JPEG2000 - A Short Tutorial

Gaetano Impoco*

April 1, 2004

Disclaimer

This document is intended for people who want to "be aware" of some of the mechanisms underlying the JPEG2000 image compression standard [2]. While writing this tutorial my objective was to give to the reader the feeling of what goes on when compressing images with JPEG2000 . Hence, this tutorial should be regarded as introductory material which requires further reading to reach a deeper understanding of the topics presented.

The text has not undergone a critical review process, so it might contain mistakes (and it contains lots of them, that's for sure!) and lack clarity. The English may appear a bit rough to native speakers. Sorry for that. It's the best I can do.

The author is not responsible for any misuse of the present document or of any part of it, or for any misunderstanding deriving from the mistakes it might contain. This document is intended for personal use only. Any public use of any part of it is subject to the permission of the author.

For any question about this document please, feel free to contact me. Suggestions and criticism are welcomed.

That's all guys. Have fun!

1 Introduction

Due to the increasing demand for amazing graphic design in low-band applications (e.g. web sites) on the one hand and in heavy duty applications such as GIS on the other, image compression has gained great attention over the last two decades. Many standard generic compression techniques have effectively been employed for image compression. However, due to the peculiar nature of image data new image-oriented techniques have been devised. In particular, lossy compression has gained increasing popularity since the release of the JPEG standard [5] whose coding pipeline is considered a general standard scheme for high-performance coding. The same scheme is used by the JPEG2000 standard with minor structural modifications. Before entropy coding the data stream is pre-processed in order to reduce its entropy (see Figure 1). A linear transform is used to obtain a representation of the input image in a different domain. A (possibly uneven) quantisation is carried out on the transformed coefficients in order to smooth out high frequencies. The rationale for this is that the human vision system is little sensitive to high frequencies. Since high frequencies carry a high information content, quantisation considerably reduces the signal entropy while retaining perceptual quality. The transformed data is re-ordered so as to reduce the relative difference between quantised coefficients. This pre-processed stream is finally entropy coded using standard techniques (e.g. Huffman + run-length is used by JPEG). A compression rate control is sometimes added.

This document is organised as follows. Section 2 deals with the sequence of operations performed onto the input data prior to the entropy coding, which is discussed in Section 3. Codestream organisation

*Visual Computing Lab - ISTI-CNR Pisa, Italy. E-mail: impoco@di.unipi.it

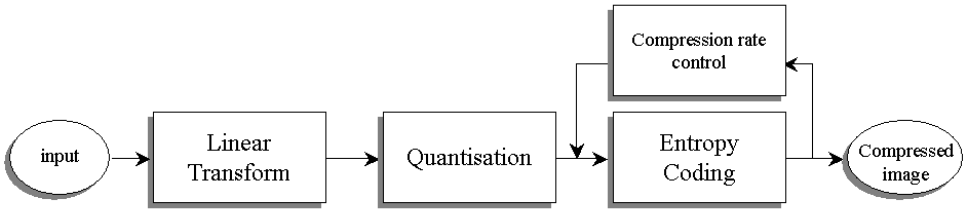

Figure 1: Common image compression pipeline (in the spirit of JPEG).

is presented in Section 4. The whole JPEG2000 coding process is summarised in Section 5. Finally, Section 6 discusses some extensions.

2 Data Pre-Processing

JPEG2000 [2] is a newborn standard for image compression and transmission. It may compress efficiently both continuous-tone and indexed images (see Section 6) in lossy or lossless mode. Resolution and quality scalability, error resilience, and region-of-interest coding are just a few of its features. JPEG2000 employs a number of mechanisms which are regarded as the state-of-the-art in lossy and lossless image compression. The pipeline of JPEG2000 is in the spirit of that described in Section 1 and is shown in Figure 2. The first step of the JPEG2000 encoder is the **colour transform**. Usually colour images

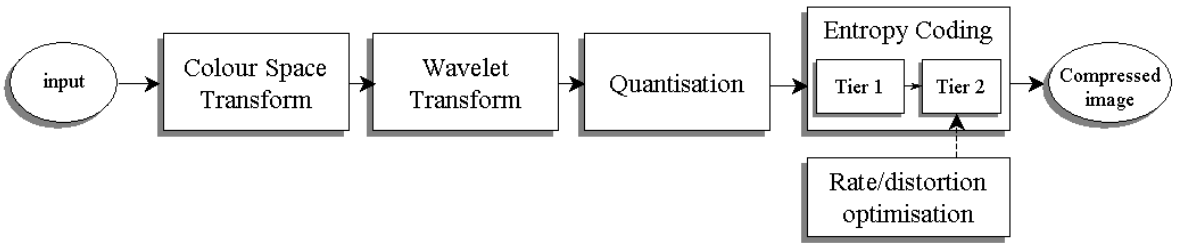

Figure 2: JPEG2000 image compression pipeline.

are stored as a series of RGB triplets. However, the RGB colour space is not colour consistent, i.e. a slight variation in one colour component of a pixel might lead to noticeable artifacts in the other colour components (*colour artifacts*). There exist other colour spaces equivalent to RGB in which this is not true. This initial colour transform step is thus performed to convert RGB images into some other more convenient representation. Colour transform does not have great impact on lossless compression since the image data is restored exactly by the decoder. On the contrary, it is a crucial step in lossy mode. Compression performance may also benefit from component independence of the transformed colour space, since it allows the user to leave the luminance channel unchanged, and down-sample colour components without any (or hardly noticeable) perceptual loss in the reconstructed image.

Colour planes are encoded separately as if they were greylevel images. They are usually divided into equally-sized blocks called *tiles*. For each tile a disjoint codestream is generated using the same pipeline. This is mainly done in order to keep low the computational burden, and for relatively random access in the compressed domain.

The **wavelet transform** is applied to the tiles before entropy coding. The benefit of employing the

wavelet transform is that transformed data usually exhibits a lower entropy and is thus more "compressible". In particular, since wavelet transform separates a tile into four sub-bands, source modelling can be tailored to each sub-band. That is exactly what JPEG2000 does. In fact, since wavelet filters are designed so as to store different frequencies into each sub-band (see Figure 3), sub-bands exhibit peculiar features which are "captured" by the JPEG2000 source modeler. Figure 4 shows the effect of frequency

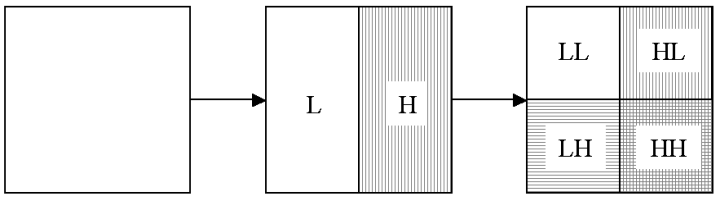

Figure 3: Wavelet transform: sub-band division and frequency filtering.

filtering on a greylevel image. A number of different wavelet filters is allowed by the standard for both

Figure 4: Wavelet transform: effect of frequency filtering on a greyscale image.

lossy and lossless compression. Lossy filters usually give better results but they involve floating point operations. Due to floating point approximation, the correct reconstruction of the input signal is not guaranteed using these filters. Filters involving only integer operations are also allowed to overcome this problem.

Wavelet transform can be followed by a **quantisation** step in lossy compression. Quantisation reduces the bitdepth of wavelet coefficients at the expenses of precision. Two different quantisation procedures are allowed by the standard: *scalar quantisation*, and *Trellis-coded quantisation*. Scalar quantisation consists of a simple truncation of less significant bits, often obtained by right shifting wavelet coefficients' magnitude. Coefficients differing only in the digits being cut off will be undistinguishable after *dequantisation*. This results in a step function as depicted in Figure 5. Scalar quantisation suffers from the so-called *dead-zone* problem. Namely, if a quantisation step Δ_b is used for a sub-band b, the coefficients whose magnitude falls below that threshold are clamped to zero. Thus the information on coefficients in a ball of radius Δ_b around zero will be completely lost (see Figure 5). Being $p[n]$ the number of right shifts for the n-th coefficient, $s[n]$, in the b-th sub-band, scalar quantisation can be expressed by the formula:

$$q^{p[n]}[n] = sign(s[n]) \cdot \left\lfloor \frac{|s[n]|}{\Delta_b^{p[n]}} \right\rfloor \tag{1}$$

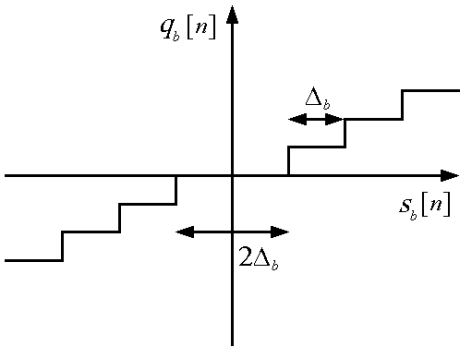

Figure 5: Scalar quantisation: a $2\Delta_b$-wide *dead-zone* is generated around zero. $s_b[n]$ indicates the input signal, while $q_b[n]$ is the output quantised signal.

where $\Delta_b^{p[n]} = 2^{p[n]} \cdot \Delta_b$. The dequantisation equation is as follows:

$$\hat{s}[n] = \begin{cases} \left\lceil \left(q^{p[n]}[n] - r \cdot 2^{M_b - p[n]}\right) \cdot \Delta_b^{p[n]} \right\rceil & q^{p[n]}[n] < 0 \\ 0 & q^{p[n]}[n] = 0 \\ \left\lfloor \left(q^{p[n]}[n] + r \cdot 2^{M_b - p[n]}\right) \cdot \Delta_b^{p[n]} \right\rfloor & q^{p[n]}[n] > 0 \end{cases} \tag{2}$$

where M_b is the maximum number of bits in the coefficients' magnitude and $r \in [0, 1)$ is usually set to $r = 0.5$. Trellis-coded quantisation is much more complicated but usually gives better results and does not suffer from dead-zone issues. We refer the reader to [2] for a detailed description.

3 EBCOT - Tier 1

The core of JPEG2000 is its coding engine, *Embedded Block Coding with Optimised Truncation* (EBCOT for short) [7] [6]. The EBCOT algorithm is conceptually divided into two layers called *Tiers*. Tier 1 is responsible for source modelling and entropy coding, while Tier 2 generates the output stream. Figure 6 shows a schematic representation of EBCOT Tier 1. Wavelet sub-bands are partitioned into small *code-*

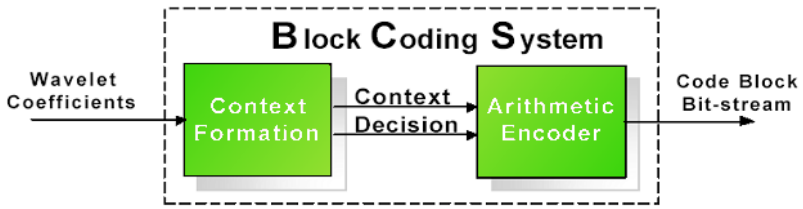

Figure 6: Schematic representation of EBCOT Tier 1.

blocks (to not be confused with *tiles*) which in turn are coded by bit-planes, i.e. coefficient bits of the same order are coded together (see Figure 7). The most significant bits are coded first, then low order bits are coded in descending order. Each bit-plane is coded separately as if it was a bi-level image, except

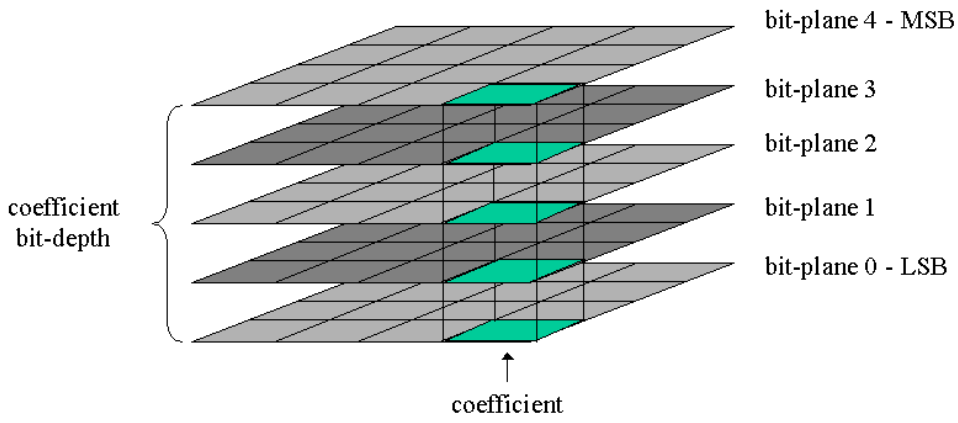

Figure 7: Bit-plane coding: an example of bit-plane coding of wavelet coefficients with bit-depth 5.

from the fact that context formation is guided not only by previously coded bits in the same bit-plane, but also by what has been seen in the higher order bit-planes. Bit-planes are further partitioned into three coding passes. Each coding pass constitutes an atomic code unit, called *chunk*. Codeword chunks are grouped into *quality layers* and can be transmitted in any order, provided that chunks belonging to the same *code-block* are transmitted in their relative order. Chunks constitute valid *truncation points*, i.e. the codeword can be truncated at the end of a chunk without compromising the correct reconstruction of the compressed data (partitioning is consistent with the codeword syntax). Both the encoder and the decoder can truncate the bit-stream in correspondence of a valid truncation point in order to recover the original data up to a target quality. Figure 8 shows an example of codestream partitioning in chunks and layers. The contribution of code-blocks to layers may be extremely variable and some code-blocks might not contribute at all to some layers. This behaviour depends on the information content of each chunk and the contribution of code-blocks to layers is intended to reflect how much "useful" this information is to reach the given target quality. This concept will be made clearer later in this section.

The output stream is made up by a number of *packets* containing chunks of the code-blocks in a given sub-band and belonging to the same layer (i.e. a row in Figure 8). Since the packet header information is highly redundant, the header itself employs wise compression techniques in order to reduce the packet overhead (see Section 4).

3.1 Context Modelling

Context modelling in JPEG2000 is a bit tricky but its conceptual complexity is compensated by its impressive compression performance. It is mostly based on JPEG-LS context modelling [5]. The current bit (*decision* in Figure 6) is entropy coded using an arithmetic coder whose probability estimates are updated using a finite automaton with 46 states. The probability estimate is updated separately for each of 18 different classes using separate *contexts* (see Figure 6).

In order to understand how JPEG2000 context modelling works three concepts must be made clear.

Coding Pass – A complete scan of the current bit-plane grouping together bits with similar statistical properties (see Figure 9). Generally a coding pass does not comprise all the bits in the current bit-plane.

Coding Primitive – A small set of operations involving the update of the state of coefficients being encoded and selection of the most appropriate *context* to code the current bit, given the available statistical information on the neighbouring pixels.

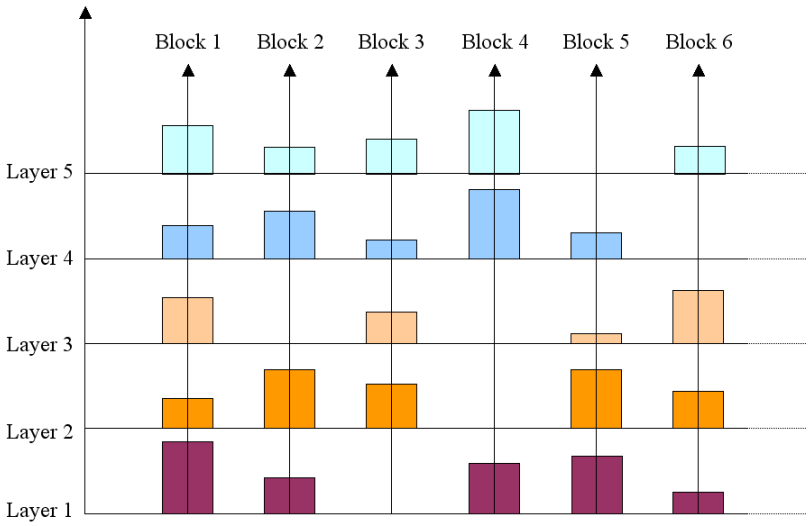

Figure 8: Codestream partitioning: five layers and six code-blocks are shown. Each rectangle represents the chunks of a code-block which have been included in the relative layer. Colours encode layers. The data coded for a block is represented as the chunks lying on the corresponding vertical arrow, while the compressed bitstream is transmitted in a per-layer fashion (i.e. lower levels first).

Context – It refers to all the information needed to keep track of the current state of the finite automaton which guides the statistical update of each coding class. Contexts are determined by coding primitives, neighbourhood statistics, and sub-band frequency properties.

Three different **coding passes** are used (see Figure 9): *Significance Propagation*, *Magnitude Refinement*, and *Normalisation (Cleanup)*. Each pass is intended to estimate symbol probabilities of one out of three different classes of bits. They are executed in the order in which they are presented here. Each bit is coded in one and only one of the three passes. Symbol probabilities are estimated taking into account what has been encoded in the previous bit-planes and the neighbouring bits. *Neighbourhood* is defined as the eight bits immediately adjacent to the current bit in the current bit-plane. A coefficient is said to be *significant* if there is at least one non-null bit among its corresponding bits coded in previous bit-planes. A coefficient that is not significant is called *non-significant*.

Magnitude refinement is used to refine the magnitude of the coefficient corresponding to the bit being coded by coding one more bit (the current one). A bit is coded in this pass if it belongs to an already significant coefficient. *Significance propagation* is used to propagate significance information. When a coefficient becomes significant, the probability that its neighbouring coefficients become in turn significant grows higher and higher as the encoding proceeds. That is why those bits are encoded using different probability estimates. Hence, significance propagation gathers all bits belonging to non-significant coefficients having a *preferred neighbourhood* (i.e. at least one coefficient in the neighbourhood of the current coefficient is significant). Finally, all bits belonging to non-significant coefficients having no significant coefficient in their neighbourhood are coded in the *normalisation* pass. Again this is done because different probability estimates are used since it is highly probable that those coefficients belong to homogeneous image areas. This bit-plane partitioning procedure is depicted in Figure 10. Coding passes, rather than providing an accurate source modelling, give a fine partitioning of bit-planes into three sets. Indeed their purpose is mainly to allow more valid *truncation points* in the compressed codestream (see Section 4).

Coding primitives are employed to obtain a finer modelling. They are responsible for finding the

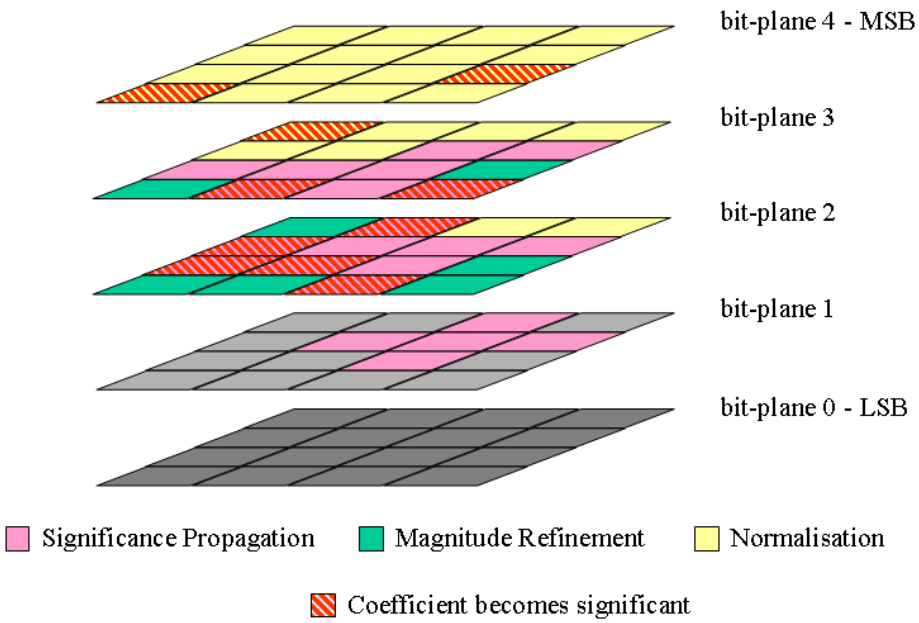

Figure 9: Coding Passes: an example of coding with bit-depth 5.

most appropriate context to encode the current bit. Four primitives are implemented: *Zero Coding (*ZC*)*, *Run-Length Coding (*RLC*)*, *Sign Coding (*SC*)*, and *Magnitude Refinement (*MR*)* (to not be confused with the Magnitude Refinement pass). The ZC primitive is run on non-significant coefficients. It can choose among nine contexts depending on the coefficient's neighbourhood (vertical, horizontal, and diagonal neighbours are differentiated) and the sub-band to which the bit being coded belongs. RLC is used in place of ZC when coding large homogeneous areas. If (exactly) four consecutive non-significant coefficients with no preferred neighbourhoods are found, only one bit is coded for the whole four-bit sequence to indicate if one of those four coefficients is going to become significant in the current bit-plane. If so an additional two-bit index pointing to the first of them is transmitted. This primitive is used to reduce the number of bits to be coded by the entropy coder when long sequences of non-significant coefficients are encountered. A particular context is reserved for the RLC primitive. SC is invoked at most once per coefficient as soon as it becomes significant, in order to encode its sign. The MR primitive is used to refine the magnitude of a significant coefficient, for which the sign has already been coded in a previous bit-plane. The correct context is chosen taking into account the coefficient's neighbourhood. Moreover, different contexts are chosen if MR is being executed for the first time onto the current coefficient or not.

Each coding pass employs these primitives to encode the bits in the current bit-plane. The Significance Propagation pass executes a ZC for each bit being coded and a SC for each bit which becomes significant in the current bit-plane. The Normalisation pass has the same behaviour except for the opportunity to apply RLC in place of ZC where possible. Finally, the Magnitude Refinement pass can call only the MR primitive.

The symbol to be encoded and the relative context are passed to the arithmetic coder, which "loads" the current state relative to the given context and uses it to encode the given symbol. The state is then updated in order to refine probability estimates for the current context.

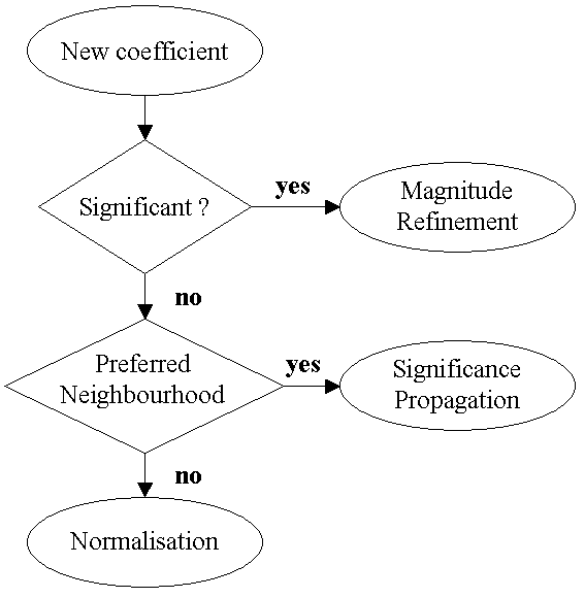

Figure 10: Bit-plane partitioning flow chart.

3.2 Arithmetic Coding: MQ-Coder

The arithmetic coding engine in JPEG2000 (*MQ-Coder*) was developed for the JBIG standard for bi-level image compression [1]. It is slightly different from standard arithmetic coders. Hence it needs a short explanation. **MQ-Coder** is an entropy coder which, rather than coding the binary value of the input symbol (the current bit), encodes a binary information which indicates if the symbol being coded is what we expected (*Most Probable Symbol*, or MPS) or not (*Less Probable Symbol*, or LPS). In particular, two peculiar features of the MQ-Coder will be briefly discussed: *renormalisation*, and *conditional exchange*.

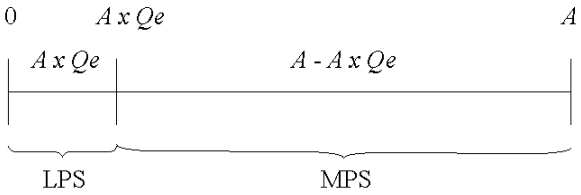

Figure 11: Interval subdivision in the MQ-Coder.

Recall that at each step of the arithmetic coding of a sequence the current subsequence is represented as an interval in $[0, 1]$. Figure 11 shows how the current interval is partitioned in MQ-Coder. Following the notation in [2], C and A refer to the base and length of the current interval, while Qe is the estimated probability of the LPS. The classical update equations can thus be rewritten as follows.

MPS coding:

$$C = C + A \times Qe \quad (3)$$
$$A = A - A \times Qe \quad (4)$$

LPS coding:

$$C \quad : \quad \text{unchanged} \tag{5}$$
$$A \quad = \quad A \times Qe \tag{6}$$

In order to avoid complex calculations, a trick is used to simplify the above equations. The length of the current interval is bounded to lie in $\Re = [0.75, 1.5)$. When A falls below the minimum it is left shifted until $A \in \Re$ (*renormalisation*). With this simple adjustment $A \approx 1$ is always true. The above equations can thus be approximated setting $A = 1$ in the multiplications on the right side of equations (3) (4) and (6).

MPS coding:

$$C \quad = \quad C + Qe \tag{7}$$
$$A \quad = \quad A - Qe \tag{8}$$

LPS coding:

$$C \quad : \quad \text{unchanged} \tag{9}$$
$$A \quad = \quad Qe \tag{10}$$

The computation of Equations 7, 8, and 10 is much more efficient than the computation of the relative non-approximate equations because they do not involve multiplications, which can be burdensome even in hardware implementations. Moreover, fixed point arithmetic can also be used to carry out computations. This implementation does not lead to any coding efficiency loss, since it does not reduce the coding space (i.e. the length of the two subintervals generated is still equal to the length of the current interval before the subinterval splitting). However, when Qe is close to 0.5 and A is small, it can happen that the MPS subinterval is smaller than the LPS subinterval. As an example, consider the situation: $A = 0.8$, $Qe = 0.43$. From Equations 8 and 10 it follows that $A_{MPS} = 0.37$ and $A_{LPS} = 0.43$, i.e. $A_{MPS} < A_{LPS}$. In order to avoid such an inversion, in those cases the interval assignment is inverted as shown in Figure 12

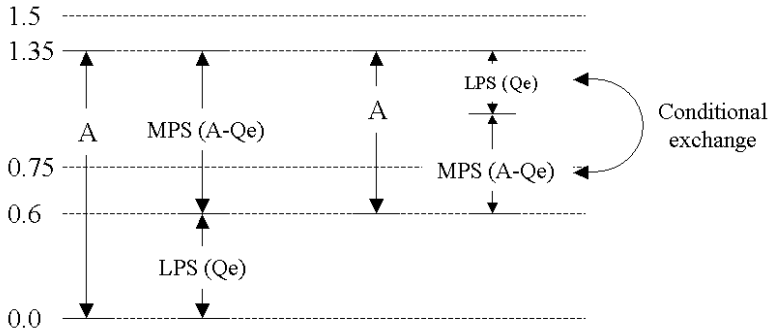

Figure 12: Conditional exchange.

(*conditional exchange*).

The algorithms for coding MPS and LPS are sketched here.

MPS coding

```
A = A - Qe                  /* Calculate MPS subinterval        */
if A < min                  /* if renormalisation is needed     */
    if A < Qe               /* if interval size is inverted     */
        A = Qe              /* Set interval to LPS subinterval  */
    else                    /* otherwise                        */
        C = C + Qe          /* Point to MPS interval base       */
    end
    renormalise A and C     /* renormalise                      */
else                        /* if no renormalisation is needed  */
    C = C + Qe              /* Point to MPS interval base       */
end
```

LPS coding

```
A = A - Qe                  /* Calculate MPS subinterval        */
if A < Qe                   /* if interval size is inverted     */
    C = C + Qe              /* Point to MPS interval base       */
else                        /* otherwise                        */
    A = Qe                  /* Set interval to LPS subinterval  */
end
renormalise A and C         /* renormalisation always needed    */
```

The probability estimation process is based on a sort of approximated counting of MPSs bounded by renormalisations of the register A. After each renormalisation the counting is reset and a new estimate for Qe is obtained from the finite automaton. If the value of Qe is too high MPS renormalisation is more probable. This in turn causes Qe to be set to a smaller value. On the other hand, if Qe is too small LPS renormalisation probability grows and as a consequence Qe tends to increase. If MPS identity is wrong Qe will increase up to approximately 0.5 causing a switch of the MPS identity.

The pseudo-code for renormalisation is given below. Here, CX is the current context as determined by the source modeler, and I is the current state index for CX. $NMPS$ and $NLPS$ indicate respectively the next state if a MPS or a LPS is encountered.

MPS renormalisation

```
I = Index(CX)               /* Current index for context CX     */
I = NMPS(I)                 /* New index for context CX         */
Index(CX) = I               /* Save this index at context CX    */
Qe(CX) = Qe_value(I)        /* New probability estimate for CX  */
```

LPS renormalisation

```
I = Index(CX)               /* Current index for context CX     */
if Switch(I) = 1            /* If a conditional exchange occurs */
    MPS(CX) = 1 - MPS(CX)   /* Exchange MPS sense               */
end                         /* (1 -> 0 or 0 -> 1)               */
I = NLPS(I)                 /* New index for context CX         */
Index(CX) = I               /* Save this index at context CX    */
Qe(CX) = Qe_value(I)        /* New probability estimate for CX  */
```

4 EBCOT - Tier 2

The JPEG2000 bitstream is scalable with respect to resolution (thanks to wavelet decomposition) and quality and provides reasonable random access to image blocks in the compressed domain. Quality scalability is obtained by means of bit-plane and coding pass partitioning. All the coded bits for a coding pass constitute an atomic unit and are transmitted as a single block. However, compressed chunks need to be transmitted only up to a minimum number to reach a user-defined quality. It is a sort of wise uneven post-compression quantisation which usually gives better results than standard quantisation methods, in terms of quality of the reconstructed image.

In order to enable the user to break the stream up to the desired resolution and/or quality, or to visualise a particular area of the image, the compressed bitstream structure must be carefully organised. This task is carried out by the Tier 2 of the EBCOT algorithm. The stream is partitioned into *packets* containing header information in addition to the stream itself. In particular, packet headers contain information useful to retrieve codeword chunks in the output stream, e.g. whether any block of a given wavelet sub-band is included in the packet or not, codeword length, and so on.

4.1 Optimal Truncation Points

The output of Tier 1 is a series of codewords (one for each code-block) with an indication on valid truncation points. This side information is used to find *optimal truncation points* in a rate/distortion sense, given a target bit-rate. The set of truncation points which together give the best tradeoff between codeword length and the corresponding amount of distortion with respect to the original image (i.e. quality), is selected for each code-block included in the packet.

Finding optimal truncation points involves the minimisation of the rate/distortion ratio. Let us indicate with $R_i^j, j = 1, 2, ...$ the sequence of valid truncation points of the codeword corresponding to the i-th code-block, and with $D_i^j, j = 1, 2, ...$ the associated distortions. If we indicate with n_i the optimal truncation point for the i-th codeword, the total bitstream length can be written as $R = \sum_i R_i^{n_i}$ and $D = \sum_i D_i^{n_i}$ is the global distortion[1]. Optimal truncation points n_i are found in correspondence of codeword lengths minimising the global distortion. Given a target bit-rate R_{max}, we want to minimise the global distortion D subject to the constraint $R = R_{max}$. A well-known method to do the job is that of *Lagrange multipliers* for constrained optimisation problems. In our setup we have to find a value λ which minimises the function $D + \lambda R$. Anyway, since there are finite many truncation points we cannot guarantee that $R = R_{max}$. This condition is thus relaxed and turn into $R \leq R_{max}$. Hence, we search for the the codeword length R closest to R_{max} which minimises our cost function. Since Lagrange multipliers are outside the scope of this tutorial we refer the reader to [3].

4.2 Packet Header

Given a target bit-rate, optimal truncation points define the number of bits that appear in a given layer for each code-block. Depending on its information content, a given code-block may be completely included in a layer or, on the other side, may not appear at all in that layer. Hence, the packet header should encode inclusion information. Other important information contained in the packet header is the codeword length for each code-block included and the number of null most-significant bit-planes, which is useful to avoid wasting time and bits for coding long sequences of zeroes. All of this three pieces of information exhibit high redundancy both inside the same layer and between different layers (see [7] and [6]). A particular tree structure, *Tag Tree*, is adopted to exploit this redundancy in order to obtain a more compact representation of packet headers. In the following, tag trees will be introduced and it will be shown how they are effectively employed to encode packet header information.

[1] Notice that here is assumed that the measure of distortion is additive.

4.2.1 Tag Trees

A tag tree is a particular tree structure devised for the efficient coding of bi-dimensional highly redundant matrices. The information to be coded is associated to the leaves. Their structure resembles that of quad-trees. However they have two key differences with respect to conventional quad-trees: tag trees do not need to be binary-valued, and the information in a tag tree may be encoded in any order, provided that the decoder follows the same order. Figure 13 shows an example of a tag tree. Let $q_1[m, n]$ denote an

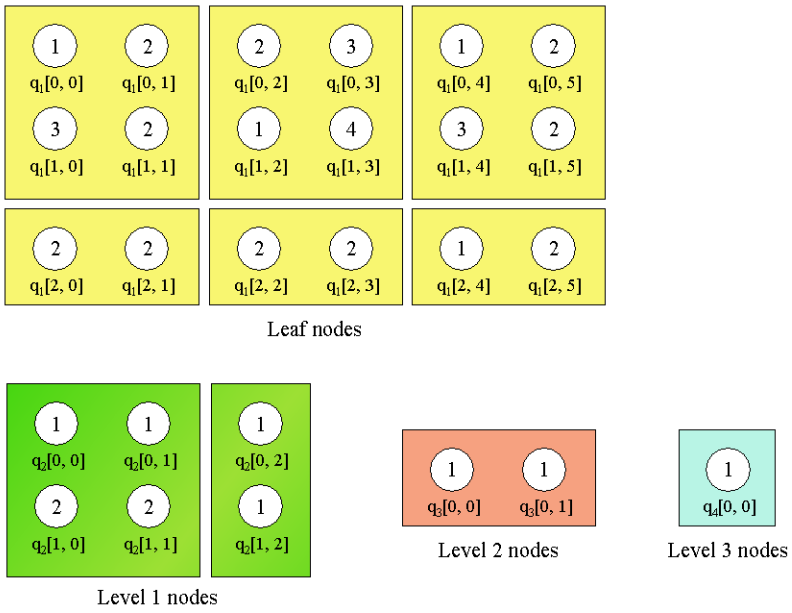

Figure 13: An example of Tag Tree.

array of quantities (non-negative integers) which we would like to represent via the tag tree. We associate these quantities with the leaf nodes of the tree. Nodes of next levels are denoted as $q_2[m, n]$, $q_3[m, n]$, and so on. Each internal node is associated with a 2×2 block of nodes in the lower level, and contains the minimum among the values of its four children. The root contains the minimum of the whole structure.

Tag trees efficiently encode the information $q_1[\overline{m}, \overline{n}] \geq t$, where $q_1[\overline{m}, \overline{n}]$ is a value in the input matrix and t is a threshold value. The algorithm for building tag trees is usually expressed as a procedure $T(m, n, t)$ which encodes the minimum-length bit string to indicate if $q_1[\overline{m}, \overline{n}] \geq t'$, $\forall t' : 1 \leq t' \leq t$, given the information which has been encoded in previous calls. Namely, the algorithm avoids coding again the information coded in previous calls for intermediate nodes. This information is maintained by means of a state variable $t_k[m, n]$, which assures that sufficient information has been coded to tell if $q_1[\overline{m}, \overline{n}] \geq t'$, $t' = 1, 2 \ldots t_k[m, n]$. It is initialised to zero since no information has been coded at the beginning. The coding algorithm is outlined in Algorithm 1. As an example consider the tag tree in Figure 13 and suppose we want to encode sufficient information for a threshold $t = 1$, using Algorithm 1. We start with $q_1[0, 0]$ and proceed in raster order (i.e. from left to right, then to the row below). As the check $t \leq t_4[0, 0]$ fails in step (5) (i.e. sufficient information has not been coded yet for the root) we are driven to step (6), where $t_4[0, 0]$ is incremented and a **0** bit is emitted. Then we are lead again to step (5). This time the the test succeeds so t_{min} is updated in step (5.2) and we go on to the lower level of the tree. For each subsequent level of the tag tree nothing is coded since the information gathered is already sufficient. Finally, we end up with a single **0** bit to code if $q_1[0, 0] \geq 1$. No further information is needed to code if $q_1[m, n] \geq 1, \forall m, n$ since we know that the root contains the minimum of the whole tree, and this value is not less than the given threshold. Figure 14 shows the bits coded for the tag tree increasing

Algorithm 1 - Tag Tree encoding procedure ($T[m, n, t]$).

(1) $k = K$ /* Start at the root node */

(2) $t_{min} = 0$ /* Used to propagate knowledge to descendants */

(3) $m_k = \lfloor \frac{m}{2^{k-1}} \rfloor$, $n_k = \lfloor \frac{n}{2^{k-1}} \rfloor$ /* $[m_k, n_k]$ is the location of the leaf ancestor node in level k */

(4) if $t_k[m, n] < t_{min}$ set $t_k[m, n] = t_{min}$ /* Update the state variable */

(5) if $t \leq t_k[m, n]$

(5.1) if $k = 1$ we are done /* We have reached the leaf. Sufficient information has been coded */

(5.2) otherwise

- set $t_{min} = min\{t_k[m_k, n_k], q_k[m_k, n_k]\}$ /* update t_{min} to propagate knowledge without sending new bits */
- decrease k and go to step (3)

(6) otherwise (i.e. if $t > t_k[m, n]$) /* send enough information to increase $t_k[m_k, n_k]$ */

- if $q_k[m_k, n_k] > t_k[m_k, n_k]$ emit a **0** bit
- else ($q_k[m_k, n_k] = t_k[m_k, n_k]$) emit a **1** bit
- increase $t_k[m_k, n_k]$ and go to step (5)

Algorithm 2 - Tag Tree decoding procedure.

(1) $k = K$ /* Start at the root node */

(2) $t_{min} = 0$ /* Used to propagate knowledge to descendants */

(3) $m_k = \lfloor \frac{m}{2^{k-1}} \rfloor$, $n_k = \lfloor \frac{n}{2^{k-1}} \rfloor$ /* $[m_k, n_k]$ is the location of the leaf ancestor node in level k */

(4) if $q_k[m, n] < t_{min}$ set $q_k[m, n] = t_{min}$ /* Update the state variable */

(5) if $s_k[m_k, n_k]$ is **true** /* Sufficient information has been coded for the current level */

- set $t_{min} = q_k[m_k, n_k]$ /* Update t_{min} to propagate knowledge */
- decrease k and go to step (3) /* and proceed to the next level */

(6) otherwise ($s_k[m_k, n_k]$ is false)

- if $t \leq q_k[m_k, n_k]$
  - if $k = 1$ we are done /* We have reached the leaf. Sufficient information has been coded */
  - otherwise
    - set $t_{min} = q_k[m_k, n_k]$ /* update t_{min} to propagate knowledge without sending new bits */
    - decrease k and go to step (3)
- otherwise (i.e. if $t > q_k[m_k, n_k]$) /* read enough information to update the value of $q_k[m_k, n_k]$ */
  - let b be the next input symbol
  - if b = **0** /* A **0** means that we must */
    - increase $q_k[m_k, n_k]$ /* update our knowledge of $q_k[m_k, n_k]$ */
  - else set $s_k[m_k, n_k]$ to **true** /* **1** means that no further information is needed to encode this node from now on */
  - go to step (5)

the threshold value to two. Notice that the sequence of thresholds used for coding must be known by the decoder. In fact, the decoder can reconstruct the original data only if the sequence is decoded in the same order as it was encoded. Hence, the of values of Figure 14 are valid only if the encoding procedure has previously been run with threshold $t = 1$.

1111	**0**	**10**	**0**	**111**	**0**
0	**0**	**1**	**0**	**0**	**0**
0	**-**	**0**	**-**	**11**	**0**

Figure 14: Bits coded for the tag tree of Figure 13. The threshold value was set to two. The encoding procedure had been previously run with threshold $t = 1$.

The decoding algorithm follows the same general idea. Given the dimensions of the original matrix and a threshold, it can extract the original data by propagating in lower levels the information already known for higher levels. A pseudocode for the decoder is given in Algorithm 2. An additional boolean state variable, $s_k[m_k, n_k]$, is used in place of the state variable $t_k[m, n]$ of the encoding procedure in order to indicate when sufficient information has been decoded to determine the value of $q_k[m_k, n_k]$. When $s_k[m_k, n_k]$ becomes true no more bits will be decoded for $q_k[m_k, n_k]$.

4.2.2 Coding Packet Header Information

Tag trees find a natural application in encoding JPEG2000 packet header information. In particular, they are used to encode two fields out of four. The values contained in packet header for each code-block are:

- inclusion information,
- maximum number of bit in the mantissa of code-block coefficients (i.e. the number of null most-significant bit-planes).
- number of new coding passes included (determined as a function of optimal truncation points),
- codeword length, and

Inclusion information is encoded using a tag tree to indicate the first layer in which a given code-block is included, then a single bit is used for further layers to indicate if the code-block is included or not. Tag trees efficiently encode this kind of information since neighbouring code-blocks tend to appear in layers close to each other. The same observation holds true for the number of null most-significant bit-planes. The number of coding passes included uses a different coding scheme, similar to Huffman coding in that low values (highly probable) get less bits than high values (less probable). Finally, codeword length is encoded as a two-part code in which the first part indicates the length of the second one. This in turn signals the codeword length. In particular, the number of bits in the second part is computed as $n_{bits} = \lfloor \log_2(p_i) \rfloor$, where p_i is the number of new coding passes included in the current packet. This tricky scheme is intended to model the fact that the codeword length is somehow proportional to the number of coding passes included.

The packet-oriented structure of the JPEG2000 compressed stream allows high scalability in the compressed domain. The decoding progression order is determined by suitably placing packets in the output stream.

5 Summary Of The JPEG2000 Algorithm

In order to have a global view of the JPEG2000 algorithm, we summarise here the operations carried out by the encoder (refer to Figure 2). The input image undergoes a preliminary colour transform

into a separable colour space. Colour planes (apart from luminance plane) are possibly sub-sampled (lossy compression). Then, the image is partitioned into large blocks called *tiles*. Wavelet decomposition is applied to tiles separately. Small blocks of (possibly quantised) wavelet coefficients are processed separately using the EBCOT algorithm (Figure 6). In Tier 1 source modelling and entropy compression take place. Source modelling is carried out at bit-plane level and is driven by statistical information on coefficients' neighbourhoods. Its output is a context which helps the arithmetic coder to correctly update symbol probabilities. After each block in a tile has been encoded, Tier 2 places it conveniently in the output compressed bit-stream. Header information is compressed as well by means of a particular tree structure called *tag tree*. A wise quantisation can take place in this phase both on the encoder and decoder side by truncating the bitstream up to an optimal truncation point. The decoder performs the same steps in inverted order.

6 Indexed Image Coding

When compressing images showing a low number of distinct colours, the *indexed image* compression scheme is usually adopted. Here the observation is that the number of colours in an image can never exceed the number of pixels, and is usually much lower. Thus, we might assign an index to each colour and store this index in place of the relative colour. A colour table (*palette*) stores the RGB triplets corresponding to the image indexes. If the number of colours is low enough (i.e. the representation of indexes is short), the overhead paid to store the palette is compensated by the gain in the representation of pixels. The matrix of indexes can be compressed using any generic lossless compression method. This scheme is very effective when the number of colours is low, while its efficiency degrades as the number of colours becomes larger. This of course depends on the dimension of the colour table. Indexed image compression methods behave well with images with a few distinct colours, such as line drawings, black and white images and small text that is only a few pixels high. On the contrary, the performance of such algorithms coding continuous-tone images, like photos, may be very poor.

JPEG2000 can be used to compress indexed images as well as continuous-tone ones. Experimental data show that JPEG2000 can yield better results than GIF or PNG if a suitable colour indexing is chosen. The idea is to *smooth* index transitions in the index image. Unfortunately, given an image I with M distinct colours, the number of possible colour indexings is $M!$. Obtaining an optimal re-indexing scheme is suspected to be an intrinsically hard problem (NP-complete). Thus only heuristic methods to pick an optimal ordering out of the many possible ones are known. In this section we will review two of them [8][4].

Both the approaches we will present collect colour adjacency statistics using a matrix C called *co-occurrence matrix*. Each entry $C(i, j)$ reports the number of times the pair of neighbouring colour symbols (c_i, c_j) has been observed over the index image. C is used to weight index pairs in order to guide the re-indexing process. The co-occurrence matrix might be re-arranged in order to profile the particular predictive scheme adopted by the compression engine. Both these algorithms build a chain of colour indexes and then derive a re-indexing simply by assigning consecutive indexes to consecutive colours.

The first method we present was proposed by Zeng et alii [8] during the JPEG2000 standardisation process. It is an iterative greedy algorithm based on the maximisation of a potential function defined as $w_{N,j} = \log_2\left(1 + \frac{1}{d_{N,j}}\right)$, where $d_{N,j}$ is the distance of the j-th index from the end of the current index chain. N indicates the length of the chain. At each step a colour index is added to one of the endpoints of the current chain by maximising the function $w_{N,j}$. The chain is initialised by adding the colour c_i maximising $\sum_{j \neq i} C(i, j)$.

In the approach introduced by Battiato at alii [4] the re-indexing problem is mapped onto an optimisation problem for a Hamiltonian path in a weighted graph. Given the co-occurrence matrix C computed

as above, a graph $G = (V, E)$ is defined as follows:

$$\begin{array}{lcl} V & = & \{c_1, c_2, \ldots c_M\} \\ E & = & V \times V \\ w & : & E \to \mathcal{N} \text{ where } w \text{ is defined as} \\ & & w(i,j) = \begin{cases} C(i,j) + C(j,i) & i \neq j \\ 0 & i = j \end{cases} \end{array}$$

A re-indexing is found by searching for the heaviest Hamiltonian path in G, i.e. a solution for the *Travelling Salesman Problem* (TSP), which is a well known NP-Complete problem. The authors suggest a very efficient solution to solve the TSP in an approximate fashion.

Experimental results show that the two algorithms presented are comparable in terms of compression ratios, computational efficiency, and memory consumption.

References

[1] ISO/IEC 11544:1993 and ITU-T Recommendation T.82. Information technology – Coded representation of picture and audio information – progressive bi-level image compression, 1993.

[2] ISO/IEC 15444-1:2000. Information technology – JPEG 2000 image coding system – Part 1: Core coding system, 2000.

[3] G. Arfken. *Mathematical Methods for Physicists*. Academic Press, Boston, 3 edition, 1985.

[4] S. Battiato, G. Gallo, G. Impoco, and F. Stanco. A color reindexing algorithm for lossless compression of digital images, 2001.

[5] W.B. Pennebaker and J.L. Mitchell. The jpeg still image data compression standard. In *Van Nostrand-Reinhold*, 1993.

[6] D. Taubman, E. Ordentlich, M. Weinberger, and G. Seroussi. Embedded block coding in JPEG 2000. In *ICIP00*, page TA02.02, 2000.

[7] D. S. Taubman. High performance scalable image compression with EBCOT. In *IEEE Trans. Image Proc.*, volume 9, July 2000.

[8] W. Zeng, J. Li, and S. Lei. An efficient color re-indexing scheme for palette-based compression. In *ICIP00*, pages 476–479, 2000.