UNIVERSIDAD POLITÉCNICA DE VALENCIA

DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

On the design of fast and efficient wavelet image coders with reduced memory usage

A thesis submitted for the degree of Doctor en Informática

José Salvador Oliver Gil

Thesis advisor
Manuel Pérez Malumbres

Valencia (Spain), February 2006

A Silvia, y a mis padres y hermanas.

Abstract

Image compression is of great importance in multimedia systems and applications because it drastically reduces bandwidth requirements for transmission and memory requirements for storage. Although earlier standards for image compression were based on the Discrete Cosine Transform (DCT), a recently developed mathematical technique, called Discrete Wavelet Transform (DWT), has been found to be more efficient for image coding.

Despite improvements in compression efficiency, wavelet image coders significantly increase memory usage and complexity when compared with DCT-based coders. A major reason for the high memory requirements is that the usual algorithm to compute the wavelet transform requires the entire image to be in memory. Although some proposals reduce the memory usage, they present problems that hinder their implementation. In addition, some wavelet image coders, like SPIHT (which has become a benchmark for wavelet coding), always need to hold the entire image in memory.

Regarding the complexity of the coders, SPIHT can be considered quite complex because it performs bit-plane coding with multiple image scans. The wavelet-based JPEG 2000 standard is still more complex because it improves coding efficiency through time-consuming methods, such as an iterative optimization algorithm based on the Lagrange multiplier method, and high-order context modeling.

In this thesis, we aim to reduce memory usage and complexity in wavelet-based image coding, while preserving compression efficiency. To this end, a run-length encoder and a tree-based wavelet encoder are proposed. In addition, a new algorithm to efficiently compute the wavelet transform is presented. This algorithm achieves low memory consumption by using line-by-line processing, and it employs recursion to automatically place the order in

which the wavelet transform is computed, solving some synchronization problems that have not been tackled by previous proposals. The proposed encoders perform in-place processing so that no extra memory is required for the coding process. Furthermore, time-consuming methods (such as iterative algorithms, high-order modeling and bit-plane coding) are avoided to reduce complexity, and we show the importance of grouping coefficients with tree structures as a method to reduce complexity.

Resumen

La compresión de imágenes es de vital importancia en sistemas y aplicaciones multimedia, ya que reduce drásticamente tanto el ancho de banda necesario para transmitir imágenes como la cantidad de memoria que hace falta para almacenarlas. Aunque los primeros estándares de compresión de imagen estaban basados en la transformada discreta del coseno, recientemente ha surgido una nueva herramienta matemática denominada transformada discreta wavelet que se considera más eficiente para la compresión de imágenes.

A pesar de las mejoras en eficiencia, los compresores de imagen basados en esta transformada necesitan mucha más memoria e incrementan considerablemente su complejidad temporal si los comparamos con aquellos basados en la transformada discreta del coseno. Una razón fundamental que provoca estos elevados requerimientos de memoria es que el algoritmo empleado comúnmente para calcular la transformada wavelet necesita que la imagen entera esté en memoria. Aunque existen algunas propuestas que reducen el uso de memoria, éstas presentan varios problemas que dificultan su implementación. Además, determinados codificadores wavelet, como SPIHT (que se ha convertido en un referente para la codificación de imagen usando wavelets), también necesitan mantener toda la imagen en memoria para realizar el posterior proceso de codificación.

Respecto a la complejidad temporal de los codificadores, SPIHT es bastante complejo debido al procesamiento por capas de bits con múltiples pasadas de la imagen que realiza. El estándar JPEG 2000, también basado en la transformada wavelet, es todavía más complejo porque mejora la compresión por medio de costosas técnicas, como por ejemplo un algoritmo iterativo de optimización basado en el método de multiplicadores de Lagrange, y el uso de un modelado de contextos de alto orden.

RESUMEN

En esta tesis, pretendemos reducir el uso de memoria y la complejidad en la codificación de imagen basada en wavelets, sin afectar por ello sus prestaciones en compresión. Con este objetivo, proponemos un codificador wavelet basado en codificación *run-length* y otro basado en árboles. Además, presentamos un nuevo algoritmo para calcular la transformada wavelet de forma eficiente. Este algoritmo reduce el uso de memoria por medio de un procesamiento línea a línea, y usa recursividad para establecer de forma automática el orden en el que la transformada se calcula, resolviendo de esta manera algunos problemas de sincronismo que no habían sido abordados en otras propuestas previas. Por otra parte, los codificadores presentados en esta tesis realizan un procesamiento directo de los coeficientes sin necesidad de memoria adicional o complejas estructuras de datos. Además, se evita el uso de métodos costosos (como por ejemplo, algoritmos iterativos, modelado de contextos de alto orden o codificación por capas de bits) para reducir así la complejidad temporal. Finalmente, con el mismo objetivo, también se muestra la importancia de agrupar coeficientes usando estructuras de árboles.

Resum

La compressió d'imatges és de vital importància en sistemes i aplicacions multimèdia, ja que reduïx dràsticament tant l'ample de banda necessari per a transmetre imatges com la quantitat de memòria que fa falta per a emmagatzemar-les. Encara que els primers estàndards de compressió d'imatge estaven basats en la transformada discreta del cosinus, recentment ha sorgit una nova ferramenta matemàtica denominada transformada discreta wavelet que es considera més eficient per a la compressió d'imatges.

A pesar de les millores en eficiència, els compressors d'imatge basats en wavelets necessiten molta més memòria i incrementen considerablement la seua complexitat temporal si els comparem amb aquells basats en la transformada discreta del cosinus. Una raó fonamental que provoca els elevats requeriments de memòria és que l'algoritme emprat comunament per a calcular la transformada wavelet necessita que tota la imatge estiga en memòria. Encara que hi ha algunes propostes que reduïxen l'ús de memòria, estes presenten diversos problemes que dificulten la seua implementació. A més, determinats codificadors wavelet, com SPIHT (que s'ha convertit en un referent per a la codificació d'imatge usant wavelets), també necessiten mantindre tota la imatge en memòria per a realitzar el posterior procés de codificació.

Respecte a la complexitat temporal dels codificadors, SPIHT és prou complex a causa del processament per capes de bits amb múltiples passades de la imatge que realitza. L'estàndard JPEG 2000, també basat en la transformada wavelet, és encara més complex perquè millora la compressió per mitjà de costoses tècniques, com per exemple un algoritme iteratiu d'optimització basat en el mètode de multiplicadors de Lagrange, i l'ús d'un modelatge de contextos d'alt orde.

En la present tesi, pretenem reduir l'ús de memòria i la complexitat en la codificació d'imatge basada en wavelets, sense afectar per això les seues prestacions en compressió. Considerant este objectiu, proposem un codificador wavelet basat en codificació *run-length* i un altre basat en arbres. A més, presentem un nou algoritme per a calcular la transformada wavelet de forma eficient. Este algoritme reduïx l'ús de memòria per mitjà d'un processament línia a línia, i usa recursivitat per a establir de forma automàtica l'orde en què la transformada es calcula, resolent d'esta manera alguns problemes de sincronisme que no havien sigut abordats en altres propostes prèvies. D'altra banda, els codificadors presentats en la tesi realitzen un processament directe dels coeficients sense necessitar memòria addicional o complexes estructures de dades. A més, s'evita l'ús de mètodes costosos (com per exemple, algoritmes iteratius, modelatge de contextos d'alt orde o codificació per capes de bits) per a reduir així la complexitat temporal. Finalment, també es mostra la importància d'agrupar coeficients usant estructures d'arbres per tal de reduir la seua complexitat.

Agradecimientos

Muchas y variadas son las circunstancias que han conducido a la realización de esta tesis doctoral. ¿Qué duda cabe que la vida es altamente circunstancial? Seguramente, el sólo lanzamiento de una moneda puede determinar tantas cosas en nuestro Universo, y quién sabe si en alguna que otra paracaja de Farnsworth. Sin duda, aquel momento en el que decidí realizar el PFC con Mels es uno de esos momentos cruciales. Tú me introdujiste en el mundo de la investigación, y me animaste en los momentos más delicados. Por eso y por más, gracias Mels y buena suerte en tu nueva andanza.

No sería justo si no agradeciera a los grupos de investigación que me han dado la oportunidad de desarrollar mi formación como investigador, al Grupo de Arquitecturas Paralelas en un comienzo, y al Grupo de Redes de Computadores en la actualidad. A ellos, a sus integrantes, gracias por todo.

A nivel personal, casi sentimental, los agradecimientos son tantos que casi sería injusto nombrar a unos y olvidar a otros. La motivación para escribir una tesis hubiera sido imposible conseguirla sin todos vosotros, sin mis amigos y mi familia, que tanto me ha dado. A Elena que me ayudó con el lifting, y a Silvia, por ser como es y por apoyarme y animarme siempre durante la tesis. A todos vosotros, gracias.

Contents

List of figures

LIST OF FIGURES

List of tables

List of algorithms

List of abbreviations

bpp: bits per pixel
CALIC: Context Adaptive Lossless Image Compression
CD: Compact disc
CDF: Cohen/Daubechies/Feauveau
CWT: Continuous Wavelet Transform
DC: Direct Current
DCT: Discrete Cosine Transform
DPCM: Differential Pulse Code Modulation
DSP: Digital Signal Processor
DST: Discrete Sine Transform
DVB: Digital Video Broadcasting
DVD: Digital Versatile Disc
DWT: Discrete Wavelet Transform
EBCOT: Embedded Block Coding with Optimized Truncation
EOL: End Of Line
EZBC: Embedded Zero Block Coding
EZW: Embedded Zero-tree Wavelet
FWT: Forward Wavelet Transform
GIF: Graphic Interchange Format
GIS: Geographic Information System
HVS: Human Visual System
iid: independent and identically distributed
ITU: International Telecommunications Union

LIST OF ABBREVIATIONS

ITU-R: ITU Radiocommunication Sector
ITU-T: ITU Telecommunication Sector
IWT: Inverse Wavelet Transform
JBIG: Joint Bi-level Image Processing Group
JPEG: Joint Photographic Experts Group
KLT: Karhunen-Loève Transform
LIP: List of Insignificant Pixels
LIS: List of Insignificant Sets
LOCO-I: LOw COmplexity LOssless COmpression for Image
LSB: Least Significant Bit
LSP: List of Significant Pixels
LTW: Lower Tree Wavelet
LZC: Layered Zero Coding
MPEG: Moving Pictures Experts Group
MSB: Most Significant Bit
MSE: Mean Square Error
NTSC: National Television System Committee
PAL: Phase Alternate Line
PCM: Pulse Code Modulation
PCRD: Post-Compression Rate Distortion
PDA: Personal Digital Assistant
pdf: probability distribution function
pixel: picture element
PNG: Portable Networks Graphic
ppm: prediction by partial matching
PROGRES: PROGRESsive resolution decomposition
PSNR: Peak to Signal Noise Ration
RCT: Reversible Color Transform
RGB: Red-Green-Blue
RLE: Run-Length Encoding
RLW: Run-Length Wavelet
ROI: Region Of Interest
SBHP: Subband-Block Hierarchical Partitioning
SECAM: Sequential Couleur Avec Mémoire

LIST OF ABBREVIATIONS

SFQ: Space-Frequency Quantization
SIMD: Single Instruction, Multiple Data
SNR: Signal to Noise Ratio
SPECK: Set Partitioning Embedded block
SPIHT: Set Partitioning In Hierarchical Trees
VCEG: Video Coding Expert Group
VGA: Video Graphics Array
VM: Verification Model
VQEG: Video Quality Experts Group
xDSL: Digital Subscriber Line

Preface

Motivation

During the last decade, several image compression schemes emerged in order to overcome the known limitations of block-based algorithms that use the Discrete Cosine Transform (DCT). These limitations include blocking artifacts and poor coding efficiency, mainly at moderate to low bitrates. Some of these alternative proposals were based on more complex techniques, like vector quantization and fractal image coding, whereas others successfully proposed the use of a different and more suitable mathematical transform, the Discrete Wavelet Transform (DWT).

The discrete wavelet transform is a new mathematical tool that has aroused great interest in the field of image processing due to its nice features. Some of these characteristics are the following: 1) it allows image multiresolution representation in a natural way, because more wavelet subbands are used to progressively enlarge the low frequency subbands; 2) we can analyze the wavelet coefficients in both space and frequency domains, thus the interpretation of the coefficients is not constrained to its frequency behavior as in Fourier; and 3) for natural images, the DWT achieves high compactness of energy in the lower frequency subbands, which is extremely useful in applications such as image compression. Therefore, the introduction of the DWT made it possible to improve some specific applications of image processing by replacing the existing tools with this new mathematical transform. Thus, while the popular JPEG standard for image compression uses the DCT, the new JPEG 2000 standard proposes the use of the wavelet transform, with better rate/distortion (R/D) performance, and which avoids blocking artifacts because an image is not separately

transformed and quantized block-by-block, but the wavelet transform is applied to the entire image.

Unfortunately, despite the benefits that wavelet-based image coding entails, other problems are introduced in these encoders, basically they are typically implemented with memory-intensive and time-consuming algorithms, and thereby system requirements are significantly higher than in other earlier image encoders like JPEG. These higher requirements represent a serious limitation when implementing multimedia applications like image compression in memory-constrained devices with relatively little computational power, such as digital cameras, mobile phones, PDAs and embedded devices. Actually, in many applications like videoconferencing (implemented with image coding if only intraframe redundancy is removed) features like low complexity and high symmetry are more important than R/D performance.

All the wavelet image coders, and in general all the transform-based encoders, consist of two main stages. During the first one, an image is transformed from spatial domain to another domain, in the case of wavelet transform a combined spatial-frequency domain called wavelet domain. In the second pass, the wavelet coefficients resulting from the transform domain are quantized and encoded in an efficient way to achieve high compression efficiency and other features.

The higher execution time and memory requirements of the wavelet coders are caused in both stages, due to the following reasons:

(1) Computation of the wavelet transform. In the usual DWT, the image decomposition is computed by means of convolution filtering and so, complexity rises as the filter length increases. Moreover, in the regular DWT computation, an image is transformed first row by row and then column by column at every decomposition level, and hence it must be held entirely in memory. These problems are not as noticeable in other image transforms as in the DWT. For example, when the DCT is used for image compression, it is applied in small block sizes, and thus a large amount of memory is not specifically needed for the transform process.

(2) Coding of the coefficients. Great efforts have been made in this stage to improve compression efficiency, achieving in this way a reduction in the bandwidth or amount of memory needed to transmit or store a compressed image. Unfortunately, many of these coding optimizations involve high complexity, requiring faster and more expensive

processors. For example, the JPEG 2000 standard uses a large number of contexts and an iterative time-consuming optimization algorithm to improve coding efficiency. Furthermore, many times, wavelet image coders have features that are not always needed, but which make them CPU and memory intensive. This way, bit-plane coding employed in many encoders (like EZW and SPIHT) allows SNR scalability with an embedded bitstream, but it results in slow coding since an image is scanned several times, focusing on a different bit-plane in each pass, which in addition causes a high cache miss rate.

In this thesis, we tackle the problem of designing new wavelet-based image encoders with lower complexity and memory usage, but preserving compression efficiency. In order to reduce memory consumption, the need to store the entire image in memory should be removed. In addition, time-consuming techniques (such as rate/distortion optimization algorithms, high-order context modeling, and bit plane coding) should be avoided to minimize complexity.

Objectives

As mentioned in the motivation section, the major goal of this thesis is the design of fast wavelet image encoders with reduced memory usage and without loss of compression efficiency. To this end, specific objectives of this thesis can be detailed as follows:

1. In-depth study of the existing algorithms to compute the wavelet transform, focusing on techniques to reduce complexity and memory requirements (like the lifting scheme and the line-based approach). Furthermore, their main drawbacks should be analyzed.
2. Design of simple algorithms to compute the wavelet transform with reduced memory usage, overcoming the synchronization problems of the algorithms found in the literature. In order to minimize execution time, the proposed algorithms should take the lifting approach.
3. In-depth study of the main wavelet image coders, analyzing their complexity and focusing on the fastest non-embedded encoders, since in the thesis we will propose non-embedded algorithms to reduce complexity.
4. Design of fast wavelet coders that avoid the complex techniques found in the main wavelet coders, although preserving state-of-the-art coding efficiency as far as possible (only for very fast versions, we will admit moderate loss of efficiency).

These encoders should be able to work in lossy and lossless modes.

5. Tuning the encoder parameters, mainly to improve rate/distortion performance.
6. A study of the proposed algorithms to compute the wavelet transform with low memory requirements applied to the new fast encoders. We should tackle the problems that arise when combining both techniques, and study the performance of the overall encoder.
7. We should develop efficient implementations to validate the models and algorithms proposed in the thesis, and to be able to compare them with other wavelet coders using real implementations (all of them written under the same conditions as far as possible).

Thesis overview

The thesis structure pursues the objectives presented in the previous section. In Chapter 1, a general introduction to image coding is given. Some general methods such as entropy coding (both Huffman and adaptive arithmetic coding) and run-length coding are described because they are used later in our proposals. Transform coding is also introduced in this chapter, and the structure of a general transform-based image encoder is shown. This structure will serve as a reference in the rest of the thesis. Finally, some guides to measure the performance of the image coder under study in this thesis are presented (namely, to measure coding efficiency, complexity and memory requirements).

In Chapter 2, we introduce the wavelet transform as a new mathematical tool to perform a frequency-spatial analysis of a signal. Then, we focus on the two-dimensional wavelet transform computation, since it is the first stage of a wavelet-based encoder. We first present the classical computation with a filter bank, showing the main filter properties desired for image coding, and then we describe the lifting scheme as an alternative and more efficient method to compute the DWT. Some of the most important wavelet filter-banks for image coding are also given in this chapter. Afterwards, Chapter 3 tackles the problem of memory reduction in the two-dimensional wavelet transform computation. A line-based approach is presented as a method to compute the DWT without the need to store the entire image in memory, but we show the difficulty to derive a practical algorithm due to the existing synchronization problems among buffers. Thus, we propose a general recursive algorithm to compute the wavelet transform, also line-by-line, which automatically solves the

synchronization problems of the earlier line-based approaches. This general algorithm is further developed to allow filter-bank and lifting based implementations, and regular and reversible transforms (with floating and integer implementations).

Once the wavelet transform is applied to an image, the next stage in transform coding is coefficient quantization and coding. In Chapter 4, we introduce the main wavelet image coders found in the literature, including a complexity analysis. We group them into two main categories: tree-based and block-based, depending on the structure used to group coefficients. Some of the encoders presented in this chapter are the well-known EZW, SPIHT, SPECK/SBHP algorithms and EBCOT (on which JPEG 2000 is heavily based). Some other non-embedded encoders are also described, in particular SFQ, PROGRES and non-embedded versions of SPITH and SPECK/SBHP. Encoders based on run-length coding and high-order context modeling are also surveyed. Finally, in this chapter we show the importance of tuning the encoder parameters in an image encoder by implementing the EZW algorithm, and then analyzing the encoder response when varying several parameters.

In Chapters 5 and 6, we propose various wavelet image coders to overcome the complexity problems addressed in the previous proposals. In these coders, we should avoid bit-plane coding, iterative optimization algorithms, and high-order context modeling, because all these methods cause an increase in complexity. First, a very simple encoder is proposed. This algorithm encodes one symbol for each coefficient by using arithmetic coding, and therefore it is still quite complex at low bitrates, because too many symbols are encoded. Thus, a second encoder is presented, which uses run-length coding to group large number of coefficients. Although this encoder is faster, coding efficiency can be improved if inter-subband redundancy is considered. With this idea in mind, a tree-based encoder is proposed in Chapter 6, which achieves better compression results with still lower complexity. All the proposed encoders are tuned to maximize coding efficiency. A comparison with other wavelet encoders and among them is always included using numerical results and real implementations.

In Chapter 7, we present an overall encoder that includes the efficient wavelet transform proposed in Chapter 3, and the run-length or tree-based encoders of Chapters 5 and 6. We compare this overall encoder with other reference wavelet encoders (mainly SPITH and JPEG 2000). Afterwards, a very fast Huffman-based variation of the tree-based encoder of Chapter 6 is presented. This encoder entails a small loss of compression efficiency, while

execution time is reduced by about three times. Then, a lossless analysis of the proposed encoders is carried out.

Finally, some publications resulting from the research of the thesis, future works and the conclusions of the thesis (also detailed in each chapter) are summarized in Chapter 8.

All this time the guard was looking at her,

first through a telescope, then through a microscope,

and then through an opera glass

Lewis Carroll, *Through the looking glass*

Chapter 1

Introduction to image coding

1.1 Data compression for a real world

We live in a real world, with real computers and real data networks, and therefore with a large number of practical limitations. One of these limitations is the need to store data in memory. Since computer's physical memory is finite, mass storage devices are used as a low-cost swap memory, at the expense of higher latency and lower throughput. Even so, all these devices always have limited storage capacity. A similar restriction is present in computer networks, which have finite bandwidth. Therefore, if we find a method to reduce the required size of a document, we will be able to exploit the available resources (amount of memory or bandwidth) in a better way. The tool that performs the reduction of memory needed to store a document is called data compressor, while a compression technique is the process that is carried out by a data compressor to encode information in an efficient way. Once data is compressed, it can be returned to its original representation by means of a reconstruction algorithm executed by a decompressor.

Although data compression is useful to improve the performance of network and storage technologies, one might wonder if it is really worth the use of compression techniques due to the capacity improvements in storage devices (such as DVD disks and larger and larger hard drives) and transmission lines (due to the use of optical fiber lines and xDSL technology). The answer to this question is yes, because as both technologies improve their performance,

their demand is increasing a lot; mainly in multimedia networks and applications, which need a large amount of memory to represent multimedia data. Thus, the use of compression techniques allows us to take advantage of the available resources in a better way, achieving better performance at lower cost. For example, a regular audio CD is able to store about 15-20 songs that are "raw coded" (uncompressed) to simplify the hardware design of stand-alone players. However, if we use a software player, audio compression can be introduced provided that the computer running the software is able to decompress it in real time. This way, hundred of songs can be stored on the same CD with the same perceptual audio quality. As time went by, when a hardware design for audio decompression is feasible at relatively low cost, new hardware players with decompression support can be introduced.

Hence, there is a trade-off between the benefits of data compression and some drawbacks that it introduces, such as higher latency and increase in memory and hardware requirements. Each system designer should evaluate if compression is necessary, and determine the impact of compression on the system to decide if it can be accepted.

1.2 Digital image coding

During the last decades, digital processing has gained popularity since information can be processed in a more complex and complete way. Moreover, digital data can be transmitted over long distances without any degradation.

One type of data that has been tremendously impacted by digital processing is multimedia data (mainly audio, image and video). During the 80s and 90s, there was significant effort from the research community focused on digital image and video processing, and now digital technologies have reached the mass market. Digital image and video processing includes daily applications such as transmission of images on the Internet, capture and storage of images in digital cameras, and digital video broadcasting (DVB) and storage (DVD).

In Section 1.1, we addressed the need for data compression. This need is further highlighted in multimedia data, which require huge amount of bytes to be represented, but having high correlation that can be removed to reduce storage requirements. In order to illustrate this, let us take the example of a 1-Megapixel digital camera. If no compression is used, a color picture takes 3 Mbytes to be stored, while every picture takes about 250 Kbytes if a compression algorithm is applied (of course the size may vary depending on the

algorithm used, and the image quality and contents). This way, with compression, the same memory card can store about twelve times more pictures than if no compression is applied. Moreover, when images are processed as a whole, they are handled faster if they are compressed. For example, images are downloaded from a digital camera to a computer much faster if they are encoded with a compression algorithm.

1.2.1 Image representation and color spaces

For the most part, digital images are represented as a set of pixel values, *P*. The simplest case of digital image is the monochrome one, in which a matrix of pixels of size $\{image\ width \times image\ height\}$ describes a grayscale image. This matrix is usually implemented as a two-dimensional array, and each element in the array (pixel) is represented using *n* bits, which is called image depth. In monochrome images, *n* is typically equal to eight because the eye does not perceive higher resolution according to the human visual system (HVS) model. In this array, the value of a pixel $p_{i,j} \in P$ indicates the luminance or brightness of the sample that is located at the position (i, j), so that the higher $p_{i,j}$ is, the brighter that sample is (a value of 0 is used for pure black color while the highest value $(2^n - 1)$ shows a white pixel)

For color images, several planes per pixel are required. Natural color images are often represented using three components, and implemented with an array per component. A usual color space is the RGB (red/green/blue), in which each pixel is described by the three primary colors that compose that pixel. Most display and capture devices work internally using the RGB color space.

Since the R/G/B planes are highly correlated in natural images, a color transform is usually applied before compressing an image, so that the new color space has less redundancy among components. Moreover, the HVS is less sensitive to color detail than to brightness detail, and thus if, the employed color space is able to handle brightness and color information separately, we can apply higher resolution on the luminance samples than on the color ones. Actually, image compression standards usually sub-sample the chrominance[1] arrays by a factor of two in each dimension, which is known as 4:2:0 sub-sampling format.

Some color spaces widely used due to their lack of correlation among components are YUV, YIQ and YCbCr. The former is employed in analog TV standards, such as NTSC, PAL

[1] Chrominance is the difference between a color component and the luminance

and SECAM composite TV. A related color space that is optionally used by the NTSC standard is YIQ. The YCbCr color space was developed as part of the ITU-R BT.601 [ITU82] and is of major importance since it is used in most digital video and image standards. Equations for conversion from RGB to YUV, YIQ and YCbCr color spaces can be found in [RAO96].

More recently, some other color spaces have been proposed, which not only have a floating-point implementation but also an integer implementation. In JPEG 2000, it is proposed a reversible color transform (RCT) that is intended to an integer-to-integer transform for lossless compression (see Section 1.2.2). The H.264 standard (high profile) introduces another integer color transform called YCoCr [MAL03]. Instead of the classical blue and red chrominance planes, this color space extracts the orange and green color information from the image, along with the luminance samples. The Forward and Inverse transformations are computed as it is shown in the flowgraphs of Figure 1.1. Besides reversibility in integer arithmetic, YCoCb shows minimal increase in dynamic range and higher coding gain than the aforementioned color spaces, which is particularly interesting in image compression.

A different color space is CIE Lab, which specifies colors in terms of human perception. CIE Lab is based on a mathematical model instead of a certain display or capture device, and hence it is device independent. On the contrary, every concrete device has a particular RGB color space, which is device dependent and thus, it has to be transformed into a device independent representation for general independent use. Other space color, such as CMY and CMYK, are more suitable for printing.

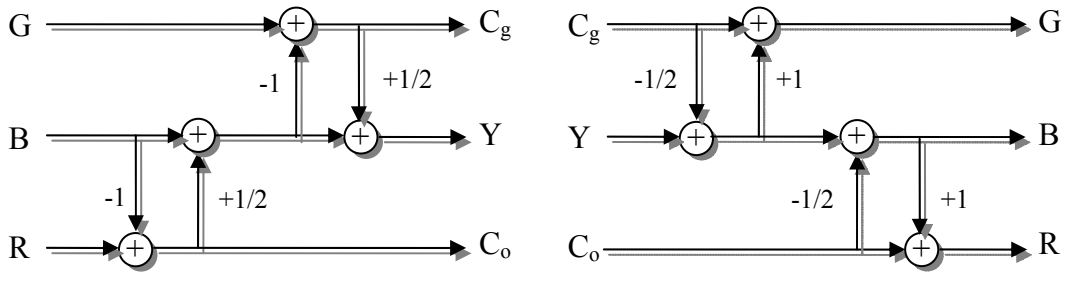


Fig. 1.1: Forward and Inverse YCoCg color transformation. In the forward transform (left), the green, blue and red components are input so as to compute the luminance and color components (green and orange) with additions and divisions. The inverse transform (right) is computed scanning the graph in the reverse order, and changing the sign of the operations.

For the rest of this thesis, we will only deal with grayscale image processing. Generalization to color image processing can be performed simply by handling every chrominance component independently, in the same way as the luminance array. Many image and video compression standards, such as JPEG 2000 and H.264, also take this approach.

1.2.2 Lossless and lossy image compression

In lossless image compression, the decoded samples (P') are exactly the same as those that were encoded (P). For this reason, we consider that there is no loss of data. However, a compression algorithm can slightly modify a source image in order to achieve higher compression ratios, but trying to keep the perceived quality unaltered according to the HVS. This is the case of lossy image compression, in which the equality $P' = P$ is not usually met.

Most lossless image coders are based on entropy coding (see Section 1.3) with various contexts and predictive techniques. Predictive coding schemes try to predict each sample from the samples that have been previously encoded, which are available to both encoder and decoder. In image compression, prediction is usually performed from nearby pixels. Once a prediction has been calculated, the residual pixel is encoded as the error committed by this prediction. This way, the better a prediction is, the lower it will be the entropy of the residual pixels. The CALIC scheme [WU96] follows this approach, becoming one of the most efficient lossless image coders in terms of compression performance. A simplification of CALIC was adopted as the JPEG-LS standard, which replaced the lossless mode of the original JPEG standard. This simplified version of CALIC is called LOCO-I [WEI00], and its performance is close to CALIC with lower complexity. Other lossless image encoders are PNG (proposed as a royalty-free alternative to GIF) and JBIG (intended to bi-level image coding and used in fax transmission).

Medical imaging is an example of application in which lossless compression is required, since all the image details must be preserved so that medical analysis is not hindered. Another application of lossless coding is image editing. In this type of application, if lossy compression is employed, accumulative errors from successive editions may seriously damage the final image quality. However, lossless compression yields poorer compression ratios when compared with lossy, and hence the former is not as frequently used as the latter.

Various approaches to lossy coding have been taken in the literature. In vector quantization [GER92], an image is represented by regular patterns of finer detail that are

stored in a codebook, which is shared by both encoder and decoder. A similar scheme is fractal coding [BAR93], in which images use themselves as their codebook. Unfortunately, both methods are time intensive, due to the search for the codebook, and might produce blocking artifacts, i.e., the edges diving two contiguous blocks could be perceptible. A more successful approach to lossy compression has been achieved by transform methods.

1.3 Background on data compression

Data compression is not as recent as one might think. An example of efficient data coding can be found in the 19th century, when Samuel Morse developed a method to send letters by telegraph systems. Letters are encoded with dots and dashes so that the letters that occur more frequently are assigned shorter sequences, and thus the average time required to send a message is reduced.

1.3.1 Entropy coding

An idea similar to Morse code is used in more modern techniques based on entropy coding. In this context, when we talk about coding, we refer to assigning bits to represent a symbol or a group of symbols. In general, if X is a discrete variable representing any possible symbol from an alphabet A, a symbol $s \in A$ can be encoded using a coding function $C(X)$ that maps s with a finite and ordered sequence of binary symbols (bits). This sequence of bits is called codeword, and the table that maps each symbol into its codeword is called codebook. Of course, real applications usually encode more than a symbol from A, and thus, when a sequence of L symbols $S = \{s_1, s_2, \ldots s_L\} : s_1, s_2, \ldots s_L \in A$ is encoded, the main goal of data compression is to achieve the shortest length for the final bitstream, i.e., to minimize $|C'(S)|$, where $C'(\{X_1, X_2, \ldots, X_L\})$ is the coding function for the whole sequence of symbols. A possible non-optimal solution to code the sequence S is to choose a codebook that minimizes $|C'(S)| = |C(X_1)| + |C(X_2)| + \ldots + |C(X_L)|$ for all possible coding assignments. However, better solutions can be achieved if we do not focus on individual symbols but in groups of them. We will see this optimal solution later.

Besides compact representation, the coding process must be reversible, guaranteeing that a decoding process can reconstruct exactly the initial symbols in the same order as they were encoded.

1.3.2 Definition of entropy

The limits of entropy coding are bounded by the concept of entropy $H(S)$ [SHA48], which denotes the minimum average number of bits needed to represent every symbol of a sequence (i.e., the minimum bits per symbol). If we consider that S is a general source of symbols from a finite alphabet A that generates sequences with the form $\{X_1, X_2, \ldots, X_L\}$, where each X_i is a discrete variable from A, the entropy for this general source is given by:

$$H(S) = \lim_{n\to\infty} \frac{1}{n} G_n \qquad (1.1)$$

where

$$G_n = -\sum_{i_1 \in A} \sum_{i_2 \in A} \cdots \sum_{i_n \in A} P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n) \log_2 P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n) \quad (1.2)$$

A less general definition of entropy is the first-order entropy H. It is defined on a single discrete random variable $X \in A$ as follows:

$$H = -\sum_{i \in A} P(X = i) \log_2 P(X = i) \qquad (1.3)$$

First-order entropy has the advantage of ease of computation, although it is not equal to the entropy of the source unless each element in the sequence is independent and identically distributed (iid), as it is proved in [SAY00]. If there is certain dependence among symbols in S, we can compute higher order entropies to approach the general definition of entropy.

Although entropy shows the lowest number of bits needed to represent a symbol using entropy coding, we still have to solve the problem of how to assign bits to a set of symbols. In the next subsections, we will introduce some entropy coders to carry out this association.

1.3.3 Huffman coding

The Huffman coding algorithm [HUF52] was proposed in 1952 as a way to map each alphabet symbol into a codeword. It generates the shortest bitstream that can be created if each symbol in the alphabet is directly associated with a sequence of bits. For this type of entropy coding, we can say that Huffman coding is optimal (see the proof in [FAN61]). In order to achieve optimality, symbols are represented with a codeword whose length is inversely proportional to the symbol probability. Assignment of codeword is carried out by building a binary tree from leaves to the root, being the leaves the symbols in the alphabet. Least probable symbols are joined to form a new node with higher probability, and each link

add one bit to the codeword that represents the involved symbols. A more detailed description and deep study of the Huffman coding algorithm and its variants can be easily found in the literature [LEL87] [STO88] [SAY00].

The entropy H bounds the average codeword length l generated by the Huffman algorithm. The bad news is that the average codeword length can never be below the entropy value, but the good news is that optimality of Huffman ensure that the average codeword length is lower than one bit per symbol above entropy (see the proof in [COV91]), in other words:

$$H \le l < H + 1 \qquad (1.4)$$

When the number of symbols in the alphabet is large, and their probability is about equally distributed, entropy is much larger than one bit per symbol, and Huffman coding is very efficient. However, if we have very few symbols and entropy is less than one, Huffman performance decreases drastically. The reason is that the theoretical optimum number of bits needed to represent a symbol is not usually an integer but a fraction, and Huffman coding only can map symbols with an integer number of bits. A way to overcome this problem is to group more than one symbol together, mapping in this way a block of symbols to a single codeword, so that $|C'(S)| = |C(X = X_1 \dots X_L)| < |C(X_1)| + \dots + |C(X_L)|$. If we block N symbols together, the bounds are tighter, and they are given by [COV91]:

$$H \le l < H + \frac{1}{N} \qquad (1.5)$$

It is easy to see that as the block size N is increased, the average codeword length tends to the entropy of the source. However, Huffman coding does not work well grouping symbols, mainly because the alphabet size becomes impracticable, and the decoding process is extremely inefficient.

1.3.4 Arithmetic coding

A more efficient entropy coding algorithm for low-entropy sources is arithmetic coding. In this technique, the whole source sequence $S = \{s_1, s_2, \dots s_L\} : s_1, s_2, \dots s_L \in A$ is mapped into only one codeword, which is associated to the probability of the sequence $P(S) = P(S_1)P(S_2)\dots P(S_L)$. The idea in arithmetic coding is that, for all the sequences S of length L, the greater $P(S)$, the shorter the codeword. This code assignment was developed by Pasco [PAS76] and Rissanen [RIS76] [RIS79] based on the early work of Shannon [SHA48].

A full explanation of arithmetic coding can be found in more recent literature [BEL90] [SAY00] [GHA03].

As it is proved in [SAY00], the average bits per symbol (l) resulting from encoding L symbols using this method is bounded by the expression:

$$H \leq l < H + \frac{2}{L} \qquad (1.6)$$

Although Huffman with blocks of L symbols may seem more efficient than arithmetic coding, we have to take into account that grouping symbols cannot be easily implemented with Huffman, most of all for large values of L. With an alphabet size of k, if L symbols are grouped, the size of the Huffman codebook is k^L, which becomes unfeasible for large L values. On the contrary, arithmetic coding does not require the construction of a codebook. Thereby, it can start generating bits of the final codeword since the first symbols of the sequence are introduced. Hence, we can encode a sequence as large as we wish, being asymptotically optimal for stationary sources (i.e., sources that are constant in their statistical parameters). To sum up, as the number of encoded symbols L goes to infinity, the average bits per symbol l approaches the entropy of the source, with no additional complexity.

Despite the advantages of arithmetic coding, the original Huffman algorithm is useful in implementations where fast processing is necessary and coding efficiency is not a major issue, because it is less complex than arithmetic coding when used with single symbols.

1.3.5 Adaptive arithmetic coding

Besides the superior performance of arithmetic coding, one of its main advantages is its capability to adapt its probability model, $P(x_i)\ \forall x_i \in A$, directly from the symbols that are input. This feature is very interesting in non-stationary sources, such as natural images, which usually change their statistics according to the portion of image being encoded. Furthermore, when the encoding process begins, the probability model of the source sequence can be unknown, and it can be built while symbols are encoded. An alternative method can be performed with a two-pass algorithm, in which the statistics are collected in the first pass, and the source is encoded during the second one. Adaptive coding can also be implemented for Huffman [FAL73] [GAL78], but with higher complexity.

A widely used C language implementation of an adaptive arithmetic encoder is described in [WIT87]. In this proposal, the probability model is estimated from a frequency count table

$f(x_i)$ that holds the number of times that each symbol has been encoded; that is to say, $f(x_i)$ is a dynamic histogram. This way, the probability $P(x_i)$ of occurring a symbol x_i from an alphabet *A* is calculated as:

$$P(x_i) = \frac{f(x_i)}{\sum_{x_n \in A} f(x_n)} \qquad (1.7)$$

where the term $\sum_{x_n \in A} f(x_n)$ is known as *cumulative frequency count*.

At the beginning of the coding process, since we do not have information about the symbol probabilities, the whole frequency count array is initialized to one, so that all the symbols are equally probable. Every time that a symbol x_i is encoded, $f(x_i)$ is increased. Since real implementations use finite-range variables, the frequency count cannot grow indefinitely. In order to avoid overflow, all entries in the array are dived by two when the cumulative frequency count reaches a threshold value. In [WIT87], the authors suggest a threshold value of $2^{14} - 1$ to operate with two-byte arithmetic precision, and they call this threshold *Max_frequency*.

1.3.6 Models for data compression

Beyond the limits established by the definition of entropy, we can extract information about any additional knowledge from the source, and describe it in the form of a model. In this way, we can use a model to modify the source sequence into another sequence with lower entropy. In a way, we can consider the probabilities used in entropy coding as a model defining the existing redundancy in the source. With adaptive coding, this model changes to adapt to local features. Higher order models not only consider the probability of a single symbol but also its correlation with other previously encoded symbols (in other words, with the context). For example, in adaptive arithmetic coding for image compression, we can use various frequency count tables depending on the value of the already-encoded neighbor samples, adapting the model to the context in order to improve compression performance.

In general, we can assume that the better a model is, the higher compression we achieve.

1.3.7 Differential encoding, run length encoding and other compression techniques

As an example of model, consider the following monotonic sequence

10 12 13 14 16 18

Clearly, we have more information about this sequence than its mere probability function. If we process these symbols as integer values, it is easy to see that there is high correlation among them. Hence, we can apply the model described by $x_i = x_{i-1} + r_i$, where the residual is given by $r_i = x_i - x_{i-1}$, so that if we encode the resulting residual sequence (assuming that $x_{-1} = 0$)

10 2 1 1 2 2

the entropy of this new sequence is lower than the original one. Actually, any coding scheme that uses this type of model is known as differential encoding. Some examples of this type of coding are DPCM [CUT52] and delta modulation [JAY70].

Another method that takes advantage of high repetition of symbols is run-length encoding (RLE). This model was first defined by Capon in 1958 [CAP58] and it is based on coding each symbol along with the length of the runs of that symbol (count of symbols), instead of individual symbols. This model does not reduce the entropy of the source sequence but the number of symbols that need to be encoded (in fact, the entropy is increased due to the removal of repeated values). RLE is easy to implement and presents low complexity, although its performance is relatively poor.

Other compression techniques, such as dictionary schemes (based on Ziv and Lempels' studies [ZIV77] [ZIV78]) and the ppm algorithm [CLE84], have also been widely used, but they are not the focus of this thesis.

1.4 Transform coding

The main idea of transform coding is to change the input samples (*P*) into another equivalent set of values in which most of the information is contained in only a few elements, and this way, we reduce the redundancy of the components. Many of theses transform methods use well-known space-to-frequency transforms, in which the canonical basis is replaced by another basis with frequency interpretation. The transformed samples (*C*) are coefficients that multiply an element from a new basis and, therefore, they are called transform coefficients.

This type of transform is particularly useful for natural images, because in this type of image, most energy tends to concentrate in the low frequency areas, and hence in a few transform coefficients. After conversion from space to frequency domain, we expect less correlation in the low-frequency coefficients than in the high-frequency ones. We can achieve larger reduction in entropy by applying quantization on the coefficients, given that many of them become 0. A quantizer usually maps an interval of real numbers to a single integer index, and it is the lossy part in transform lossy compression, because the exact reconstruction of the original coefficients is not guaranteed. An important parameter that determines the compression ratio and the quality lost is the length of the mapping intervals in the quantizer. So, the highest this interval is, the greater compression is achieved at the expense of poorer image quality. The last step in an image encoder is the entropy coding of the quantized coefficients (C^q). Since these coefficients have low entropy, an entropy coder is able to generate a compressed bitstream from them in a very efficient way. The complete transform coding and decoding processes are depicted in Figure 1.2.

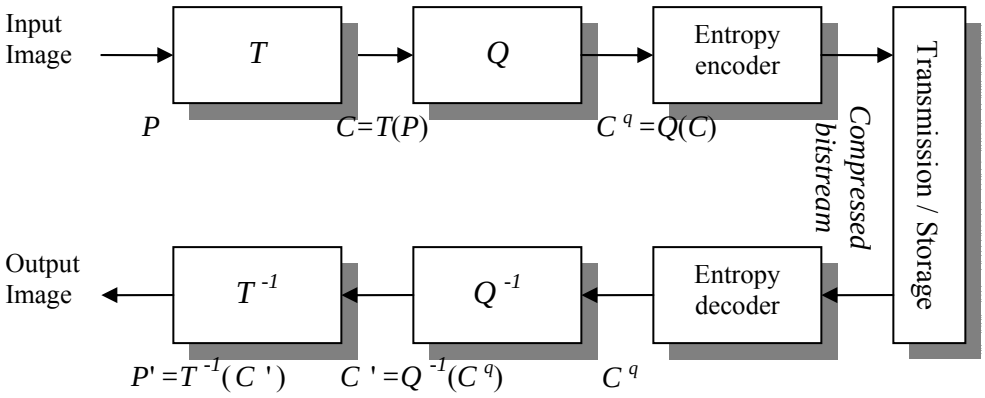


Fig. 1.2: Overview of an image coder and decoder based on transform coding. T and T^{-1} are the forward and inverse transform functions respectively. Q and Q^{-1} are the quantizer and dequantizer functions respectively. The original set of pixels is represented by P.

Most transforms (T) used for image coding are invertible, and hence there is a T^{-1} function so that $P = T^{-1}(T(P))$. However, in floating-point implementations, this equality is not ensured due to possible rounding errors, and it is necessary the use of integer arithmetic with reversible operations to guarantee the previous equality. On the other hand, after the quantization and dequantization processes, all the coefficients belonging to the same interval are recovered to the same value in that interval, independently of their initial value, and as a

consequence $C \neq Q^{-1}(Q(C))$.

Although the Karhunen-Loève Transform (KLT) has been proved to decorrelate the energy better than any other block-based image transform, it is not used in image compression because it has to compute a different basis functions for each image and transmit it to the decoder along with the coefficients, increasing the required bandwidth. In addition, it is computationally very inefficient. Currently, the Discrete Cosine Transform (DCT) and the Discrete Wavelet Transform (DWT) are the main transforms used in image compression.

1.4.1 DCT-based image compression

The DCT is almost as efficient as the KLT, with the advantage that its complexity is much lower. Contrary to the KLT, the basis functions for the DCT are always the same cosines at various frequencies. This transform is usually applied to images in small block sizes (typically 8×8). Since it is separable, a block of pixels can be transformed using a one-dimensional DCT transform first in columns, and then in rows (or vice versa). The JPEG standard [ISO92] takes this approach, with a structure similar to that shown in Figure 1.2. Hence, images are divided into 8x8 blocks, and each block is quantized and entropy coded using zigzag order and run-length encoding.

Since the DCT is applied in small block sizes, only a reduced amount of memory is specifically needed for the transformation process. In addition, due to the current importance of DCT-based coders, the complexity of the DCT is a well-studied issue [RAO90], and some proposals have been done to reduce the complexity of the Inverse DCT [MUR98] and, more recently, of the Forward one [LEN04].

1.4.2 Wavelet-based image compression

One of the lateral effects of block processing in the DCT is that blocking artifacts appear in moderate to high compression ratios. According to the HVS model, block edges are easily identified and hence, the visual image quality is highly degraded. Moreover, redundancy is not optimally removed from an image because each block is encoded independently, and only the DC component is decorrelated using differential coding. The wavelet transform is able to overcome these drawbacks given that it may be applied to a complete image, and hence it achieves better redundancy removal without blocking artifact. Thus, if the DWT is used for image coding, better visual quality and compression performance is achieved. For this reason,

the JPEG 2000 standard [ISO00] replaced the use of the DCT by the DWT. In addition, lossless coding can be performed in JPEG 2000 by applying a reversible integer-to-integer wavelet transform [ADA00] with exactly the same compression algorithm as in the lossy case.

On the other hand, some new problems arise when the DWT is used instead of the DCT. In the regular DWT computation, the entire image must be kept in memory, increasing the memory requirements compared with the 8x8 block of pixels needed in a DCT-based coder. Moreover, wavelet-based coders are typically implemented by memory-intensive and time-consuming algorithms. In order to avoid these usual drawbacks, in this thesis, we will tackle these problems in the main stages of a wavelet encoder, i.e., both the wavelet computation and the quantization and coding stages.

1.5 Important factors in the design of an image encoder

The main goal of an image compression system is to reduce the amount of bytes needed to represent an image, and hence this has been considered the key factor in the design of an image encoder. This feature can be easily measured in a lossless encoder simply evaluating the compression ratio, but in lossy compression, it cannot be considered as a single factor but we should evaluate the curve relating compression ratio and image quality, usually referred to as rate/distortion (R/D) performance.

Substantial reduction of finite system resources, such as bandwidth or disk space, can be achieved through image compression. However, the introduction of a compression module may establish higher requirements in other system resources, mainly in the required amount of physical memory (particularly RAM memory) and processor power. These new higher requirements were not present in the initial system and might not be acceptable. This way, depending on the application purpose and the available system resources, lower bandwidth reduction could be preferred if it entails a reduction in the encoder complexity (both computational and memory). This trade-off between the R/D curve, amount of memory and computational complexity is of major importance in the design of an image encoder. The degree of importance of each of these three factors depends on the final application. For example, when a computer is used to decode a picture, plenty of memory and CPU power are usually available; conversely, it is desirable that the digital camera used to take a picture is able to perform the coding process with as little memory and CPU power as possible, in order

to reduce hardware costs.

The relationship between these three factors is equally balanced in time, because while electronic devices are getting faster and faster, they come with more memory, and larger disk drives and faster networks are available. As a result, these factors are reduced by technology evolution but boosted by the growing demands of applications.

Great efforts are often made to achieve slightly improvements in rate/distortion performance (and thus in bandwidth or disk space demand), while the reduction of memory consumption and complexity of image coders is a less studied issue, but of great importance in resource-constrained devices like mobile phones, digital cameras and PDAs. In this thesis, we will deal with this issue.

In order to compare image encoders depending on the factors that we have previously defined, it is important to have a method to measure them. In the next subsections, we will address some exiting methods, and we will discuss which of them are suitable for our purpose.

1.5.1 How to measure the rate/distortion performance

For a given image and a given encoder, a rate/distortion curve can be simply computed from several iterations, in which the same image is encoded at various compression ratios (from low to high compression ratios), achieving different image quality. The compression ratio reached in each execution can be easily calculated as the relationship between the size of the compressed image and the number of pixels in the image, and it is generally known as bits per pixel (bpp) or bit rate (or simply rate). Rate/distortion performance is often given in the form of a table containing the image quality at various bit rates (typically, 0.125, 0.25, 0.5 and 1 bpp for grayscale images). Contrarily to compression ratio, it is not easy to find a quality metric that is universally accepted to determine the distortion (or quality loss) introduced by lossy compression. Two different approaches to quality measurement are usually followed: subjective and objective metrics.

Subjective quality measurement is based on the exhibition of an unimpaired image and the same image after compression (i.e., being coded and decoded). Non-expert viewers express their opinion about the quality of the decompressed image, scoring it in a range from 1 (bad) to 5 (excellent) (sometimes referred to as Mean Opinion Scored). Non-expert viewers' opinion is preferred because experts' judgment is biased, since they know how

compression works and how to search for artifacts that other people would not notice. This subjective quality evaluation, which is more detailed in ITU-R Recommendation BT.500-10 [ITU00], presents several drawbacks. First, subjective tests are influenced by the viewer and viewing conditions, which leads to human judgments that vary from one person to another, and from one time to another. Moreover, these massive surveys are expensive and time-consuming processes. For this reason, we need another method for a faster and more direct image quality evaluation, which can be helpful to decide which options are valuable (from a rate/distortion point of view) while an image encoder is being developed.

Due to the problems of subjective metrics, objective measures are heavily used as an easier way to compare image encoders and to choose the proper parameters for optimal performance. This type of metric relies on mathematical calculations. We consider it an objective method because, for a given image and an impaired version, it always produces the same results. The most widely used objective metric is the Peak to Signal Noise Ration (PSNR), which is defined as:

$$PSNR = 10 \log_{10} \left(\frac{MAX^2}{MSE} \right) \qquad (1.8)$$

PSNR measures quality given that it is based on the inverse value of the Mean Square Error (MSE) between the original and compressed image. It is calculated relative to the square of the highest sample value that can be represented in the original image, and that is why it is called Peak SNR. Most times, for a grayscale image, image depth is 8 and so $MAX = 2^8 - 1$. A logarithmic operation is used to approach the measurement to how the HVS perceive image errors, more logarithmically than linearly. Despite the wide use of PSNR, it has been often criticized because human perception of error usually differs from that exposed by PSNR. For instance, very high and concentrated errors, such as pixel overflow artifacts and blocking artifacts, slightly affect PSNR. However, this type of distortion attracts the viewer's attention, and hence it severely degrades the image quality from a human perspective. For this reason, some proposals have been made to replace PSNR as objective metric [WUH01] [TAN00], and the emerging ITU-T Video Quality Experts Group (VQEG) is trying to develop and standardize some alternatives, although this is still an open research issue. Therefore, in this thesis, like in almost all the papers in the current literature, we will use PSNR as quality metric. In our experiments, this approach makes sense because we are dealing with wavelet-based encoders, in which the transform is applied to the whole image.

Therefore, the distortion introduced by uniform quantization will not create border effects or will be focused on a block of pixels, but it is distributed over several areas in the image. Hence, we consider PSNR a valid method to compare the performance of different wavelet-based encoders. In fact, in a recent report, the video coding expert group (VCEG) considers it an open problem, but they admit that PSNR is one of the best methods among those currently available [VCE].

In conclusion, we will compare rate/distortion performance using several standard images, with different features and detail level to characterize from low to high frequency images, and computing a R/D table (bpp/PSNR) for each image.

1.5.2 How to measure complexity

The time complexity of short algorithms, such as many ordering algorithms, is usually expressed as the order of the asymptotic complexity using O-notation. In these algorithms, each single operation is considered to be executed in constant time, and the time behavior of the algorithm (number of operations executed) is given as a function of the amount of data to be processed, typically a linear, a quadratic or an exponential function. Unfortunately, an image encoder is a complex algorithm that involves many types of operations with different execution cost. Therefore, the use of asymptotic complexity is not suitable. An example is the entropy coding stage in transform coding. The execution time needed to encode a set of coefficients greatly depends on whether a Huffman encoder is used, or a more complex adaptive arithmetic encoder with several contexts is applied. Moreover, asymptotic complexity does not evaluate the correct use of specific architectures, e.g., it does not assess good cache use and the lack of data dependency in computer architectures.

Instead of asymptotic complexity, execution time measurement will be used in this thesis as a way to assess complexity. Actually, execution time does not measure the complexity of an algorithm but of an implementation of that algorithm running on a specific system. Since execution time is largely dependant on the optimization level (e.g., use of platform dependant assembly code or multimedia SIMD instructions), all the implementations evaluated in this thesis are written under the same conditions to be compared as fairly as possible, using standard ANSI C/C++ language with the same optimization level for all of them.

Most Operating Systems provide tools to measure the execution time of a program (e.g., *time* command in UNIX), but with too low accuracy. Instead of them, we will read the time-

stamp counter (a 64-bit register available for Intel© Processors), which is incremented in every clock cycle, allowing much finer-granularity for execution time measuring.

1.5.3 How to measure memory usage

The third parameter that we need to measure is the memory required to implement a compression algorithm. In transform coding, it includes the memory needed to compute the image transform, and additional data required to hold the state of the coding process. Recall that, in general, DCT-based encoders need less memory than most wavelet-based encoders, due to the way the DCT is computed, in blocks, compared to the DWT, in which an image is processed as a whole. On the other hand, the dynamic statistical model used by an adaptive arithmetic encoder is an example of state information to be held, which is implemented as a frequency count table $f(x_i)$.

In some algorithms, memory usage can be easily assessed through a theoretical model (e.g., in the transform computation only the amount of coefficients have to be counted). However, sometimes the memory related to state information is not always so predictable, because it largely depends on the features of the incoming image. Therefore, in a more practical manner, memory usage can be measured (in bytes) by means of a counter that is increased as dynamic memory is being allocated. This second approach requires modifying the source program under evaluation to incorporate the memory count. Another practical alternative to avoid this code modification consists in looking up the memory statistics provided by the Operating System (e.g., the "peak memory usage" column in the Windows XP task manager), although in this case, we have to take into account that those results may include additional process memory such as the program memory. In this thesis, we will choose one of these metric methods depending on the test conditions.

1.6 Other features

Besides the characteristics presented in the previous sections, other features may be considered of interest in an image compression scheme, depending on the application purpose. One of these characteristics is symmetry. If the encoder and decoder have the same complexity, we consider that they are symmetric. Symmetry can refer to execution time, memory requirements or both. Vector quantization and fractal coding are examples of

asymmetric coding, because the search for the codebook is a time-consuming process that is only carried out by the encoder. Real-time multimedia applications, such as videoconferencing, require symmetric complexity while other applications, like video on demand and pre-recorded DVB, admit slower coding than decoding, since only the latter has real-time requirements.

Other properties define the output bitstream and its interpretation rather than the encoder and decoder architecture. If the encoder generates a bitstream that defines several versions of the same image at different resolution levels, we say that the bitstream is resolution scalable, in other words, it allows multiresolution representation. If this feature is present in a bitstream, the decoder can decode a small-resolution version of the image from the first portion of the bitstream and, as it gets more information, it can reconstruct larger versions of the image. This property is also inherent of the wavelet transform, as we will see in the next chapter.

A bitstream is SNR scalable if the decoder can reconstruct a lower-quality full-resolution version of an image from the first portion of the bitstream, and the image quality (measured as SNR) improves as more information is received. SNR scalability is usually implemented with bit plane coding, by transmitting first the bit planes that achieve greater quality improvement. Note that the term progressive coding can refer to resolution progressive or SNR progressive coding (or even both).

A bitstream is said to be embedded if one can extract a smaller part of the bitstream achieving a characteristic that is the same as though the bitstream would have been generated directly with that smaller size.

Rate control in image compression is the capability of a coding algorithm to achieve a target bit rate when coding an image. For a given encoder, optimal compression performance is demanded for the target bitrate. Although some rate control methods are very exact, many times, precise rate control is not needed and the achieved bitrate can differ slightly to the target bitrate.

Another interesting bitstream feature is the random access. This property defines the ability to decode only a specific area of the image from the bitstream. In order to accomplish random access, some regions of the image are encoded without statistical dependence (e.g., using different probability models in each area for adaptive arithmetic coding). This region can be rectangular or arbitrary shaped.

Finally, in some cases, it can be desirable to encode an area of the image at a higher level of quality (e.g., a face in a portrait). This area is called region of interest (ROI). ROI coding can be achieved by applying lighter quantization to the coefficients in the ROI than to those in the background.

Chapter 2

Wavelet transform computation

In Chapter 1, we referred to the importance of transform coding within the state-of-the-art image coding algorithms, being those based on the wavelet transform the most efficient in terms of rate/distortion performance. For this reason, we will focus on wavelet-based image coding. In these encoders, the computation of the wavelet transform imposes serious restrictions on the global memory consumption and on the entire encoder complexity. Thus, it is a critical part. It is even more important at very low bit-rates, since in that case, the time spent on entropy coding might become almost negligible, and most time is spent in the transform computation. In this chapter, we introduce the discrete wavelet transform (DWT) and various algorithms to compute it. Particularly, we will focus on the two-dimensional DWT for image coding, introducing some of the current proposals for efficient computation of the DWT.

2.1 Introduction to wavelet transform for image coding

2.1.1 Why a new transform?

The main idea behind transform coding is being able to find a better representation for an input signal, which can be, for instance, a set of pixels *P*. When we pursue data compression, "a better representation" means that the new set of coefficients *C* multiplying the new basis has less redundancy (i.e., less correlation among their components) than the original one. This way, the energy of the coefficients is more compacted so that entropy coding can be applied more efficiently, most of all after quantization.

In general, in a finite and discrete linear transform, we can express a discrete signal *S* as a

set of N coefficients $c_i \in C$ multiplying each basis vector T_i as:

$$S = \sum_{i=1}^{N} c_i T_i \qquad (2.1)$$

For an efficient representation of the input signal, a few basis vectors should be able to approach S, so that only a small number of coefficients are necessary to represent the original signal in a compact way, or at least to approximate it in the case of lossy coding. Thus, the selected transform plays an important role in the coding efficiency. The efficiency can be improved if we choose a transform that matches the characteristics of the input signal using as few basis vectors as possible. Hence, we need to analyze the characteristics of the input signal in order to find the suitable transform.

Since we aim at digital image compression, and pictures are size-limited, one of the features of the input signal is that it is limited in space. In addition, the Nyquist theorem can be used to prove that digital images are also limited in frequency (i.e., band-limited). In fact, natural images tend to concentrate most energy in low frequency components. Moreover, the human eye is less sensitive to high frequency components, and therefore small loss in those components can be accepted without visually perceptible distortion. So, we can discard high frequencies if they are small enough. As a conclusion, we have to search for a basis that matches these features, both space and band limitation.

Fourier-based transforms[1] can efficiently represent band-limited signals, but they have the inconvenience that the functions forming its basis (sines and/or cosines) are not space-limited. In a practical sense, this means that we can determine all the frequencies in the input signal, but we cannot know where these frequencies are placed. Moreover, since these basis functions are not space-limited (or time-limited), the Fourier transform is useful to approach stationary sources, which do not vary their behavior in time. Unfortunately, this is not the case of natural images, which present a non-stationary behavior, and hence it would be preferable to perform local analyses of the signal instead of the global analysis that the Fourier transform performs.

In the opposite case, an input signal can be analyzed in small portions to determine the existing frequencies in each area. In the extreme case, in which that area is infinitesimal, we

[1] Examples of Fourier-based transforms are the DCT, used for highly correlated sources, or the DST, which is less frequently employed because it fits better highly variable sources.

can use the Dirac[1] pulse as basis function. This transform is space-limited but it has the drawback that it contains all the possible frequencies, and hence no practical information about frequencies can be extracted. In the equivalent discrete case, we can extend the pulse width to the sampling period, but we still cannot extract frequency but only spatial information (in fact, this is the case of the canonical representation).

An intermediate solution is the short-term Fourier Transform [GAB46], in which an input signal is broken into fixed-length fragments, and the Fourier analysis is applied to each of them. However, blocking artifacts appear if this approach is applied for image coding. In addition, we still have to use sines and cosines as basis functions, which is not always the best option to approximate natural images. Another important disadvantage of this transform is that all the functions in the basis (and thus the transform coefficients) cover exactly the same area, independently of their frequency. This effect is presented in Figure 2.1, in which coefficients from a Fourier-based transform are given along with their corresponding basis functions. In this example, all the basis functions have the same support length (i.e., the length of the area that they cover is the same). A better image analysis can be done if higher frequency functions cover smaller signal pieces, which is known as a multiresolution representation.

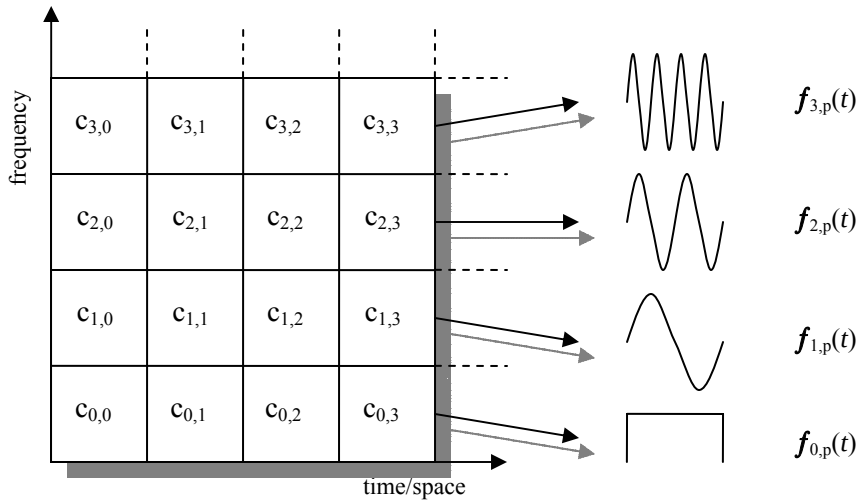


Fig. 2.1: Coefficients from a Fourier-like basis and the basis functions. For each coefficient ($c_{i,j}$), the first subindex (i) indicates the frequency of its basis function, while the second one (j) shows its position from the origin. Notice that all the basis functions cover exactly the same signal portion, independently of its frequency.

[1] We define the Dirac pulse as $f(x)=1$ for $x=0$ and $f(x)=0$ for the rest of x.

2.1.2 Wavelet transform

In contrast to the Fourier transform, the wavelet analysis uses a basis formed by functions that are finite in both the frequency and spatial domain, which are called wavelet functions $\Psi_{s,p}(t)$. All these wavelet functions are generated from translation and dilation of a single function, called mother wavelet $\Psi(t)$. The wavelet function generation is defined as:

$$\Psi_{s,p}(t) = \frac{1}{\sqrt{s}}\Psi\left(\frac{t-p}{s}\right) \qquad (2.2)$$

where *s* is the scaling parameter and *p* is the translation parameter. Notice that as the scaling parameter gets larger, the generated function gets wider, so it covers a larger area but a narrower frequency band (we will prove it later with the Fourier transform). The translation parameter serves to shift the function support, indicating its position from the origin. We will refer later to Figure 2.3, but at this moment, we can compare in this figure three wavelet functions at different scales ($\Psi_{1,p}(t)$, $\Psi_{2,p}(t)$ and $\Psi_{3,p}(t)$). This figure shows the effect of varying the scale parameter in a wavelet function.

In the continuous version of the wavelet transform, a finite-energy function $f(t)$ can be represented with wavelet functions as:

$$f(t) = \int\int \psi(s,p)\Psi_{s,p}(t)\, ds\, dp \qquad (2.3)$$

where the $\psi(s,p)$ function can be calculated using the continuous wavelet transform (CWT):

$$\psi(s,p) = \int f(t)\Psi^{*}_{s,p}(t)dt \qquad (2.4)$$

Actually, the wavelet transform is not unique, but we can define a different type of wavelet transform (also called wavelet family) depending on the selected mother function. This allows us to choose the wavelet family that fits better with the signal that we want to compress. In the Fourier transform, this choice cannot be made.

A mother function needs to meet several requirements to form a valid wavelet family. One of these requirements gives name to these functions. In particular, a valid wavelet function must have an average value of 0 in the time domain, i.e.:

$$\int \Psi(t)dt = 0 \qquad (2.5)$$

This means that it must be oscillatory, and that is why we call it a wavelet function. A related feature of a wavelet family is the number of vanishing moments of a function. We say that a function $\Psi(t)$ has *n* vanishes moments if:

$$\int \boldsymbol{\Psi}(t) t^i \, dt = 0 \quad \forall i = 0,1,\ldots,n-1 \qquad (2.6)$$

The number of vanishing moments characterizes how well a wavelet basis approaches polynomials, so the higher the number of vanishing moments, the higher order polynomials a wavelet basis approaches (see admissibility condition in [SHE96]), and to a certain degree, it approaches better smooth functions. Since most natural images have smooth features, a certain degree of vanishing moments is desirable in order to approach images for compression purposes.

Since we want to encode digital signals, we are more interested in the discrete wavelet transform (DWT), in which the *s* and *p* parameters are not continuous but discrete. In the DWT, an infinite but discrete number of wavelet coefficients $\psi_{s,p}$ can decompose a finite-energy function as follows:

$$f(t) = \sum_{s=-\infty}^{\infty} \sum_{p=-\infty}^{\infty} \psi_{s,p} \, \boldsymbol{\Psi}_{s,p}(t) \qquad (2.7)$$

For a real implementation, we still need to decompose it into a finite amount of coefficients to be able to compute them with a computer. We can ensure that the translation parameter *p* is finite simply by limiting the set of functions that the DWT can describe to those that are space-limited. That is not very difficult in our case, because we handle digital images and this requirement is inherent to this type of data. On the other hand, although we deal with finite-energy functions, the finiteness of the scaling parameter is not as easily achieved. The reason for this difficulty can be better understood by studying the effect of the scaling parameter on the frequency of the input signal. For this purpose, we will take a different approach to the wavelet transform as a band-pass filter.

Consider a frequency-limited input signal, with a frequency band denoted by the dotted area in Figure 2.2(a) (e.g., the input signal can be one sampled at a frequency $2f_s$, in which frequencies up to f_s are captured according to the Nyquist theorem). One of the features of the DWT is that the upper half subband of the input signal can be described using first-level wavelet functions, as we will see in the multiresolution analysis (the following subsection). In addition, if the amount of functions (and therefore coefficients) needed to represent this upper-half subband is half the number of samples in the input signal, the transform is said to be non-expansive.

The next level of wavelet functions is generated by scaling the first-level functions by a factor of 1/2, which results in space dilation. From the Fourier analysis, we know that dilation

in space implies contraction and shift in frequency, as described in:[1]

$$\mathcal{F}\left\{ f\left(\frac{t}{s}\right)\right\} = sF(s\omega) \qquad (2.8)$$

Thereby, the second-level wavelet functions cover half the remaining low-band spectrum of the input samples. In a dyadic decomposition, as we define more wavelet levels in the same way, we cover the upper subband of the remaining low-bands at each level. However, the whole spectrum of the input signal cannot be covered using a finite number of wavelet functions at different levels, because we always have a remaining low frequency subband to be covered at any level, as it is shown in Figure 2.2(a). In the extreme case, we know that the DC component cannot be represented by a function or set of orthogonal functions integrating to 0, such as the wavelet functions. Hence, another function $\Phi_l(t)$ must be introduced to cover the remaining low-frequency subband at any given level (l), as it is depicted in Figure 2.2(b). This new function is called scaling function.

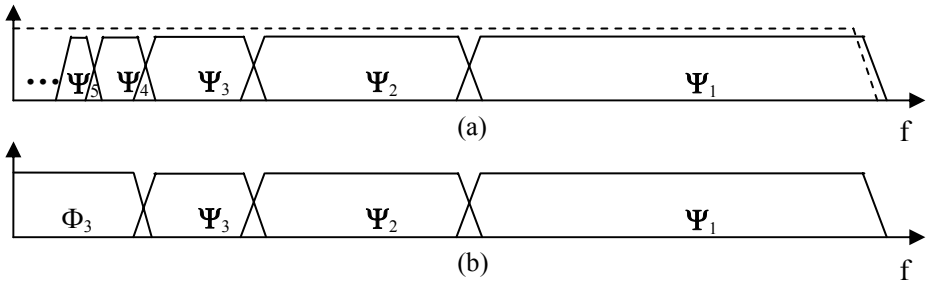


Fig. 2.2: Frequency band covered by wavelet functions at various levels in a dyadic decomposition. The upper plot (a) reveals the need for another type of function (scaling functions) to cover the low-frequency band as it is shown in the lower plot (b).

The generation of scaling functions at several levels is similar to that explained in the case of the wavelet functions, i.e., dilation and translation of a generating function $\Phi(t)$. This function has a low-pass behavior and it has to meet some requirements, such as:

$$\int \Phi(t)dt = 1 \qquad (2.9)$$

With the introduction of the scaling function, we can establish now a finite wavelet decomposition of a space-limited discrete signal S as:

$$S = \sum_{s=1}^{L}\sum_{p=1}^{N/2^s} \psi_{s,p}\,\boldsymbol{\Psi}_{s,p}(t) + \sum_{p=1}^{N/2^L} \varphi_{l,p}\Phi_{l,p}(t) \qquad (2.10)$$

[1] The Fourier transform of a function $f(t)$ is expressed as $\mathcal{F}\{f(t)\} = F(\omega)$

where N is the number of samples in the input signal, L is the decomposition level (number of wavelet scales), and finally $\psi_{s,p}$ and $\varphi_{s,p}$ are the resulting wavelet and scaling coefficients respectively (a non-expansive transform is being considered).

Observe that the number of wavelet coefficients required to represent S at a level decreases as the scale level increases, which involves that higher frequency bands are represented with higher resolution (lower scales represent higher frequencies). This is a difference of wavelets compared with the short-term Fourier transform, in which all the frequency bands are represented using the same spatial resolution. This difference between both transforms can be seen comparing Figures 2.1 and 2.3, where the coefficients resulting from the Fourier and wavelet transforms are shown. In this figures, each coefficient is placed in a tile indicating the frequency band and the space area that it covers. In Figure 2.3, the y-axis shows the frequency spectrum that each wavelet level covers, in a consistent way with Figure 2.2, while the x-axis measures the area that each coefficient covers, with higher spatial resolution on higher frequency bands, and hence showing the better adaptability of the wavelet transform to details of the image.

In this section, we have presented an intuitive introduction to wavelets rather than a formal description. An approach to wavelets with more mathematical background and proofs of some assumptions that we have made can be readily found in the literature [DAU92] [VET95] [STR96].

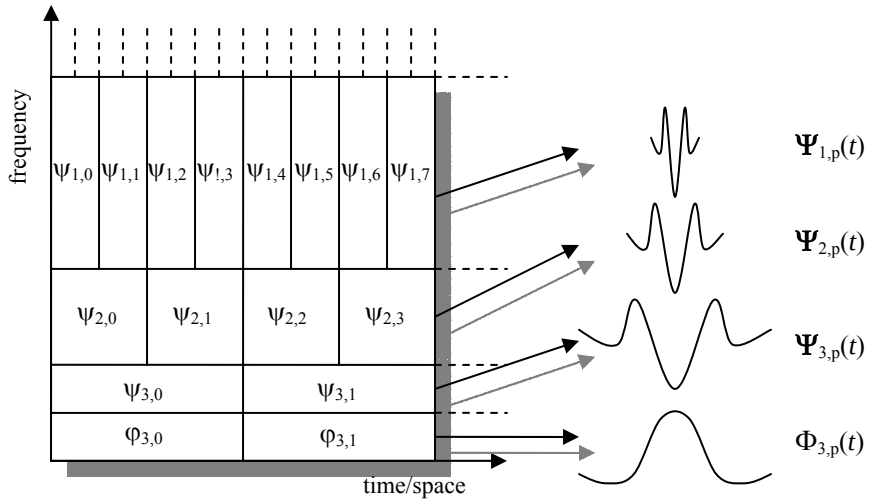


Fig. 2.3: Coefficients from a wavelet basis, and its basis functions (including the scaling function). For each wavelet coefficient ($\psi_{i,j}$), the first subindex (i) indicates the scale of its basis function, while the second one (j) shows its position. In contrast to Fourier, in the wavelet transform, basis functions with higher frequency have lower support, and hence higher spatial resolution.

2.1.3 Multiresolution Analysis

One of the main benefits of wavelets for image analysis is the introduction of a multiresolution representation, which was first formally described by Mallat in [MAL89]. For this analysis, consider V_0 the space generated by a scaling function with the *s* parameter equal to 0 so that a function $f(t)$ can be defined in V_0 as:

$$f(t) = \sum_p \varphi_{0,p} \Phi_{0,p} \qquad (2.11)$$

In general, we call V_n to the space generated by the linear combination of $\Phi_{n,p}(t)$.

Before continuing with the Multiresolution analysis, it is important to see that the scaling function posses the self similarity property, in other words, it can be recursively generated by scaled (shrunk to half) and shifted versions of itself, which is mathematically described by:

$$\Phi(t) = \sum_p c_p \Phi(2t - p) \qquad (2.12)$$

This is important because it shows that any function of V_{n+1} is also defined in V_n, while the reverse is not true. In general, it allows us to nest the subspaces V_n as follows:

$$\ldots \subset V_L \subset \ldots \subset V_{n+1} \subset V_n \subset \ldots \subset V_1 \subset V_0 \subset \ldots \qquad (2.13)$$

If we focus on an nth subspace level, the fact that $V_{n+1} \subset V_n$ implies that there is a subspace W_{n+1} that is the complement of V_{n+1}, so that V_n can be decomposed as:

$$V_n = W_{n+1} \oplus V_{n+1} \qquad (2.14)$$

The W_{n+1} subspace is generated by the corresponding wavelet functions at a level n+1, and hence both the wavelet and scaling functions should be chosen in a way that these nested relations and space decomposition hold.

As a result, the multiresolution analysis in signal processing implies that a signal defined in V_0 can be represented by two coefficient vectors: one defined in W_1 and another defined in V_1. In an intuitive way, the signal representation resulting from the scaling subspace V_1 resembles the original signal, but represented with lower spatial resolution, while the one from the wavelet subspace W_1 captures the details that are lost by the resolution reduction. This process is known as one-level wavelet decomposition, while a *L*-level dyadic wavelet decomposition is achieved by recursively applying *L* successive decompositions on the scaling coefficients, so that a signal defined in V_0 is decomposed in coefficients from the *L* wavelet spaces and one scaling subspace that decompose the original space as shown in the

following expression:

$$V_0 = W_1 \oplus W_2 \oplus \ldots \oplus W_{L-1} \oplus W_L \oplus V_L \qquad (2.15)$$

2.2 DWT computation using filter banks

2.2.1 1D DWT computation

From the multiresolution analysis, we know that both V_{n+1} and W_{n+1} are subspaces of V_n, i.e., $V_{n+1} \subset V_n$ and $W_{n+1} \subset V_n$. Then, a scaling function of V_{n+1} can be computed in terms of functions of V_n, and so can a wavelet function of W_{n+1}. Thereby, there are two sets of coefficients $\{h_k\}$ and $\{g_k\}$ satisfying that:

$$\Phi(t) = \sum_k h_k \Phi(2t-k) \qquad (2.16)$$

$$\Psi(t) = \sum_k g_k \Phi(2t-k) \qquad (2.17)$$

where $\Phi(t) \in V_{n+1}$, $\Psi(t) \in W_{n+1}$ and $\Phi(t), \Psi(t), \Phi(2t-k) \in V_n \ \forall k$.

By means of these relationships from the multiresolution analysis, we can define the wavelet transform computation as a simple two-band filtering operation. The set of filter coefficients (or filter taps) $\{h_k\}$ defines a scaling function and it has a low-pass behavior. On the other hand, the filter $\{g_k\}$ defines a wavelet function and thereby, it is a high-pass filter.

Recall that any function $f(t)$ of V_0 can be represented with functions of V_1 and W_1:

$$f(t) = \sum_p \varphi_{0,p} \Phi_{0,p} = \sum_p \varphi_{1,p} \Phi_{1,p} + \sum_p \psi_{1,p} \Psi_{1,p} \qquad (2.18)$$

Consider that the input signal S is represented by the set of scaling coefficients $\{\varphi_{0,p}\}$. The wavelet and scaling coefficients for one-level wavelet decomposition ($\{\psi_{1,p}\}$ and $\{\varphi_{1,p}\}$ respectively) can be computed from $\{\varphi_{0,p}\}$ using the previous filter bank:

$$\varphi_{1,p} = \sum_k h_k \varphi_{0,p+k} \qquad (2.19)$$

$$\psi_{1,p} = \sum_k g_k \varphi_{0,p+k} \qquad (2.20)$$

This pair of low-pass and high-pass filters is known as analysis filter bank. After applying the filters, the resulting coefficients are redundant, and hence we use a down-

sampling by a factor of two on each coefficient set (i.e., the odd coefficients are discarded), having the same number of coefficients after transformation as in the input signal (note that we use a non-expansive transform, also known as critical sampling). The reconstruction of the original scaling coefficients from the transformed coefficients can be performed with another pair of low-pass and high-pass filters ($\{\tilde{h}_k\}$ and $\{\tilde{g}_k\}$ respectively), which form the synthesis filter bank. Before applying the synthesis filters, the transformed coefficients have to be up-sampled by a factor of two by inserting 0s between every two samples. Once the transformed coefficients are returned to their original resolution, the exact input signal $\{\varphi_{0,p}\}$ can be recovered (without quantization errors) by adding the result of applying the low-pass synthesis filter to the scaling coefficients ($\{\varphi_{1,p}\}$) and the high-pass synthesis filter to the wavelet coefficients $\{\psi_{1,p}\}$. The whole process is depicted in Figure 2.4.

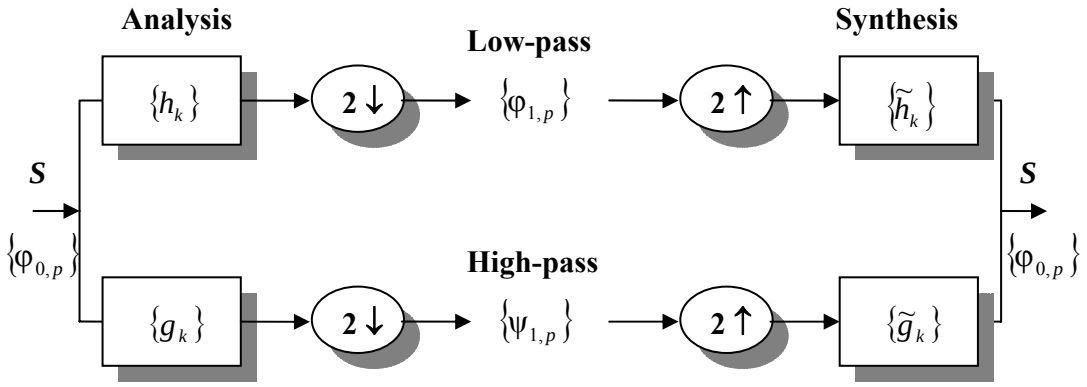
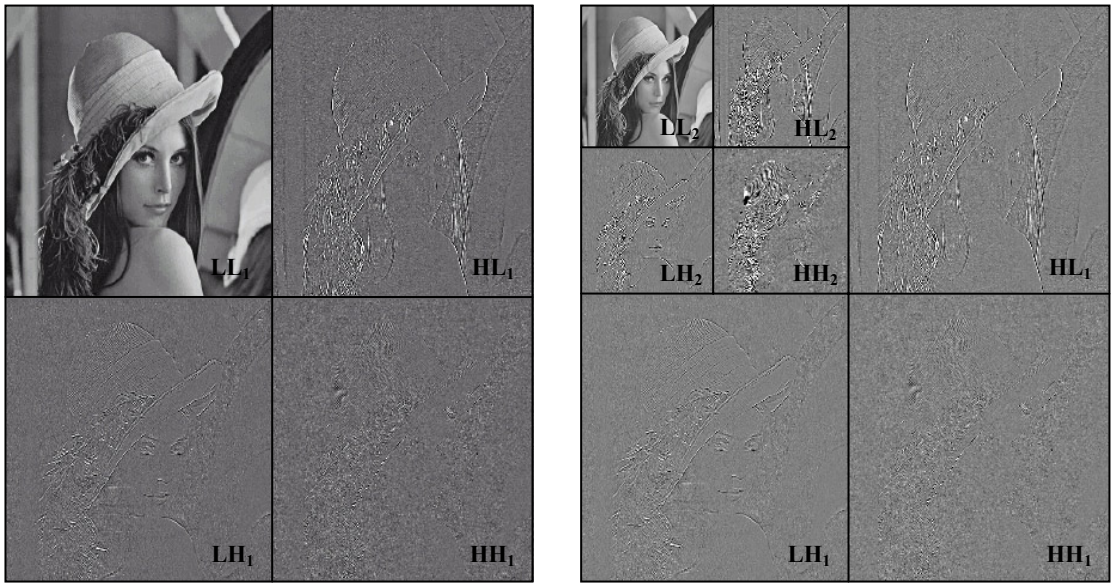


Fig. 2.4: One-level wavelet decomposition of an input signal using an analysis filter bank, and signal reconstruction using a synthesis filter bank.

Since this algorithm involves a simple filtering operation, its asymptotic complexity is O(*n*), which is lower than the complexity of the DCT, O(*n*log*n*). For this reason, the DWT can be applied on larger image pieces (or even the whole image), while the DCT is processed in small blocks (recall that another reason is the lack of spatial locality of the DCT). The exact number of operations required to execute this algorithm depends on the filter-length and its symmetry. Thus, the number of multiplications required to transform *n* input samples using a filter with *N* taps is $n \times N$, while it is reduced to $n \times N/2$ if the filter employed is symmetric, as we will see later.

Because of the multiresolution analysis, this one-level wavelet computation algorithm can be generalized to any *N*-level dyadic wavelet decomposition by applying the same

decomposition to any intermediate scaling coefficients at a level n $\{\varphi_{n,p}\}$. In the same way, the corresponding wavelet and scaling coefficients at the following level ($\{\psi_{n+1,p}\}$ and $\{\varphi_{n+1,p}\}$ respectively) are computed. This process is repeated until the desired decomposition level (N) is reached. Then, the forward wavelet transform (FWT) is completed. On the other hand, the inverse wavelet transform (IWT) is computed by applying the synthesis filters in the reverse order, from the N level to the first one.



(a) (b)

Fig. 2.5: (a) One-level wavelet decomposition of an input image, (b) two-level wavelet decomposition, in which the LL_1 subband of the decomposition in (a) is further decomposed.

2.2.2 Higher order wavelet transform

The 1D DWT computation can be extended to higher dimensions by applying filtering on each dimension in a separable way. Since we are interested in the wavelet transform for image compression, and images are two-dimensional, we will use a filter-bank to achieve a one-level 2D DWT decomposition of an image by carrying out the 1D DWT both horizontally and vertically, i.e., first in rows and then in columns. The result of applying one-level 2D DWT decomposition on the standard Lena image is shown in Figure 2.5(a). From this decomposition, we get four different image subbands, which are usually called LL, HL, LH and HH subbands. The first letter in the subband name identifies the filter that has been applied horizontally, the second letter identifies the vertical filtering, and the number identifies the decomposition level. In the example of Figure 2.5(a), only subbands from the

first decomposition level have been computed. This way, the HL subband contains vertical low-pass information and horizontal high-pass information. As a result, it displays vertical details, since the result of a horizontal high-pass filter in an image is seen as vertical details. Conversely, the LH subband is the result of applying a low pass filter horizontally and a high pass filter vertically, and it shows horizontal details. Finally, the HH subband contains pure high frequency information, both horizontally and vertically, while the LL subband contains only low pass information, and hence it is a low-resolution version (half sized) of the original image. We cannot consider the LL subband as a wavelet subband because the coefficients of this subband multiply a basis with only scaling functions (both horizontal and vertical).

Most energy is concentrated in the LL subband, and hence further image decorrelation can be achieved if the same process is applied to this subband. Thus, a second decomposition level is obtained the way it is presented in Figure 2.5(b). In a dyadic wavelet decomposition, this type of decomposition is recursively applied to the remaining low frequency subband (LL) until a desired decomposition level (N) is reached. Other types of wavelet decompositions, such as wavelet packets [COI92], recursively apply the same scheme not only to the low frequency subband, but also to other wavelet subbands, in order to remove redundancy in these subbands (most of all in images with many high-frequency components). However, the algorithm to determine if a wavelet subband should be refined is time-consuming, and many times little benefit is gained. Therefore, this type of decomposition is not commonly used.

Multiresolution is an important feature of the wavelet transform for image coding given that it provides resolution scalability in a natural way. For this type of scalability, a decoder receives the LL_N subband in first place, which is the lowest resolution version of the original image. Then, the decoder can receive the wavelet subbands at that level (LH_N, HH_N and HL_N) and use them along with the LL_N subband to compute the following low-frequency subband (LL_{N-1}), doubling the resolution of the first version of the image (LL_N). As the decoder receives more wavelet subbands, it continue doubling the image size, until the desired image size is reached, or no more subband levels are available because the first-level wavelet subbands have been received, whatever comes first.

2.2.3 Desired filter and transform properties

Wavelet filter design for image coding is not within the scope of this thesis. Instead, the

reader is referred to the available literature [VET92] [COH92] [GAL88]. However, some filter properties considered in the design of wavelet filter banks are also of interest in the design of a wavelet image encoder. Unfortunately, not all these properties can be satisfied simultaneously, and different filter banks are constructed to meet different requirements depending on the application purposes (e.g., compression in our case).

Some of these interesting filter and transform properties are:

- *Perfect reconstruction*: This property ensures that the input signal can be exactly recovered after transformation if no quantization error is introduced. This property still holds in spite of the introduction of a down-sampling by two in the scaling and wavelet coefficients resulting from each decomposition level.
- *Linear phase*: If a filter has a symmetric response to an impulse, it is said to possess the linear phase property. This property can be achieved by using symmetric filters. In image coding, if a non-linear phase filter is employed, each pixel does not contribute equally to the transformed coefficients that are equally distant on each side of the pixel. After quantization and reconstruction, this unequal contribution introduces visually disturbing edge distortion in the image [LIM90].

  Another practical implication of this type of filter is their lower complexity compared to a non-symmetric filter of the same length, since symmetric filters can be implemented in a factorized form, halving the amount of multiplications that are necessary.

  Finally, the use of symmetric filters is crucial in order to solve the boundary treatment problem. When filtering a space-limited signal, we need to make up the values that are beyond the signal limits to be able to apply filtering on the samples that are close to the signal boundary. A simple assumption could be that these samples are 0, which is known as zero-padding. However, this approach has two drawbacks: 1) perfect reconstruction is only guaranteed if we generate more output coefficients than the number of samples in the input signal, in other words, we cannot apply critical sampling without error; and 2) in most cases, the introduction of zero padding will cause sharp discontinuities and hence large artificial high frequencies are generated. These consequences are harmful to image coding given that we need to encode extra coefficients, and in addition, the new unexpected high frequencies make the image more difficult to be encoded, achieving lower compression

efficiency because most compression algorithms are based in the fact that most energy is concentrated in low-frequency bands. These problems can be avoided by using symmetric extension along with a symmetric filter [BRI95]. In symmetric extension, an input signal represented by **abcde** (where each letter represents an input sample) is extended as edcb|**abcde**|dcba, i.e., the samples for signal extension in the boundary are copied from the input signal in a symmetric way. This extension is known as whole point symmetric extension, while another variant is the half point extension, which includes the last input sample in the signal extension, resulting in the following extension: edcba|**abcde**|edcba. The use of linear phase filters and symmetric extension allows perfect reconstruction with critical sampling, and avoids introducing extra high frequencies.

For all these reasons, we will always use linear phase filters for image compression in this thesis.

- *Orthogonality, orthonormality and bi-orthogonality*: We say that a two-band wavelet transform is orthogonal if the high pass and low pass filters are orthogonal to each other, i.e.:

$$\sum_k h_k g_k = 0 \qquad (2.21)$$

One way to build orthogonal filters is that $\{g_k\}$ satisfies the following condition:

$$g_k = (-1)^k h_{1-k} \qquad (2.22)$$

Orthogonal filters are convenient because the input signal is split into a low and a high frequency band without duplication of information. With this type of filter, the same filter pair is applied for the analysis and synthesis processes. In addition, if the transform is energy preserving, this transform is called orthonormal. In an energy preserving transform, the sum of the squares of the input samples is identical to the sum of the squares of the transform coefficients, which is known as the Parseval's theorem [SAY00]. In our case, for one-level decomposition, at a level *s*, energy preserving means that:

$$\sum_p (\varphi_{s,p})^2 = \sum_p (\varphi_{s+1,p})^2 + \sum_p (\psi_{s+1,p})^2 \qquad (2.23)$$

Orthonormal transforms are of interest for image coding given that the quantization error introduced in the transformed coefficients is identical to the resulting error in

the spatial domain, and hence for this type of transform, minimizing the MSE of the quantized coefficients is the same as minimizing it in the reconstructed image.

Unfortunately, a two-band filter bank cannot be orthogonal and linear phase simultaneously (except the Haar transform, which is not useful for image coding due to its lack of smoothness). Only bi-orthogonal transforms have this property [VAI93]. In a bi-orthogonal filter bank, we do not use the same analysis and synthesis filter pairs, and the high and low pass filters have unequal length, achieving symmetric filters while preserving perfect reconstruction. The term bi-orthogonal stems from the fact that the low-pass synthesis filter is orthogonal with respect to the high-pass analysis filter, and the same happens to the high-pass analysis and low-pass synthesis filters, that is:

$$\sum_k h_k \tilde{g}_k = 0 \qquad \sum_k \tilde{h}_k g_k = 0 \qquad (2.24)$$

A bi-orthogonal filter bank can be designed to achieve a very close to orthonormal transform. Hence, we will use this type of transform because they possess the valuable linear-phase property being nearly orthonormal.

- *Filter normalization*: The output of the analysis filters can be normalized by scaling the resulting coefficients by a constant factor provided that it is undone before the synthesis filtering. The normalization of the wavelet transform can be expressed in terms of the DC gain of the low-pass analysis filter, and the Nyquist gain of the high-pass analysis filter (G_{DC}, G_{Nyquist}), which can be computed as follows:

$$G_{\text{DC}} = \left| \sum_k h_k \right| \qquad G_{\text{Nyquist}} = \left| \sum_k (-1)^k g_k \right| \qquad (2.25)$$

With the use of the proper normalization ($\sqrt{2}$, $\sqrt{2}$), we can achieve an orthonormal transform if the original transform is orthogonal (or a close to orthonormal wavelet transform if a bi-orthogonal filter is used), allowing the use of the same uniform quantization step for all the wavelet subbands. However, a different normalization can be used to avoid dynamic range expansion for minimal memory usage in implementations using data types with low bits per sample (e.g., if we want to compute the DWT using 16-bit integers for images that are defined with up to 10 bpp [ADA00]). With a (1, x) normalization, the mean value of the low-pass band coefficients is the same as the mean value of the original signal. Hence, the various

LL subbands that the decoder computes are low-resolution versions of the original image in approximately the same range as the original image (hence it can be displayed without renormalization).

- *Other filter features*: Other useful filter properties are complexity and coding gain. In general, it can be stated that the higher the filter length, the higher the filtering complexity (although the filter complexity can be halved through the use of symmetric filters). Coding gain is a filter parameter that quantifies the benefits associated with the use of a filter for image coding. The coding gain of a wavelet transform can be computed as the relation between the MSE of the input pixels quantized and independently encoded at a certain bit rate, and the MSE of the resulting transform coefficients quantized and encoded at the same bit rate. A comparison of coding gain for various wavelet transforms can be found in [ADA00]. Some other characteristics of a wavelet transform, such as regularity[1] and the number of vanishing moments, are employed as conditions for the design of orthonormal and bi-orthogonal wavelets [DAU98] [DAU92], but this issue is not analyzed in this thesis, while the reader is referred to the available literature [TAU02].

k	Analysis low-pass, $\{h_k\}$	k	Analysis high-pass, $\{g_k\}$
0	+ 0.602949018236360	−1	+ 0.557543526228500
± 1	+ 0.266864118442875	−2, 0	− 0.295635881557125
± 2	− 0.078223266528990	−3, +1	− 0.028771763114250
± 3	− 0.016864118442875	−4, +2	+ 0.045635881557125
± 4	+ 0.026748757410810		

k	Synthesis low-pass, $\{\widetilde{h}_k\}$	k	Synthesis high-pass, $\{\widetilde{g}_k\}$
0	+ 0.557543526228500	+1	+ 0.602949018236360
± 1	+ 0.295635881557125	0, +2	− 0.266864118442875
± 2	− 0.028771763114250	−1, +3	− 0.078223266528990
± 3	− 0.045635881557125	−2, +4	+ 0.016864118442875
		−3, +5	+ 0.026748757410810

Table 2.1: Bi-orthogonal Daubechies (9,7) filter bank with (1,1) normalization, and floating point implementation.

[1] Examples of regularity features are continuity and differentiability of the scaling and wavelet functions, and it is typically employed to measure smoothness.

2.2.4 Popular wavelet transforms for image coding

In Table 2.1 it is presented a near to optimal filter bank for image compression. This wavelet transform is known as bi-orthogonal 9/7 [COH92] [ANT92] because the low and high-pass analysis filters have 9 and 7 taps respectively (sometimes it is called Daubechies 9/7, Cohen/Daubechies/Feauveau 9/7 or simply CDF 9/7 or B9/7). From all the wavelet transforms compared in [ADA00], this one exhibits the highest code gain for 6-level decomposition. It is also considered the best transform for image compression from the filters analyzed in [VIL95]. This bi-orthogonal filter-bank is symmetric, and hence each coefficient can be computed with only 4 or 5 multiplication operations, instead of 7 or 9. In addition, due to its symmetric definition, it possesses the linear phase property. Therefore, it is non-expansive with perfect reconstruction, and it does not present visual distortion after quantization unlike non-linear phase filters, as described in the previous section.

The bi-orthogonal 9/7 transform, as it is defined in Table 2.1, is (1,1) normalized. However, this normalization is not generally employed for image coding. Tree-based encoders, like SPIHT [SAI96] (which we will study later), usually apply the $(\sqrt{2}, \sqrt{2})$ normalized version of this filter-bank, being $\{\sqrt{2}h_k\}$, $\{\sqrt{2}g_k\}$, $\{\sqrt{2}\tilde{h}_k\}$ and $\{\sqrt{2}\tilde{g}_k\}$ the applied filter-bank. With this normalization, this wavelet transform is close to orthonormal, and hence it is almost energy preserving. Thus, the same quantization step can be applied to all the subbands, simplifying the design of the bit-plane coding and the quantization process of the coefficient trees (see Chapter 4). The JPEG2000 standard also uses this filter-bank by default, being the main wavelet transform in its baseline definition (Part 1), but with a different normalization, a (1,2) normalization. In this normalization, the filter-bank is formed by $\{h_k\}$, $\{2g_k\}$, $\{2\tilde{h}_k\}$ and $\{\tilde{g}_k\}$. It is not energy preserving, therefore, each subband has to be independently quantized depending on the wavelet subband and the decomposition level. However, this type of normalization avoids dynamic range expansion at each decomposition level, and thus architectures with fewer bits per sample can be used to compute the DWT with less overflow risk.

Another simpler bi-orthogonal filter-bank is commonly adopted for fast DWT integer implementation, and lossless coding. This transform is known as 5/3, and the associated

filter-bank is described in Table 2.2. This filter-bank allows integer implementation because it can be implemented with integer division operations, although with loss of precision due to rounding. Later, we will describe how to perform reversible integer-to-integer transform with this filter-bank in spite of the loss of precision. The (1,2) normalized version of this wavelet transform also must be implemented by any standard JPEG2000 codec.

This simple wavelet transform offers an excellent opportunity to show graphically how the multiresolution analysis works. From the multiresolution analysis equations, and the 5/3 filter-bank defined in Table 2.2, we know that an analysis scaling function $\Phi(t) \in V_{n+1}$ and a synthesis wavelet function $\widetilde{\Psi}(t) \in W_{n+1}$ can be computed from scaling functions $\Phi(2t+k) \in V_n$ as:

$$\Phi(t) = \frac{1}{4}\Phi(2t+1) + \frac{1}{2}\Phi(2t) + \frac{1}{4}\Phi(2t-1) \qquad (2.26)$$

$$\widetilde{\Psi}(t) = \frac{-1}{4}\Phi(2t+1) + \frac{1}{2}\Phi(2t) + \frac{-1}{4}\Phi(2t-1) \qquad (2.27)$$

The graphical analysis is shown in Figure 2.6(b), where a 5/3 scaling function of V_{n+1} has been recursively defined as the sum of three time-scaled and shifted versions of the same basis function of V_n, multiplied by 1/4, 1/2 and 1/4, as it is given by the above equations. These three generating functions are depicted in Figure 2.6(a). In a similar manner, the wavelet function of W_{n+1} shown in Figure 2.6(d) is build from the three scaling functions of V_n depicted in Figure 2.6(c). The synthesis scaling function and analysis wavelet function can be defined in the same way, but they are not shown due to the irregular nature of the 5/3 synthesis scaling function.

k	$\{h_k\}$	k	$\{g_k\}$	k	$\{\widetilde{h}_k\}$	k	$\{\widetilde{g}_k\}$
0	+ 3/4	–1	+ 1/2	0	+ 1/2	+1	+ 3/4
± 1	+ 1/4	–2, 0	– 1/4	± 1	+ 1/4	0, +2	– 1/4
± 2	– 1/8					–1, +3	– 1/8

Table 2.2: Bi-orthogonal (5,3) filter-bank with (1,1) normalization, for integer implementation.

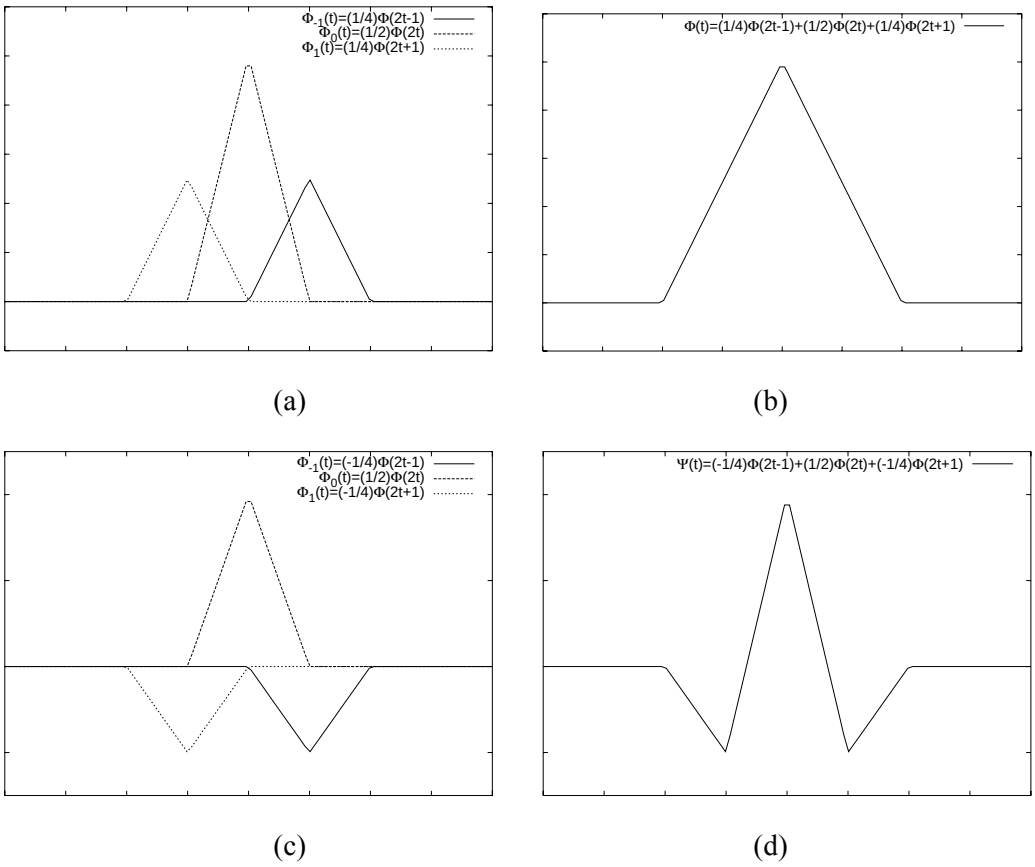


Fig. 2.6: Construction of a scaling and a wavelet function using scaling functions of the previous level for the 5/3 wavelet transform. Figures (a) and (c) show the synthesis scaling functions that are added in order to form the next-level analysis scaling (b) and synthesis wavelet functions (d) respectively.

2.3 DWT computation using the lifting scheme

One of the drawbacks of the DWT is that it doubles the memory requirements because it is implemented as a filter. A proposal that reduces the amount of memory needed for the computation of the 1D DWT is the lifting scheme [SWE96]. Despite this advantage, the main benefit of this scheme is the reduction in the number of operations needed to perform the wavelet transform if compared with the usual filtering algorithm (also known as convolution algorithm). The order of this reduction depends on the type of wavelet transform, as shown in [DAU98b].

The lifting scheme implements the DWT decomposition as an alternative algorithm to the classical filtering algorithm introduced in the previous section. In the filtering algorithm, in-place processing is not possible because each input sample is required as incoming data for

the computation of its neighbor coefficients. Therefore, an extra array is needed to store the resulting coefficients, doubling the memory requirements. On the other hand, the lifting scheme provides in-place computation of the wavelet coefficients and hence, it does not need extra memory to store the resulting coefficients. In addition, the lifting scheme can be computed on an odd set of samples, while the regular transform requires the number of input samples to be even.

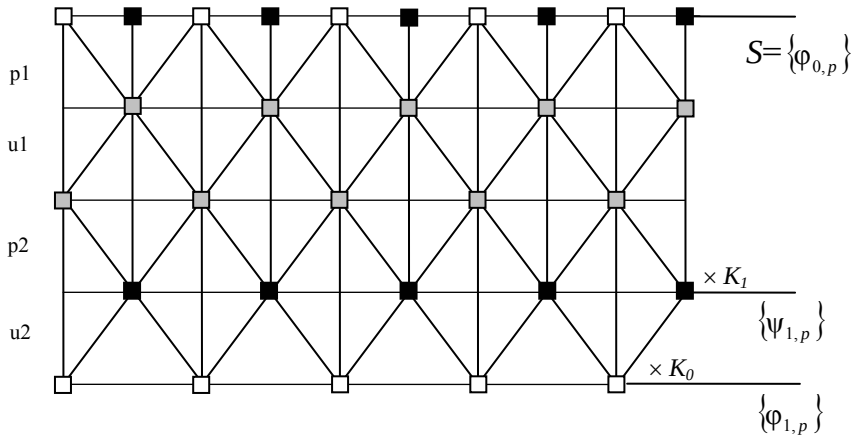


Fig. 2.7: Overview of a wavelet decomposition of an input signal using the lifting scheme for the B9/7 FWT.

Let us outline how this approach works for a one-level wavelet transform. In the lifting scheme, the wavelet coefficients are computed by means of several steps on the input samples *S* (see Figure 2.7). In the first step, the samples in odd positions (black squares in the figure) are processed from the contiguous even samples (the white ones). In particular, for the wavelet transform presented in Figure 2.7, each sample in an odd position is added to its neighboring samples (scaled by a weighting factor p1). This way, we try to predict each odd sample as a linear combination of the even ones, and the error from this estimation is stored in the same odd position, as an intermediate value represented in Figure 2.7 by the first row of grey samples. This step is called prediction step. In the second step, the even values are computed from the contiguous odd ones using a different weighting factor (u1), and it is called update step. In this manner, we compute successive prediction and update steps. The total number of steps depends on the DWT transform being computed. Finally, the odd values calculated in the last prediction step are normalized by a constant factor (K_1) to achieve the normalized high-frequency wavelet coefficients. The values from the last update

step are normalized by a different factor (K_0) to get the low-frequency scaling coefficients. Again, the normalization constant factors depend on the desired features of the transformation (close to orthonormal, preserve the dynamic range of the coefficients, etc.), although in the lifting scheme, a final scaling factor is usually necessary even for (1,1) normalization.

This one-level wavelet decomposition using the lifting scheme can be extended to any decomposition level, and to higher order wavelet transform (e.g., 2D DWT for image compression) in the same way as it has been described for the filtering algorithm.

The lifting scheme depicted in Figure 2.7 is for the bi-orthogonal 9/7 transformation. The numeric values for the weighting and normalization factors shown in this figure are given in Table 2.3. The general method to derivate weighting factors from a filter bank is described in [DAU98b] as well as the complexity reduction achieved by the lifting scheme, which is asymptotically twice as fast as the filtering algorithm when the filter length grows. In particular, for the B9/7 transform, the convolution algorithm requires 23 operations per each pair of scaling and wavelet coefficients, while using the lifting scheme only 14 operations are needed.

p_1	−1.586134342	u_1	−0.052980119
p_2	0.882911076	u_2	0.443506852

(a)

(1,1) normalization	$K_0 = \frac{1}{K}$	$K_1 = \frac{K}{2}$
(1,2) normalization	$K_0 = \frac{1}{K}$	$K_1 = K$
$(\sqrt{2}, \sqrt{2})$ normalization	$K_0 = \frac{\sqrt{2}}{K}$	$K_1 = \frac{\sqrt{2}}{2} K$

with K = 1.230174104914

(b)

Table 2.3: (a) Weighting values for prediction and update steps, (b) and various normalization factors, for the bi-orthogonal 9/7 wavelet transform.

In Figure 2.8, we present a diagram to illustrate the general lifting process. The whole process consists of a first lazy transform, one or several prediction and update steps, and coefficient normalization. In the lazy transform, the input samples are split into two data sets, one with the even samples and the other one with the odd ones, corresponding with the white

and black squares of Figure 2.7. Thus, if we consider $\{x_i\} = \{\varphi_{n,p}\}$ the input samples at a level *n*, we define:

$$\{s_i^{\,0}\} = \{x_{2i}\} \qquad (2.28)$$

$$\{d_i^{\,0}\} = \{x_{2i+1}\} \qquad (2.29)$$

Then, in a prediction step (sometimes called dual lifting), each sample in $\{d_i^{\,0}\}$ is replaced by the error committed in the prediction of that sample from the samples in $\{s_i^{\,0}\}$:

$$d_i^{\,1} = d_i^{\,0} - P(\{s_i^{\,0}\}) \qquad (2.30)$$

while in an update step (also known as primal lifting), each sample in the set $\{s_i^{\,0}\}$ is updated by $\{d_i^{\,1}\}$ as:

$$s_i^{\,1} = s_i^{\,0} + U(\{d_i^{\,1}\}) \qquad (2.31)$$

After *m* successive prediction and update steps, the final scaling and wavelet coefficients are achieved as follows:

$$\{\varphi_{n+1,p}\} = K_0 \times \{s_i^{\,m}\} \qquad (2.32)$$

$$\{\psi_{n+1,p}\} = K_1 \times \{d_i^{\,m}\} \qquad (2.33)$$

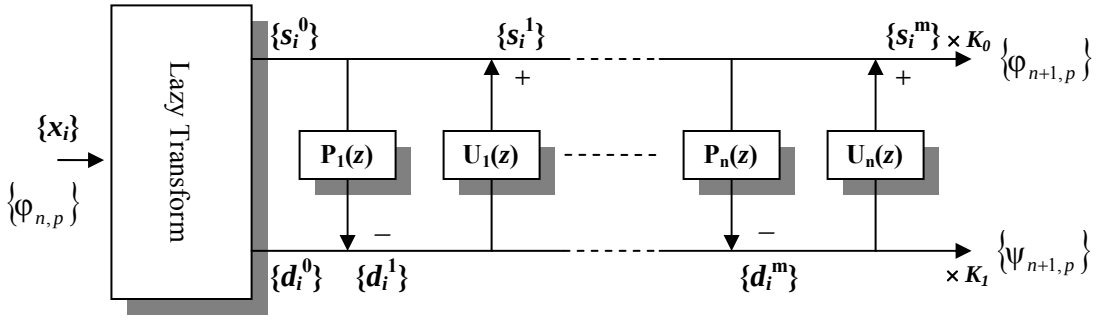


Fig. 2.8: General diagram for a wavelet decomposition using the lifting scheme.

2.3.1 Inverse wavelet transform using the lifting scheme

A nice feature of the lifting scheme is that it is formed by very simple steps, and each of these steps is easily invertible, which leads to an almost trivial inverse transform. For the inverse transform, we only have to perform the inverse operations in the reverse order. Hence, from the subsets $\{\varphi_{n+1,p}\}$ and $\{\psi_{n+1,p}\}$, we can get $\{s_i^{\,m}\}$ and $\{d_i^{\,m}\}$ simply by dividing these

coefficients by the scaling factors:

$$\{s_i^{\,m}\} = \{\varphi_{n+1,p}\} / K_0 \qquad (2.34)$$

$$\{d_i^{\,m}\} = \{\psi_{n+1,p}\} / K_1 \qquad (2.35)$$

Then, an inverse update operation can be done from these data sets as follows:

$$s_i^{\,m-1} = s_i^{\,m} - U(\{d_i^{\,m}\}) \qquad (2.36)$$

and at this moment, we can apply the inverse prediction step:

$$d_i^{\,m-1} = d_i^{\,m} + P(\{s_i^{\,m-1}\}) \qquad (2.37)$$

After *m* successive inverse update and prediction steps, we get the initial sets of even and odd samples, $\{d_i^{\,0}\}$ and $\{s_i^{\,0}\}$. Then, we can interleave these data sets to obtain the original set of samples $\{x_i\} = \{\varphi_{n,p}\}$.

This process guarantees perfect reconstruction even without signal extension. Anyway, symmetric extension is still recommended for image compression in order to avoid the introduction of high frequencies around the image boundary.

2.3.2 Integer-to-integer transform

With the above scheme, floating-point arithmetic is needed despite having integer input samples (e.g., image pixels) if the weighting factors are floating-point, and not integer or rational like in Table 2.3. However, integer implementation can be desirable if the DWT is going to be implemented in hardware architectures that only support integer arithmetic, or for lossless compression (see Chapter 1). Actually, even if rational filters are employed, the precision required to perform lossless operation with fixed-point arithmetic grows with each mathematical operation if we do not change the scheme described above.

Fortunately, the lifting scheme can be slightly modified to achieve reversible integer-to-integer wavelet transform [CAL98]. Since the lifting scheme is formed by several simple steps, the whole process can be reversible if we perform each single step in a reversible way.

For the forward transform, we have seen that each prediction step has the form:

$$d_i^{\,m} = d_i^{\,m-1} - P(\{s_i^{\,m-1}\}) \qquad (2.38)$$

In a wavelet transform for integer implementation, the prediction operation $P(\{s_i^{\,m-1}\})$ involves rational weighting factors (e.g., division by two), and hence the resulting data is not

integer. If a rounding operation is added after the prediction operation, an integer variable can be used to store the result of that operation, and hence each d_i^m can be computed from d_i^{m-1} and the $\{s_i^{m-1}\}$ set using integer values as follows:

$$d_i^m = d_i^{m-1} - \lfloor P(\{s_i^{m-1}\}) \rfloor \qquad (2.39)$$

In the inverse transform, it can be easily seen that the exact value of each d_i^{m-1} can be recovered from d_i^m and the $\{s_i^{m-1}\}$ set as follows:

$$d_i^{m-1} = d_i^m + \lfloor P(\{s_i^{m-1}\}) \rfloor \qquad (2.40)$$

Thereby, perfect reconstruction is guaranteed despite the rounding operation. The same analysis can be performed for an update operation with integer data type, being the forward update:

$$s_i^m = s_i^{m-1} + \lfloor U(\{d_i^m\}) \rfloor \qquad (2.41)$$

and the inverse update:

$$s_i^{m-1} = s_i^m - \lfloor U(\{d_i^m\}) \rfloor \qquad (2.42)$$

Although we have used the floor operator for rounding in the above equations, any other rounding operation, such as ceil or rounding to the nearest integer, can be used as long as the same operator is employed in both the forward and inverse transforms.

Finally, a reversible integer-to-integer transform can only be obtained if the normalization factors K_0 and K_1 are integer values. They cannot be rational factors given that a rounding operation in the resulting coefficients would negatively affect the reversibility of the transform.

A drawback of rounding is that the new wavelet transform is no longer linear. Hence, for a 2D wavelet transform, the reverse column-row order of the forward transform has to be used in the inverse transform to achieve perfect reconstruction (e.g., if we decompose each LL subband first by columns and then by rows, the inverse transform has to be applied first by rows and then by columns).

The 5/3 wavelet transform, which is described as a filter-bank in Table 2.2, is a typical wavelet for integer-to-integer transform, being part of the JPEG2000 standard for lossless compression. It can be computed in terms of the lifting scheme as we have described. Thus, after the lazy transform, in which the input signal $\{x_i\}$ is split into $\{d_i^0\}$ and $\{s_i^0\}$, the dual lifting is calculated as:

$$d_i^{\,1} = d_i^{\,0} - \left\lfloor \frac{1}{2}\left(s_i^{\,0} + s_{i+1}^{\,0}\right) \right\rfloor \qquad (2.43)$$

while the primal lifting is (notice the different rounding operation):

$$s_i^{\,1} = s_i^{\,0} + \left\lfloor \frac{1}{4}\left(d_i^{\,1} + d_{i-1}^{\,1}\right) + \frac{1}{2} \right\rfloor \qquad (2.44)$$

These operations can be easily performed with integer data types and integer arithmetic. For example, in C language, the two above equations can be efficiently computed as:

```
d1[i]=d0[i]-((s0[i]+s0[i+1])>>1);
s1[i]=s0[i]+((d1[i]+d1[i-1]+2)>>2);
```

Where `d0`, `d1`, `s0` and `s1` are arrays of integers, and `>>` is the right shift operator in C ($a >> b$ is equivalent to the division of a by 2^b with floor rounding). For in-place computation, `d0` and `d1` can be replaced by a single array `d`, and `s0` and `s1` by another array `s`, or even computation can be directly performed in the source array with the `s` and `d` arrays interleaved, like in Figure 2.7. However, if in-place computation is applied, the low-frequency coefficients are interleaved with the wavelet coefficients, and the subsequent wavelet processing can be non-optimal (especially in cache-based systems), requiring more careful processing. We can overcome this problem with coefficient reordering, at the cost of increasing the complexity of the algorithm.

For a lossless transform, the normalization factors K_0 and K_1 are equal to 1, achieving (1,2) normalization. Thus, the set $\left\{d_i^{\,1}\right\}$ is directly the final wavelet coefficient set, and the set $\left\{s_i^{\,1}\right\}$ is the scaling one. If (1,1) normalization is desired, K_1 should be 1/2, but in this case the wavelet transform would not be reversible.

The inverse transform to recover losslessly the original samples is given by:

$$s_i^{\,0} = s_i^{\,1} - \left\lfloor \frac{1}{4}\left(d_i^{\,1} + d_{i-1}^{\,1}\right) + \frac{1}{2} \right\rfloor \qquad (2.45)$$

$$d_i^{\,0} = d_i^{\,1} + \left\lfloor \frac{1}{2}\left(s_i^{\,0} + s_{i+1}^{\,0}\right) \right\rfloor \qquad (2.46)$$

Other reversible integer-to-integer wavelet transforms are given in [ADA00], including an integer version of the bi-orthogonal 9/7 transform. In this paper, an intensive analysis of filter features for various wavelet transforms is performed, being compared for lossy and lossless compression.

2.4 Summary

In this chapter, we introduced some theory behind the wavelet transform, focusing on image compression. The use of the wavelet transform for image compression is justified not only due to the high compactness achieved in low frequency subbands, but also because it allows image multiresolution representation, adapting to image details in a natural way. In addition, wavelet coefficients can be analyzed in both space and frequency domains.

Afterwards, from the multiresolution analysis, we derived how to perform both one-dimensional and two-dimensional DWT computation using a filter-bank. After the study of some filter characteristics, we may conclude that one of the most interesting filter features for image compression is linear phase, so as to perform a non-expansive transform with symmetric extension, and to avoid edge distortion after quantization. Since an orthogonal filter cannot be linear phase, the most popular wavelet transforms for image compression are not orthogonal but bi-orthogonal, which can be linear-phase. In particular, the most widely used filter-bank for image compression is the bi-orthogonal 9/7, defined with floating-point taps, although for lossless compression and integer transform, a different bi-orthogonal 5/3 filter is commonly employed.

Finally, we presented the lifting scheme as an alternative method to perform the DWT. It allows in-place computation, so no extra-memory is needed, and in general, it requires fewer operations to be computed. With this method, the inverse transform is easily computed reversing the order and type of operations applied in the forward one. In addition, reversible integer-to-integer transform can be easily performed by rounding each prediction and update step.

However, for the two-dimensional wavelet transform, the use of the lifting scheme shows little benefit in memory reduction because the entire image has to be kept in memory. In the DWT algorithms presented in this chapter (both filtering and lifting), an image is transformed at every decomposition level first row by row and then column by column, and for this reason, it must be kept entirely in memory. In the next chapter, we will reduce the amount of memory required to compute the two-dimensional wavelet transform.

Chapter 3

Efficient memory usage in the 2D DWT

In this chapter, the problem of reducing the memory requirements of the two-dimensional wavelet transform is tackled. For this purpose, a new algorithm to efficiently compute the 2D DWT is presented. This algorithm aims at low memory consumption and reduced complexity, meeting these requirements by means of line-by-line processing. In this proposal, we use recursion to automatically place the order in which the wavelet transform is computed. This way, we solve some synchronization problems that have not been tackled in previous proposals. Furthermore, unlike other similar proposals, our proposal can be straightforwardly implemented from the algorithm description. To this end, a general algorithm is given, which is further detailed to allow its implementation with a simple filter-bank or using the more efficient lifting scheme.

3.1 Introduction

As mentioned in the previous chapter, the two-dimensional wavelet transform is typically implemented with memory-intensive algorithms, requiring the whole image in memory to be computed. Other image transforms for image coding, such as the DCT, are applied in small block sizes and thus, a large amount of memory is not specifically needed for the transformation process.

The high memory requirement of the wavelet transform may seriously affect memory-constrained devices that deal with digital images, such as digital cameras, personal digital assistants (PDA) and mobile phones. In addition, since the memory usage for the 2D DWT

computation grows linearly with the image size, even high-performance workstations with plenty of memory can find it difficult to deal with the wavelet transform of large images. For example, in a Geographical Information System (GIS), where large digital maps are handled [UYT99], the uncompressed color map of the Iberian Peninsula (581,000 square meters approx.) needs more than 1.6 Terabytes to be stored (scale 1 pixel: 1 square meter). If a DWT-based coder is used, a prohibitive amount of memory is required to perform the regular DWT computation. Therefore, in these cases, the fact that the entire image has to be in memory to compute the DWT imposes a serious limitation for wavelet-based coding.

Another disadvantage of the usual 2D DWT algorithm is the poor memory performance in cache-based architectures due to the way memory is accessed. Recall that this transform computes the 1D transform first on each row, and then on each column. This process is successively repeated on the low-frequency subbands. Due to the organization of images in memory (usually row by row), the column access does not exploit memory locality, and hence does not take advantage of the cache memory. If we can arrange the memory access strictly on rows in the DWT computation, we will be able to improve the cache performance. We will take this approach later in this chapter.

3.1.1 Previous proposals to reduce memory usage

The simplest solution to reduce the amount of memory needed to compute the wavelet transform of an image is to split the whole image into smaller pieces, so that the DWT can be calculated on each of them separately. This approach is called image tiling and is supported by JPEG 2000. However, it presents several problems. On the one hand, we have image blocks again, and then blocking artifacts may reappear, especially when we use small tile size and high quantization. On the other hand, we do not decorrelate the entire image, but only the part that is being transformed, and then the compactness achieved is lower. In JPEG 2000, this low compactness causes the PSNR to drop by more than 1 dB at low bit rates [RAB02].

Fortunately, there are other cleverer strategies to save even more memory. One of them is to get rid of wavelet coefficients as soon as they have been calculated. With this approach, we compute all the decomposition levels simultaneously, and when a coefficient is not going to be used anymore, it is discarded (compressed in our case, or saved or processed for a different purpose).

Several proposals have dealt with low-memory DWT implementations following this

scheme. In [VIS94], it is introduced a first solution to overcome this drawback using this idea for the 1D DWT. Later, one of the first approaches to reduce memory consumption for wavelet-based image compression was done in [COS98]. The proposed algorithm includes image compression by means of zerotree coding [SHA93] [SAI96] (tree-based coding will be reviewed in the next chapter). In order to reduce memory requirements, the encoder reorders the output bit-stream so that the wavelet coefficients that represent the same area in all the subbands are placed together. This way, when the decoder receives this group of coefficients, it can compute a fragment of the inverse DWT, and produce several image lines. Once this group of lines is decoded, the memory used by these coefficients can be released and more lines can be read in the same way. This line-based algorithm was further refined in [CHR00], where (1) reduction of memory is dealt in both the forward and inverse transform (in [COS98] it is done only in the decoder); (2) the order of the coefficients is rearranged with some extra buffers to allow efficient use of memory in the encoder and the decoder; (3) zerotree coding is replaced with a new entropy coding algorithm. In the next subsection, we will describe a general line-based approach in more detail, and we will address some issues that arise from this strategy hindering its implementation.

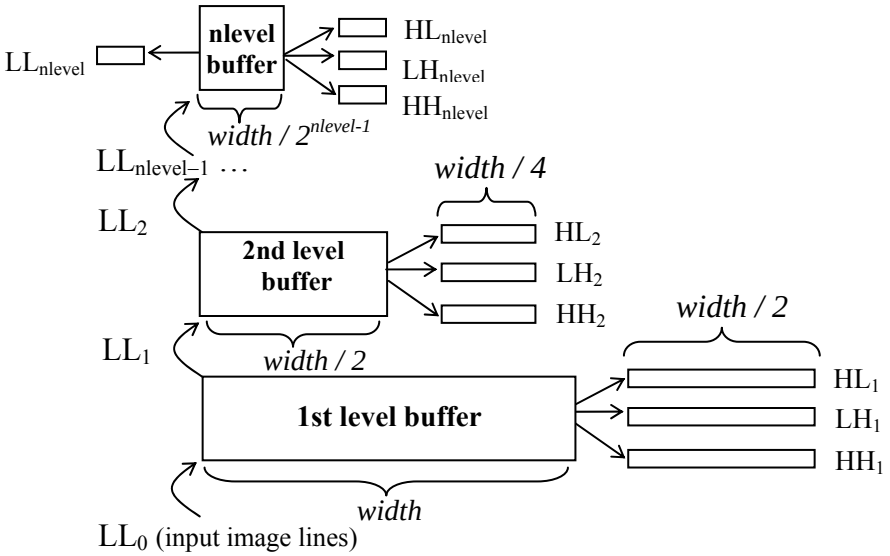


Fig. 3.1: Overview of a line-based forward wavelet transform.

3.1.2 The line-based scheme

Recall that in the regular DWT the image is transformed level by level, using the 1D DWT first on rows, and then on columns, and so it must be kept entirely in memory. In order to

keep in memory only the part of image strictly necessary, and therefore reduce the amount of memory required, the order of the regular wavelet algorithm must be changed. We cannot compute every decomposition level successively, but this computation has to be interleaved amongst the different levels. In this section, we will describe the line-based DWT from a recursive point of view instead of the iterative approach presented in [CHR00].

For the first level of the wavelet decomposition, the algorithm reads lines from the input image, one by one. On each input line, a one-level 1D wavelet transform is applied so that it is divided into two parts, representing the details and a low-frequency version of this line. Then, these transformed lines are stored in a first-level buffer. When there are enough lines in the buffer to perform a column wavelet transform, the vertical transform is computed and the first line of the HL_1, LH_1 and HH_1 wavelet subbands, along with the first line of the LL_1 scaling subband are calculated. At this moment, for a dyadic wavelet decomposition, we can compress and release the first line of every wavelet subband. However, the first line of the LL_1 subband is not part of the result but it is needed as incoming data for the following decomposition level. Afterward, when the lines in the first-level buffer have been used, this buffer is shifted twice (using a rotation operation on the buffer) so that two lines are discarded while another two image lines can be input at the other end. This way, once the buffer is updated, the same process can be repeated, obtaining the following lines of the wavelet subbands.

At the second level, the buffer is filled with the LL_1 lines that have been computed at the first level. Once this buffer is completely filled, it is processed in the very same way as we have described for the first level. In this manner, the lines of the second-level wavelet subbands are calculated, and the low-frequency lines from LL_2 are passed to the third level. As it is depicted in Figure 3.1, this process can be repeated until the desired decomposition level (*nlevel*) is reached.

In [CHR00], the description of a line-based strategy is given in an iterative way, but no detailed algorithm is described. Some major problems arise when the line-based DWT is implemented using an iterative algorithm. The main drawback is the synchronization among buffers. Before a buffer can produce lines, it must be completely filled with lines from the previous buffer, therefore they start working at different moments, i.e., they have different delays. Moreover, all the buffers exchange their result at different intervals, depending on their level.

The time in which each line is passed to the following buffer depends on several factors, such as the filter size, the number of decomposition levels, the level and number of line being computed and the image size. In a hardware implementation, with a fixed image size and a constant decomposition level, a pre-computed unit control can be employed to establish the order of the computations in the buffers for a given filter-bank. Thus, several hardware implementations of this line-based strategy have been proposed, and they can be found in the literature [ZER01] [CHA01] [DIL03] [ACH05]. However, a general case of this algorithm cannot be easily implemented in software or hardware due to the synchronization problems exposed above.

In the next section, we propose a general recursive algorithm that clearly specifies how to perform this communication among buffers, solving the synchronization problem in an automatic way by means of a recursive definition.

3.2 A recursive algorithm for buffer synchronization

In this section, we present a forward and an inverse wavelet transform algorithm (FWT and IWT) that solve the synchronization problems that have been addressed in the introduction of this chapter. In order to overcome these drawbacks, both algorithms are defined with a recursive function that obtains the next low-frequency subband (LL) line from a contiguous level. The wavelet transform is implemented first with a simple filter-bank, and then using the lifting-scheme, which is faster and requires less memory.

3.2.1 A general algorithm

Let us depict our algorithm briefly. The main task of the FWT is carried out by a recursive function that successively returns lines of a low frequency (LL_n) subband at a given level (n). Thus, the whole FWT is computed by requesting LL lines at the last level (*nlevel*). As seen in Figure 3.1, the *nlevel* buffer must be filled up with lines from the *nlevel*-1 level before it can generate lines. In order to get them, the function calls itself in a backward recursion, until the level zero is reached. At this point, it no longer needs to call itself since it can return an image line, which can be read directly from the input/output system. Although we are calculating a forward wavelet transform, we do it by means of backward recursion, since we go from *nlevel* to zero.

The function that implements this recursive algorithm is called GetLLlineBwd() (see

```
function GetLLlineBwd( level )
1) First base case:
    If there are no more lines to return at this level
        return EOL
2) Second base case:
    If level = 0
        return ReadImageLineIO( )
3) Recursive case
3.1) If buffer_level is empty
        Fill up buffer_level calling GetLLlineBwd(level-1)
3.2) else if no more lines can be read from level-1
        Start cleaning buffer_level
3.3) else
        Update buffer_level calling GetLLlineBwd(level-1)
    Get subband lines from buffer_level
    Process the high freq. subband lines {HLline, LHline, HHline}
    return LLline
end of fuction
```

```
function LowMemUsageFWT( nlevel )
    set buffer_level = empty    ∀level ∈ nlevel
    repeat
        LLline = GetLLlineBwd( nlevel )
        if (LLline!=EOL) Process the low freq. line( LLline )
    until LLline=EOL
end of function
```

Algorithm 3.1: Recursive FWT computation with *nlevel* decomposition. The backward recursive function GetLLlineBwd(*level*) returns a line from the low-frequency subband (LL_{level}) at a level determined by the function parameter. The first time that this function is called at a certain level, it returns the first line of the LL_{level} subband, the second time it returns the second line, etc. If there are no more lines at this level, it returns the EOL tag. As the n^{th} line of the LL_{level} subband is computed and returned, the corresponding n^{th} lines of the HL, LH and HH subbands at that level are also computed, processed and released.

Algorithm 3.1). This function receives a decomposition level as a parameter, calculates a line of each wavelet subband (LH, HL and HH) at that level, and returns a line from the low-frequency (LL) subband at that level. In order to get all the subband lines, the first time that this function is called at a certain level, it computes the first line of every subband at that level, the following time it computes the second one, and so forth.

When this function is called for the first time at a level, its buffer ($buffer_{level}$) is empty, and so it has to be recursively filled with lines from the previous level (case 3.1). Once a line

is input, it must be transformed using a 1D DWT before inserting it into the buffer. On the other hand, if the buffer is not empty, it simply has to be updated by discarding some lines and introducing additional lines from the previous level. We do it by means of a recursive call again (case 3.3). However, if there are no more lines from the previous level, this recursive call returns *End Of Line* (EOL). That points out that we are about to finish the computation at this level, but we still have to calculate some subband lines from the remaining lines in the buffer (case 3.2). We will give more details of each recursive case in the next subsections, since they are handled in a different way depending on whether we are dealing with a filter-bank approach or with the lifting scheme. However, in both convolution and lifting, we have computed a subband line from LH, HL and HH at the end of the recursive case. These lines are processed and released depending on the application purpose (e.g., compression), and the function returns an LL line.

Every recursive function needs at least one base case to stop recursion. This function has two base cases. The first one is reached when all the lines at this level have been read. In this case, the function returns EOL. The second base case occurs when the backward recursion gets the level zero, and then no further recursive call is needed because an image line is read and returned directly from the I/O system.

Once we have defined this recursive function, we can compute the wavelet transform with *nlevel* decomposition simply by using this function to compute the whole LL_{nlevel} subband. This is done by the function *LowMemUsageDWT*(*nlevel*) in Algorithm 3.1, which calls *GetLLlineBwd*(*nlevel*) until it returns EOL.

This algorithm can be implemented easily because the synchronization among buffers and the problem of different buffer delays are solved directly with recursion, which automatically sets the rhythm of the transformation steps. The iterative alternative is more complicated because a simple nested loop is not enough, and a complex control to trigger the operations at the correct moment for each level is required.

The inverse DWT algorithm (IWT), which is described in Algorithm 3.2, is similar to the forward one, but applied in reverse order. Thus, it carries out forward recursion, from zero to *nlevel*. It computes LL lines at a certain level from an LL line recursively computed from the following level, along with the corresponding LH, HL and HH lines, which are input from the compressed bitstream.

```
function GetLLlineFwd( level )
    updatebuff er_level = ¬updatebuff er_level
1) First base case:
    If there are no more lines to return at this level
        return EOL
2) Second base case:
    If level = nlevel
        return DecodeLLline( )
3) Recursive case
3.1) If buffer_level is empty
        Fill up buffer_level calling GetMergedLineFwd (level+1)
3.2) else if no more lines can be read from level+1 and updatebuff er_level
        Start cleaning buffer_level
3.3) else if updatebuff er_level
        Update buffer_level calling GetMergedLineFwd (level+1)
    if updatebuff er_level
        Get the first LLline from updated buffer_level
    else
        Get the second LLline from updated buffer_level
    return LLline
end of function
```

```
subfunction GetMergedLineFwd( level )
    odd_level = ¬odd_level
    if odd_level
        return { GetLLlineFwd( level ) + DecodeHLline( level ) }
    else
        return { DecodeLHline( level ) + DecodeHHline( level ) }
end of subfunction
```

```
function LowMemUsageIWT( nlevel )
    set buffer_level = empty    ∀level ∈ nlevel
    set odd_level = updatebuff er_level = false    ∀level ∈ nlevel
    repeat
        imageLine = GetLLlineFwd( 0 )
        if (imageLine!=EOL) WriteImageLineIO(imageLine )
    until imageLine =EOL
end of function
```

Algorithm 3.2: Recursive IWT computation with *nlevel* decomposition. The forward recursive function GetLLlineFwd(*level*) returns a line from a low-frequency subband as Algorithm 3.1 does, but using forward recursion. Thus, it retrieves a line of the HL, LH and HH subbands (from the compressed bitstream), and an LL line from the following level, which is computed by a recursive subfunction called GetMergedLineFwd( ). With these lines, this function can compute two new lines of the following LL subband and return them alternatively.

3.1) **if** $buffer_{level}$ is empty
    for $i = N \ldots 2N$
        $buffer_{level}(i)$ = 1DFWT(GetLLlineBwd(*level*-1))
    FullSymmetricExtension($buffer_{level}$)
3.2) **if not**(more_lines(*level*-1))
    repeat twice
        Shift($buffer_{level}$)
        $buffer_{level}(2N)$ = SymmetricExt($buffer_{level}$)
3.3) **else**
    repeat twice
        Shift($buffer_{level}$)
        $buffer_{level}(2N)$ = 1DFWT(GetLLlineBwd(*level*-1))
For *3.1)*, *3.2)* and *3.3)*
    $\{LLline, HLline\}$ = ColumnFWT_LowPass($buffer_{level}$)
    $\{LHline, HHline\}$ = ColumnFWT_HighPass($buffer_{level}$)

Algorithm 3.3: Filter-bank implementation, recursive case.

Since the recursive function goes forward, the second base case is changed from the FWT to be reached when the parameter level is equal to *nlevel*, and then a line from the low-frequency subband LL_{nlevel} is retrieved directly from the compressed bitstream.

In the recursive case, there are mainly two changes with respect to the backward function. The first modification is the introduction of a new function, *GetMergedLineFwd*(*level*), which is used to get buffer lines. This function alternatively returns the concatenation of a line from the LL and HL subbands, or from the LH and HH subbands, at a specified level. Contrary to the lines from HL, LH and HH, which are retrieved directly from the compressed bitstream, the LL line is computed recursively from the following level. The second difference is the introduction of a logical variable, $updatebuffer_{level}$, which defines whether the buffer needs to be updated or not. In the IWT, two LL lines can be computed once a buffer is full or updated. Therefore, this variable shows if the buffer is updated, and if so, another line can be computed without updating it. More details on how the recursive case is implemented are given in the next subsections.

Once the recursive function for the IWT is defined, all the image lines can be computed just by calling this recursive function with the *level* parameter set to zero, until no more lines are available, as it is shown in Algorithm 3.2.

3.2.2 Filter bank implementation

In the previous subsection, we described a general recursive algorithm for the DWT computation. Now, we will use that description to implement the DWT computation using a filtering algorithm. In this implementation, we insert lines into the buffer until a vertical low-pass and a vertical high-pass filter-bank can be applied on it. Therefore, each buffer must be able to keep $2N+1$ lines, where $2N+1$ is the number of taps for the largest filter in the filter-bank (the low-pass or high-pass filter). We only consider odd filter lengths because they have higher compression efficiency, however this analysis could be extended to even filters as well.

Since the base cases are completely defined in Algorithm 3.1, we only have to describe the recursive case. In this case, when a buffer is empty (case 3.1), its upper half (from N to $2N$) is recursively filled with lines from the previous level. Once the upper half buffer is full of lines, the lower half is filled using symmetric extension (in which the $N+1$ line is copied into the $N-1$ position, the $N+2$ into the $N-2$,…, the $2N$ is copied into the 0 position). On the other hand, if the buffer is not empty, we only have to update it (case 3.3). Therefore, we shift it one position so that the line contained in the first position is discarded and a new line can be introduced in the last position ($2N$) using a recursive call. This operation is repeated twice because we are going to use two filter-banks in each step (a high-pass and a low-pass filter). Finally, if no more lines can be computed from the previous level (case 3.2), we fill the buffer using symmetric extension again. Actually, this is the same case as in 1D DWT, in which symmetric extension is used at both ends.

In all the cases, once there are enough lines in the buffer to perform a vertical wavelet transform, the convolution process is calculated vertically twice, first using the low-pass filter and then with the high-pass filter. This way, we get a line of every wavelet subband. This whole process is described in Algorithm 3.3.

For the inverse transform, we need to change Algorithm 3.3 slightly. We only have to replace every *FWT()* function by the corresponding *IWT()*, change every $level-1$ for $level+1$, and use *GetMergedLineFwd()* instead of *GetLLlineBwd()*. Then, we can incorporate this modified version of Algorithm 3.3 as the recursive case in Algorithm 3.2. However, recall that, as described in Algorithm 3.2, we have to use the $updatebuffer_{level}$ variable to execute cases 3.2 and 3.3 only when needed, and to compute an LL line using the *ColumnIWT_LowPass(*$buffer_{level}$*)* and *ColumnIWT_HighPass(*$buffer_{level}$*)* functions

3.1) **if** $buffer_{level}$ is empty

for $i = W \ldots 0$

$buffer_{level}(i) = 1\text{DFWT}(\text{GetLLlineBwd}(\ level\text{-}1))$

Successively predict and update the lines in $buffer_{level}$,

provided that the required lines are in the buffer.

3.2) **if not**(more_lines(*level*-1))

repeat twice Shift($buffer_{level}$)

Update and predict the remaining lines in the buffer

3.3) **else**

repeat twice

Shift($buffer_{level}$)

$buffer_{level}(0) = 1\text{DFWT}(\text{GetLLlineBwd}(\ level\text{-}1\ ))$

$buffer_{level}(1) = (buffer_{level}(0) + buffer_{level}(2))p_1 + buffer_{level}(1)$

$buffer_{level}(2) = (buffer_{level}(1) + buffer_{level}(3))u_1 + buffer_{level}(2)$

…

$buffer_{level}(W) = (buffer_{level}(W-1) + buffer_{level}(W+1))u_{W/2} + buffer_{level}(W)$

For *3.1)*, *3.2)* and *3.3)*

$\{LLline, HLline\} = buffer_{level}(W) * K_0$

$\{LHline, HHline\} = buffer_{level}(W-1) * K_1$

Algorithm 3.4: Lifting implementation, recursive case.

alternatively (the low-pass filter bank is applied when $updatebuffer_{level}$ is true).

3.2.3 Implementation with the lifting scheme

The convolution implementation that has been presented in the previous subsection introduces wide benefits in memory usage, since we only keep in memory a few low-frequency lines ($2N$+1) for each decomposition level instead of the whole subbands. However, we can still reduce the amount of memory required and speed up its execution time by using the lifting scheme, which was described in the previous chapter.

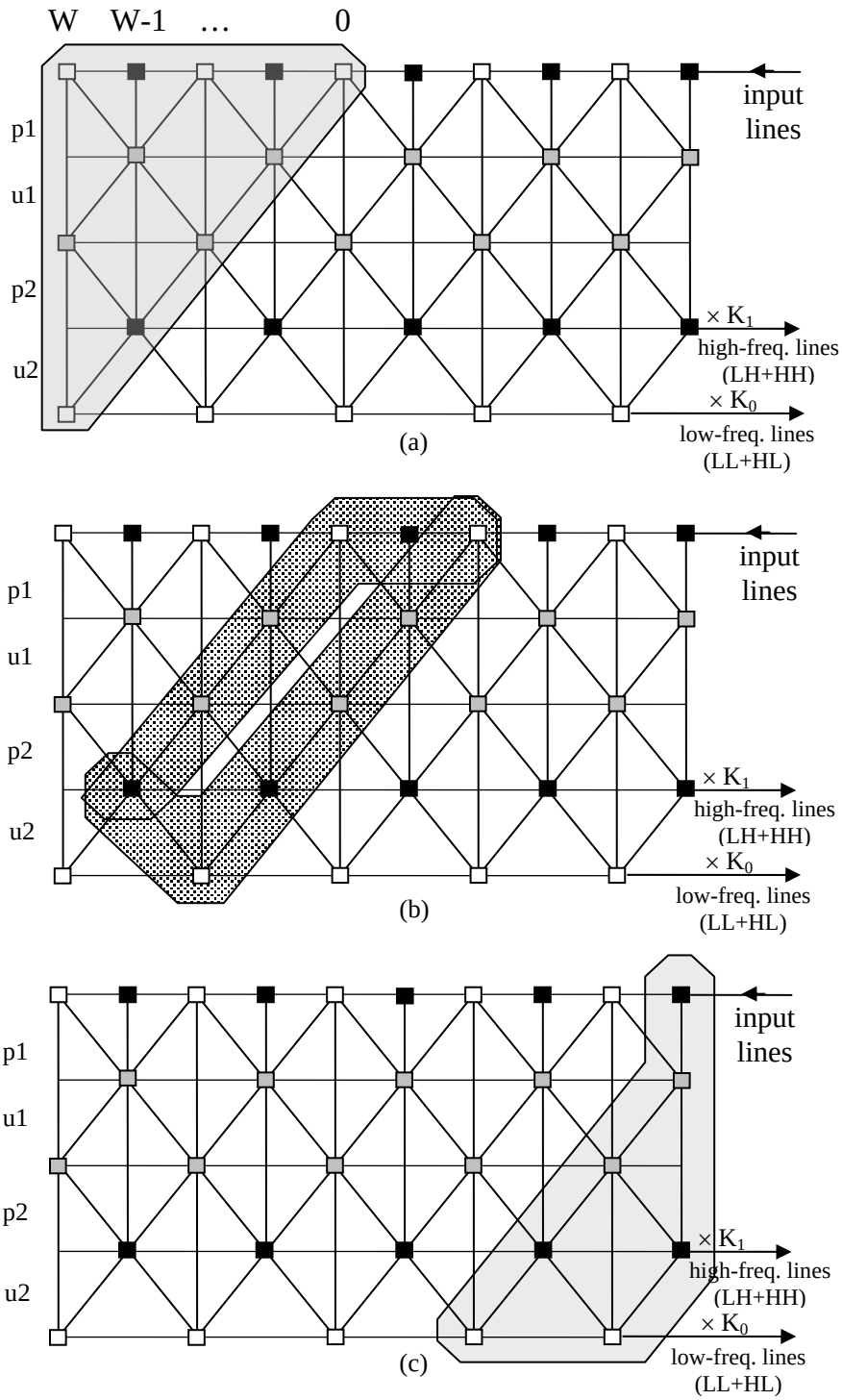


Fig. 3.2: Overview of the lifting scheme for the proposed FWT.

The main advantage of the use of the lifting scheme instead of convolution is the extra reduction of memory achieved. Let us define *W* as the total number of weighting factors (prediction and update) for a DWT. Then, the height of each buffer using the lifting scheme

has to be W+2, so it can perform the W prediction and update steps needed to compute a low and a high-frequency line in a segmented way, as we will see later. Observe that, despite computing W sample lines, we need two additional lines to calculate the first and last values. Later, we will see that these additional lines (the first and the last lines in the buffer) are read but not modified in each step. The reason why the lifting scheme can be used to save memory in our algorithm is that, in general, W+2 is lower than 2N+1 (see [DAU98b] for details) and therefore we need less lines in the buffers. For example, for B9/7, 2N+1 is 9 while W+2 is only 6. For the sake of clarity, in the rest of this section, we will consider that the number of sample values in each decomposition level is even, and so W is (although it is not difficult to extend it to the general case).

In Algorithm 3.4, we describe how to implement the recursive case of Algorithm 3.1 using the lifting scheme. In this algorithm, when the buffer is empty (case 3.1), we fill it from W to 0 (W+1 is left empty) by using a recursive call. Then, we compute successive prediction and update steps, using only the lines in the buffer. This way, we compute fewer lines in every step, since the rest of lines rely on information that still has not been input. Finally, we get a low frequency line (with the first line of the LL and HL subbands) and a high frequency line (with one line of LH and HH). The lines that have been handled in this step are shown in the highlighted area on the left of Figure 3.2(a). In this figure, every square represents a subband line, unlike Figure 2.7, in which a square represented a sample value (a pixel). The samples at the top of Figure 3.2 are input lines (or LL lines from the previous level), the grey samples in the middle of the figure are intermediate states of those lines, while those at the bottom are the concatenation of LH and HH lines (black squares), or LL and HL lines (white squares).

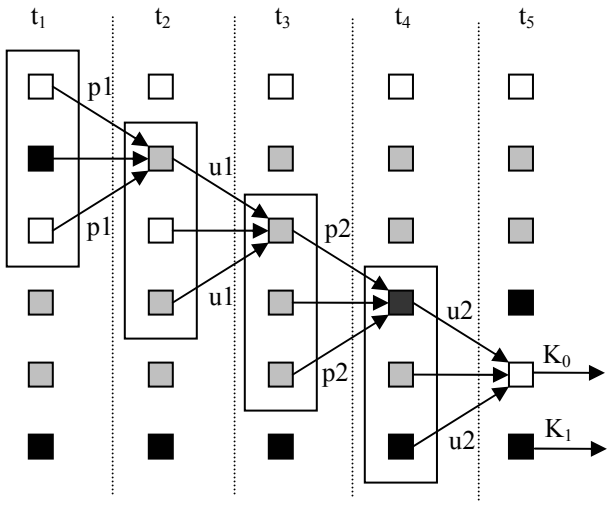


Fig. 3.3: Line processing in a buffer for the lifting scheme. The evolution of the buffer in time is shown in five steps.

At this moment, we can see in Figure 3.2(a) that all the lines in a diagonal (on the hypotenuse of the triangle that defines the left area) are intermediate states for the following lines of the wavelet subbands (i.e., are predicted or updated lines). Hence, if we introduce two more lines, and discard other two lines (recall that the line in W was processed and encoded, and W+1 was empty), we can compute two more lines in a segmented way (see case 3.3 of Algorithm 3.4). Every time that we introduce two lines in the buffer, the lines are processed as described in Figure 3.3 (for a B9/7 transform). The first column in this figure indicates the initial state, in which we have two new lines (white and black squares) at the top of the figure. Then, the new odd line is predicted from its two contiguous even lines, and this square becomes grey because it is an intermediate value storing the error of the prediction. Afterwards, we update the third line in the buffer from the contiguous even lines, and so forth. At the end of this process, we have computed two new lines, which represent four subband lines. The new high-frequency line is not released because it is necessary for the following pass. Thus, we normalize and release the new low-frequency line, and the high-frequency line that was computed in the previous pass. The computation of these four subband lines is represented in the middle of Figure 3.2(b) as the pass from the left dotted area, which represents the initial state in Figure 3.3 (first column), to the right dotted area, which is the final state in Figure 3.3 (last column).

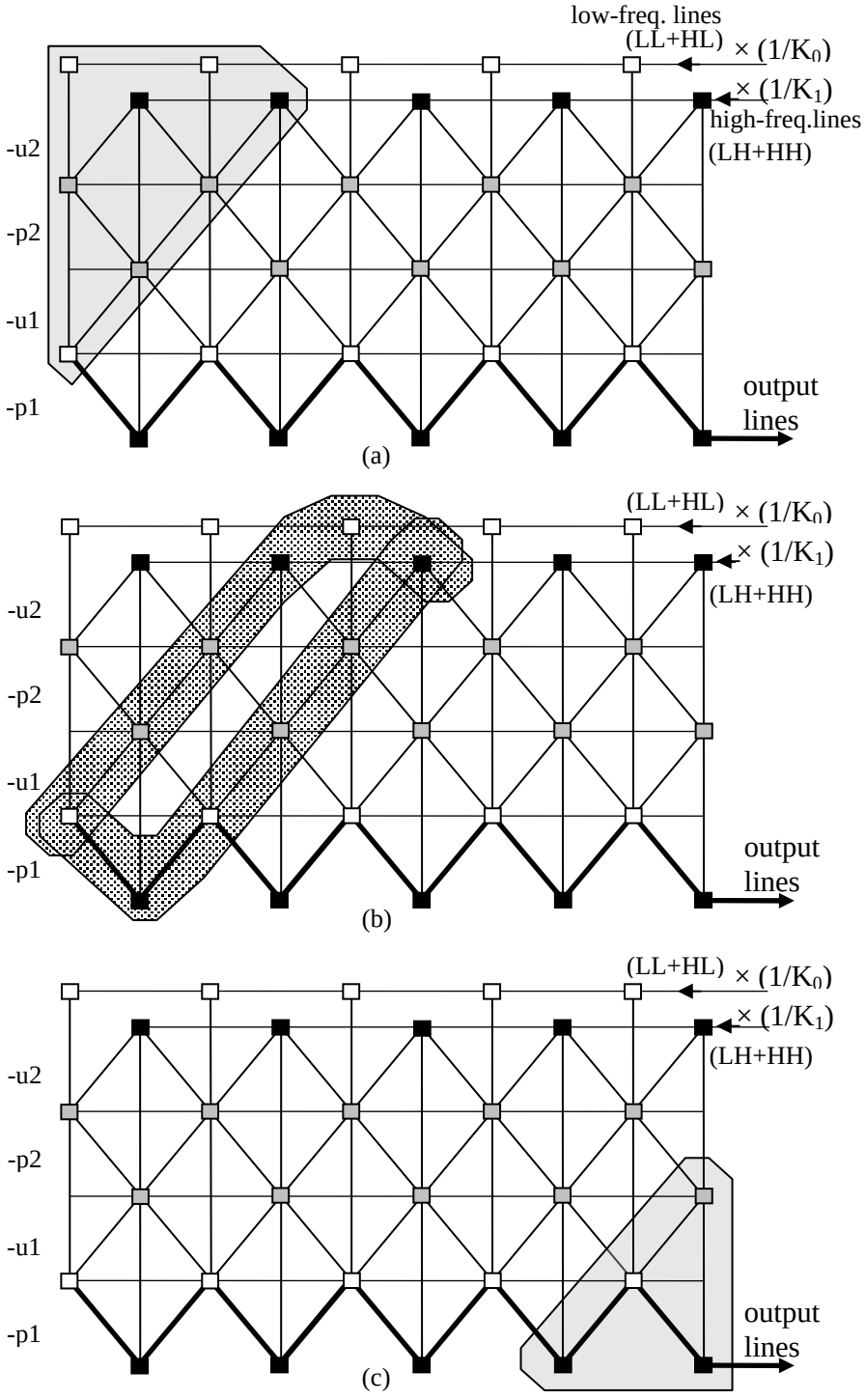


Fig. 3.4: Overview of the lifting scheme for the proposed IWT.

Finally, the highlighted area on the right of Figure 3.2(c) shows the lines that are processed when no more lines can be read from the previous level (case 3.2 of Algorithm

3.4). In this case, we use the remaining intermediate lines to generate more subband lines while we are cleaning the buffer by shifting it two positions in each call.

Although symmetric extension is not as necessary in the lifting scheme as in a filter-bank implementation, because the lifting scheme is non-expansive even if no signal extension is performed, the use of signal extension is preferred to improve the coding efficiency. A simple way to apply symmetric extension in this algorithm is to double the weighting factor when there is only a line available to predict/update another line, in other words, if the predicted/updated line is on an edge in Figure 3.2(a) or 3.2(c). The advantage of performing signal extension by doubling the weighting factors as we propose is that no extra memory is needed to place the extended part, while the result of the wavelet transform is exactly the same as though it were computed with true signal extension.

As we described in the previous chapter, for the inverse transform with the lifting scheme, we have to perform the same operations as in the forward DWT but in the reverse order. Moreover, the sign of the weighting factors have to be changed, and the scaling factors are inverted. These modifications are shown in Figure 3.4, (a) for the case 3.1, (b) for the general case 3.3, and (c) for the case 3.2. In these figures, the input data are compound subband lines (LL+HL and LH+HH interleaved), and the output data are low-frequency lines of the previous level. The IWT algorithm is similar to the one given for the forward implementation (Algorithm 3.4), but considering the same changes that were described in the filter-bank implementation. Recall that two lines are computed in every step, and thus the $updatebuffer_{level}$ variable is used to know if there is a line available in the buffer from the previous pass, or the buffer needs updating to compute two more lines. Also consider that the order and sign of the weighting factors has to be changed, i.e., instead of applying the constants as in Algorithm 3.4, ranging form p_1 to $u_{w/2}$, we have to use them from $u_{w/2}$ to p_1.

3.2.4 Reversible integer-to-integer implementation

The lifting scheme can be employed to perform a reversible wavelet transform with integer coefficients as described in the previous chapter. However, due to the rounding operator included in the transform to make possible the use of integer operations and variables, the wavelet transform is no longer linear, and hence it is important the order in which the one-dimensional DWT is applied to perform a higher order transform. Thus, a reversible transform is only achieved by following the reverse row-column order in the inverse

transform with respect to the one applied in the forward one.

Despite not having the whole image in memory, which could seem a problem at first sight, we still can take this reversible approach. In the description of the proposed wavelet transform given in the previous section, in both the forward and inverse transforms, the separable 1D transforms are performed first horizontally, when a line is input (or a compound line in the IWT), and then vertically, by applying one step of the wavelet transform in a segmented way. Thereby, for a reversible transform, the order of the inverse transform has to be changed. The horizontal 1D IWT is not applied once a compound line is read (with the GetMergedLineFwd() function) and introduced into the buffer, but it is delayed until the end of the GetLLlineFwd() function (see Algorithm 3.2). Therefore, just before the execution of the return order at the end of this function (and thus after the vertical transform), we apply the horizontal 1D inverse transform. This way, we follow the correct order (i.e., horizontal FWT, vertical FWT, vertical IWT, horizontal IWT) and the transform is fully reversible.

A consequence of the use of integer coefficients is that we can represent coefficients with smaller data types. For instance, if the wavelet transform avoids dynamic range expansion of the coefficients, we can use 16-bit integer coefficients instead of 32-bit float data types, halving the memory usage. Furthermore, if the applied transform is the bi-orthogonal 5/3 for integer transform instead of Daubechies 9/7, the W parameter mentioned in the previous subsection is reduced from 4 to 2, reducing every buffer height in two lines, and so the global memory requirements, although at the expense of coding efficiency.

3.2.5 Some theoretical considerations

The main advantage of line-based algorithms is their lower memory requirements compared with the regular wavelet transform. In the filter-bank implementation, every buffer contains $2N$+1 lines, so it needs to store $(2N+1)\times BufferWidth$ coefficients. If the image width is w, the width of the first-level buffer is w coefficients, and this width is halved at every level. So the memory requirements for all the buffers are

$$(2N+1)\times w+(2N+1)\times w/2+\cdots+(2N+1)\times w/2^{nlevel-1} \qquad (3.1)$$

coefficients, which is asymptotically (as *nlevel* approaches infinity) $2\times(2N+1)\times w$ coefficients. Memory reduction is even better when the lifting scheme is used, because only $2\times(W+2)\times w$ coefficients are needed (the relationship between the $2N$+1 and W is shown

in [DAU98b], and asymptotically $2N+1$ is twice W). We can compare these memory requirements with the regular wavelet transform, which requires $height \times width$ coefficients to be computed. Since for efficient filter banks $2 \times (W+2) < 2 \times (2N+1) << height$ (i.e., twice the buffer height is considerably lower than the image height), a line-based approach uses much less memory than the regular one.

The reduction of memory has another beneficial side effect when the algorithm is implemented in a cache-based system. The subband buffers are more likely to fit in cache memory than the whole image, and thus the execution time is substantially reduced. Moreover, we have replaced the column access of the regular wavelet transform, which clearly affects the cache performance, with a more sophisticated access, which is arranged in line buffers. In addition, the lifting scheme version reduces the computational cost of the algorithm since it performs fewer floating-point operations per sample.

A drawback that has not been considered yet is the need to reverse the order of the subbands, from the FWT to the IWT. The former starts generating lines from the first levels to the last ones, while the latter needs to get lines from the last levels before getting lines from the first ones. This problem can be solved using some additional buffers at both ends to reverse the coefficients order, so that data are supplied in the right order [CHR00]. Other simpler solutions are: to save every level in secondary storage separately so that it can be read in a different order and, if the WT is used for image compression, to keep the compressed coefficients in memory. For the sake of simplicity, we will use the last option in this thesis, although any of them could be used.

3.3 Experimental results

In order to compare the regular wavelet transform and our proposals, we have implemented them, using standard ANSI C language on a regular PC computer (with a 500 MHz Pentium Celeron processor with 256 KB L2 cache). These implementations are available at http://www.disca.upv.es/joliver/thesis.

For these test, we have used the well-known Daubechies 9/7 and bi-orthogonal 5/3 wavelet transforms presented in the previous chapter. Moreover, the coefficients for the B5/3 transform are implemented as floating-point, integer and short integer values in order to assess the effects of employing different data types for the transform computation.

In the tests, we have used the standard Lena (512x512) and Woman (2048x2560) images. With six decomposition levels using D9/7, the regular WT needs 1030 KB for Lena and 20510 KB for Woman, while the filter-bank implementation proposed in this section requires 41 KB for Lena and 162 KB for Woman, i.e., it uses 25 and 127 times less memory. Our proposal using the lifting scheme still needs less memory, requiring 26 KB for Lena and 102 KB for Woman, which means that it only requires 60% of memory with respect to the filter-bank algorithm.

If the B5/3 transform is used instead of D9/7, in the regular transform the entire image is also kept in memory and therefore it requires exactly the same amount of memory, unless short-integer data type is employed, in which case it requires half memory (short-integer variables are 16-bit while integer and float variables are 32-bit). When the lifting scheme is used along with the B5/3 transform, fewer lines need to be introduced in every buffer. As a result, the memory usage is reduced to 19 KB for Lena and 79 KB for Woman (or even to 9 KB and 39 KB respectively for short-integer coefficients).

In addition, Table 3.1 shows that our proposals are much more scalable than the usual DWT. In Table 3.1(a), we present the amount of memory needed to apply the D9/7 transform to images ranging from low-resolution VGA to 20-Megapixel, using the regular algorithm, and the proposed convolution and lifting algorithms. In all cases, floating-point arithmetic is used. Table 3.2(b) shows the same comparison, applying the B5/3 transform instead, and working with different data types.

Image size (megapixel)	Regular WT	Proposed convolution	Proposed lifting
20 (4096 x 5120)	81,980	324	205
16 (3712 x 4480)	65,013	293	186
12 (3200 x 3968)	49,647	253	160
8 (2560 x 3328)	33,319	202	128
5 (2048 x 2560)	20,510	162	103
4 (1856 x 2240)	16,266	147	93
3 (1600 x 1984)	12,423	127	80
2 (1280 x 1664)	8,340	101	64
1.25 (1024 x 1280)	5,125	81	51
VGA (512 x 640)	1,288	41	26

(a)

Image size (megapixel)	Regular WT (float and integer)	Regular WT (short integer)	Proposed lifting (float and integer)	Proposed lifting (short integer)
20 (4096 x 5120)	81,980	40,990	158	79
16 (3712 x 4480)	65,013	32,506	143	71
12 (3200 x 3968)	49,647	24,823	123	62
8 (2560 x 3328)	33,319	16,659	98	49
5 (2048 x 2560)	20,510	10,255	79	39
4 (1856 x 2240)	16,266	8,133	71	36
3 (1600 x 1984)	12,423	6,211	61	31
2 (1280 x 1664)	8,340	4,170	49	24
1.25 (1024 x 1280)	5,125	2,562	39	19
VGA (512 x 640)	1,288	644	19	9

(b)

Table 3.1: Memory requirements (KB) comparison among our proposals and the usual algorithm for various image sizes using the (a) D9/7 and (b) B5/3 transforms.

In Figure 3.5, we present an execution time comparison between our proposals and the regular DWT for D9/7. It shows that, while our algorithms display linear behavior, the regular wavelet transform approaches to an exponential curve. This behavior is mainly due to the ability of our algorithms to fit in cache for all the image sizes (162 KB and 102 KB for the 5-megapixel image with convolution and the lifting scheme respectively). On the contrary, the usual wavelet transform rapidly exceeds the cache limits (e.g., it needs 1287 KB for the VGA resolution). This figure shows that the lifting scheme implementation is about 40% faster than the convolution algorithm because fewer floating-point operations are performed, and both are faster than the usual wavelet transform, due to their better use of the cache memory.

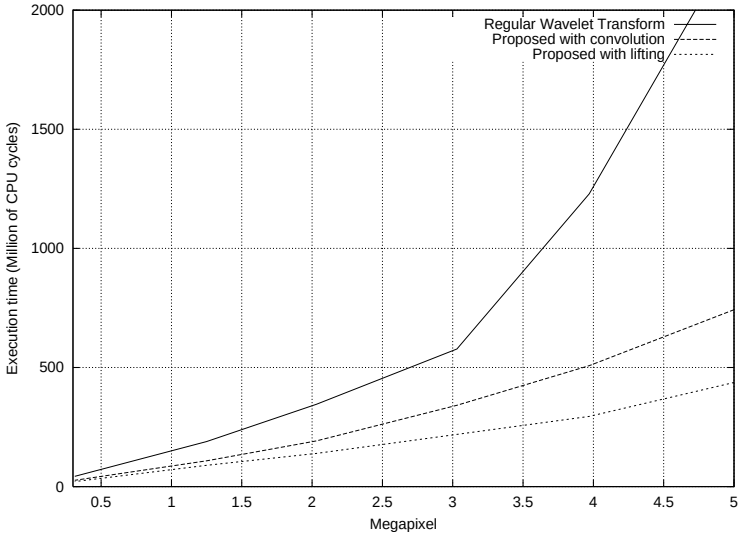


Fig. 3.5: Execution time comparison (excluding I/O time) between the regular transform and both proposals (convolution and lifting) using D9/7 and floating coefficients.

In Figure 3.6, we present an execution time comparison of different data types using our proposal with the B5/3 transform and the lifting scheme. The most noticeable result in this graph is the high execution time of the floating-point implementation that uses the floor operator, that is to say, with rounding (see section 2.3.2 for definitions), due to the temporal complexity of this operation. If we avoid using rounding (we can simply omit it if the target variable of the operation is floating-point), the execution time of the floating-point implementation is significantly reduced, although being still above the implementations with integer or short-integer coefficients. Anyway, these three implementations are faster than the proposal with the lifting scheme that is shown in Figure 3.5, because in Figure 3.6 we are dealing with a smaller filter size, requiring fewer operations per sample.

Finally, in Figure 3.7, an execution time comparison between the regular wavelet transform and the lifting proposal (both with the B/5 transform) is given for different data types. Figure 3.7(b) (integer implementation) and 3.7(c) (floating-point without rounding) are similar to Figure 3.5, displaying an exponential behavior for the regular transform. In the short-integer implementation, the lower memory usage of the regular wavelet transform prevents the exponential behavior. In the last graph, where floating-point arithmetic is used with floor operations, the cost of rounding causes both the regular and proposed transform to be highly complex.

Since the forward and inverse transform are symmetric, further experiments have shown that the IWT has similar results in memory requirements and execution time.

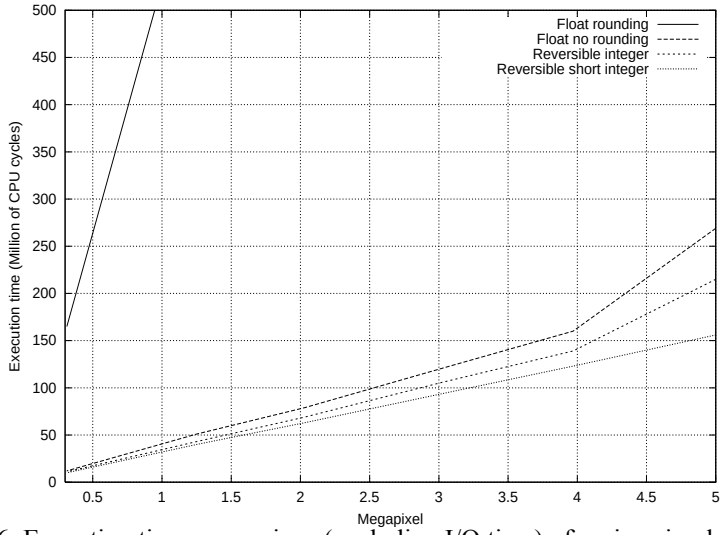


Fig. 3.6: Execution time comparison (excluding I/O time) of various implementations using float (with and without rounding), integer and short integer coefficients, with the B5/3 transform and the lifting proposal.

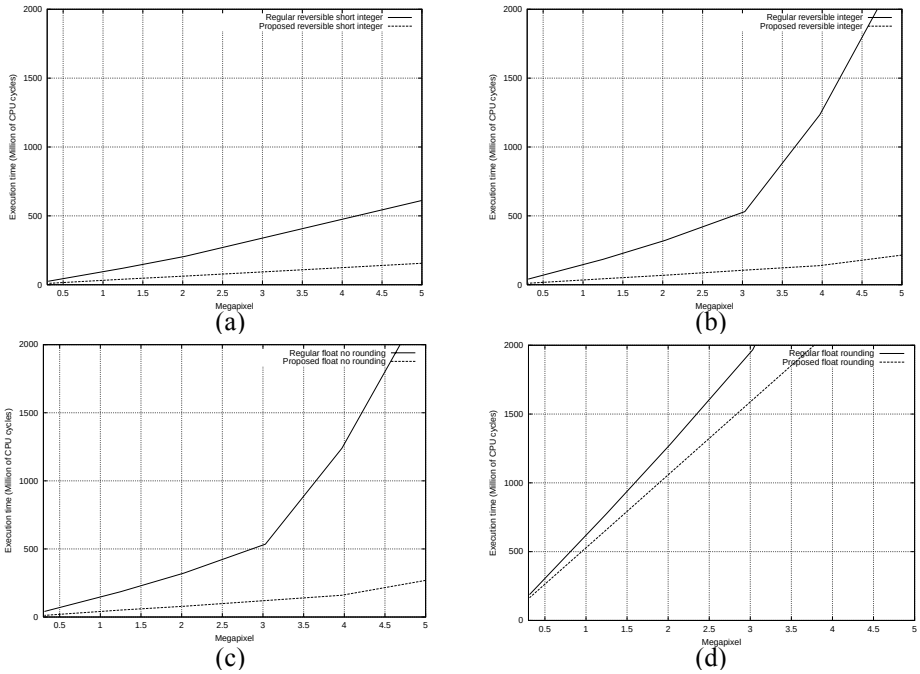


Fig. 3.7: Execution time comparison (excluding I/O time) of the regular wavelet transform and the lifting proposal, applying the B5/3 transform, with (a) short-integer coefficients, (b) integer coefficients, (c) floating-point arithmetic without rounding, and (d) floating-point arithmetic with rounding.

3.4 Summary

In this chapter, we have introduced a recursive line-by-line wavelet transform algorithm, which presents very low memory requirements and reduced execution time. The execution order of the wavelet transform is automatically placed by recursion and this way, the problems about different delay and rhythm among buffers are solved. A first general description has been further detailed to allow two different implementations (filter-bank and lifting algorithms), being the former simpler to implement but the latter more efficient in terms of complexity and memory requirements.

When a 5-Megapixel image is transformed with Daubechies' 9/7, experimental results show that the proposed wavelet transform using the lifting scheme requires 200 times less memory and it is five times faster than the regular one. Memory usage and complexity can be reduced if integer arithmetic is used with a bi-orthogonal 5/3 transform, although at the expense of less coding gain. For example, the regular reversible integer-to-integer B5/3 wavelet transform requires 260 times more memory and it is ten times slower than the

proposed new one.

Our proposals can be used as part of a compression algorithm such as JPEG 2000, speeding up its execution time and reducing its memory requirements. However, in the next chapters, we will design new wavelet encoders to be used along with this recursive wavelet transform, displaying state-of-the-art compression performance with lower complexity.

As a conclusion, the main contribution of the proposed wavelet transforms is their lower memory requirements with a straightforward implementation, which makes them good candidates for many embedded systems and other memory-constrained applications (such as digital cameras and PDAs) or for large scale images (such as those used in GIS).

Chapter 4

Coding of wavelet coefficients

The wavelet transform described in Chapter 2 is able to decorrelate the image pixels in a linear way. However, more complex dependencies exist in natural images. Therefore, we still need a good coding model, beyond simple entropy coding, in order to reduce these high-order statistical dependencies and so improve compression efficiency. The way in which wavelet coefficients are encoded establishes this model and it is the main difference among different encoders. In this chapter, we survey some of the most important wavelet-based image encoders that have been reported in the literature. We group them into tree-based and block-based, according to the coefficient structure that is used. In the performance analysis of each proposal, we not only focus on their coding efficiency but also on their complexity, since reduced complexity is one of the objectives of this thesis. Finally, we show the importance of properly tuning the parameters of a wavelet encoder, with a thorough analysis of a classic wavelet encoder, the EZW algorithm.

4.1 Introduction

In the two previous chapters, we introduced the discrete wavelet transform and various algorithms to improve its computation speed and reduce its memory requirements. However, as it was shown in Figure 1.2, the transform computation represents only the first step in transform coding, and it is employed to decorrelate the input samples (pixels in the case of image coding), achieving a less redundant smaller area of coefficients, which concentrates most energy, whereas the rest of coefficients are reduced and, in many cases, become zero or

very close to zero. Therefore, the DWT is a common point in wavelet coding, and there is almost no difference in this part from one wavelet-based encoder to another one. In this step, almost the only degree of freedom for an encoder is the wavelet family and the type of wavelet decomposition. Although most schemes are based on the B9/7 transform [ANT92] with a dyadic decomposition, other wavelet families and wavelet decompositions (such as wavelet packets [COI92] [RAM93]) have been employed [XIO98] [MEY00] [SPR02] [RAJ03].

Following the scheme depicted in Figure 1.2, after the DWT computation, an encoder must define the way in which rate/distortion is modified through quantization, and how to encode the quantized coefficients. For the latter step, the applied method seldom differs from a type of entropy coding. However, the way quantization and coding is applied defines a specific model for each wavelet encoder, and it is probably the main difference among different encoders.

Some wavelet encoders apply in combination the quantization and entropy coding steps, so as to improve coding performance by means of optimization algorithms (such as the Lagrange multiplier method [SHO84] [RAM93]), or to allow other features, like SNR scalability (e.g., by applying quantization through successive approximation [SHA93] [SAI96]). Actually, the model employed not only establishes the compression performance but also other additional features of the output bitstream. E.g., depending on the order in which coefficients are encoded, an image can be decoded with resolution or quality scalability. In addition, the availability of other features mentioned in Section 1.6 also depends on the coding model.

A wide variety of wavelet-based image compression schemes have been reported in the literature, ranging from simple entropy coding to more complex techniques such as vector quantization [DAS96] [MUK02], tree-based coding [SHA93] [SAI96], block-based coding [TAU00] [PEA04], edge-based coding [MAL92], joint space-frequency quantization schemes [XIO97] [XIO98], trellis coding [JOS95], etc.

The early wavelet-based image coders [WOO86] [ANT92] were designed in order to exploit the ability of the wavelet transform to compact the energy of an image in a simple way. They employed scalar or vector quantizers and variable-length entropy conding, showing little improvement with respect to popular DCT-based algorithms, like JPEG. In fact, in [HIL94], some early wavelet encoders were compared with JPEG, concluding that

these encoders obtained better results than JPEG only when very low bit rates were used (below 0.25 bpp for an original grey-scale 8 bpp image). However, despite a not very brilliant beginning, the DWT has been successfully employed later in the field of image coding.

In this chapter, some of the most relevant and efficient wavelet coding techniques that have been proposed recently are surveyed. We will group them into two main types of wavelet encoders: tree-based and block-based. Among the wide variety of efficient encoders available in the literature, we highlight the non-embedded proposals and the fastest coding/decoding schemes. The reason why we focus on this type of encoder is that we are interested in models with low computational requirements; thus in the following chapters, we will propose several encoders that meet these requirements through non-embedded techniques.

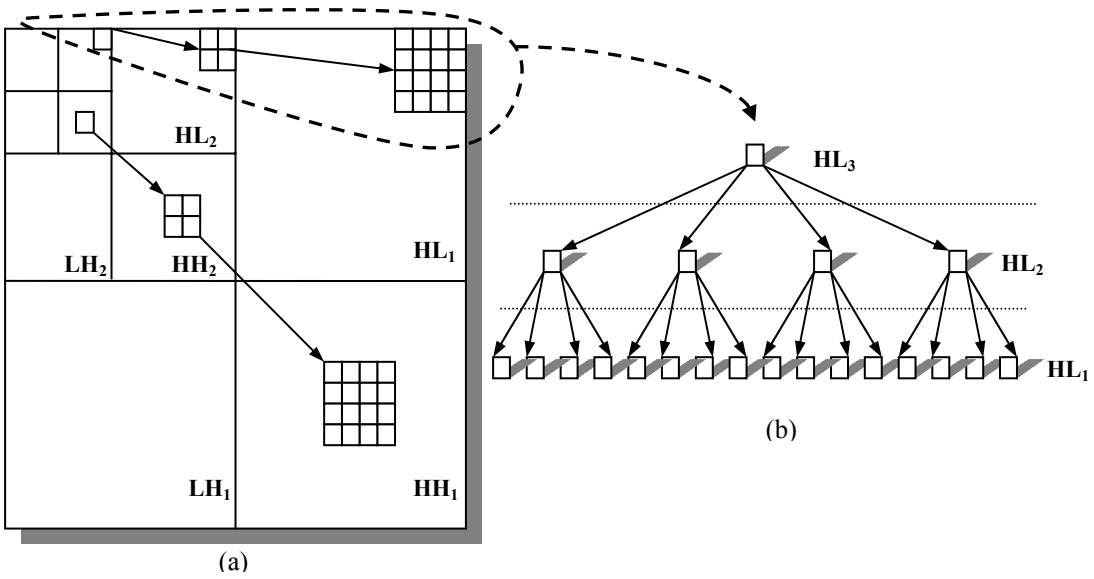


Fig. 4.1: Definition of wavelet coefficient trees. In (a), it is shown that coefficients of the same type of subband (HL, LH or HH) representing the same image area through different levels can be logically arranged as a quadtree, in which each node is a wavelet coefficient. The parent/children relation between each a pair of nodes in the quadtree is presented in (b).

4.2 Tree-based coding

4.2.1 Embedded zero-tree wavelet (EZW) coding

In the early 90s, there was the general idea that more efficient image coding would only be achieved by means of sophisticated techniques with high complexity, or by the combination

of some of them. The embedded zero-tree wavelet encoder (EZW) [SHA93] can be considered the first wavelet image coder that broke that trend. This encoder exploits the properties of the wavelet coefficients more efficiently than the rest of early techniques and thereby, it considerably outperforms their coding performance.

The EZW algorithm is mainly based on two basic ideas: (a) the similarity between the same type of wavelet subband, with higher energy as the subband level increases, and (b) a type of quantization based on a successive-approximation scheme that can be adjusted in order to get a specific bit rate in an embedded way. The former idea is exploited by means of coefficient trees, whereas the latter is usually implemented with bit-plane coding. In addition, the encoder includes an adaptive arithmetic encoder to encode the generated symbols. Although the EZW technique never became a standard, it is of great historical importance in the field of wavelet-based image coding because the aforementioned two principles were later used and refined by many other coding methods.

Let us define the coefficient trees employed in EZW. In a dyadic wavelet decomposition (e.g., the one shown in Figure 2.5), there are coefficients from different subbands representing the same spatial location, in the sense that one coefficient in a scale corresponds spatially with four coefficients in the correspondent previous subband. This connection can be extended recursively with these four coefficients and the corresponding direct descendants (sometimes called offspring) at the previous levels, so that coefficient trees can be defined as shown in Figure 4.1. Since each node in a tree has four direct descendants (except the coefficients at the first level, corresponding with the leaf nodes), this type of tree is sometimes called quadtree. Note that a quadtree (or subquadtree) can be built from each coefficient by considering it as the root node of a tree.

The key idea employed by EZW to perform tree-based coding is that, in natural images, most energy tends to concentrate at coarser scales (i.e., higher decomposition levels). Then, it can be expected that the closer to the root node a coefficient is, the larger magnitude it has. Therefore, if a node of a coefficient tree is lower than a threshold, its descendant coefficients are likely to be lower as well. In other words, the probability for all four children to be lower than a threshold is much higher if the parent is also lower than that threshold. We can take advantage of this fact by coding the subband coefficients by means of trees and successive-approximation, so that when a node and all its descendant coefficients are lower than a threshold, just a symbol is used to encode that entire branch.

The EZW algorithm is performed in several steps, with two stages per step: the dominant pass and the subordinate pass. Successive-approximation can be implemented as a bit-plane encoder, so that the method can be outlined as follows (note that an implementation perspective is taken). Consider that we need n bits to represent the highest coefficient of the image (in absolute value). Then, the first step will be focused on all those coefficients that need exactly n bits to be coded (ranging from 2^{n-1} to 2^n-1)), which are considered to be significant with respect to n. In the dominant pass, each coefficient falling in this range (in absolute value) is labeled and encoded as significant positive/negative (*sp*/*sn*), depending on its sign. These coefficients will no longer be processed in further dominant passes, but in subordinate passes. On the other hand, the rest of coefficients (those in the range $[0, 2^{n-1}[$) are encoded as zero-tree root (*zr*) if all its descendants also belong to this range, or as isolated zero (*iz*) if any descendant is significant. Note that no descendant of a zero-tree root needs to be encoded in this step, because they are already represented by the zero-tree root symbol. In the subordinate pass, the bit n of coefficients labeled as *sp*/*sn* in any prior step is coded. In the next step, the n value is decreased in one, so that we focus now on the following bit (from MSB to LSB). This compression process finishes when the desired bit rate is reached, and the decoder can partially use the incoming bitstream to reconstruct a progressively improved version of the original image. That is why this coder is called embedded.

In the dominant pass, four types of symbols need to be coded: *sp*, *sn*, *zr*, and *iz*, whereas in the subordinate pass only two are needed (bit zero and bit one). In order to get higher compression, an adaptive arithmetic encoder is used to encode the symbols computed during the dominant pass.

Due to its successive-approximation nature, EZW is SNR scalable, although at the expense of sacrificing spatial scalability. In addition, line-based wavelet transforms are not suitable for this encoder (see previous chapter), because the whole image is needed in memory to perform several image scans focusing on different bit planes and searching for zero-trees. Moreover, it needs to compute coefficient trees and performs multiple scans on the transform coefficients, which involves high computational time, most of all in cache-based architectures due to the higher cache miss rate.

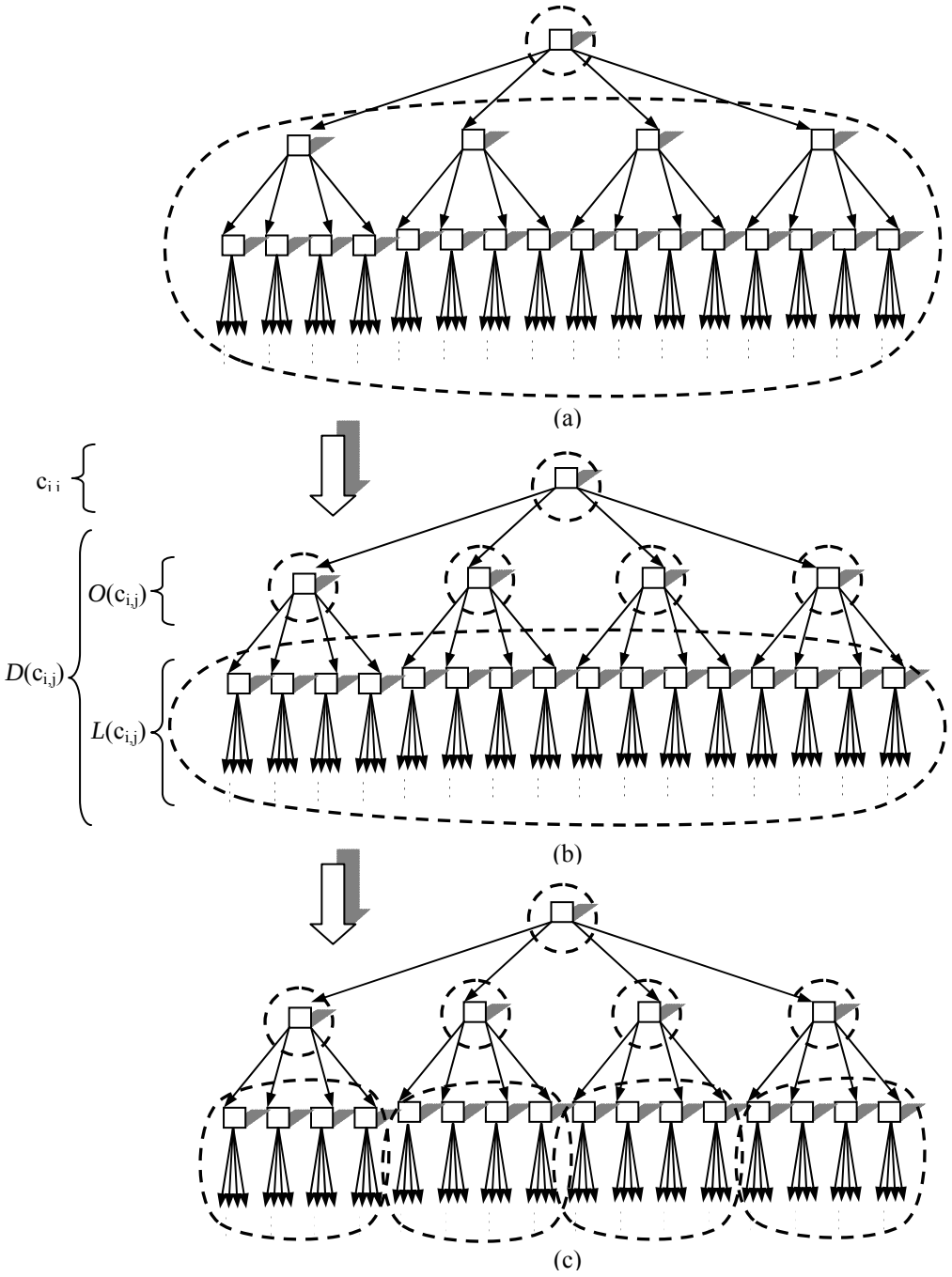


Fig. 4.2: Example of division of coefficient sets arranged in spatial orientation trees. This division is carried out by the set partitioning sorting algorithm executed in the sorting pass of SPIHT. The descendants of $c_{i,j}$ presented in (a) are partitioned as shown in (b); if needed, the subset of (b) is divided as shown in (c), and so on.

4.2.2 Set partitioning in hierarchical trees (SPIHT)

Said and Pearlman [SAI96] proposed a variation of EZW, called SPIHT (set partitioning in hierarchical trees), which is able to achieve better results than EZW even without arithmetic coding. SPIHT is based on the same principles as EZW. However, improvements are mainly due to the way it searches for significant coefficients in the quadtrees, by splitting them with a novel partitioning algorithm.

Like in EZW, SPIHT encodes the wavelet subbands in successive steps, focusing on a different bit-plane in each step. For a certain bit-plane (n), the set partitioning sorting algorithm included in SPIHT identifies the insignificant coefficients in the transformed image. This algorithm encodes the coefficient significance by means of significance tests, which query each set to know if it has at least one significant coefficient. If so, it divides that set into more subsets and it then repeats the same question, otherwise we have identified a group of insignificant coefficients with respect to the current bit plane. The result of each query is encoded with a simple binary symbol, so that the decoder can reconstruct the same groups of insignificant sets. The subsets with significant coefficients are successively divided until each single significant coefficient is identified. When all the subsets are found to be insignificant with respect to the current bit-plane, all the significant coefficients have been located, and the sorting pass is over for this step. The algorithm then encodes the corresponding bit (n) of those coefficients found significant in previous steps, which is called refinement pass. Afterwards, it focuses on the following bit-plane (n–1) and repeats the same process until the desired bitrate is reached. Note that the sorting and refinement passes of SPIHT are equivalent in concept to the dominant and subordinate passes of EZW respectively.

SPIHT uses spatial orientation trees (which are basically the quadtrees of Figure 4.1) to construct the initial set of coefficients and to establish the rules to divide them in the sorting algorithm. The notation employed in the algorithm is shown in Figure 4.2(b). For a given coefficient $c_{i,j}$, $D(c_{i,j})$ is the set of all the descendant coefficients of $c_{i,j}$. This set can be split into direct descendants (or offspring) $O(c_{i,j})$ and non direct descendants $L(c_{i,j})$.

In the SPIHT algorithm, the initial sets of coefficients are defined as $D(c_{i,j})\ \ \forall c_{i,j} \in LL_N$. The way a set $D(c_{i,j})$ is partitioned in a sorting pass is shown in Figure 4.2. Each set $D(c_{i,j})$, such as the one shown in Figure 4.2(a), is partitioned into its four direct descendants $\{d_1, d_2, d_3, d_4\} \in O(c_{i,j})$ as four single coefficients, and its non direct descendants $L(c_{i,j})$ as a

new subset (see Figure 4.2(b)). Later, if the $L(c_{i,j})$ subset has to be partitioned, it is divided into four subsets formed by $D(d_1)$, $D(d_2)$, $D(d_3)$ and $D(d_4)$, as shown in Figure 4.2(c). Each of these subsets can be further partitioned as we have just described.

The detailed coding and decoding algorithms is described in [SAI96]. In these algorithms, the sorting pass includes two lists to identify single coefficients: a list for the significant coefficients (called list of significant pixels, LSP) and another for the insignificant ones (list of insignificant pixels, LIP). On the other hand, the insignificant subsets are identified with another list (called list of insignificant sets, LIS), in which each subset can be of type $D(c_{i,j})$ or $L(c_{i,j})$ (an extra tag is needed to specify it). Note that there is no list of significant subsets because when a subset is found to have a significant coefficient, it is successively partitioned until the significant coefficient or coefficients are refined to the granularity of a single coefficient.

The coding efficiency of SPIHT can be improved by using adaptive arithmetic coding to encode as a single symbol the significance values resulting from the significance tests (queries).

The SPIHT algorithm has been considered a reference benchmark for wavelet image coding in a large number of papers. In addition, many papers have been published based on the tree-based SPITH algorithm, including video coding [MAR99] [KIM00], hyperspectral image coding [TAN03] and a generalization of the set partitioning algorithm [HON02]. A block-based version of SPIHT [PEA04] will be described later.

Due to its similarities to EZW, the features of SPIHT are the same as those mentioned for EZW in the last paragraph of the previous subsection, except for the improvements in coding performance.

4.2.3 Non-embedded tree-based coding

4.2.3.1 Space-frequency quantization (SFQ)

Not all the tree-based algorithms in the literature are based on successive quantization implemented with bit-plane coding, leading to an embedded bitstream. In [XIO97], a non-embedded tree-based image encoder called space-frequency quantization (SFQ) encoder is presented. In order to minimize distortion for a target bitrate, this algorithm relies on: (1) the construction of trees of zero-coefficients (which is considered a space quantization) and, (2) a single common uniform scalar quantization applied to the wavelet subbands (this is the

frequency quantization). The join application of (1) and (2) is performed in an optimal manner, with the Lagrange multiplier method [EVE63]. To this end, the algorithm tries to identify the optimal subset of coefficients to be discarded by encoding them as a quadtree, and the optimal step-size to quantize the rest of coefficients by applying a uniform scalar quantizer. In order to determine the best option for the space quantization, the algorithm considers not only entire quad-trees, like the one shown in Figure 4.1, but also different shapes of trees, by pruning tree branches. Information about tree pruning and the rest of quantized coefficients, along with the employed step-size, are encoded with entropy coding and sent to the decoder as part of the compressed bitstream.

Despite not being embedded, SFQ achieves precise rate control due to the use of an iterative rate/distortion optimization algorithm for a given bit rate. As a result of this algorithm, the coding performance of SFQ is slightly better than SPIHT. However, this iterative optimization algorithm is time-consuming and causes the SFQ encoder to be about five times slower than SPIHT.

4.2.3.2 Non-embedded SPIHT

In [PEA01], Pearlman introduces the discussion about the general necessity of embedding in image coding. As we have mentioned in subsection 4.2.1, bit plane coding slows the execution of both the encoder and decoder, and sometimes it gives no benefit to the application, or even worse, it is not feasible. In particular, a line-based wavelet transform cannot be employed along with bit plane coding unless further rearrangement of the bit stream is performed, needing at least the entire bit stream in memory. On the other hand, we may just want to encode an image at a constant quality. In this case, successive approximation is not strictly required, except eventually to improve coding efficiency.

The variation of SPIHT introduced in [PEA01] is to send all the bits down to a given bit plane (r) once a single coefficient has been found significant, so as to avoid the refinement passes. In this version, the coding process finishes when that bit plane (r) is reached in a sorting pass. Another option is to pre-quantize all the coefficients with a uniform scalar quantizer, and then encode all the bit-planes (again without refinement passes). The desired distortion level (or compression level) is controlled by modifying the r parameter in the first variation, or the quantization step in the second one. Note that in both approaches, the LSP list of SPIHT is no longer needed.

Although this version is faster than the original one, neither multiple image scan nor bit-

plane processing of the sorting passes is avoided. Hence, the problems addressed in subsection 4.2.1 still remain.

4.2.3.3 PROGRES (progressive resolution decomposition)

The modification of SPIHT described in the previous subsection is neither SNR nor resolution scalable. Recently, the authors of SPIHT have proposed a new version of SPIHT [CHO05] for very fast resolution scalable encoding, based on the principles of decreasing energy of the wavelet coefficients along the subband levels, and the fact that the energy is quite similar for coefficients at the same level. Since it supports resolution scalability with great speed, the authors consider that it is an excellent choice for remote sensing and GIS applications, where rapid browsing of various scales of large images is necessary.

PROGRES uses a pre-defined constant quality factor, just like the non-embedded SPIHT algorithm. In order to reduce complexity, bit plane coding is avoided and each coefficient is visited only once. Entropy coding is also avoided.

For each coefficient, the goal is to encode the sign and the bits below the most significant non-zero bit. To this end, the number of bits required for each coefficient must be known in advance. Basically, at a subband level, for each coefficient $c_{i,j}$ in that subband, the PROGRES algorithm identifies the number of bits needed to encode the highest coefficient in a SPIHT-like subset $D(c_{i,j})$ (let us call this value r), and then it encodes each coefficient contained in $O(c_{i,j})$ with that number of bits. In order that the decoder can reconstruct the original coefficients, r is also encoded. In the next subband level, PROGRES repeats the same operation for each $D(d_{m,n})\ \forall d_{m,n} \in O(c_{i,j})$. This algorithm is repeated through the successive subband levels, from the LL_N subband down to the first subband level. However, when the number of bits needed to encode a subset is found to be zero, a group of insignificant coefficients has been identified, and then this subset is no longer partitioned and encoded.

In order to improve coding efficiency, each r for a given subset is not encoded as a single value, but as the difference between that value in this subset and in its parent subset (i.e., the direct subset from which a subset stems). Since this difference is always positive (or zero), and its probability distribution is higher as it approaches zero, unary coding[1] is employed. Some other implementation details and the complete encoding algorithm are given in [CHO05].

Experimental results show that PROGRES is up to two times faster in coding and four

[1] In unary coding, a number n is represented with n ones followed by a zero.

times faster in decoding than the binary version of SPIHT (i.e., SPIHT without entropy coding). However, its coding efficiency is relatively poor, being slightly worse than binary SPIHT. The low coding performance is not only due to its lack of entropy coding, but also because it always employs the number of bits required by the highest coefficient in a subset. This problem especially affects highly detailed images. These images are more likely to have high descendant coefficients, which could cause their parents to use more bits than actually needed.

4.3 Block-based coding

A drawback of tree-based coding is that the tree structures hinder the bitstream organization for advanced scalability features. E.g., in EZW and SPIHT, when a tree is identified at a bit-plane level (for a certain quality level), we not only encode information about insignificant coefficients in the image resolution corresponding to that subband level but also about all its descendant coefficients in larger scales (i.e., in the previous subband levels). On the other hand, PROGRES encodes entire coefficients as a unit and no SNR scalability is provided.

For advanced scalability, we would like to be able to rearrange the generated bitstream to attain both spatial and SNR (quality) scalability. The former can be attained by encoding whole coefficients, subband-by-subband, in decreasing order of level, while the latter can be implemented with bit-plane coding, and it is attained by encoding the bit-planes according to their importance for distortion reduction (namely, from the most significant bit-plane to the least one). A combination of these techniques provides both scalabilities at the same time.

None of the tree-based coding techniques described so far are suitable to provide advanced scalability. To this end, we need to remove the inter-subband dependency introduced by the coefficient trees.

Another disadvantage of tree-based coding is that errors propagate through subbands, due to the inter-band dependency. E.g., in EZW, if a zero-tree root symbol is not correctly decoded due to an error in the bitstream (for example, a transmission error), this error is not only reflected in that subband level but also in all the descendant levels.

In order to overcome the above drawbacks, new coding techniques have been proposed based in coding independent coefficient blocks. Due to the lack of dependency among blocks, these methods ease the bitstream reorganization and avoid error propagation. However, the coding efficiency decreases because only the intra-subband dependency is exploited, and not

the inter-subband dependency. In fact, the actual dependency granularity is even finer, since each block is usually encoded with an independent coding model, not only to improve robustness but also to allow random access when decoding. In addition, it is a known fact that statistics of wavelet coefficients vary noticeably from one spatial region to another (i.e., natural images are non-stationary), and therefore it makes sense to encode each subband region with a different statistic model.

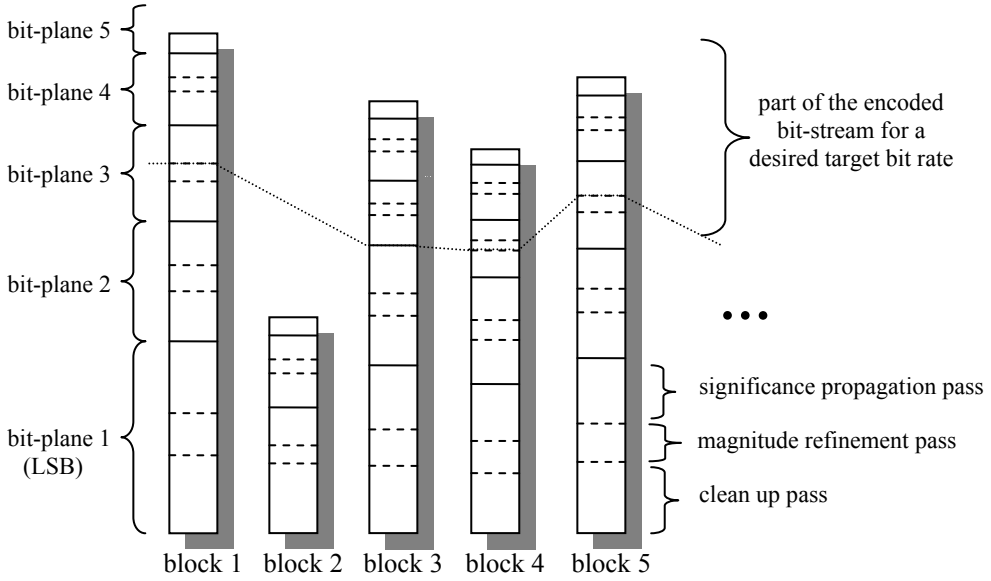


Fig. 4.3: Example of block coding in JPEG2000. In tier 1 coding, each code-block is completely encoded bit-plane by bit-plane, with three passes per bit-plane (namely signification propagation, magnitude refinement and clean up passes). Only part of each code-block is included in the final bitstream. In this figure, the truncation point for each code-block is pointed out with a dotted line. These truncation points are computed with an optimization algorithm in tier 2 coding, in order to match with the desired bit rate with the lowest distortion.

4.3.1 Embedded block coding with optimized truncation (EBCOT)

The EBCOT [TAU00] encoder is certainly the most important block-based wavelet encoder reported in the literature. This encoder is a refined version of the layered zero coding (LZC) technique proposed by Taubman and Zakhor in [TAU94]. The importance of EBCOT lies in the fact that it was selected to be included as the coding subsystem of the JPEG 2000 standard [ISO00]. EBCOT achieves most requirements of JPEG 2000, such as a rich embedded bitstream with advanced scalability, random access, robustness, etc., by means of block-based coding for the reasons given above. Furthermore, the decrease in coding efficiency caused by the lack of inter-band redundancy removal is compensated by the use of more contexts in the arithmetic encoder, a finer-granularity coding algorithm (with three

passes per bit-plane instead of two), and a post-compression rate distortion (PCRD) optimization algorithm based on the Lagrange multiplier method.

Due to the importance of EBCOT in the JPEG 2000 standard, we will describe it in some detail. For a more complete and general description, there are many other references such as [TAU02], [ACH05], [RAB02] or even the standard document [ISO00]. Note that the EBCOT algorithm originally published by Taubman in [TAU00] was slightly changed for the JPEG 2000 standard in order to reduce complexity and other issues. We will focus on this adapted version.

After applying the DWT to the image, the EBCOT algorithm encodes the wavelet coefficients in fixed-size code-blocks. In this first step, called tier 1 coding, each code-block is completely and independently encoded, getting in this manner an independent bitstream for each code-block. Then, in the second step, tier 2 coding, fragments of bitstream of each code-block are selected to achieve the desired target bitrate (rate control) in an optimal way (i.e., minimizing distortion), and it is arranged in such a way so that the selected scalability is accomplished.

Prior to EBCOT, a uniform scalar quantization with deadzone is applied to the wavelet coefficients. All the code-blocks in the same subband are quantized with the same step-size so that blocking artifacts are avoided. Therefore, in general, this quantization has little rate control meaning, which is performed later in tier 2 coding. Rather, it is used to balance the importance of the coefficient values (recall that the DWT employed in JPEG 2000 avoids dynamic range expansion but is not energy preserving, as described in see Section 2.2.4), and in a practical way, to convert the floating point coefficients resulting from most wavelet transforms (e.g., the one described in Table 2.3) into integer data. Another way to select the quantizer step size is depending on the perceptual importance of each subband to improve visual quality based on the human visual system [ALB95] [ZEN02] [MAR02].

Regarding the code-block size, the total number of coefficients in a block should not exceed 4096, and both width and height must be an integer power of two. Thereby, the typical code-block size is 64x64, although other smaller sizes can be used (e.g., for memory saving or complexity issues). Of course, once a block size is determined, smaller code-blocks can appear on the subband boundary or in subbands smaller than a regular block.

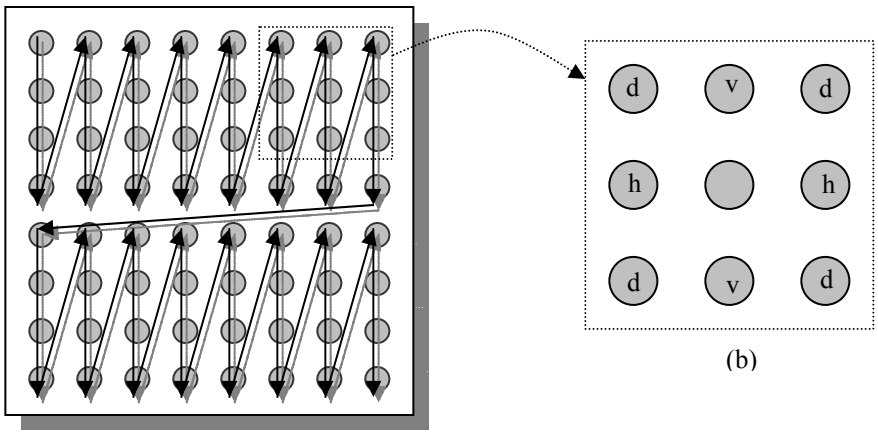


Fig. 4.4: (a) Scan order within an 8x8 code-block in JPEG2000, and (b) context employed for a coefficient, formed by its eight neighbor coefficients (two horizontal, two vertical, and four diagonal)

4.3.1.1 Block coding: tier 1 coding

Once the wavelet subbands are divided into blocks, an independent bitstream is generated from each code-block in the tier 1 coding stage. Each bitstream is created with a special adaptive binary arithmetic encoder with several contexts called MQ-coder [SLA98]. The MQ-coder is a reduced-complexity version of the usual arithmetic encoder [WIT87], limited to coding binary symbols. The JPEG 2000 standard document [ISO00] gives a detailed flowchart description of this encoder.

In this stage, each code-block is encoded bit-plane by bit-plane, starting from the most significant non-zero bit-plane. For each bit-plane, several passes are given in order to identify the coefficients which become significant in this bit-plane, and to encode the significant bits of those coefficients found significant in previous bit-planes (see Section 4.2.1 for a definition of significant coefficient). This working philosophy is shared by many others well-known encoders like EZW and SPIHT. However, unlike these encoders, three passes (instead of two) are given for each bit-plane[1]. In the first pass, called significance propagation pass, the significance of the coefficients that were insignificant in previous bit-planes but are likely to become significant in this bit-plane is encoded. Then, in the second pass, called magnitude refinement pass, a significant bit is encoded for each coefficient found significant in a previous bit-plane. Finally, the significance of the rest of coefficients (i.e., those that were insignificant and are likely to remain insignificant in this bit-plane) is encoded in the third pass, called clean up pass.

[1] The original EBCOT algorithm [TAU00] had four passes instead of three.

In tier 2 coding, the bitstream resulting from several contiguous full passes are selected from each code-block to build the final bitstream. Therefore, the bitstream generated from each pass is the lowest granularity for the final bitstream formation. In each code-block, the point in which its bitstream is truncated to contribute to the final bitstream for a given bit rate is called truncation point. Figure 4.3 illustrates the encoding process and gives an example of truncation points.

The order of the passes has been decided according to their contribution to rate/distortion improvements, so that a pass that is more likely to introduce more reduction of distortion with lower rate increase is encoded in first place. Of course, after encoding the three passes, the same reduction of distortion and the same bit rate is reached independently of the order of the passes. However, the proposed order yields more benefits if the truncation point is not at the end of a bit-plane coding (i.e., it is not between a clean up pass and a significance propagation pass), but in the middle of it.

If we compare this algorithm with EZW or SPIHT in broad terms, we see that the main difference (apart from the lack of trees) is that the pass employed to identify new significant coefficients (called dominant pass in EZW and sorting pass in SPIHT) has been split into two passes in order to have more passes from which to choose a truncation point.

For implementation convenience, the order in which coefficients are scanned in a code-block is in stripes formed by columns of four coefficients, as shown in Figure 4.4(a).

Let us see more details of each coding pass. In the significance propagation pass, a coefficient is said to be likely to become significant if, at the beginning of that pass, it has at least one significant neighbor. Certainly, this condition does not guarantee that it will become significant in this bit-plane, and therefore its significance still has to be encoded. In order to improve coding efficiency, nine contexts are used according to the significance of its eight immediate neighbors (see Figure 4.4(b)). The exact context assignment, mapping from the 2^8–1 possible contexts to nine contexts, can be found in [TAU00]. In addition, when a coefficient eventually becomes significant, its sign is also arithmetically encoded with five different contexts.

In the case of the magnitude refinement pass, a significant bit is arithmetically encoded with two contexts if it has just become significant in the previous bit-plane (i.e., it is the first bit encoded for this coefficient). For the rest of bits, they are considered to have even distribution and thereby another single context is used without dependence of the neighboring

values.

The clean up pass is implemented in a similar manner to the signification propagation pass, with the same nine contexts employed to encode the significance of a single coefficient. However, the clean up pass includes a novel run mode, which serves to reduce complexity, rather than improve coding efficiency. Observe that most coefficients are insignificant in this pass, and therefore the same binary symbol is encoded many times. We can reduce complexity if we take advantage of this fact and reduce the number of encoded symbols. To this end, when four coefficients forming a column have insignificant neighbors, a run mode is entered. In this mode, we do not encode single coefficients but a binary symbol that specifies if any of the four coefficients in a column is significant. This binary symbol is encoded with a single context.

Note that, for the most significant non-zero bit-plane (i.e., the first bit-plane that is encoded), neither a significance propagation pass nor a magnitude refinement pass is performed, because there is no previous significant coefficient (see example in Figure 4.3).

Finally, it is also worth to mention that, from the above description, we can deduce that the MQ-coder must be able to support (at least) eighteen contexts.

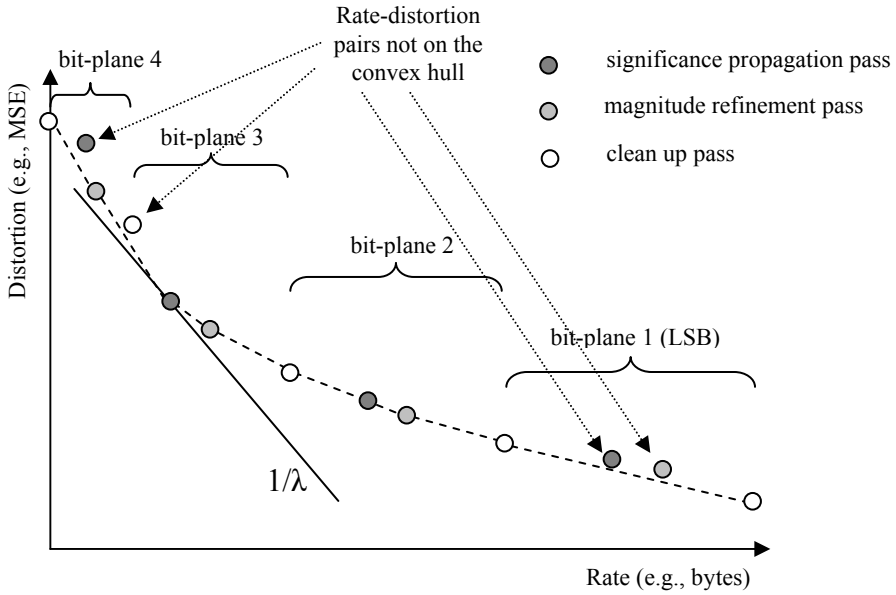


Fig. 4.5: Example of convex hull formed by distortion-rate pairs from the block 1 of Figure 4.3. In a convex hull, the slopes must be strictly decreasing. Four rate-distortion pairs are not on the convex hull, and therefore they are not eligible for the set of possible truncation points. A line with a slope of $1/\lambda$ determines the optimal truncation point for a given value of λ.

4.3.1.2 Bitstream organization: tier 2 coding

In tier 2 coding, the bitstreams generated from each code-block are multiplexed using a specific file format to accomplish the desired scalability. Rate control tasks are also performed in this second stage.

In order to determine the optimal truncation point in each code-block for a desired bit rate, EBCOT proposes a post-compression rate distortion (PCRD) optimization algorithm, which is basically a variation of the Lagrange multiplier method [EVE63]. This algorithm computes a convex hull (where slopes must be strictly decreasing) for each code-block from a set of distortion-rate pairs (see Figure 4.5 for an example of convex hull). Each pair defines the contribution of a coding pass to reduce image distortion (e.g., measured as MSE reduction) and the cost of that pass (e.g., the number of bytes required to encode that pass). For an optimal bitstream formation, the rate-distortion pairs in the interior of the convex hull cannot be selected as truncation points.

Given the set of convex hulls for each code-block, an optimal bitstream can be achieved as follows. Consider a factor λ that defines a straight line with $1/\lambda$ slope. The optimal truncation point for each convex hull is given by the point to which that line is "tangent-like"[1]. In other words, it is the point at which the rate/distortion slope changes from being greater than $1/\lambda$ to less than it (see example in Figure 4.5). In this way, we can compute an optimal bitstream by calculating a truncation point for each code-block with a given λ. However, no rate control is performed. In order to achieve a target bit rate, the value of λ is iteratively changed and the optimal set of truncation points are recomputed with each value of λ. From all the sets of truncation points iteratively computed that do not exceed the desired bit rate, the one that yields the highest distortion reduction is selected. In other words, we select the largest bitstream that does not exceed the target bitrate.

Quality (SNR) scalability can be achieved if this rate control algorithm is executed several times, once for each partial target bitrate (R_1, R_2… R_n). Therefore, the selected coding passes that optimally lead to a bitrate R_1 are said to form the quality layer 1; then, the added coding passes that lead to a bitrate R_2 form the quality layer 2, and so on. In this way, EBCOT produces an embedded bitstream, but with a coarser granularity than the one of EZW

[1] Formally speaking, the given convex hulls are not curves and then we cannot consider a line tangent to it. Here, we actually mean a line that touches a convex hull and does not intersect it. Note that in the case of a curve, there is only a line tangent to each point, whereas in our convex hulls, there are many "tangent-like" lines for each possible truncation point.

and SPIHT. On the other hand, for resolution scalability, we just have to arrange the selected code-blocks depending on the subband level, from the LL_N to the first-level wavelet subbands. A wide variety of types of scalability is accomplished by combining various quality layers and the suitable code-block arrangement in the final bitstream.

4.3.1.3 Performance and complexity analysis

Although EBCOT only exploits intra-block redundancy, it generally performs as well as SPIHT, or even better than it, in terms of coding efficiency, mainly due to (1) the use of more contexts, (2) the introduction of a third pass to encode the most important information in first place, and (3) the PCRD optimization algorithm. In addition, if we consider artificial images or highly detailed natural images, EBCOT clearly outperforms SPIHT, because in this type of image, SPIHT can establish fewer coefficient trees, and also due to the use of more contexts in EBCOT, which allows it a better and more precise adaptation of its probability model.

Let us perform a complexity analysis of EBCOT. Recall that the main complexity problem in SPIHT is introduced by bit-plane coding. Nonetheless, although both EBCOT and SPIHT use bit-plane coding, EBCOT avoids the locality problems that increase the cache miss rate by encoding an image block-by-block. Moreover, the set of code-block bitstreams is more likely to fit into cache, and therefore further post-processing does not cause so many cache misses. In spite of this, the EBCOT algorithm can be considered more complex than SPIHT (except for very large images in cache-based systems). There are several reasons for this. First, bit plane coding is still present, and for each bit-plane, it must be performed for all the coefficients in a block. Compare it with EZW and SPIHT, where the coefficients in a tree are neither encoded nor scanned. Second, the significance analysis is more complex in EBCOT, since more contexts are used. Third, in a regular implementation of EBCOT, each coefficient is fully encoded, bit-plane-by-bit-plane, despite that fact that some bit-planes will not be included in the final bitstream due to rate control restrictions, although some advanced implementations of JPEG 2000 perform a conservative heuristic for incrementally estimating the number of coding passes that will be included in the final bitstream, and determine those bit-planes that do not need to be computed. Finally, the PCRD optimization algorithm is very time-consuming due to its iterative nature.

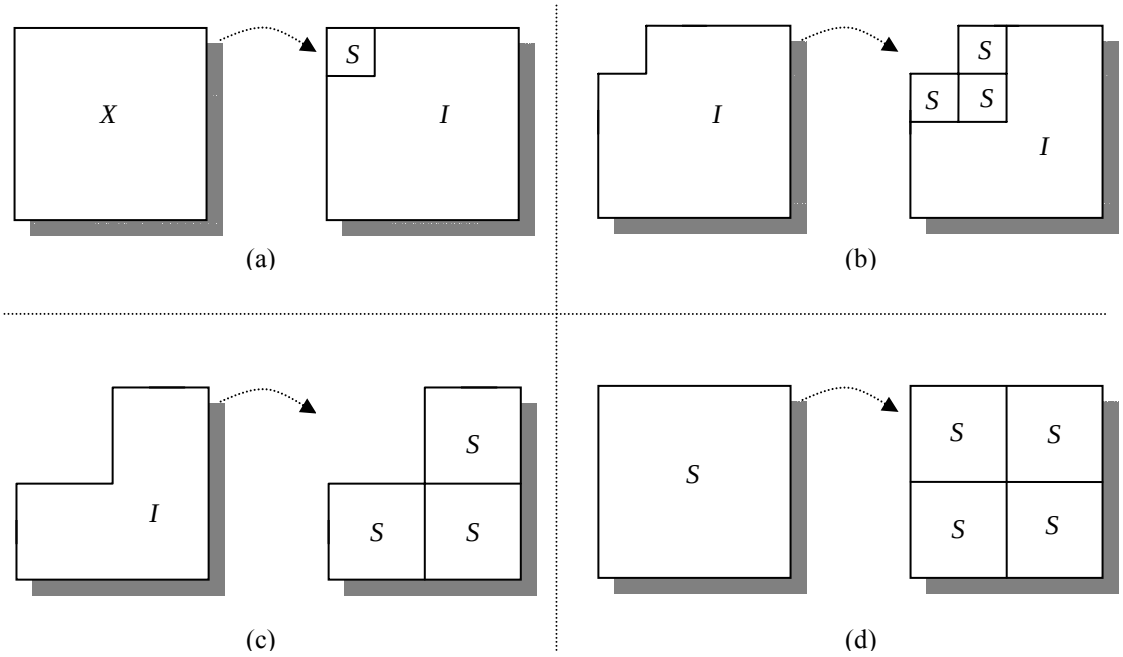


Fig. 4.6: Set division in SPECK. (a) A transformed image is initially split into two sets, one of type *S* and another of type *I*. A set of type *I* is subsequently divided into four sets: three sets of type *S*, and one set of type *I* with the remaining coefficients, as shown in (b), except for the last set of type *I*, in which the remaining set of type *I* is an empty set (see (c)). This type of division is called octave-band partitioning. Finally, a set of type *S* is subsequently divided into four sets of type *S* as shown in (d).

4.3.2 Set Partitioning Embedded Block (SPECK)

In Section 4.2.2 we described SPIHT, which is an image encoder based on the successive division of coefficient sets according to their significance with respect to a threshold, which is determined by the current bit-plane. Since the partitioning process is carried out according to the significance of the coefficients, it is said to be based on a significance test. In SPIHT, the initial set of coefficients and the consequent rules to divide them make use of a typical quadtree structure, which is characteristic of tree-based coding. The SPECK algorithm [ISL99] [PEA04] is similar to SPIHT, in the sense that it also performs significance tests to identify significant coefficients in a set, but it differs from SPIHT in that it makes use of sets in the form of blocks instead of quadtrees. Therefore, it can be thought of as a block-based version of SPIHT.

Instead of trees, in SPECK there are two different types of coefficient sets: sets *S* and sets *I*. The sets of type *S* form a rectangular area of coefficients, while the sets of type *I* are a rectangular area in which a smaller rectangular portion of this area has been removed from

the top left corner. Examples of *S* and *I* sets can be found in Figure 4.6.

Let us see the partition rules for SPECK. First, a transformed image *X* is initially partitioned into two coefficient sets: a set of type *S*, which corresponds to the low-frequency subband (LL), and the rest of wavelet subbands, forming a single set *I*. This initial partition is presented in Figure 4.6(a). Then, as in SPIHT, the SPECK algorithm is executed in successive sorting and refinement passes, going from the most significant bit to the zero bit-plane, until all the coefficients are encoded or the target bitrate is reached. However, the main difference between SPIHT and SPECK lies in the partitioning algorithm employed in the sorting pass.

In SPECK, when a set *S* is found to be significant in the current step (in other words, it has at least one significant coefficient), it is further partitioned into four sets of type *S*, and the significance of each new subset is encoded. The partition of a set *S* can be seen in Figure 4.6(d). This process is repeated on each new set *S* that remains significant, until each single significant coefficient is located (when a pixel-level is reached).

After all the sets of type *S* are refined until they are left insignificant, the significance test is applied on the set *I*. If the set *I* is significant, it is partitioned into four sets: three sets of type S, which corresponds to the following wavelet subbands, and a set *I* containing the remaining wavelet subbands. Therefore, for an N-level wavelet transform, the first time that the set *I* is divided, the resulting sets *S* correspond to the LH_N, HL_N and HH_N wavelet subbands, and the remaining coefficients form the new set *I*. Then, the second time that the set *I* is partitioned, the new sets *S* are LH_{N-1}, HL_{N-1} and HH_{N-1}, and so on, until the set *I* is formed by the LH_1, HL_1 and HH_1 subbands. At this moment, the set *I* is partitioned into only three sets *S*, because there are no more wavelet subbands that can be assigned to the set *I* (in other words, we can consider the set *I* empty). The partition of the set *I* is shown in Figures 4.6(b) for the general case, and 4.6(c) for the final partition. Note that when a set *I* is found to be significant and as a result it is subsequently divided, the significance of the four new sets is encoded, and if any of the new set *S* is significant, it should be further partitioned as well, by following the partitioning rules of the sets of type *S*.

The way the set *I* is partitioned exploits the hierarchical pyramidal structure of the wavelet transform in natural images, where energy is usually concentrated at the subbands of higher level (i.e., those of lower frequency). Thereby, when the set *I* is significant, it is likely to have many or all its significant coefficients in the wavelet subbands of higher level, and

that is why they are the ones being decomposed in first place as set *S*. Note that this is roughly the same assumption used by EZW and SPIHT. In addition, with the way the sets of type *S* are partitioned, we try to exploit the clustering of energy of the transformed images in a better way.

Although SPECK can be considered a block-based encoder, due to its partitioning algorithm in blocks instead of trees, each block is not independently encoded as it happens with EBCOT. Later, a variant of SPECK that really processes blocks of coefficients in a separate way will be introduced. This method is called SBHP [CHY00] and was a serious alternative candidate to EBCOT for the JPEG 2000 standard.

Some details about the SPECK algorithm can be found in [PEA04]. The implementation proposed in this paper uses one list for significant coefficients (for further refinement passes), and another list for insignificant sets *S*, named as in SPIHT, list of significant pixels (LSP) and list of insignificant sets (LIS) respectively. For the insignificant sets, the size of each set *S* should be included in the list. A minor change with respect to SPIHT is that the elements of the LIP list (list of insignificant pixels) of SPIHT are now included in the LIS list, considering each single coefficient as a set with only one element. In addition, the sets in LIS are tested and partitioned in increasing order of their size (i.e., smaller sets first), since smaller sets are more likely to possess significant coefficients, and hence they are more valuable to reduce distortion. Finally, as in SPIHT, the result of the significance tests can be arithmetically encoded instead of binary encoded to improve efficiency.

The basic SPECK algorithm is only SNR scalable. However, an extension of SPECK, called S-SPECK, was introduced in [XIE05] as a version of SPECK with additional features, such as resolution scalability and region of interest (ROI), but with similar complexity and compression efficiency as the basic SPECK algorithm.

If we analyze the coding performance of SPECK, in general, its PSNR results are slightly worse than SPIHT, in a range of 0.1-0.4 dB at 1 bpp. Only in highly detailed images SPECK performs better than SPIHT, due to the reasons given in the performance analysis section of EBCOT. A higher complexity encoder, called embedded zero block coding (EZBC) [HSI00], was proposed as a combination of SPECK and EBCOT. It uses the SPECK algorithm to locate the significant coefficients, and then it employs the context-based adaptive arithmetic coding of EBCOT to encode the information, achieving in this way almost 1 dB more than SPECK at 1 bpp, at the expense of higher complexity.

In terms of complexity, the authors of SPECK reported in [PEA98] that SPECK was several times faster than an early verification model of JPEG 2000 based on EBCOT (particularly, VM3.2A). However, bit-plane coding is still present in SPECK. Furthermore, although a significance test can be performed very fast on a coefficient set by maintaining information about the highest coefficient in the set, once a set has to be partitioned, a more complex search for the highest coefficient in each new set is needed to evaluate the significance of the new sets. Observe that this search can be implemented faster if an analysis of partial results is performed before starting coding, at the expense of higher memory requirements.

4.3.2.1 Subband-Block Hierarchical Partitioning (SBHP)

SBHP [CHY00] is a low-complexity implementation of SPECK that was introduced in the JPEG 2000 framework as an alternative to tier 1 coding to reduce its complexity. Contrary to SPECK, this algorithm is applied directly and independently to square blocks as it happens in EBCOT, with code-blocks no larger than typically 64×64 or 128×128.

The partitioning method of each code-block is pretty similar to that used by SPECK to encode an entire transformed image. A code-block is initially partitioned into two coefficient sets, a set *S*, formed in this case by 2×2 coefficients in the upper left corner, and a set *I*, formed by the rest of coefficients. Then, when a set is significant, it is further partitioned with the rules of SPECK (see Figure 4.6). In order to reduce complexity, instead of arithmetic coding, a Huffman code with 15 symbols is chosen to encode the result of the significance tests. Once the code-blocks are independently generated, the final bitstream is reorganized with EBCOT's tier 2 coding, achieving rate control and the desired scalability.

The use of Huffman instead of arithmetic coding, and the introduction of block processing, causes a slight decrease in coding performance but entails a reduction of complexity.

4.3.2.2 Non-embedded SBHP/SPECK

As we mentioned in the description of SPIHT, sometimes embedded bit plane coding is not needed. Hence, to speed up coding and decoding, a non-embedded variant of SPECK or SBHP can be introduced if one bypasses the refinement passes and output the binary code of a coefficient immediately upon its being found significance. As in SPIHT, if this approach is taken, there is no need to maintain an LSP list. Experimental results in [CHY00] led to the conclusion that a non-embedded implementation of an SBHP decoder was approx. 35% faster

than the embedded one.

4.4 Other wavelet encoders

4.4.1 Run-length coding

Obviously, not all the wavelet encoders proposed so far have grouped insignificant coefficients in trees (such as EZW and SPIHT) or blocks (as SPECK does). Another group of wavelet encoders is formed by those based on run-length coding.

One of the first image encoders that employed run-length coding was proposed by Tsai, Villasenor and Chen in [TSA96], and it is known as stack-run coding. This algorithm has a similar structure to the baseline JPEG standard. That is, after the transform decomposition, the wavelet coefficients are quantized using a classic scalar quantization scheme. Then, quantized coefficients are encoded using a run-length encoder (RLE) and, finally, an arithmetic encoder is used. The originality of this algorithm resides on the use of a 4-symbol set {0, 1, +, −} that employs the symbols {0, 1} to binary encode the value of each non-zero coefficient, and the {+, −} symbols to separately represent the sign of a coefficient or the binary representation of a run-length count. This type of symbol assignation is made so that once a coefficient is decoded, using a binary representation (0 and 1 symbols) and its sign, the decoder can distinguish according to the type of the following symbol if there is another non-zero coefficient (if the next symbol is 0 or 1) or a run-length count (if it is a + or − symbol).

The coding efficiency of stack-run coding is half-way between EZW and SPIHT. As for complexity, although bit-plane coding is not performed in various steps with several scans through the image, the stack-run encoder is not very fast because each bit is separately stored with arithmetic coding, which is much slower than the raw coding of the significant bits (and sign) employed in other encoders, such as EZW, SPIHT and SPECK.

Note that the stack-run algorithm is not SNR scalable. Another run-length wavelet-based coder, which encodes coefficients bit-plane by bit-plane providing quality scalability, was presented in [BER01]. The authors state that this proposal is faster than the typical tree-based encoders due to the lack of computation needed to define coefficient-trees, but it also leads it to poorer R/D performance. Experimental results show that its coding efficiency is similar to SPIHT only in highly detailed images in which SPIHT performs poorly, otherwise it is

worse. Moreover, bit plane processing is still present, limiting the speed-up of the encoder. In fact, the encoder is only about 30% faster than SPIHT, and the decoder is even slower than SPIHT at 1 bpp.

Note that run-length coding is included even in the clean-up pass of EBCOT's tier-1 coding, but with the objective of reducing complexity rather than improving coding performance.

4.4.2 High-order context modeling

Most wavelet encoders that we have described so far (namely tree, block, or run-length based) are effective to remove data redundancy in the form of a long-term trend (i.e., for not very detailed images), but not for short-term signals (like edges in an image). The reason is that these encoders impose artificial structures, such as trees or blocks, and only context of this shape can be used, whereas wavelet coefficients from natural images can form areas of many arbitrary and complex shapes. Furthermore, most previous encoders largely ignore the sample dependency between neighboring structures (i.e., different trees or blocks).

In addition, a wavelet transform does not achieve total decorrelation between the input pixels because other complex nonlinear correlations exit, maintaining high-order statistical dependencies between the computed wavelet coefficients.

For the aforementioned reasons, best R/D performance in wavelet coding is achieved by high-order context modeling techniques, like those presented in [WU98] [WU01]. In fact, the high number of contexts used in JPEG 2000 is an approach to these techniques, to improve coding efficiency in more detailed or artificial images. In these coders, not only contiguous coefficients are considered for context formation, but also coefficients that are more distant are evaluated (including some in other subbands).

In this type of coder, it could seem that the higher the order of the model, the higher compression. However, this is not true in general. The problem is that natural images are non-stationary and we do not have prior knowledge of the source. Hence, statistics for a model are generally learned in a dynamic way using adaptive coding. In order to fit a statistical model to the source, a large number of samples are needed. Therefore, if we use many contexts, an image might not provide sufficient samples to reach a good model, or even worse, correlative coefficients in the same context could be so distant in the image that the statistics could have changed. This problem is known as context dilution [RIS84].

In order to reduce the number of contexts, the encoder presented in [WU01] takes advantage of the fact that HL subbands exhibit predominantly vertical samples structures, while LH subbands exhibit more horizontal structures. Hence, for the former subbands, a vertically prolonged modeling context is employed, while for the latter subbands, the neighbor samples for context formation are consider in a horizontal structure. This way, the number of contexts is reduced and hence the context dilution problem is mitigated. Note that inter-subband dependency is also exploited because parent coefficients (and even their neighbors) are also considered for context modeling. The exact context formation and more details on this encoder can be found in [WU01].

Despite its excellent compression performance (in PSNR, 0.5 dB above SPIHT for natural images, and more than 1 dB above it with highly detailed images), a main drawback arises from this type of coder. High-order context computation in which these coders lie is time intensive, especially the texture pattern extraction from neighboring samples and the linear combination of them, which results in a great increase of the encoder complexity.

4.5 Tuning and optimizing the performance of the EZW algorithm

In general, when a new wavelet coder is proposed, the corresponding algorithm needs a set of optimizations in order to be competitive in performance. These optimizations are mainly related to the final algorithm implementation. Therefore, it is very important to evaluate the different implementation choices when developing the final version of the wavelet coder, in order to improve performance. In the following chapters, we will propose several wavelet-based image encoders, and then we will study some implementation options to improve their performance. In this section, we will perform a similar analysis with one of the classic wavelet image encoders that we have presented in this chapter, namely the EZW algorithm. A partial analysis of this encoder was performed in [ALG95], but a deeper study and more parameters can be examined.

In order to perform this study, we have implemented a version of the EZW wavelet still image coder based on Shapiro's descriptions [SHA93]. Then, we will test different implementation choices in order to show their impact on the overall coder performance. Also, we will compare the results obtained with our implementation with those published by the author of the EZW in order to check its correctness.

Shapiro's EZW is a relatively complex algorithm, with several stages and parameters that

can be optimized. Our first task was to implement the EZW algorithm to be able to identify the main parameters of this algorithm and in this way get the best performance by tuning them. Hence, in this section, we present different implementations alternatives that we can establish in the algorithm, some of them mentioned by Shapiro and others not, and we evaluate their contribution to the performance of EZW. We have grouped all these options in four categories: (a) wavelet family, (b) coefficient pre-processing, (c) improvements on the EZW, and (d) improvements related to the adaptive arithmetic encoder.

Note that, when results are presented (in tables or curves), all configuration options are assumed to be set to its default value (which will be given in each subsection) with the exception of those explicitly mentioned, and that the default image to perform this study will be the standard Lena (unless something different is mentioned).

4.5.1 Choosing the best filters

In Chapter 2, we saw that choosing a good filter set is crucial to achieve good compactness of the image in the low frequency subband, this way we reduce the amount of non-zero coefficients and its magnitude, and therefore the image entropy. Shapiro uses an Adelson 9-tap QMF filter-bank. With this filter and the standard image Lena, he obtains the results shown in Table 4.1 (Orig column). Our implementation, with the same image and filter (Adel column), shows similar results. We think that these results validate our implementation. However, other bi-orthogonal filters, in particular B9/7 and Villasenor 10/18 (Vil), make a better energy compactness and consequently provide better results. Simpler filter-banks, like Daubechies 4-tap filter (D4), get poorer compactness and hence lower PSNR values. Similar results are obtained with the standard image Baboon. However, with this image, Villasenor 10/18, with a higher filter size (and thereby complexity), achieves remarkably better performance, showing a great capability to efficiently decompose highly detailed images. All these results are also shown in Figure 4.7.

	PSNR Lena Image					PSNR Baboon image		
rate	Orig	Adel	Vil	B9/7	D4	Adel	Vil	B9/7
2	N/A	44.03	44.05	44.18	43.90	31.86	32.46	32.02
1	39.55	39.53	39.64	39.63	39.17	27.46	27.83	27.39
0.5	36.28	36.28	36.59	36.49	35.54	23.84	24.50	23.88
0.25	33.17	33.18	33.50	33.43	32.23	22.37	22.54	22.70

Table 4.1: Filter-bank comparison with Lena and Baboon source images and EZW coding.

Another important aspect in wavelet processing is the number of decomposition levels. It mainly depends on the image size and the number of filter taps. As we can see in Figure 4.8, for a 512×512 image and a 9-tap QMF filter, it is highly interesting decomposing the successive LL subbands up to four times. However, less improvement is attained with more than four decomposition levels. By default, we, as Shapiro does, will perform a six-level dyadic decomposition with Adelson 9-tap QMF filter on the 512×512 standard image Lena.

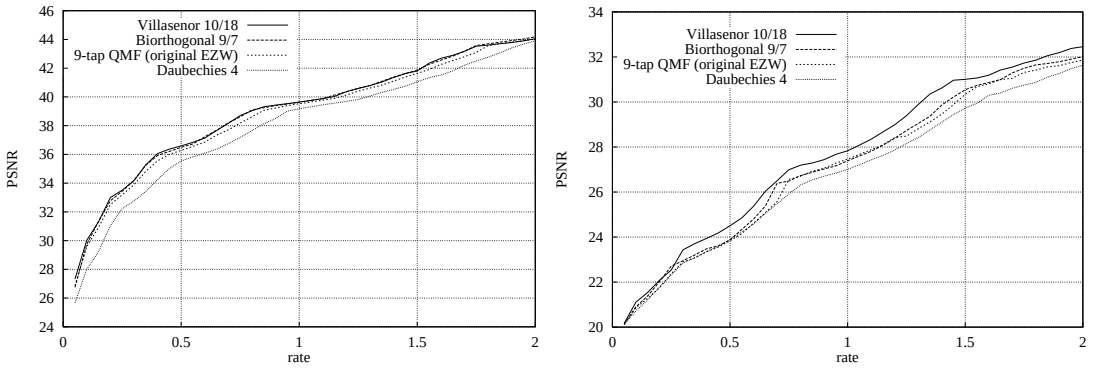


Fig. 4.7: Evaluating various filter-banks for (a) Lena and (b) Baboon using EZW coding.

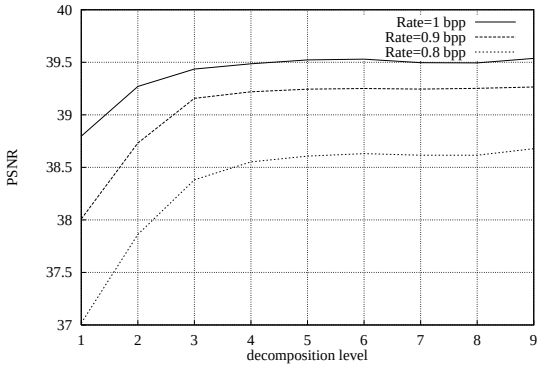


Fig. 4.8: Impact of the number of decomposition levels on performance.

4.5.2 Coefficient pre-processing

Shapiro proposes that the image mean can be removed before the EZW algorithm is applied, and transmitted to the decoder separately. Figure 4.9 shows the effect of this idea. It was performed in two different manners: (a) removing simultaneously the mean of all the bands, and (b) removing it only from the LL band (similar to remove the original image mean). As wavelet subbands are expected to be zero-mean, trying to remove the mean of these bands

does not seem to be a good idea, as Figure 4.9 shows. On the other hand, the effect of removing the mean of the LL band is nearly negligible (thus, the default value will be no mean removed).

An important effect that appears in all the Rate/Distortion curves based on the EZW algorithm is its scalloped aspect; it can be easily noticed the peaks in the performance, which correspond with the end of a full EZW iteration. This is due to its embedded nature: the EZW presents the best performance when the algorithm finishes its bit budget just at the end of a subordinate pass. A uniform pre-quantization of the wavelet coefficients before applying the EZW could move these peaks along different rates. This effect is shown in Figure 4.10, where at established bit rates, different quantization factors (q values) have been used. In this case, with 1 bpp, best performance is obtained at $q=0.2\times2^k$, being k integer. These peaks are shifted to the right as the bit rate decreases, until a 0.5 bpp value is reached. Then, a full pass is completed and peaks repeat again at $q=0.2\times2^k$.

Therefore, with the suitable value of q, results from Table 4.1 can be significantly improved (almost 1 dB with respect to the original EZW results when combined with Villasenor 10/18). These results are shown in Table 4.2 (from now on, we will consider that no uniform pre-quantization is used as default value).

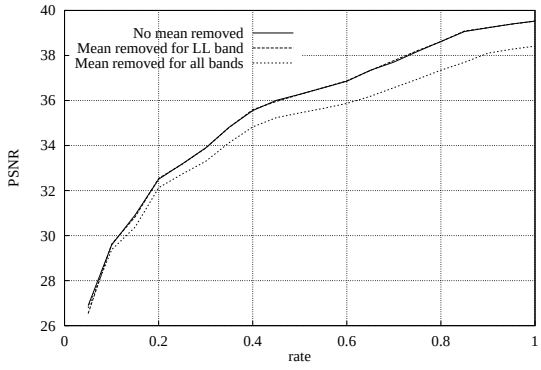


Fig. 4.9: Mean value removing option in EZW.

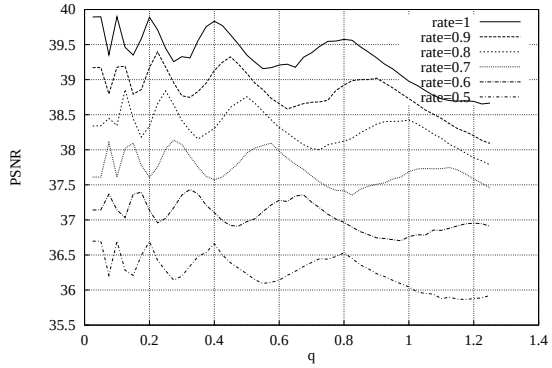


Fig. 4.10: Uniform pre-quantization option introduced in EZW to shift performance peaks. This graph shows the PSNR for various quantization factors at constant bit rates.

	PSNR 9-tap QMF filter			PSNR Villasenor 10/18 filter	
rate	Orig	No quant.	q = 0.8	No quant.	q = 0.4
2	n/a	44.03	44.49	44.05	44.79
1	39.55	39.53	39.83	39.64	40.20
0.5	36.28	36.28	36.87	36.59	37.04
0.25	33.17	33.18	33.52	33.50	33.97

Table 4.2: Optimized PSNR results after introducing a pre-quantization process to shift performance peaks in EZW.

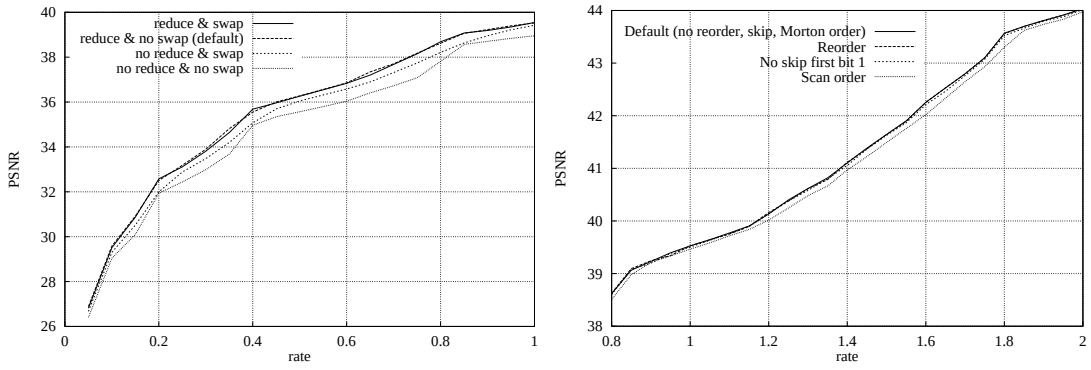
4.5.3 Improvements on the main EZW algorithm

Some options can be established in the main EZW algorithm. The curve "no reduce & no swap", in Figure 4.11(a), shows the different gradient existing between a dominant and a subordinate passes. This means that bits from subordinate passes are more valuable than those from dominant passes. Hence, performing a swap between the order of these stages could be a good idea. The curve "no reduce & swap" shows the results of performing the subordinate pass first, and then the dominant pass for every EZW iteration. In this way, when we run out of bits, no bit from the dominant pass is processed prior than one from the subordinate pass (with the same threshold). In fact, we have seen in this chapter that this type of improvement was also carried out in other encoders like JPEG 2000, encoding the most valuable pass first, and SPECK, testing the subbands in increasing order of their size.

Another improvement on EZW consists in reducing the uncertainty interval at the decoder. The decoder must predict the bits that the encoder could not send because it finished its bit budget. It can assume that the rest of bits are 0, or maybe that all are 1. But the best

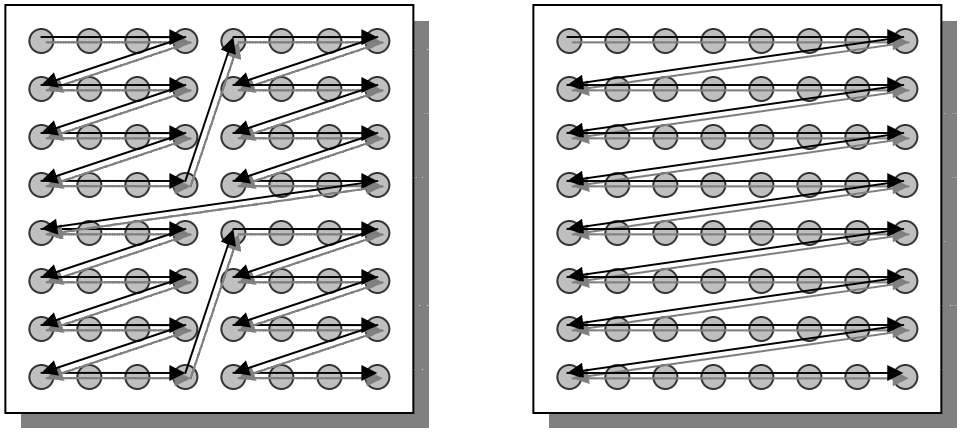
option seems to suppose that, for every coefficient, the more significant predicted bit is 1 and the rest 0, so we would have a lower uncertainty interval and, as consequence, a lower mean error. The curves "reduce" from Figure 4.11(a) shows the improvement of this action, and how performing a swap is not actually significant when the uncertainty interval is already reduced.

Other options on the EZW coder are shown in Figure 4.11(b). One of them is the scanning order of the coefficients in the dominant pass. We can see that a Morton order (see example in Figure 4.12), which performs the scan in small groups, improves the performance of the algorithm if compared with a typical raster scan order (line-by-line in each subband), due to the best adaptivity to local features achieved in the arithmetic encoder. Another improvement is not to encode the first bit of a coefficient, because the decoder can deduce it from the significant symbols in the dominant pass. The last option is to sort the coefficients in the subordinate pass according to their magnitude, so that bigger coefficients are coded before smaller ones. This option was originally proposed by Shapiro in [SHA93], but it increases the coder complexity with almost no improvement in efficiency. In fact, Figure 4.11(b) shows that, after evaluating these options, only the scan order seems to be really important, being the Morton order better than regular raster order.



(a) (b)

Fig. 4.11: Improvements on the main EZW algorithm: (a) Swapping dominant and subordinate passes. (b) Coefficient scanning order.



(a) (b)

Fig. 4.12: (a) Morton scan order, in which coefficients are scanned in small blocks, and (b) Raster scan order, with the subbands scanned line-by-line.

4.5.4 Improvements on the arithmetic encoder

Several actions can be tackled on the adaptive arithmetic encoder. A regular adaptive encoder uses a dynamic histogram (context) in order to estimate the current probability of a symbol. To improve this estimation we can use a different histogram depending on the significance of the previously coded coefficient and its parent coefficient in the current pass. For example, if we code a coefficient with a significant parent, and the previous coefficient is insignificant, we would use a different histogram from the one used if the parent is insignificant or the previous coefficient is significant. Therefore four histograms can be used, depending on the significance of the parent and the previous coefficients.

In addition, all the histograms can be restarted at the end of a full pass, to improve their adaptivity. Finally, since the last subband levels do not have offspring, we do not need to use a four-symbol alphabet for these bands, and another arithmetic encoder (without the isolated-zero symbol) can be used. By default, all these improvements are used. Figure 4.13 shows the contribution of each option to the performance of the algorithm by removing them one by one. Only the use of contexts seems to be of really interest (notice the smaller scale of this graph).

It is also important to consider the maximum frequency count in the adaptive arithmetic encoder (see more details in Section 1.2.5 and [WIT87]) and how much the histogram increases with each symbol (the default values that we have used are 512 and 6 respectively).

Note that although most results have been presented for the Adelson 9-tap QMF filter

bank, because it was the one used by Shapiro, the rest of filters from subsection 4.5.1 behave similarly for most options.

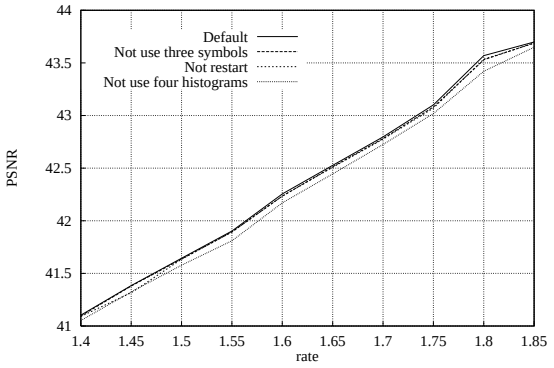


Fig. 4.13: Arithmetic encoder evaluation.

4.6 Summary

In this chapter, we have described the main wavelet-based image encoders, some of them tree-based (such as EZW, SPIHT, SFQ and PROGRES), others block-based (e.g., EBCOT, SPECK and SBHP), and also encoders using run-length coding or high-order context modeling. In the coding efficiency assessment, we can conclude that rate-distortion optimization algorithms, and the use of more complex contexts provide the best results, but they imply time-consuming processes. In addition, many wavelet encoders achieve SNR scalability through bit plane coding, which slows the encoder (most of all in cache-based architectures). Some alternatives have been proposed to avoid bit plane coding in parts of the encoders (like for example in the refinement passes of SPIHT and SPECK), speeding up the execution, but bit plane coding still remains in other passes of these coders (namely, in the sorting passes).

In addition, in the last section we have performed a deep study of an implementation of EZW, evaluating many alternatives (up to 15) in the different stages of the encoder. Also, we have shown that it is possible to get better results (about half dB) than those published by the author under the same conditions, and if more efficient filter banks are used, the EZW performance significantly increases (up to 0.8 dB), which stresses the importance of a proper parameter tuning when a new encoder is proposed.

Chapter 5

Fast run-length coding of coefficients

In the previous chapter, we presented some of the most important wavelet-based image encoders. In general, these algorithms are moderate to high complex, because they achieve good coding performance with quite complex techniques such as bit-plane coding, rate/distortion optimization algorithms and high-order context modeling. Apart from compression efficiency, other features, like SNR/resolution scalability and error resilience, are also usually considered. In this chapter, we introduce a new wavelet image encoder that is extremely simple and faster than most of the previous proposals, most of all with a run-length mode. Despite its simplicity, real implementations have shown that its coding performance is within the state-of-the-art. Thus, we will show that the proposed encoder has the same rate/distortion performance as SPIHT and Jasper (an official implementation of JPEG 2000) while they are several times slower. Moreover, our proposal is resolution scalable and is more robust than SPIHT (and other tree-based encoders), due to its lack of inter-subband dependency. In addition, it performs in-place processing and therefore, it does not require additional lists or complex data structures, so there is no memory overhead.

5.1 Introduction

In Chapter 2, we described several strategies that have been proposed in order to speed-up the computation time of the discrete wavelet transform (e.g., the lifting scheme and line-based approaches, which are faster in cache-based systems). Furthermore, in this thesis we have proposed a complete line-based algorithm to recursively compute the bi-dimensional DWT

using the lifting scheme.

However, the coding stage is not usually improved in terms of computational cost and memory requirements. When designing and developing a new wavelet image encoder, the most important factor to be optimized is usually the rate/distortion (R/D) performance, while other features like embedded bit-stream, SNR scalability, spatial scalability and error resilience may be also considered. In fact, many times, wavelet-based image encoders have additional features that are not really needed for many applications, but which make them both CPU and memory intensive. Often fast processing is preferable. An example of an application with real-time restrictions is intra-frame video capturing, where compression must be performed in a limited time slot. Low complexity may also be desirable for image compression in high-resolution digital camera shooting, where high delays could result annoying and even unacceptable for the final user (even more for modern digital cameras, which tend to increase their resolution).

In this chapter, we propose an algorithm aiming to achieve similar rate/distortion performance to current state-of-the-art image coders, and considerably reduce the required execution time. Moreover, due to the in-place processing of the wavelet coefficients performed in the proposed encoder, there is no memory overhead. It only needs memory to store the source image (or even a part of it) to be encoded. In addition, our algorithm is naturally spatial scalable, and it is possible to achieve SNR scalability, and certain degree of error resilience, due to its lack of inter-subband dependency.

In order to speed up the execution time of the coding and decoding processes, a run-length mode will be used. Furthermore, for a really low-complexity implementation, bit-plane coding existing in many wavelet coders [SHA93] [PEA96] [TAU00] [CHY00] [BER01] [WU01] [PEA04] must be avoided so that multiple scan of the transform coefficients, which leads to high cache miss rate in cache-based systems, is not performed. In addition, rate/distortion optimization algorithms (like those employed in [XIO97] [TAU00]) are really time-consuming processes and must be avoided as well. Finally, the use of high-order contexts (like in [TAU00] [WU01]) also should be avoided to reduce complexity.

In this chapter, we first propose a simple coding algorithm that will be used as a starting point for the proposed run-length encoder, which is explained in detail later. In addition, we tune the algorithm later, similarly to that proposed for the EZW algorithm in the previous chapter. Finally, we assess the performance of the proposed algorithm, focusing on its

complexity, which is compared using real implementations.

5.2 A simple multiresolution image encoder

As we have seen in Chapter 4, one of the main drawbacks in many wavelet image encoders is their high complexity. Many times, that is mainly due to the bit plane coding, which is performed along different iterations, using a threshold that focuses on a different bit plane in each iteration. This way, it is easy to achieve an embedded bit-stream with progressive coding, since more bit planes add more SNR resolution to the recovered image.

Although embedded bit-stream is a nice feature in an image coder, it is not always needed and other alternatives, like spatial scalability, can be more valuable depending on the final purpose. In this section, we propose a very simple algorithm that is able to encode the wavelet coefficients without performing one loop scan per bit plane. Instead of it, only one scan of the transform coefficients is needed.

5.2.1 Quantization method

In this algorithm, the quantization process is performed by two strategies: one coarser and another finer. The finer one consists in applying a scalar uniform quantization to the coefficients, and it is performed just after the DWT is applied. If the lifting scheme is used to compute the wavelet transform, this fine quantization can be applied with the filter normalization (see Chapter 2), simply by combining both scaling (K_1) and quantization ($1/2Q$) parameters in only one scaling factor ($K_1/2Q$). In this way, no additional multiplications are needed for the finer quantization.

On the other hand, the coarser quantization is based on removing bit planes from the least significant part of the coefficients, and it is performed while our algorithm is applied. Related to this bit plane quantization, we define *rplanes* as the number of less significant bits to be removed. This parameter is useful to implement a fast quantization method, or for quantization in hardware architectures that only support integer arithmetic.

Note that we will use this combined finer and coarser quantization in all the coding algorithms proposed in this thesis. For this reason, more explanation about this joint scalar/bit plane quantization process is given separately in Appendix A.

5.2.2 Coding algorithm

Let us explain now the proposed encoder. In the initialization of the encoder, the maximum number of bits needed to represent the highest coefficient is calculated. We call this parameter *maxplane*. As an alternative, this value can be obtained while computing the DWT, in order to avoid a second scan of the wavelet coefficients. Note that this parameter can be computed very efficiently with a simple bitwise OR operation. To this end, all the coefficients should be successively combined using the OR operator, and the *maxplane* parameter is given by the number of bits required to represent the result of the successive OR operations.

Afterwards, the *maxplane* and *rplanes* parameters are output to the decoder. Then, an adaptive arithmetic encoder that is used to encode the exact number of bits required by each coefficient is initialized. In our proposal, those coefficients ($c_{i,j} \in C$) that require more than *rplanes* bits to be coded (in other words, those $|c_{i,j}| \geq 2^{rplanes}$) are output using this type of symbol. Thus, only $2^{maxplane} - 2^{rplanes}$ symbols are needed to represent this information. However, an additional symbol is required to indicate those coefficients that are lower than the established threshold ($2^{rplanes}$). This symbol is called *LOWER*.

Note that the *maxplane* parameter is required to compute the number of symbols employed by the arithmetic encoder. Nevertheless, this parameter and its search could be omitted if the arithmetic encoder is initialized with enough symbols to encode the highest coefficient that is estimated that can appear in the transformed image, which basically depends on the image size, the number of decomposition levels, the wavelet normalization, and the frequency behavior of the image. In this case, the coding algorithm is slightly faster (since the bitwise OR operations are not needed), but the coding performance is lower, and there is a risk of overflow if the estimated number of symbols is not large enough.

Here, we have introduced another feature of our wavelet encoder: it represents the significance map of the wavelet coefficients with a single symbol per coefficient, encoded with arithmetic coding. Recall that the significance map is the way an encoder locates the significant coefficients (i.e., those coefficients different to zero) and their magnitude. A critical aspect of a wavelet encoder is how to represent this map. We saw in Chapter 4 that in the EZW algorithm, the significance map is encoded with successive dominant passes. Similarly, both SPIHT and SPECK represents this information in the sorting passes. Finally, in EBCOT, the significance map is represented in a more efficient way, with two different

passes per bit-plane: the signification propagation pass and the clean up pass.

In the coding algorithms presented in this thesis, we say that $c_{i,j}$ is a significant coefficient when it is different to zero after discarding the least significant *rplanes* bits, in other words, if $|c_{i,j}| \geq 2^{rplanes}$. Otherwise, it is insignificant.

After the initialization stage, the wavelet coefficients are encoded as follows. For each subband, from the N level to the first one, all the coefficients are scanned in Morton order (i.e., in medium-sized blocks, such as Figure 4.12(a) shows). For each coefficient in that subband, if it is significant, a symbol indicating the number of bits required to represent that coefficient is arithmetically encoded. Since coefficients in the same subband have similar magnitude, and due to the order we have established to scan the coefficients (in small blocks to cluster local features), an adaptive arithmetic encoder is able to represent very efficiently this information. However, we do not have enough information to correctly reconstruct the coefficient; we still need to encode its significant bits and sign.

On the other hand, if a coefficient is not significant, we transmit a *LOWER* symbol so that the decoder can determine that it has been absolutely quantized. Thereby, it does not have associated information, neither coefficient bits nor sign.

Note that when encoding the bits of a significant coefficient, the first *rplanes* bits and the most significant non-zero bit are not coded. The decoder can deduce the most significant non-zero bit through the arithmetic symbol that indicates the number of bits required to encode this coefficient. Moreover, in order to speed up the execution time of the algorithm, the bits and sign of significant coefficients are “raw encoded” (binary encoded), which results in very small lost in R/D performance.

The proposed encoding algorithm is described in Algorithm 5.1.

function SimpleWaveletCoding( )
1) Initialization:
output *rplanes*
output $maxplane = \max_{\forall c_{i,j} \in C} \left\{ \left\lceil \log_2 \left(\left| c_{i,j} \right| \right) \right\rceil \right\}$
2) Output the coefficients:
Scan the subbands in the established order.
For each $c_{i,j}$ in a subband
$nbits_{i,j} = \left\lceil \log_2 \left(\left| c_{i,j} \right| \right) \right\rceil$
if $nbits_{i,j} > rplanes$
arithmetic_output $nbits_{i,j}$
binary_output $\text{bit}_{nbits_{(i,j)}-1} \left(\left| c_{i,j} \right| \right) \ldots \text{bit}_{rplane+1} \left(\left| c_{i,j} \right| \right)$
binary_output $\text{sign}(c_{i,j})$
else
arithmetic_output *LOWER*
end of fuction

Note: $\text{bit}_n(c)$ is a function that returns the n^{th} bit of c.

Algorithm 5.1: Simple wavelet coding.

5.2.3 A simple example

Figure 5.1 shows how this algorithm can be applied to a small 4×3 example subband. The coefficients of this figure have already been scalar quantized, and their unsigned binary representation is given in a heap, representing their sign along with their most significant non-zero bit. The selected *rplanes* parameter for the coarser quantization is two bit-planes, so that all coefficients under this threshold are discarded. The *maxplane* parameter is directly computed from the highest coefficient, and it is the number of bits needed for its unsigned binary representation. In our example, it is equal to six. On the left part in this figure, we can see boxes with the arithmetic symbols that result from encoding each line. In these boxes, significant coefficients are represented by their length (in number of bits), while the symbol *L* (*LOWER*) is used for those insignificant values.

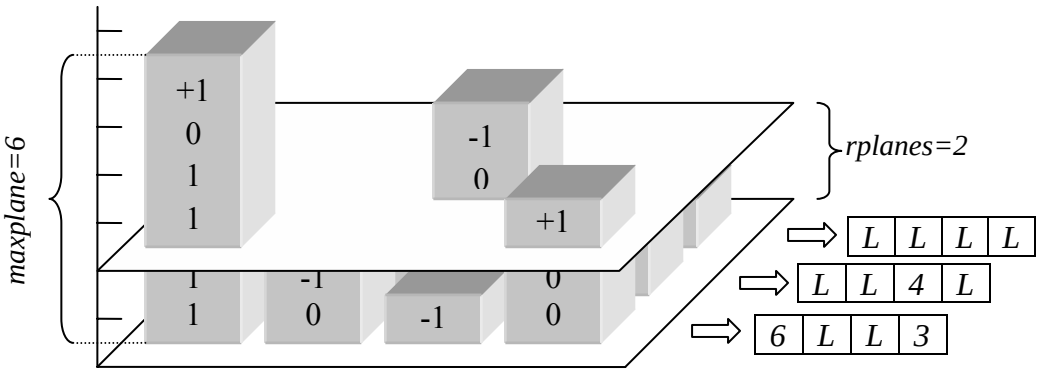


Fig. 5.1: 3D view of a 4×3 wavelet subband and the resulting arithmetic symbols for Algorithm 5.1.

6, +, $\{011_B\}$	*L*	*L*	3,+, {}	*L*	*L*	4, -, $\{0_B\}$	*L*	*L*	*L*	*L*	*L*

Fig. 5.2: Appearance of the 3×4 subband of Figure 5.1 encoded using Algorithm 5.1.

The bitstream obtained after encoding this subband is illustrated in Figure 5.2. In this example, coefficients are scanned row-by-row, from the lowest left corner. As we have seen, for the significant coefficients, not only the arithmetic symbol is necessary, but also their sign and significant bits are "raw encoded" (except the most significant bit, which is always 1).

5.2.4 Features of the algorithm

The proposed Algorithm 5.1 is resolution scalable due to the selected scanning order (in decreasing order of subband level) and the nature of the wavelet transform. This way, the first subband that the decoder attains is the LL_N, which is a low-resolution scaled version of the original image. Then, the decoder progressively receives the remaining subbands, from lower frequency subbands to higher ones, which are used as a complement to the low-resolution image to recursively double its size, which is know as Mallat decomposition (see Chapter 2).

Note that spatial and SNR scalability are closely related features. Spatial resolution allows us to have different resolution images from the same image. Through interpolation techniques, all these images could be resized to the original size, so that the larger an image is, the closer to the original it gets. Therefore, to a certain extent, this algorithm could also be used for SNR scalability purposes.

The robustness of this algorithm lies in the low dependency among the encoded

information, since there is no inter-subband dependency. Dependency among the encoded information is only present in consecutive symbols, and thus the use of synchronism marks and independent probability models between marks would increase the error resilience, at the cost of slightly decreasing the R/D performance. However, the lack of dependency that yields higher robustness also causes lower compression performance in the encoding process. The correlation among coefficients and subbands can be exploited in a more efficient way, if inter-subband redundancy is removed, as we will see in the next chapter, which takes this algorithm as a starting point to define a more efficient tree-based encoder, with similar characteristics to the one presented in this section (except robustness).

A disadvantage of this simple algorithm is its higher complexity when working at low bit rates. Observe that one symbol is always encoded with arithmetic coding for every coefficient. The cost of this operation is not negligible, most of all if it is repeated many times. At low bit rates, most of the encoded coefficients are represented by the *LOWER* symbol. An arithmetic encoder is able to take advantage of this redundancy and achieves good coding performance. However, a simple grouping algorithm is needed in order to decrease the total number of encoded symbols, and thus the computational cost of Algorithm 5.1. We will tackle this problem later in this chapter.

Another drawback of this algorithm, and the rest of encoders that we propose in this thesis, is that it is not embedded (in SNR), and no iterative optimization algorithm is applied, and hence precise rate control cannot be achieved. Actually, most fast encoders have this problem, like for example, the original baseline JPEG, PROGRES and the non-embedded SPITH and SPECK algorithms. In these coders, a quality parameter is employed to achieve greater or smaller compressed files, varying the image quality. However, when compressing an image with a constant quality factor, the final image size is unknown a priori because it depends on the image features. In our algorithms, although there are two quantization parameters available to indicate the desired image quality, we can achieve rate control to a certain degree by analyzing the characteristics of the input image (mainly the first-order entropy of the wavelet coefficients). This analysis is general for all the image encoders proposed in this chapter and in the next one, and it is performed in Appendix B.

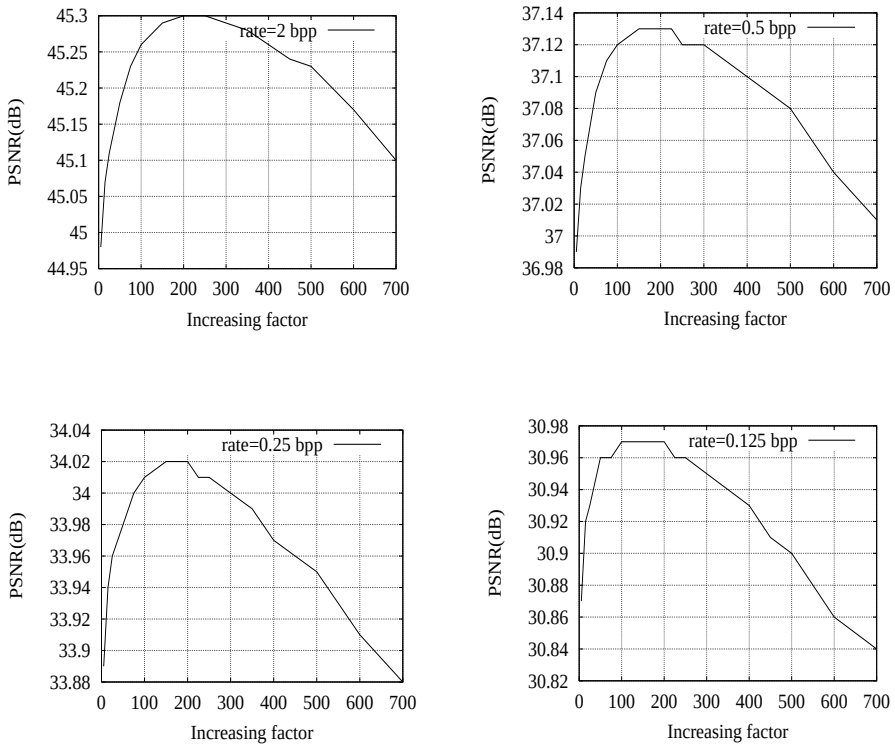


Fig. 5.3: PSNR performance depending on the increasing factor in the adaptive model, with high, medium and low bit rates.

5.2.5 Tuning the proposed algorithm

Despite the simplicity of the proposed algorithm, its rate/distortion performance is competitive with the state-of-the-art image coders, provided the suitable tuning of the encoder parameters is applied. In this section, we present some details on the algorithm, which make it more efficient. In fact, in the previous chapter, we saw the importance of parameter tuning with the EZW encoder.

In order to perform the desired tests, we have selected the standard Lena image as a basis pattern.

5.2.5.1 Tuning the adaptive arithmetic encoder

In Algorithm 5.1, an adaptive arithmetic encoder is used to efficiently encode the symbols that are output in the coding process. As we saw in Chapter 1, a regular adaptive arithmetic encoder uses a dynamic histogram to estimate the current probability of a symbol. In order to update this probability, a frequency count associated to a symbol is increased each time that it

is encoded. Thus, we can consider a new parameter regarding how much the histogram is increased with every symbol; we call this parameter *increasing factor*.

In the original adaptive arithmetic encoder [WIT87], a value of one was used for this the *increasing factor* parameter. We have observed that if this value is greater than one, the adaptive arithmetic encoder may converge faster to local image features, leading to higher compression ratios. Recall that, in general, images are non-stationary, and therefore they change their statistics. As a consequence, a higher increase may cause a faster convergence from a transient state to a steady state when an image changes its statistics. However, increasing it too much may turn the model (pdf estimation) inappropriate, leading to poorer performance.

In addition, this parameter must be evaluated along with a *maximum frequency count* (sometimes called *Max_frequency*). When this value is exceeded by the sum of all the counts in the arithmetic encoder histogram (this count is called *cumulative frequency count* in Section 1.2.5), all these counts are halved and thus overflow is prevented (again, see more details in [WIT87] and Chapter 1). The original proposal for this parameter is 16384 (when using 16 bits), but experimental tests have led us to use a slightly lower value, 12500, so the model is normalized more often.

In Figure 5.3, we have evaluated the PSNR performance for several increasing factors, using low, medium and high compression rates (2bpp, 0.5bpp, 0.25 and 0.125bpp). This figure shows that, for the proposed maximum frequency count (12500), an optimal increasing factor is located around 200 for all the bit rates, achieving an improvement of about 0.1-0.3 dB when it is compared to the original proposal (i.e., increasing only one).

5.2.5.2 Context modeling

As we mentioned previously, coefficients in the same subband have similar magnitude. In order to take better advantage of this fact, different histograms may be handled according to the magnitude of previously coded coefficients, i.e., according to the context of the coefficients. In particular, we propose the use of two different contexts according to the significance of the left and upper coefficients (which is already encoded if a Morton scan order is performed with each block scanned in raster order). In this way, if both coefficients are insignificant, the coefficient being encoded is likely to be also insignificant, and thus a specific probability model is used.

coder / rate (bpp)	Proposed algorithm	Proposed with context	Jasper/ JPEG 2000	SPIHT
2	45.30	45.30	44.62	45.07
1	40.24	40.32	40.31	40.41
0.5	36.95	37.13	37.22	37.21
0.25	33.79	34.02	34.04	34.11
0.125	30.79	30.97	30.84	31.10

Table 5.1: PSNR (dB) with different bit rates and coders using Lena.

The benefit of using contexts is shown in Table 5.1, where the R/D performance is presented for both cases (first and second columns), attaining a profit of up to 0.2 dB (mainly at low bit rates). On the other hand, in this table, we see that our coder is within the state-of-the-art in terms of rate/distortion performance, displaying similar PSNR results to SPIHT [SAI96] and Jasper [ADA02], an official implementation of JPEG 2000 included in the ISO/IEC 15444-5 standard.

5.2.6 Discussion

Although our algorithm executes faster than SPIHT and Jasper (it will be shown in Section 5.4), it presents a major drawback for low bit rate images. If we analyze the description of Algorithm 5.1, we can observe that all the symbols are explicitly encoded, i.e., *width*×*height* symbols are arithmetically encoded. We know that the adaptive arithmetic encoder is one of the slower parts of an image coding system (most of all in simple encoders like this). In our algorithm, experimental results have shown that, for low bit rates, more than 3/4 part of time is spent in the arithmetic encoder. In addition, most symbols being encoded have been absolutely quantized, and are always represented by the same symbol: *LOWER*.

In order to overcome this problem and to reduce the complexity in these cases, a way of grouping large streams of *LOWER* symbols seems necessary. It will be introduced in the next section.

6, +, $\{011_B\}$	*L*	*L*	3,+, {}	*L*	*L*	4, -, $\{0_B\}$	*R*, 3, $\{01_B\}$

Fig. 5.4: Appearance of the 3×4 subband of Figure 5.1 encoded using Algorithm 5.2, with an *enter_run_mode* parameter of 4.

5.3 Fast run-length mode

In this section, a run mode is introduced in the algorithm proposed in the previous section. This run mode serves to reduce complexity when a large number of consecutive *LOWER* symbols appear, which usually occurs in moderate to high compression ratios. Minor improvements in compression performance are expected, due to the fact that we are replacing many likely symbols by a symbol or symbols that indicate the count of *LOWER*s, which will be less likely. Therefore, although less number of symbols are encoded, the probability dispersion affects adversely the adaptive arithmetic encoder, since it works better with probability concentration.

In this new version, a run length count of *LOWER* symbols is performed. However, this run mode is only applied when the *LOWER* count passes a threshold value (called *enter_run_mode* parameter). Otherwise, the compression performance of the algorithm would decrease due to the large number of less likely run-length symbols introduced, replacing short streams of *LOWER* symbols.

In this new version of the algorithm, when the run count is interrupted by a significant symbol, and the run count value is high enough (that is, it is greater than the *enter_run_mode* parameter), the value of the run length count must be output to the decoder, and the run count is reset.

At this point, a new symbol is introduced: the *RUN* symbol. This symbol is used to indicate that a run value is going to be encoded. After encoding a *RUN* symbol, the run count is stored in a similar way as the significant values. First, the number of bits needed to encode the run value is arithmetically output (using a different context to those used to encode a significant coefficient), afterwards, the bits are raw output.

The new run-length version of the simple encoder presented in the previous section is detailed in Algorithm 5.2.

function RunLengthWaveletCoding( )

1) Initialization:

output *rplanes*

output $maxplane = \max_{\forall c_{i,j} \in LL_N} \left\{ \left\lceil \log_2 \left(\left| c_{i,j} \right| \right) \right\rceil \right\}$

run_length=0

2) Output the coefficients:

Scan the subbands in the established order.
For each $c_{i,j}$ in a subband

$nbits_{i,j} = \left\lceil \log_2 \left(\left| c_{i,j} \right| \right) \right\rceil$

if $nbits_{i,j} \le rplanes$

increase *run_length*

else

if $run\_length \neq 0$

if $run\_length < enter\_run\_mode$

repeat *run_length* **times**

arithmetic_output *LOWER*

else

arithmetic_output *RUN*

$rbits = \left\lceil \log_2 (run\_length) \right\rceil$

arithmetic_output *rbits*

output $\text{bit}_{rbits-1}(run\_legth) \ldots \text{bit}_1(run\_length)$

run_length=0

arithmetic_output $nbits_{i,j}$

output $\text{bit}_{nbits_{(i,j)}-1}\left(\left| c_{i,j} \right| \right) \ldots \text{bit}_{rplane+1}\left(\left| c_{i,j} \right| \right)$

output $\text{sign}(c_{i,j})$

end of fuction

Note: $\text{bit}_n(c)$ is a function that returns the n^{th} bit of c.

Algorithm 5.2: Run-length wavelet coding.

Note that Algorithm 5.2 is also resolution scalable, again due to the selected scanning order of the subbands (from higher levels to lower ones), and the nature of the wavelet transform.

Let us give some details on the algorithm. This algorithm uses the variable *run_length* to count the number of consecutive *LOWER* symbols that appear in the transformed image.

When a significant coefficient appears, and this run count is broken, the *run_length* value must be encoded if it is different to 0, but it is encoded in the form of several consecutive *LOWER* symbols if it does not exceed the threshold value pointed out by the *enter_run_mode* parameter. For this reason, an auxiliary array is needed to temporally retain the context value of each *LOWER* symbol that is not encoded, because in case that the run length count does not reach the threshold value, the context to encode each single *LOWER* symbol must be known.

Furthermore, a detail not described in Algorithm 5.2 (for simplicity) is that, once the algorithm finishes encoding a subband, if the run count is different to zero (that is, if at least the last coefficient is insignificant), this count also has to be encoded, because there is no symbol to indicate the end of a subband.

Another detail to consider is the scanning order of the coefficients in each subband. In Chapter 2 and Section 4.4.2, we saw that the HL subbands exhibit predominantly vertical samples structures, while LH subbands exhibit more horizontal structures. Therefore, it seems clear that in order to get a greater amount of consecutive *LOWER* symbols, coefficients should be scanned horizontally in the LH subbands, and vertically in the HL subbands. A more detailed analysis about the scan path for run length coding of wavelet coefficients can be found in [BER01].

Finally, Figure 5.4 shows an example of applying Algorithm 5.2 to encode the subband of Figure 5.1. For this example, we have used a value of 4 for the *enter_run_mode* parameter. The main difference between this example and the one shown in Figure 5.2 is that the sequence of 5 *LOWER* symbols is encoded with a *RUN* symbol (represented with an *R* in the example) along with the number of bits needed to encode the run count (3 in this case), and the significant bits of its binary representation (in this example, we encode the bits 01b to represent 101b).

5.4 Numerical results

We have implemented these algorithms in order to evaluate their compression efficiency and complexity. The reader can easily perform new tests by using the win32 version of these coders, available at *http://www.disca.upv.es/joliver/thesis*. All the simulation tests have been carried out on a regular Personal Computer (with a 500 MHz Pentium Celeron processor with 256 KB L2 cache), generating image files that contain the compressed images, with the

required file headers.

In order to compare our algorithm with other wavelet encoders, the standard Lena (which is a quite smooth image) and Barbara (which is much more highly detailed) images are used. In the next chapter, we will give more results for this run-length algorithm with other images (namely, Café, Woman and Goldhill).

The coding results for Lena using a run-length mode are practically the same as those shown in Table 5.1, version with contexts. With the correct selection of the *enter_run_mode* parameter (128 in our tests), the compression performance for high bit rates is certainly the same in Algorithm 5.1 and 5.2 (+/– 0.01dB), and at low bit rates, very small improvement is achieved with the run-length mode (+0.03 dB).

Table 5.2 shows similar compression performance comparison using a more detailed image, Barbara. In this case, our algorithm is clearly better than SPIHT, but it is unable to reach the performance of JPEG 2000, due to the high number of contexts and the optimization algorithm (PCRD) employed in this standard.

We have shown that including a run-length mode has not significantly improved compression performance. However, the main goal of this mode was reducing the complexity of the algorithm, most of all at low bit rates. We can see in Tables 5.3 and 5.4 the execution time for coding and decoding the Lena image with various encoders and decoders. These tables show that this objective has been accomplished. In fact, the number of symbols arithmetically encoded at 0.125 bpp has passed from 512×512 (262,144) to only 27,485 and then, the execution time spent in the arithmetic encoder has decreased from 3/4 to less than 1/3 part of the total compression time.

In these tables, we also can compare both proposals with Jasper/JPEG 2000 and the official implementation of SPIHT (implemented by the authors of SPIHT, and which was originally downloaded from ftp://ipl.rpi.edu/pub/EW_Code at the time it was available)[1]. Note that we exclude the DWT computation time from this comparison, since all the compared encoders employ exactly the same bi-orthogonal 9/7 wavelet transform (see

[1] In Chapter 1, we saw that measuring the complexity of algorithms is a hard issue. Execution time of their implementation is largely dependant of the optimization level. This way, there are commercial implementations of JPEG 2000 not included in the ISO/IEC 15444-5 that are faster than Jasper, however they are usually implemented using platform dependant piece of code (in assembly language) and multimedia SIMD instructions. In our tests (in this chapter and in the next chapter), SPIHT, JPEG 2000 and LTW implementations are, as far as possible, written under the same conditions, using plain C/C++ language for all them.

Chapter 2).

Our final run-length proposal (Algorithm 5.2) is up to ten times faster than Jasper/JPEG 2000 when encoding, and up to twice when decoding. In addition, compared with SPIHT, our algorithm is approx. twice faster in the coding process, and about 35 % faster in the decoding process at 1 bpp. In the next chapter, we will give execution time results of this run-length algorithm for larger images (5-Megapixel).

coder/ rate(bpp)	Proposed run-length	Jasper/ JPEG2000	SPIHT
2	42.63	43.13	42.65
1	36.54	37.11	36.41
0.5	31.66	32.14	31.39
0.25	27.95	28.34	27.58
0.125	25.12	25.25	24.86

Table 5.2: PSNR (dB) with various bit rates and coders using Barbara.

codec\ rate	SPIHT	Jasper / JPEG 2000	Proposed (with ctxt.)	Proposed (run-length)
2	210.4	278.5	91.2	95.4
1	119.4	256.1	64.3	61.2
0.5	72.3	238.2	52.7	37.0
0.25	48.7	223.4	47.0	25.9
0.125	36.8	211.3	44.0	20.3

Table 5.3: Execution time for coding Lena (Million of CPU cycles).

codec\ rate	SPIHT	Jasper / JPEG 2000	Proposed (with ctxt.)	Proposed (run-length)
2	180.9	108.8	91.3	93.7
1	92.8	72.3	70.2	63.0
0.5	48.7	51.4	60.3	36.3
0.25	29.5	38.1	55.4	24.5
0.125	20.5	31.3	53.0	18.3

Table 5.4: Execution time for decoding Lena (Million of CPU cycles).

5.5 Summary

In this chapter, we have presented a new simple wavelet algorithm with a run-length mode. This coder is simpler than previous proposals, but its compression performance is within the state-of-the-art. Although the complexity of a simple version of this algorithm is lower (in

general) than others (like JPEG 2000 and SPIHT), a run-length mode is introduced in order to decrease it, especially at low bit rates. This way, we have shown that our proposal is up to 10 times faster than Jasper, and 2 times faster than SPIHT.

Due to its lower complexity, the lack of memory overhead (contrary to SPIHT/SPECK and EBCOT, no extra memory is needed for auxiliary lists or partial bitstreams), the possibility of robustness (there is no inter-subband dependency), and high symmetry in coding and decoding execution times, it can be considered a good candidate for real-time interactive multimedia communications, allowing implementations both in hardware and software.

On the other hand, if inter-subband dependency is removed, higher coding efficiency is expected. In the next chapter, we will introduce the use of a quadtree structure as a faster and more efficient way of encoding wavelet coefficients.

Chapter 6

Lower tree wavelet image coding

In this chapter, a new image compression algorithm is proposed based on the efficient construction of wavelet coefficient lower trees (a type of quadtree in which all the coefficients are lower than a threshold). The main contribution of this lower-tree wavelet (LTW) encoder is that it utilizes coefficient trees, not only as an efficient method of grouping coefficients and removing inter-subband dependency, but also as a fast way of coding them. Thus, it presents state-of-the-art compression performance, while its complexity is lower than that of other wavelet encoders, like SPIHT and JPEG 2000. This fast execution is achieved by means of a simple two-pass coding and one-pass decoding algorithm. In addition, as in the algorithms proposed in the previous chapter, it does not need additional lists and therefore no memory overhead is introduced. A formal description of the algorithm is provided, so that an implementation can be performed straightforwardly. Numerical results show that our codec is faster than SPIHT and JPEG 2000 (up to three times faster than SPIHT and fifteen times faster than JPEG 2000), it achieves similar coding performance and presents higher symmetry.

6.1 Introduction

In the previous chapter, we used run-length coding as a fast method of grouping insignificant coefficients to encode them. Other wavelet-based encoders have previously used run-length coding with the same purpose (e.g., EBCOT [TAU00] uses run-length coding to reduce complexity). In this chapter, we propose a different structure, namely coefficient quadtrees

(similar to those described in Chapter 4), to accomplish the same objective more effectively, while coding efficiency is also improved by introducing inter-subband redundancy removal.

The key idea of the proposed algorithm is the use of wavelet coefficient trees as a fast method of efficiently grouping coefficients. In Chapter 4, we saw that tree-based wavelet coders have been widely used in the literature [SHA93] [SAI96] [XIO97] and they have evidenced good rate/distortion performance. However, we think that their excellent possibilities for fast processing of quantized coefficients has not been clearly shown so far. Coefficient trees are a simple way of grouping coefficients, reducing the total amount of symbols to be coded. This way, they not only achieve good compression performance but also fast processing. However, in the aforementioned encoders, coefficient trees are established with bit-plane coding (like in [SHA93] and [SAI96]), or the encoder introduces an optimization algorithm (such as [XIO97]) that hinder fast execution. As we did in the previous chapter, both strategies must be avoided in order to reduce complexity [1].

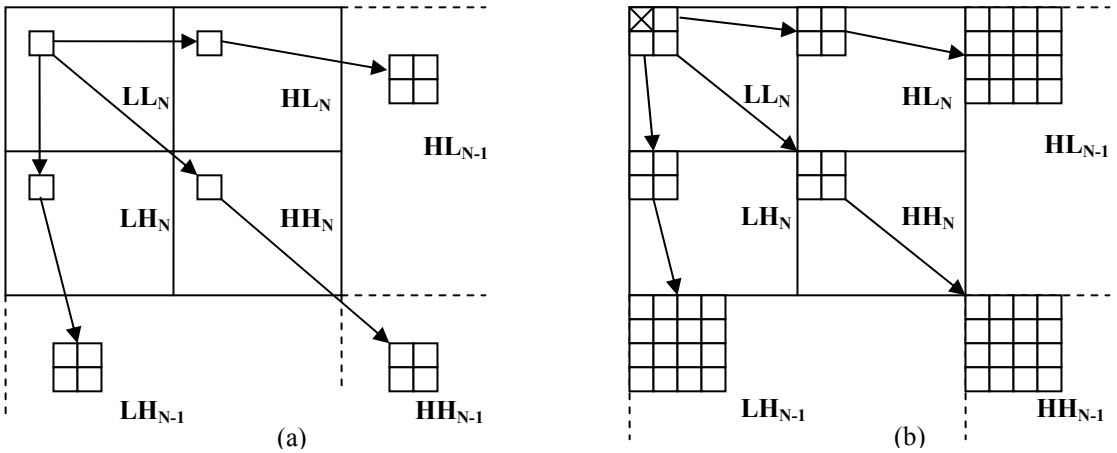


Fig. 6.1: (a) EZW-like and (b) SPIHT-like coefficient trees, focusing on the LL subband. The only difference between both types of trees is that a coefficient in the LL subband has three descendants in EZW, while in SPIHT the coefficients have four direct descendants (including those in the LL subband, except one each 2×2 coefficients, which has no offspring).

6.2 Two-pass efficient coding using lower trees

We saw in Chapter 4 that tree-based wavelet image encoders are proved to efficiently store

[1] Although the PROGRES encoder [CHO05] described in Chapter 4 is a low complexity tree-based image encoder, with neither bit plane coding nor optimization algorithms, it is a later work than the LTW algorithm presented in this chapter, and it borrows these principles from LTW (which in fact is referenced in that paper).

the wavelet transform coefficients in a dyadic decomposition, achieving good coding performance. In these algorithms, two stages can be established. The first one consists in encoding the significance map, i.e., the location and amount of bits required to represent the coefficients that will be encoded (significant coefficients). In the second stage, the significant coefficients are encoded, i.e. their sign and significant bits are stored. In Chapter 4, we referred to the first stage as dominant pass in EZW or sorting pass in SPIHT, and to the second stage as subordinate pass in EZW or refinement pass in SPIHT. However, both tree-based wavelet encoders use bit plane processing in the two stages, resulting in higher complexity.

The algorithm presented in this chapter is an extension of the one-pass algorithm described in the previous chapter (Algorithm 5.1). In this new algorithm, a tree-based structure is introduced in order to reduce inter-subband data redundancy. We call this structure lower-tree, since it is a quadtree in which all its coefficients are lower than a threshold. Actually, this structure is not different to that employed in EZW/SPIHT or SFQ. In EZW/SPIHT, a zero-tree is formed by coefficients that are insignificant at a certain bit-plane level, being considered 0 at that level. However, we can also consider that the coefficients in the zero-tree are lower than a threshold established by the bit-plane level. In SFQ, a zero-tree is somewhat different. It is formed by coefficients that have been quantized to 0 in a previous scalar quantization, although we can consider again that a coefficient is quantized to 0 when it is lower than a threshold value.

The use of coefficient trees not only reduces the redundancy among subbands but also is a simple and fast way of grouping coefficients. Therefore, it is presented in this section as an alternative to the run-length mode introduced in Algorithm 5.2.

As in the rest of tree-based coding techniques, coefficients from *C* are logically arranged as trees. In these trees, the direct descendants (or offspring) of a coefficient $c_{a,b}$ can be computed as follows:

$$if\ c_{a,b} \in LL_N \wedge a \in \{2k_1\} \wedge b \in \{2k_2\} : k_1, k_2 \in Z$$

$$offspring(c_{a,b}) = \{\phi\}$$

$$if\ c_{a,b} \in LL_N \wedge a \in \{2k_1 + 1\} \wedge b \in \{2k_2\} : k_1, k_2 \in Z,\ w = width(LL_N)$$

$$offspring(c_{a,b}) = \{c_{a+w,b},\ c_{a+w-1,b},\ c_{a+w,b+1},\ c_{a+w-1,b+1}\}$$

$$if\ c_{a,b} \in LL_N \wedge a \in \{2k_1\} \wedge b \in \{2k_2+1\} : k_1, k_2 \in Z,\ h = height(LL_N)$$

$$offspring(c_{a,b}) = \{c_{a,b+h},\ c_{a+1,b+h},\ c_{a,b+h-1},\ c_{a+1,b+h-1}\}$$

$$if\ c_{a,b} \in LL_N \wedge a \in \{2k_1+1\} \wedge b \in \{2k_2+1\} : k_1, k_2 \in Z,\ w = width(LL_N),\ h = height(LL_N)$$

$$offspring(c_{a,b}) = \{c_{a+w,b+h},\ c_{a+w-1,b+h},\ c_{a+w,b+h-1},\ c_{a+w-1,b+h-1}\}$$

$$if\ c_{a,b} \notin LL_N \wedge c_{a,b} \notin HL_1 \wedge c_{a,b} \notin LH_1 \wedge c_{a,b} \notin HH_1$$

$$offspring(c_{a,b}) = \{c_{2a,2b},\ c_{2a+1,2b},\ c_{2a,2b+1},\ c_{2a+1,2b+1}\}$$

$$if\ c_{a,b} \in HL_1 \vee c_{a,b} \in LH_1 \vee c_{a,b} \in HH_1$$

$$offspring(c_{a,b}) = \{\phi\}$$

The first four expressions define the offspring of a coefficient belonging to the LL_N subband; the fifth expression indicates that coefficients in a first-level subband are leaves and thereby have no offspring; and finally, the last expression gives the general case. This tree definition is introduced in SPIHT [SAI96] so that, if a coefficient has offspring, they always form a quadtree (in other words, it always has four children coefficients). The early tree definition employed by EZW [SHA93] differs in the way children coefficients are defined in the LL_N subband, with only three children per coefficient. Therefore, in EZW, the first four expressions are replaced by the following:

$$if\ c_{a,b} \in LL_N,\ w = width(LL_N), h = height(LL_N)$$

$$offspring(c_{a,b}) = \{c_{a+w,b},\ c_{a,b+h},\ c_{a+w,b+h}\}$$

The difference between both types of trees is illustrated graphically in Figure 6.1.

In our algorithm, we will use SPIHT-like trees, in which if a coefficient has offspring, the direct descendants always form a 2×2 block of coefficients, and the rest of descendant levels are attained by successively calculating the offspring of these direct descendants. This type of tree is chosen for our proposal because we want a 2×2 block of coefficients to share the same parent coefficient in all the subbands (except in LL_N, which has no parent coefficient) in order to ease significance propagation as we will see later.

The quantization used in this new algorithm is the same as in Algorithm 5.1 (see Appendix A for more details). This type of quantization allows constant quality, and in addition, a method to achieve approximate rate control from the quantization parameters is given in Appendix B.

In this tree-based algorithm, a coefficient is called lower-tree root if this coefficient and all its descendants are lower than $2^{rplanes}$. The set formed by all these coefficients forms a lower-tree. In the new algorithm, we use the label *LOWER* with a different meaning. It points out that a coefficient is the root of a lower-tree, and not simply an insignificant coefficient. The rest of coefficients in the lower-tree are labeled as *LOWER_COMPONENT*. On the other hand, if a coefficient is lower than $2^{rplanes}$ but it does not belong to a lower-tree (because it has at least one significant descendant), it is considered as an *ISOLATED_LOWER*.

6.2.1 Lower tree encoding algorithm

Once we have defined the basic concepts to understand the algorithm, we are ready to describe the encoding process. It is a two-pass algorithm. During the first pass, the wavelet coefficients are properly labeled according to their significance, and all the lower-trees are identified and formed. This coding pass is new with respect to Algorithms 5.1 and 5.2. In the second image pass, the coefficient values are coded in a similar way as in the second stage of Algorithm 5.1, although taking into account the new symbols introduced in this algorithm. Note that both the significance map and significant coefficients are encoded in the second pass in only one iteration.

The coding algorithm, Algorithm 6.1, is described in three parts, with the main body in Algorithm 6.1(a), and the first and second passes in 6.1(b) and 6.1(c) respectively.

function LowerTreeWaveletCoding( )
    1) Initialization:
        output *rplanes*
        output $maxplane = \max_{\forall c_{i,j} \in C} \left\{ \left\lceil \log_2 \left(\left| c_{i,j} \right| \right) \right\rceil \right\}$
    2) First image pass:
        LTWCalculateSymbols( )
    3) Second image pass:
        LTWOutputCoefficients( )
end of fuction

Algorithm 6.1(a): Lower tree coding. General description.

subfunction LTWCalculateSymbols( )
Scan the first level subbands (HH_1, LH_1 and HL_1) in 2×2 blocks.
For each block B_n
  if $|c_{i,j}| < 2^{rplanes}$ $\forall c_{i,j} \in B_n$
    set $c_{i,j}$ =*LOWER_COMPONENT* $\forall c_{i,j} \in B_n$
  else
    for each $c_{i,j} \in B_n$
      if $|c_{i,j}| < 2^{rplanes}$
        set $c_{i,j}$ =*LOWER*
Scan the rest of subbands (from level 2 to N) in 2×2 blocks.
For each block B_n
  if ($|c_{i,j}| < 2^{rplanes} \wedge$ descendant($c_{i,j}$)=*LOWER_COMPONENT*) $\forall c_{i,j} \in B_n$
    set $c_{i,j}$ =*LOWER_COMPONENT* $\forall c_{i,j} \in B_n$
  else
  for each $c_{i,j} \in B_n$
    if $|c_{i,j}| < 2^{rplanes} \wedge$ descendant($c_{i,j}$)=*LOWER_COMPONENT*
      set $c_{i,j}$ =*LOWER*
    if $|c_{i,j}| < 2^{rplanes} \wedge$ descendant($c_{i,j}$) $\neq$ *LOWER_COMPONENT*
      set $c_{i,j}$ =*ISOLATED_LOWER*
end of subfuction

Algorithm 6.1(b): Lower tree coding. Symbol computation.

subfunction LTWOutputCoefficients( )
  Scan the subbands (from N to 1, in 2×2 blocks)
  For each $c_{i,j}$ in a subband
    if $c_{i,j} \neq LOWER\_COMPONENT$
      if $c_{i,j} = LOWER$
        arithmetic_output *LOWER*
      else if $c_{i,j} = ISOLATED\_LOWER$
        arithmetic_output *ISOLATED_LOWER*
      else
        $nbits_{i,j} = \lceil \log_2(|c_{i,j}|) \rceil$
        if descendant($c_{i,j}$) $\neq LOWER\_COMPONENT$
          arithmetic_output $nbits_{i,j}$
        else
          arithmetic_output $nbits_{i,j}^{LOWER}$
        output $\text{bit}_{nbits_{i,j}-1}(|c_{i,j}|) \ldots \text{bit}_{rplane+1}(|c_{i,j}|)$
        output sign($c_{i,j}$)
end of subfuction

Note: $\text{bit}_n(c)$ is a function that returns the n^{th} bit of c.

Algorithm 6.1(c): Lower tree coding. Output the wavelet coefficients.

If we analyze the first pass (subfunction *LTWCalculateSymbols*()), we observe that labeling of lower-trees is performed in a recursive way, building lower-trees from leaves to root. In the first level subbands, coefficients are scanned in 2×2 blocks and, if the four coefficients are insignificant (i.e., lower than $2^{rplanes}$), they are considered part of the same lower-tree, being labeled as *LOWER_COMPONENT*. Then, when scanning higher level subbands, if a 2×2 block has four insignificant coefficients, and all their descendants are *LOWER_COMPONENT*, the coefficients in that block are also labeled as *LOWER_COMPONENT*, increasing the size of the lower-tree.

However, when at least one coefficient in a block is significant, the lower-tree cannot continue growing. In that case, each insignificant coefficient in the block is labeled as *LOWER* if all its descendants are *LOWER_COMPONENT*, otherwise the insignificant coefficient is labeled as *ISOLATED_LOWER* (note that significant coefficients do not need to be labeled).

In order to reduce memory overhead, labels are applied by overwriting the coefficient value by an integer value associated to the corresponding label, which must be outside the

possible range of significant coefficients (typically, by reusing the values in the quantized range [0.. $2^{rplanes}$]).

In the second pass (subfunction *LTWOutputCoefficients*()), all the subbands are explored from the N^{th} level to the first one. This is the order in which the decoder requires the coefficients in the input bitstream, and in addition, this order allows spatial scalability. Within a subband, all the coefficients are scanned in 2×2 blocks, and following a Morton order (in medium-sized clusters) to take advantage of data locality.

For each coefficient in a subband, if it is a lower-tree root or an isolated lower, the corresponding *LOWER* or *ISOLATED_LOWER* symbol is output. On the other hand, if the coefficient has been labeled as *LOWER_COMPONENT* no output is needed because this coefficient is already represented by the lower-tree to which it belongs.

A significant coefficient is coded in a similar way as in Algorithm 5.1. A "numeric symbol" indicating the number of bits required to represent that coefficient is arithmetically coded, and the significant bits and sign are raw coded. However, two types of "numeric symbols" are used according to the coefficient offspring. (a) A "regular numeric symbol" ($nbits_{i,j}$) shows the number of bits needed to encode a coefficient, (b) and a special "*LOWER* numeric symbol" ($nbits_{i,j}^{LOWER}$) not only indicates the number of bits of the coefficient, but also the fact that all its descendants are labeled as *LOWER_COMPONENT*, and thus they belong to a lower-tree not yet codified. This type of symbol is able to represent efficiently some special lower-trees, in which the root coefficient is significant and the rest of coefficients are insignificant[1]. The number of symbols needed to represent both sets of numeric symbols is $2\times(2^{maxplane}-2^{rplanes})$, therefore the arithmetic encoder must be initialized to handle at least this amount of symbols, along with two additional symbols: the *LOWER* and *ISOLATED_LOWER* symbols (recall that a *LOWER_COMPONENT* coefficient is never encoded).

An important difference between our tree-based wavelet algorithm and others like EZW [SHA93] and SPIHT [SAI96] is how the coefficient tree building process is detailed.

[1] Cho and Pearlman formalize in [CHO05b] the definition of a tree in which the root coefficient is significant and the rest are insignificant, as a degree-1 zerotree. At the same time, if all the coefficients in a tree are insignificant, it is considered a degree-0 zerotree. In EZW, only degree-0 zerotrees are encoded with a single symbol, while in LTW both degree-0 and degree-1 zerotrees can be encoded with a single symbol. For a study about a general degree-k zerotree model, and the coding benefits and limits of higher degree zerotrees, the reader is referred to [CHO05b].

Algorithm 6.1 includes a simple and efficient recursive method (the *LTWCalculateSymbols*() function) to determine if a coefficient has significant descendants in order to form coefficient trees. However, EZW and SPIHT leave it as an open and an implementation dependent aspect, which increases drastically the algorithm complexity if it is not carefully solved (for example, if searches for significant descendants are independently calculated for every coefficient whenever they are needed). Anyway, an efficient implementation of EZW and SPIHT would increase their memory consumption due to the need to store the maximum descendant coefficient for every coefficient, which can be obtained during a pre-search stage.

6.2.2 Lower tree decoding algorithm

The decoding algorithm performs the reverse process in only one pass, without the need to calculate the coefficient symbols, which are directly input from the encoder side.

The decoder is described in Algorithm 6.2; however, in order to understand how this algorithm works, some explanations should be made.

function LowerTreeWaveletDecoding( )
1) Initialization:
input *rplanes, maxplane*
2) Input coefficients (in a single pass):
Scan the subbands (from N to 1, in 2×2 blocks)
For each 2×2 block B_n
if $\exists \text{ascendant}(B_n) \wedge \text{ascendant}(B_n) = 0$
set $c_{i,j} = 0 \quad \forall c_{i,j} \in B_n$
else if $\exists \text{ascendant}(B_n) \wedge \text{ascendant}(B_n)$ is *ROOT-MARKED*
set $c_{i,j} = 0 \quad \forall c_{i,j} \in B_n$
remove *ROOT-MARK* from ascendant(B_n)
else
{
if $\exists \text{ascendant}(B_n) \wedge \text{ascendant}(B_n) =$ *ISOLATED_LOWER*
ascendant(B_n) = 0
for each $c_{i,j} \in B_n$
arithmetic_input $nbits_{i,j}$
if $nbits_{i,j} =$ *ISOLATED_LOWER*
set $c_{i,j}$ = *ISOLATED_LOWER*
else if $nbits_{i,j} =$ *LOWER*
set $c_{i,j}$ = 0
else
setbit $\text{bit}_{nbits_{i,j}}\left(|c_{i,j}|\right)$
input $\text{bit}_{nbits_{i,j}-1}\left(|c_{i,j}|\right) \ldots \text{bit}_{rplane+1}\left(|c_{i,j}|\right)$
setbit $\text{bit}_{rplane}\left(|c_{i,j}|\right)$
input sign($c_{i,j}$)
if $nbits_{i,j}$ has *LOWER* mark (i.e., it is $nbits_{i,j}^{LOWER}$)
insert *ROOT-MARK* to $c_{i,j}$
}
end of subfuction
Note: setbit $\text{bit}_n(c)$ is a function that sets the n^{th} bit of c

Algorithm 6.2: Lower tree decoding.

In the decoder, the wavelet subbands are recovered in decreasing order of level (i.e., from N to 1), providing spatial scalability if the inverse wavelet transform is applied as soon as an entire subband is decoded. The coefficients in each subband must be scanned in the same order used on the encoder side. The finer granularity of this scan is 2×2 coefficient blocks, so

that coefficients that share the same parent (i.e., sibling coefficients) are handled together. Note that all the 2×2 coefficient blocks have ascendant (parent) except those in the LL_N subband.

For all the insignificant coefficients, we set its value to 0, since their order of magnitude is unknown, and we only know that they are lower than $2^{rplanes}$. In our decoding algorithm, insignificant coefficients in a lower-tree are automatically propagated. When the parent of four sibling coefficients has been set to 0, all the descendant coefficients are also assigned a value of 0, so lower-trees are recursively generated. However, if an isolated lower coefficient has been decoded, it must not be propagated as a lower-tree. Hence, a different value must be assigned. For this case, we keep this coefficient as *ISOLATED_LOWER* until its 2×2 offspring coefficients are scanned. At that moment, we can safely update its value to 0 without risk of unwanted propagations because no more direct descendants of this coefficient will be scanned (in quadtrees, a parent only has four children).

On the other hand, if there is no tree propagation (the parent coefficient is not 0), a symbol is input for each coefficient in the 2×2 block. This symbol indicates the significance of that coefficient. If the decoded symbol indicates that a coefficient is significant ($nbits_{i,j}$), its sign and significant bits are also input, and its most significant non-zero bit is set to one. The bit in the position *rplanes* is also set to one in order to reduce the error interval of the recovered significant coefficients (see Appendix A).

Recall that there is a special "*LOWER* numeric symbol" ($nbits_{i,j}^{LOWER}$) for those trees in which the root coefficient is significant (similar to a degree-1 zerotree). When a symbol of this type is decoded, the coefficient must be marked (with *ROOT-MARK*) so that its offspring can spread correctly the lower-tree. To represent this mark, the least significant bit of a significant coefficient may be used, provided that *rplanes* >1 (because the first *rplanes* bits are decoded to 0, and hence the least significant bit is always 0).

51	42	-9	2	4	4	0	-1
25	17	10	11	3	1	0	2
12	3	3	-2	2	-2	-5	3
-9	-3	3	-3	0	3	-1	2
-4	1	1	-2	0	2	1	3
2	-3	0	2	1	-1	-1	-2
1	3	2	1	1	2	-3	1
-2	-3	3	-12	2	0	2	1

Fig. 6.2: 2-scale wavelet transform of an 8x8 example image.

6	**6**	**4**	*L*	**3**	**3**	*	*
5	$\mathbf{5}^L$	$\mathbf{4}^L$	**4**	*L*	*L*	*	*
4	*L*	*	*	*	*	**3**	*L*
$\mathbf{4}^L$	*I*	*	*	*	*	*L*	*L*
3	*L*	*	*	*	*	*	*
L	*L*	*	*	*	*	*	*
*	*	*L*	*L*	*	*	*	*
*	*	*L*	**4**	*	*	*	*

Fig. 6.3: Symbols resulting from applying Algorithm 6.2 to the example image of Figure 6.2.

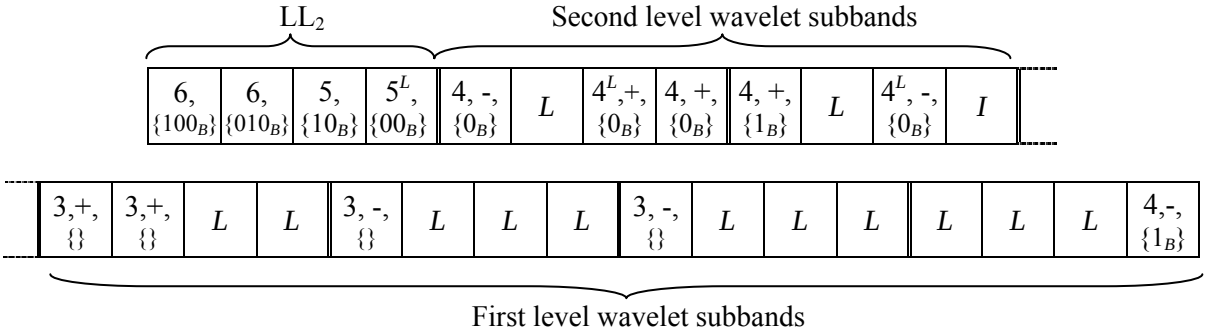


Fig. 6.4: Example image of Figure 6.2 encoded with Algorithm 6.2.

6.2.3 A Simple Example

Figure 6.2 shows a small 8×8 image that has been transformed using a 2-level DWT. In this section, we are going to apply our tree-based encoding algorithm using this very simple example. The wavelet coefficients have been previously quantized, and the *rplanes* parameter has been selected to be equal to 2. Regarding the *maxplane* parameter, it can be easily

computed as $maxplane = \lceil \log_2(51) \rceil = 6$.

When the coarser quantization is applied with *rplanes* = 2, the values within the interval $]2^2 \ldots -(2^2)[$ are absolutely quantized. Thus, all the significant coefficients can be represented using from 3 to 6 bits, and hence, the symbol set needed to represent the significance map is {**3**, **4**, **5**, **6**, $\mathbf{3}^L$, $\mathbf{4}^L$, $\mathbf{5}^L$, $\mathbf{6}^L$, *L*, *I* }. In this symbol set, special "*LOWER* numeric symbols" are marked with a superscript *L*, an *I* symbol represents an isolated lower coefficient, and an *L* indicates a regular lower symbol (the root of a lower-tree). Those coefficients that belong to a previously encoded lower-tree (i.e., those labeled as *LOWER_COMPONENT*) are not encoded and, in our example, they are represented with a star (*).

Figure 6.3 shows the symbols resulting from applying our algorithm to the example image. In the first image pass, the symbols are calculated. The scanning process begins with the first-level subbands in 2×2 blocks. When all the coefficients in a block are insignificant (in other words, they belong to the interval $]2^2 \ldots -(2^2)[$), they are marked with the *LOWER_COMPONENT* symbol (*), and they will not be encoded in the second pass. On the contrary, if a coefficient in the block is significant, those significant ones are encoded with a symbol that indicates the number of bits needed for its representation, whereas an insignificant coefficient is represented by a *LOWER* symbol.

In the second level, three special cases should be remarked. We label a coefficient as *ISOLATED_LOWER* when this coefficient is insignificant but its descendants are not *LOWER_COMPONENT*. In our example, an insignificant coefficient (–3) is labeled as isolated lower for this reason. On the other hand, if a coefficient is insignificant and its descendants are *LOWER_COMPONENT*, but it belongs to a block with at least one significant element, it is the root of the growing tree, and therefore it is labeled as *LOWER*, as it is shown in the second tree depicted in Figure 6.3. The last tree marked in the example is a significant coefficient whose descendants are all *LOWER_COMPONENT*. This type of lower tree with a significant root must be distinguished using the $\mathbf{4}^L$ symbol instead of a regular **4** symbol.

In our example, the HH_1 subband has been absolutely quantized and all their components are pointed out with a star. The following level, HH_2, is formed by only one 2x2 block. Since all the coefficients in this block are insignificant, and all their descendants are *LOWER_COMPONENT*, the tree continues growing, and the coefficients in this block are also marked with stars. Finally, the symbol $\mathbf{5}^L$ in the LL_2 subband will be used to represent

the entire lower-tree.

In the second image pass, the significant bits and signs, and all the symbols previously calculated (except *LOWER_COMPONENT*s) are output.

If we scan the subbands from the highest level (N) to the lowest one (1), in 2x2 blocks, from left to right and top to bottom, the resulting bitstream is the one illustrated in Figure 6.4. Note that the sign is not necessary for the LL subband since its coefficients are always positive.

6.3 Implementation considerations

As we have studied in previous chapters, implementation details and further adjustments can improve the performance of a compression algorithm. In this section, we give some guides for a successful implementation of the proposed Algorithms 6.1 and 6.2. Note that all the coding improvements introduced in this section should preserve fast processing and vice versa.

6.3.1 Analyzing the adaptive arithmetic encoder

Context coding has been widely used to improve the rate/distortion performance in image compression by creating different probability models depending on the already encoded neighbor coefficients. Although high-order context modeling presents high complexity [WU01], simpler context coding can be efficiently employed without noticeable increase in execution time. As we did in Algorithm 5.1, we propose the use of two contexts based on the significance of the left and upper coefficients, thus if they both are insignificant or close to insignificant, a different model is used for coding.

Adding a few more models in order to establish more significance levels (and thus more number of contexts) could improve compression efficiency. However, it would slow down the execution of the algorithm in a real implementation, mainly because of the context formation evaluation, and due to the higher memory requirements causing higher cache miss rate due to the introduction of new statistical tables.

Recall that *maxplane* indicates the number of bits needed to represent the highest coefficient in the wavelet decomposition. This value (along with *rplanes*) determines the number of symbols needed for the arithmetic encoder. However, coefficients in different subbands tend to be different in magnitude order, so this parameter can be set specifically for

every subband level. In this manner, the arithmetic encoder is initialized exactly with the number of symbols needed in every subband, which increases the coding efficiency.

Slight improvement can be attained by arithmetically encoding the significant bits and sign of the wavelet coefficients (instead of raw encoding). In order to reduce the overhead impact in the execution time, a faster binary entropy coder employed by JPEG 2000, called MQ-coder [SLA98], can be used. This fast-approximation can be applied not only for encoding the coefficient bits, but also for storing the rest of symbols, using the extension from binary to general coding purpose proposed in [TAU02], at the cost of lower compression performance. Observe that none of these optimizations have been performed in our implementation, and a regular arithmetic encoder has been used.

6.3.2 Analyzing the quantization process

Some considerations can be made related to the quantization process. The scalar uniform quantization can be applied within the first image scan so that the cache efficiency is improved, or even within the filter normalization if the DWT is computed using the lifting scheme. This quantization (or filter normalization in lifting) implies a floating-point multiplication for every wavelet coefficient, and a rounding operation. However, this multiplication and rounding can be avoided for the insignificant coefficients, since they are going to be discarded anyway. Equations from Appendix A can be used to determine the necessary threshold value to avoid scalar quantization of the insignificant coefficients.

Related to the coarser quantization, consider the case in which three coefficients in a block are insignificant, and the fourth value is very close to insignificant. In this case, we can consider that the entire block is insignificant and all its coefficients can be labeled as *LOWER_COMPONENT*. The slight error introduced is compensated by the saving in bits. Another similar strategy is saturating some coefficient values, from a perfect power of two to the same value decreased in one (for example, from 0010 0000b to 0001 1111b). In this manner, the error introduced for that coefficient is in magnitude only one, whereas it needs one less bit to be stored, resulting in higher compression efficiency.

6.4 Numerical results

We have implemented the Lower-Tree Wavelet (LTW) encoder and decoder algorithms in order to test their performance. They have been implemented using standard C++ language

(without using assembly language or platform-dependant features), and all the simulation tests have been performed on a regular Personal Computer (500 MHz Pentium Celeron processor with 256 KB L2 cache), generating real images files that contain the compressed images, including all the file-headers needed for self-containing decompression. The reader can easily perform new tests using the LTW implementation available at *http://www.disca.upv.es/joliver/thesis*.

In order to compare our algorithm with other wavelet encoders, the classical Lena, Goldhill and Barbara images (monochrome, 8bpp, 512x512) and other two images from the JPEG 2000 test bed, Café and Woman (monochrome, 8bpp, 2560x2048), have been selected. On the one hand, the widely used Lena image (from the USC) allows us to compare the LTW with practically all the published algorithms, since results are commonly expressed using this image. On the other hand, the Café and Woman images are less blurred and more complex than Lena and represent pictures taken with a 5-Megapixel high definition digital camera.

Table 6.1 provides R/D performance when compressing the Lena image with several wavelet-based image encoders. We can see that LTW achieves better results than the other coders do, including the JPEG 2000 standard, whose results have been obtained using Jasper [ADA02], an official implementation used as reference software in the ISO/IEC 15444-5 standard. When LTW is compared with the run-length wavelet (RLW) encoder described in the previous chapter (Algorithm 5.2), we see that the inter-subband removal introduced by lower-trees causes a PSNR increase of about 0.25 dB.

Results for the rest of images are shown in Table 6.2. In this Table, only SPIHT, SPECK and Jasper have been compared with RLW and LTW, since the compiled versions of the rest of coders have not been released, or results are not published for these images.

We can observe that R/D performance for Café is still higher using our algorithm, although Jasper performs similarly to LTW. For Woman and Goldhill images, we can see that our algorithm exceeds in performance the rest of encoders.

On the other hand, Jasper encodes high-frequency images, like Barbara, better than LTW (and in general the rest of tree-based encoders) at high bit rates. Two reasons may explain this.

First, note that when high-frequency images are encoded at high bit rates, many coefficients in high-frequency subbands are significant, and hence our algorithm is not able to build large lower-trees. This effect is even more serious in SPIHT, because many groups

are partitioned very early in the coding process in order to find significant coefficients in these subbands, and in later passes, the significance test has to be encoded for each group, increasing the size of the compressed image.

Second, in Chapter 4 we mentioned that JPEG 2000 includes many contexts, and it results in higher performance for very detailed images. In our experiments, we have observed that if more than two contexts are used in LTW, the R/D performance for Barbara is close to the results shown with Jasper, but at the cost of higher execution time.

codec\ rate	EZW	SPIHT	Stack Run	Embedded Run-length	SPECK	Jasper/ JPEG2000	RLW	LTW
2	n/a	45.07	n/a	n/a	n/a	44.62	45.30	45.47
1	39.55	40.41	n/a	40.28	40.25	40.31	40.31	40.50
0.5	36.28	37.21	36.79	37.09	37.10	37.22	37.12	37.35
0.25	33.17	34.11	33.63	34.01	34.03	34.04	34.02	34.30
0.125	30.23	31.10	n/a	n/a	n/a	30.84	31.00	31.26

Table 6.1: PSNR (dB) with different bit rates and coders using Lena (512×512).

	Café (2560×2048)					*Woman* (2560×2048)				
codec\ rate	SPIHT	SPECK	Jasper/ JP2K	RLW	LTW	SPIHT	SPECK	Jasper/ JP2K	RLW	LTW
2	38.91	38.75	39.09	38.75	39.09	43.99	43.73	43.98	44.05	44.15
1	31.74	31.47	32.04	31.62	32.02	38.28	38.07	38.43	38.28	38.49
0.5	26.49	26.31	26.80	26.51	26.85	33.59	33.46	33.63	33.53	33.80
0.25	23.03	22.87	23.12	23.01	23.24	29.95	29.88	29.98	29.91	30.14
0.125	20.67	20.61	20.74	20.62	20.76	27.33	27.34	27.33	27.32	27.50
	Goldhill (512×512)					*Barbara* (512×512)				
codec\ rate	SPIHT	SPECK	Jasper/ JP2K	RLW	LTW	SPIHT	SPECK	Jasper/ JP2K	RLW	LTW
2	42.02	n/a	41.92	41.85	42.17	42.65	n/a	43.13	42.63	42.87
1	36.55	36.36	36.53	36.49	36.74	36.41	36.49	37.11	36.54	36.72
0.5	33.13	33.03	33.19	33.08	33.32	31.39	31.54	32.14	31.66	31.76
0.25	30.56	30.50	30.51	30.47	30.67	27.58	27.76	28.34	27.95	28.07
0.125	28.48	n/a	28.35	28.43	28.60	24.86	n/a	25.25	25.12	25.24

Table 6.2: PSNR (dB) with different bit rates and wavelet-based image encoders for Café, Woman, Goldhill and Barbara.

The main advantage of the LTW algorithm is its lower complexity. Table 6.3 shows that our algorithm greatly outperforms SPIHT and Jasper in terms of execution time. For medium sized images (512×512), our encoder is from 3.25 to 11 times faster than Jasper, whereas LTW decoder executes from 1.5 to 2.5 faster than Jasper decoder, depending on the bit rate. In the case of SPIHT, our encoder is from 2 to 2.5 times faster, and the decoding process is from 1.7 to 2.1 times faster.

With larger images, like Café (2560×2048), the advantage is greater. The LTW encoder is from 5 to 16 times faster than Jasper and the decoder is from 1.5 to 2.5 times faster. With respect to SPIHT, our algorithm encodes Café from 1.7 to 3 times faster, and decodes it from 1.5 to 2.5 times faster.

We also have included in this table execution times for RLW, showing that the use of lower-trees, not only improves compression efficiency, but also reduces the execution time, due to the simple algorithm proposed to propagate the significance of the trees. Moreover, the decoder is even faster because the first pass to compute symbols is not needed.

As in the previous chapter, in these tables we have only evaluated the coding and decoding processes, and not the transform stage, since the wavelet transform used is the same in all the cases: the popular Daubechies 9/7 bi-orthogonal wavelet. Other wavelet transforms, like Daubechies 23/25, have shown better compression performance. However, this improvement is achieved by using more filter taps, and thus increasing the execution time of the wavelet transform.

Fast execution is not the only desirable feature related to the complexity of an encoding/decoding algorithm, also the symmetry of both processes is an important issue, most of all in real time applications involving both processes. Table 6.4 shows the high symmetry of LTW. It is more symmetric than SPIHT and much more symmetric than Jasper. Only at very low bit rates, the LTW is slightly asymmetric, due to the computation of the symbols, which is performed only at the encoder side. Although other operations are performed only on the decoder side (e.g., evaluating the marks of the parent coefficient), they are less complex.

	Lena coding (512×512)				*Lena decoding* (512×512)			
Codec\ rate	SPIHT	Jasper / JP2K	RLW	LTW	SPIHT	Jasper / JP2K	RLW	LTW
2	210.4	278.5	95.4	86.2	180.9	108.8	93.7	85.0
1	119.4	256.1	61.2	51.3	92.8	72.3	63.0	47.1
0.5	72.3	238.2	37.0	32.9	48.7	51.4	36.3	27.1
0.25	48.7	223.4	25.9	24.0	29.5	38.1	24.5	17.4
0.125	36.8	211.3	20.3	19.1	20.5	31.1	18.3	12.3
	Café coding (2560×2048)				*Café decoding* (2560×2048)			
codec\ rate	SPIHT	Jasper / JP2K	RLW	LTW	SPIHT	Jasper / JP2K	RLW	LTW
2	4368.7	7393.1	1651.2	1494.7	3775.7	2373.0	1721.1	1505.6
1	2400.1	6907.6	1138.5	963.4	1935.3	1475.2	1178.0	911.6
0.5	1399.3	6543.9	808.1	654.3	1024.1	991.8	815.2	569.5
0.25	889.8	6246.2	597.5	476.7	586.9	763.6	570.0	366.6
0.125	624.5	6058.2	460.6	378.8	372.1	635.3	409.9	253.4

Table 6.3: Execution time comparison for Lena and Café (time in Million of CPU cycles).

	Lena (512×512)			*Café* (2560×2048)		
codec\ rate	SPIHT	Jasper / JPEG 2000	LTW	SPIHT	Jasper / JPEG 2000	LTW
2	0.86	0.39	1.00	0.86	0.32	1.00
1	0.78	0.28	0.92	0.81	0.21	0.95
0.5	0.67	0.22	0.82	0.73	0.15	0.87
0.25	0.61	0.17	0.73	0.66	0.12	0.77
0.125	0.56	0.15	0.65	0.59	0.10	0.67

Table 6.4: Symmetry index (decoding time/coding time) for SPIHT, Jasper/JPG2K and LTW.

Besides compression performance and complexity, the third major issue that we consider in an image encoder is its memory requirements. Our wavelet encoder and decoder presented in this chapter and in the previous chapter are able to perform in-place processing of the wavelet coefficients, and thus they do not need to handle extra lists or other memory consuming structures. This way, only 21 Mbytes are needed to encode the Café image using LTW (notice that 20 Mbytes are needed to completely store the image in memory using integer type in a 32-bit processor), whereas SPIHT and Jasper require 42 Mbytes and 64 Mbytes respectively[1].

[1] Results obtained with the Windows XP task manager, peak memory usage column.

6.5 Summary

In this chapter, we have presented a new wavelet image encoder based on the construction and efficient coding of wavelet lower-trees (LTW). Its compression performance is within the state-of-the-art, achieving better results than other popular algorithms (SPIHT is commonly improved in 0.2-0.4 dB, SPECK in 0.3-0.5 dB, and JPEG 2000 with Lena in 0.35 dB as mean value).

However, we have shown that the main contribution of this algorithm is its lower complexity. Depending on the image size and bit-rate, it is able to encode an image up to 15 times faster than Jasper and 3 times faster than SPIHT. Moreover, other wavelet encoders, like SFQ and high-order context modeling coders, are even slower than the ones compared here.

If we compare the lower-tree algorithm introduced in this chapter with the run-length encoder proposed in the previous chapter, this new algorithm is more efficient (0.25 dB with Lena, and 0.1-0.4 dB in general, depending on the features of the image, and the compression ratio), since it takes advantage of the parent-children dependency occurring in coefficient trees. In addition, both the new encoder and decoder are faster than the run-length version because coefficient trees are built in a very fast way, and coding of coefficients is performed straightforwardly.

Although memory consumption is reduced to the amount of memory needed to store the original image, without additional lists (like in SPITH) or partial bitstreams (like in JPEG 2000), it can be greatly reduced if this algorithm is applied along with the DWT algorithm proposed in Chapter 3. On the other hand, execution time can be reduced if adaptive arithmetic coding is replaced by Huffman coding, at the expense of lower compression efficiency. All these alternatives (and some others) are studied and assessed in the next chapter.

Chapter 7

Advanced coding: low memory usage and very fast coding

Reduced memory usage and fast image coding are two of the main objectives of this thesis. The wavelet encoders proposed in the two previous chapters perform in-place processing. Therefore, only the amount of memory required to hold the image in memory is used, and no extra memory is needed. However, a much more efficient use of memory can be made if an image is input line-by-line, being transformed with the algorithm presented in Chapter 3, and encoded as soon as the wavelet coefficients are obtained. A complete image encoder that combines the LTW algorithm with the DWT of Chapter 3 is presented in this chapter. Afterwards, we introduce a very fast Huffman-based modification of LTW, which largely reduce the execution time, at the expense of loss in coding efficiency. This variation is faster and more efficient than other very fast wavelet encoders described in Chapter 4, like PROGRES (our proposal is from 4 to 9 times faster in coding, and surpasses it in up to 0.5 dB in PSNR). Finally, we evaluate the coding efficiency of our encoder used in lossless coding, and we compare these results with JPEG 2000 operating in lossless mode, and LOCO-I, a lossless compression algorithm used within the JPEG-LS standard.

7.1 Coding with low memory consumption

In Chapter 3, we presented an algorithm to compute the wavelet transform with very low memory requirements. Although this wavelet transform is general-purpose, some wavelet encoders cannot use it, because an important restriction introduced by the line-based wavelet

transforms is that only a part of each subband is available at every moment, and therefore we needed coding schemes that do not require global knowledge of the image and do not perform several image scans. Hence, some of the wavelet encoders described in Chapter 4 cannot be used with this efficient wavelet transform (e.g., EZW, SPIHT and SPECK) and thus, these encoders do not allow low memory implementations. Some other encoders, like EBCOT, can employ this DWT but require post-processing of the coefficients (in particular, an optimization algorithm and coefficient reordering). On the other hand, the run-length (RLW) and tree-based (LTW) image encoders presented in Chapter 5 and 6 can be easily used along with the wavelet transform of Chapter 3, because coefficients can be encoded as soon as they are visited, in only one pass. In this section, we will see how to adapt both proposals to greatly reduce their memory usage. Then, some numerical results are given in order to evaluate the amount of memory reduction, and how it affects the coding efficiency and the execution time of the algorithms.

7.1.1 Run-length coding with low memory usage

In the wavelet transform proposed in Chapter 3, once a subband line is calculated, it has to be encoded as soon as possible in order to release memory and reduce memory consumption. However, entropy coders need to exploit local similarity in the image to be efficient. Therefore, better compression performance can be achieved if we group subband lines in an encoder buffer. These buffers store the lines released by the DWT and group them before the coding stage. This way, when we consider that there are enough lines in a buffer to perform an efficient compression, the run-length algorithm proposed in Chapter 5 is called (Algorithm 5.2).

However, some changes must be done in this algorithm so that it can be incorporated in this efficient wavelet transform. The main changes are:

(1) Global knowledge of the image is no longer available, and therefore the *maxplane* parameter cannot be computed in an initialization stage. Instead, an estimation of the highest coefficient that may appear should be made, mainly depending on the type of wavelet normalization and the pixel resolution of the source image (in bpp). Finally, to ensure the correctness of the encoder, an escape code should be used for values outside the predicted range.

(2) Whole subbands cannot be encoded at once. The encoding process is now interleaved,

and fragments of various subbands at various levels are progressively encoded. Therefore, a different run count (*run_length* variable) must be hold for each subband level to be able to continue run counts at the same level. Furthermore, the adaptive arithmetic encoder must handle several probability models simultaneously, one for each subband level. Therefore, when the encoder buffer is full, the wavelet transform algorithm calls the run-length encoder in order to encode the lines from the buffer and release them. In this call, two function parameters are needed: the buffer to be encoded and the corresponding subband level (to determine the probability model and the *run_length* variable to be used).

(3) Now, in the run-length encoder, the coefficients in the buffer are scanned column by column to exploit their locality.

(4) Since coefficients from different subband levels are interleaved (due to the computation order of the proposed wavelet transform), instead of a single bitstream, we should generate a different bitstream for every subband level. These bitstreams can be held in memory or saved in secondary storage, and are employed to form the final ordered bitstream.

(5) The rate control method of Appendix B cannot be directly used in this encoder due to the lack of global knowledge of the image (including the entropy of the entire image). However, a partial analysis of each encoder buffer (including the entropy of this part) is still possible, allowing an adaptive quantization process to achieve the target bitrate. Note that with this adaptive method, the image quality is not constant.

7.1.1.1 Tradeoff between coding efficiency and, speed and memory requirements

The proposed algorithm can be tuned depending on the final application requirements. Thus, some parameters can be adjusted to improve the compression efficiency at the expense of slightly higher memory requirements or execution time. This way, the number of lines in every encoder buffer can be 8 for a good R/D performance, but coding efficiency can be improved with 16 lines, increasing the memory requirements. Another parameter that can be tuned is the *enter_run_mode* variable of Algorithm 5.2. When this parameter is increased, larger run-lengths are encoded by successive *LOWER* symbols, which results slower but a bit more efficient in R/D performance. Another tradeoff between compression efficiency and complexity is the use of arithmetic coding (with contexts) instead of raw coding to encode the sign of the coefficients, since a dependence among the sign of the wavelet coefficients exists inside a subband. In general, each of these improvements may increase the PSNR of an image encoded at 1 bpp in about 0.1 dB, while the last two improvements increase the execution

time in about 20% each one.

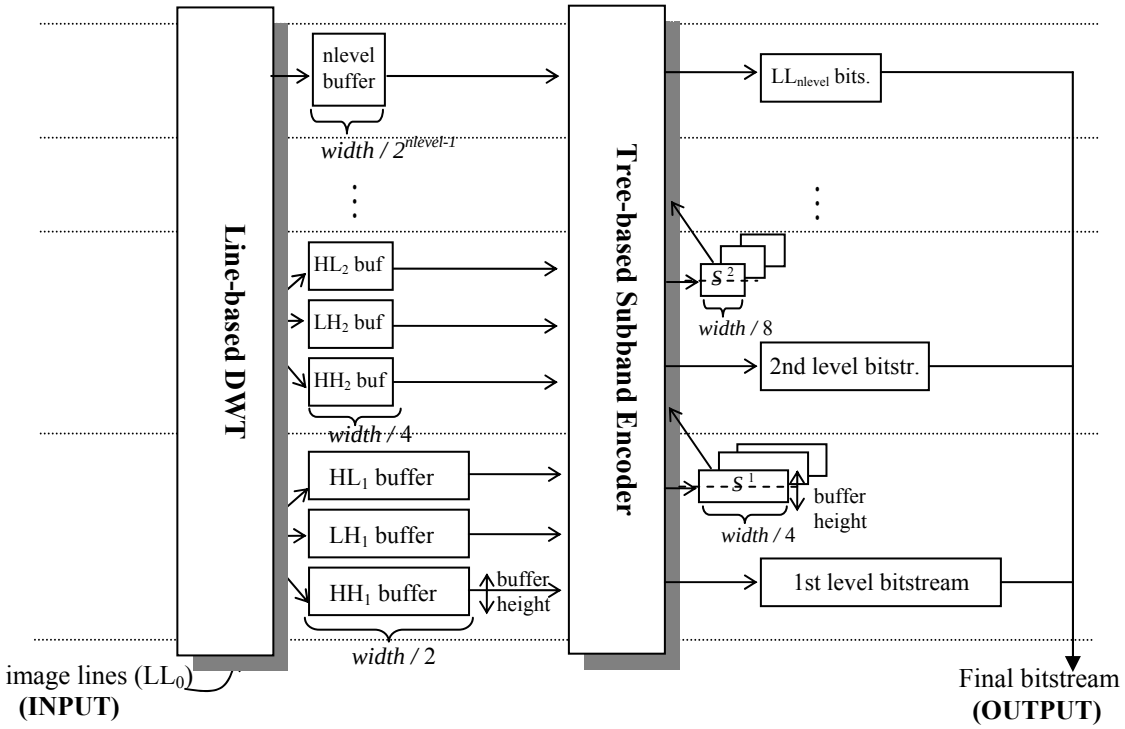


Fig. 7.1: Overview of the proposed tree-based encoder with efficient use of memory.

7.1.2 Fast tree-based coding with efficient use of memory

The wavelet transform proposed in Chapter 3 can also be applied with the lower-tree algorithm presented in Chapter 6, although the tree structure of this encoder requires a more careful treatment, and therefore, this adapted algorithm is described with more detail in this section.

Figure 7.1 shows our overall system. It is similar to the run-length encoder described in the previous section, and the changes aforementioned must be made as well. But in addition, a binary significance map (that will be described later) is needed at each level. In this scheme, when the DWT releases subband lines, they are inserted into an encoder buffer, which is passed to the tree-based encoder once it is full.

An important different between this version and the LTW presented in Chapter 6 is that the new adapted encoder must process coefficients in only one-pass, and therefore symbols must be computed and output at once. However, in this case, it is not an important drawback because the order of the wavelet coefficients is later arranged for the decoder with an

independent bitstream generation at each level (see the fourth change in the previous section).

The adapted encoding algorithm is formally described in Algorithm 7.1. Let us see it with some detail. The encoder has to determine if each 2×2 block of coefficients in the buffer is part of a lower-tree. If the four coefficients in the block are lower than the quantization threshold $2^{rplanes}$, and their descendant offspring are also insignificant, they are part of a lower-tree and do not need to be encoded. In order to know if their offspring are significant, we need to hold a binary significance map of every encoder buffer (S^L in the figure) because the encoder buffer is overwritten by the wavelet transform once it is encoded, and hence the significance for their ascendant coefficients is not automatically held. Obviously, this significance map was not needed in the original LTW because the whole image was available for the encoder.

The width of each significance map is sized half the width of the encoder buffer that it represents, since the significance is held for each 2×2 block. The height is not the half but the same as the buffer height because each buffer at a level l is encoded with double frequency compared to the l+1 level. Therefore, the fist half of the buffer at level l (the lines from 0 to buffer height/2) and the second half (from buffer height/2 to buffer height) are used alternatively to encode the significance of the encoder buffers at level l, and this complete buffer is used later as a reference to encode level l+1.

The significance of a block can be held with a single bit. Therefore, the memory required for these significance maps is almost negligible when compared with the rest of buffers.

As in Algorithm 6.1, when there is a significant coefficient in the 2×2 block or in its descendant coefficients, we need to encode each coefficient separately. Recall that in this case, if a coefficient and all its descendants are insignificant, we use the *LOWER* symbol to encode the entire tree, but if it is insignificant, and the significance map of its four direct descendant coefficients shows that it has a significant descendant, the coefficient is encoded as *ISOLATED_LOWER*. Finally, when a coefficient is significant, it is encoded with a *numeric symbol* along with its significant bits and sign.

It is important to note that trees are built from leaves to roots, spreading the significance of the coefficients. Fortunately, this is exactly the order in which the line-based DWT algorithm described in Chapter 3 generates the wavelet coefficients. For example, the first two lines released by the wavelet transform are from the first subband level, and then, the following line is computed from the second level. Observe that the coefficients in this line are

exactly all the ascendant coefficients of those computed in the two first lines. This order remains for the rest of the wavelet transform computation at all the levels, i.e., as we get two lines at a level *l*, we then get its parents at *l*+1 before getting two more lines at *l*.

At the last level (*N*), the tree cannot be propagated upward, and for this reason, we always encode all the coefficients at this level. Moreover, as in the previous section, we can keep the compressed bit-stream in memory, which allows us to invert the order of the bitstream for the inverse procedure.

function SubbandCode(*level* , *Buffer,* $S^{level-1}$, S^{level})
Scan the *Buffer* in 2×2 blocks ($B_{x,y}$) (and column by column)
for each block $B_{x,y} = \{c_{2x,2y}, c_{2x+1,2y}, c_{2x,2y+1}, c_{2x+1,2y+1}\}$
  if $level \neq N \wedge \left(c_{i,j} < 2^{rplanes} \wedge S_{i,j}^{level-1}\right.$ **is** *Insignif*. $\left.\forall c_{i,j} \in B_{x,y}\right)$
    set $S_{x,y}^{level} = Insignif.$
  else
    set $S_{x,y}^{level} =$ S*ignif*.
    for each $c_{i,j} \in B_{x,y}$
      if $|c_{i,j}| < 2^{rplanes}$
        if $S_{i,j}^{level-1}$ **is** *Insignif*.
          arithmetic_output *LOWER*
        else
          arithmetic_output *ISOLATED_LOWER*
      else
        $nbits_{i,j} = \lceil \log_2(|c_{i,j}|) \rceil$
        if $S_{i,j}^{level-1}$ **is** *Insignif*.
          arithmetic_output $nbits_{i,j}^{LOWER}$
        else
          arithmetic_output $nbits_{i,j}$
        output $\text{bit}_{nbits_{i,j}-1}(|c_{i,j}|) \ldots \text{bit}_{rplane+1}(|c_{i,j}|)$
        output $\text{sign}(c_{i,j})$
      endif
  endif
end of function

Note: $\text{bit}_n(c)$ is a function that returns the n^{th} bit of *c*.

Algorithm 7.1: Lower tree wavelet coding with reduced memory usage.

7.1.3 Numerical results

As in the rest of this thesis, we have implemented the proposed encoders in ANSI C language to compare them with the rest of proposals and with the state-of-the-art wavelet image encoders SPIHT and JPEG 2000/Jasper. These implementations were compiled with Visual C++ 6.0 and are available at *http://www.disca.upv.es/joliver/thesis*.

Lena (512x512)						
Codec\ rate	SPIHT	Jasper/ JP2K	Original RLW	Modified RLW	Original LTW	Modified LTW
1	40.41	40.31	40.31	40.23 (+0.14)	40.50	40.45 (+0.07)
0.5	37.21	37.22	37.12	37.05 (+0.10)	37.35	37.29 (+0.05)
0.25	34.11	34.04	34.02	33.95 (+0.08)	34.30	34.23 (+0.03)
0.125	31.10	30.84	31.00	30.93 (+0.04)	31.26	31.23 (+0.00)
Barbara (512x512)						
Codec\ rate	SPIHT	Jasper/ JP2K	Original RLW	Modified RLW	Original LTW	Modified LTW
1	36.41	37.11	36.54	36.47 (+0.35)	36.72	36.58 (+0.23)
0.5	31.39	32.14	31.66	31.61 (+0.29)	31.76	31.63 (+0.16)
0.25	27.58	28.34	27.95	27.90 (+0.22)	28.07	27.95 (+0.06)
0.125	24.86	25.25	25.12	25.11 (+0.08)	25.24	25.16 (+0.03)
Woman (2560x2048)						
Codec\ rate	SPIHT	Jasper/ JP2K	Original RLW	Modified RLW	Original LTW	Modified LTW
1	38.28	38.43	38.28	38.28 (+0.21)	38.49	38.46 (+0.11)
0.5	33.59	33.63	33.53	33.57 (+0.15)	33.80	33.77 (+0.05)
0.25	29.95	29.98	29.91	29.96 (+0.08)	30.14	30.13 (+0.02)
0.125	27.33	27.33	27.32	27.36 (+0.04)	27.50	27.52 (-0.03)
Café (2560x2048)						
Codec\ rate	SPIHT	Jasper/ JP2K	Original RLW	Modified RLW	Original LTW	Modified LTW
1	31.74	32.04	31.62	31.63 (+0.26)	32.02	31.96 (+0.12)
0.5	26.49	26.80	26.51	26.51 (+0.16)	26.85	26.82 (+0.06)
0.25	23.03	23.12	23.01	22.98 (+0.12)	23.24	23.24 (+0.03)
0.125	20.67	20.74	20.62	20.61 (+0.06)	20.76	20.79 (+0.01)

Table 7.1: PSNR (dB) comparison with different bit rates and coders for the evaluated images (Lena, Barbara, Woman and Café). The numbers in parenthesis correspond to the increase in performance if the R/D improvements discussed in Subsection 7.1.1.1 are applied.

Table 7.1 shows a compression comparison for the images Lena, Barbara, Woman and Café. This table indicates that the need to encode each subband in fragments (determined by the size of the encoder buffer) causes a very slight decrease in PSNR, from the original RLW and LTW algorithms to the new versions proposed in this section. This decrease is less than 0.05 dB in most cases. Furthermore, this small loss in R/D performance can be avoided if the improvements discussed in Subsection 7.1.1.1 are applied. In this table, the increase in PSNR introduced by these improvements is shown in parenthesis. With respect to the rest of the table, a comparison between RLW/LTW with SPIHT and JPEG 2000 was already made in the previous two chapters and thus, the reader is referred to these chapters for an analysis.

codec\ rate	Compres-sed image	SPIHT	Jasper/ JP2K	Original RLW	Modified RLW	Modified RLW with bitream in memory	Original LTW	Modified LTW	Modified LTW with bitstream in memory
1	640	42,888	62,768	21,180	1,017	1,657 (+180)	21,188	1,096	1,736 (+216)
0.5	320	35,700	62,240	21,180	953	1,273 (+180)	21,188	1,032	1,352 (+216)
0.25	160	31,732	61,964	21,180	953	1,113 (+180)	21,188	1,032	1,192 (+216)
0.125	80	28,880	61,964	21,180	937	1,017 (+180)	21,188	1,016	1,096 (+216)

Table 7.2: Total amount of memory (in KB) required to encode the Woman image using several encoding algorithms. The numbers in parenthesis correspond to the extra memory that is necessary if the R/D improvements are not used.

As expected, the comparison in which our modified encoders clearly outperform both SPIHT and the evaluated implementation of JPEG 2000 (Jasper) is the memory consumption. Table 7.2 shows that for a 5-Megapixel image, both proposals require between 25 and 40 times less memory than SPIHT, and more than 60 times less memory than Jasper/JPEG 2000. When comparing both modified versions between them, the tree-based version requires somewhat more memory (about 100 KB) due to the need to hold the significance map.

In this table, the sixth and the last columns refer to the cases in which the entire bitstream (i.e., the compressed image) is held in memory while it is generated. Recall that having a different bitstream for each level eases the decompression process, since the order in the inverse transform is just the reverse of the order in the forward one. Note that we have

estimated that the amount of memory needed for a single process (written in C and running under Windows XP) is about 650 KB, and therefore the data memory employed by each encoder is just the remaining memory to the total amount shown in Table 7.2.

Codec\ rate	SPIHT	Jasper/ JP2K	Original RLW	Modified RLW	Original LTW	Modified LTW
1	3,669	23,974	1,832	1,214 (+587)	1,563	1,067 (+325)
0.5	2,470	23,864	1,490	860 (+377)	1,257	760 (+183)
0.25	1,939	23,616	1,289	657 (+259)	1,087	591 (+114)
0.125	1,651	23,563	1,169	532 (+197)	993	503 (+75)

Table 7.3: Execution time (in Million of CPU Cycles) needed to encode Woman at various bitrates. The numbers in parenthesis correspond to the additional complexity introduced when R/D improvements are applied.

Finally, in table 7.3 we compare the second main issue that concerns us in this thesis, the complexity of these algorithms. Now, the execution time includes the wavelet transform computation since it is not separable from the coding algorithm in the new modified proposals. Only the encoding time is given because the decoding time is very similar for our modified proposals due to the high symmetry of both methods. This table shows that our proposals are faster than SPIHT and JPEG 2000, for the reasons given in the previous chapters. Compared with the original versions, the new variations are able to encode an image, line-by-line, in only one-pass, applying the DWT and the coding stage interleaved. Thus, memory access is optimized (mainly in the wavelet transform computation) and thereby these versions are up to two times faster than the original ones. If both proposals are compared, the tree-based encoder is still faster than the run-length version. Finally, observe that if the improvements of Subsection 7.1.1.1 are applied, the execution time increases (mainly at higher bit rates), most of all in the run-length encoder, since the *enter_run_mode* is increased, and then, more single *LOWER* symbols are arithmetically encoded.

7.2 Very fast coding of wavelet lower trees

In this section, a very fast Huffman-based variation of the Lower-Tree Wavelet (LTW) image encoder is presented. This alternative encoder serves to largely reduce the execution time, at

the expense of loss in coding efficiency. Furthermore, this encoder could be used along with the modifications described in the previous section, achieving an extremely fast encoder with reduced memory usage.

7.2.1 Proposed modifications

The main change proposed for Algorithm 6.1 is the replacement of arithmetic coding [PAS76] by Huffman coding [HUF52], which is much faster because each symbol is directly encoded with a binary representation of that symbol using a previously precomputed table (see Chapter 1 for details). So it is almost as fast as "raw coding".

This fast modified algorithm consists of three stages. In the first one, all the symbols needed to efficiently represent the transform image are calculated. This stage is equivalent to Algorithm 6.1(b). During this stage, statistics are collected in order to compute a Huffman table in a second stage. Finally, the last stage consists in encoding the symbols computed during the first one by using Huffman coding, and it is equivalent to Algorithm 6.1(c).

During the first stage, all the symbols must be computed (this time including the "numeric symbols": $nbits_{i,j}$ and $nbits_{i,j}^{LOWER}$) because the probability model is now built during this stage and not while coding in the last stage. As an optimization in this first pass, in order to increase the appearance of 2×2 blocks of *LOWER_COMPONENT*, whenever the four coefficients have insignificant descendants, the threshold to compare these four coefficients to assign them a *LOWER_COMPONENT* label is increased from $2^{rplanes}$ to $2^{rplanes+1}$, so as to extend an existing lower-tree more easily (see details in Algorithm 6.1(b), lines 4 and 12).

In the second stage, Huffman codes are built with the probability model from the source (i.e., the symbols computed in the first stage), once this probability model has been acquired. A different Huffman code set is computed for each subband level, since statistics vary from one level to another. The computed tables containing the Huffman codes are output so that the decoder can use them to decode the symbols. The *rplane* parameter is also output in this stage, which actually replaces the initialization stage of Algorithm 6.1(a).

Finally, in the third stage, the symbols computed in the first stage are Huffman encoded using the codes computed in the second stage. Recall that no *LOWER_COMPONENT* is encoded, and that significant bits and sign are needed, and therefore binary encoded, for each significant coefficient.

Observe that since no adaptive coding and context-modeling is performed, the order in

which coefficient blocks are scanned in each subband does not affect compression efficiency, and therefore, a typical raster scan order is followed because it avoids cache misses, being faster. In the original LTW, a scan in clusters was used in order to take advantage of spatial locality with an adaptive arithmetic encoder with two-contexts, increasing the PSNR but being a bit slower.

Contrary to the original RLW and LTW algorithms, which mainly group coefficients to reduce complexity (most of all in the run-length Algorithm 5.2 compared with the simpler Algorithm 5.1) without a significant increase in coding efficiency, in this new fast LTW, the tree structure used to efficiently group coefficients becomes essential to achieve good compression efficiency. It is easy to see the importance of grouping coefficients in this algorithm if we recall that the length of the shortest Huffman code is one bit. Therefore, if a symbol is encoded for each coefficient (as in Algorithm 5.1), the minimum achievable target bitrate is 1 bpp, and we still have to add the significant bits and sign, and consider that many encoded symbols will be longer than one bit.

Fortunately, LTW works quite well with Huffman coding (as we will see later in the coding results), because there are many different symbols (in particular, a lower symbol, an isolated symbol, and all the "regular numeric symbols" and "*LOWER* numeric symbols"), but the probability is highly concentrated in a few symbols, which is the best condition for Huffman coding.

7.2.2 Efficient Huffman decoding

Although Huffman encoding is very simple, because it only requires looking up in a table, that is not the case of the usual Huffman decoding, which is performed bit by bit, following a binary tree as each bit from the bitstream is input. A faster alternative is to use a lookup table of length 2^L, being L the length of the largest generated Huffman code. In this table, each entry in a position P stores a symbol. This symbol corresponds to the Huffman code C that is equal to the N most significant bits of the binary representation of P. This correspondence is unique for each entry if full codes are considered, since two different symbols never have the same code (although they can be partially the same, in the most significant bits). The length of the Huffman code (N) is also stored in that entry. This table can be built very easily as follows: for each symbol S with a Huffman code C of length N, we fill this table with (S, N) in the positions ranging from $C\times2^{L-N}$ to $(C+1)\times2^{L-N}-1$ (i.e., in the positions with C in the N

most significant bits, and ranging from "all 0s" to "all 1s" in the $L–N$ least significant bits). This way, we use L bits of the bitstream to look up an entry in the table at the position indicated by these L bits, which gives us the corresponding symbol S and the number of bits N that the Huffman code actually requires. Hence, for the next symbol, N bits are input from the bitstream, which are completed with $L–N$ bits that were already read but did not belong to the previous Huffman code.

The main drawback of this Huffman decoding technique is that it does not scale very well because growth is exponential as the size of L increases (since it requires 2^L entries). Therefore, we use a combination of both methods. First, we use a small table to decode a symbol. For short Huffman codes, the symbol can be successfully decoded, however, for long codes, the complete symbol will not be decoded, but a binary tree will be used to finish the decoding operation following the usual Huffman decoding process. Note that the increase in complexity is very low because short codes are much more likely than long codes, and thus, most times, the decoding operation will be solved with the lookup table.

codec\ bitrate	SBHP	PROGRES	LTW Huffman	LTW Orig.	JPEG 2000
Lena (512×512)					
0.125	n/a	30.59	31.06	31.27	30.84
0.25	n/a	33.71	34.03	34.31	34.04
0.5	n/a	36.85	37.03	37.35	37.22
1	n/a	39.89	40.11	40.50	40.31
Café (2560×2048)					
0.125	20.49	n/a	20.56	20.76	20.74
0.25	22.64	n/a	22.90	23.24	23.12
0.5	26.01	n/a	26.31	26.85	26.80
1	31.08	n/a	31.30	32.03	32.04
Woman (2560×2048)					
0.125	27.09	26.89	27.23	27.52	27.33
0.25	29.59	29.40	29.70	30.16	29.98
0.5	33.11	33.02	33.15	33.82	33.63
1	37.98	37.75	37.76	38.53	38.43

Table 7.4: PSNR (dB) with different bitrates and very fast encoders.

7.2.3 Numerical results

In Table 7.4, we compare the compression efficiency of the new proposed variation of LTW

with the original LTW, JPEG 2000 and other fast wavelet encoders described in Chapter 4 that also avoid bit-plane coding and iterative methods, namely PROGRES [CHO05] and (*non-embedded*) SBHP [CHY00]. Obviously, the original LTW is more efficient than the Huffman variation (from to 0.2 to 0.7 dB in PSNR, depending on the bit-rate and image), mainly due to the use of adaptivity, context modeling and arithmetic coding. When compared with JPEG 2000, PSNR is similar at low bitrates, and about 0.6 lower at high bitrates. However, our new proposal is more efficient than PROGRES (up to 0.5 dB at low bitrates), and than SBHP in slightly detailed images, like Café, although in low-frequency images, like Woman, SBHP works slightly better, ranging from 0.05 to 0.2 dB.

However, the main advantage of this new proposal is its very low execution time. In table 7.5, execution time is compared with the original LTW and PROGRES[1]. For the coding process, the original LTW is faster than PROGRES (up to 3 times) except in high bit-rates. If we compare the new fast LTW proposal with PROGRES, this advantage is increased, being from 4 to 9 times faster, depending on the bit-rate. The decoding process is much faster than the coding process in PROGRES because the decoder does not need to compute the highest coefficient in each sub-tree, and consequently only an improvement of about 20% is achieved with the proposed LTW at 1 bpp, while this improvement becomes smaller as the bitrate is reduced. Although there are no execution time results (or reference software) available to compare our proposal with SBHP, the use of bitplane coding to compute the significance map and a sorting algorithm in SBHP probably make it slower. In fact, our encoder, when coding Woman at a range from 1 bpp to 0.125 bpp, is from 18 to 38 times faster than JPEG 2000 reference software (in particular, Jasper, written in C), while in [CHY00] authors state that SBHP coding was 4 times faster than JPEG 2000 VM.

As in the rest of chapters, for more tests, our implementations are available at http://www.disca.upv.es/joliver/thesis.

[1] For the execution time comparison, similar processors have been employed. Results for PROGRES were published in [CHO05] with an Intel Xeon 2 Ghz Processor, and results for LTW in this section are obtained with an Intel PentiumM 1.6 Ghz Processor. On the other hand, all the implementations are written in C language and compiled with Visual C++ 6.0 and the same speed optimization level

codec\ bitrate	PRO-GRESS	LTW Huffman	LTW Orig.	PRO-GRESS	LTW Huffman	LTW Orig.
	CODING			DECODING		
Lena (512×512)						
0.125	23.7	2.7	8.2	1.6	1.6	4.8
0.25	26.1	3.5	12.1	2.6	2.4	8.6
0.5	29.0	5.0	19.7	4.6	3.9	15.8
1	34.8	8.1	36.4	8.3	6.7	30.8
Woman (2048×2048)						
0.125	378.4	51.3	149.5	24.1	26.1	83.4
0.25	404.3	68.8	217.2	41.9	40.5	147.3
0.5	450.1	100.2	337.3	74.7	63.2	266.6
1	528.4	140.0	568.7	128.4	101.5	484.2

Table 7.5: Execution time comparison of the coding process for very fast encoders (excluding DWT) (time in million of CPU cycles).

7.3 Lossless coding

In Chapter 1, we discussed the importance of lossless coding in some applications (such as medical imaging and image editing). For this reason, it is important for an encoder to be able to provide lossless compression with little or no modification of the usual algorithm, so that an implementation of that algorithm can work in lossy or lossless mode, depending on the specific application, simply by varying the input parameters. The algorithms presented in this thesis possess this feature if no quantization is applied and an integer-to-integer wavelet transform is used. In order to skip the quantization process, the parameters of the Appendix A can be set as *rplanes*=0, Q=1/2 and K=0, although it is faster if we simply omit all the operations related to the scalar quantization. For the wavelet transform, we will use the reversible bi-orthogonal 5/3 filter bank for integer implementation, which is described in Chapter 2.

In Table 7.6, we compare the results of losslessly encode six images (grayscale 8 bpp) with our encoder, JPEG 2000 and the LOCO-I algorithm (in which the JPEG-LS standard is based). In JPEG 2000, the same bi-orthogonal 5/3 transform is used. In this table, results are expressed as the number of bits per pixel needed for the compressed image, and in general it is reduced from 8 bpp (in the original image) to 4-5 bpp after lossless coding. LTW and JPEG 2000 are general purpose and perform almost the same in all the images, with no more than 0.05 bpp difference between them (about 1% in performance). This is a good result for

our encoder, if we take into account that lossless coding is mainly based in predictive techniques and context modeling, and LTW, contrary to JPEG 2000, only handles two contexts. LOCO-I [WEI00] is a specific prediction-based lossless technique. However, it is not much more efficient than the other two encoders under evaluation, requiring about 0.1-0.2 bpp less than JPEG 2000 and LTW (LOCO-I's coding efficiency is not higher than 5% compared with JPEG 2000 and LTW).

codec \ image	LOCO-I	JPEG 2000	LTW
Lena (512×512)	4.24	4.31	4.26
Barbara (512×512)	4.86	4.78	4.83
Goldhill (512×512)	4.71	4.84	4.78
Woman (2560×2048)	4.45	4.51	4.50
Café (2560×2048)	5.09	5.35	5.36
Bike (2560×2048)	4.36	4.53	4.56

Table 7.6: Lossless coding comparison of various image encoders with six greyscale 8 bpp images. Results are given in bits per pixel (bpp) needed to losslessly encode the original image.

7.4 Summary

Some variations of the algorithms presented in chapters 5 and 6 have been proposed in this chapter, in order to obtain versions with very low memory consumption and/or with very fast coding of the wavelet coefficients.

The versions with reduced memory usage use the wavelet transform of Chapter 3, and, with almost the same coding efficiency as the original algorithms, are faster and require 20 times less memory usage when considering process memory (data memory usage is actually much lower).

The very fast version of the lower-tree wavelet encoder uses Huffman coding and other strategies to reduce execution time. The loss of coding efficiency is compensated by the reduction in execution time. In fact, the encoder is less complex than some of the fastest wavelet encoders reported in the literature, being up to 9 times faster than PROGRES (and much more symmetric than it), while PSNR is from 0.3 to 0.5 dB higher at low bit-rates.

Finally, results for the methods proposed in this thesis applied to lossless image

compression are shown, revealing that its coding efficiency in lossless mode is not far from specific lossless encoders like LOCO-I.

Chapter 8

Conclusions and future work

8.1 Contributions of this thesis

Although a detailed summary section with the main contributions and conclusions is presented at the end of each chapter, it is interesting to summarize some of the main contributions introduced in this thesis.

In Chapter 3, we propose a general recursive algorithm to compute the DWT in a line-based fashion. We give two implementations of this recursive algorithm, first by means of a simple filter-bank and later using the lifting scheme to improve its efficiency. Both convolution and lifting algorithms are fully described and can be straightforwardly implemented. In fact, we give an implementation in ANSI C language, which differs little from the pseudo-code description. While in an iterative approach (like that proposed in [CHR00]) we need to handle several buffers with different delay and rhythm, which is a difficult task in a software-based implementation, the main contribution of our proposal with respect to other proposals is the solution of the problem of synchronous communication among buffers by means of a recursive function. At the same time, we introduce a line-based DWT based on the lifting scheme that reduces the memory requirements by half and improves the overall execution time, and an integer-to-integer implementation that allows a reversible wavelet transform. Furthermore, other previous proposals are given with an implicit assumption of a hardware design implementation, whereas our proposal is valid for

software-based ones.

In Chapters 5 and 6, we introduce various non-embedded wavelet image encoders that avoid bit-plane coding and do not use iterative optimization algorithms or high-order context modeling in order to reduce complexity. Although these encoders are not SNR scalable, they support resolution scalability because coefficients are encoded at once, in decreasing order of subband level. With these coders, we introduce a discussion about the convenience of complex coding techniques to achieve features that are not always necessary. With this in mind, we present a run-length (RLW) coder and a tree-based (LTW) wavelet coder. In addition, LTW reveals that tree-based coding can be used as a very fast method of grouping coefficients, and not only as an efficient way to encode them. Based on these ideas, other tree-based proposals emerged later, like for example PROGRES [CHO05].

Finally, we show that a line-based wavelet transform can be efficiently applied along with a tree-based scheme, significantly reducing the memory requirements.

8.2 Conclusions

In this thesis, we have proposed several algorithms to reduce complexity and memory usage for wavelet coding. Although great efforts have been made to improve compression efficiency and allow additional features, there are many open opportunities in reducing complexity, as stated by H. Malvar, a director of Microsoft Research, in a plenary talk at the 24th Picture Coding Symposium [MAL04], referring to the new lines of research to be explored in the field of image compression after the release of the JPEG 2000 standard. With this in mind, the coding scheme proposed in Chapter 7, which stems from the joint application of the tree-based image coder of Chapter 6 and the recursive wavelet transform with reduced memory usage of Chapter 3, is able to encode an image with state-of-the-art coding performance, while complexity is reduced a lot if compared with the encoders commonly used as benchmarks, like SPIHT and the JPEG 2000 standard. In fact, execution time is reduced several times, in particular more than three times compared with SPIHT and more than 25 compared with the JPEG 2000 reference software (all of them written in C language and compiled under the same conditions), while memory requirements are reduced 22 times with respect to SPIHT and up to 50 times compared with the JPEG 2000 reference software.

Although complexity reduction may seem of limited interest in high-performance

workstations with plenty of memory, some particular applications, like image editing for large images and especially GIS applications, cannot be easily tackled with the complexity of previous encoders. In addition, many other common applications, like simple slide show, become annoying if the decoding delay is too high.

On the other hand, the memory requirements of these encoders may seriously affect memory-constrained devices dealing with digital images, such as mobile phones, digital cameras and personal digital assistants (PDA). The complexity of these encoders is another issue that affects these devices, since they usually contain DSPs or processors with lower computational power than regular desktop workstation processors. Both memory requirement and complexity impose severe restrictions on coding applications running on this type of devices, in terms of required working memory and processing time.

From a hardware design perspective, complexity reduction can be thought of as a cost reduction when implementing a particular application. Thus, in order to implement a time-consuming and memory-intensive coding algorithm, an expensive DSP is needed (for example, a 1000 MHz TMS320C6455 DSP with 2048 KB L2 cache, which is currently valued at approximately 350 $US). If we reduce complexity and improve cache utilization for the same type of application, a much cheaper DSP can run the new algorithm (for example a 400 MHz TMS320DM640AGDK4 DSP with 160 KB L2 cache that costs less than 25 $US).

In conclusion, we think that the encoders presented in this thesis are good candidates for real-time interactive multimedia communications and other applications, allowing simple implementation both in hardware and in software.

8.3 Future lines of research

There are several open problems and more work to be done related with the subject of this thesis. Future work includes:

- To develop parallel versions of the wavelet transform and the image encoders proposed in this thesis to ease GIS processing.
- A study of hardware designs for the algorithms presented in the thesis, since we have shown that they are suitable for efficient image compression implemented in hardware with low-cost systems.
- Non-iterative rate control methods for non-embedded image coding can be further analyzed. These methods are expected to be useful for other non-embedded

encoders apart from RLW and LTW, such as PROGRES and non-embedded SPITH and SPECK/SBHP.

- More investigation is needed to employ the proposed wavelet image coders in error-prone environments and to add more robustness to them for wireless transmission.
- The use of the wavelet transform in video compression is still a topic of interest, because while some proposals have been made, most of them are still below the efficiency of DCT-based encoders like H.264. In the next section, we briefly describe how video compression can be implemented with the techniques proposed in this thesis.

8.3.1 A future application: extension to video coding

Following the classical schemes of wavelet-based video coding, many of the techniques and algorithms applied in this thesis can be extended to video, ether by using a motion-compensation scheme, by considering time as a third dimension (with time filtering), or even by combining both ideas.

8.3.1.1 Motion Compensation (MC)

In motion compensated video coding (like [BLA98], [MAR99] and [WIE00]), a frame is predicted from those frames previously encoded in the video sequence. In these encoders, after the motion compensation algorithm is applied, the prediction residue is then encoded with a two-dimensional wavelet-based algorithm, along with the motion vectors. Actually, this framework is very similar to the well-known family of standards MPEG, but in order to decorrelate the energy of the residue, a wavelet transform is employed instead of the DCT. Within this scheme, any of the image encoders presented in this thesis can be easily applied to encode the residue, once a motion compensation algorithm has reduced the redundancy between frames. Clearly, the motion estimation algorithm will be the computational bottleneck in this video encoder, and therefore more research is needed to ease this part. In addition, the expected coding efficiency is not very high, based on similar experiments previously published ([BLA98] [MAR99] [WEI00]).

8.3.1.2 Temporal Filtering (TF)

Another scheme typically used for video coding is the three-dimensional (3D) wavelet video coding, in which the wavelet transform is applied in the three directions, i.e., in the spatial

directions (horizontal and vertical) and in the time direction (which is known as temporal filtering). Afterwards, the resulting wavelet coefficients are entropy encoded. The first problem that arises in this case is the extremely high memory consumption of the wavelet transform if the regular algorithm is used, since a group of frames must be kept in memory before applying temporal filtering, and in general, the greater temporal decorrelation, the more number of frames are needed in memory.

Some efforts have already been done to reduce memory requirements and the execution time of the 3D DWT [MOY01] [BER02]. As an alternative to them, the ideas introduced in Chapter 3 can be easily adapted to compute the wavelet transform in the three dimensions with very low memory usage. Therefore, Algorithm 3.1 can be modified to achieve a frame-based 3D wavelet transform simply by replacing the word "line" by "frame" in the algorithm description. In this new version, at every level, each buffer must be able to keep either $2N+1$ or $W+2$ low frequency frames, depending on whether it is implemented as a filter-bank or with the lifting scheme (see Chapter 3 for details). As presented in Figure 8.1, which shows the new proposal compared with the classical one, each buffer at a level i needs a quarter of coefficients if compared with the previous level ($i-1$). Therefore, for a frame size of ($w \times h$) and an *nlevel* time decomposition, the number of coefficients required by the version implemented with a filter-bank is

$$(2N+1)\times(w\times h)+(2N+1)\times(w\times h)/4+\cdots+(2N+1)\times(w\times h)/4^{nlevel-1} \qquad (8.1)$$

which is asymptotically (as *nlevel* approaches infinity)

$$\sum_{n=0}^{\infty}\frac{(2N+1)\times(w\times h)}{4^n}=(2N+1)\times(w\times h)\times\frac{4}{3} \qquad (8.2)$$

independently of the number of frames to be encoded, much lower than the regular case.

Another drawback of the regular 3D DWT and the need to split the video sequence into group of frames is that it causes discontinuities on the first and last frames, displaying blocking artifacts in the time domain.

In addition, an important difference between both proposals is how video can be decoded from the middle of the bitstream, that is, when the user begins to receive the video broadcast while it is already in progress. In the regular algorithm, the current group of frames being received is ignored, and then, the following group is stored in memory. After it has been entirely received, it can be decoded, and the 3D IWT can be applied. On the other hand, for the inverse transform in the frame-based scheme, the decoding process begins immediately

by filling up the highest-level buffer (*nlevel*) with the information received from the bitstream. During this process, other information from the bitstream is ignored. Afterwards, once this buffer is full, we begin to accept also information from the previous level (*nlevel*–1), and so forth, until all the buffers are full. At that moment, the video can be sequentially decoded as usual. The latency of this process is determinist and depends on the filter length and the number of decomposition levels (the higher they are, the higher latency). However, for the regular 3D algorithm, the latency depends on the remaining number of frames in the current group when the process begins, and the size of the group of pictures.

An important issue to consider in this proposal is the need to reverse the order of the coefficients in the decoder, from the highest level to lowest one. A simple solution is to hold a different bitstream for each decomposition level. However, more complex and suitable investigations should be done, trying to reverse the order of the coefficients as they are computed in the encoder and read by the decoder, as suggested in [CHR00].

Finally, the transform coefficients can be entropy encoded by using run-length coding techniques or 3D lower trees in a similar way as in Chapter 7. In this case, to form 3D lower trees, each coefficient (except the leaves) has eight descendant coefficients instead of four, as proposed in [KIM00]. The expected complexity of this encoder is lower than the motion compensated one, but the coding efficiency will be low in moderate-motion sequences due to the appearance of misaligned objects in the time direction, causing an energy increase in high-frequency subbands, and thus preventing the lower-tree formation.

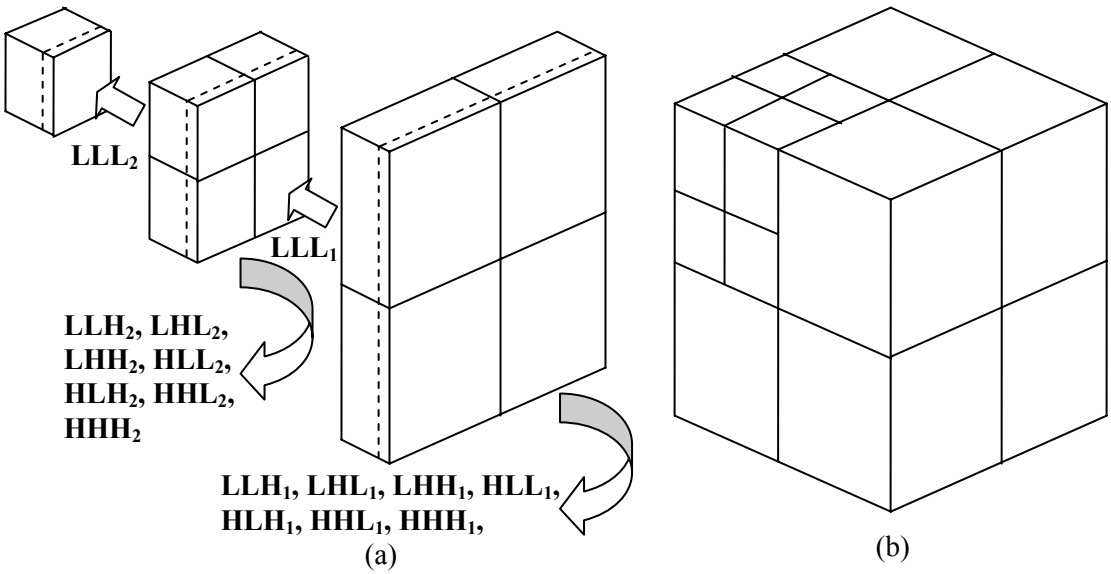


Fig. 8.1: Overview of the 3D DWT computation in a two-level decomposition, (a) following a frame-based scheme as an evolution of Algorithm 3.1 or (b) the regular 3D DWT algorithm.

A different application of the 3D DWT is to encode 3D surfaces. In [AVI05], the authors propose the use of the LTW algorithm to encode 3D models in a scalable fashion (with various LODs, Levels Of Detail). Numerical results show that a LTW-based encoder achieves higher coding efficiency than a SPIHT-based encoder.

8.3.1.3 Motion Compensated Temporal Filtering (MCTF)

So far, the best results in wavelet-based video coding are achieved with a combination of the two previous methods. In these techniques, in order to compensate the object (or pixel) misalignment between frames, and hence to avoid the significant amount of energy that appears in high-frequency subbands, a motion compensation algorithm is introduced to align all the objects (or pixels) in the frames before being temporal filtered. This approach is called Motion Compensated Temporal Filtering (MCTF) and currently, there is considerable research activity focused on it [SEC01] [SON05] [CAG05]. Most of them use very simple lifting transforms for the temporal filtering, such as the B5/3 transform described in equations (2.43) and (2.44) or even a simplified version of this transform that skips the update step (equation (2.44)). The fast image encoders proposed in this thesis can also be extended to video and then used in the MCTF framework to compensate the high complexity of the motion estimation. In addition, as a future work, the 3D DWT described in the previous section could be combined with a motion compensation scheme, so that more complex wavelet transforms can be easily applied. A future research should establish if the introduction of longer filter-banks is really beneficial in MCTF.

8.4 Publications resulting from this thesis

Some fragments of the work presented in this thesis have been published in proceeding of international conferences. In particular, the main contributions are:

From Chapter 3:

- J. Oliver, M.P. Malumbres, "A fast wavelet transform for image coding with low memory consumption," in proceedings of the 24th IEEE Picture Coding Symposium, California (USA), 2004. A first version of the recursive wavelet transform with low memory usage is presented using a simple filter-bank.
- J. Oliver, E. Oliver, M.P. Malumbres, "On the efficient memory usage in the lifting scheme for the two-dimensional wavelet transform computation," in

proceedings of the IEEE International Conference on Image Processing, Genoa (Italy), 2005. The recursive transform of Chapter 3 is generalized and it is implemented with the lifting scheme.

- J. Oliver, E. Oliver, M.P. Malumbres, "Fast integer-to-integer reversible lifting transform with reduced memory consumption," in proceedings of the IEEE International Symposium on Signal Processing and Information Technology, Athens (Greece), 2005. An integer-to-integer reversible version of the efficient wavelet transform of Chapter 3 is described.

From Chapter 4:

- J. Oliver, M.P. Malumbres, "Tuning and optimizing the performance of the EZW algorithm," in proceedings of the International Conference on Image and Signal Processing, Agadir (Morocco), 2001. An implementation of the EZW algorithm is presented and tuned, analyzing many coding parameters.

From Chapter 5:

- J. Oliver, M.P. Malumbres, "A simple picture coding algorithm with fast run-length mode," in proceedings of the 23th IEEE Picture Coding Symposium, Saint Malo (France), 2003. The simple and run-length encoders of Chapter 5 are proposed.

From Chapter 6:

- J. Oliver, M.P. Malumbres, "A new fast lower-tree wavelet image encoder," in proceedings of the IEEE International Conference on Image Processing, Thessaloniki (Greece), 2001. An early version of the tree-based LTW encoder is introduced. This version only makes use of degree-0 trees, and it is compared only with EZW.
- J. Oliver, M.P. Malumbres, "Design options on the development of a new tree-based wavelet image encoder," in Lecture Notes in Computer Science, proceedings of the 8th International Workshop, Very Low Bit-Rate Video, Madrid (Spain), 2003. An analysis of the coding parameters of LTW is made to improve compression efficiency.
- J. Oliver, M.P. Malumbres, "Fast and efficient spatial scalable image compression using wavelet lower trees," in proceedings of the IEEE Data Compression Conference, Snowbird, Utah (USA), 2003. A complete version

of the LTW algorithm, with degree-1 trees among other optimizations, is presented.

From Chapter 7:

- J. Oliver, M.P. Malumbres, "A fast run-length algorithm for wavelet image coding with reduced memory usage," in Lecture Notes in Computer Science, proceedings of the 2nd Iberian Conference on Pattern Recognition and Image Analysis, 2005. The memory requirements of the run-length encoder of Chapter 5 are greatly reduced with the use of the recursive wavelet transform of Chapter 3.
- J. Oliver, M.P. Malumbres, "Fast-tree based wavelet image coding with efficient use of memory," in proceedings of the SPIE/IEEE Visual Communications and Image Processing, Beijing (China), 2005. In this paper, the tree-based encoder of Chapter 6 is now combined with the wavelet transform of Chapter 3, and the problem of tree construction for wavelet coding with this DWT is tackled.
- J. Oliver, M.P. Malumbres, "Huffman Coding of Wavelet Lower Trees for Very Fast Image Compression," in proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Toulouse (France), 2006. A very fast version of the tree-based encoder of Chapter 6 is presented, reducing the execution time several times with moderate loss of coding efficiency. This encoder is compared with other recent very fast wavelet encoders such as PROGRES.

Bibliography

[ACH05] T. Acharya, P. Tsai, *JPEG 2000 Standard for Image Compression: Concepts, Algorithms and VLSI Arquitectures*, Chapter 5, Wiley, October 2005.

[ADA00] M. Adams, F. Kossentini, *Reversible Integer-to-Integer Wavelet Transforms for Image Compression: Performance Evaluation and Analysis*, IEEE Transactions on Image Processing, vol. 9, pp. 1010-1024, June 2000.

[ADA00] M. D. Adams, F. Kossentini, *Reversible Integer-to-Integer Wavelet Transforms for Image Compression: Performance Evaluation and Analysis*, IEEE Transactions on Image Processing, vol. 9, no 6, pp. 1010-1024, June 2000.

[ADA02] M. Adams, Jasper Software Reference Manual (Version 1.600.0), ISO/IEC JTC 1/SC 29/WG 1 N 2415, Oct. 2002.

[ALB95] M. Albanesi, S. Bertoluzza, *Human vision model and wavelets for high-quality image compression*, International Conference in Image Processing and its Applications, July 1995.

[ALG95] V. R. Algazi, R. R. Estes, *Analysis-based coding of image transform and subband coefficients*, SPIE Applications of Digital Image Processing XVII, vol. 2564, pp. 11-21, August 1995.

[ANT92] M. Antonini, M. Barlaud, P. Mathieu, I. Daubechies, *Imagen Coding Using Wavelet Transform*, IEEE Transactions on Image Processing, vol. 1, no2, April 1992.

[AVI05] M. Avilés, F. Morán, N. García, *Progressive lower trees of wavelet coefficients: efficient spatial and SNR scalable coding of 3D models*, Pacifica Rim Conference on Multimedia, November 2005.

[BAR93] M. Barnsley, L. Hurd. *Fractal Image Compression*, AK Peters, Wellesley, 1993.

[BEL90] T. C. Bell, J. G. Cleary, I. H. Witten, *Text Compression*, Advanced Reference Series, Englewood Cliffs, Prentice Hall, 1990.

[BER01] W. Berghorn, T. Boskamp, M. Lang, H. Peitgen, *Fast variable run-length coding for embedded progressive wavelet-based image compression*, IEEE Transactions on Image Processing, vol. 10, pp. 1781-1790, December 2001.

[BER02] G. Bernabé, J. González, J.M. García, J. Duato, *Memory conscious 3D wavelet transform*, EUROMICRO Conference, September 2002.

[BLA98] D. Blasiak, W.Y. Chan, *Efficient wavelet coding of motion compensated prediction residuals*, IEEE International Conference on Image Processing, October 1998.

[BRI95] C. M. Brislawn, *Preservation of Subband Symmetry in Multirate Signal Coding*, IEEE Transactions on Signal Processing, vol. 43, no 12, pp. 3046-3050, December 1995.

[CAG05] M. Cagnazzo, *Wavelet transform and three/dimensional data compression*, Ph.D. thesis, Università degli studi di Napoli / Université de Nice-Sophia Antipolis, 2005.

[CAL98] R. C. Calderbank, I. Daubechies, W. Sweldens, B. L. Yeo, *Wavelet transforms that map integers to integer*, Journal of Applied Computational and Harmonic Analysis, vol. 5, pp. 332-369, 1998.

[CAP58] J. Capon, *A probabilistic model for Run-Length coding of pictures*, IRE Transactions on Information Theory, 157-163, 1958.

[CHA01] W. Chang, Y. Lee, W. Peng, C. Lee, *A Line-Based, Memory Efficient and Programmable Architecture for 2D DWT using Lifting Scheme*, International Symposium on Circuits and Systems (ISCAS), 2001.

[CHO05] Yushin Cho, W. A. Pearlman, A. Said, *Low complexity resolution progressive image coding algorithm: PROGRES (Progressive Resolution Decompression)*, IEEE International Conference on Image Processing, September 2005.

[CHO05b] Yushin Cho, W. A. Pearlman, *Quantifying the coding power of zerotrees of wavelet coefficients: a degree-k zerotree model*, IEEE International Conference on Image Processing, September 2005.

[CHR00] C. Chrysafis, A. Ortega, *Line-based, reduced memory, wavelet image compression*, IEEE Transactions on Image Processing, vol. 9, pp. 378-389, March 2000.

[CHY00] C. Chrysafis, A. Said, A. Drukarev, A. Islam, W. A. Pearlman, *SBHP- A low complexity wavelet coder*, IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 2035-2038, 2000.

[CLE84] J. G. Cleary, I. H. Witten, *Data Compression Using Adaptive Coding and Partial String Matching*, IEEE Transactions on Communications, Vol. 32, 396-402, 1984.

[COH92] A. Cohen, I. Daubechies, J.C. Feauveau, *Biorthogonal Bases of Compactly Supported Wavelets*, Communications on Pure and Applied Mathematics, vol.45, no5, pp 485-560, June 1992.

[COI92] R. R. Coifman, M. V. Wickerhauser, *Entropy-based algorithms for best basis selection*, IEEE Transactions on Information Theory, vol. 38, pp. 713-718, March 1992.

[COS98] P. Cosman, K. Zeger, *Memory constrained wavelet-based image coding*, Proc. UCSD Conf. Wireless Communications, March 1998.

[COV91] T. M. Cover, J. A. Thomas, *Elements of information theory*, Wiley Series in Communications, 1991.

[CUT52] C. C. Cutler, *Differential Quantization for Television Signals*, U. S. Patent 2,605,361, July 29 1954.

[DAS96] E. A. B. Da Silva, D. G. Sampson, M. Ghanbari, *A successive approximation vector quantizer for wavelet transform image coding*, IEEE Transactions on Image Processing, vol. 5, pp. 299-310, February 1996.

[DAU92] I. Daubechies, *Ten Lectures on Wavelets*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1992.

[DAU98] I. Daubechies, *Orthonormal Bases of Compactly Supported Wavelets*, Communications on Pure and Applied Mathematics, vol. 41, pp 909-996, November 1998.

[DAU98b] I. Daubechies, W. Sweldens, *Factoring wavelet transforms into lifting steps*, Journal of Fourier Analysis, no 3, 1998.

[DIL03] G. Dillen, B. Georis, J. Legat, O. Cantineau, *Combined Line-Based Architecture for the 5-3 and 9-7 Wavelet Transform of JPEG 2000*, IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, pp. 944-950, September 2003.

[EVE63] H. Everett, *Generalized Lagrange multiplier method for solving problems of optimum allocation of resources*, Operations Research, vol. 11, pp. 399-417, 1963.

[FAL73] N. Faller, *An adaptive system for data compression*, Record of the 7th Asilomar Conference on Circuits, Systems and Computers, 593-597, Piscataway, IEEE Press, 1973.

[FAN61] R. M. Fano, *Transmission of Information, a statistical theory of communication*, Cambridge MA, The MIT Press, 1961.

[GAB46] D. Gabor, *Theory of communication*, Journal of the IEE, pp. 429-457, 1946.

[GAL78] R. G. Gallagher, *Variations on a theme by Huffman*, IEEE Transactions on Information Theory, Vol. 6, 668-674, November 1978.

[GER92] A. Gersho, R.M. Gray, *Vector Quantisation and Signal Compression,* Kluwer Academic Publishers, 1992.

[GHA03] M. Ghanbari, *Standard Codecs: Image Compression to Advanced Video Coding*, Institution of Electrical Engieneers (IEE) Telecommunications series, 43-53, 2003.

BIBLIOGRAPHY

[HIL94] M.L. Hilton, B.D. Jawerth, A. Sengupta, *Compressing still and moving images with wavelets*, Multimedia Systems, vol. 2, 1994.

[HON02] E. S. Hong, R. E Ladner, *Group testing for image compression*, IEEE Transactions on Image Processing, vol. 11, pp. 901-911, August 2002.

[HSI00] S.T. Hsiang, J. W. Woods, *Embedded image coding using zeroblocks of subband/wavelet coefficients and context modelling*, IEEE International Conference on Circuits and Systems, pp. 662-665, 2000.

[HUF52] D. A. Huffman, *A method for the construction of minimum-redundancy codes*, Proceedings of the Institute of Electrical and Radio Engineers, Vol. 40, 1098-1101, September 1952.

[ISL99] A. Islam, W. A. Pearlman, *An embedded and efficient low-complexity hierarchical image coder*, SPIE Visual Communications and Image Processing (VCIP conference), pp. 294-305, 1999.

[ISO00] ISO/IEC 15444-1, *JPEG2000 image coding system*, 2000.

[ISO92] ISO/IEC 10918-1/ITU-T Recommendation T.81, *Digital Compression and Coding of Continuous-Tone Still Image*, 1992.

[ITU00] Recommendation ITU-T BT.500-10, *Methodology for the Subjective Assessment of the Quality of Television Pictures*, ITU-T 2000.

[ITU82] ITU-T Recommendation 602, *Encoding parameters of digital television for studios*, 1982.

[JAY70] N. S. Jayant, *Adaptive Delta Modulation with One-Bit Memory*, Bell Systems Technical Journal, Vol. 49, 321-342, March 1970.

[JOS95] R. L. Joshi, V. J. Crump, T. R. Fischer, *Image subband coding using arithmetic coded trellis coded quantization*, IEEE Transactions on Circuits and Systems for Video technology, vol. 5, December 1995.

[KIM00] B. J. Kim, Z. Xiong, W. A. Pearlman, *Low bit-rate scalable video coding with 3d set partitioning in hierarchical trees (3d spiht)*, IEEE Transactions on Circuits and Systems for Video technology, vol. 10, pp. 1374-1387, December 1995.

[KIM00] B.J. Kim, Z. Xiong, W.A. Pearlman, *Low bit-rate scalable video coding with 3D set partitioning in hierarchical trees (3D SPIHT)*, IEEE Transactions on Circuits and Systems for Video Technology, vol. 10, pp. 1374-1387, December 2000.

[LEG88] D. Le Gall, A. Tabatabai, *Subband Coding of Digital Images Using Sysmmetric Kernel Filters and Arithmetic Coding Techniques*, International Conference on Acoustics, Speech Signal Processing, New York, USA, pp. 761-764, April 1988.

[LEL87] D. A. Lelewer, D. S. Hirschberg, *Data Compression*, ACM Computing Surveys, Vol. 9, 261-296, September, 1987.

[LEN04] K. Lengwehasatit, A. Ortega, *Scalable variable complexity approximate forward DCT*, IEEE Transactions on Circuits and Systems for Video Technology, vol. 14, pp. 1236-1248, November 2004.

[LIM90] J. S. Lim, *Two-Dimensional Signal and Image Processing*, Chapter 4.1, Prentice-Hall, 1990.

[MAL03] H. S. Malvar, G. J. Sullivan, *YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range*, Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, Document No. JVT-I014r3, July 2003.

[MAL04] H. Malvar, *Is there life after JPEG2000 and H.264?*, Plenary talk, Picture Coding Symposium, San Francisco, December 2004.

[MAL89] S. Mallat, *A Theory for Multiresolution Signal Decomposition*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 11, pp. 674-693, July 1989.

[MAL92] S. Mallat, S. Zhong, *Characterization of signals from multiscale edges*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 14, pp. 710-732, July 1992.

[MAR02] M. W. Marcellin, M. A. Lepley, A. Bilgin, T. J. Flohr, T. T. Chinen, J. H. Kasner, *An overview of quantization in JPEG2000*, Signal Processing: Image Communication, vol. 17, 2002.

[MAR99] D. Marpe, H. Cycon, *Very low bit-rate video coding using wavelet-based techniques*, IEEE Transactions on Circuits and Systems for Video Technology, vol. 9, pp. 85-94, February 1999.

[MAR99] D. Marpe, H. L. Cycon, *Very low bit-rate video coding using wavelet-based techniques*, IEEE Transactions on Circuits and Systems for Video technology, vol. 9, pp. 85-94, February 1999.

[MEY00] F. G. Meyer, A. Z. Averbuch, J. O. Strömberg, *Fast adaptive wavelet packet image compression*, IEEE Transactions on Image Processing, vol. 9, pp. 792-800, May 2000.

[MOY01] E. Moyano, F.J. Quiles, A. Garrido, L. Orozco, J. Duato, *Efficient 3-D wavelet transform decomposition for video compression*, International Workshop on Digital and Computational Video, February 2001.

[MUK02] D. Mukherjee, S. K. Mitra, *Successive refinement lattice vector quantization*, IEEE Transactions on Image Processing, vol. 11, pp. 1337-1348, December 2002.

[MUR98] E. Murata, M. Ikekawa, I. Kuroda, *Fast 2D IDCT implementation with multimedia instructions for a software MPEG2 decoder*, in Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 3105-3108, 1998.

[PAS76] R. Pasco, *Source Coding Algorithms for Fast Data Compression*, Ph.D. thesis, Standford University, 1976.

[PEA01] W. A. Pearlman, *Trends of tree-based, set partitioning compression techniques in still and moving image systems*, Picture Coding Symposium, pp. 1-8, April 2001.

[PEA04] W. A. Pearlman, A. Islam, N. Nagaraj, 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, pp. 1219-1235, November 2004.

[PEA98] W.A. Pearlman, A. Islam, *Brief report on testing for the JPEG2000 core experiment*, appendix B in compression vs. complexity tradeoffs for quadratic splitting systems, ISO/IEC/JTC1/SC29, June 1998.

[RAB02] M. Rabbani, R. Joshi, *An overview of the JPEG2000 still image compression standard*, Signal Processing: Image Communication, vol. 17, 2002.

[RAJ03] N. M. Rajpoot, R. G. Wilson, F. G. Meyer, R. R. Coifman, *Adaptive wavelet packet basis selection for zerotree image coding*, IEEE Transactions on Image Processing, vol. 12, pp. 1460-1472, December 2003.

[RAM93] K. Ramchandran, M. Vetterli, *Best wavelet packet bases in a rate-distortion sense*, IEEE Transactions on Image Processing, vol. 2, pp. 160-175, February 1993.

[RAO90] K.R. Rao, P. Yip, *Discrete Cosine Transform, Algorithms, Advantages, Applications*, Academic Press Professional, 1990.

[RAO96] K. R. Rao, J. J. Hwang, *Techniques and Standards for Image, Video and Audio Coding*, Prentice Hall PTR, 1996.

[RIS76] J. J. Rissanen, *Generalized Kraft Inequality and Arithmetic Coding*, IBM Journal of Research and Development, Vol. 20, 198-203, May 1976.

[RIS79] J. J. Rissanen, G. G. Langdon, *Arithmetic Coding*, IBM Journal of Research and Development, Vol. 23, 149-162, March 1976.

[RIS84] J. Rissanen, *Universal coding, information, prediction, and estimation*, IEEE Transactions on Information Theory, Vol. 30, pp. 629-636, July 1984.

[SAI96] A. Said, 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, no3, June 1996.

[SAY00] K. Sayood, *Introduction to Data Compression*, 2nd edition, an imprint of academic press, Morgan Kaufmann Publishers, 2000.

[SEC01] A. Secker, D. Taubman, *Motion/compensated highly scalable video compression using an adaptive 3D wavelet transform based on lifting*, IEEE International Conference on Image Processing, October 2001.

[SHA48] C. E. Shanon, *A Mathematical Theory of Communication*, Bell System Technical Journal, Vol. 27, 379-423, 623-656, 1948.

[SHA93] J.M. Shapiro, *Embedded image coding using zerotrees of wavelet coefficients*, IEEE Transactions on Signal Processing, Vol. 41, n12, December 1993.

[SHE96] Y. Sheng, *Wavelet Transform*, The transforms and applications handbook, pp. 747-827, CRC Press, 1996.

[SHO84] Y. Shoham, A. Gersho, *Efficient bit allocation for an arbitrary set of quantizers*, IEEE Transactions on Acoustic, Speech, Signal Processing, vol. 36, pp. 1445-1453, September 1984.

[SLA98] M. J. Slattery, J. L. Mitchell, *The Qx-coder*, IBM Journal of Research and Development, vol. 42, pp. 767-784, November 1998.

[SON05] L. Song, J. Xu, H. Xiong, F. Wu, *Content adaptive update for lifting-based motion-compensated temporal filtering*, Electronic Letters, vol. 41, pp. January 2005.

[SPR02] N. Sprljan, S. Grgic, M. Mrak, M. Grgic, *Modified SPIHT algorithm for wavelet packet image coding*, International Symposium on Video/Image Processing and Multimedia Communications (VIPromCom), 2002.

[STO88] J. A. Storer, *Data Compression- Methods and Theory*, New York: Computer Science Press, 1988.

[STR96] G. Strang, T. Nguyen, *Wavelets and Filter Banks*, Wellsley Cambridge Press, MA, 1996.

[SWE96] W. Sweldens, *The lifting scheme: a custom-design construction of biorthogonal wavelets*, Journal of Applied Computational and Harmonic Analysis, vol. 3, pp. 186-200, 1996.

[TAN00] K. T. Tan, M. Ghanbari, *A Multi-metric Objective Picture Quality Measurement Model for MPEG Video*, IEEE Transactions On Circuits and Systems for Video Technology, vol. 10, October 2000.

[TAN03] X. Tang, S. Cho, W. A. Pearlman, *Comparison of 3D set partitioning methods in hyperspectral image compression featuring an improved 3D-SPIHT*, Data Compression Conference, March 2003.

[TAU00] D. Taubman, *High Performance Scalable Image Compression with EBCOT*, IEEE Transactions on Image Processing, vol. 9, pp. 1158-1170, July 2000.

[TAU02] D. S. Taubman, M. W. Marcellin, *JPEG 2000: Image Compression Fundamentals, Standards and Practice*, Kluwer Academic Publishers, pp 262-281, 2002.

[TAU94] D. Taubman, A. Zakhor, *Multirate 3-d subband coding of video*, IEEE Transactions on Image Processing, vol. 3, pp. 572-588, September 1994.

[TSA96] M.J. Tsai, J. Villasenor, F. Chen, *Stack-run image coding*, IEEE Transactions on Circuits and Systems for Video Technology, vol. 6, pp. 519-521, October 1996.

[UYT99] G. Uytterhoeven, *Wavelets: software and applications*, Ph.D. dissertation, Dep. Computerwetenschappen, Katholieke Universiteit Leuven, April 1999.

[VAI93] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, Signal Processing Series, 1993.

[VCE] Video Coding Expert Group ftp server, available at http://ftp3.itu.int/av-arch/

[VET92] M. Vetterli, C. Herley, *Wavelets and Filter Banks: Theory and Design*, IEEE Transactions on Signal Processing, vol. 40, pp. 2207-2232, 1992.

[VET96] M. Vetterli, J. Kovacevic, *Wavelets and Subband Coding*, Prentice Hall, Englewood Cliffs, 1995.

[VIL95] J. Villasenor, B. Belzer, Judy Liao, *Wavelet Filter Evaluation for Image Compression*, IEEE Transactions on Image Processing, vol. 4, no 8, pp. 1053-1060, August 1995.

[VIS94] M. Vishwanath, *The recursive pyramid algorithm for the discrete wavelet transform*, IEEE Transactions on Signal Processing, March 1994.

[WEI00] M. Weinberger, G. Seroussi, G. Sapiro, *The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS*, IEEE Transactions on Image Processing, Vol. 9, 1309-1324, August 2000.

[WIE00] M. Wien, *Hierarchical wavelet video coding using warping prediction*, IEEE International Conference on Image Processing, September 2000.

[WIT87] I. H. Witten, R. M. Neal, J. G. Cleary, *Arithmetic coding for compression*, Commun. ACM, Vol. 30, 520-540, June 1987.

[WOO86] J.W. Woods, S. O'Neil, "Subband coding of images", IEEE Transactions on Acoustic, Speech, Signal Processing, vol. 34, pp. 1278-1288, October 1986.

[WU01] X. Wu, *Compression of Wavelet Transform Coefficients*, The Transform and Data Compression Handbook, pp. 347-378, CRC Press, 2001.

[WU96] X. Wu, N.D. Memon, *CALIC- A Context Based Adaptive Lossless Image Coding Scheme*, IEEE Transactions on Communications, Vol. 45, 437-444, May 1996.

[WU98] X. Wu, *High-order context modeling and embedded conditional entropy coding of wavelet coefficients for image compression*, in Proc. 31st Asilomar Conf. Signals, Systems, Computers, vol. 23, pp. 1378-1382, 1998.

[WUH01] H. Wu, Z. Yu, S. Winkler, T. Chen, *Impairment Metrics for MC/DPCM/DCT Encoded Digital Video*, Picture Coding Symposium, Seoul, April 2001.

[XIE05] G. Xie, H. Shen, *Highly-scalable, low-complexity image coding using zeroblocks of wavelet coefficients*, IEEE Transactions on circuits and systems for video technology, Vol. 15, pp. 763-770, June 2005.

[XIO97] Z. Xiong, K. Ramchandran, M. Orchard, *Space-frequency quantization for wavelet image coding*, IEEE Transactions on Image Processing, vol. 46, pp. 677-693, May 1997.

[XIO98] Z. Xiong, K. Ramchandran, M. T. Orchart, *Wavelet packet image coding using space-frequency quantization*, IEEE Transactions on Image Processing, vol. 7, pp. 892-898, June 1998.

[ZEN02] W. Zeng, S. Daly, S. Lei, *An overview of the visual optimization tools in JPEG2000*, Signal Processing: Image Communication, vol. 17, 2002.

[ZER01] N. Zervas, G. Anagnostopoulos, V. Spiliotopoulos, Y. Andreopoulos, C. Goutis, *Evaluation of Design Alternatives for the 2-D-Discrete Wavelet Transform*, IEEE Transactions on Circuits and Systems for Video Technology, vol. 11, pp. 1246-1262, December 2001.

[ZIV77] J. Ziv, A. Lempel, *A Universal Algorithm for Data Compression*, IEEE Transactions on Information Theory, Vol. 23, 337-343, May 1977.

[ZIV78] J. Ziv, A. Lempel, *Compression of Individual Sequences via Variable-Rate Coding*, IEEE Transactions on Information Theory, Vol. 24, 530-536, September 1978.

Appendix A

Joint scalar/bit-plane uniform quantization

All the algorithms proposed in this thesis employ a quantization mechanism based on two parameters, one finer (Q) and another coarser (*rplanes*). Thus, the quantized image is the result of jointly applying two quantization methods. The first method performs a scalar quantization with a step-size of $2Q$, and it can be computed during the wavelet transform normalization process. The second quantization step consists in removing the *rplanes* least significant bits of all coefficients, being a simple bit-plane quantization process.

The use of both quantization processes may seem a bit strange, but it reveals more natural when the algorithms are studied in depth. The coarser quantization is useful to shorten the number of bits needed to represent a coefficient, and to concentrate the symbol probability. In addition, it allows the introduction of quantization in architectures that only support integer arithmetic. Finally, with this type of quantization, some values are never employed by significant coefficients (in particular those $|c_{i,j}| < 2^{rplanes}$), and this range can be used to represent specific marks and control symbols (such as *LOWER* and *ISOLATED_LOWER*), allowing in-place symbol computation (which avoids the introduction of extra memory to store those symbols). In the bit-plane quantization, the available step-sizes are always power of two, and thus its granularity is very low. Therefore, a fine control of the image compression is not possible with only this quantization parameter. In order to perform a finer rate control, a scalar quantization stage is required.

In spite of the use of two quantization methods, the quantized wavelet coefficient can be expressed mathematically formulated with the following equations (let us call *c* the initial coefficient and c_Q the quantized coefficient):

$$if\ c>0 \quad c_Q = \left\lfloor \left(\left\lfloor \frac{c}{2Q} + 0.5 \right\rfloor + K \right) \Big/ 2^{rplanes} \right\rfloor \tag{A.1}$$

$$if\ c<0 \quad c_Q = \left\lceil \left(\left\lceil \frac{c}{2Q} - 0.5 \right\rceil - K \right) \Big/ 2^{rplanes} \right\rceil \tag{A.2}$$

$$if\ c=0 \quad c_Q = 0 \tag{A.3}$$

Note that an integer constant *K* can be used to adjust the bit plane quantization (by taking some values out of the deadzone), which may be useful in some cases. Experimental tests have revealed that *K=1* is a good value, increasing the R/D performance for most source images. On the other hand, the coefficients recovered on the decoder side are (let us call c_R a coefficient recovered from c_Q):

$$if\ c_Q>0 \quad c_R = \left(2\left(\left(2c_Q+1\right)2^{rplanes-1} - K\right)-1\right)Q \tag{A.4}$$

$$if\ c_Q<0 \quad c_R = \left(2\left(\left(2c_Q-1\right)2^{rplanes-1} + K\right)+1\right)Q \tag{A.5}$$

$$if\ c_Q=0 \quad c_R = 0 \tag{A.6}$$

In both dequantization processes, i.e. in the standard scalar dequantization and in the dequantization from the bit-plane removing, the c_R value is adjusted to the midpoint within the recovering interval, reducing in this way the quantization error.

These equations may be clearer if we observe both dequantization processes separately. First, we have to recover the initial number of bits of the scalar quantized coefficient, thus if $c_Q > 0$

$$c'_R = c_Q 2^{rplanes} + 2^{rplanes-1} = \left(2c_Q+1\right)2^{rplanes-1} \tag{A.7}$$

then the constant *K* is subtracted and the scalar dequantization is applied

$$c_R = \left(2\left(c'_R - K\right)-1\right)Q \tag{A.8}$$

With the expressions shown above, a relation between both quantization parameters, *Q* and *rplanes*, can be established. In fact, we can see that two different *rplanes* values may represent approximately the same global quantization whenever we choose the suitable *Q* value. In particular, if *rplanes* is decreased by one, *Q* should be multiplied by two to preserve the same global quantization, whereas if *rplanes* is increased by one *Q* should be divided by two.

As an example, Figure A.1 shows the relation between both quantization parameters and the final bit rate achieved with Lena encoded with LTW. In these curves, it can be easily seen the equivalence previously mentioned. The bit rate corresponding to *Q*=0.2 and *rplanes*=6 is roughly 0.5 bpp. If we decrease *rplanes* in one, the same bit rate is achieved with *Q*=0.4, and if we decrease it again it is achieved with *Q*=0.8. In addition, with a constant finer quantization parameter, this figure shows that when rplanes is decreased by one, the bit rate is halved. E.g., for a fixed value *Q*=0.8, the bit rate is 1 bpp with *rplanes*=2, it is 0.5 with *rplanes*=3, it is 0.25 with *rplanes*=4, and so on.

The same relation between both quantization parameters can be observed in Figure A.2, where the image quality (PSNR) is evaluated as a function of the quantization parameters. As in the previous graph, we see that the same PSNR is achieved with *Q*=0.2 and *rplanes*=4, with *Q*=0.4 and *rplanes*=3, etc.

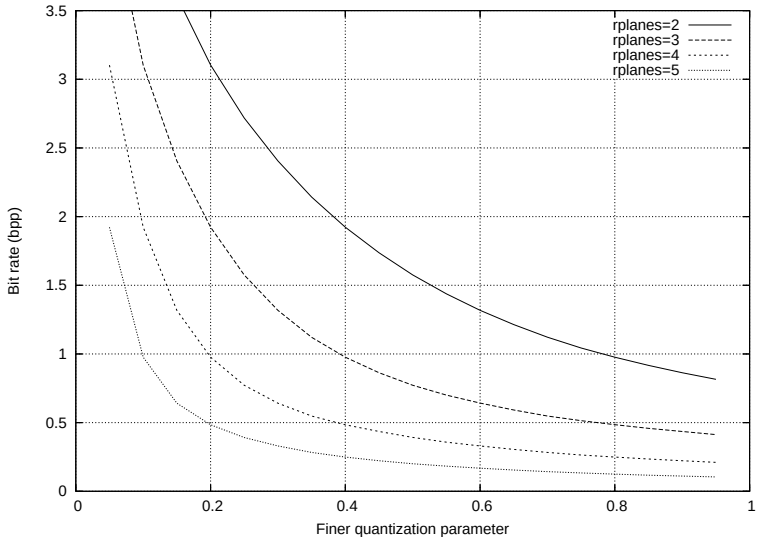


Fig.A.1: Relation between quantization parameters and bit rate for Lena.

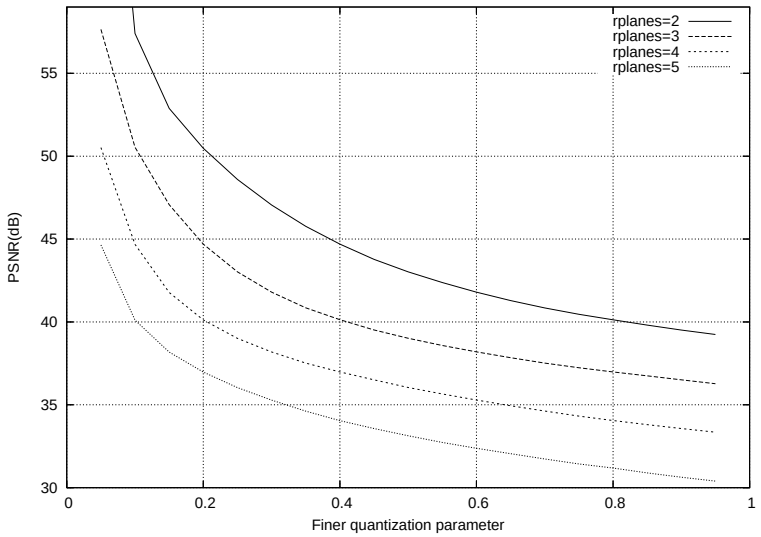


Fig.A.2: Relation between quantization parameters and PSNR.

Appendix B

Rate control in the proposed algorithms

Precise rate control is not feasible in the algorithms RLW and LTW presented in Chapters 5 and 6, because in order to achieve precise rate control, we need bit plane coding or iterative methods that significantly increase the complexity of the encoder, which is unsuitable for our purposes. For example, EZW and SPIHT exhibit very precise rate control by coding the coefficients bit-plane by bit-plane, but being slow and increasing the cache miss rate. Other encoders, like EBCOT and SFQ, achieve less precise rate control by using iterative methods with rate-distortion optimization algorithms, which are even slower than bit plane coding (although more efficient). Therefore, the RLW and LTW encoders do not present any type of iterative method, and each coefficient is encoded as soon as it is visited. So it is not possible to perform precise rate control and optimal SNR scalability (which can be approximated by magnifying the different low-frequency subbands as the inverse transform is computed). Both RLW and LTW simply apply a constant quantization (determined by Q and *rplanes*) to all the wavelet coefficients, encoding the image at a constant and uniform quality, as it happened in the former JPEG standard (where only a quality parameter was available, and no rate control was performed). Note that other very fast wavelet image encoders reported in the literature (like PROGRES) have also taken this approach.

Although sometimes constant quality is a desired feature when encoding a group of images (and it is easily achieved by applying the same quantization to all of them), a constant size of the compressed image is required more often. For example, in a digital camera, for a given memory size, the number of remaining photos that still can be taken and held in memory is commonly given. In order to obtain this feature, a certain rate control is needed.

For our algorithms, which work with fixed quantization parameters, a rate control can be seen as a function (RC) that returns the suitable quantization (*Q* and rplanes) for a given transform image *C* and a desired bit rate *b*. That is

$$\mathrm{RC}(C,b)=(Q,rplanes) \qquad \text{(B.1)}$$

The rate control function could be implemented by modeling the encoding process with a model M that returns the final bit rate achieved when encoding the image with the given quantization parameters,

$$\mathrm{M}(C_{Q,rplanes})=b \qquad \text{(B.2)}$$

However, this rate control method presents several drawbacks. First, it is an iterative method, because the model should be applied with several quantization parameters in order to determine the combination of quantization parameters that obtains the desired bit rate (or that is close enough to it). Nevertheless, it could be a valid method if we could establish a very fast model that would precisely return that bit rate. Unfortunately, it is not possible unless we actually encode the image (or at least analyze the number of trees that can be formed and the third-order entropy of the symbols to be encoded), which results in a very slow rate control method. Therefore, another alternative is needed.

The rate control that we propose is based on a fast analysis of the transform image features, extracting one or more numerical parameters that are employed to determine the quantization parameters needed to approximate a bit rate. In order to simplify the quantization process, we suggest the use of a fixed *rplanes* value. Particularly, we propose *rplanes* equal to two to be able to perform in-place processing (see Appendix A). Then, we can use the first-order entropy of the wavelet coefficients in order to determine the complexity of an image (level of detail) and therefore to estimate the number of lower trees that will be formed. In Figure B.1, we show a simple study of the *Q* parameter employed to encode seven popular images (with different entropy) at various bit rates. With these curves, we can conclude that there is a correspondence between the first-order entropy of the coefficients and the quantization parameter, which allows us to predict the quantization step needed to encode an image at a given bit rate simply by computing its entropy. However, this correspondence becomes less precise as the bit rate decreases, mainly because at low bit rates, the compressed image file is very small and a slight error in the final image size largely modifies the final bit rate with respect to the desired one (in other words, an exact rate control is more difficult at low bit rates). Other groups of images have also been analyzed, with

similar results to those shown in Figure B.1.

Although the proposed rate control method is not very precise, it is very fast and can be employed to determine some parameters in real implementations and specific applications, such as the number of pictures that still can be introduced and stored in a memory card. In addition, when successively encoding a group of images, the quantization parameter can be adaptively tuned to progressively compensate small errors.

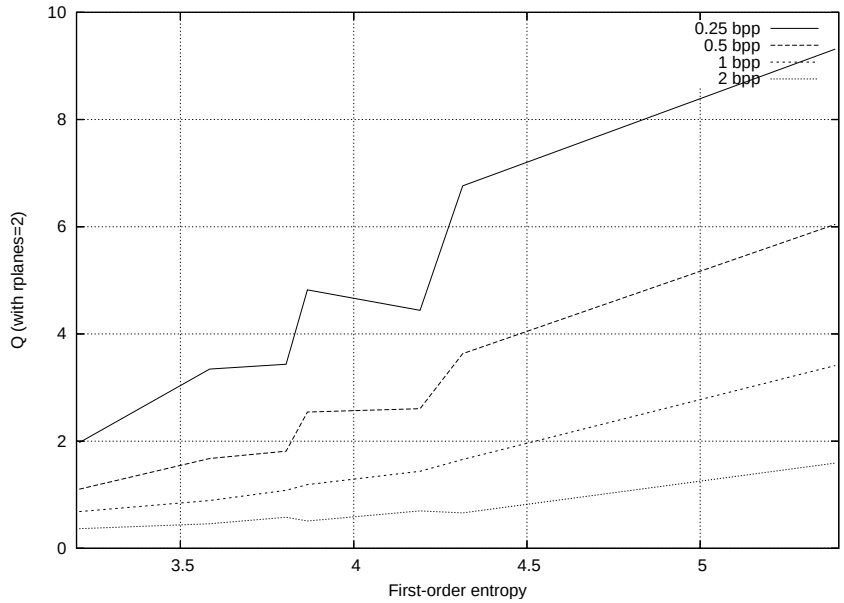


Fig.B.1: Quantization parameter (Q) used to encode with LTW images with various entropies at different bit rates. The images employed are (in order of entropy of the wavelet coefficients): Zelda (3.2), Lena (3.58), Peppers (3.80), Boat (3.86), Goldhill (4.19), Barbara (4.31) and Baboon (5.39).

Appendix C

Implementation of the efficient DWT

In this Appendix, an operative implementation of the filtering algorithm to compute the DWT proposed in Chapter 3 is given in ANSI C language. This implementation is also available at http://www.disca.upv.es/joliver/thesis.

In the first two sections, the forward wavelet transform (FWT) is implemented. Section C.1 implements the recursive backward function that is used in Subsection C.2 in order to compute the wavelet transform. In the following two sections, the inverse wavelet transform (IWT) is implemented in a similar way. In section C.5, some functions and variables that are used by the previous functions are given. Finally, some useful function headers are defined.

C.1 Backward Recursion Function

```
int GetLLlineBwd(int step, int width, float *Dest)
{
int f,g;
float **BuffLevel=BufferTransf[step];

if (step<0) return(ReadImageLine(Dest));
if (SymmetryPoint[step]<=NTaps) return EOL;
if (!BuffLevel[0])
{
       for (f=0;f<BufferSize;f++)
              BuffLevel[f]=(float *)malloc(width*sizeof(float));
       for (f=0;f<NTaps+1;f++)
       {
              GetLLlineFwd(step-1, width*2, BuffLevel[f+NTaps]);
              LineTransform(width,BuffLevel[f+NTaps]);
       }
       for(f=NTaps-1,g=NTaps+1;f>=0;f--,g++)
              LineCopy(width,BuffLevel[g],BuffLevel[f]);
}
```

```
else repeat(2)
{
       ShiftLines(BuffLevel);
       If (GetLLlineFwd(step-1,width*2,BuffLevel[BufferSize-1])!=EOL)
              LineTransform(width,BuffLevel[BufferSize-1]);
       else
       {
              LineCopy(width,BuffLevel[SymmetryPoint[step]-
                            Radious[step]],BuffLevel[BufferSize-1]);
              Radious[step]++,SymmetryPoint[step]--;
       }
}
ColumnTransformLow(width, BuffLevel, Dest, HL);
ColumnTransformHigh(width, BuffLevel, LH, HH);
ProcessSubbands(HL,LH,HH,step);
return OK;
}
```

C.2 Implementation of the Wavelet Transform

```
int WaveletTransform(int Nsteps,int width,int height)
{
int f,g;
NTapsMax=NTaps>NTapsInv?NTaps:NTapsInv;
BufferSize=(NTapsMax<<1)+1;

CoefExt=(float *)malloc((width+(NTapsMax<<1))*sizeof(float));
CoefExtIni=CoefExt+NTapsMax;
BufferTransf=(float***) malloc(Nsteps*sizeof(float(**)));
for (f=0;f<Nsteps;f++)
   BufferTransf[f]=(float**) malloc(BufferSize*sizeof(float(*)));
for (f=0;f<Nsteps;f++)
       for (g=0;g<BufferSize;g++)
              BufferTransf[f][g]=NULL;
SymmetryPoint=(int *)malloc(Nsteps*sizeof(int));
for (f=0;f<Nsteps;f++)
       SymmetryPoint[f]=BufferSize-2;
Radious=(int *)malloc(Nsteps*sizeof(int));
for (f=0;f<Nsteps;f++)
       Radious[f]=1;
LL=(float *)malloc((width>>Nsteps)*sizeof(float));
LH=(float *)malloc((width>>1)*sizeof(float));
HL=(float *)malloc((width>>1)*sizeof(float));
HH=(float *)malloc((width>>1)*sizeof(float));

for (f=0;f<(height/(1<<Nsteps));f++)
{
       GetLLlineFwd(Nsteps-1, width/(1<<(Nsteps-1)), LL);
       ProcessLLSubband(LL);
}

for (f=0;f<Nsteps;f++)
       for (g=0;g<BufferSize;g++)
              free(BufferTransf[f][g]);
free(LL);free(HL);free(LH);free(HH);
free(Radious); free(SymmetryPoint);
for (f=0;f<Nsteps;f++)
       free(BufferTransf[f]);
free(BufferTransf); free(CoefExt);
return 0;
}
```

C.3 Forward Recursion Function

```
int GetLLlineFwd(int step, int width, float *Dest)
{
int f,g;
float **BuffLevel=BufferTransf[step];
int halfwidth=width/2;

if (step>lastStep) return ReadLLline(Dest);
if (LinesInBuffer[step])
{
       InvColumnTransformLow(width, BuffLevel, Dest);
       LinesInBuffer[step]=0;
       return OK;
}
if (SymmetryPoint[step]<=NTapsInv) return EOL;
if (!BuffLevel[0])
{
       for (f=0;f<BufferSize;f++)
              BuffLevel[f]=(float *) malloc(width*sizeof(float));
       for (f=NTapsInv;f<BufferSize;)
       {
              GetLLlineBwd(step+1, halfwidth, BuffLevel[f]);
              ReadSubbandLine(
                           BuffLevel[f]+halfwidth,
                           BuffLevel[f+1],
                           BuffLevel[f+1]+halfwidth,
                           step);
              InvLineTransform(width,BuffLevel[f++]);
              InvLineTransform(width,BuffLevel[f++]);
       }
       for (f=NTapsInv-1,g=NTapsInv+1;f>=0;f--,g++)
              LineCopy(width,BuffLevel[g],BuffLevel[f]);
}
else
{
       ShiftLines(BuffLevel);
       ShiftLines(BuffLevel);
       if (GetLLlineBwd(step+1,halfwidth, BuffLevel[BufferSize-2])==OK)
       {
         ReadSubbandLine(
                           BuffLevel[BufferSize2]+halfwidth,
                           BuffLevel[BufferSize-1],
                           BuffLevel[BufferSize-1]+halfwidth
                           step);
         InvLineTransform(width,BuffLevel[BufferSize-2]);
         InvLineTransform(width,BuffLevel[BufferSize-1]);
       }
       else
       {
              LineCopy(width,BuffLevel[SymmetryPoint[step]-
                           Radious[step]-1],BuffLevel[BufferSize-2]);
              LineCopy(width,BuffLevel[SymmetryPoint[step]-
                           Radious[step]-2],BuffLevel[BufferSize-1]);
              Radious[step]+=2;
              SymmetryPoint[step]-=2;
       }
}
InvColumnTransformHigh(width, BuffLevel, Dest);
LinesInBuffer[step]=1;
return OK;
}
```

C.4 Implementation of the Inverse Wavelet Transform

```
int InvWaveletTransform(int Nsteps,int width,int height)
{
float *ImageLine;
int f,g;

NTapsMax=NTaps>NTapsInv?NTaps:NTapsInv;
BufferSize=(NTapsMax<<1)+1; lastStep=Nsteps-1;
CoefExt=(float *) malloc((width+(NTapsMax<<1))*sizeof(float));
CoefExtIni=CoefExt+NTapsMax;
BufferTransf=(float***) malloc(Nsteps*sizeof(float(**)));
for (f=0;f<Nsteps;f++)
       BufferTransf[f]=(float **) malloc(BufferSize*sizeof(float(*)));
for (f=0;f<Nsteps;f++)
       for (g=0;g<BufferSize;g++)
              BufferTransf[f][g]=NULL;
SymmetryPoint=(int *) malloc(Nsteps*sizeof(int));
for (f=0;f<Nsteps;f++)
       SymmetryPoint[f]=BufferSize-3;
Radious=(int *)malloc(Nsteps*sizeof(int));
for (f=0;f<Nsteps;f++)
       Radious[f]=0;
LinesInBuffer=(int *)malloc(Nsteps*sizeof(int));
for (f=0;f<Nsteps;f++)
       LinesInBuffer[f]=0;
ImageLine=(float *)malloc(width*sizeof(float));
for (f=0;f<height;f++)
{
       GetLLlineBwd(0, width, ImageLine);
       ProcessLine(ImageLine);
}
for (f=0;f<Nsteps;f++)
       for (g=0;g<BufferSize;g++)
              free(BufferTransf[f][g]);
free(ImageLine); free(LinesInBuffer);
free(Radious); free(SymmetryPoint);
for (f=0;f<Nsteps;f++)
       free(BufferTransf[f]);
free(BufferTransf); free(CoefExt);
return 0;
}
```

C.5 Auxiliary Functions and Global Variables

In order to get a complete module for the proposed wavelet transform, the previous functions can be appended to the functions and variables defined in this section.

```
#include <stdlib.h>
#include <string.h>
#include "external_headers.h"

#define OK 0
#define EOL 1

static float ***BufferTransf;
static float *CoefExt,*CoefExtIni;
static int *SymmetryPoint, *Radious, *LinesInBuffer;
```

```
static int BufferSize,NTapsMax,lastStep;
static float *LL,*HL,*LH,*HH;
static int NTaps=4;
static int NTapsInv=3;
static float B79_AnLowPass[]=
       {0.85269868F, 0.37740286F,-0.11062440F, -0.02384947F, 0.03782846F};
static float B79_AnHighPass[]=
       {0.78848562F, -0.41809227F, -0.04068942F, 0.06453888F};
static float B79_SynLowPass[]=
       {0.85269868F, 0.41809227F, -0.11062440F, -0.06453888F, 0.03782846F};
static float B79_SynHighPass[] =
       {0.78848562F,-0.37740286F,-0.04068942F, 0.02384947F};
#define repeat(x) for (int __indx=0;__indx<x;__indx++)

#define LineCopy(n, source, dest) memcpy(dest,source,n*sizeof(float))

inline void InterleaveLine (float *Src,float *Dest, int width)
{
       float *HalfSrc=Src+(width>>1);
       for (int x=0;x<(width>>1);x++) {*Dest++=*Src++; *Dest++=*HalfSrc++;}
}

inline void ShiftLines(float **Buff)
{
       float *Aux=Buff[0];
       for (int f=0;f<BufferSize-1;f++)
              Buff[f]=Buff[f+1];
       Buff[BufferSize-1]=Aux;
}

inline void SymmetricExt(float *ini,float *end)
{
       for (int x=1;x<=NTapsMax;x++)
       {
              ini[-x]=ini[x];
              end[x]=end[-x];
       }
}

inline float FourTapsFilter(float *c,float *t)
{
       return (
              t[0]*c[0]+
              t[1]*(c[1]+c[-1])+
              t[2]*(c[2]+c[-2])+
              t[3]*(c[3]+c[-3]));
}

inline float FiveTapsFilter(float *c,float *t)
{
       return (
              t[0]*c[0]+
              t[1]*(c[1]+c[-1])+
              t[2]*(c[2]+c[-2])+
              t[3]*(c[3]+c[-3])+
              t[4]*(c[4]+c[-4]));
}

void LineTransform(int width, float *Line)
{
float *CoefAuxL,*CoefAuxH;
LineCopy(width,Line,CoefExtIni);
SymmetricExt(CoefExtIni,CoefExtIni+width-1);
CoefAuxL=Line; CoefAuxH=Line+(width>>1);
for (int x=0;x<width;)
{
       *CoefAuxL++= FiveTapsFilter(CoefExtIni+x,B79_AnLowPass);
       x++;
```

```
        *CoefAuxH++= FourTapsFilter(CoefExtIni+x,B79_AnHighPass);
        x++;
}
}

void InvLineTransform(int width, float *Line)
{
InterleaveLine(Line,CoefExtIni, width);
SymmetricExt(CoefExtIni,CoefExtIni+width-1);
for (int x=0;x<width;)
{
        *Line++=FourTapsFilter(CoefExtIni+x,B79_SynHighPass);
        x++;
        *Line++=FiveTapsFilter(CoefExtIni+x,B79_SynLowPass);
        x++;
}
}

inline float FiveTapsColumnFilter(float *t, float **B, int x)
{
        return (
                t[0]*B[4][x]+
                t[1]*(B[3][x]+B[5][x])+
                t[2]*(B[2][x]+B[6][x])+
                t[3]*(B[1][x]+B[7][x])+
                t[4]*(B[0][x]+B[8][x]));
}

inline float FourTapsColumnFilter(float *t, float **B, int x)
{
  return (
                t[0]*B[5][x]+
                t[1]*(B[6][x]+B[4][x])+
                t[2]*(B[7][x]+B[3][x])+
                t[3]*(B[8][x]+B[2][x]));
}

void ColumnTransformLow(int width, float **BuffLevel, float *LL, float *HL)
{
int x=0;
while (x<width/2)
        *LL++=FiveTapsColumnFilter(B79_AnLowPass, BuffLevel, x++);
while (x<width)
        *HL++=FiveTapsColumnFilter(B79_AnLowPass, BuffLevel, x++);
}

void ColumnTransformHigh(int width, float **BuffLevel, float *LH, float *HH)
{
int x=0;
while (x<width/2)
        *LH++=FourTapsColumnFilter(B79_AnHighPass, BuffLevel, x++);
while (x<width)
        *HH++=FourTapsColumnFilter(B79_AnHighPass, BuffLevel, x++);
}

void InvColumnTransformHigh (int width, float **BuffLevel, float *LL)
{
for (int x=0;x<width;x++)
        *LL++=FourTapsColumnFilter(B79_SynHighPass, BuffLevel-2, x);
}

void InvColumnTransformLow(int width, float **BuffLevel, float *LL)
{
    for (int x=0;x<width;x++)
                *LL++=FiveTapsColumnFilter(B79_SynLowPass, BuffLevel, x);
}
```

C.6 External Headers

A few functions employed in the previous procedures are dependant on the final application, and are left as external and open functions. In general, these functions are used to read data line-by-line (namely, `ReadImageLine()` is used to read an image line in the FWT, and `ReadLLline`() `ReadSubbandLine`() serve to read subband lines for the IWT) and to encode the generated coefficients line-by-line (in particular, `ProcessSubbands`(), `ProcessLLSubband`() are used to encode and release a subband line once it is calculated in the FWT, and `ProcessLine`() is called when an image line is obtained in the IWT).

```
/* EXTERNAL FUNCTIONS USED BY THE FWT */

/* Read an image line on Dest */
int ReadImageLine(float *Dest);

/* Process or store one line achieved from every wavelet subband (HL, LH and
HH) at level=step */
void ProcessSubbands(float *HL,float *LH,float *HH,int step);

/* Process or store a line achieved from the LL subband */
void ProcessLLSubband(float *LL);

/* EXTERNAL FUNCTIONS USED BY THE IWT */

/* Read a line from the LL subband */
int ReadLLline(float *Dest);

/* Read a line from every wavelet subband (HL, LH and HH) at level = step */
void ReadSubbandLine(float *HL, float *LH, float *HH, int step);

/* Process or store an achieved image line */
void ProcessLine(float *ImageLine);
```

Appendix D

Reference images

In this Appendix, the reference images that have been employed in this thesis to evaluate the proposed encoders and to compare them with other proposals are shown. All of them are standard images. In particular, Lena, Goldhill and Barbara are from the USC database, while Café and Woman belong to the JPEG 2000 test bed.

D.1 Lena

Grayscale, 8 bpp, 512×512.

D.2 Goldhill

Grayscale, 8 bpp, 512×512.

D.3 Barbara

Grayscale, 8 bpp, 512×512.

D.4 Café

Grayscale, 8 bpp, 2048×2560.

D.5 Woman

Grayscale, 8 bpp, 2048×2560.

Appendix E

Post compressed images for subjective comparison

In this Appendix, the reference images of Appendix D are shown after being compressed at moderate to low bit rates (in particular, at 1 bpp and 0.125 bpp), using the LTW encoder of Chapter 6, and the state of the art encoders SPIHT and JPEG 2000. These images allow us to perform a subjective comparison of the encoders. However, since the three encoders achieve similar PSNR results, the resulting visual quality of the images compressed with these encoders is not very different, which validates the use of the PSNR as an objective metric in our research.

E.1 Lena

LTW, at 1 bpp, PSNR= 40.50 dB.

SPIHT, at 1 bpp, PSNR= 40.41 dB.

JPEG 2000, at 1 bpp, PSNR= 40.31 dB.

LTW, at 0.125 bpp, PSNR= 31.25 dB.

SPIHT, at 0.125 bpp, PSNR= 31.10 dB.

JPEG 2000, at 0.125 bpp, PSNR= 30.84 dB.

E.2 Goldhill

LTW, at 1 bpp, PSNR= 36.74 dB.

SPIHT, at 1 bpp, PSNR= 36.55 dB.

JPEG 2000, at 1 bpp, PSNR= 36.53 dB.

LTW, at 0.125 bpp, PSNR= 28.60 dB.

SPIHT, at 0.125 bpp, PSNR= 28.48 dB.

JPEG 2000, at 0.125 bpp, PSNR= 28.35 dB.

E.3 Barbara

LTW, at 1 bpp, PSNR= 36.72 dB.

SPIHT, at 1 bpp, PSNR= 36.41 dB.

JPEG 2000, at 1 bpp, PSNR= 36.54 dB.

LTW, at 0.125 bpp, PSNR= 25.24 dB.

SPIHT, at 0.125 bpp, PSNR= 24.86 dB.

JPEG 2000, at 0.125 bpp, PSNR= 25.25 dB.

E.4 Café

LTW, at 1 bpp, PSNR= 32.04 dB.

SPIHT, at 1 bpp, PSNR= 31.74 dB.

JPEG 2000, at 1 bpp, PSNR= 32.04 dB.

LTW, at 0.125 bpp, PSNR= 20.76 dB.

SPIHT, at 0.125 bpp, PSNR= 20.67 dB.

JPEG 2000, at 0.125 bpp, PSNR= 20.74 dB.

E.5 Woman

LTW, at 1 bpp, PSNR= 38.49 dB.

SPIHT, at 1 bpp, PSNR= 38.28 dB.

JPEG 2000, at 1 bpp, PSNR= 38.43 dB.

LTW, at 0.125 bpp, PSNR= 27.50 dB.

SPIHT, at 0.125 bpp, PSNR= 27.33 dB.

JPEG 2000, at 0.125 bpp, PSNR= 27.33 dB.