-stream of the layer
            buf=osb{band}(c_osb(band):c_osb(band)+leng7(band)-1);
            c_osb(band)=c_osb(band)+leng7(band);
            bn=tranb(band,5); % band number needs 5 bits
            p_n2=tranb(layer,4); % layer number
            out=[...
               out,bn,p_n2,0,length6,buf,... % band number, layer number,
significant/refine mark,
               % length of bit-stream, bit-stream
               ];
            % adjust the pass mark
            if c_layer(band)==top_layer(band)
               add_l(band)=0; % remain the previous state
               topsent(band)=1;
            else
               add_l(band)=1;
            end
         else
            % send the refinement bit-stream
            % adjust the length for store of bit-stream for refinement pass
            pl31(band)=ceil(log(nop{band}(c_layer(band)+1)+3)/log(2));
            % get the length of the significant pass
```

```
            length6=osb{band}(c_osb(band):c_osb(band)+pl31(band)-1);
            leng6=transB(length6);
            c_osb(band)=c_osb(band)+pl31(band);

            % get the bit-stream of the layer
            buf=osb{band}(c_osb(band):c_osb(band)+leng6-1);
            c_osb(band)=c_osb(band)+leng6;
            bn=tranb(band,5); % band number needs 5 bits
            p_n2=tranb(layer,4); % layer number
            out=[...
               out,bn,p_n2,1,length6,buf,... % band number, layer number,
significant/refine mark,
               % length of bit-stream, bit-stream
               ];
            % adjust the pass mark
            add_l(band)=0;
         end
         % check the overall length of bit-stream
         if bitnumber~=0
            remainbit=remainbit-length(out);
         end
      end
      dd=0;
   end
   dd=0;
else
   % assembling process using optimized rearrangement coding
   cl=1;
   for sb=1:level*3+1
      len=dout(cl);
      cl=cl+1;
      buf1=dout(cl:cl+len-1); % read bit-stream
      cl=cl+len;
      %buf1=buf1';
```

```
    osb(sb)={buf1};
end
% ----------- rearrange the bit stream -------------------------
overall_band=level*3+1;
ob=tranb(overall_band,5); % overall_band number
lb=tranb(level,5); %level number
rb=tranb(r,12); % size of image
out=[lb,rb]; %initiate the out: overall band number, row size of image
if bitnumber~=0
    remainbit=bitnumber-17; % 16 bits are used up
end

% set up the index point to the begin of every band
c_osb(1:overall_band)=2;
% initiate every state
for i=1:overall_band
    rs(i)=osb{i}(c_osb(i)); % get the row size
    c_osb(i)=c_osb(i)+1;
    % maximum length of one layer using binary represent
    pl2(i)=ceil(log(rs(i)*rs(i)*2)/log(2))+1;
    r_l2(i)=ceil(log(rs(i))/log(2)); %r_l is the binary length of row number
    pn2=osb{i}(c_osb(i):c_osb(i)+3); % get the maximum layer number of every band
    p_n2=transB(pn2);
    c_layer(i)=p_n2;
    add_l(i)=0; % add_l  means adding which pass, 1: refine, 0:new significant
    topsent(i)=0;
end

% top layer
top_layer=c_layer;

% maximum layer number
pn1=max(c_layer);
```

```
% first output the top layer of LL band
c_osb(1)=1; % set up the index point to the begin of every band
ln=osb{1}(c_osb(1)); % get the index point of first band number
c_osb(1)=c_osb(1)+1;
lr=osb{1}(c_osb(1)); % get the row size
c_osb(1)=c_osb(1)+1;
pl2=ceil(log(lr*lr*2)/log(2))+1; % maximum length of one layer using binary represent
r_l2=ceil(log(lr)/log(2)); %r_l is the binary length of row number
%p_nm=osb{1}(c_osb(1):c_osb(1)+3); % get the maximum layer number

for i=1:1 % output the first top layers
    p_n2=osb{1}(c_osb(1):c_osb(1)+3); % get the layer number
    p_n1=transB(p_n2);
    c_osb(1)=c_osb(1)+4;
    nop{1}(p_n1)=osb{1}(c_osb(1));% get the whole significant point
    c_osb(1)=c_osb(1)+1;
    length6=osb{1}(c_osb(1):c_osb(1)+pl2-1); % get the length of the layer
    leng6=transB(length6);
    c_osb(1)=c_osb(1)+pl2;
    buf=osb{1}(c_osb(1):c_osb(1)+leng6-1); % get the bit-stream of the layer
    c_osb(1)=c_osb(1)+leng6;
    bn=tranb(1,5); % sub-band number needs 5 bits
    out=[...
        out,bn,p_n2,0,length6,buf,... % band number, layer number, significant/refine mark,
        % length of bit-stream, bit-stream
        %out,p_n2,length6,buf,...
        ];
    if bitnumber~=0
        remainbit=remainbit-length(out);
    end
end
% calculate the mse of top layers
```

```
LLD(1:r,1:c)=0;
A=set_a(pn1);

% extract the top layer of LL band
for rowb=1:r
   for colb=1:c
      if rowb<=S(1,1) & colb <=S(1,1)
         if X(rowb,colb)>=0
            LLD(rowb,colb)=bitand(X(rowb,colb),A(p_n1));
            if LLD(rowb,colb)>0
               LLD(rowb,colb)=LLD(rowb,colb)+2^(p_n1-2);
            end
         else
            LLD(rowb,colb)=bitand(abs(X(rowb,colb)),A(p_n1));
            if LLD(rowb,colb)>0
               LLD(rowb,colb)=LLD(rowb,colb)+2^(p_n1-2);
               LLD(rowb,colb)=-LLD(rowb,colb);
            end
         end
      end
   end
end
LLB={LLD}; % store the coefficients in the LLB
LLD1=LLD;
add_l(1)=0; % add_l  means adding which layer, 1: refine, 0:new significant

if lossy==1
% irreverse wavelet transform
%9/7 filter from JPEG2000 verification model 7.0[WG1N1684(p81)]
Lo_D=[0.026748757411,-0.016864118443,-
0.078223266529,0.266864118443,0.602949018236,0.266864118443,-
0.078223266529,-0.016864118443,0.026748757411];
```

```
Hi_D=[0.09127176311424948,-0.05754352622849957,-
0.5912717631142470,1.115087052456994,-0.5912717631142470,-
0.05754352622849957,0.09127176311424948];
% calculate synthesis filter from analyse filter
[Lo_R,Hi_R]=gfromh(Lo_D,Hi_D);

% dequantization
%Nor97=[8.41675 4.18337 2.07926 1.99681 0.96722 1.01129 0.52022];
Nor97=[33.924847 17.166698 8.686716 8.534109 4.3004827 4.18337 2.07926
1.99681 0.96722 1.01129 0.52022];
Nor97=Nor97/2;
LLD(1:S(1,1),1:S(1,2))=LLD(1:S(1,1),1:S(1,2))*(1/Nor97(1));
for sb=1:level
   % extract other three sub-bands and do coding

LLD((1+S(sb+1,1)):S(sb+2,1),1:S(sb+1,1))=LLD((1+S(sb+1,1)):S(sb+2,1),1:S(sb+1,1))
*(1/Nor97(sb*2)); % LH band

LLD(1:S(sb+1,1),(1+S(sb+1,1)):S(sb+2,1))=LLD(1:S(sb+1,1),(1+S(sb+1,1)):S(sb+2,1))
*(1/Nor97(sb*2)); % HL band

LLD((1+S(sb+1,1)):S(sb+2,1),(1+S(sb+1,1)):S(sb+2,1))=LLD((1+S(sb+1,1)):S(sb+2,1)
,(1+S(sb+1,1)):S(sb+2,1))*(1/Nor97(sb*2+1)); % HH band
end
%LL4 = func_InvDWT_J2T(LLD1, S, Lo_R, Hi_R, level);
LL4 = func_InvDWT_J2(LLD, S, Lo_R, Hi_R, level);
%figure,imshow(uint8(LL4));
else
   LL4=idwt53(LLD,level,S);
end
% calculate the mse
mse4=calcumse(Y,LL4);

% set up the size of every band
```

```
S1=S;
S1(1,1)=0;
S1(1,2)=0;

% set up the pointer, get the top layer number
c_layer(1)=p_n1-1; % the counter for significant layer number of LL band

% refine layer
cr_layer=top_layer-1;

if bitnumber==0
   remainbit=2;
end
% output the bitstream untill reach to rate bit
while (remainbit>0)
   % compare the mse per bit of one band
   for band=1:overall_band
      if c_layer(band)==0 & cr_layer(band)==0 % means this band all layer is out
         mse_a(band)=0;
         continue;
      end

      % calculate the mse of current top layer of every sub-band
      % initiate the wavelet matrix as zero
      LLC(1:r,1:c)=0;
      %wl=0; % which layer is the current layer belong?

      % transfer the band to resolution and LH, HL and HH area
      reso=floor((band+4)/3);
      area=mod(band-1,3);
      if reso==1
         pox0=1;
         poy0=1;
         pox1=S(2,1);
```

```
            poy1=S(2,2);
        else
            if area==1 %HL area
                pox0=1;
                poy0=S(2,1)*2^(reso-2)+1;
                pox1=S(2,1)*2^(reso-2);
                poy1=S(2,1)*2^(reso-1);
            else
                if area==2 %LH
                    pox0=S(2,1)*2^(reso-2)+1;
                    poy0=1;
                    pox1=S(2,1)*2^(reso-1);
                    poy1=S(2,1)*2^(reso-2);
                else % HH
                    pox0=S(2,1)*2^(reso-2)+1;
                    poy0=S(2,1)*2^(reso-2)+1;
                    pox1=S(2,1)*2^(reso-1);
                    poy1=S(2,1)*2^(reso-1);
                end
            end
        end

        % check whether there are some new significant
        if add_l(band)==0
            p_n2=osb{band}(c_osb(band):c_osb(band)+3); % get the layer number
            p_n3(band)=transB(p_n2);
            c_osb(band)=c_osb(band)+4;
            % get the whole significant points
            nop{band}(p_n3(band))=osb{band}(c_osb(band)); % get the whole
significant point
            % adjust the length for store of bit-stream for significant
            % pass
            if c_layer(band)==top_layer(band)
```

```
            pl3(band)=ceil(log((S1(reso+1,1)-S1(reso,1))*(S1(reso+1,1)-
S1(reso,1))*2)/log(2))+1; % maximum length of one layer using binary represent
        else
            %if (rs(band)*rs(band)-nop{band}(c_layer(band)+1))==0
            if                    ((S1(reso+1,1)-S1(reso,1))*(S1(reso+1,1)-S1(reso,1))-
nop{band}(p_n3(band)+1))==0
                pl3(band)=5;
            else
                %pl3(band)=ceil(log((rs(band)*rs(band)-
nop{band}(c_layer(band)+1))*2)/log(2))+1;
                pl3(band)=ceil(log(((S1(reso+1,1)-S1(reso,1))*(S1(reso+1,1)-
S1(reso,1))-nop{band}(p_n3(band)+1))*2)/log(2))+1;
            end
        end
        c_osb(band)=c_osb(band)+1;
        length6=osb{band}(c_osb(band):c_osb(band)+pl3(band)-1); % get the length
of the layer
        leng7(band)=transB(length6);
        if leng7(band)==0 % means no new significant
            % change the pointer
            c_osb(band)=c_osb(band)+pl3(band);
            add_l(band)=1;
            c_layer(band)=p_n3(band)-1;
        else
            c_osb(band)=c_osb(band)-5; % recovery the point
        end
    end

    if add_l(band)==0 % try send new significant

        % send new significant layer
        for rowb=1:r
            for colb=1:c
                if pox0<=rowb & rowb<=pox1 & poy0<=colb & colb<=poy1
```

```
                if c_layer(band)==top_layer(band)
                    if X(rowb,colb)>=0
                        LLC(rowb,colb)=bitand(X(rowb,colb),A(c_layer(band)));
                        if LLC(rowb,colb)>0
                            LLC(rowb,colb)=LLC(rowb,colb)+2^(c_layer(band)-2);
                        end
                    else
                        LLC(rowb,colb)=bitand(abs(X(rowb,colb)),A(c_layer(band)));
                        if LLC(rowb,colb)>0
                            LLC(rowb,colb)=LLC(rowb,colb)+2^(c_layer(band)-2);
                            LLC(rowb,colb)=-LLC(rowb,colb);
                        end
                    end
                else
                    if LLB{1}(rowb,colb)==0 % means this bit may be new significant,
LLB includes the coefficients which have been sent out
                        if X(rowb,colb)>=0
                            LLC(rowb,colb)=bitand(X(rowb,colb),A(c_layer(band)));
                            if LLC(rowb,colb)>0
                                LLC(rowb,colb)=LLC(rowb,colb)+2^(c_layer(band)-2);
                            end
                        else
                            LLC(rowb,colb)=bitand(abs(X(rowb,colb)),A(c_layer(band)));
                            if LLC(rowb,colb)>0
                                LLC(rowb,colb)=LLC(rowb,colb)+2^(c_layer(band)-2);
                                LLC(rowb,colb)=-LLC(rowb,colb);
                            end
                        end
                    else
                        LLC(rowb,colb)=LLB{1}(rowb,colb);
                    end
                end
            else
                LLC(rowb,colb)=LLB{1}(rowb,colb);
```

```
        end
      end
    end
  else
    % adjust the length for store of bit-stream for refinement pass
    pl31(band)=ceil(log(nop{band}(p_n3(band)+1)+3)/log(2));
    %pl31(band)=ceil(log(nop{band}(c_layer(band)+1)+3)/log(2));
    for rowb=1:r % try send refinement bit
      for colb=1:c
        if pox0<=rowb & rowb<=pox1 & poy0<=colb & colb<=poy1
          rfl=cr_layer(band)+1;
          ud=bitand(abs(X(rowb,colb)),A(rfl));
          if abs(ud)>0 % means this bit is refinement bit
            if X(rowb,colb)>=0
              LLC(rowb,colb)=bitand(X(rowb,colb),A(cr_layer(band)));
              if LLC(rowb,colb)>0
                LLC(rowb,colb)=LLC(rowb,colb)+2^(cr_layer(band)-2);
              end
            else
              LLC(rowb,colb)=bitand(abs(X(rowb,colb)),A(cr_layer(band)));
              if LLC(rowb,colb)>0
                LLC(rowb,colb)=LLC(rowb,colb)+2^(cr_layer(band)-2);
                LLC(rowb,colb)=-LLC(rowb,colb);
              end
            end
          else
            LLC(rowb,colb)=LLB{1}(rowb,colb);
          end
        else
          LLC(rowb,colb)=LLB{1}(rowb,colb);
        end
      end
    end
  end
```

```
LLO(band)={LLC}; % store the every kinds of coefficients matrix to the buffer area
%LLC1=LLC;

if lossy==1
% denormalization
LLC(1:S(1,1),1:S(1,2))=LLC(1:S(1,1),1:S(1,2))*(1/Nor97(1));
for sb=1:level
   % extract other three sub-bands and do coding

LLC((1+S(sb+1,1)):S(sb+2,1),1:S(sb+1,1))=LLC((1+S(sb+1,1)):S(sb+2,1),1:S(sb+1,1))*(1/Nor97(sb*2)); % LH band

LLC(1:S(sb+1,1),(1+S(sb+1,1)):S(sb+2,1))=LLC(1:S(sb+1,1),(1+S(sb+1,1)):S(sb+2,1))*(1/Nor97(sb*2)); % HL band

LLC((1+S(sb+1,1)):S(sb+2,1),(1+S(sb+1,1)):S(sb+2,1))=LLC((1+S(sb+1,1)):S(sb+2,1),(1+S(sb+1,1)):S(sb+2,1))*(1/Nor97(sb*2+1)); % HH band
end


   LL4 = func_InvDWT_J2(LLC, S, Lo_R, Hi_R, level);
%LL4 = func_InvDWT_J2T(LS, Lo_R, Hi_R, level);
%figure,imshow(uint8(LL4));
else
   LL4=idwt53(fix(LLC),level,S);
end

% calculate the mse
mse(band)=calcumse(Y,LL4);

if add_l(band)==0 % send new significant
   p_n2=osb{band}(c_osb(band):c_osb(band)+3); % get the layer number
   p_n3(band)=transB(p_n2);
```

```
            c_osb(band)=c_osb(band)+5; % pass the layer number
        end
        if add_l(band)==0
            length6=osb{band}(c_osb(band):c_osb(band)+pl3(band)-1); % get the length of the signifcant layer
        else
            length6=osb{band}(c_osb(band):c_osb(band)+pl31(band)-1); % get the length of the refinement layer
        end
        leng7(band)=transB(length6);
        mse_a(band)=(mse4-mse(band))/leng7(band); % averge mse per bit
        if add_l(band)==0
            c_osb(band)=c_osb(band)-5; % recovery the point
        end
    end   %( for band=1:overall_band

    mse_a=abs(mse_a); % if there is nagetive mse

    % find out the band which has the best a_mse as the candidate of output
    msea=mse_a(1);
    bd=1;
    for band=2:overall_band
        if msea<mse_a(band)
            bd=band;
            msea=mse_a(band);
        end
    end

    % send the bit-stream of that band to output
    if c_layer(bd)==top_layer(bd)
        nr=add_l(bd);
        add_l(bd)=0;
        c_osb(bd)=c_osb(bd)+5+pl3(bd);
        c_layer(bd)=p_n3(bd)-1;
```

```
    else
      if add_l(bd)==0  % send new significant bit
        c_osb(bd)=c_osb(bd)+5+pl3(bd);
        nr=add_l(bd);
        add_l(bd)=1;
        c_layer(bd)=p_n3(bd)-1;
      else
        c_osb(bd)=c_osb(bd)+pl31(bd); %send the refinement bit
        nr=add_l(bd);
        add_l(bd)=0;
        cr_layer(bd)=cr_layer(bd)-1;
      end
    end
    if bd==1 & cr_layer(bd)==1
      %  pause;
      dd=0;
    end
    if (length(osb{bd})-c_osb(bd)+1)>=buf
      buf=osb{bd}(c_osb(bd):c_osb(bd)+leng7(bd)-1); % get the bit-stream of the
layer
    else
      dd=0;
    end
    bn=tranb(bd,5); % band number needs 5 bits
    ln=tranb(p_n3(bd),4); % layer number
    if nr==0
      length7=tranb(leng7(bd),pl3(bd)); % length of significant bit-stream
    else
      length7=tranb(leng7(bd),pl31(bd)); % length of refinement bit-stream
    end
    % band number, layer number, significant/refine mark, length of
    % bit-stream, bit-stream
    out=[...
      out,bn,ln,nr,length7,buf,...
```

```
        ];
    % change the pointer
    c_osb(bd)=c_osb(bd)+leng7(bd);

    % chang the LLB matrix
    LLB=LLO(bd);
    mse4=mse(bd);

    % if the total bit is enough, break
    if bitnumber~=0
        if bitnumber<=length(out)
            out=out(1:bitnumber);
            break;
        else
            remainbit=bitnumber-length(out);
        end
    end
    % if all band are out, break
    cout=find(c_layer>0);
    crout=find(cr_layer>0);
    if length(cout)==0 & length(crout)==0
        break;
    end
end % {while}
dd=0;
end
```

A.3 BDC_decoder

BDC_decoder is a main controlling program for generating the reconstructed image at any target bit-rate, and calculates the PSNR value. It has four processes: bit-plane decoding, dequantization, IDWT and post-processing.

```
function BDC_decoder
% Matlab implementation of BDC
%
% decoder program
%
% description: variable X is for selection of different source image;
%              variable rate is for rate-control
% the reconstructed image: img_r22
% author name: Jihai Zhu
% Contact email: jihaizhu1@hotmail.com
% 2006

clear all; close all;

%---------------- the filter parameters ----------------------------------

%9/7 filter from JPEG2000 verification model 7.0[WG1N1684(p81)]
Lo_D=[0.026748757411,-0.016864118443,-
0.078223266529,0.266864118443,0.602949018236,0.266864118443,-
0.078223266529,-0.016864118443,0.026748757411];
Hi_D=[0.09127176311424948,-0.05754352622849957,-
0.5912717631142470,1.115087052456994,-0.5912717631142470,-
0.05754352622849957,0.09127176311424948];

% calculate synthesis filter from analyse filter
%g0=h1(-Z);
%g1=-h0(-z);
lf0=length(Lo_D);
lf1=length(Hi_D);
for i=1:lf1
   Lo_R(i)=Hi_D(i)*(-1)^((lf1+1)/2-i);
end
for i=1:lf0
   Hi_R(i)=Lo_D(i)*(-1)^((lf0+1)/2-i);
end

%-------------read the bit-stream file-----------------------------------
fd2=fopen('bit-stream.txt','r');
%fd2=fopen('barbara53opt.txt','r');
%fd2=fopen('barbara53inorder.txt','r');
%fd2=fopen('lena53inorder.txt','r');
%fd2=fopen('goldhill53inorder.txt','r');
%fd2=fopen('barbara97inorder.txt','r');
%fd2=fopen('lena97inorder.txt','r');
%fd2=fopen('goldhill97inorder.txt','r');
ch=fread(fd2,inf,'char');
fclose(fd2);
ch=ch';
% transform character to bit
img_s2=transcb(ch);
```

```
% read the lossy mark
lossy=img_s2(1);
img_s2=img_s2(2:length(img_s2));

% get the size of the image
br=img_s2(6:17);
r=transB(br);

%------------------ rate control ----------------------------------------
rate=0; % 0 means the coding all bits, the unit is bpp(bits per pixel)
%[r,c] = size(X);
max_bits = floor(rate * r*r);
if max_bits~=0
  img_s2=img_s2(1:max_bits);
end

% ------bit-plane decoding-----------------------------------------------
[img_w2,level]=N46_ECPL_Declis(img_s2);
img_w21=img_w2;
S=gens(img_w2,level); % S represents the every sub-band size

%--------------- dequantization -------------------------------------
if lossy==1
   img_w2=dequantization(img_w2,level,S);
end

%-------------- IDWT -------------------------------------------
if lossy==1
   img_r2 = func_InvDWT_J2(img_w2, S, Lo_R, Hi_R, level);
else
   img_w2=fix(img_w2);
   img_r2=idwt53(img_w2,level,S);
end

%------------post-processing -----------------------------------------
if lossy==1
   img_r1=128+img_r2;
else
   img_r1=128-img_r2;
end
img_r22=round(img_r1);
figure, imshow(uint8(img_r22));
title('the reconstructed image with BDC');

%--choose the original picture for comparison with the reconstructed image-

X1 = func_ReadRaw('lena512.raw', 512*512, 512, 512);
X2 = func_ReadRaw('barbara_t.raw', 512*512, 512, 512);
X3 = func_ReadRaw('goldhill_t.raw', 512*512, 512, 512);
```

```
X4 = func_ReadRaw('baboon_t.raw', 512*512, 512, 512);
X5 = func_ReadRaw('peppers_t.raw', 512*512, 512, 512);
X8=imread('lena128.pgm'); %128X128

%figure,imshow(uint8(X1));
X=X8;
X=double(X);

%--------------- calculate PSNR  ----------------------------------------
PSNR=cal_psnr(X,img_r22);
disp(sprintf('PSNR for BDC = +%5.4f dB',PSNR));  % display PSNR in dB unit
ed=0; % stop here
% end of file!
```

A.4 BDC_Pass_decoding

BDC_pass_decoding is for undoing the received bit-stream; it explains the head information for every packet, and then calls the bit-plane decoding to reconstruct the wavelet coefficients.

```
function [out,level]=BDC_pass_decoding(in);
% This method is for decoding the bit-stream from every pass.
%
% in: the received bit stream;
% out: coefficients metrix in wavelet domain;
% level: wavelet transform level


%-----------  Initialization  ----------------------------------------
% undo the main parameter
li=length(in); % get the length of in
index=1;
band=in(index:index+4); % get the level number
level=transB(band);
band=level*3+1; %overall sub-band number
index=index+5;
r=in(index:index+11); % get the size of matrix
r=transB(r);
index=index+12;
out(1:r,1:r)=0; % initiate as zero firstly
s(1:r,1:r)=2;% initiate sign metrix as 2
Rs=s;

% calculate the size of matrix
band1=(band-1)/3+1;
if band1>2
   d=2;
```

```
    for i=3:band1
        d=d+2^(i-2);
    end
else
    d=2;
end
sr=r/d; % the smallest size

lnt(1:band)=0; % represent the top layer of the block
ln(1:band)=0; % represent the layer of significant point
lnr(1:band)=0;% represent the layer of refinement point
sbs(1:band)=0; % represent the small block size for coding
bss(1:band)=0; % bs' state, after first reading the sbs, it change from 0 to 1
%for ib=1:band
LIS_m{band,1}=[]; % buffer for list of insignificant set
LIS2_m{band,1}=[]; % buffer for list of 2x2 block insignificant set
LIP_m{band,1}=[]; % buffer for insignificant pixel
LSP_m{band,1}=[]; % buffer for significant pixel
%end
%switch_nr(1:band)=1; % the received bit-stream should be new
% significant and refinement regulaly
eb=0; % represent the length of bit-stream is right

%

% process every layer from every band
while (index<li)
    if index+9>li
        break;
    end
    if index==204117 %test point
        dd=0;
    end
    bdn=in(index:index+4); % get the  band number of this layer
    bandn=transB(bdn);
    index=index+5;
    lynumber=in(index:index+3); % get the layer number
    layernumber=transB(lynumber);
    index=index+4;

    %bandn=in(index); % band number
    %index=index+1;
    %layernumber=in(index); % layer number
    %index=index+1;
    %ln(bandn)=layernumber;

    %process the bss
    if bandn>16 %test point
        dd=0;
    end
```

```
if bss(bandn)==0
   bss(bandn)=1;
end
if index>li
   break;
end
n_r=in(index); % n_r=0, new significant, 1: refine
index=index+1;
if n_r==0
   if ln(bandn)==0 % means this is the first time
      ln(bandn)=layernumber;
      lnt(bandn)=layernumber;
   else
      ln(bandn)=layernumber;
   end
else
   lnr(bandn)=layernumber;
end

% Process the length of layer
if bandn==1
   srb=sr;
   % maximum length of one layer using binary represent
   if n_r==0
      if ln(bandn)==lnt(bandn) % lnt: top layer number
         pl2=ceil(log(srb*srb*2)/log(2))+1;
      else
         %pl2=ceil(log((srb*srb-nop{bandn}(ln(bandn)+1))*2)/log(2))+1;
         if (srb*srb-nop{bandn}(ln(bandn)+1))==0
            pl2=5; % at least, there are 5 bits
            nop{bandn}(ln(bandn))=srb*srb;
         else
            pl2=ceil(log((srb*srb-nop{bandn}(ln(bandn)+1))*2)/log(2))+1;
         end
      end
   else
      if nop{bandn}(lnr(bandn)+1)==srb*srb;
        nop{bandn}(lnr(bandn))=srb*srb;
      end
      pl2=ceil(log(nop{bandn}(lnr(bandn)+1)+3)/log(2));
   end
else
   reso=floor((bandn+4)/3);
   srb=sr*2^(reso-2);
   if n_r==0
      if ln(bandn)==lnt(bandn)
         pl2=ceil(log(srb*srb*2)/log(2))+1;
      else
         %pl2=ceil(log((srb*srb-nop{bandn}(ln(bandn)+1))*2)/log(2))+1;
         if (srb*srb-nop{bandn}(ln(bandn)+1))==0
```

```
            pl2=5; % at least, there are 5 bits
            nop{bandn}(ln(bandn))=srb*srb;
          else
            pl2=ceil(log((srb*srb-nop{bandn}(ln(bandn)+1))*2)/log(2))+1;
          end
        end
      else
        if nop{bandn}(lnr(bandn)+1)==srb*srb;
          nop{bandn}(lnr(bandn))=srb*srb;
        end
        pl2=ceil(log(nop{bandn}(lnr(bandn)+1)+3)/log(2));
      end
    end
    if bandn==6 & ln(6)==1
      dd=0;
    end

    % get the the length of this layer
    if index+pl2>li
      break;
    end
    layer_len1=in(index:index+pl2-1);
    layer_len=transB(layer_len1);
    index=index+pl2;
    %layer_len=in(index); % length of bit-stream
    %index=index+1;

    % get the bit-stream
    if length(in)-index+1<layer_len
      bs=in(index:li); % get the not enough bit-stream
      eb=1;
      index=li+1;
    else
      bs=in(index:index+layer_len-1); % get the full bit-stream
      index=index+layer_len;
    end

  bs=double(bs);
  % choose the different process for bit-stream
  if bandn==1
    % extract every layer or pager of coefficients
    % ectract LL band bit-stream
    sbs1=sbs(bandn);
    bss1=bss(bandn);
    LIS=LIS_m{bandn,1};
    LIS2=LIS2_m{bandn,1};
    LIP=LIP_m{bandn,1};
    LSP=LSP_m{bandn,1};
```

```
[out,s,Rs,sbs1,bss1,nopa,cp,LIS,LIS2,LIP,LSP]=N928_ECPL_DecLL_ac(out,s,layernu
mber,sr,bs,eb,n_r,Rs,bandn,sbs1,bss1,pl2,LIS,LIS2,LIP,LSP,layer_len,pl2);
      LIS_m{bandn,1}=LIS;
      LIS2_m{bandn,1}=LIS2;
      LIP_m{bandn,1}=LIP;
      LSP_m{bandn,1}=LSP;
      sbs(bandn)=sbs1;
      bss(bandn)=bss1;
      nop{bandn}(ln(bandn))=nopa;
  else
      % coding other three bands from top level to lowest level
      srh=sr; %sr*2^(bandn-2);
      sbs1=sbs(bandn);
      bss1=bss(bandn);
      LIS=LIS_m{bandn,1};
      LIS2=LIS2_m{bandn,1};
      LIP=LIP_m{bandn,1};
      LSP=LSP_m{bandn,1};
      if bandn==14 & length(bs)==11539 %
         dd=0;
      end

[out,s,Rs,sbs1,bss1,nopa,cp,LIS,LIS2,LIP,LSP]=N928_ECPL_DecLL_ac(out,s,layernu
mber,sr,bs,eb,n_r,Rs,bandn,sbs1,bss1,pl2,LIS,LIS2,LIP,LSP,layer_len,pl2);
      LIS_m{bandn,1}=LIS;
      LIS2_m{bandn,1}=LIS2;
      LIP_m{bandn,1}=LIP;
      LSP_m{bandn,1}=LSP;
      sbs(bandn)=sbs1;
      bss(bandn)=bss1;
      nop{bandn}(ln(bandn))=nopa;
  end
end

%out=out-0.5;
% doing sign transform
[r,c]=size(out);
for i=1:r
   for j=1:c
      if s(i,j)==1
         out(i,j)=-out(i,j);
      end
   end
end
dd=0;
% end of the program
```