Overview of JPEG2000

Christos Chrysafis, David Taubman and Alex Drukarev
Hewlett Packard Laboratories
1501 Page Mill Road, Bldg 3U-3, Palo Alto California 94304-1126

Introduction

In this paper we will describe the coding algorithm, which forms the basis for the verification model 3.0A, parts of this document were taken from {D. Taubman 1998}. As a first step to image compression a bi-orthogonal wavelet transform is applied on an input image. Each subband of wavelet coefficients is divided up into blocks of samples, measuring say 32 or 64 samples in each direction, except at image boundaries where some blocks may have smaller dimensions. Every block is coded completely independently using exactly the same algorithm in every subband. Let $\{B_i\}_{i=1,2,\ldots}$ denote the set of all blocks, which represent the image. For each block, B_i, a separate bit-stream is generated without utilizing any information from any of the other blocks. Moreover, the bit-stream has the property that it can be truncated to a variety of discrete lengths, $R_i^1, R_i^2, R_i^3, \ldots$, and the distortion incurred when reconstructing from each of these truncated subsets is estimated and denoted by $D_i^1, D_i^2, D_i^3, \ldots$. Once the entire image has been compressed, a post-processing operation passes over all the compressed blocks and determines the extent to which each block's embedded bit-stream should be truncated in order to achieve a particular target bit-rate. It then composes the final compressed bit-stream by stringing the blocks together in any pre-defined order, together with information to identify the number of bytes, R_i^n, which are used to represent each block. This basic idea is extended to the formation of bit-streams in which the representation of each block is distributed across multiple (perhaps as many as 50 or more) so-called "layers".

The coder is essentially a bit-plane coder, using the same Layered Zero Coding (LZC) techniques which have been employed in a number of embedded Wavelet coders and were originally proposed by Taubman and ZakhorTaubman 1994 #10}" {D. Taubman 1994} with the addition of fractional bit-plane coding and a simple embedded quad-tree algorithm.

Rate-Distortion Optimization

The rate-distortion optimization algorithm itself need not be defined in any compression standard, since the decoder need not be aware of its existence. In fact, the layers defined by the bit-stream syntax may be used to convey information about individual code blocks in any sequence at all and in arbitrary increments. Nevertheless, the fact that the coding algorithm and bit-stream syntax enable the generation of rate-distortion optimized bit-streams is an important element in the justification of the new coding algorithm. The number of code bytes included into the bit-stream is given by:

$$R = \sum_i R_i^{n_i}$$

and we wish to find the set of n_i (n_i is the truncation point for block B_i) values which minimizes the distortion D subject to the constraint $R \le R_{\max}$. The solution to this constrained optimization problem by the method of Lagrange multipliers is well known. Specifically, the problem is equivalent to minimizing

$$\sum_i \left(R_i^{n_i} + \lambda D_i^{n_i} \right)$$

where the value of λ must be adjusted until the rate yielded by the truncation points which minimize the above equation satisfies $R = R_{\max}$. In practice, we find that it is usually possible to find values of λ, such that R is very close to $R_{\max}$ (almost always within 100 bytes), so that any residual sub-optimality is of little concern. For each block, B_i, we must find the truncation point, n_i, which minimizes $\left(R_i^{n_i} + \lambda D_i^{n_i} \right)$ this is trivial as can be seen in Taubman 1998)" (D. Taubman 1998). Also the search for the optimal λ is extremely fast and occupies a negligible proportion of the overall image compression time.

Block Packing

Each subband is partitioned into a set of blocks. All blocks have the same size, say 32×32 or 64×64, with the possible exception of blocks which lie on the image boundaries (i.e. the boundaries of the array of samples from the relevant subband which represent the image). Apart from this exception, the block size is also identical for all subbands, so those blocks in lower resolution subbands span a larger region in the original image.

Different Subbands and Block Transposition

Let the unquantized sample values associated with any given code block be denoted by $s[m,n]$, where m is the row index and n is the column index. At any given resolution level we have up to four different types of subbands, which we denote LL, LH, HL and HH. The LL band appears only in the lowest resolution level and contains DC sample values. The LH band corresponds to horizontal low-pass filtering and vertical high-pass filtering. The HL band corresponds to vertical low-pass filtering and horizontal high-pass filtering. The HH band corresponds to high-pass filtering in both the horizontal and vertical directions. All code blocks from HL sub-bands are transposed before applying the block-based coder, so that both HL and LH subbands can be encoded with the same algorithm.

Bit-Plane Coding and Dead-Zone Quantization

Let the quantization step size for the relevant subband be denoted by δ. Let $\chi[m,n]$ denote the sign of $s[m,n]$ (0 if +ve and 1 if -ve) and let $v[m,n]$ denote the quantized magnitude, i.e., $v[m,n] = [|s\{m,n\}|/\delta]$. The representation of $s[m,n]$ in terms of $\chi[m,n]$ and $v[m,n]$ is equivalent to dead-zone quantization in which the central dead-zone is twice as large as the step size, δ. Let $v_p[m,n]$ denote the p'th bit in the P bit integer representation of $v[m,n]$, where p runs from 0 to P-1 and p = 0 corresponds to the least significant bit. The idea behind bit-plane coding is to send first the most significant bits, $v_{P-1}[m,n]$, for all samples in the code block, then the next most significant bits and so on until all bit planes have been sent. In order to efficiently encode $v_p[m,n]$, it is important to exploit previously encoded information about the current sample, $s[m,n]$(from previous bit-planes), and neighboring samples. We do this primarily by means of a binary-valued state variable, $\sigma[m,n]$, which is initialized to 0, but transitions to 1 when the relevant sample's first non-zero bit-plane value, $v_p[m,n] \neq 0$, is encoded. We refer to the state, $\sigma[m,n]$, as the "significance" of sample $s[m,n]$.

Embedded Quad-Tree Front End

Each block, B_i, is partitioned into a number of sub-blocks, B_i^j, whose maximum dimension, is typically set to 16. All sub-blocks have the same dimension, except those at the lower and right hand boundaries of the code block, which might have smaller dimensions. Thus, if the basic block size is 64×64, all blocks will be partitioned into a 4×4 array of 16×16 sub-blocks, except those blocks, which lie on the lower and right-hand image boundaries.

The main reason for introducing the notion of sub-blocks is to avoid running the zero coding algorithm on large regions of zeros.

Types of Coding Operations

Four different coding operations are used to encode single sample, $s[m,n]$, for some bit-plane, p. If the sample is non yet significant, i.e. $\sigma[m,n] = 0$, a combination of the "Zero Coding" (ZC) and "Run-Length Coding" (RLC) operations is used to encode whether or not the symbol is significant in the current bit-plane, i.e. whether or not $v_p[m,n] = 1$. If so, the "Sign Coding" (SC) operation must also be invoked to send the sign, $\chi[m,n]$. If the sample is already significant, i.e. $\sigma[m,n] = 1$, the "Magnitude Refinement" operation is used to encode the new bit-position $v_p[m,n]$.

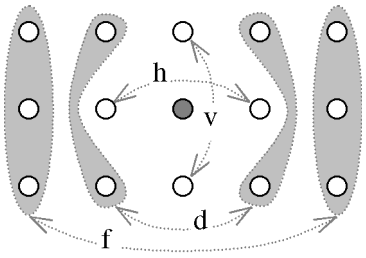

Figure 1. Neighbors involved in context formation for Zero Coding

Zero Coding (ZC)

Upon entry, $\sigma[m,n] = 0$. The binary symbol, which must be coded, is $v_p[m,n]$. We use one of 10 different context states to code the value of this symbol, depending upon the significance of the neighbors in Figure 1. Note that neighbors which lie beyond the boundary of the code block are understood as being insignificant when forming contexts so as to avoid dependence between code blocks. No such assumptions are imposed on neighbors that lie beyond the current sub-block.

Run-Length Coding (RLC)

The RLC operation is used in conjunction with the ZC operation described above, in order to reduce the average number of binary symbols which must be encoded using the arithmetic coding engine. The operation is invoked in place of the ZC operation if and only if the following conditions hold:

1. Four consecutive samples must have a zero state variable, i.e. $\sigma[m,n] = 0$.
2. All four samples must have identically zero neighborhoods. That is, the ZC neighborhood context variables, h, v, d and f must be 0 for each of the four consecutive samples.
3. All four samples must be horizontally adjacent and reside within the same sub-block.
4. The group of four samples must be aligned on a two-sample boundary. That is, legal groups of four samples must start at the 1st, 3rd, 5th ... sample of the relevant sub-block line.

When a group of four symbols satisfying the above conditions is encountered, a single symbol is encoded to identify whether any sample in the group is becoming significant in the current bit-plane (1 if so; 0 otherwise). This symbol is coded using a single arithmetic coding context state. If any of the four symbols becomes significant, i.e. $v_p[m,n] = 1$, the zero-based index of the first significant sample in the group is sent as a two-bit quantity. The most significant bit is sent first, followed by the least significant bit.

Sign Coding (SC)

When a coefficient $v[m,n]$ is first found to be significant the state variable is toggled to $\sigma[m,n] = 1$ and the binary-valued sign bit, $\chi[m,n]$, is encoded with respect to one of 5 different context states, depending upon the sign and significance of the immediate vertical and horizontal neighbors.

Magnitude Refinement (MR)

Upon entry, $\sigma[m,n] = 1$. The binary symbol, which must be coded, is $v_p[m,n]$. We use a total of three magnitude refinement contexts, depending upon the sample's neighborhood, and whether or not this is the first bit-plane in which magnitude refinement is being applied. At this point it is convenient to define a second state variable, $\breve{\sigma}[m,n]$ which identifies whether or not the magnitude refinement operation has already

been applied to the sample in a previous bit-plane. This new state variable is initialized to zero and is set to 1 at the last step before the magnitude refinement operation completes. The magnitude refinement context depends upon the value of $\breve{\sigma}[m,n]$ and also upon whether or not any of the immediate horizontal and vertical neighbors are significant.

Fractional Bit-Planes and Scanning Order

For each bit-plane, the coding proceeds in four distinct passes. Let p_p^1, p_p^2, p_p^3 and p_p^4 denote the information encoded in each of these four different types of coding passes, for bit-plane p. Also, let Q_p denote the quad-tree information encoded in bit-plane p, where p runs from 0 to P-1.

Figure 2 uses this notation to illustrate the composition and organization of the final embedded bit-stream generated for any given code block.

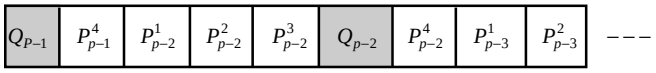

Figure 2. Composition and organization of an embedded block bit-stream.

Notice that the quad-tree code, which identifies the significant blocks for bit-plane p, appears immediately before the final coding pass for that bit-plane. But after the first three coding passes for bit-plane p. Within any given coding pass, all sub-blocks, which have not yet been found to be significant, are ignored. This means that coding pass p_p^4 skips over all sub-blocks which are insignificant in bit-plane p, whereas coding passes p_p^1, p_p^2 and p_p^3 skip over all sub-blocks which are insignificant in bit-plane p+1, even though some of those sub-blocks might contain significant samples in bit-plane p. This is because the quad-tree information for bit-plane p does not appear until after coding passes p_p^1, p_p^2 and p_p^3. It can and indeed often does happen that one or more leading coding passes are empty, since no sub-blocks are significant at all. These empty coding passes consume no bits, since the bit-plane, $p_i^{\max}$ in which each code block, B_i, first becomes significant is identified via a separate mechanism in the bit-stream syntax. We allow the bit-stream to be truncated at the end of each coding pass, but never between the quad-tree coding phase and the following coding pass. To assist in the definition and implementation of these coding passes, it is helpful to introduce a final state variable, $\eta[m,n]$, which is initialized to 0 at the end of coding pass p_p^4 and is set to 1 whenever the relevant sample is visited in some coding pass.

"Forward Significance Propagation pass",

During this pass we visit the sub-blocks in forward scan-line order (i.e. $B_i^1, B_i^2, B_i^3, \ldots$), skipping over all sub-blocks which have not yet been found significant (as identified by the quad-tree code for bit-plane p+1). Within each sub-block we visit the samples in scan-line order, skipping over all samples except those which are insignificant and have a so-called "preferred neighbor-hood". For blocks from the LL, LH and HL subbands, a sample, $s[m,n]$, is said to have a preferred neighborhood if at least one of its horizontal neighbors is significant. For blocks from the HH subband, a sample, $s[m,n]$, is said to have a preferred neighborhood if at least one of its diagonal neighbors is significant. Note that any neighbor which lies beyond the code block boundaries is interpreted as insignificant. To each sample, which is currently insignificant, i.e. $\sigma[m,n] = 0$, and has the relevant preferred neighborhood, we apply the ZC operation, to code whether or not it continues to be insignificant with respect to the current bit-plane. If it becomes significant, i.e. $v_p[m,n]$, the SC operation is also invoked, to code the sign bit.

"Backward Significance Propagation pass", p_p^2

This coding pass is identical to p_p^1 except in three key respects: 1) we pass through the sub-blocks in reverse scan-line order and through the samples of each significant sub-block in reverse scan-line order as well, starting from the bottom right corner and ending at the top left; 2) we skip over any samples, $s[m,n]$, for which the "visited" state variable, $\eta[m,n]$, is set to 1; and 3) instead of a "preferred neighborhood", we consider any sample for which at least one of the immediate eight neighbors is significant.

"Magnitude Refinement Pass", p_p^3

During this pass we visit the sub-blocks in forward scan-line order, as in p_p^1 skipping over all sub-blocks which have not yet been found to be significant (as identified by the quad-tree code for bit-plane p+1). Within each sub-block we visit samples in scan-line order, starting from the top left corner of the sub-block and finishing at the bottom right corner, skipping over all samples, except those which are significant, i.e. $\sigma[m,n] = 1$, and for which no information has yet been coded in the current bit-plane, i.e. $\eta[m,n] = 0$. These samples are processed with the MR operation described earlier on.

"Normalization Pass", p_p^4

During this pass we visit the sub-blocks in forward scan-line order, as in p_p^1 and p_p^3 skipping over all sub-blocks which have not yet been found to be significant (as identified by the quad-tree code for bit-plane p in this case). Within each sub-block we visit samples in scan-line order, skipping over all samples, except those which are insignificant, i.e. $\sigma[m,n] = 0$, and for which no information has yet been coded, i.e. $\eta[m,n] = 0$. These samples are processed with a combination of the ZC and RLC operations and, if necessary, the SC operation. At the end of this coding pass all samples have their "visited" state variable reset to $\eta[m,n] = 0$, in preparation for the first coding pass of the next bit-plane.

Rich Bit-Stream Syntax

There are 6 different features we can assign to the bit stream:

- *RANDOM_ACCESS*: it should be possible to independently decode a subset of the bit-stream in order to reconstruct smaller regions of the image.

- *SNR_PROGRESSIVE*: it should be possible to decode only the first N bytes of the bit-stream, where N is a user-specified value over which the algorithm has no control.
- *SNR_PARSABLE*: it should be possible to parse the bit-stream in order to extract an N byte subset, where N is a user-specified value over which the algorithm has no control.
- *RESOLUTION_PROGRESSIVE*: the bit-stream should be organized so as to include all information relevant to lower resolution levels before information from higher resolution levels.
- *RESOLUTION_PARSABLE*: it should be possible to parse the bit-stream in order to extract a subset which represents a lower resolution image, for each of the resolution levels offered by the wavelet transform.
- *COMPONENT_PARSABLE:* it should be possible to parse the bit-stream in order to extract a subset, which contains a smaller number of components.

Bit-Stream Layers

To understand the structure of the remainder of the bit-stream, it is necessary to introduce the concept of a bit-stream layer. Typical bit-stream might have anywhere from 1 to 50 or more layers, depending upon the need for SNR scalability. The number of bit-stream layers corresponds to the number of distinct bit-rates for which a rate-distortion optimal subset of the original bit-stream is readily identifiable. If SNR scalability is not required, i.e. neither the *SNR_PROGRESSIVE* nor the *SNR_PARSABLE* profile flags is required. Then a single layer suffices and this layer will generally contain a rate-distortion optimal subset of all code words which were generated by block coding, optimized for some target bit-rate or distortion constraint.

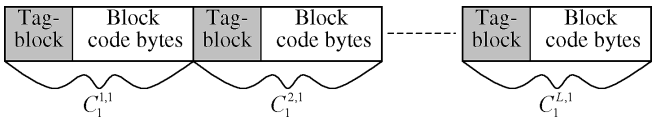

Figure 3. Single layer bit-stream organization.

In general, we have Λ bit-stream layers, labeled λ = 1,2,…Λ. Every bit-stream layer is inherently scalable with respect to resolution and number of image components (i.e. color), because it is composed of a separate component, $C_\lambda^{l,c}$ for each resolution level, l=1,2,...,L and each image component, c. Resolution level l=1 corresponds to the lowest resolution level in the Wavelet transform; it contains the LL band, in addition to the usual LH, HL and HH bands. Each layer component, $C_\lambda^{l,c}$ commences with a "tag-block", which is followed immediately by the code bytes associated with every code block represented by the layer component. The tag-block starts on a byte boundary and is padded to an integral number of bytes, if necessary. The tag-block identifies the *code blocks* from each subband in the relevant resolution level which are included in the layer component. It also identifies the *truncation points* for each included code-block, from which the decoder can determine the set of coding passes for which information is available. Other information represented by the tag-block includes the *number of code bytes* which are being sent *for each included code block* and the *maximum bit-depth*, $p_i^{\max}$ for each code block which is being included for the first time. We begin by considering a bit-stream with only a single layer, which represents a monochrome image (i.e. only one image component, c=1). In this case, the layer components always appear in order of increasing resolution, as illustrated in Figure 3. In this way, the bit-stream will possess the *RANDOM_ACCESS, RESOLUTION_PARSABLE* and *RESOLUTION_PROGRESSIVE* features. In fact, all bit-streams constructed using the layered component syntax outlined here will possess the *RANDOM_ACCESS* and *RESOLUTION_PARSABLE* features, at a minimum.

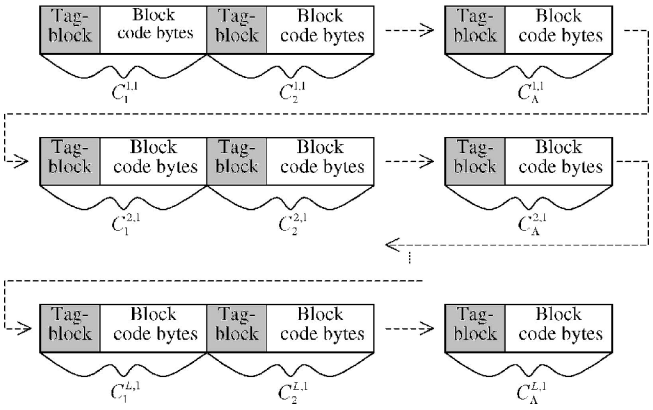

Figure 4. Multi-layer RESOLUTION_PROGRESSIVE bit-stream organization

We define two different organizations for the com-ponents of these different bit-stream layers. The first organization, illustrated in Figure 4, is suitable for gener-ating bit-streams with the *RESOLUTION_PROGRESSIVE* feature; of course, such a bit-stream is also SNR scal-able. The second organization, illustrated in Figure 5, is suitable for generating bit-streams with the *SNR_PRO-GRESSIVE* feature.

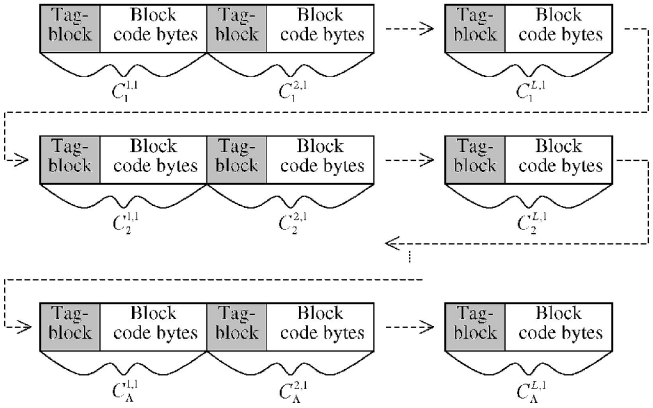

Figure 5. Multi-layer SNR_PROGRESSIVE bit-stream organization

Tag Trees

Before we can discuss the organization of the tag blocks mentioned above, it is important to introduce the concept of what we call a "tag tree". This is a particular type of tree structure, which provides the framework for efficiently representing tag block information, which exhibits significant redundancy between the different blocks of a subband, and between different bit-stream layers. We will use the same tag tree structure for representing different types of information within tag blocks. There are two key differences between tag trees and conventional quad-tree coding schemes. Firstly, the values associated with each node need not be binary-valued. Secondly, and most importantly, the information in a tag tree may be coded in any order. In particular, the coding process for a tag tree is driven from the leaves rather than the root.

Let $q_1[m,n]$ denote a two-dimensional array of quantities, which we would like to represent via the tag tree. We associate these quantities with the leaf nodes of the tree. In practice, we will have one node for each code block in a given subband, so that the two-dimensional array, $q_1[m,n]$, corresponds to the array of code blocks which partition the relevant subband. The values themselves are all non-negative integers. Let $q_2[m,n]$ denote the nodes at the next level of the quad-tree structure. Each of these nodes is associated with a 2x2 block of leaf nodes, except at the boundaries of the original array. We continue in this fashion, defining smaller and smaller arrays, $q_3[m,n]$, $q_4[m,n]$ ..., at higher levels in the tag tree until we reach the root level, which consists of a single node, $q_K[0,0]$. The quantity associated with each non-leaf node is the minimum of all descendant nodes.

Each node in the tag tree maintains a separate state variable, which we shall denote $t_k[m,n]$, whose interpretation is that sufficient information has already been encoded to identify whether or not $q_k[m,n] \geq t_k[m,n]$. This state variable is initialized to zero for all nodes, before any information is coded. As mentioned, the tag tree coding algorithm is driven from the leaves. The algorithm may be summed up as a procedure, $T(m,n,t)$, where m and n identify the row and column indices of the leaf node for which information is requested and t identifies a threshold. The procedure sends the sufficient information to identify whether or not $q_1[m,n] \geq t$. Details of the algorithm can be found in {D. Taubman 1998}. The intuitive idea behind it is as follows: we start at the root node sending the minimal amount of information to identify whether or not $q_K[0,0] \geq t$. If this fact is not already known; we then move down the tree toward the leaf node which we are interested in. Updating the node to reflect any information which can be deduced from what is known about the parent (i.e. the value must be no smaller than $t_{\min}$) and repeating the process for that node. This procedure, $T(m,n,t)$, may be invoked many times in constructing the tag block for any particular bit-stream layer component.

Anatomy of a Layer Component

Recall that the layer component $C_\lambda^{l,c}$ represents new information from the code blocks of the subbands in resolution level l, from image component c, which is being introduced in bit-stream layer λ. The component commences with a tag-block, which consists of a sequence of code bits identifying which code blocks are included from each subband, along with additional information concerning the maximum bit-depth, truncation point, and number of code bytes being sent for each included code block. The tag-block is padded out to an integral number of bytes and followed immediately by the code bytes themselves, for each block included in the layer component.

Inclusion Information

We utilize a separate tag tree for each subband, to efficiently represent the bit-stream layer in which a code block is included for the first time. The leaf nodes of the tag tree correspond to the code blocks in the subband and the quantities, $q_1[m,n]$, associated with each of these leaf nodes represent the index of the bit-stream layer in which the code block is first included, minus 1. For any given code block, the inclusion information is represented in one of two different ways, depending upon whether or not the block has been included in a previous bit-stream layer. If it has already been included in a previous layer, i.e. $\lambda[m,n] < \lambda$, we simply send a single bit to identify whether or not any new information for the code block is included in the current layer. If the bit-stream has not yet been included in any previous bit-stream layer, we invoke the tag tree coding procedure $T(m,n,\lambda)$. This operation emits any bits required to identify whether or not $q_1[m,n] \geq \lambda$, i.e. whether or not $\lambda[m,n] > \lambda$, which is exactly the information required.

Maximum Bit-Depth Information

For each code block, $B[m,n]$, which is included in the bit-stream for the first time, we must identify the most significant bit-plane, $p^{\max}[m,n]$, with respect to which any sample in the code block is significant. Now the maximum value, which can be assumed by, $p^{\max}[m,n]$ is $P-1$ where P is the number of bits used to represent the relevant quantized magnitudes. We utilize a second tag tree to efficiently represent $p^{\max}[m,n]$ via the number of missing most significant bit-planes, i.e. $P-1-p^{\max}[m,n]$.

Truncation Point Information

For every code block, B_i, which is to be included in the bit-stream, we must identify the new truncation point which applies to the code block's bit-stream. Let $n_i^{\min}$ denote the minimum value, which this new truncation points, can take on. For each code block which is to be included in the bit-stream, we must then identify the difference between the new truncation point and its minimum value, i.e. $n_i - n_i^{\min}$, which must lie in the range $0 \leq n_i - n_i^{\min} < 4(P-1)$. We send this difference by means of a simple variable length code, see {D. Taubman 1998} for details.

Code Size Information

We need to identify the number of bytes, which are being sent for each code block included in the layer component. The number of bytes, which must be sent for block, B_i is $\Delta R_i = R_i^{n_i} - R_i^{n_i^{\min}-1}$, where $n_i^{\min}$ is defined above and $n_i - n_i^{\min}+1$ is the number of new coding passes for which information is being included. The number of bits used to represent ΔR_i is β_i

$= \beta_\lambda + [\log_2 (n_i - n_i^{\min} + 1)]$, where β_λ is determined at the beginning of the pass so as to be large enough to represent the ΔR_i values for all code blocks from the subband which are included in the layer.

References

1. D. Taubman. EBCOT (Embedded Block Coding with Optimized Truncation). ISO/IEC JTC 1/SC 29/WG1. 1998 Oct 21.
2. D. Taubman, A. Zakhor. Multirate 3-D Subband Coding of Video. *IEEE Transactions on Image Processing.* 1994 Sep; 3(5):572-588.