Bandwidth Compression (BWC) Guide for

JPEG 2000 Visually Lossless and Numerically Lossless Compression of Imagery Data

Working Draft 1.2

22 January 2004

P.O.C. James H. Kasner
Eastman Kodak Company
2600 Park Tower Drive
Suite 601
Vienna, VA 22180
(571) 226-1620
james.kasner@kodak.com

Table Of Contents

List Of Figures

List Of Tables

1 Introduction

1.1 Brief Algorithm Description

The core of the BWC algorithm is based on Part 1 of the Joint Photographic Experts Group 2000 (JPEG 2000) still image compression standard. After a process of tiling, wavelet transform, quantization, entropy coding, and JPEG 2000 codestream formation, a compressed file is produced.

1.2 Heritage Overview and Background

The heritage compression algorithms used today include several versions of the JPEG Discrete Cosine Transform (DCT) algorithm that is the predecessor to JPEG 2000:

- National Imagery Transmission Format (NITF) JPEG Discrete Cosine Transform (DCT) algorithm for primary dissemination.
- National Imagery Transmission Format Standard (NITFS) JPEG DCT algorithm for secondary/tactical dissemination throughout NGA.
- NGA Method 4 (also known as Down-sample JPEG) for use in secondary/tactical dissemination throughout NGA to very bandwidth-constrained users.

Other heritage algorithms include the DCT and DPCM based Tape Format Requirements Document (TFRD) algorithms:

Most of the technology used in these heritage compression algorithms was developed a decade ago

2 Algorithm Description

This section of the document discusses the processes that comprise a full JPEG 2000 image coding system. The coding framework is specified extensively in the 15444-x family of documents that are published by the ISO/IEC international organizations. The international standard is currently composed of ten parts. Part 1 describes the baseline coding system, which represents the minimum functionality required for compliance with the standard. Part 2 describes extensions to the baseline system that may be useful for specific applications but are not necessary for compliance. Part 3 specifies video specific extensions. Part 4 discusses the procedures to apply when testing for conformance. Part 5 will publish reference software written in the C and Java programming languages to assist developers in understanding the intricacies of the baseline system. Part 6 of the standard describes the Mixed Raster Content (MRC) model for describing compound documents and the JPM file format. Part 8 of the standard deals with security issues related to intellectual property and imagery. Part 9 of the standard describes the JPIP protocol that allows imagery to interactively streamed over networks. Part 10 of the standard defines more advanced 3D encoding techniques and includes methods for dealing with floating point imagery. Part 11 of the standard deals with wireless networks and use of JPEG 2000.

This technical memo only discusses elements from Part 1 of the international standard, which will henceforth be referred to as the "baseline standard", or simply the "standard". There are a number of technologies of interest in other portions of the standard that will someday impact enterprise systems. Future versions of this document will address new technologies as the imaging community adopts them.

The focus of this technical memo is on the JPEG 2000 compression system, while the baseline standard describes JPEG 2000 from a decompression standpoint. However, the baseline standard presents many guidelines for developing the compressor. As such, there will be considerable overlap between this technical memo and the standard. Throughout the discussion of JPEG 2000 technologies and processes, references will be made to sections of the standard to assist in linking the two documents together. Note that the technical memo is not intended to be a stand-alone document with respect to understanding the framework of JPEG 2000. This technical memo coupled with the standard itself should provide the necessary information for implementing the coding system.

The standard discusses several compression-side options that are left open to the developer for customization and optimization purposes. This flexibility is advantageous from an open architecture standpoint, but is a burden to the developer if all options are to be supported. The standard also leaves several key elements of the coding process unspecified because of scope limitations (e.g., rate-control, wavelet and quantization normalization, etc.). The main objectives of this technical memo and the JPEG 2000 sections of the other related requirements documents are to fix the open parameters (i.e., a system-wide JPEG 2000 profile), clarify vague portions of the standard, and provide complete guidelines and algorithms for developing the end-to-end compression system.

2.1 Algorithm Details

The JPEG 2000 image-coding algorithm consists of several advanced technologies that provide a high degree of compression efficiency and functionality. The technologies are outlined in Section 5.3 of the standard. The core operations that are implemented in the baseline algorithm consist of a discrete wavelet transform, dead-zone scalar quantization, and three-pass bit-plane coding using a context-dependent adaptive arithmetic coder. These coding elements form the typical chain of operations for a transform-based compression system (i.e., transform, quantization, and entropy coding). Following the core operations are the layer, packet, and codestream formation stages. These stages manipulate and order the compressed data in the proper constructs, format, and syntax that is specified in the baseline standard.

A brief summary of the image processing stages follows:

1. **(Section 2.3.1)** The image is registered onto a reference grid. For multi-component images with components at different resolutions, integer sample separation factors are defined such that each component will have the same spatial coverage on the reference grid.
2. **(Section 2.3.2)** The image is separated into contiguous, non-overlapping tiles. A tile is defined relative to the reference grid, not relative to the component coordinate space. A multi-component image tile will have a tile from each component, each of which is called a tile-component. Tiles are independently coded to promote parallel processing.
3. **(Section 2.3.3)** Each tile undergoes the following processing stages:
    1. Each tile-component is DC level shifted to change unsigned values to signed and symmetrical about zero.
    2. A multi-component (color) transform is applied to the first three components in the tile. (Note: we reference here the Irreversible Component Transform, ICT, and the Reversible Component Transform, RCT. Readers familiar with Part 2 of the standard should not confuse these transforms with the more general Multiple Component Transform, MCT, framework in 15444-2.)
    3. Each tile-component undergoes the following processing stages:
        1. **(Section 2.3.3.1)** Discrete wavelet transform to decompose the samples into multiple resolutions. Each resolution is comprised of subbands, which are spatially localized sets of wavelet coefficients that are related to horizontal, vertical, and diagonal spatial frequencies.
        2. **(Section 2.3.3.2)** Quantization to reduce the precision of the wavelet coefficients for the purpose of controlling the trade-off between overall rate and distortion.
        3. Region-of-interest (ROI) bit-wise shift is applied to the identified quantized subband wavelet coefficients.
        4. Subband partitioning into contiguous, non-overlapping precincts and code-blocks to localize the data, which enhances compression efficiency, limits complexity, facilitates rate-control, and enables decoder random access and extraction. Precincts can be defined per resolution; code-blocks are defined once for the image. Precincts and code-blocks are bounded by subband extent. Code-blocks are further bounded by precinct extent.
        5. **(Section 2.3.3.3)** Code-block entropy coding to reduce the number of bits required to represent the quantized coefficients. Code-blocks are independently coded to promote parallel processing.
    4. **(Section 2.3.3.4)** Organization of tile compressed data into quality layers using a rate-control procedure.
    5. **(Section 2.3.3.5)** Organization of tile compressed data into small segments called packets, which is the smallest unit of the compressed data. A packet corresponds to the compressed data produced for one tile, component, resolution, layer, and precinct.
    6. **(Section 2.3.3.7)** Formation of the tile codestream, which is composed of marker segments for conveying tile-specific coding information and the packets in a pre-defined order. The tile codestream can be split into multiple segments called tile-parts.
4. **(Section 2.3.3.9)** Formation of the image codestream, which is composed of marker segments for conveying image-level coding information and the tile codestreams in a pre-defined order. Tile data nominally appear in the image codestream in raster-scan order (this is not a requirement of 15444-1 but it is the most common ordering of data and recommended by this guideline), but tile-parts relating to a particular tile are not necessary contiguous and can appear interleaved with tile-parts from other tiles. Tile-parts for a given tile appear in order according to the packets that are contained in the tile-part.

Note that the notion of multiple components, precincts, regions-of-interest, and to some extent, the reference grid may be irrelevant to the compression of some types of imagery systems. The discussion of these elements in this document is merely for the sake of completeness, and is by no means thorough. Developers are referred to the standard for more information.

2.2 Definitions and Processing Conventions

This section defines the image data structures and coding elements, and processing operations for the JPEG 2000 compression algorithm. The image data structures are tile, tile-component, resolution, subband, precinct, and code-block. The coding elements that are defined in JPEG 2000 are bit-plane, coding pass, layer, packet, and tile-part. Sections B.1 through B.9 of the standard describe these structures in detail.

JPEG 2000 describes the concept of a reference grid for registering the components of the image to a common coordinate space. All components share the same top-left coordinate on the reference grid, location (0,0). The actual start of image data need not coincide with this location. It can be offset down and to the right of the (0,0) anchor point, but the fundamental alignment of image samples between components may not be altered. The area of the reference grid that contains valid image samples is called the image area and the reference grid point denoted by (XOsiz, YOsiz) is its top left coordinate. When the original components have different resolutions, integer sample separation factors are defined such that each component will have the same spatial coverage on the reference grid. Note that tiles are defined relative to the reference grid and can be offset relative to the component anchor point (reference XTOsiz, YTOsiz). The fact that the tiles are defined in the grid coordinate space implies that the tile size may vary when mapped to the component coordinate space, where the samples are not spread by the integer sample separation factor. The reference grid is convenient for defining the location of different image data structures in different coordinate spaces. The reference grid can also be used to define basic image manipulation operations such as cropping, flipping, and rotation by integer multiples of 90 degrees.

Figure 2.1 shows a single-component image that is decomposed into a set of tiles. The image offset point is also shown to be (0,0), or the reference grid origin. When multiple components exist, each tile consists of tile-components. Tile-components are defined in the coordinate space of the component. Therefore, the tile-component dimensions may differ depending on the reference grid separation factors that are defined for the components. The figure portrays a case where the image dimensions are not divisible by the tile dimensions. The tiles along the right and bottom of the image do not have nominal dimensions in this situation. Tiles are independently processed, which implies that processing order is arbitrary. Tile codestreams may appear in any order in the final image codestream. A tile index is included in the tile header structure that allows the tiles to be assembled into their proper order. This guideline recommends that tile data appear in raster-scan order in the final image codestream. This is the ordering in which the majority of imaging applications will generate, store, and utilize imagery data.

Figure 2.2 shows a single tile-component that is decomposed by a two-dimensional discrete wavelet transform into a set of subbands. The dyadic decomposition style defined in the baseline standard, where the transform is recursively applied to the low-pass signal, is shown in the figure. The two-dimensional transform is implemented as two separable one-dimensional transforms (i.e., horizontal and vertical). Each successive decomposition level reduces the resolution and dimensions by approximately a factor of two. The wavelet subband coefficients describe the image in terms of horizontal, vertical, and diagonal spatial frequencies at different resolutions. The coefficients are localized in spatial influence due to the finite support of the wavelet filters used in the transform. Figure 2.2 shows the wavelet decomposition from a resolution and subband perspective. The first example shows the multiple resolutions that are created by the transform. In this example, three decomposition iterations are performed, resulting in four unique resolutions. The resolutions are labeled R0 through R3, where R0 is the lowest resolution rendition of the image available. Note that the resolution labeling scheme shown in Figure 2.2 is that adopted by the JPEG 2000 standard. Other communities have adopted a different scheme to denote resolution sets, or rsets, of an image. Using the enterprise rset vernacular, a full resolution image is denoted as R0, a 1/4 resolution image is denoted R1, and a 1/16 resolution image is denoted as R2, etc. The two labeling schemes essentially run in opposite directions.

The second example in Figure 2.2 shows the subbands and the labeling convention that is employed by the standard. The LL, HL, LH, and HH designations refer to a particular combination of one-dimensional decompositions in horizontal then vertical order. For example, HL means that the coefficients underwent a high-pass horizontal decomposition and then a low-pass vertical decomposition; LH means that the coefficients underwent a low-pass horizontal decomposition and then a high-pass vertical decomposition. The standard includes equations to calculate the size of an intermediate resolution and the associated subbands, which can be used to locate the data with pixel precision.

Figure 2.3 shows an example of precinct and code-block partitions for resolution R3, or subbands 1HL, 1LH, and 1HH. Precincts and code-blocks are contiguous, non-overlapping, rectangular regions that are defined to spatially localize the subband wavelet coefficients for the purposes of entropy coding and enabling random access to the codestream. Entropy coding is performed on the code-blocks independently (i.e., no data from outside a code-block is used during the coding operation), which promotes parallel processing. Both precincts and code-blocks are anchored from location (0,0) on the reference grid. Precinct size can be customized at the component and resolution level, while code-block size is defined at the image or component level. Specific rules are set by the standard with

respect to the sizes of precincts and code-blocks. For example, code-block dimensions must be an integer power-of-two, the minimum value for a dimension is 4, the maximum is 1024, and the product of the dimensions cannot exceed 4096.

Precincts and code-blocks are bounded by subband extent. Code-blocks are further bounded by precinct extent. The example in Figure 2.3 shows a partitioning that neatly fits within the confines of the subband and tile boundaries. This is possible when the image and tile anchors are set at (0,0) and the tile dimensions are a power of two. Note that a precinct includes information from the HL, LH, and HH subbands (e.g., $P_3 1$). This is because the notion of a precinct relates to resolution, and is not subband specific. The second example in Figure 2.3 illustrates a code-block partition. The code-blocks for a precinct are ordered sequentially by subband in the following order: LL or HL, LH, and HH. Within a subband, the code-blocks are raster-scan ordered. This results in the numbering shown in the second example. The ordering is important when forming the contents of a packet.

Entropy coding of the integer quantization indices is performed using a three-pass bit-plane coding scheme that utilizes a context-dependent adaptive arithmetic coder. Figure 2.4 shows an example of bit-planes and coding passes from code-blocks at three different resolutions. The quantization indices within a code-block are bit-plane coded starting from the most significant bit-plane with a non-zero value to the least significant. Each bit in a bit-plane is visited only once in three scans, referred to as coding passes. The three coding passes are labeled significance propagation, magnitude refinement, and cleanup. Each coding pass uses contextual information regarding the received bits in a given bit-plane to condition the arithmetic coder's probability models. For example, the most significant bit-plane within a code block will largely be comprised of "0" bits, with "1" bits occurring infrequently. The arithmetic coder's significance propagation probability models take this into account. Once the most significant bit in a wavelet coefficient is encoded, the magnitude refinement probability contexts are used. The cleanup coding pass is used to handle all other wavelet coefficients for which we have not yet received any bits.

After the three coding passes, the coder will have considered all bits in a code-block bit-plane. The most significant bit plane of each code-block is always encoded with just a cleanup pass. The coding passes provide a convenient truncation point for ending the code-block contribution to the codestream. By coding each bit-plane as three separate sequences, the compression algorithm is able to control the rate at a finer level. Considering this bit-plane coding framework, the conceptual operation of a rate-control module in a JPEG 2000 compression system is the selection of the most relevant coding passes from every code-block to achieve a target average bit rate.

A layer in the JPEG 2000 coding framework is defined to be some number of consecutive coding passes from every code-block in a tile. It is important to note that layers do not span tiles; the only mechanism to control layering or ordering of codestream data amongst tiles is to use tile-parts (see below). Conceptually, each layer adds more bits of precision to the quantized wavelet coefficients, which improves the reconstructed image quality. The number of coding passes can be different for each code-block. A code-block can even contribute zero bits to a layer. Layer formation is performed by a rate-control module, since layers are typically defined by an average bit rate.

Packets are defined by JPEG 2000 as the compressed data produced for one tile, component, resolution, layer, and precinct. Packets represent the smallest unit of the compressed data. For example, in Figure 2.3, a packet would be created from the code-block coding pass data corresponding to precinct, $P_3 1$. The sequential code-block order depicted in the example would be used to form the packet. During the tile codestream formation stage, packets are interleaved in one of several progression orders defined in the standard (e.g., Layer-Resolution-Component-Precinct, or LRCP). Packets are important from the perspective of being able to efficiently extract partial image data without the need to perform full decompression. For example, if an application requires compressed data from an intermediate resolution and quality level, a parsing tool can satisfy this request by extracting just the relevant packets.

Tile-parts are segments of the codestream for a tile. Tile-parts simply break the tile codestream at the end of coding passes and can be used to distribute and interleave the data throughout the file codestream. JPEG 2000 requires that each tile-part hold at least one packet. The tile-part syntax defined by the standard allows packet progression order changes to be signaled in the tile-part marker segment. This is another reason to use tile-parts beyond layering and rate control concerns. Two tile-parts would exist in this example; the first has a Resolution-Layer-Component-Position, or RLCP, packet progression order, while the second would signal a switch to LRCP.

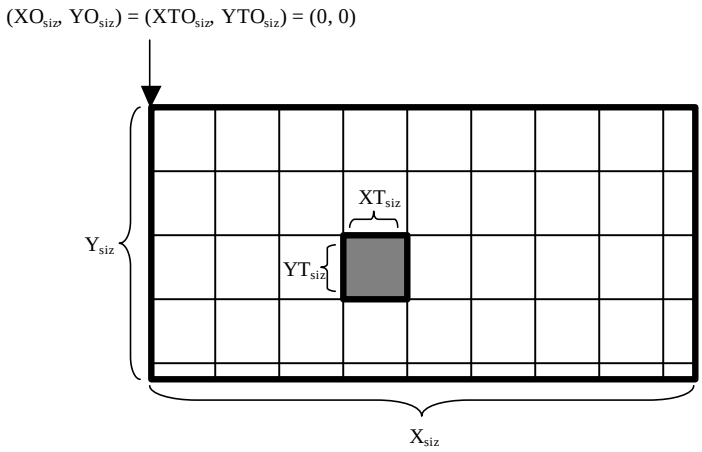

Figure 2.1. An example of the tile partitions.

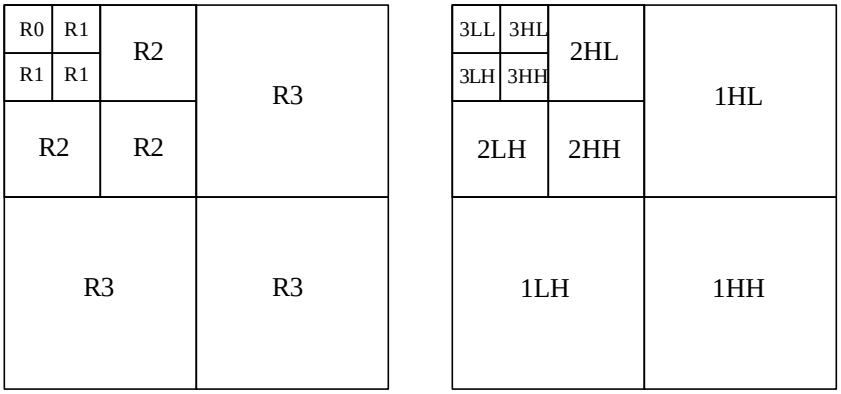

Figure 2.2. An example of a tile-component decomposition into wavelet subbands.

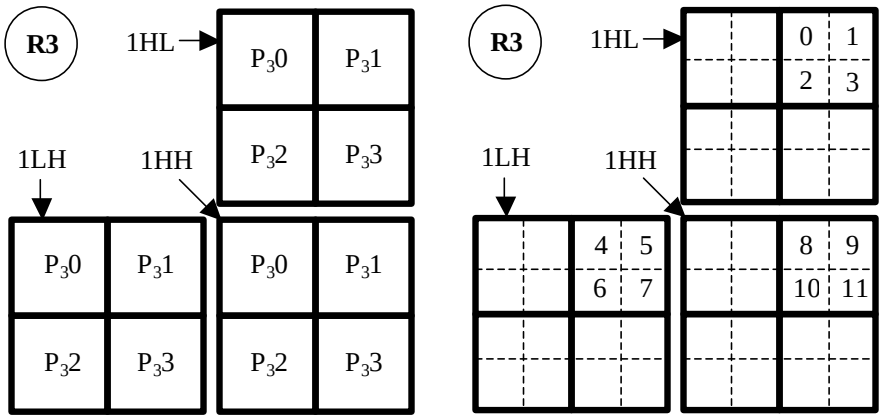

Figure 2.3. An example of the precinct and code-block partitions.

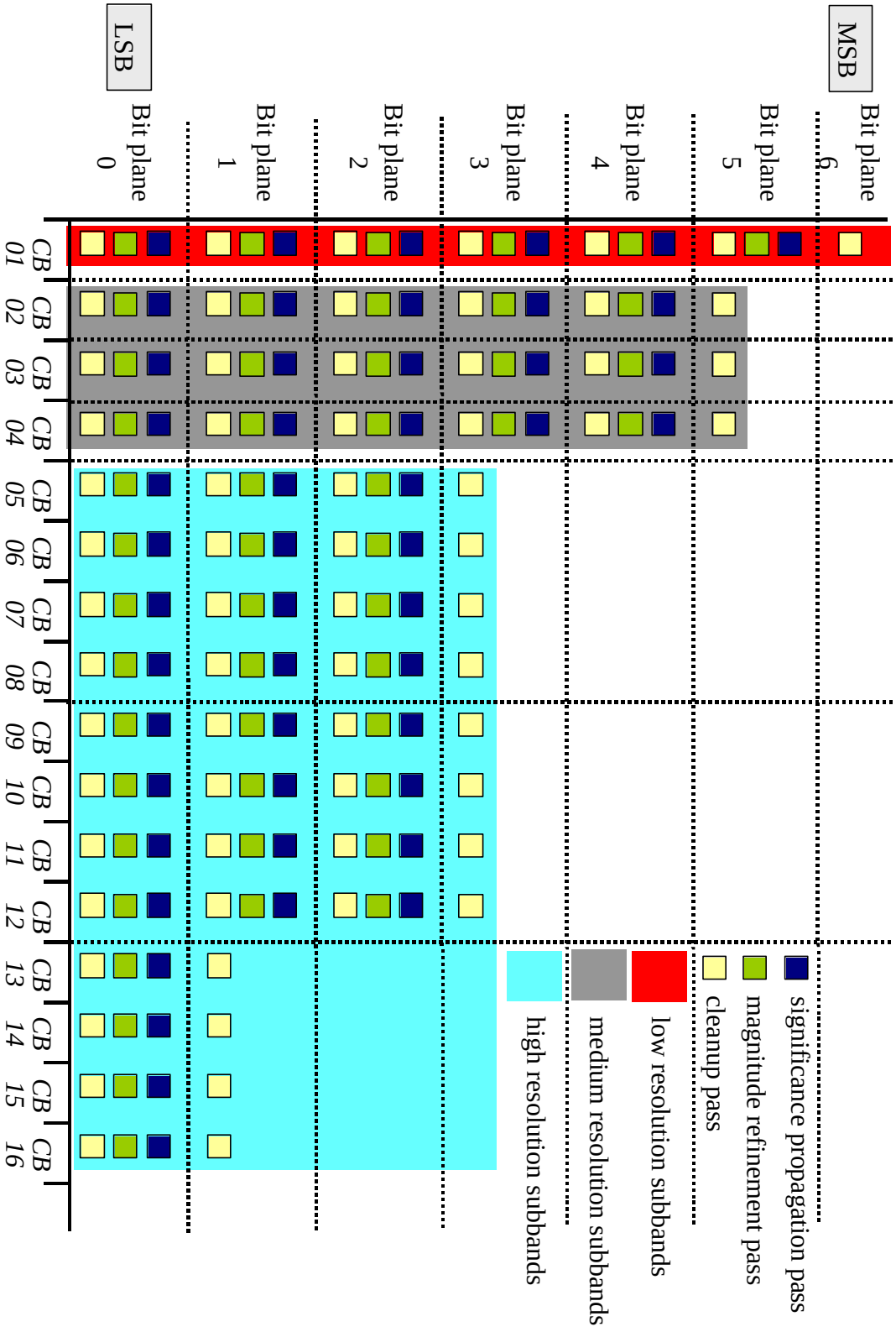

Figure 2.4. An example of code-block bit-planes and coding passes.

2.3 Methods

2.3.1 Reference Grid (Annex B)

Annex B of ISO/IEC IS 15444-1 describes the image data structure concepts of JPEG 2000. One of the fundamental constructs of JPEG 2000 is the *reference grid*. The reference grid may be thought of as a grid of points upon which all spatial entities in JPEG 2000 are located. Two of these spatial entities include the *image area* where the visible contents of the image lie and *tiles* which partition the image area into contiguous, non-overlapping rectangular regions. The reference grid also provides for subsampling of image components relative to the reference grid. The influence of the reference grid extends beyond the definition of image area, tiles, and component subsampling. Wavelet *subbands* are mapped to the reference grid at different resolutions, as are other constructs such as *code-blocks* and *precincts*.

We will not venture into all of the subtle nuances of the reference grid in this section. For most imagery, many of the choices we shall make regarding the reference grid will greatly simplify its use. The reader is encouraged to read Annex B.1 through Annex B.9 to get a feel for the reference grid and its importance in JPEG 2000. The following paragraphs are specific in that the parameter values are used for a specific imaging system. The discussion, however, is applicable to any JPEG 2000 system.

2.3.2 High-Level Image Processing (Annex B.1 – B.5)

Figure 2.5 shows the high-level image processing of the JPEG 2000 encoder. The image to be compressed is first partitioned into non-overlapping contiguous tiles. Each tile is processed independently to generate the individual tile codestream portions of the JPEG 2000 compressed codestream. The tile processing is repeated for every tile until all tiles have been processed. Finally the JPEG 2000 compressed image codestream is formed and the encoding terminates. The following recommendation section considers more specific tiling-related quantities and their relationship to the JPEG 2000 reference grid. The section also covers many of the parameters in the SIZ marker segment and their values.

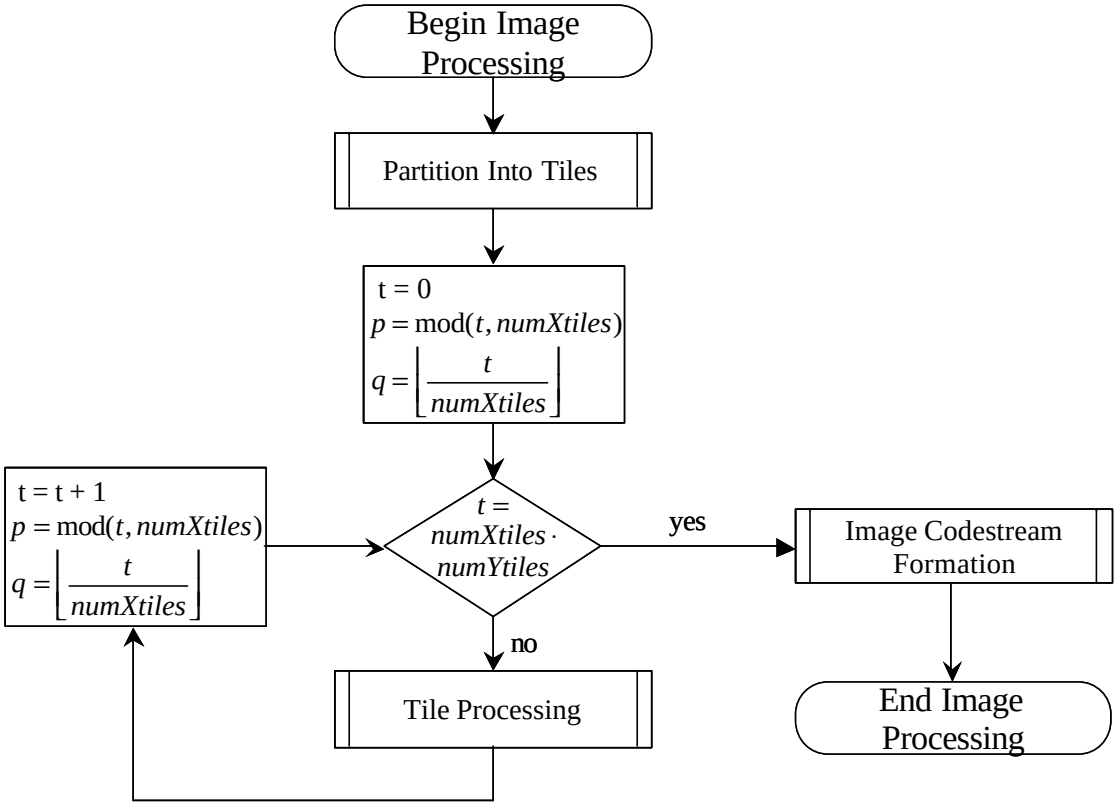

Figure 2.5. Image processing flow diagram for encoder

Recommendation

Tiles provide a simple method by which large imagery may be processed in parallel. None of the JPEG 2000 constructs (i.e., code-blocks, precincts, wavelet transform, layers, etc.) cross tile boundaries. It is therefore possible to independently compress tiles if desired. (Note: sophisticated rate control schemes may want to jointly compress tiles to control the final bitrate.) Current enterprise compression algorithms (e.g. NITF or TFRD) utilize tiling schemes for rapid access image blocks. This practice has been adopted for JPEG 2000 compression within this guideline as well. JPEG 2000 tiles are subtly different from NITF and TFRD tiles. While all of these tiling schemes utilize independently coded tile codestreams, the JPEG 2000 tiles still exist within the overall file codestream. Within previous compression standards, each tile or image block was encoded as an independent subimage with each tile or block forming a complete codestream. The 1,024 x 1,024 tile size was chosen as a good compromise between random access and compression efficiency. It also mimics the FAF block size used by TFRD. Precincts are another construct within JPEG 2000 that allows rapid access to spatial regions within an image. Tiling was chosen over precincts due to its similar conops with current systems.

Imagery compressed according to this guideline shall be partitioned into contiguous non-overlapping tiles nominally containing 1,024 pixels x 1,024 pixels. It is possible that the image dimensions will not be integer multiples of 1,024. In this case the right and bottom sides of the image will contain tiles that have one or possibly both dimensions less than 1,024. The nominal size of the tiles on the reference grid is communicated in the SIZ marker segment that appears in the main header of a JPEG 2000 codestream (see Figure A-3 and Annex A.5.1 of ISO/IEC IS 15444-1). The coordinate limits of the image area on the reference grid are also contained within the SIZ marker segment. The dimensions of a component (i.e. the number of rows or *height*, and the number of columns or *width*, is given by Equation 2.1 (Equation B.2):

$$(width, height) = (x_1 - x_0, y_1 - y_0)$$

Equation 2.1

where,

$$x_0 = \left\lceil \frac{XOsiz}{XRsiz^c} \right\rceil \quad x_1 = \left\lceil \frac{Xsiz}{XRsiz^c} \right\rceil \quad y_0 = \left\lceil \frac{YOsiz}{YRsiz^c} \right\rceil \quad y_1 = \left\lceil \frac{Ysiz}{YRsiz^c} \right\rceil$$

Equation 2.2

The coordinates (x_0,y_0) and (x_1,y_1) specify the top-left and bottom-right corners of the image area taking into account the component columnar and row subsampling factors $XRsiz^c$ and $YRsiz^c$. These parameters are a function of image component, hence the superscript *c*. The parameters, *XOsiz* and *YOsiz* are the column and row offsets of the active image area from the top-left corner (0,0) of the reference grid. The point (*XOsiz*,*YOsiz*) is the top-left corner of the image area. The parameters, *Xsiz* and *Ysiz* determine the bottom-right corner of the image. The bottom-right sample of the image area is at location (*Xsiz* -1,*Ysiz* -1) (see Figure B-1 in ISO/IEC IS 15444-1). The sample (*Xsiz*,*Ysiz*) is not in the image area; it is the boundary of the image area. In general, the right and bottom boundaries of reference grid constructs (tiles, image area) are not included in the construct, but the top and left boundaries are.

It is recommended that the values of ($XRsiz^c$,$YRsiz^c$) = (1,1) and (*XOsiz*,*YOsiz*) = (0,0). This recommendation applies to all original image creators. If a file is chipped to a reduced resolution or to a collection of tiles, the image offsets (XOsiz, YOsiz), tile offsets (XTOsiz, YTOsiz), and reference grid sampling factors (XRsiz, YRsiz) are typically modified. This is done so that no wavelet coefficients need be recomputed. In certain circumstances, it may be possible to retain the original values, but in general it is necessary to change these values or recompute the wavelet coefficients. Since most applications will wish to maintain parsing speed, decoders must be able to handle reference grid parameters that differ from those recommended for original image providers.

The parameters (*Xsiz*,*Ysiz*) should be set to the number of columns and rows in the image respectively. All of these parameters are stored in the SIZ marker segment that is present in the main header of the JPEG 2000 codestream. These parameter choices force the imagery data to be sampled one to one on the reference grid and force the top left corner of the image to be coincident with the top-left corner of the reference grid. The variable, *Csiz*, is set to the number of components in the image. (*Csiz* = *# of Components* in the SIZ marker segment.)

Four additional SIZ marker segment parameters, *XTOsiz*, *YTOsiz*, *XTsiz*, and *YTsiz*, determine the tiling of the image area. The parameters (*XTOsiz*,*YTOsiz*) specify the tile offset from the top-left corner, (0,0), of the reference grid. The nominal (width,height) of a tile on the reference grid are given by the parameters (*XTsiz*,*YTsiz*). The values of *XTOsiz*, *YTOsiz*, *XTsiz*, and *YTsiz*, are constrained by the following sets of relationships (see Equation B.3 and Equation B.4),

$$0 \le XTOsiz \le XOsiz \quad 0 \le YTOsiz \le YOsiz$$

Equation 2.3

and

$$XTsiz + XTOsiz > XOsiz \quad YTsiz + YTOsiz > YOsiz$$

Equation 2.4

Equation 2.3 guarantees that the top-left corner of the top-left tile is either in the image offset area whose bottom-right corner is (*XOsiz*-1,*YOsiz*-1) or just outside of it at first image area sample, (*XOsiz*,*YOsiz*). This guarantees that

no image area samples on the top or left are outside of the image tiles. Equation 2.4 guarantees that the top-left tile contains at least one sample in the image area (i.e. it does not lie completely in the image offset area). For this system's imagery, (*XTOsiz*,*YTOsiz*) shall be set to (0,0) and (*XTsiz*,*YTsiz*) shall be set to (1024,1024).

The numbers of tiles in the column and row directions are given by *numXtiles* and *numYtiles* respectively. These values are computed using Equation 2.5 (see Equation B.5),

$$numXtiles = \left\lceil \frac{Xsiz - XTOsiz}{XTsiz} \right\rceil \quad numYtiles = \left\lceil \frac{Ysiz - YTOsiz}{YTsiz} \right\rceil$$

Equation 2.5

Tiles are indexed from 0 to $numXtiles \cdot numYtiles - 1$ in raster-scan order (left to right, top to bottom, see Figure B-4). Tiles are also indexed in a (column, row) fashion with the variables *(p,q)*, where $p \in [0, numXtiles - 1]$ and $q \in [0, numYtiles - 1]$. The *(p,q)* tile indices may be computed from the tile index, $t \in [0, numXtiles \cdot numYtiles - 1]$, by the expressions in Equation 2.6 (see Equation B.6),

$$p = \mathrm{mod}(t, numXtiles) \quad q = \left\lfloor \frac{t}{numXtiles} \right\rfloor$$

Equation 2.6

Given the *(p,q)* indices of a tile, the top-left and bottom-right coordinates of the tile on the reference grid are determined from Equation 2.7 (see Equations B.7 through B.10),

$$\begin{aligned} tx_0(p,q) &= \max\left(XTOsiz + p \cdot XTsiz, XOsiz\right) \\ ty_0(p,q) &= \max\left(YTOsiz + q \cdot YTsiz, YOsiz\right) \\ tx_1(p,q) &= \min\left(XTOsiz + (p+1) \cdot XTsiz, Xsiz\right) \\ ty_1(p,q) &= \min\left(YTOsiz + (q+1) \cdot YTsiz, Ysiz\right) \end{aligned}$$

Equation 2.7

The (width, height) dimensions of a tile on the reference grid are given by ($tx_1 - tx_0$, $ty_1 - ty_0$) (see Equation B.11). Within the domain of an image component (taking into account subsampling on the reference grid), a tile's top-left and bottom-right coordinates are determined using Equation 2.8 (see Equation B.12),

$$tcx_0 = \left\lceil \frac{tx_0}{XRsiz^c} \right\rceil \quad tcx_1 = \left\lceil \frac{tx_1}{XRsiz^c} \right\rceil \quad tcy_0 = \left\lceil \frac{ty_0}{YRsiz^c} \right\rceil \quad tcy_1 = \left\lceil \frac{ty_1}{YRsiz^c} \right\rceil$$

Equation 2.8

The (width, height) dimensions of the *tile-component* (a tile mapped into component space) are given by ($tcx_1 - tcx_0$, $tcy_1 - tcy_0$) (see Equation B.13 of the ISO standard). It is recommended that the reference grid sampling factors ($XRsiz^c$,$YRsiz^c$) be set to (1,1); therefore, there is no difference between the coordinate sets (tx_1, tx_0, ty_1, ty_0) and (tcx_1, tcx_0, tcy_1, tcy_0). It is worth noting here that the true number of pixels in a tile is given by (tcx_1 - tcx_0, tcy_1 - tcy_0) and not (tx_1 - tx_0, ty_1 - ty_0). Although tile sizes are defined on the reference grid, the number of pixels in a tile is determined by the tile size and subsampling on the reference grid.

A reduced resolution version of the tile may be formed at resolution level, r, by using the nLL subband (see Annex F and Section 2.3.3.1 of this document for definitions of resolution levels, decomposition levels, and subbands), where $n = N_L - r$. Tile-component corner coordinates may be mapped into a given resolution level, r, yielding top-left corner coordinates, (trx_0,try_0), and bottom-right corner coordinates, (trx_1-1,try_1-1), using the expressions in Equation 2.9 (see Equation B.14 of the ISO standard). The dimensions of the wavelet subbands at decomposition level, n_b, are computed using Equation 2.10 (see Equation B.15 of the ISO standard), where the parameters xo_b and yo_b are functions of subband orientation, b, and are given in Table B-1 of ISO/IEC IS 15444-1.

$$trx_0 = \left\lceil \frac{tcx_0}{2^{N_L - r}} \right\rceil \quad try_0 = \left\lceil \frac{tcy_0}{2^{N_L - r}} \right\rceil \quad trx_1 = \left\lceil \frac{tcx_1}{2^{N_L - r}} \right\rceil \quad try_1 = \left\lceil \frac{tcy_1}{2^{N_L - r}} \right\rceil$$

Equation 2.9

$$tbx_0 = \left\lceil \frac{tcx_0 - \left(2^{n_b - 1} \cdot xo_b\right)}{2^{n_b}} \right\rceil \quad tby_0 = \left\lceil \frac{tcy_0 - \left(2^{n_b - 1} \cdot yo_b\right)}{2^{n_b}} \right\rceil$$

$$tbx_1 = \left\lceil \frac{tcx_1 - \left(2^{n_b - 1} \cdot xo_b\right)}{2^{n_b}} \right\rceil \quad tby_1 = \left\lceil \frac{tcy_1 - \left(2^{n_b - 1} \cdot yo_b\right)}{2^{n_b}} \right\rceil$$

Equation 2.10

Equation 2.10 gives the width, $tbx_1 - tbx_0$, and height, $tby_1 - tby_0$, of each subband in the wavelet decomposition for the current tile. Equation 2.10 is very important to the wavelet processing. It determines not only subband dimensions, but also the order in which wavelet subband coefficients are produced (low-pass first or high-pass first).

2.3.3 Tile Processing

Figure 2.6 shows the tile processing flow for the JPEG 2000 encoder. The discrete wavelet transformation, quantization of the wavelet coefficients, and code-block entropy coding (T1 engine) are run on every component for the current tile. After all components in the current tile have been processed, layer formation, packet formation (T2 engine), and tile codestream generation occur. The tile processing procedure is called for every tile in the image (see Figure 2.5).

Tiles are processed independently in JPEG 2000. No procedure within the standard spans more than a single tile. The wavelet transform, quantization, entropy coding, and layer and packet formation all are restricted to the current tile. It is permitted within JPEG 2000 to split up the entropy-coded data associated with a tile into one or more *tile-parts*. Tile-parts may be interspersed throughout the codestream, but they must appear in order of increasing *tile-part index* within the codestream. This allows a type of meta layering to be performed with the data from the various tiles in the compressed codestream.

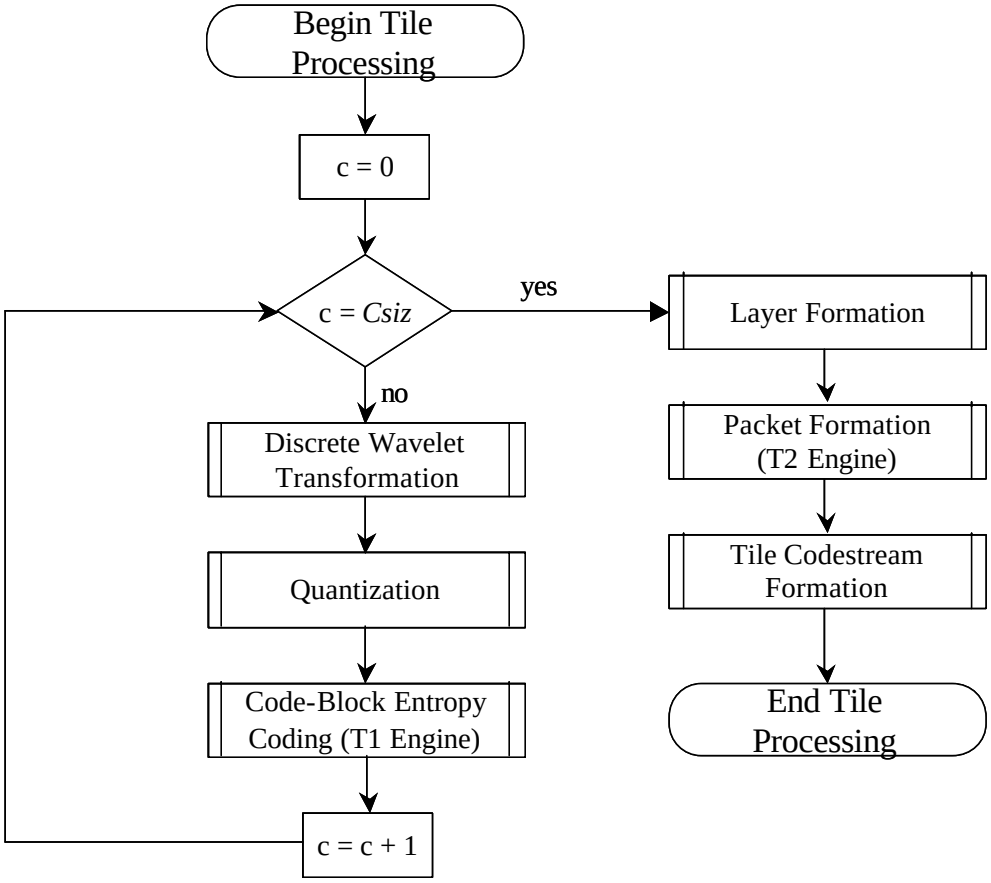

Figure 2.6. Tile processing flow diagram for encoder

2.3.3.1 Discrete Wavelet Transform (DWT) (Annex F)

The 9-7 irreversible (9-7I) discrete wavelet transformation will be used for all *visually lossless* (VL) processing. The 5-3 reversible (5-3R) discrete wavelet transformation will be used for all *numerically lossless* (NL) processing. This section and its subsections serve to further explain and restrict the discrete wavelet transformation description found in Part 1 of the JPEG 2000 standard (ISO/IEC IS 15444-1). The forward and inverse wavelet transform processing is described in Annex F of ISO/IEC IS 15444-1. Annex F of ISO/IEC IS 15444-1 only specifies the normative inverse wavelet transformation processing of a decoder. This section of this document additionally specifies the normative forward wavelet transformation processing that a recommended encoder shall follow. This section will make frequent reference to the sections and equations in Annex F of ISO/IEC IS 15444-1.

The processing steps for the 9-7I and 5-3R wavelet transformations are quite similar. The only differences lie in the lengths of symmetric extension needed for the two transforms and the actual filtering steps of the transforms. Unless explicitly stated otherwise, the text in this section and all of its subsections applies equally to both wavelet transformations. Those areas where the two transforms differ will be clearly indicated in a given subsection or by creation of separate subsections within the text.

2.3.3.1.1 Forward DWT Processing (FDWT) (Annex F.4)

The forward wavelet transformation is informatively described in Annex F.4 of ISO/IEC IS 15444-1. For a recommendation compliant encoder, the procedures described in Annex F.4 are normative. The two-dimensional

discrete wavelet transformation is computed using separable one-dimensional wavelet transformation along the rows and then the columns of the array being processed. Each column or row is sent through a non-expansive filtering operation that creates a set of low-pass (L) and high-pass (H) wavelet coefficients.

The 9-7I wavelet transformation is called "irreversible" because the wavelet transformation is defined using irrational numbers. In general it is not possible within a finite precision floating-point architecture to guarantee reversibility. Thus the 9-7I wavelet transformation is suitable only for lossy (including visually lossless) compression applications. The 5-3R wavelet transformation is "reversible" because it is defined using integer arithmetic and special rounding rules that allow an integer-based architecture to guarantee reversibility when operating on integer data. JPEG 2000 only allows for integer data as input to the compression algorithm and therefore the 5-3R wavelet transformation is truly reversible.

Application of the separable one-dimensional wavelet transformation in the row and columnar directions generates four sets of wavelet coefficients or *subbands* that are labeled: LL, HL, LH, and HH. These labels are also referred to as subband *orientations*. The LL subband consists of those wavelet coefficients resulting from low-pass filtering in row and columnar directions. The HL subband consists of wavelet coefficients resulting from high-pass filtering in the row direction and low-pass filtering in the columnar direction. The LH subband consists of wavelet coefficients resulting from low-pass filtering in the row direction and high-pass filtering in the columnar direction. The HH subband consists of wavelet coefficients resulting from high-pass filtering in both the row and columnar direction. The process of taking an image or wavelet subband and processing it with a forward wavelet transformation is called decomposition or *analysis*.

The FDWT takes as its input the DC-level shifted tile samples $I(x,y)$ and the desired number of decomposition levels (see below), N_L. It generates as its output a set of wavelet coefficient subbands denoted by $a_b(u_b,v_b)$, where b indicates the decomposition level associated with a given subband. The variables u and v are the horizontal and vertical subband coordinates within a particular subband for the current tile being processed. Figure 2.7 illustrates the wavelet subbands generated from a two level wavelet transformation ($N_L = 2$) of the tile samples.

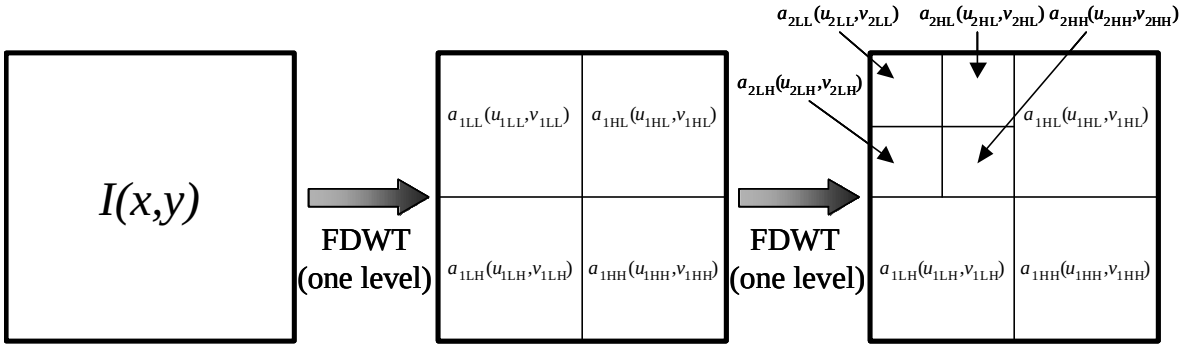

Figure 2.7. Application of the FDWT ($N_L = 2$)

The wavelet subbands are generated by recursive application of the wavelet transformation to the LL subband using the 2D_SD procedure in Annex F.4.2 of ISO/IEC IS 15444-1. This particular type of wavelet decomposition is known as a *dyadic* or *Mallat* decomposition. Each time the tile samples or the LL wavelet subband are processed with the forward wavelet transformation, a new *decomposition level* is formed. The original tile samples, $I(x,y)$, correspond to the subband $a_{0LL}(u_{0LL},v_{0LL})$ and are said to lie at decomposition level zero.

One may also discuss the notion of *resolution level* in conjunction with the wavelet transformation. Resolution level and decomposition level run in the opposite direction. As decomposition level increases, resolution level decreases. In the Figure 2.7, there are two decomposition levels [1,2] and three resolution levels [0,1,2]. The subband $a_{2LL}(u_{2LL},v_{2LL})$ corresponds to resolution level 0, $a_{1LL}(u_{1LL},v_{1LL})$ corresponds to resolution level 1, and $I(x,y)$ corresponds to resolution level 2. It is important to understand this distinction between decomposition and resolution level since the JPEG 2000 standard uses both terms. To further confuse this issue, the notion of resolution level in ISO/IEC IS 15444-1 runs counter to the notion of Reduced Resolution Data Sets (RRDS) used within the community, where R0 is full resolution, R1 is half resolution, etc.

The nomenclature $b = lev$SS in Figure 2.7 refers to the subband with orientation SS (where SS ∈ [LL, HL, LH, HH]) generated at decomposition level lev. The wavelet subband coordinates u and v are bounded by the range $tbx_0 \le u < tbx_1$ and $tby_0 \le v < tby_1$ (see Equation F.8). The parameters tbx_0, tbx_1, tby_0, and tby_1, represent the coordinate limits within each subband at a given resolution level for the current tile. These quantities are computed from Equation 2.10 (Equation B.15 in ISO/IEC IS 15444-1) and they take into consideration the effects of resolution level, subband orientation, and component sampling on the reference grid.

Recommendation

It is recommended that the number of forward wavelet transformation decomposition levels be always set to five, $N_L = 5$. (I.e., there are six resolution levels.) The number of decomposition levels, N_L, and type of wavelet transformation, 9-7I or 5-3R, are found in the applicable COD marker segment for the current tile.

2.3.3.1.1.1 The 2D_SD Procedure (Annex F.4.2)

The FDWT repeatedly calls the 2D_SD procedure to perform the forward wavelet transformation (see Figure F-19). The FDWT procedure is initialized by setting the variable $lev = 1$ and $a_{0LL}(u,v) = I(x,y)$. The 2D_SD procedure is then called once for each decomposition level with the LL subband from the previous decomposition level being fed back into the 2D_SD procedure to generate the next decomposition level. The call to 2D_SD is of the form $(a_{lev\text{LL}},a_{lev\text{HL}},a_{lev\text{LH}},a_{lev\text{HH}}) = \text{2D\_SD}(a_{(lev\text{-}1)\text{LL}},u_0,u_1,v_0,v_1)$. The wavelet transformation is non-expansive, so the total number of wavelet coefficients in the four newly created subbands $(a_{lev\text{LL}},a_{lev\text{HL}},a_{lev\text{LH}},a_{lev\text{HH}})$ is equal to the number of wavelet coefficients in the parent subband $a_{(lev\text{-}1)\text{LL}}$. The inputs, u_0, u_1, v_0, and v_1, are the values tbx_0, tbx_1, tby_0, and tby_1, corresponding to subband $b = (lev\text{-}1)$LL (i.e. the parent LL subband).

The internal calls made by the 2D_SD procedure are shown in Figure F-22 of ISO/IEC IS 15444-1. The first call is to the procedure VER_SD which wavelet transforms the columns of the array $a_{(lev\text{-}1)\text{LL}}$ to generate a new array $a_{\text{VER\_SD}}$. The array $a_{\text{VER\_SD}}$ is then passed on to the HOR_SD procedure which transforms the rows of $a_{\text{VER\_SD}}$, generating a new array, a, containing four *interleaved* subbands. Finally a 2D_DEINTERLEAVE procedure is called which separates the interleaved subband samples in a into the four subbands $(a_{lev\text{LL}},a_{lev\text{HL}},a_{lev\text{LH}},a_{lev\text{HH}})$.

The wavelet transformation processing employed within ISO/IEC IS 15444-1 utilizes the lifting form of the 9-7I and 5-3R filters. Traditional wavelet and other types of subband processing use a convolution and decimation based approach. In this approach two distinct filters, a low-pass and a high-pass filter, are separately convolved with the input data and every other output sample thrown out. Lifting implementations perform these processes simultaneously using a series of prediction and update steps. There is no distinct low-pass and high-pass filter, although we can equate the lifting processing to a pair of low-pass and high-pass convolution filters.

With lifting, the low-pass and high-pass filtering operations are performed jointly. One consequence of this type of processing is that the low-pass and high-pass output channels are interleaved. Every other output sample belongs to either the low-pass or high-pass output channel. Thus a deinterleaving procedure is needed to separate the subband data.

2.3.3.1.1.2 The VER_SD Procedure (Annex F.4.3)

The VER_SD procedure performs wavelet transformation processing in the columnar direction. It takes as its input $a_{(lev\text{-}1)\text{LL}}$, and the coordinate bounds of the parent subband u_0, u_1, v_0, and v_1. The procedure produces as its output the array $a_{\text{VER\_SD}}$, which is columnar filtered version of $a_{(lev\text{-}1)\text{LL}}$. The internal calls made by the VER_SD procedure are shown in Figure F-24 of ISO/IEC IS 15444-1.

Note - We are using a different variable name $a_{\text{VER\_SD}}$, for the output of this procedure than that in ISO/IEC IS 15444-1. In the JPEG 2000 standard, the variable $a_{\text{VER\_SD}}$ is simply referred to as a (see Figure F-24). This variable

name is used in other places as well, so to help avoid confusion we will distinguish which variable we are referencing.

Each column of $a_{(lev-1)LL}$ is extracted, filtered using the 1D_SD procedure (Annex F.4.6. and Section 2.3.3.1.1.5 of this document), and then placed back into the array, a_{VER_SD}. Figure 2.8 illustrates the interleaving effects of the VER_SD procedure. In this figure it has been assumed that the top left corner of $a_{(lev-1)LL}$ lies at an even index (u_0 is even). Furthermore it has been assumed that the number of rows in $a_{(lev-1)LL}$ is even. If u_0 is an odd index, then the wavelet transformation will generate a high-pass output sample first. If the number of rows in $a_{(lev-1)LL}$ were odd, then the bottom row in Figure 2.8 would not be present and we would have one more low-pass sample than high-pass samples for even u_0 and the converse would be true for odd u_0.

This illustrates an important concept in JPEG 2000. If the starting index in a subband array is even, then the wavelet transformation processing will be "low-pass first". If the starting index is odd, the wavelet transformation processing will be "high-pass first". This behavior is rather unique and may not be familiar to readers who have performed wavelet processing before. For this imagery, the tile and image area offsets are set to zero and the tile widths are also even. Thus all wavelet transform processing for this specific imagery will be "low-pass first". This however, may not be the case for all other types of data (e.g. non-zero offsets may be used to align other data sets with one another).

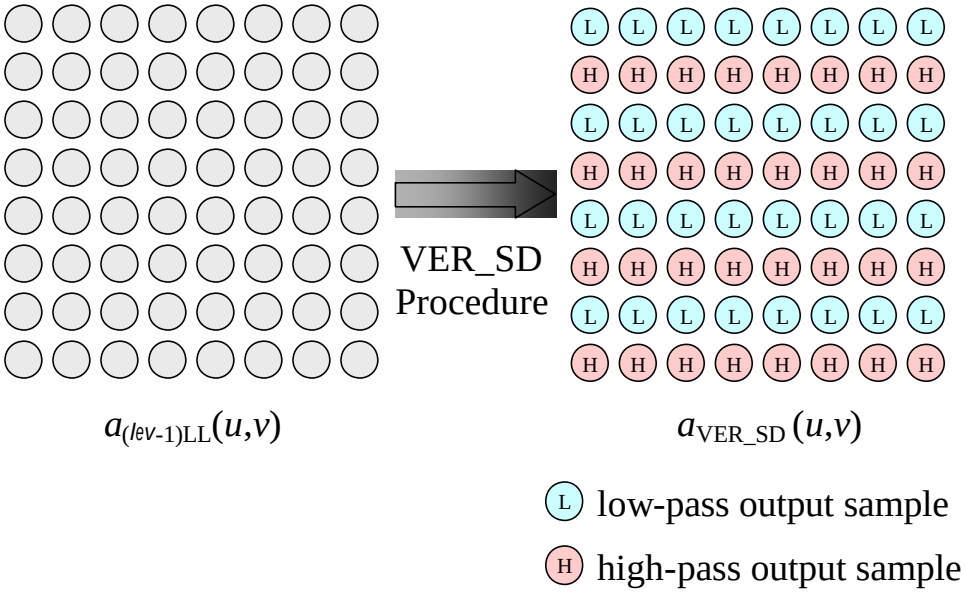

Figure 2.8. Application of the VER_SD procedure

2.3.3.1.1.3 The HOR_SD Procedure (Annex F.4.4)

The HOR_SD procedure performs wavelet transformation processing in the row direction. It takes as its input the array, a_{VER_SD}, which has been generated by the VER_SD procedure, and the coordinate bounds of the parent subband u_0, u_1, v_0, and v_1. The procedure produces as its output a new array a, which is the row filtered version of a_{VER_SD}. Each row of a_{VER_SD} is extracted, filtered using the 1D_SD procedure, and then placed back into the array, a. The internal calls made by the HOR_SD procedure are shown in Figure F-26 of ISO/IEC IS 15444-1.

Figure 2.9 illustrates the interleaving effects of the HOR_SD procedure. In this figure it has been assumed that the top left corner of a_{VER_SD} lies at an even index (u_0 is even). Furthermore it has been assumed that the number of rows in a_{VER_SD} is even. If u_0 is an odd index, then the wavelet transformation will generate a high-pass output sample first.

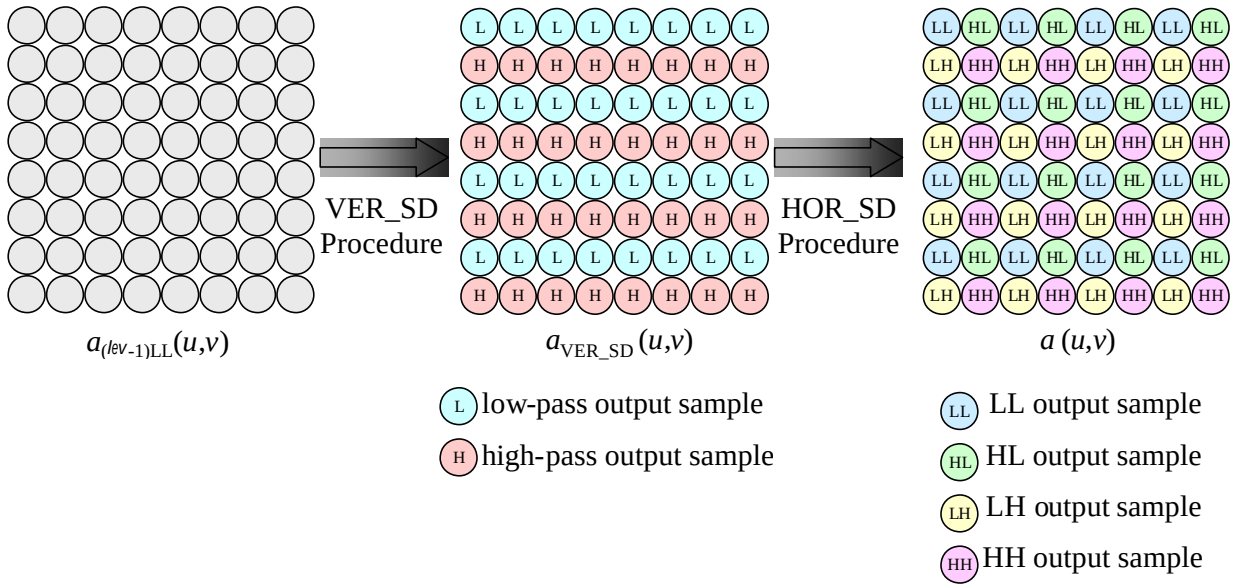

Figure 2.9. Application of the HOR_SD procedure

2.3.3.1.1.4 The 2D_DEINTERLEAVE Procedure (Annex F.4.5)

The 2D_DEINTERLEAVE procedure takes the output from the HOR_SD procedure and separates the interleaved wavelet subband coefficients in array, a, into four distinct subbands, $a_{lev\mathrm{LL}}$, $a_{lev\mathrm{HL}}$, $a_{lev\mathrm{LH}}$, and $a_{lev\mathrm{HH}}$. It takes as its inputs a, and the parent subband dimensions, u_0, u_1, v_0, and v_1, and outputs the above wavelet subbands. Although the flowchart for this procedure looks very complex (see ISO/IEC IS 15444-1 Figure F-28), it does nothing more than picks apart the interleaved data into four new arrays. Depending upon the even/odd nature of u_0 and v_0, the top left corner of the array a can belong to any of the four subbands LL, HL, LH, or HH. Figure 2.9 has been drawn assuming that the coordinates u_0 and v_0 are both even.

2.3.3.1.1.5 The 1D_SD Procedure (Annex F.4.6)

The 1D_SD procedure calls the procedures that perform the wavelet transformation. It takes as input a one-dimensional array, *X(i)*, with coefficient extent defined by $i_0 \le i < i_1$. Note i_0 and i_1 are set either to v_0 and v_1 or u_0 and u_1 depending upon the calling routine, VER_SD or HOR_SD respectively. The 1D_SD procedure produces as its output an array, *Y(i)*, with the same index range $[i_0, i_1\text{-}1]$. In the event that the input array has a length of one, the 1D_SD procedure sets $Y(i_0) = X(i_0)$ if i_0 is even and $Y(i_0) = 2X(i_0)$ if i_0 is odd. For this special case, no further processing is performed (1D_EXTD and 1D_FILTD are not called).

For array lengths greater than one, 1D_SD uses the 1D_EXTD procedure to symmetrically extend the input *X(i)* to create $X_{ext}(i)$. Once $X_{ext}(i)$ has been formed, 1D_SD calls the 1D_FILTD procedure to perform the wavelet transformation and to create the output array *Y(i)*.

2.3.3.1.1.6 The 1D_EXTD Procedure (Annex F.4.7)

The 1D_EXTD procedure is identical to the 1D_EXTR procedure (see Annex F.3.7 and Section 2.3.3.1.2.6 of this document). The 1D_EXTD procedure takes as its input the one-dimensional array of samples *X(i)*, and i_0, and i_1, which define the extent of *X(i)*. The output of the procedure is $X_{ext}(i)$, the symmetrically extended version of *X(i)*. The number of samples that must be extended on the left and right sides of the array X(i) is given in Tables F-8 and

F-9 of ISO/IEC IS 15444-1. For visually lossless compressed imagery, we use the columns labeled "$i_{left_{9-7}}$" and "$i_{right_{9-7}}$" in Tables F-8 and F-9 of ISO/IEC IS 15444-1, because we are using the 9-7I wavelet transformation. For numerically lossless compressed imagery, we use the columns labeled "$i_{left_{5-3}}$" and "$i_{right_{5-3}}$"in Tables F-8 and F-9 of ISO/IEC IS 15444-1, because we are using the 5-3R wavelet transformation.

The type of symmetric extension applied to *X(i)* is known as "whole-sample" symmetric extension. In whole-sample symmetric extension, the array data is reflected around its endpoints without repetition of the endpoints. This is illustrated in Figure F-15 of ISO/IEC IS 15444-1 and in the figures of section 2.3.3.1.1.8 of this document.

2.3.3.1.1.7 The 1D_FILTD Procedure (Annex F.4.8, Annex F.4.8.1, and Annex F.4.8.2)

The 1D_FILTD procedure performs the lifting wavelet transform. It takes as its input the one-dimensional array $X_{ext}(i)$ and creates the filtered output *Y(i)*. Additionally, the variables i_0 and i_1 are input to the procedure. These variables define the index extent of *Y(i)* (which is the same as that of *X(i)*) and indirectly the index extent of $X_{ext}(i)$ (via Tables F-8 and F-9).

2.3.3.1.1.7.1 9-7I Wavelet

There are four lifting steps and two scaling steps in the forward 9-7I wavelet transformation. These steps are described in detail in Annex F.4.8.2 and are shown in signal flow graph form in Figure 2.10. Steps 1 – 4 in Figure 2.10 are the lifting steps and Steps 5 – 6 are the scaling steps. As can be seen from the figure, the lifting steps alternately predict and update the even and odd samples in the array *Y(i)*. The steps that operate on odd samples are indicated by "*Y(2n+1)*" and those that operate on the even samples are indicated by "*Y(2n)*". The range of the variable *n* is also indicated for each step. Annex F.4.8.2 describes in detail how to determine the range of *n* given i_0 and i_1. The lifting coefficients, α, β, γ, δ, and *K* are given in Table F-4 of ISO/IEC IS 15444-1.

The output array, *Y(i)*, contains low-pass and high-pass samples in interleaved form. Whether or not the first and last samples are low-pass or high-pass is a function of i_0 and i_1. If i_0 is even, the first sample in *Y(i)* is low-pass (low-pass first). If i_0 is odd, the first sample is high-pass (high-pass first). The last sample in *Y(i)* is low-pass or high-pass depending upon whether i_0 is even or odd and the length of *Y(i)* ($i_1 - i_0$). Figure 2.11 shows the application of the 1D_SD procedure (including 1D_EXTD and 1D_FILTD) to an eight-sample array *X(i)*. All intermediate computations for all lifting steps are shown.

2.3.3.1.1.7.2 5-3R Wavelet

There are two lifting steps in the forward 5-3R wavelet transformation; they are described in detail in Annex F.4.8.1 and shown in signal flow graph form in Figure 2.12. The lifting steps and range of the variable *n* are given in Equation 2.11. The 5-3R lifting steps differ from those of the 9-7I in one important way – rounding is performed in each lifting step. Given integer input data, the 5-3R wavelet will produce integer output data. The 5-3R wavelet may only be used with integer input data. As was the case with the 9-7I wavelet, the 5-3R lifting steps alternately predict and update the even and odd samples in the output array *Y(i)*.

The output array, *Y(i)*, contains low-pass and high-pass samples in interleaved form. If i_0 is even, the first sample in *Y(i)* is low-pass (low-pass first). If i_0 is odd, the first sample is high-pass (high-pass first). The last sample in *Y(i)* is low-pass or high-pass depending upon whether i_0 is even or odd and the length of *Y(i)* ($i_1 - i_0$). Figure 2.13 shows the application of the 1D_SD procedure (including 1D_EXTD and 1D_FILTD) to an eight-sample array *X(i)*. All intermediate computations for all lifting steps are shown.

2.3.3.1.1.8 FDWT Processing and Examples

In this section we offer further insight into the forward wavelet transformation processing for both the 9-7I and 5-3R filters. We also include an example one-level one-dimensional analysis of a signal to illustrate the intermediate lifting computations in both wavelet transformations.

2.3.3.1.1.8.1 9-7I Wavelet

Figure 2.10 illustrates application of the 1D_EXTD procedure for the 9-7I wavelet. In this figure an eight-sample input vector with $i_0 = 0$ and $i_1 = 8$, is first symmetrically extended to form $X_{ext}(i)$ using the 1D_EXTD procedure. Both i_0 and i_1 are even, so in accordance with Tables F-8 and F-9 ISO/IEC IS 15444-1, the array $X(i)$ has been extended by four samples on the left and three samples on the right in the formation of array $X_{ext}(i)$. Equation F.4 of ISO/IEC IS 15444-1 can be used to map the index range of $X_{ext}(i)$ back into that of $X(i)$. For example, if we want to map $i = -3$ back into the original range of $X(i)$, Equation F.4 yields a value of 3. This means that $X_{ext}(-3) = X(3)$, which is exactly what we see in Figure 2.10. The "square" samples in $X_{ext}(i)$ represent the symmetrically extended samples from $X(i)$. These samples have been copied from the locations in $X(i)$ with matching letters. Once $X_{ext}(i)$ has been formed, it may be processed with the wavelet transform.

Steps 1 through 6 in Figure 2.10 illustrate the lifting implementation of the forward 9-7I wavelet transformation in signal flow graph form, for an input length of eight samples ($i_0 = 0$, $i_1 = 8$). Each line represents multiplication of a sample by a number (for example α, β in the figure). If a line has no figure next to it, the sample is passed through (multiplication by 1). Locations where lines meet represent a summation of all pre-multiplied sample values entering the summation. For example, the first sample in the row beneath $X_{ext}(i)$ is $Y(-3) = X_{ext}(-3) + \alpha(X_{ext}(-4) + X_{ext}(-2))$.

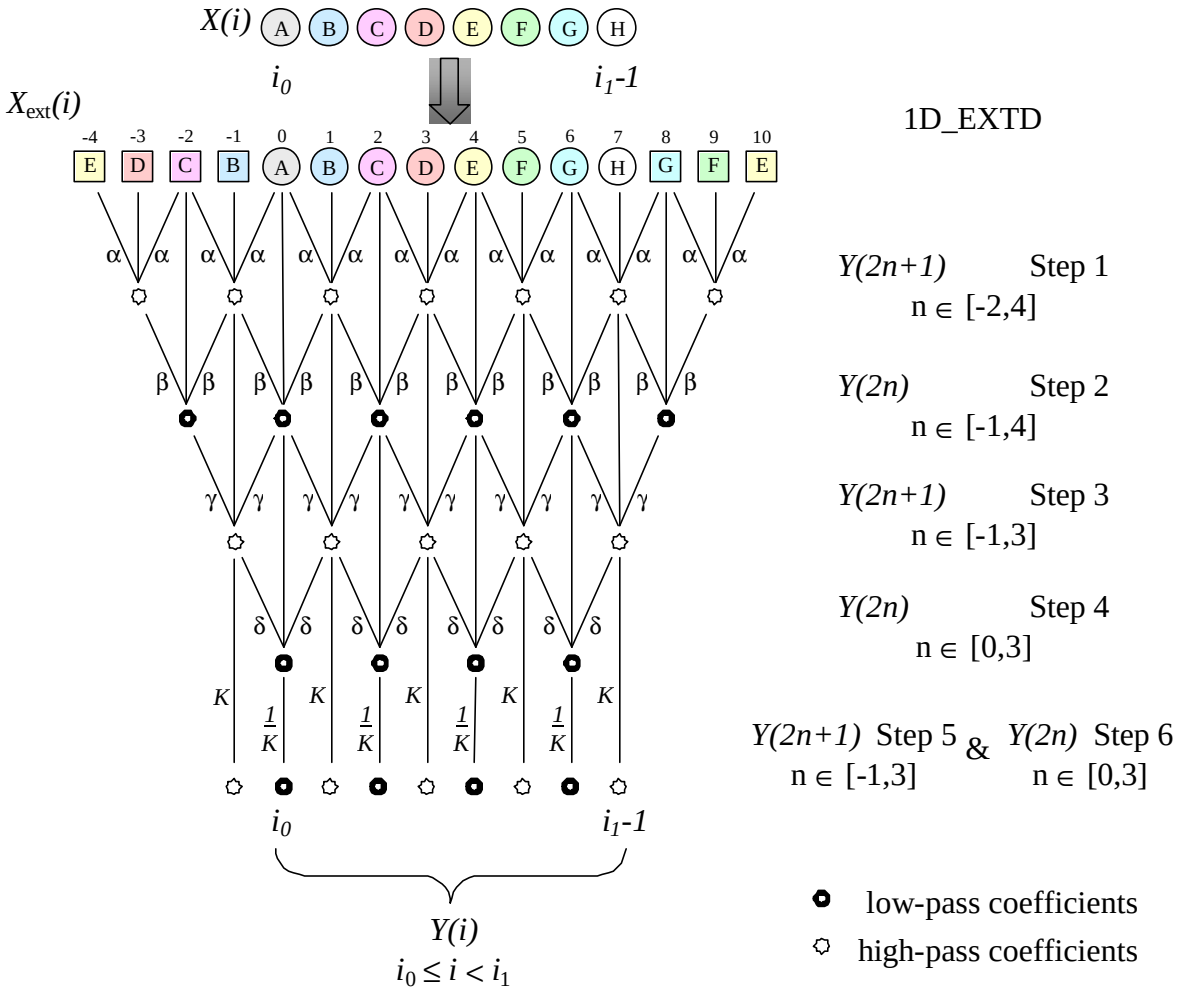

Figure 2.10. Application of the 1D_SD procedure (9-7I wavelet)

Figure 2.11 shows an example applying the 1D_SD procedure to an eight-sample signal, $X(i)$. This figure is presented in the same signal flow graph form as that given in Figure 2.10. The intermediate computations of the

lifting steps are shown and the resulting interleaved output signal, *Y(i)*, is shown in the bottom line of the figure. Given that i_0 and i_1 are both even, the first sample in *Y(i)* is a low-pass sample and the last is a high-pass sample. These calculations can be easily reproduced in a spreadsheet to validate proper understanding of the 9-7I wavelet filtering procedure.

alpha :	-1.5861	gamma :	0.88291														
beta :	-0.053	delta :	0.44351														
K :	1.23017																
X(i)						8	2	4	1	6	9	11	3				
Xext(i)	6	1	4	2	8	2	4	1	6	9	11	3	11	9	6		
Step 1		-14.861		-17.034		-17.034		-14.861		-17.964		-31.895		-17.964			
Step 2			5.6898		9.80489		5.6898		7.73911		13.6415		13.6415				
Step 3				-3.3532		-3.3532		-3.0048		0.91293		-7.8064					
Step 4					6.83057		2.86998		6.81134		10.5842						
Step 5 & 6				-4.125	5.55252	-4.125	2.33299	-3.6964	5.53689	1.12307	8.60386	-9.6032					
Y(i)					5.55252	-4.125	2.33299	-3.6964	5.53689	1.12307	8.60386	-9.6032					

Figure 2.11. Example of the 1D_SD procedure (9-7I wavelet)

2.3.3.1.1.8.2 5-3R Wavelet

Figure 2.12 illustrates application of the 1D_EXTD procedure for the 5-3R wavelet. As was shown in Figure 2.10, we assume an eight-sample input signal, *X(i)*. Again both i_0 and i_1 are even, so in accordance with Tables F-8 and F-9 ISO/IEC IS 15444-1, the array *X(i)* has been extended by two samples on the left and one sample on the right in the formation of array $X_{ext}(i)$. Equation F.4 of ISO/IEC IS 15444-1 still applies and can be used to map the index range of $X_{ext}(i)$ back into that of *X(i)*. Thus there is no difference in the symmetric extension procedures between the 5-3R and 9-7I wavelet transformations other than the number of samples that must extended. This difference simply results from the fact that the wavelet filters have different lengths. The "square" samples in $X_{ext}(i)$ represent the symmetrically extended samples from *X(i)*. These samples have been copied from the locations in *X(i)* with matching letters. Once $X_{ext}(i)$ has been formed it may be processed with the wavelet transform.

Steps 1 and 2 in Figure 2.12 illustrate the lifting implementation of the forward 5-3R wavelet transformation in signal flow graph form, for an input length of eight samples ($i_0 = 0$, $i_1 = 8$). Each line represents multiplication of a sample by a number (for example 1/2, 1/4 in the figure); if a line has no value next to it, the sample is passed through (multiplication by 1). Locations where lines meet represent a summation. Negative multipliers indicate summations that involve a subtraction. In order to achieve reversibility with the integer coefficients of the 5-3R wavelet, rounding must be applied in a very specific manner during the calculation (see Equation 2.11). Dashed lines in the flow graph indicate the need to follow the special rounding rules instead of performing a simple multiplication and summation.

Equation 2.11 shows the lifting steps for the 5-3R wavelet and the range of variable, *n*, given the index range of *X(i)*, $[i_0, i_1)$. The rounding operations associated with the dashed lines in the signal flow graph of Figure 2.12 are readily apparent in Equation 2.11.

$$\text{Step 1:}\quad Y(2n+1) = X_{ext}(2n+1) - \left\lfloor \frac{X_{ext}(2n) + X_{ext}(2n+2)}{2} \right\rfloor \qquad \left\lceil \frac{i_0}{2} \right\rceil - 1 \le n < \left\lceil \frac{i_1}{2} \right\rceil$$

$$\text{Step 2:}\quad Y(2n) = X_{ext}(2n) + \left\lfloor \frac{Y(2n-1) + Y(2n+1) + 2}{4} \right\rfloor \qquad \left\lceil \frac{i_0}{2} \right\rceil \le n < \left\lceil \frac{i_1}{2} \right\rceil$$

Equation 2.11

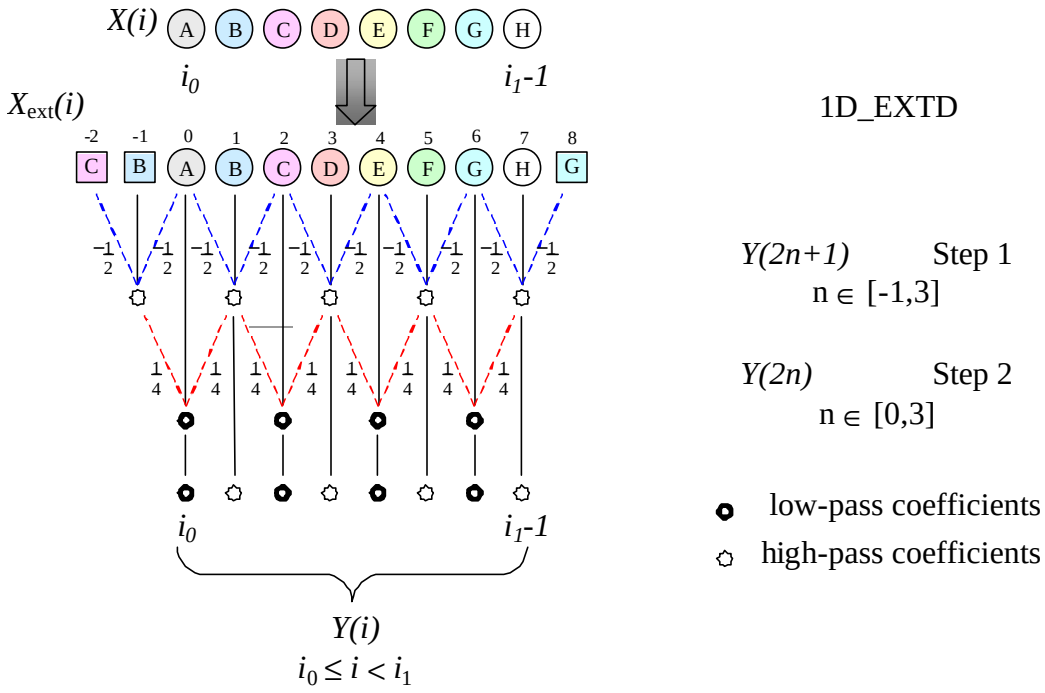

Figure 2.12. Application of the 1D_SD procedure (5-3R wavelet)

Figure 2.13 shows an example applying the 1D_SD procedure to the eight-sample signal, *X(i)*. This is the same eight-sample signal used in the 9-7I example in Figure 2.11. Figure 2.13 is presented in the same signal flow graph form as that given in Figure 2.12. The intermediate computations of the lifting steps are shown and the resulting interleaved output signal, *Y(i)*, is shown in the bottom line of the figure. Given that i_0 and i_1 are both even, the first sample in *Y(i)* is a low-pass sample and the last is a high-pass sample. These calculations can be easily reproduced in a spreadsheet to validate proper understanding of the 5-3R wavelet filtering procedure. Looking at Figure 2.13, we notice that the output of the 5-3R filtering, *Y(i)*, as well as all intermediate lifting steps, are integers. Since the 5-3R transformation is a one-to-one mapping, the 5-3R wavelet allows for lossless reconstruction of integer data.

X(i)			8	2	4	1	6	9	11	3	
Xext(i)	4	2	8	2	4	1	6	9	11	3	11
Step 1		-4		-4		-4		1		-8	
Step 2			6		2		5		9		
Y(i)			6	-4	2	-4	5	1	9	-8	

Figure 2.13. Example of the 1D_SD procedure (5-3R wavelet)

2.3.3.1.2 Inverse DWT Processing (IDWT) (Annex F.3)

The inverse wavelet transformation is normatively described in Annex F.3 of ISO/IEC IS 15444-1. For a *decoder*, the procedures described in Annex F.3 are normative. However, for the recommended compressor, only the previously described forward wavelet transform procedure is normative. Section 2.3.3.1.2 is included here strictly as an informative section.

As was the case for the forward wavelet transformation, a separable one-dimensional wavelet transform in the row and columnar directions is used to implement the two-dimensional inverse wavelet transformation. The IDWT takes as its input a set of subbands, $a_b(u_b,v_b)$, and transforms them into DC-level shifted tile samples *I(x,y)*. The IDWT

also takes as an input the number of decomposition levels, N_L. The decoder determines the number of decomposition levels by reading the applicable COD/COC marker segment for the current tile. Figure 2.14 illustrates the wavelet subband reconstruction from an inverse two level wavelet transformation ($N_L = 2$).

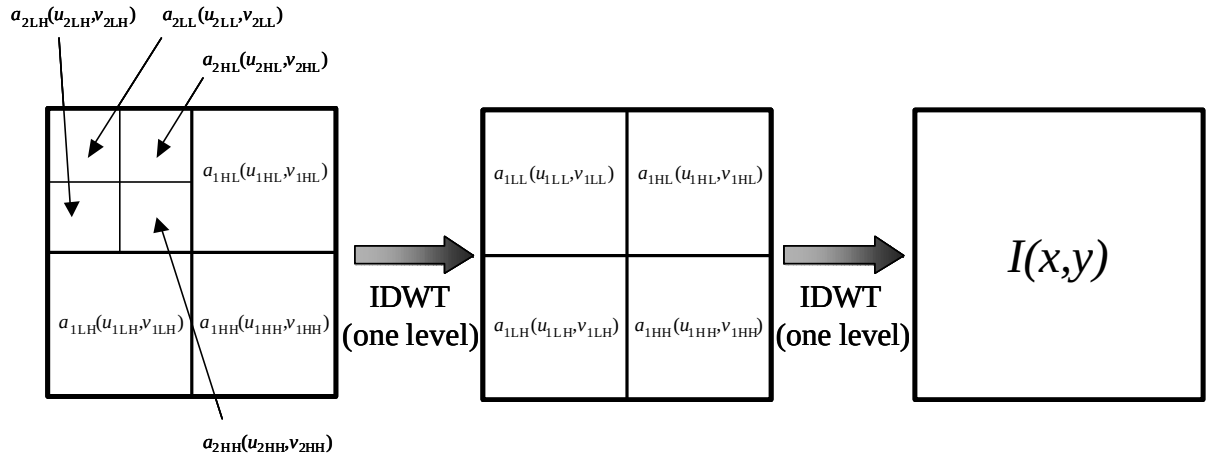

Figure 2.14. Application of the IDWT (N_L = 2)

Figure F-3 in ISO/IEC IS 15444-1 shows the main loop of the IDWT procedure. Each pass through the loop calls the procedure 2D_SR, which performs the reconstruction or *synthesis* of subband $a_{(lev-1)LL}(u,v)$ from the four subbands, $a_{levLL}(u,v)$, $a_{levHL}(u,v)$, $a_{levLH}(u,v)$, and $a_{levHH}(u,v)$. This is illustrated in Figure 2.14. Thus, the IDWT works in a recursive manner undoing the operations performed by the FDWT procedure in reverse order. As was the case with the FDWT, the IDWT is a non-expansive transform. The nomenclature and terminology introduced in Section 2.3.3.1.1 for the FDWT apply to the IDWT as well. The notions of resolution level, decomposition level, subband orientation, and computation of subband ranges for u and v are the same.

2.3.3.1.2.1 The 2D_SR Procedure (Annex F.3.2)

The IDWT repeatedly calls the 2D_SR procedure to perform the inverse wavelet transformation (see Figure F-3). The IDWT procedure is initialized by setting the variable $lev = N_L$. The 2D_SR procedure is then called once for each decomposition level. The call to 2D_SR is of the form $a_{(lev-1)LL}$ = 2D_SR(a_{levLL},a_{levHL},a_{levLH},a_{levHH},u_0,u_1,v_0,v_1). The output LL subband generated from the current call to 2D_SR is fed back into the next call to the 2D_SR procedure. The inverse wavelet transformation is non-expansive, so the total number of wavelet coefficients in the reconstructed parent subband, $a_{(lev-1)LL}$, is equal to the number of wavelet coefficients in the child subbands (a_{levLL},a_{levHL},a_{levLH},a_{levHH}). The inputs, u_0, u_1, v_0, and v_1, are the values tbx_0, tbx_1, tby_0, and tby_1, corresponding to subband b = (lev-1)LL.

The internal calls made by the 2D_SR procedure are shown in Figure F-6 of ISO/IEC IS 15444-1. The first call is to the procedure 2D_INTERLEAVE, which takes the four subbands (a_{levLL},a_{levHL},a_{levLH},a_{levHH}) and interleaves them to form the two-dimensional array a. Next, the HOR_SR procedure is called to synthesize the rows of the array a and reconstruct the array a_{VER_SD}. Finally, the VER_SR procedure is called which reconstructs the array $a_{(lev-1)LL}$ from a_{VER_SD}. Note that the processing order of the IDWT procedure is exactly the opposite of that in FDWT.

2.3.3.1.2.2 The 2D_INTERLEAVE Procedure (Annex F.3.3)

The 2D_INTERLEAVE procedure takes as input the four child subbands, $a_{lev\text{LL}}$, $a_{lev\text{HL}}$, $a_{lev\text{LH}}$, and $a_{lev\text{HH}}$ and interleaves them to form the array, a, prior to inverse wavelet transformation processing. In addition to the four child subbands, the parent subband dimensions, u_0, u_1, v_0, and v_1, are inputs to the procedure. The flowchart for this procedure is given ISO/IEC IS 154444-1 Figure F-9. Although this flowchart looks complex, the 2D_INTERLEAVE procedure does nothing more than arrange and place subband samples within the matrix, a. It is the inverse of the 2D_DEINTERLEAVE procedure described in section 2.3.3.1.1.4. Depending upon the even/odd nature of u_0 and v_0, the top left corner of the array a can belong to any of the four subbands LL, HL, LH, or HH. Figure 2.15 (see below) has been drawn assuming that the coordinates u_0 and v_0 are both even.

2.3.3.1.2.3 The HOR_SR Procedure (Annex F.3.4)

The HOR_SR procedure performs inverse wavelet transformation processing in the row direction. It takes as its input the array, a, and reconstructs the array $a_{\text{VER\_SD}}$, as its output. For clarity's sake we have once again adopted the intermediate variable, $a_{\text{VER\_SD}}$, to distinguish the intermediate output of the HOR_SR procedure from input array, a. The coordinate bounds, u_0, u_1, v_0, and v_1 of the parent subband, $a_{(lev\text{-}1)\text{LL}}$ (and array a), also form inputs to the HOR_SR procedure. Each row of the array, a, is extracted and filtered using the 1D_SR procedure and then placed back into the array, a. The internal calls made by the HOR_SD procedure are shown in Figure F-10 of ISO/IEC IS 15444-1. Figure 2.15 illustrates the deinterleaving effects of the HOR_SR and VER_SR procedures. In this figure it has been assumed that the top left corner of a lies at an even index (u_0 and v_0 are even). Furthermore, it has been assumed that the number of rows and columns in a are even.

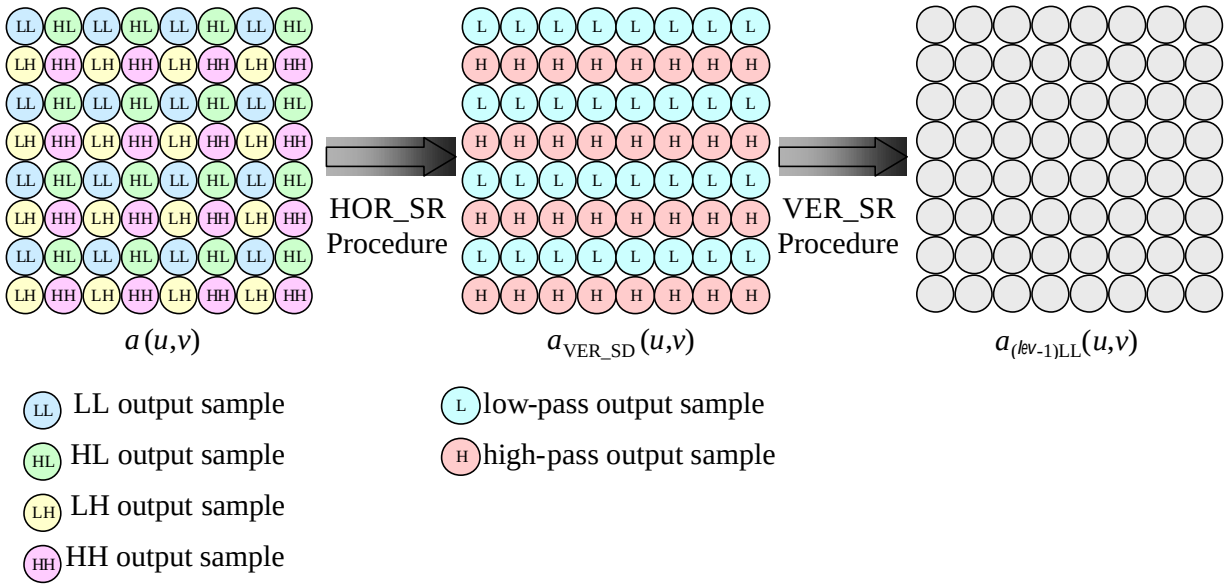

Figure 2.15. Application of the HOR_SR and VER_SR procedures

2.3.3.1.2.4 The VER_SR Procedure (Annex F.3.5)

The VER_SR procedure performs inverse wavelet transformation processing in the columnar direction. It takes as its input $a_{\text{VER\_SD}}$ and the coordinate bounds of the parent subband u_0, u_1, v_0, and v_1. The procedure produces as its output the array $a_{(lev\text{-}1)\text{LL}}$, the reconstructed parent subband. The internal calls made by the VER_SD procedure are shown in Figure F-12 of ISO/IEC IS 15444-1. Each column of $a_{\text{VER\_SD}}$ is extracted, filtered using the 1D_SR procedure (Annex F.3.8 and Section 2.3.3.1.2.5 of this document), and then placed back into the array, a. Figure 2.15 illustrates the deinterleaving effects of the VER_SD procedure. In this figure it has been assumed that the top

left corner of a_{VER_SD} lies at an even index (u_0 and v_0 are even). Furthermore it has been assumed that the number of rows and columns in a_{VER_SD} is even.

2.3.3.1.2.5 The 1D_SR Procedure (Annex F.3.6)

The 1D_SR procedure calls the procedures that perform the inverse wavelet transformation. It takes as input a one-dimensional array, *Y(i)*, with coefficient extent defined by $i_0 \le i < i_1$. Note i_0 and i_1 are set either to u_0 and u_1 or v_0 and v_1 depending upon the calling routine, HOR_SR or VER_SD respectively. The 1D_SR procedure produces as its output an array, *X(i)*, with the same index range [i_0, i_1-1]. In the event that the input array has a length of one, the 1D_SR procedure sets $X(i_0) = Y(i_0)$ if i_0 is even and $X(i_0) = Y(i_0)/2$ if i_0 is odd. For this special case, no further processing is performed (1D_EXTR and 1D_FILTR are not called).

For array lengths greater than one, 1D_SR uses the 1D_EXTR procedure to symmetrically extend the input *Y(i)* to create $Y_{ext}(i)$. Once $Y_{ext}(i)$ has been formed, 1D_SR calls the 1D_FILTR procedure to perform the wavelet transformation and create the output array *X(i)*.

2.3.3.1.2.6 The 1D_EXTR Procedure (Annex F.3.7)

The 1D_EXTR procedure is identical to the 1D_EXTD procedure (see Annex F.4.7 and Section 2.3.3.1.1.6 of this document). The 1D_EXTR procedure takes as its input the one-dimensional array of samples *Y(i)*, and i_0, and i_1, which define the extent of *Y(i)*. The output of the procedure is $Y_{ext}(i)$, the symmetrically extended version of *Y(i)*. The number of samples that must be extended on the left and right sides of the array Y(i) is given in Tables F-2 and F-3 of ISO/IEC IS 15444-1. Visually lossless compressed imagery uses the 9-7I wavelet transformation and the columns "$i_{left_{9-7}}$" and "$i_{right_{9-7}}$" in these tables. Numerically lossless compressed imagery uses the 5-3R wavelet transformation and the "$i_{left_{5-3}}$" and "$i_{right_{5-3}}$" columns in Tables F-2 and F-3 of ISO/IEC IS 15444-1.

The type of symmetric extension applied to *Y(i)* is known as "whole-sample" symmetric extension. In whole-sample symmetric extension, the array data is reflected around its endpoints without repetition of the endpoints. This is illustrated in Figure F-15 of ISO/IEC IS 15444-1 and in the figures of section 2.3.3.1.2.8 of this document.

2.3.3.1.2.7 The 1D_FILTR Procedure (Annex F.3.8, Annex F.3.8.1, and Annex F.3.8.2)

The 1D_FILTR procedure performs the inverse lifting wavelet transform. It takes as its input the one-dimensional array $Y_{ext}(i)$ and creates the filtered output *X(i)*. Additionally, the variables i_0 and i_1 are input to the procedure. These variables define the index extent of *X(i)* (which is the same as that of *Y(i)*) and indirectly the index extent of $Y_{ext}(i)$ (via Tables F-2 and F-3).

2.3.3.1.2.7.1 9-7I Wavelet

There are two scaling steps and four lifting steps in the inverse 9-7I wavelet transformation. These steps are described in detail in Annex F.3.8.2 and shown in signal flow graph form in Figure 2.16. Steps 1 – 2 are the scaling steps, and Steps 3 – 6 are the lifting steps. As can be seen from the figure, the lifting steps alternately predict and update the even and odd samples in the array *X(i)*. The steps that operate on odd samples are indicated by "*X(2n+1)*" and those that operate on the even samples are indicated by "*X(2n)*". The range of the variable *n* is also indicated for each step. Annex F.3.8.2 describes in detail how to determine the range of *n* given i_0 and i_1. The lifting coefficients, α, β, γ, δ, and *K* are given in Table F-4 of ISO/IEC IS 15444-1.

Note - There is an error in the formula for the range of *n* for Step 2 in the 1D_FILTR procedure in ISO/IEC FDIS15444-1 (dated 5 July 2001). This error has been corrected in ISO/IEC FDIS15444-1 (dated 28 September 2001). The proper formula is repeated here:

$$\left\lfloor \frac{i_0}{2} \right\rfloor - 2 \le n < \left\lfloor \frac{i_1}{2} \right\rfloor + 2$$

The input array, *Y(i)*, contains low-pass and high-pass samples in interleaved form as generated by the 2D_INTERLEAVE procedure (section 2.3.3.1.2.2). The filtered output array, *X(i)*, generated by the 1D_FILTR procedure is not interleaved due to the wavelet synthesis processing. Figure 2.17 shows the application of the

1D_SR procedure (including 1D_EXTR and 1D_FILTR) to an eight-sample array *Y(i).* This is the same eight-sample output array, *Y(i),* generated in Figure 2.11 using the 1D_SD procedure.

2.3.3.1.2.7.2 5-3R Wavelet

There are two lifting steps in the inverse 5-3R wavelet transformation; they are described in detail in Annex F.3.8.1 and shown in signal flow graph form in Figure 2.18. The lifting steps and range of the variable *n* are given in Equation 2.12. The 5-3R lifting steps differ from those of the 9-7I in one important way – rounding is performed in each lifting step. Given integer input data, the 5-3R wavelet will produce integer output data. The 5-3R wavelet may only be used with integer input data. As was the case with the 9-7I wavelet, the 5-3R lifting steps alternately predict and update the even and odd samples in the output array *X(i).*

The input array, *Y(i),* contains low-pass and high-pass samples in interleaved form as generated by the 2D_INTERLEAVE procedure (section 2.3.3.1.2.2). The filtered output array, *X(i),* generated by the 1D_FILTR procedure is not interleaved due to the wavelet synthesis processing. Figure 2.19 shows the application of the 1D_SR procedure (including 1D_EXTR and 1D_FILTR) to an eight-sample array *Y(i).* This is the same eight-sample output array, *Y(i),* generated in Figure 2.13 using the 1D_SD procedure.

2.3.3.1.2.7.3 The 1D_FILTR$_{9\text{-}7I}$ Parameters (Annex F.3.8.2.1)

Table F-4 through Table F-7 give the 9-7I wavelet lifting coefficients as well as precise mathematical expressions to enable computation of the filter coefficients to any precision desired. While the JPEG 2000 standard does not mandate any precision requirements, for the recommended processing the coefficients in Table F-4 are of sufficient precision.

2.3.3.1.2.8 IDWT Processing and Examples

In this section we offer further insight into the inverse wavelet transformation processing for both the 9-7I and 5-3R filters. An example one-level one-dimensional synthesis of a signal is included to illustrate the intermediate lifting computations in both wavelet transformations. The signal synthesized in the following examples is the output signal generated by the examples from section 2.3.3.1.1.8. Thus the following examples also serve as a check on the reversibility of the 9-7I and 5-3R wavelet transformations.

2.3.3.1.2.8.1 9-7I Wavelet

Figure 2.16 illustrates application of the 1D_EXTR procedure for the 9-7I wavelet. It illustrates the inverse wavelet transformation processing and is the counterpart to Figure 2.10. In this figure, an eight-sample input vector with i_0 = 0 and i_1 = 8 is first symmetrically extended to form $Y_{ext}(i)$ using the 1D_EXTR procedure. Since i_0 and i_1 are even, the array *Y(i)* has been extended by three samples on the left and four samples on the right, according to Tables F-2 and F-3 of ISO/IEC IS 15444-1. The number of samples that must be extended is a function of the even/odd nature of i_0 and i_1. Equation F.4 of ISO/IEC IS 15444-1 can be used to map the index range of $Y_{ext}(i)$ back into that of *Y(i).* For example, if we want to map i = 10 back into the original range of *Y(i),* Equation F.4 yields a value of 4 (i.e. $Y_{ext}(10) = Y(4)$). Once $Y_{ext}(i)$ has been formed, it may be processed with the inverse wavelet transform.

Steps 1 through 6 in Figure 2.16 illustrate the lifting implementation of the inverse 9-7I wavelet transformation in signal flow graph form, for an input length of eight samples (i_0 = 0, i_1 = 8). Each line represents multiplication of a sample by a number (for example γ, δ in the figure). If a line has no figure next to it then the sample is passed through (multiplication by 1). Locations where lines meet represent a summation of all pre-multiplied sample values entering the summation. For example, the first sample output from step 3 is: $X(-2) = X(-2) - \delta(X(-4) + X(-2))$.

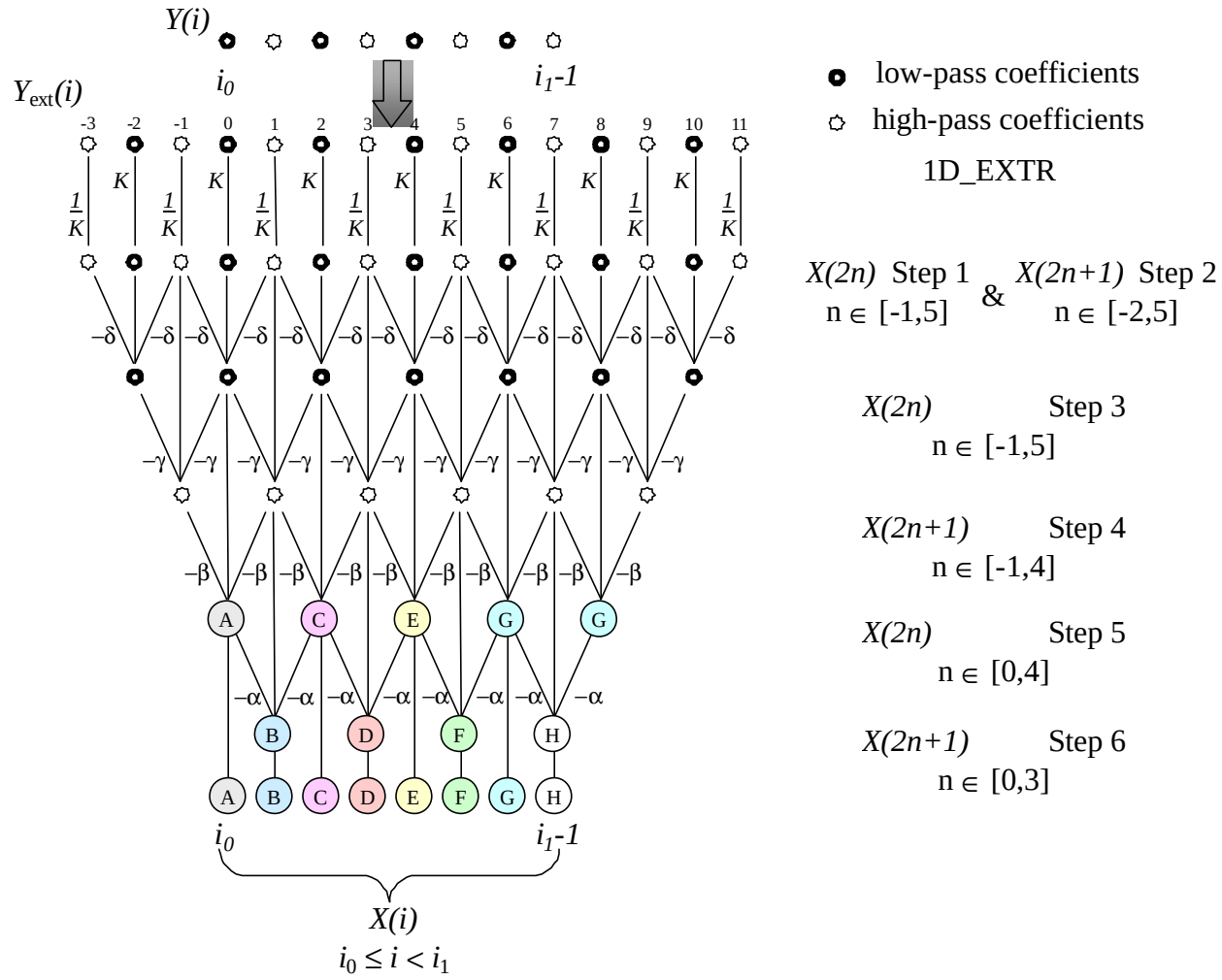

Figure 2.16. Application of the 1D_SR procedure (9-7I wavelet)

Figure 2.17 shows the application of the 1D_SR procedure (including 1D_EXTR and 1D_FILTR) to the eight-sample array, *Y(i)*, the output array from Figure 2.11. Thus we should reconstruct the original array, *X(i)*, from Figure 2.11 since we are undoing the forward wavelet transformation performed there. In this example we have set $i_0 = 0$ and $i_1 = 8$. All intermediate computations for all lifting steps are shown. As can be seen, the original signal is reconstructed to within the precision of the lifting coefficients (16 digits in Table F-4).

alpha :	-1.5861	gamma :	0.88291													
beta :	-0.053	delta :	0.44351													
K :	1.23017															
Y(i)				5.55252	-4.125	2.33299	-3.6964	5.53689	1.12307	8.60386	-9.6032					
Yext(i)	-3.6964	2.33299	-4.125	5.55252	-4.125	2.33299	-3.6964	5.53689	1.12307	8.60386	-9.6032	8.60386	1.12307	5.53689	-3.6964	
Step 1 &	-3.0048	2.86998	-3.3532	6.83057	-3.3532	2.86998	-3.0048	6.81134	0.91293	10.5842	-7.8064	10.5842	0.91293	6.81134	-3.0048	
Step 3		5.6898		9.80489		5.6898		7.73911		13.6415		13.6415		7.73911		
Step 4			-17.034		-17.034		-14.861		-17.964		-31.895		-17.964			
Step 5				8		4		6		11		11				
Step 6					2		1		9		3					
X(i)				8	2	4	1	6	9	11	3					
Original				8	2	4	1	6	9	11	3					

Figure 2.17. Example of the 1D_SR procedure (9-7I wavelet)

2.3.3.1.2.8.2 5-3R Wavelet

Figure 2.18 illustrates application of the 1D_EXTR procedure for the 5-3R wavelet. It illustrates the inverse wavelet transformation processing and is the counterpart to Figure 2.12. In this figure, an eight-sample input vector with i_0 = 0 and i_1 = 8 is first symmetrically extended to form $Y_{ext}(i)$ using the 1D_EXTR procedure. Since i_0 and i_1 are even, the array $Y(i)$ has been extended by one sample on the left and two samples on the right, according to Tables F-2 and F-3 of ISO/IEC IS 15444-1. Equation F.4 of ISO/IEC IS 15444-1 still applies and can be used to map the index range of $Y_{ext}(i)$ back into that of $Y(i)$. Thus there is no difference in the symmetric extension procedures between forward and inverse transformations for either the 5-3R and 9-7I wavelet other than the number of samples that must extended. Once $Y_{ext}(i)$ has been formed, it may be processed with the wavelet transform.

Steps 1 and 2 in Figure 2.18 illustrate the lifting implementation of the inverse 5-3R wavelet transformation in signal flow graph form, for an input length of eight samples (i_0 = 0, i_1 = 8). Each line represents multiplication of a sample by a number (for example 1/2, 1/4 in the figure); if a line has no value next to it, the sample is passed through (multiplication by 1). Locations where lines meet represent a summation. Negative multipliers indicate summations that involve a subtraction. In order to achieve reversibility with the integer coefficients of the 5-3R wavelet, rounding must be applied in a very specific manner during the calculation (see Equation 2.12). Dashed lines in the flow graph indicate the need to follow the special rounding rules instead of performing a simple multiplication and summation.

Equation 2.12 shows the lifting steps for the 5-3R wavelet and the range of variable, n, given the index range of $Y(i)$, $[i_0, i_1)$. The rounding operations associated with the dashed lines in the signal flow graph of Figure 2.18 are readily apparent in Equation 2.12.

$$\text{Step 1:} \quad X(2n) = Y_{ext}(2n) - \left\lfloor \frac{Y_{ext}(2n-1) + Y_{ext}(2n+1) + 2}{4} \right\rfloor \qquad \left\lfloor \frac{i_0}{2} \right\rfloor \le n < \left\lfloor \frac{i_1}{2} \right\rfloor + 1$$

$$\text{Step 2:} \quad X(2n+1) = Y_{ext}(2n+1) + \left\lfloor \frac{X(2n) + X(2n+2)}{2} \right\rfloor \qquad \left\lfloor \frac{i_0}{2} \right\rfloor \le n < \left\lfloor \frac{i_1}{2} \right\rfloor$$

Equation 2.12

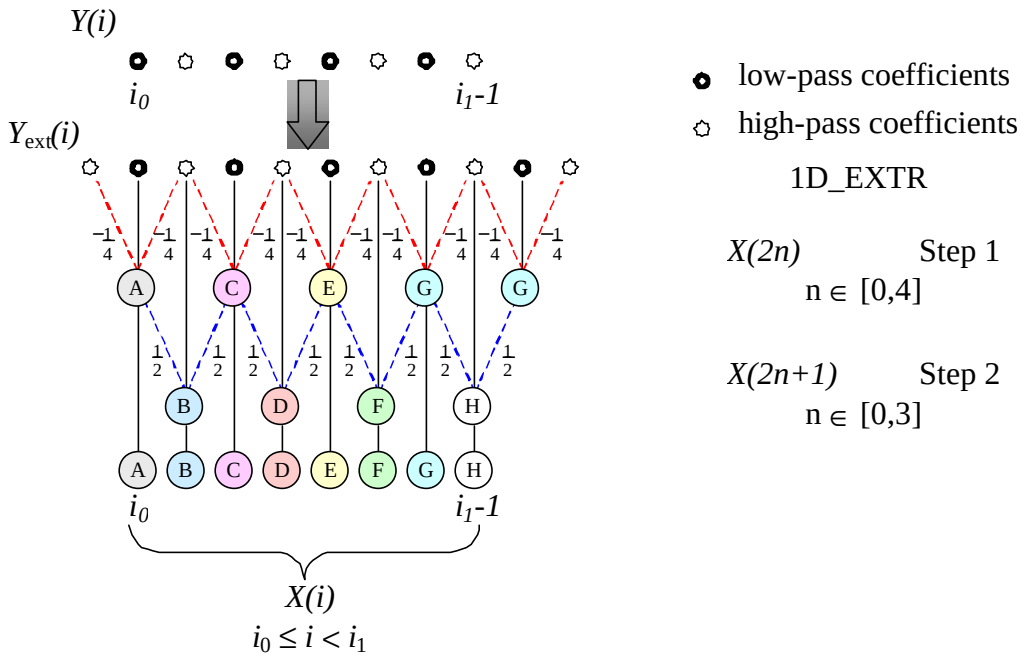

Figure 2.18. Application of the 1D_SR procedure (5-3R wavelet)

Figure 2.19 shows an example applying the 1D_SR procedure (including 1D_EXTR and 1D_FILTR) to the eight-sample array, *Y(i)*, the output array from Figure 2.13. Thus we should reconstruct the original array, *X(i)*, from Figure 2.13 since we are undoing the forward wavelet transformation performed there. Figure 2.19 is presented in the same signal flow graph form as that given in Figure 2.18. The intermediate computations of the lifting steps are shown. As can be seen, the original signal is reconstructed without any loss.

Y(i)		6	-4	2	-4	5	1	9	-8		
Yext(i)	-4	6	-4	2	-4	5	1	9	-8	9	1
Step 1		8		4		6		11		11	
Step 2			2		1		9		3		
X(i)		8	2	4	1	6	9	11	3		
Original		8	2	4	1	6	9	11	3		

Figure 2.19. Example of the 1D_SR procedure (5-3R wavelet)

2.3.3.1.3 Remarks

The purpose of this section is to give further guidance and understanding regarding the wavelet transformation procedures in ISO/IEC IS 15444-1. Some of this material is not normative in nature; it is intended to clarify and tie together concepts that are not clear in the standard. Some of this information will be critical in understanding implementation specific issues discussed in the quantization section of this document (Section 2.3.3.2). Note that not all of this material is applicable to both wavelet transformations. We will explicitly indicate in the text what sections apply to which wavelet transformation.

2.3.3.1.3.1 Convolution Equivalent Wavelet Filtering and Normalization

Most of this section is not applicable to the 5-3R integer wavelet transformation. There is no exact convolution equivalent to the lifting implementation of the 5-3R wavelet. This is due to the rounding non-linearities present in its lifting steps. The quantization procedures for the 5-3R wavelet are quite different than those of the 9-7I wavelet and we will not be concerned with determining convolution equivalent filters for the 5-3R wavelet. However, the normalization of the 5-3R wavelet is of importance in determination of the proper number of guard bits (see below).

Lifting implementations of wavelet transforms are a relatively new concept. Prior to their development, wavelets were implemented like any QMF or subband filter, by using convolution, decimation and interpolation. Figures Figure 2.20 and Figure 2.21 show the classical convolution processing steps involved in wavelet filtering prior to lifting implementations. In Figure 2.20 we see that prior to filtering with the low-pass and high-pass analysis filters a symmetrically extended sequence is generated from the input vector, *X(n)*. The symmetric extension that is performed is identical to that described for the lifting implementation presented in ISO/IEC IS 15444-1. After filtering is performed the low-pass and high-pass signals are decimated by a factor of two (every other sample thrown out). In the lifting implementation this happens naturally during the computation of the even and odd samples and deinterleaving.

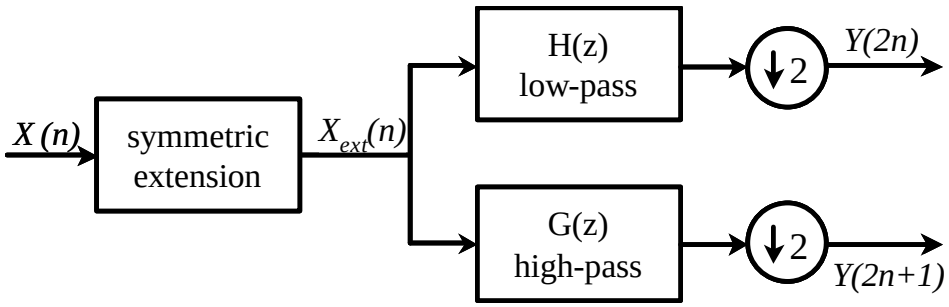

Figure 2.20. Wavelet analysis (convolution implementation)

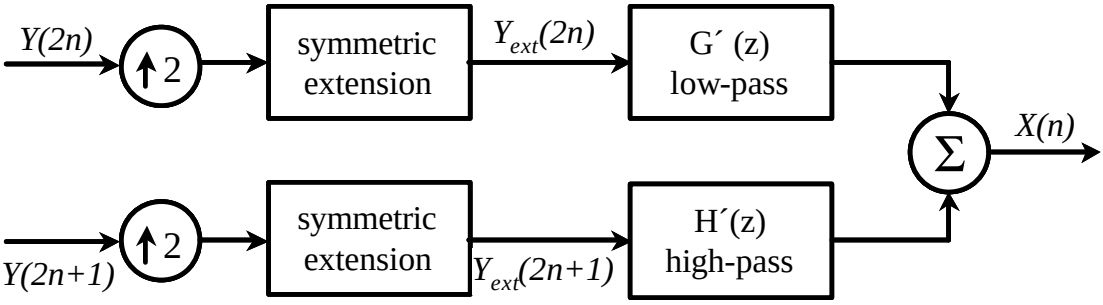

Figure 2.21. Wavelet synthesis (convolution implementation)

Figure 2.21 shows the convolution implementation of wavelet synthesis processing. The low-pass and high-pass subband samples, *Y(2n)* and *Y(2n+1)*, are interpolated by a factor of two (zeros are injected between samples, doubling the sequence length). The sequences are symmetrically extended using the same procedures as in the lifting case. The extended sequences are then filtered and summed together to form the reconstructed signal, X(n). The interpolation and summation steps are buried in the lifting implementation in the interleaving and alternating computation of even and odd samples. As in the lifting implementation, the convolution processing is non-expansive for both analysis and synthesis. The convolution two-dimensional wavelet transformation is implemented as separable one-dimensional transformations.

One can derive convolution equivalent forms of the 9-7I lifting wavelet filter. The analysis and synthesis convolution equivalent filter taps are given below in Table 2.1 through Table 2.4.

Table 2.1. 9-7I Low-pass analysis filter, *h(n)*. (9-tap filter)

Tap	Coefficient
-4	0.026 748 757 411
-3	-0.016 864 118 443
-2	-0.078 223 266 529
-1	0.266 864 118 443
0	0.602 949 018 236
1	0.266 864 118 443
2	-0.078 223 266 529
3	-0.016 864 118 443
4	0.026 748 757 411

Table 2.2. 9-7I High-pass analysis filter, *g(n)*. (7-tap filter)

Tap	Coefficient
-3	0.091 271 763 114
-2	-0.057 543 526 228
-1	-0.591 271 763 114
0	1.115 087 052 457
1	-0.591 271 763 114
2	-0.057 543 526 228
3	0.091 271 763 114

Table 2.3. 9-7I Low-pass synthesis filter, *g˜(n)*. (7-tap filter)

Tap	Coefficient
-3	-0.091 271 763 114
-2	-0.057 543 526 228
-1	0.591 271 763 114
0	1.115 087 052 457
1	0.591 271 763 114
2	-0.057 543 526 228
3	-0.091 271 763 114

Table 2.4. 9-7I High-pass synthesis filter, *h˜(n)*. (9-tap filter)

Tap	Coefficient
-4	0.026 748 757 411
-3	0.016 864 118 443
-2	-0.078 223 266 529
-1	-0.266 864 118 443
0	0.602 949 018 236
1	-0.266 864 118 443
2	-0.078 223 266 529
3	0.016 864 118 443
4	0.026 748 757 411

Those familiar with linear system theory and wavelet processing may notice that the high-pass analysis and low-pass synthesis filters are a little different than might be expected. The analysis filters have what is called a "(1,2)" normalization and the synthesis filters have a "(2,1)" normalization. What this means is that the analysis low-pass filter has a DC gain of 1, and the analysis high-pass filter has a Nyquist gain of 2. Mathematically we may express this as:

$$\sum_{n=-4}^{4} h(n) = 1 \qquad \text{low - pass analysis filter}$$

$$\sum_{n=-3}^{3} (-1)^{|n|} g(n) = 2 \qquad \text{high - pass analysis filter}$$

Equation 2.13

The interpolation process shown in Figure 2.21 that occurs during wavelet synthesis requires that the interpolated signals be scaled by a factor of two. This scaling of the signals may occur anywhere in the processing chain since we are dealing with linear systems. In typical wavelet processing systems the scaling is performed during the synthesis filtering stage by scaling the filter coefficients. This is not the case in JPEG 2000. The Nyquist gain on the analysis filter shown in Equation 2.13 represents the scaling factor of two needed during wavelet synthesis. Thus JPEG 2000 effectively scales the high-pass data, *Y(2n+1)*, by a factor of two.

Closer examination of the low-pass and high-pass synthesis filters reveals the following,

$$\sum_{n=-3}^{3} g'(n) = 2 \qquad \text{low - pass synthesis filter}$$

$$\sum_{n=-4}^{4} (-1)^{|n|} h'(n) = 1 \qquad \text{high - pass synthesis filter}$$

Equation 2.14

The low-pass synthesis filter has a gain of two that will be applied to *Y(2n)*, and the high-pass filter has a gain of one. Thus the scaling factor for the low-pass subband is applied during synthesis, and the scaling factor for the high-pass subband is applied to the data during analysis.

This somewhat odd behavior was adopted to make the 9-7I wavelet transformation and the reversible 5-3R wavelet transformation behave in a similar manner. The 5-3R wavelet transformation also has (1,2) normalization. In fact, it is possible to use the inverse 5-3R wavelet transformation to reconstruct a signal that was processed with the forward 9-7I wavelet transformation (and vice versa) and still achieve recognizable results. The normalization of the 9-7I and 5-3R wavelet filters does provide the real possibility of signal dynamic range expansion in the high-pass subbands (HL, LH, and HH orientation). To accommodate the signal expansion, JPEG 2000 employs what are known as *guard bits*.

2.3.3.1.3.2 Guard Bits

Even after accounting for the subband filter gains, it is necessary to consider the possibility of overflow in the integer representation of the wavelet coefficients, especially with multiple levels of transform. Guard bits are bits beyond the nominal dynamic range of the data that provide for overflow protection. Although the wavelet filters have been normalized at the DC and Nyquist frequencies, examination of their frequency domain responses reveals that they do not have flat responses in their pass-bands. For example, the low-pass 5-3R analysis filter has a significant side-lobe with a gain greater than one. Additional levels of wavelet transform will increase this gain. One guard bit is usually sufficient to provide for overflow protection with most natural images. Two guard bits are sufficient for virtually all natural and realistic synthetic images.

The following example illustrates a case where a 1D synthetic signal requires one guard bit for the 9-7I wavelet filters. The original signal is an 8-bit square wave signal with varying frequency. Figure 2.22 shows the signal after a DC level-shift of 128 has been applied.

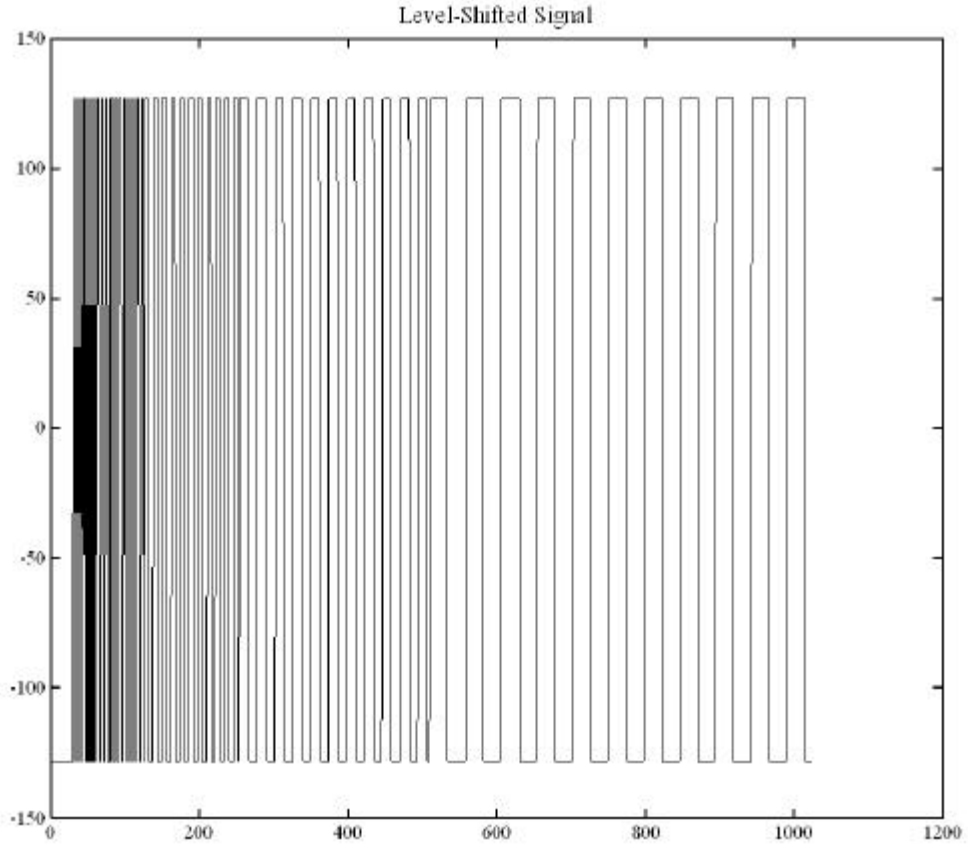

Figure 2.22. Guard bits example: level-shifted 8-bit square wave signal

Figure 2.23 shows the signal after three levels of wavelet decomposition with the 9-7I wavelet filter. The vertical dotted lines denote the subband boundaries. The horizontal dashed lines at –128/+127 denote the lower and upper bounds for an 8-bit signed integer. The horizontal dashed lines at –256/+255 denote the lower and upper bounds for a 9-bit signed integer. From the (1,2) normalization policy of the wavelet analysis filters, it was expected that 9 bits would be needed to represent the wavelet coefficients. One can plainly see, however, that some of the wavelet coefficients require a signed 10-bit representation because they exceed the 9-bit bounds. Thus, one guard bit, in addition to the expected 9 bits, is needed to fully represent the wavelet coefficient.

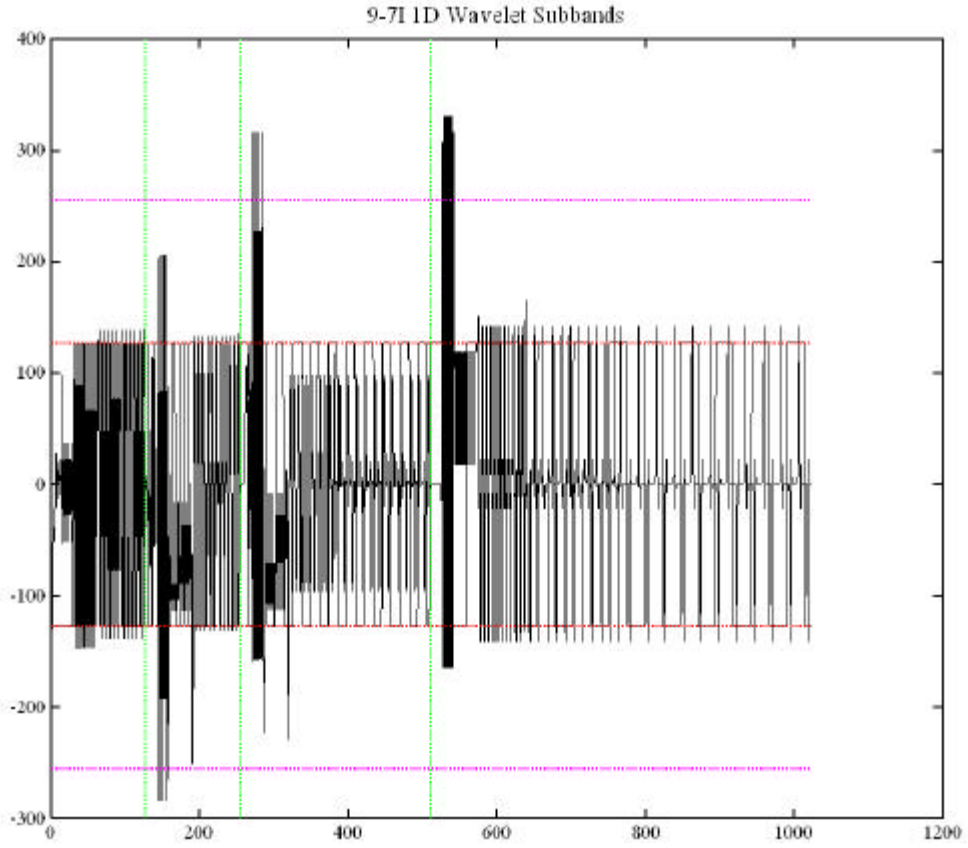

Figure 2.23. Guard bits example: three-level 9-7I decomposition of the square wave signal

One can easily image extending this example to two dimensions. In 2D, the wavelet processing is applied separately in the row and columnar directions. As the rows in a 2D signal are analyzed, we may require a guard bit to fully represent these intermediate values. After these values are analyzed in the columnar direction to complete the wavelet analysis, an additional guard bit may be required to fully represent the 2D wavelet coefficients.

Recommendation

For this system, the number of guards bits used during encoding and signaled in each applicable QCD/QCC marker segment shall be set to 2.

2.3.3.1.3.3 One-dimensional Signal Wavelet Transformation

In the recommended processing, the image data is broken into tiles that are 1,024 pixels x 1,024 pixels. It is possible that the image size is not an integer multiple of 1,024 pixels in size, and we may be forced to deal with small tiles on the right and bottom sides of the image (remember that the image and tile offsets are all set to zero). In fact, we may encounter a situation where the width or height (or possibly both) of a tile is less than 32 pixels. If this is the case, there will not be enough data in the tile to occupy every subband in a five level decomposition ($N_L = 5$). At some point in the subband decomposition structure, subbands will be created that are one-dimensional (with their width and/or height equal to one) or empty.

The forward and inverse wavelet transformations handle the case of transforming a one-dimensional signal (see Sections 2.3.3.1.1.5 and 2.3.3.1.2.5 of this document, Annex F.4.6 and Annex F.3.6 of ISO/IEC IS 15444-1). While there may not be any compression benefits in transforming small tiles, it is possible to do so. To change the wavelet decomposition used on a specific tile requires the use of a COD or COC marker segment in a first tile-part header. If the wavelet decomposition is changed on a tile, the number of packets present in the JPEG 2000 codestream for that tile will change. This will impact the layering and parsing envisioned for this system's processing. Instead, we shall perform the same number of wavelet decomposition levels on all tiles even if the signal becomes one-dimensional. If this does occur, the modified synthesis energy weights will be signaled through use of a tile-specific QCD marker segment.

Table 2.5. Subband sizes in 8 x 1,024 tile (five level decomposition)

Subband	Dimension	Parent Subband	Dimension
1HL	4 x 512	0LL	8 x 1,024
1LH	4 x 512		
1HH	4 x 512		
2HL	2 x 256	1LL	4 x 512
2LH	2 x 256		
2HH	2 x 256		
3HL	1 x 128	2LL	2 x 256
3LH	1 x 128		
3HH	1 x 128		
4HL	0 x 64	3LL	1 x 128
4LH	1 x 64		
4HH	0 x 64		
5HL	0 x 32	4LL	1 x 64
5LH	1 x 32		
5HH	0 x 32		
5LL	1 x 32		

To better understand what happens with small tiles, we consider the example shown in Table 2.5. In this example we are left with a tile of size 8 x 1,024 (8 pixels wide, 1,024 pixels high). Clearly there is not enough data in the column dimension to support five wavelet decomposition levels and have data appearing in every subband. Nevertheless, we can perform five levels of decomposition. At decomposition levels four and five we start generating empty subbands that contain no wavelet coefficients. Gray shading indicates the empty subbands.

Where the empty subbands appear is a function of the tile dimension and where the top-left corner lies, on an even or odd location on the reference grid. With the tiling used for this system's images, the top-left corner of all tiles will be located on an even location in the row and columnar directions. If the top-left corner were located just right on the reference grid in the row direction, the empty subbands could have been 4LH, 4LL, 5HL, 5LH, 5HH, and 5LL, with 4HL and 4HH containing all of the remaining subband data. The subband dimension equations in Annex B (Equation B.15) will indicate when a subband contains no data. The width, height, or both, given the tile coordinates within a subband, will become zero (i.e. $tbx_1 - tbx_0 = 0$ and/or $tby_1 - tby_0 = 0$).

When this happens, we still treat the subbands as if they exist, by coding empty packets for these subbands. By adopting this philosophy, every tile will have the same number of decomposition levels and the same number of packets per layer.

2.3.3.2 Quantization (Annex E)

Quantization is a many-to-one mapping that allows the arithmetic entropy coder to represent wavelet coefficients in a fewer number of bits. Quantization also introduces distortion into the reconstructed wavelet coefficients generated by the decoder because the process of quantization reduces wavelet coefficient precision. The forward and inverse quantization procedures for the 9-7I and 5-3R wavelet transformation coefficients are markedly different. The output of the 9-7I FDWT processing is a collection of wavelet subbands containing floating-point wavelet coefficients. Quantization of these coefficients is used to transform them into an integer representation with reduced entropy.

The output of the 5-3R FDWT is a collection of wavelet subbands containing integer wavelet coefficients. There is no explicit quantization of the 5-3R integer wavelet coefficients in JPEG 2000. For the 5-3R wavelet transformation, different bit rates or image qualities are achieved through the layering and rate control procedures (see Section 2.3.3.4). In a given layer in a codestream, the decoder receives some subset of the bit-planes forming each wavelet coefficient. If a decoder does not receive all of the layers in a numerically lossless encoded file, it cannot reconstruct the wavelet coefficients to full precision. The missing bit-planes are interpreted as zeros. This is an *implicit* form of quantization. One may equivalently view this behavior as quantization of the original wavelet coefficients by powers of two.

For NL encoded imagery, reception of all encoded layers by a decoder ensures numerically lossless decoding. Bit rates of NL encoded files will vary from file to file based on scene content. The use of layer truncation points would provide different quality levels subject to a rate constraint. It has been recommended that 19 such layer truncation points be used. Unless all 19 layers are received by a decoder, a decoded image will not be lossless. Due to the differences between the quantization procedures for the 9-7I and 5-3R wavelets, we shall describe them separately.

2.3.3.2.1 9-7I Wavelet

The quantizer used in ISO/IEC IS 15444-1 is a dead-zone scalar quantizer. Figure 2.22 illustrates the effects of quantization and reconstruction with a quantization step size, $\Delta_b = 0.5$, and reconstruction parameter, $r = 0.5$. The central bin of the quantizer is twice as wide as the other bins. This bin is called the "dead-zone". The parameter, r, allows a decoder to adjust where the reconstruction level is within the quantization bins. It is required that $0 \leq r < 1$. This is purely a decoder choice and it is not signaled in the ISO/IEC IS 15444-1 codestream. A value of $r = 0.5$ is commonly chosen, as it places the reconstruction levels of the quantization bins at their midpoints.

It is important to remember that the quantization does not produce an approximation of the original floating-point wavelet coefficient. Instead it maps the floating-point coefficients using a many to one mapping into a new integer form. This integer form is entropy coded and stored in the JPEG 2000 codestream. The decoder takes this integer representation and through dequantization generates an approximation to the original floating-point wavelet coefficient value. Figure 2.23 shows the input wavelet coefficients and the dequantized approximation that the decoder creates.

2.3.3.2.1.1 Forward Wavelet Coefficient Quantization Procedure (Annex E.2)

The forward quantization procedure (or simply "quantization") is described in Annex E.2. This is an informative section of the JPEG 2000 standard, but it is a normative part of this document. Each floating-point 9-7I wavelet transformation coefficient, $a_b(u,v)$, in a given subband b, is quantized or mapped to an integer value, $q_b(u,v)$, according to Equation 2.15. We will still refer to this value as a quantized wavelet coefficient since there exists a one to one mapping from this integer representation to the dequantized wavelet coefficient generated by the decoder. It is the dequantized wavelet coefficients generated in the decoder that are quantized approximations to the original wavelet coefficients.

$$q_b(u,v) = \text{sign}\left(a_b(u,v)\right) \cdot \left\lfloor \frac{|a_b(u,v)|}{\Delta_b} \right\rfloor$$

Equation 2.15

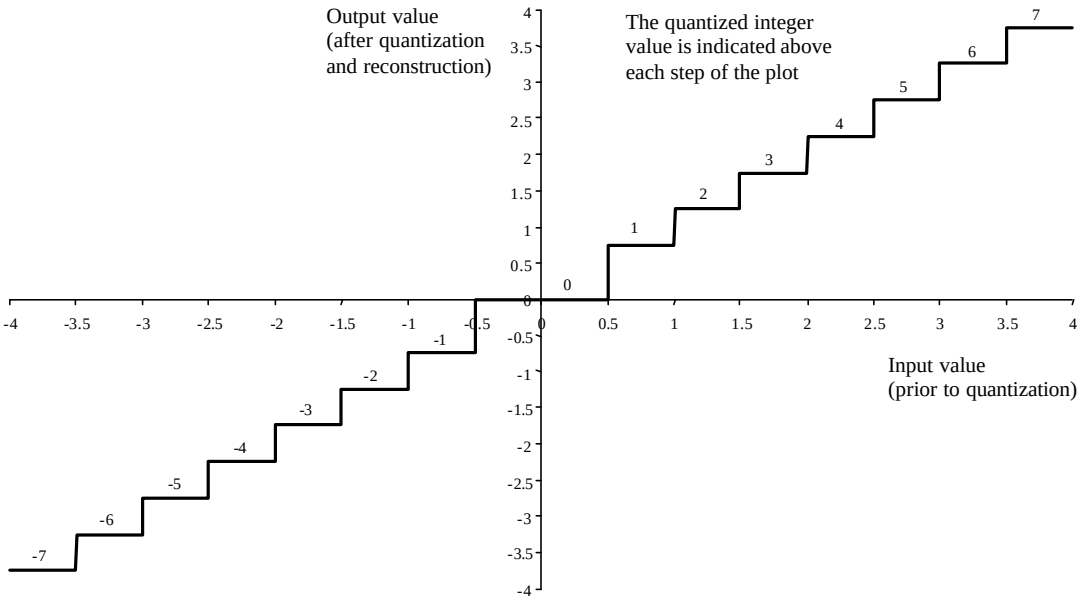

Figure 2.24. Quantization example (Δ_b = 0.5, *r* = 0.5)

Equation 2.16 preserves six additional fractional bits in the integer representation of $q_b(u,v)$ that are not transmitted by the encoder in the compressed codestream. These bits are used by the layering formation procedure (see Section 2.3.3.4) in the computation of rate-distortion values. The relationship between $\tilde{q}_b(u,v)$ and $q_b(u,v)$ is simply a right shift of six bit-planes. It is therefore straightforward in an implementation to use $\tilde{q}_b(u,v)$ and transmit $q_b(u,v)$ by simply not encoding the six least significant bit-planes. Equation 2.15 gives the quantization integers that are stored in the compressed codestream.

$$\tilde{q}_b(u,v) = \text{sign}\left(a_b(u,v)\right) \cdot \left\lfloor 2^6 \cdot \frac{|a_b(u,v)|}{\Delta_b} \right\rfloor$$

Equation 2.16

2.3.3.2.1.2 Inverse Wavelet Coefficient Quantization Procedure (Annex E.1)

The inverse wavelet coefficient quantization procedure is normatively described in Annex E.1 of ISO/IEC IS 15444-1. For a *decoder*, the procedures described in Annex E.1 are normative. However, for this system's compressor, only the previously described forward wavelet coefficient quantization procedure is normative. Section 2.3.3.2.1.2 is included here strictly as an informative section.

For each wavelet transform coefficient, $a_b(u,v)$, the decoder constructs the integer representation, $\overline{q}_b(u,v)$, of this coefficient at the current number of decoded bit-planes using Equation 2.17 (Equation E.1). The JPEG 2000 encoder encodes the integer representation of the wavelet coefficients, $q_b(u,v)$, in an embedded manner (see Section 2.3.3.3). This means that the bits associated with the integers, $q_b(u,v)$, are distributed throughout the codestream. At any particular time in the decoding of the codestream the decoder may be forced to stop due to truncation of the compressed codestream; it may elect to stop, having met some quality or rate constraint enforced on the reconstructed image, or the encoder may have elected to not place all of the bit-planes associated with a given quantized wavelet coefficient in the compressed codestream. In any case, the decoder may not receive all of the bit-planes making up the quantized integer representation of a particular wavelet coefficient.

$$\overline{q}_b(u,v) = (1 - 2s_b(u,v)) \cdot \left(\sum_{i=1}^{N_b(u,v)} MSB_i(b,u,v) \cdot 2^{M_b - i} \right)$$

Equation 2.17

Equation 2.17 accounts for this behavior by assembling as much of the current quantized wavelet coefficient as has currently been received by the decoder. The output of Equation 2.17, $\overline{q}_b(u,v)$, is therefore an approximation to the complete integer representation of the quantized wavelet coefficient, $q_b(u,v)$. The parameter $s_b(u,v)$, represents the sign bit associated with the current integer representation at location (u,v) in subband, b. The manner in which sign bits are encoded in the codestream ("just in time coding") are explained in Annex D of ISO/IEC IS 15444-1 and Section 2.3.3.3.1.1 of this document. The sign bit is 0 for positive and zero coefficients and 1 for negative coefficients.

The parameter, $N_b(u,v)$, in Equation 2.17 represents the number of decoded magnitude bits for the quantized wavelet coefficient at location (u,v) in subband b. This parameter keeps track of how many bit-planes have been decoded for the quantized wavelet coefficient under consideration. Included in $N_b(u,v)$ is the number of zero most significant bit-planes signaled in the packet headers for the code-block associated with the quantized wavelet coefficient (see Annex B.10.5 of ISO/IEC IS 15444-1). The parameter, $MSB_i(b,u,v)$, contains the bit in bit-plane i, of subband b, of the wavelet coefficient at location (u,v). Finally the parameter, M_b, represents the number of bit-planes that are being used to represent quantized wavelet coefficients for the current tile-component. This parameter is determined from the applicable QCD/QCC marker segment for the current tile-component. It is computed from Equation 2.18 (Equation E.2).

$$M_b = G + e_b - 1$$

Equation 2.18

The parameter, G, in Equation 2.18 is the number of guard bits, signaled in the applicable QCD/QCC marker segment (also see Annex A.6.4 and Annex A.6.5 of ISO/IEC IS 15444-1). For the recommended implementation of JPEG 2000, $G = 2$ (see Section 2.3.3.1.3.2). The other parameter, e_b, is related to the base 2 exponent of the magnitude of the quantization step size, Δ_b. Thus, M_b, is the number of bit-planes in which we are representing the integer representation of the quantized wavelet coefficients (note the "-1" accounts for the sign bit). Re-examining Equation 2.18, we see that $\overline{q}_b(u,v)$ contains the sign and $N_b(u,v)$ most significant bits of $q_b(u,v)$. It is simply the portion of the integer representation, $q_b(u,v)$, that has been received so far at the decoder. If only the top $N_b(u,v)$ bit-planes of $q_b(u,v)$ are decoded, this is equivalent to using a scalar quantizer with step size $2^{M_b - N_b(u,v)} \cdot \Delta_b$. If all bit-planes in the quantized wavelet coefficient are used (i.e. $N_b(u,v) = M_b$) then we have, $\overline{q}_b(u,v) = q_b(u,v)$.

The quantization step size, Δ_b, for the current wavelet subband, b, is determined by the QCD/QCC that applies to the current tile-component (see Annex A.6.4 and A.6.5). The quantization step sizes are expressed as exponent, mantissa pairs within a 16-bit integer value consisting of a 5-bit exponent, e_b, and 11-bit mantissa, m_b. The (e_b, m_b) pairs are ordered in the QCD/QCC marker segment in the same order as the wavelet subbands in Annex F.3.1.

Equation 2.19 (Equation E.3) shows how the subband quantization step size, Δ_b, is recovered from the (exponent, mantissa) format (e_b, m_b). The parameter, R_b, represents the number of bit-planes used to represent the original wavelet coefficients. Equation 2.20 (Equation E.4) gives the expression for R_b. Equation 2.19 shows that the quantization parameters, (e_b, m_b), signaled in the codestream are *relative* quantization step size parameters. They specify a relative quantization step size, Δ_{rel_b}, where $\Delta_{rel_b} \in (0,1]$. In other words, the quantization step size is signaled relative to the wavelet coefficient bit depth. For example, suppose $R_b = 8$, that is the wavelet coefficients in the current subband are eight bits in magnitude. If we set (e_b, m_b) = (0,0), then $\Delta_b = 256$ and we quantize all of the wavelet coefficients to zero, since no wavelet coefficient can have a magnitude greater than 255. The output of Equation 2.19, Δ_b, is an *absolute* quantization step size.

$$\Delta_b = 2^{R_b - e_b}\left(1 + \frac{m_b}{2^{11}}\right) = 2^{R_b} \cdot \Delta_{rel_b}$$

Equation 2.19

$$R_b = R_I + \log_2(gain_b)$$

Equation 2.20

In Equation 2.20, R_I represents the bit depth associated with the current component, which is extracted from the SIZ marker segment (see Annex A.5.1). (Note that R_I is the bit depth, which is one greater than the value coded in the Ssiz field.) The parameter $gain_b$ represents the subband gain associated with subbands with the same orientation as that of subband b. In Sections 2.3.3.1.3.1 and 2.3.3.1.3.2, we looked at the normalization of the 9-7I analysis wavelet filters and saw that the LH and HL subband orientations had a gain of 2 and the HH subband orientation had a gain of 4. The LL subband orientation was found to have a gain of 1. The parameter $gain_b$ is precisely this gain (see Table E-1 in ISO/IEC IS 15444-1). Thus Equation 2.20 accounts for the signal expansion of the forward wavelet transformation. It is not necessarily true that the original wavelet coefficients and their quantized integer representation possess the same bit depth. Therefore it is necessary to separately maintain the two bit depths represented by M_b and R_b.

For irreversible wavelet transformations, two types of signaling may be used for the quantization step sizes: derived and expounded. If derived quantization signaling is used, a single (e_b, m_b) pair is signaled for the N_LLL subband and all other quantization step sizes are derived from this value by a power of 2 scaling described by Equation E.5. Expounded quantization step size signaling stores an (e_b, m_b) pair for each subband in the wavelet decomposition.

Once the approximation to the integer representation of the quantized wavelet coefficient, $\overline{q}_b(u,v)$, has been determined, the quantized reconstructed wavelet coefficient may be computed. Equation 2.21 (Equation E.6) gives the proper relationship. The reconstruction parameter, r, was mentioned in Section 2.3.3.2.1. It allows a decoder the flexibility of moving the reconstruction level within the quantization bins. The JPEG 2000 standard does not specify a value of r, but the recommended implementation is that the value of r shall be 0.5. Under some circumstances alternative values of r may produce decoded imagery that are more visually pleasing or have a lower mean squared error relative to the original images. The output of Equation 2.21 is the reconstructed quantized wavelet coefficients. Note that if no magnitude bits are available for the wavelet coefficient under consideration ($N_b(u,v) = 0$), the reconstructed quantized wavelet coefficient value shall be zero.

$$Rq_b(u,v) = \begin{cases} \left(\overline{q}_b(u,v) + r2^{M_b - N_b(u,v)}\right)\cdot \Delta_b & \text{for } \overline{q}_b(u,v) > 0 \\ \left(\overline{q}_b(u,v) - r2^{M_b - N_b(u,v)}\right)\cdot \Delta_b & \text{for } \overline{q}_b(u,v) < 0 \\ 0 & \text{for } \overline{q}_b(u,v) = 0 \end{cases}$$

Equation 2.21

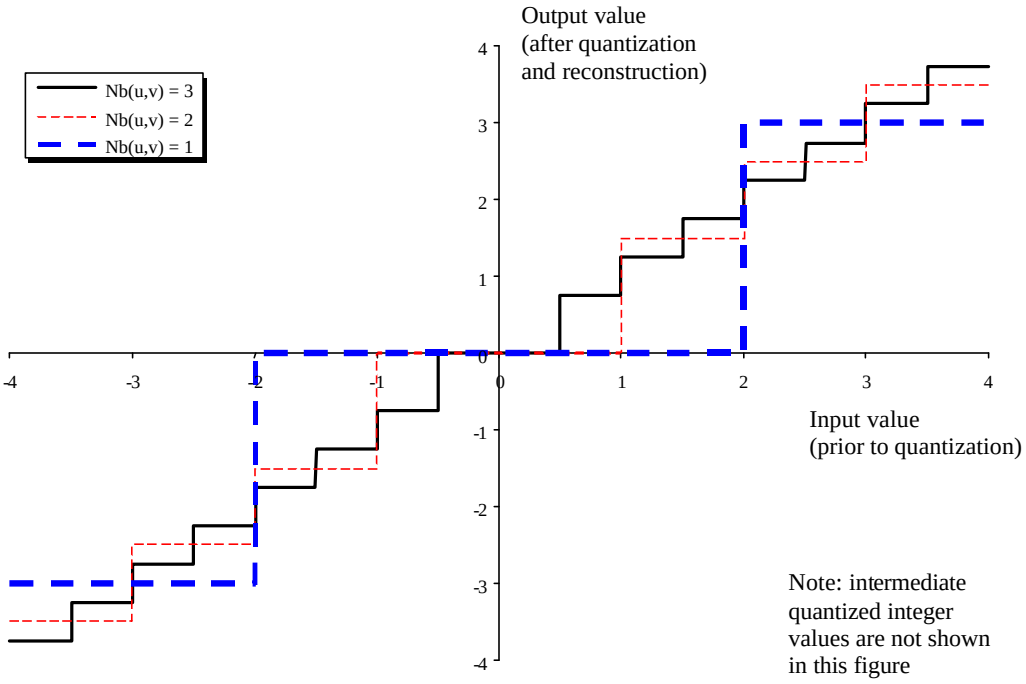

Figure 2.25. Quantization example varying the number of decoded bit-planes (D_b = 0.5, r = 0.5, M_b = 3)

Figure 2.25 shows the effects of varying the number of decoded bit-planes in the integer representation of the quantized wavelet coefficients. The solid line in Figure 2.25 represents the case where we have received the sign and all three magnitude bit-planes ($N_b(u,v)$ = 3), and is the same as that in Figure 2.22. For this case, the quantized integer representations range from –7 to 7 and the reconstructed coefficients range from –3.75 to 3.75. Keep in mind that the input value in Figure 2.25 is the unquantized wavelet coefficient, not the quantized integer representation. The thin dashed line shows what the quantization looks like if we have only received the top two magnitude bit-planes ($N_b(u,v)$ = 2). Since we do not have the least significant bit, we can no longer distinguish between integers –7 and –6, or 3 and 2. The result is that the odd integers map to even integers and the only integers allowed are (-6, -4, -2, 0, 2, 4, 6). For this case the output ranges –3.5 to 3.5, but there are fewer intermediate values, roughly half as many. The thick dashed line shows the effects of receiving only the most significant magnitude bit ($N_b(u,v)$ = 1). The only integers we can represent are (-4, 0, 4) and the only output values are (-3, 0, 3). *Note that receiving fewer bit planes is equivalent to originally quantizing with a larger step size*.

2.3.3.2.1.3 Base Step Size Quantization

Base step size quantization is a way to specify a set of subband quantization step sizes using a single step size, Δ_{base}, that is then scaled and adapted for each subband to yield the actual Δ_b step size values used for each subband, b. It is similar in operation to the derived quantization described in the JPEG 2000 standard, but the relationship between the subband step sizes is not based on power of two relationships between decomposition levels. Instead, the quantization step size is adjusted based on the L2 norm of the synthesis filter chain that leads from a particular subband, b, back to the resolution level of the original image. Expounded quantization step size signaling is used to indicate the step sizes for each subband.

This form of quantization strives to make the error contributions from each wavelet subband the same. The normalization procedures for the 9-7I wavelet filters, along with the fact that the filters are only near-orthonormal, causes the same numerical error in each subband in the wavelet decomposition to contribute slightly different mean

squared error in the reconstructed image. To normalize the contributions between the wavelet subbands, one must determine how errors in the subband wavelet coefficients for each subband are amplified in the final reconstructed image. If we were to populate a particular wavelet subband with unit-variance white noise, leaving all other subbands empty (i.e. full of zero coefficients), and synthesize the resulting decomposition; we would find amplification of the variance of the reconstructed noise from that subband. Linear system theory tells us that this amplification factor is given by the square of the L2 norm of the impulse response that links a particular wavelet subband with the final reconstructed image. The square root of this amplification factor is called an *energy weight*.

To determine what this impulse response is, it is best to think of the wavelet synthesis in terms of the equivalent convolution processing described in Section 2.3.3.1.3.1. Consider a simple one-dimensional one-level wavelet synthesis from a low-pass subband as shown in Figure 2.26:

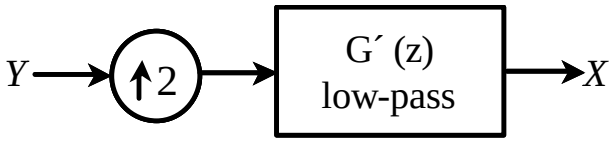

Figure 2.26. One-dimensional, one-level, low-pass synthesis

Interpolation zeros are first inserted into the input signal, Y, which is then convolved with the impulse response, g', of the low-pass synthesis filter G'. The input to the system, Y, represents a 1L subband orientation. If we fill this subband with white noise of variance, s_Y^2, we may determine that the variance of the output signal, s_X^2, is related to s_Y^2 by the following expression:

$$s_X^2 = \sum_{i=-m}^{m} g(i)^2 \cdot s_Y^2$$
$$= EW^2 \cdot s_Y^2$$

Equation 2.22

where the filter impulse response is defined over the interval [-*m*,*m*] and EW is the energy weight. If more than one level of wavelet synthesis is required to restore a subband back to the original image resolution, the determination of the energy weight is somewhat more complex.

Figure 2.27 shows an example where a subband must pass through two levels of wavelet synthesis processing, a high-pass filtering operation followed by a low-pass filtering operation. Again we consider a one-dimensional wavelet decomposition for simplicity. In this example, the input signal, Y, corresponds to a 2H subband orientation. We may equate this set of high-pass filtering, intermediate upsampling, and low-pass filtering operations with a system like that shown in Figure 2.26. This is illustrated in the bottom half of Figure 2.27. The aggregate convolution filter, F(z), is created by upsampling the high-pass synthesis filter, H'(z), and convolving it with the low-pass synthesis filter, G'(z). Once the aggregate filter, F(z), has been determined, the two-level synthesis problem has been reduced to the one-level synthesis problem and Equation 2.22 may be applied with f(i) replacing g(i) in the equation and modifying the summation limits appropriately.

Extending this process to the two-dimensional case is straightforward. We may treat the row and column directions separately since separable 1-D filtering is used to implement the 2-D wavelet transform. Consider the three level wavelet decomposition in Figure 2.28. The processing steps needed to reconstruct the subband 3HL back to the full image resolution in the row direction are: upsample and high-pass filter, upsample and low-pass filter, and upsample and low-pass filter. In the column direction there are three upsample and low-pass filter operations.

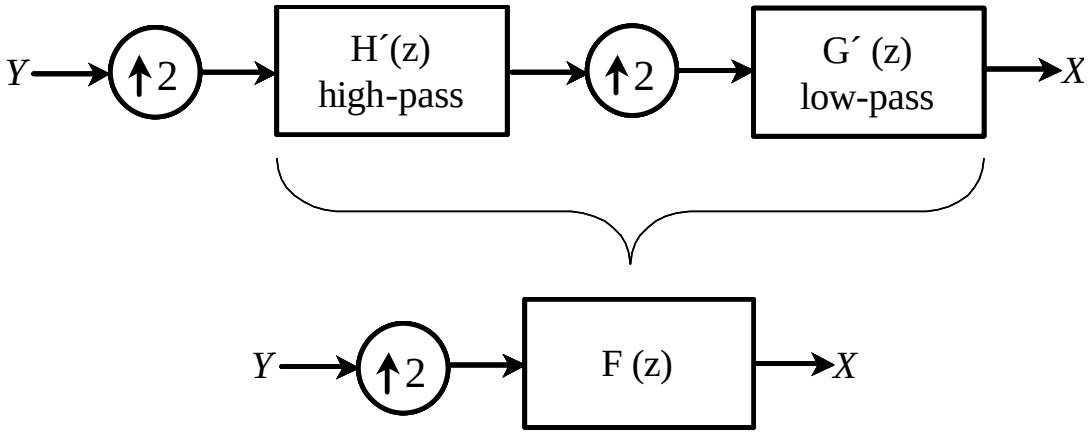

Figure 2.27. One-dimensional, two level synthesis.

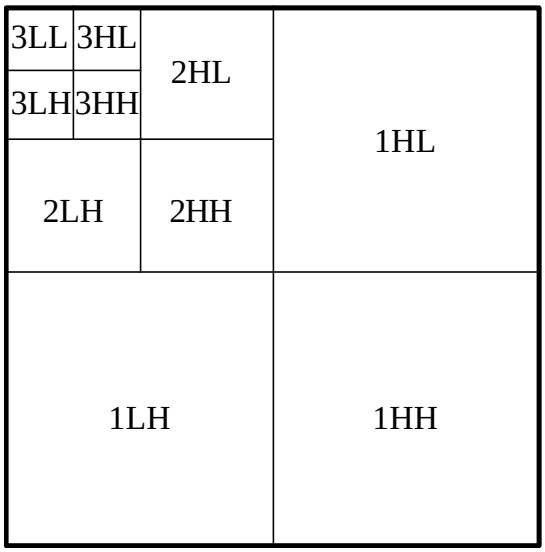

Figure 2.28. Three level wavelet decomposition

Table 2.6 lists the row and column operations for each subband in the three level decomposition shown in Figure 2.28. In this table an "L" indicates a low-pass synthesis filtering operation and an "H" indicates a high-pass synthesis filtering operation. In between each filtering operation there is an upsampling stage. To determine the filter associated with the impulse response between each wavelet subband and the full image resolution, we may take the convolution form of the wavelet filters and then upsample and convolve them against each other in the orders given by Table 2.6. Thus for the row direction in subband 3HL (HLL), we would take the high-pass synthesis convolution filter, upsample it by a factor of two, convolve it with the low-pass synthesis filter, upsample the result by a factor of two, and convolve this with the low-pass filter. Similarly, in the column direction for subband 3HL (LLL), we would perform the same operations with the low-pass filter at each stage. Note that the number of upsample stages that the filters see is one less than the number of upsample stages that the wavelet subband coefficients see.

Table 2.6. Wavelet synthesis row and column operations

Subband	Row Operations	Column Operations
3LL	LLL	LLL
3HL	HLL	LLL
3LH	LLL	HLL
3HH	HLL	HLL
2HL	HL	LL
2LH	LL	HL
2HH	HL	HL
1HL	H	L
1LH	L	H
1HH	H	H

For the row and column directions we may determine the aggregate convolution filter that forms the impulse response from the subband back to the original image resolution. The L2 norm of each filter is given by the square root of the sum of the squares of the filter coefficients. Let CF_{L2b} represent the L2 norm of the aggregate convolution filter in the column direction and RF_{L2b} represent the L2 norm of the aggregate convolution filter in the row direction. See Tables Table 2.3 and Table 2.4 in Section 2.3.3.1.3.1 for the convolution synthesis filter coefficients for the 9-7I wavelet, but note that the high-pass coefficients from Table 2.4 must be multiplied by a factor of 2. This is necessary because the normalization of the high-pass synthesis filter is equal to one. When computing the L2 norm of the aggregate convolution filters the factor of 2 needed for synthesis must be included. This factor is already present in the low-pass filter (the synthesis filters have (2,1) normalization) but is not present in the high-pass filter (remember that the factor of 2 was multiplied into the wavelet coefficients).

The normalization of the 9-7I wavelet filters affects how we compute the L2 norms. If we place the factors of 2 in the low-pass and high-pass synthesis filters when computing the L2 norms as described above, we are computing the norms under (1,1) analysis normalization (note the synthesis normalization is (2,2) in this case). Under (1,1) normalization, the HL and LH subband orientations have not been pre-multiplied by 2 and the HH subband orientation has not been pre-multiplied by 4 as we found in the case (1,2) analysis normalization. If the factors of 2 (and 4) are pre-multiplied into the wavelet coefficients, then these factors should not be included in the L2 norm calculation. We must therefore account for this difference in the (1,1) L2 norm versus the (1,2) L2 norm. A correction factor, $gain_b$, is included in Equation 2.23 for this purpose. The $gain_b$ factor is simply the subband gains from Table E-1 in ISO/IEC IS 15444-1, the same factor shown in Equation 2.20. The factors CF_{L2b} and RF_{L2b} are also known as *energy weights*, since they describe how the error variance within a subband is amplified during wavelet synthesis filtering.

$$\overline{\Delta}_b = \frac{\Delta_{base} \cdot gain_b}{(CF_{L2b})(RF_{L2b})}$$

Equation 2.23

The base quantization step size, Δ_{base}, is adapted for each subband using Equation 2.23. The true absolute subband quantization step sizes, $\overline{\Delta}_b$, are not exactly the same as what the decoder will determine from the codestream using Equation 2.19 and the (e_b, m_b) pairs from the appropriate QCD/QCC marker segment. The (e_b, m_b) representation of Δ_{rel_b}, is a quantized version of the true relative quantization step sizes, $\overline{\Delta}_{rel_b}$, where $\overline{\Delta}_b = 2^{R_b} \cdot \overline{\Delta}_{rel_b}$. Given a true absolute quantization step size, $\overline{\Delta}_b$, we may compute the (e_b, m_b) approximation, using Equation 2.24. It is

important to remember when deriving quantization step sizes to ensure that the encoder is using the same step sizes as the decoder.

$$e_b = \left\lceil R_b - \log_2\left(\overline{\Delta}_b\right) \right\rceil$$
$$m_b = \left\lfloor 2^{11}\left(\frac{\overline{\Delta}_b}{2^{R_b - e_b}} - 1 \right) + 0.5 \right\rfloor$$

Equation 2.24

Therefore on the encoder side, when deriving quantization step sizes, be sure to convert them into (e_b, m_b) form and back again to ensure that the encoder uses the same numeric representation of the step sizes as the decoder does.

2.3.3.2.1.3.1 Energy Weight Examples (Full Tiles)

In this section we provide some examples of energy weight computation along with figures to illustrate the processes involved. We will assume that the tiles being wavelet transformed have sufficient width and height to support the desired number of wavelet decomposition levels without them becoming one-dimensional in either the row or column dimension (see section 2.3.3.1.3.3).

Figure 2.29 is a plot of the analysis and synthesis 9-7I wavelet convolution kernels with (1,2) analysis normalization. In both the top and bottom panes of the figure, the solid line corresponds to the low-pass filter and the dashed line to the high-pass filter. These filters are the convolution equivalent of the 9-7I lifting wavelets. When determining energy weights, it is easiest to use the convolution representation of the wavelet processing. In the following example, we compute the energy weights for the subband decomposition shown in Figure 2.28. Looking at Table 2.6, we see that we must determine the "aggregate convolution" filters (i.e. the "F(z)" equivalent in Figure 2.27) for the LLL, HLL, LL, and HL synthesis processing. Remember that the aggregate convolution filters for one level of wavelet synthesis are simply the wavelet filters themselves.

Figure 2.28 shows the synthesis LL aggregate convolution filter in the bottom pane. In the top pane of the figure, the low-pass synthesis filter is shown with interpolation zeros inserted. The bottom pane shows the resulting filter generated by convolving the filter in the top pane of the figure with the low-pass synthesis filter shown in Figure 2.29. Thus the bottom half of Figure 2.30 is the result of taking the low-pass synthesis filter, interpolating it, and convolving that result with the low-pass synthesis filter. Figure 2.31 shows the synthesis HL aggregate convolution filter in its bottom pane. These two filters are all we need to determine the energy weights for the 2LL, 2HL, 2LH, and 2HH subband orientations. In particular, if we let:

$$EW_{LL} = \sqrt{\sum_i LL(i)^2}$$
$$EW_{HL} = \sqrt{\sum_i HL(i)^2}$$

then we have:

$$\begin{aligned}
\left(CF_{L2(2LL)}\right)\left(RF_{L2(2LL)}\right) &= EW_{LL} \cdot EW_{LL} \\
\left(CF_{L2(2HL)}\right)\left(RF_{L2(2HL)}\right) &= EW_{HL} \cdot EW_{LL} \\
\left(CF_{L2(2LH)}\right)\left(RF_{L2(2LH)}\right) &= EW_{LL} \cdot EW_{HL} \\
\left(CF_{L2(2HH)}\right)\left(RF_{L2(2HH)}\right) &= EW_{HL} \cdot EW_{HL}
\end{aligned}.$$

These are the four energy weights for the 2XX subbands.

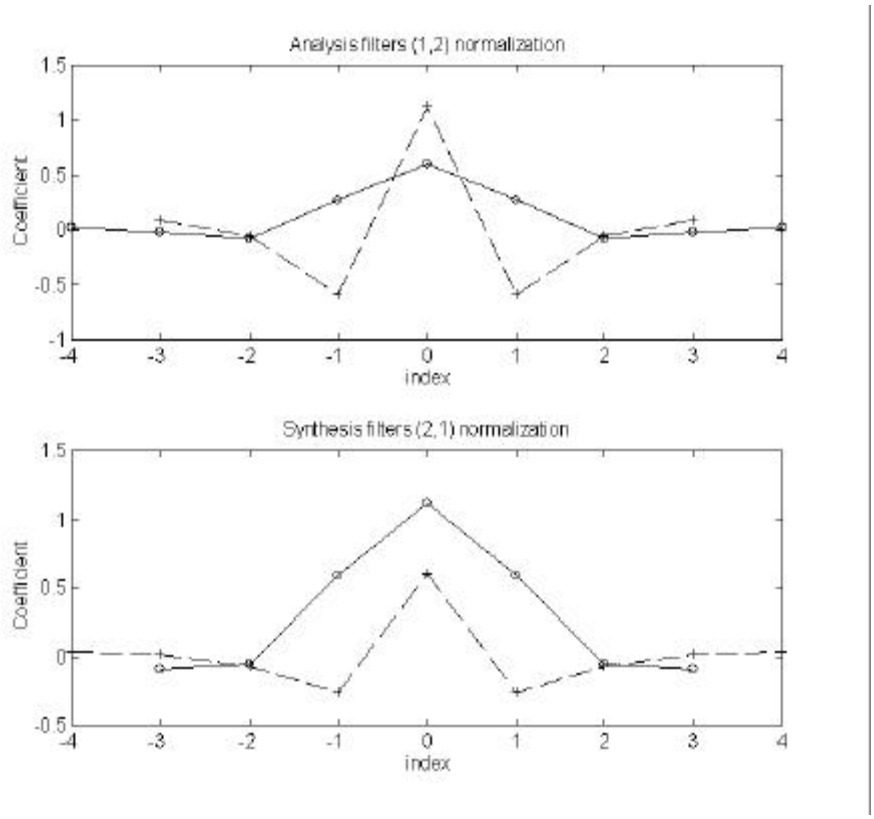

Figure 2.29. 9-7I Convolution filter kernels

In our three level decomposition example, there is no 2LL subband; it is split into 3LL, 3HL, 3LH, and 3HH subbands. Therefore we have another set of aggregate convolution filters to find, LLL and HLL. These filters are determined by taking the LL and HL aggregate filters, inserting interpolation zeros and convolving the result with the low-pass synthesis filter. This is shown in Figure 2.32 and Figure 2.33. As in the 2XX subband case, if we let:

$$EW_{LLL} = \sqrt{\sum_i LLL(i)^2}$$
$$EW_{HLL} = \sqrt{\sum_i HLL(i)^2}$$

then we have

$$\begin{aligned}
\left(CF_{L2(3LL)}\right)\left(RF_{L2(3LL)}\right) &= EW_{LLL} \cdot EW_{LLL} \\
\left(CF_{L2(3HL)}\right)\left(RF_{L2(3HL)}\right) &= EW_{HLL} \cdot EW_{LLL} \\
\left(CF_{L2(3LH)}\right)\left(RF_{L2(3LH)}\right) &= EW_{LLL} \cdot EW_{HLL} \\
\left(CF_{L2(3HH)}\right)\left(RF_{L2(3HH)}\right) &= EW_{HLL} \cdot EW_{HLL}
\end{aligned}.$$

These are the 3XX subband energy weights.

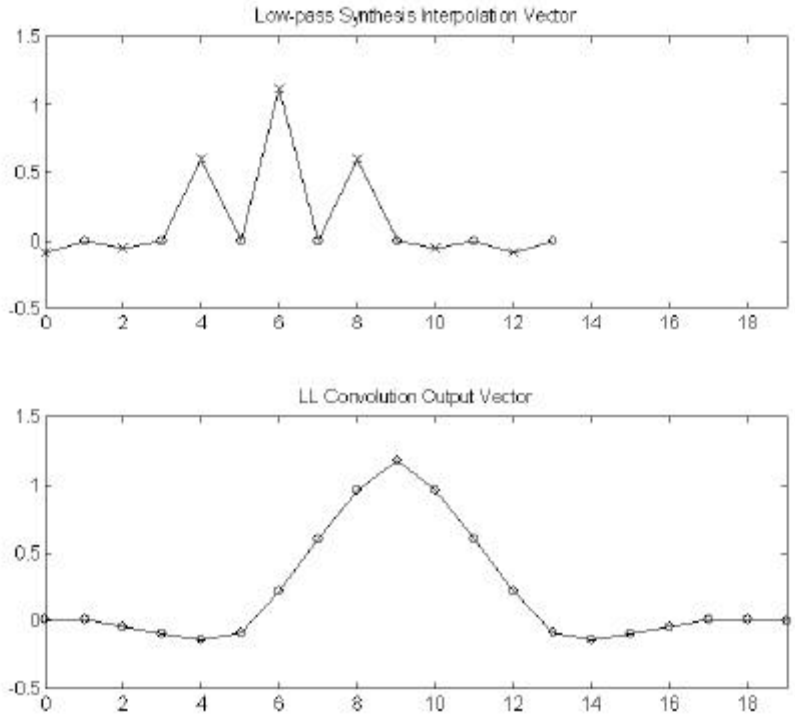

Figure 2.30. 9-7I LL synthesis aggregate convolution filter

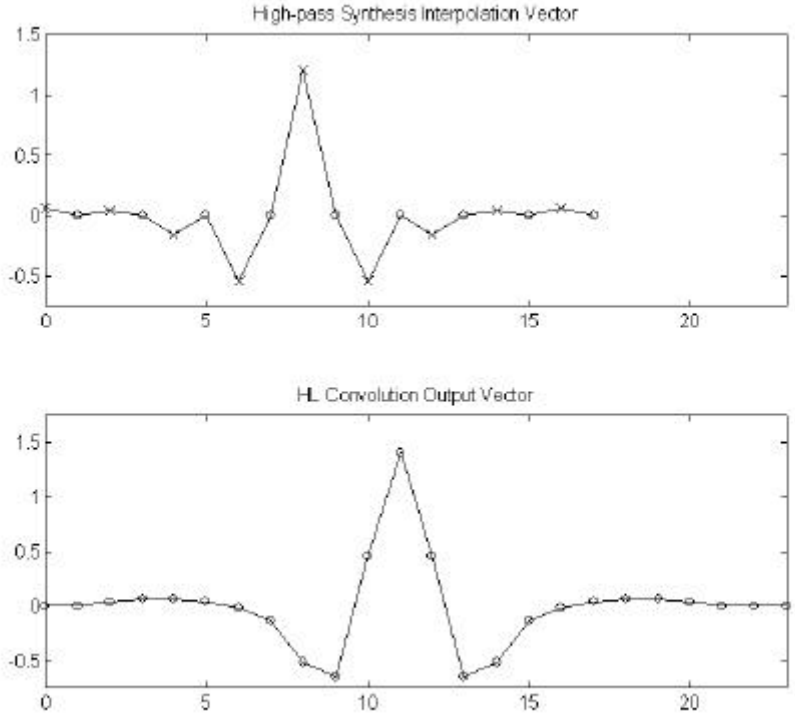

Figure 2.31. 9-7I HL synthesis aggregate convolution filter

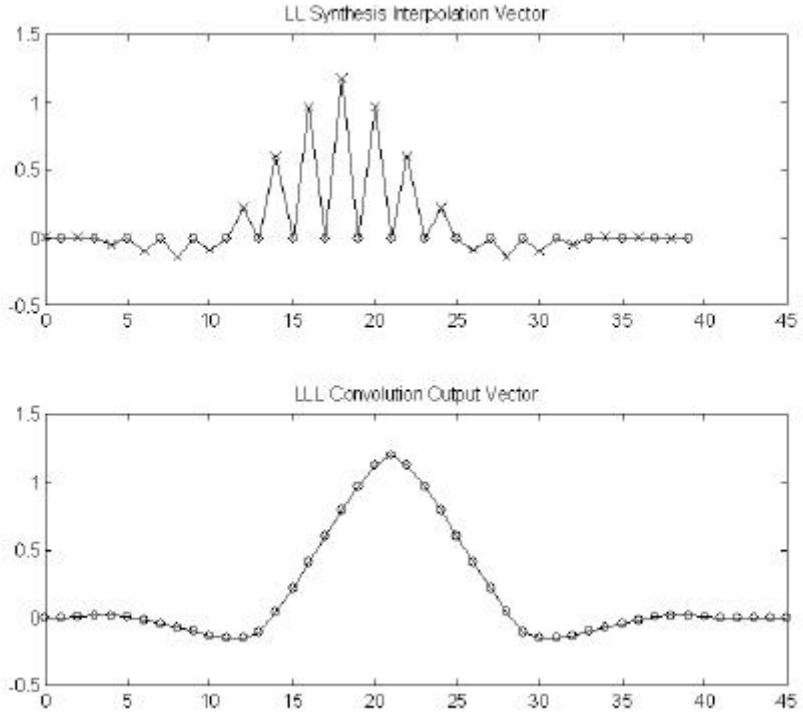

Figure 2.32. 9-7I LLL synthesis aggregate convolution filter

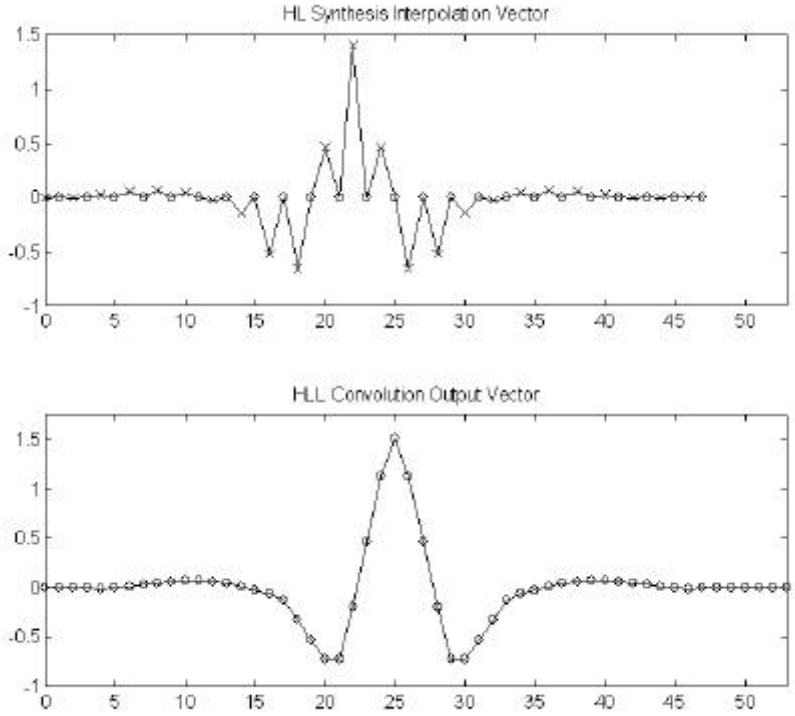

Figure 2.33. 9-7I HLL synthesis aggregate convolution filter

Table 2.7. Energy weight calculations for five level 9-7I wavelet decomposition

Decomposition Level	Subband Orientation	Energy Weight	Aggregate Filter L2 Norm	
1	LL	1.965907	L	1.402108
	HL	2.022573	H	1.442523
	LH	2.022573		
	HH	2.080872		
2	LL	4.122410	LL	2.030372
	HL	3.993625	HL	1.966943
	LH	3.993625		
	HH	3.868863		
3	LL	8.416744	LLL	2.901163
	HL	8.366735	HLL	2.883925
	LH	8.366735		
	HH	8.317022		
4	LL	16.935572	LLLL	4.115285
	HL	17.068231	HLLL	4.147521
	LH	17.068231		
	HH	17.201929		
5	LL	33.924927	LLLLL	5.824511
	HL	34.333452	HLLLL	5.894650
	LH	34.333452		
	HH	34.746896		

Table 2.7 gives the computed energy weights for a five level wavelet decomposition using the 9-7I wavelet. The shaded cells in the table indicate the set of energy weights that are needed. The LL subband weights for decomposition levels 1 – 4 are not needed because those subbands were further split during generation of the wavelet decomposition. They are included here for completeness and they may be useful if the number of wavelet decomposition levels is changed. The right column in this table gives the L2 norms of the two aggregate convolution filters that each level of wavelet synthesis creates. The products of these L2 norms form the subband energy weights as discussed above.

2.3.3.2.1.3.2 Energy Weight Examples (Small Tiles)

Computation of the energy weights for small tiles is subtly different than the computation of energy weights in the full tile case. As we saw in section 2.3.3.1.3.3 and Table 2.5 (also see sections 2.3.3.1.1.5 and 2.3.3.1.2.5), when a tile has a small enough row and/or column dimension, there may be an insufficient number of data samples to support the full number of wavelet decomposition levels. In this case we will generate subbands that are either

empty or one-dimensional (i.e. the width and/or height of the subband is one). Further wavelet processing is not performed in row and/or column direction of a wavelet subband if the width and/or height of that subband are equal to one. Instead, a simple scaling of the coefficients is performed. This affects the energy weight computations.

To better understand this, let us consider a 4 x 16 tile on which we wish to perform five levels of wavelet decomposition. Table 2.8 shows the subband sizes resulting from this decomposition, assuming that the top-left corner of the tile lies on an even row and even column location of the reference grid, where both the row and column numbers are integer multiples of $2^{number\ of\ decompositions}$ (this will be the case for this system's imagery). For those subbands that are empty (i.e. the row and/or column dimension is zero), the energy weight is not needed and it is set to equal to zero. This is merely good programming practice since the quantization step size parameters (e_b, m_b), are always set to (0,0) for empty subbands (see below).

Table 2.8. Subband sizes in 4 x 16 tile (five level decomposition)

Subband	Dimension	Parent Subband	Dimension
1HL	2 x 8	0LL	4 x 16
1LH	2 x 8		
1HH	2 x 8		
2HL	1 x 4	1LL	2 x 8
2LH	1 x 4		
2HH	1 x 4		
3HL	0 x 2	2LL	1 x 4
3LH	1 x 2		
3HH	0 x 2		
4HL	0 x 1	3LL	1 x 2
4LH	1 x 1		
4HH	0 x 1		
5HL	0 x 0	4LL	1 x 1
5LH	0 x 0		
5HH	0 x 0		
5LL	1 x 1		

When the width and/or height of a parent subband is equal to one, the row and/or column energy weight is no longer computed as described in section 2.3.3.2.1.3.1. Instead, the row and/or column energy weight of the parent subband is simply multiplied by the scaling factor applied during a one-dimensional wavelet synthesis (see section 2.3.3.1.2.5). If the sample is low-pass, the factor (L*) is 1.0; and if the sample is high-pass, the factor (H*) is 2.0. Again, we assume that both the row and column of the top-left corner of the tile on the reference grid are integer multiples of $2^{number\ of\ decompositions}$, and this allows only low-pass samples to have one-dimensional parents. Table 2.9 shows the energy weights for the 4 x 16 tile case. As was the case with Table 2.7, only the shaded energy weights are needed for a five level decomposition. The remaining weights are included for completeness.

Table 2.9. Energy weight calculations for 4 x 16 tile, five level 9-7I wavelet decomposition

Decomposition Level	Subband Orientation	Energy Weight	Aggregate Filter L2 Norm	
1	LL	1.965907	L	1.402108
	HL	2.022573	H	1.442523
	LH	2.022573		
	HH	2.080872		
2	LL	4.122410 (LL · LL)	LL	2.030372
	HL	3.993625 (HL · LL)	HL	1.966943
	LH	3.993625 (LL · HL)		
	HH	3.868863 (HL · HL)		
3	LL	5.890440 (L*·LL · LLL)	LLL	2.901163
	HL	0.0 (empty)	HLL	2.883925
	LH	5.855440 (L*·LL · HLL)	L*	1.0
	HH	0.0 (empty)		
4	LL	8.355557 (L*·L*·LL · LLLL)	LLLL	4.115285
	HL	0.0 (empty)	HLLL	4.147521
	LH	8.421010 (L*·L*·LL · HLLL)	L*	1.0
	HH	0.0 (empty)		
5	LL	8.355557 (L*·L*·L*·LL · L*·LLLL)	L*	1.0
	HL	0.0 (empty)		
	LH	0.0 (empty)		
	HH	0.0 (empty)		

Recommendation

It is possible that the image dimensions and 1,024 pixel x 1,024 pixel tiling will lead to small tiles whose dimensions do not support a fully populated wavelet decomposition (see Section 2.3.3.1.3.3). The case of small tiles and one-dimensional or empty subbands requires special handling with respect to quantization step sizes. If the calculated energy weights for a tile do not match those of the full tile (as specified in the main header QCD marker segment), a tile-header QCD marker segment must be used to override the main header QCD. For those subbands that are empty (i.e., the row and/or column dimension is zero), the energy weight is not needed, and for convenience this may be expressed as a weight of 0.0. As a good programming practice, the quantization parameters (e_b, m_b) for an empty subband shall be set to (0,0).

2.3.3.2.2 Visual Weighting (Annex J.12)

The base step size procedure described above operates on the principle that the error contributions from each quantized wavelet subband should be equal. The human visual system (HVS) is not linear in its response to quantization noise, and improved visual quality can sometimes be achieved with the use of *visual weights*. Visual weights are used to modify the subband quantization step sizes according to HVS principles. The use of visual weights will increase the mean squared error between the original and decoded image. Equation 2.25 illustrates the use of visual weights in determining quantization step sizes for base step size quantization. A new subband specific weighting factor, VW_b, has been added to adjust the quantization step size for HVS perceptual improvement.

$$\overline{\Delta}_b = \frac{\Delta_{base} \cdot gain_b}{VW_b (CF_{L2b})(RF_{L2b})}$$

Equation 2.25

The use of visual weights for this system's imagery is TBR. If they are used, they will be supplied separately and multiplied into the calculated base step sizes.

2.3.3.2.3 5-3R Wavelet

For the 5-3R wavelet transformation, no quantization is performed. The quantization step size for every wavelet subband, Δ_b, is defined to be one ($\Delta_b = 1$). The QCC and QCD marker segments are still used with the 5-3R wavelet. These marker segments are used to signal the reversible dynamic range of the wavelet coefficients in each subband.

2.3.3.2.3.1 Forward Wavelet Coefficient Quantization Procedure (Annex E.2)

The forward quantization procedure (or simply "quantization") is described in Annex E.2. This is an informative section of the JPEG 2000 standard, but it is a normative part of this document. To enable us to use the same rate control procedures presented in section 2.3.3.2.1.1of this document, we shall multiply the wavelet coefficients generated by the 5-3R wavelet transformation, $a_b(u,v)$, by a factor of 2^6 as shown in Equation 2.26.

Equation 2.26 is simply Equation 2.16 where we have set, $\Delta_b = 1$; thus the $\tilde{q}_b(u,v)$ integers are not really quantized 5-3R wavelet coefficients – they are simply the 5-3R wavelet coefficients multiplied by 64. This allows the same piece of code to perform rate-distortion optimization for both the 9-7I and 5-3R wavelet transformations. As was done with the 9-7I wavelet coefficients, these six "fractional" bit-planes are not encoded. There is no requirement that an implementation operate in this fashion; however, the engineering code for the standard imagery test data set is written this way.

$$\begin{aligned}\tilde{q}_b(u,v) &= 2^6 \cdot a_b(u,v) \\ &= \text{sign}\left(a_b(u,v)\right) \cdot \left\lfloor 2^6 \cdot \frac{|a_b(u,v)|}{1} \right\rfloor\end{aligned}$$

Equation 2.26

Several of the parameters in the QCD and QCC marker segment for the current tile are interpreted differently for the 5-3R wavelet transformation. In Annex A.6.4 and Annex A.6.5 of ISO/IEC IS 15444-1, it is somewhat unclear which tables and parameter choices apply for the 5-3R wavelet. The "no quantization" and "reversible" tables and parameter values should be used. The Sqcd/Sqcc parameter is set to "xxx00000", which indicates that no explicit quantization is performed. Even though there are no quantization step sizes (i.e., $\Delta_b = 1$), the e_b quantization parameter is still signaled. The interpretation of the quantization parameters, (e_b, m_b), changes for the 5-3R wavelet. The m_b parameter is not used at all and it is not signaled in the QCD or QCC marker segment. The e_b parameter is used to signal the reversible dynamic range of each subband. For each subband, the value of e_b is given by (Equation E.10):

$$\begin{aligned}e_b &= R_I + \log_2\left(gain_b\right) + z_c \\ &= R_b + z_c\end{aligned}$$

Equation 2.27

Note - Table A-29 in Annex A.6.4 incorrectly references Equation E.5; it should reference Equation E.10 instead.

Whereas the R_b, R_I, and $gain_b$ of Equation 2.27 have the same meaning they had in Equation 2.20, the variable, z_c, accounts for expansion of the component bit depth if the reversible multiple component transform (RCT) is used (z_c = 1). The RCT transforms RGB components into an approximation of a YC_bC_r space to aid in compression performance. The RCT is applicable *only* to three-band RGB color images encoded using the 5-3R wavelet and shall not be used for imagery not of this nature. Therefore we set, $z_c = 0$, in Equation 2.27 when dealing with other than three-band RGB imagery. The increase in wavelet coefficient bit depth as a function of subband orientation, $gain_b$, is given in Table E-1, the same values as that for the 9-7I wavelet. This is due to the fact that the 9-7I (1,2) normalization is the same as that of the 5-3R wavelet.

2.3.3.2.3.2 Inverse Wavelet Coefficient Quantization Procedure (Annex E.1)

The inverse wavelet coefficient quantization procedure is normatively described in Annex E.1 of ISO/IEC IS 15444-1. For a recommended *decoder*, the procedures described in Annex E.1 are normative. However, for the recommended compressor, only the previously described forward wavelet coefficient quantization procedure is normative. Section 2.3.3.2.3.2 is included here strictly as an informative section.

The inverse quantization procedures for the 5-3R wavelet transformation are very similar to those described for the 9-7I wavelet transformation. For each wavelet transform coefficient, $a_b(u,v)$, the decoder constructs the integer representation, $\overline{q}_b(u,v)$, of this coefficient at the current number of decoded bit-planes using Equation 2.17 (Equation E.1). The parameters $N_b(u,v)$, $s_b(u,v)$, $MSB_i(b,u,v)$, and M_b have the same meaning when encoding with the 5-3R wavelet as they do for the 9-7I wavelet. The number of bit-planes used to represent the wavelet coefficients, M_b, is given by Equation 2.18. This is why the e_b quantization parameter is always signaled, even for the 5-3R wavelet.

Once the approximation to the integer representation of the quantized wavelet coefficient, $\overline{q}_b(u,v)$, has been determined, the reconstructed wavelet coefficient, $Rq_b(u,v)$, may be computed. If $N_b(u,v) = M_b$, then the proper relationship is given by Equation 2.28 (Equation E.7).

$$Rq_b(u,v) = \overline{q}_b(u,v)$$

Equation 2.28

If $N_b(u,v) < M_b$, the reconstructed wavelet coefficient, $Rq_b(u,v)$, is given by Equation 2.29 (Equation E.8 – note that Equation E.8 has Δ_b in the equation, but $\Delta_b = 1$ and it has been omitted here).

$$Rq_b(u,v) = \begin{cases} \left\lfloor \overline{q}_b(u,v) + r2^{M_b - N_b(u,v)} \right\rfloor & \text{for } \overline{q}_b(u,v) > 0 \\ \left\lceil \overline{q}_b(u,v) - r2^{M_b - N_b(u,v)} \right\rceil & \text{for } \overline{q}_b(u,v) < 0 \\ 0 & \text{for } \overline{q}_b(u,v) = 0 \end{cases}$$

Equation 2.29

The reconstruction parameter, r (see Equation 2.29), was mentioned in Section 2.3.3.2.1. It allows a decoder the flexibility of adjusting the reconstruction level. For the 5-3R wavelet transformation, once all of the bit-planes for a wavelet coefficient have been received, it is important that we set $r = 0$, since the wavelet coefficient is now exact. This is what Equation 2.28 is telling us to do. If all bit-planes have not been received by the decoder then we may use a value of r other than 0. The JPEG 2000 standard does not specify a value for r, but for the recommended implementations the value of r will be 0.5.

For example, if the value of a given wavelet coefficient is 22, but the decoder has not received the last two bit-planes (i.e. $M_b(u,v) - N_b(u,v) = 2$); the decoder knows that the wavelet coefficient is in the set [20, 21, 22, 23]. If we set $r = 0$, the decoder will guess the wavelet coefficient to be equal to 20. If we set $r = 0.5$, the decoder will guess 22, which is closer to the midpoint of the range [20, 23]. Under some circumstances alternative values of r may produce decoded imagery that are more visually pleasing or have a lower mean squared error relative to the original images. Note that if no magnitude bits are available for the wavelet coefficient under consideration ($N_b(u,v) = 0$), the reconstructed quantized wavelet coefficient value shall be zero.

2.3.3.3 Code-Block Entropy Coding (Annex C)

For each code-block, quantized coefficients are entropy coded to form a single compressed bit stream. This is referred to as Tier 1 (T1) coding, and represents the bulk of the complexity of the overall coding algorithm. Each code-block is entropy coded independently, using context-based adaptive binary arithmetic coding to code the coefficients bit-plane by bit-plane.

Given a code-block, the number of initial all-zero bit-planes is calculated. This information is temporarily stored for later inclusion in the codestream as header information (described in more detail in Section 2.3.3.5), and these bit-planes are subsequently skipped. The code-block is then encoded bit-plane by bit-plane beginning with the most significant bit-plane.

The coding of each bit-plane is divided into three coding passes: significance propagation, magnitude refinement, and clean-up. For a single bit-plane, each coefficient is encoded in exactly one of the three coding passes. In the most significant bit-plane, the first two coding passes (significance propagation and magnitude refinement) are skipped, and all coefficients are coded using only a clean-up coding pass. All remaining bit-planes are coded using all three coding passes in the following order: significance propagation, magnitude refinement, and clean-up.

Each time a coefficient bit-plane (i.e. a one-bit magnitude symbol) is encoded, a context is formed based on the currently coded values of neighboring coefficients as well as previous bit-planes of the current coefficient. The context is provided to the arithmetic coder – the MQ coder – along with the symbol to be encoded. The MQ coder maintains an adaptive binary probability estimate for each context, and efficiently codes the one-bit symbol according to the probability estimate.

The necessary internal information for coding each bit-plane can be efficiently represented with five bits for each coefficient:

1) significance of the coefficient (a coefficient is initially insignificant, and becomes significant when its first '1' magnitude bit is coded)
2) whether the coefficient has been refined at least once
3) sign of the coefficient
4) magnitude bit for that bit-plane
5) flag to indicate if the coefficient has been coded yet for this bit-plane

These five pieces of information are used to determine in which bit-plane coding pass each coefficient is coded, as well as to determine the context for each coefficient.

2.3.3.3.1 Bit-Plane Coding Passes (Annex D.3)

The bit-plane coding algorithms are causal so that the decoder can exactly reproduce the various context formations and probability estimates produced by the encoder. While the most important points are outlined here, the JPEG 2000 standard contains the exact details necessary to implement the coding pass encoder algorithms.

For each coding pass, the coefficients are scanned in groups of four rows, when possible (at the bottom of a code-block, the last grouping may contain fewer than four rows), starting at the top of the code-block. Within that grouping of four rows, the coefficients are scanned one column at a time, with the columns scanned left to right. Within a column, the four (unless constrained by the bottom of the code-block) coefficients are scanned top to bottom. After the entire four rows have been scanned, the group of four rows immediately beneath the current group is processed next. Figure 2.28 shows this scanning order pictorially:

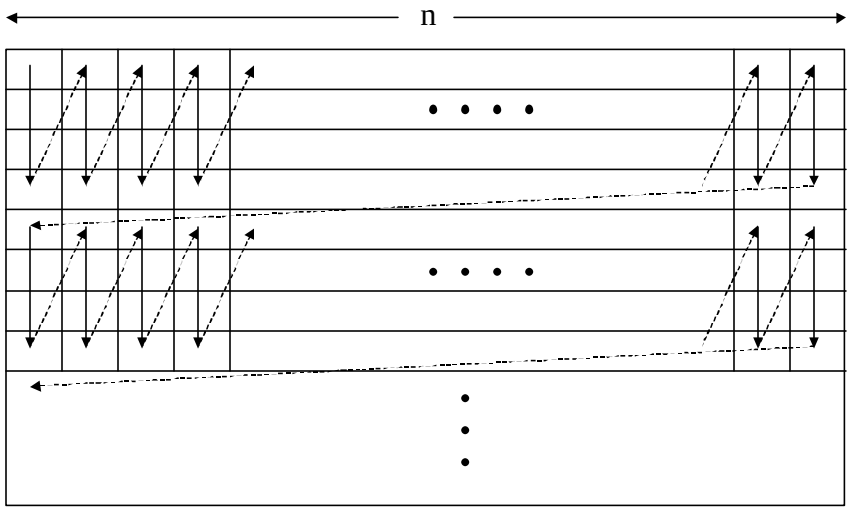

Figure 2.34. Scanning order within a code-block

The neighbors considered in any context formation are the eight neighbors shown in Figure 2.35 below. The central coefficient, X, is the current coefficient being coded. If the coefficient being encoded lies on a code-block boundary

such that some of the eight neighbors do not exist, those neighbors are assumed to be insignificant when determining the context. The significance and sign of each neighbor is based on its current coded value, and thus includes any updates made during the current coding pass.

D_0	V_0	D_1
H_0	X	H_1
D_2	V_1	D_3

Figure 2.35. Neighbors used to form context.

2.3.3.3.1.1 Significance Propagation (Annex D.3.1)

The significance propagation pass is intended to code a bit-plane for coefficients that are likely to become significant in the current bit-plane. The coefficients are scanned in the previously described order. Coefficients whose significance state bit has been set already (they became significant in a previous bit-plane) are skipped – they will have their current bit-plane encoded during the magnitude refinement pass. Otherwise, a context is formed considering the significance of the eight neighbors of the current coefficient. The $2^8 = 256$ initial contexts are quantized to nine possible contexts. One of the contexts corresponds to the case when all eight of the neighbors are insignificant. For this context, the coefficient is skipped, and is later encoded during the clean-up pass. Otherwise, the quantized context is sent to the MQ coder, along with the actual bit to be encoded. If the bit is '1', signaling significance, the sign bit for that coefficient is coded immediately afterwards.

The sign bit is encoded similarly to the significance bit. A context is formed using the sign of four neighbors: H0, H1, V0, and V1. Each may be positive, negative, or not coded yet. The $3^4 = 81$ initial contexts are quantized to five final contexts. The actual bit encoded ('1' indicates the coefficient is negative, '0' indicates positive) is the exclusive-or (XOR) of the sign bit with a second predictor bit:

$$D = signbit \otimes XORbit$$

Equation 2.30

where *D* is the bit sent to the MQ coder, *signbit* is the sign of the current coefficient, and the *XORbit* is derived from the initial context. Thus, *D*, along with the quantized context, is sent to the MQ coder to code the sign information.

If the coefficient is significant, its internal state is changed to indicate that it has become significant. Also, its internal state is updated to indicate the sign of the coefficient. Regardless of the significance of the coefficient, the internal state flag is set to indicate that this coefficient has been coded already for this bit-plane (this is necessary to keep the magnitude refinement pass, which now recognizes the coefficient as significant, from trying to code this coefficient as well).

The lookup tables that define how the various contexts are formed and quantized can be found in the JPEG 2000 standard document.

2.3.3.3.1.2 Magnitude Refinement Pass (Annex D.3.3)

After the significance propagation pass is complete, the code-block is scanned again, in the same order as described earlier. Any coefficient that is either insignificant or already flagged as having been coded in this bit-plane is skipped. All other coefficients have a magnitude refinement bit coded. The context is formed based on the significance of eight neighbors as well as the internal state bit indicating whether or not this coefficient has been refined previously. The context, along with the refinement bit, is sent to the MQ coder. If necessary, the internal state bit for this coefficient is then adjusted to reflect that it has been refined at least once. It is not necessary to set the flag indicating that this coefficient has been coded for this bit-plane, since the remaining clean-up pass ignores all significant coefficients. The lookup table that defines the various quantized magnitude refinement contexts can be found in the JPEG 2000 standard document.

2.3.3.3.1.3 Clean-Up Pass (Annex D.3.4)

After the magnitude refinement pass is complete, the code-block is scanned again, in the same order as described earlier. Any coefficient that is either significant or flagged as having been coded already in this bit-plane is not coded. Contexts are formed as for significance propagation, taking into account all new significance information coded during the significance propagation pass from the same bit-plane. Thus the context formed for a coefficient during the clean-up pass may differ from the context formed for the same coefficient during the significance propagation pass.

An additional run-length coding step is used to exploit the high probability that coefficients will be insignificant. Coefficients are processed in groups of four (if possible) corresponding to one column within one strip of four rows. If there are fewer than four coefficients (last strip of rows contains fewer than four rows) or one of the four coefficients does not currently have the zero context (the context associated with not having any significant eight-neighbors), or one of the coefficients already has been coded this bit-plane, then run-length coding is not used, and this column of up to four coefficients is coded using the same procedure as for significance propagation. If, however, all four coefficients are to be coded during the clean-up pass and all four coefficients have the zero context, then one run-length symbol is encoded to indicate whether or not all four coefficients remain insignificant. A ‘0’ is coded if all four coefficients remain insignificant; a ‘1’ is coded otherwise. The symbol, along with the unique run-length context, is sent to the MQ coder. If all four coefficients are insignificant, the encoder moves on to the next column. Otherwise, two symbols must be encoded to indicate which of the four coefficients is the first (starting from the top) to become significant. That is followed by coding the sign of that coefficient using the significance propagation sign contexts, and then encoding the remaining coefficients of that column (there may be zero, one, two, or three) using the regular significance propagation contexts. Any coefficient that becomes significant also has its internal state bits set to reflect its sign and that it is now significant.

At the conclusion of this pass, in preparation for coding the next bit-plane, all coefficients have their internal state flag set to indicate that they have not been coded yet at the current bit-plane. Note also that this flag must be initialized prior to the first coding pass (which is a clean-up pass) since all coefficients are coded during this first pass. The lookup tables used for encoding symbols related to the clean-up pass can be found in the JPEG 2000 standard document.

2.3.3.3.2 MQ Coder (Annex C)

The MQ coder is a binary adaptive arithmetic coder. It takes as input a sequence of (symbol, context) pairs, and produces a compressed bit stream. During coding, it outputs to the compressed bit stream whole bytes at a time and thus naturally produces a final compressed codestream that is an exact number of bytes. For JPEG 2000, there are 18 coding contexts in addition to a uniform context. Nine contexts are used for significance coding, five contexts are used for sign coding, three contexts are used for magnitude refinement and one context is used for run-length coding during the clean-up pass. Each context has associated with it a probability estimate that is represented through a finite state machine. As symbols are encoded, the state associated with a particular context may be updated, changing the probability estimate. The context states are initialized prior to the first coding pass. The details of the MQ coder can be obtained from the JPEG 2000 standard.

Termination of the MQ coder is performed using the FLUSH routine outlined in the standard (Annex C.2.9). When used at the end of a coding pass, the FLUSH routine terminates encoding operations and outputs to the bit-stream enough bytes so that the decoder can correctly decode the current coding pass. Terminated coding passes end on byte boundaries. This fully flushed codestream, however, is typically several bytes longer than necessary for the decoder. Therefore, a more efficient codestream may be achieved by appropriate truncation of the fully flushed codestream. This is accomplished via a near optimal length computation, as described in terms of rate estimation in Section 2.3.3.3.4.1.

In normal operation, the MQ coder associated with each code-block is terminated only once, at the end of the final coding pass. The context states for each code-block are also initialized only once, prior to the beginning of the first coding pass. If selective arithmetic coding bypass is enabled, though, the MQ coder associated with each code block is terminated after each cleanup pass starting with the fourth bit-plane coded. In this case, due to the frequent termination, a less complex termination algorithm is desirable, and it may be useful to consider the predictable termination algorithm (Annex D.4.2), even though it is on average one bit less efficient than the full flush followed by near optimal length truncation.

2.3.3.3.3 Entropy Coding Options (Annex A.6.1)

JPEG 2000 allows several entropy coding options that can be selected in any combination. These options are used to increase error resilience, allow parallel processing, and decrease computational complexity. The presence of these options in the codestream is signaled in the COD markers (see Section 2.3.3.9). These options include the following:

1) Selective arithmetic coding bypass
2) Reset context probabilities on coding pass boundaries
3) Termination on each coding pass
4) Vertically causal context
5) Predictable termination
6) Segmentation symbols

Note: Further image quality studies need to be performed to see if the selective arithmetic coding bypass mode should be employed for this system. This entropy coding option may reduce the computational burden on the processing system. If enabled, the significance propagation and magnitude refinement passes are coded in raw form starting with the fifth most significant bit plane for a code-block. Since all the information being coded is binary, coding in raw form simply implies that either a ‘1’ or ‘0’, corresponding to the bit value in current coefficient bit-plane, is written directly to the codestream. This eliminates the need to compute contexts and bypasses the MQ coder as well. Results of testing performed by the JPEG committee shows that while the arithmetic coding bypass mode can be invoked without impacting visual quality for many classes of imagery, certain imagery types *are* negatively impacted by this option. Further experimentation needs to be performed to see if this option is useful for this system’s imagery.

2.3.3.3.4 Rate-Distortion Estimation (Annex J.14)

In order to perform post-compression rate-distortion optimization, the encoder must calculate and temporarily store rate and rate-distortion slope information for each coding pass of each code-block.

2.3.3.3.4.1 Rate Estimation

At the end of a coding pass coded using the raw coder (arithmetic coding bypass), the total rate can be calculated explicitly as the number of bytes output to the codestream plus one if there is a partially filled byte. A partially

filled byte can only occur at the end of a significance propagation pass, since the raw coder is terminated at the end of magnitude refinement passes.

On the other hand, if the arithmetic coder is not terminated at the end of the coding pass, or if the coding pass is terminated using the FLUSH routine of the MQ coder, the codestream must be appropriately truncated. A conservative rate estimate is used that guarantees enough data is available to decode the desired coding passes. Although a very simple estimate may be calculated directly using only L_k and CT_k, the resulting length is very conservative, typically two or more bytes longer than necessary for correct decoding. This may be acceptable for some applications, but at low bit rate layers this overly conservative estimate may significantly degrade image quality. Instead, we will use a more complicated algorithm that computes near optimal truncation lengths. In fact, this algorithm will report the optimum truncation length under all conditions except if more than one extra byte is initially pushed out of the byte buffer, which is an extremely rare circumstance.

The algorithm to calculate a near optimal truncation length, R_k, for each such coding pass, k, is as follows:

1. Let *C_lower*, *B_lower*, *C_upper*, and *B_upper* be variables which represent lower and upper bounds for the *C* and *B* registers in the MQ coder:
   $C\_lower = C_k$
   $C\_upper = C_k + A_k$
   $B\_lower = B\_upper = B_k$
2. Normalize to the state corresponding to *CT=0* and deal with the carry bits:
   Shift *C_lower* left by the count in CT_k
   Shift *C_upper* left by the count in CT_k
   If the carry bit (0x08000000) is set in *C_lower*
      Reset the carry bit and increment *B_lower*
   If the carry bit (0x08000000) is set in *C_upper*
      Reset the carry bit and increment *B_upper*
3. Set the initial length $R_k = L_k$, and set *Curr_byte* = the last byte pushed out to the codestream
   (Note: some implementations may delay the output of 0xFF bytes, using a flag *E = 1*; for such implementations, the initial length would be calculated as $R_k = L_k + E_k$.)
4. Test for sufficiency of the current length:
   If (*Curr_byte* = 0xFF) and (*B_lower* < 128) and (*B_upper* > 127)
      We have enough bytes; in fact, we do not need the last byte, which was 0xFF
      Set $R_k = R_k - 1$ and finish
   If (*Curr_byte* != 0xFF) and (*B_lower* < 256) and (*B_upper* > 255)
      We have enough bytes so finish
5. Retrieve the next byte from the codestream, and update the bounds and byte count as follows:
   Curr_byte = next byte from the actual codestream (i.e. the byte following the first R_k bytes)
   $B\_lower = B\_lower - Curr\_byte$
   $B\_upper = B\_upper - Curr\_byte$
   $R_k = R_k + 1$
   If (*Curr_byte* = 0xFF)
      Shift *B_lower* left by 7 bits
      Load the vacant bits of *B_lower* with the 7 MSBs (bits 20 – 26) of *C_lower*
      Shift *C_lower* left by 7 bits
      Shift *B_upper* left by 7 bits
      Load the vacant bits of *B_upper* with the 7 MSBs (bits 20 – 26) of *C_upper*
      Shift *C_upper* left by 7 bits
   Else
      Shift *B_lower* left by 8 bits
      Load the vacant bits of *B_lower* with the 8 MSBs (bits 19 – 26) of *C_lower*
      Shift *C_lower* left by 8 bits
      Shift *B_upper* left by 8 bits
      Load the vacant bits of *B_upper* with the 8 MSBs (bits 19 – 26) of *C_upper*
      Shift *C_upper* left by 8 bits
6. Loop back to step 4.

L_k represents the total number of bytes output to the bit stream through the first k coding passes of the code-block. CT_k is the counter which identifies the number of additional renormalizing shifts of the A and C registers before some of the C register bits will need to be pushed out into the byte buffer, at the end of the k^{th} coding pass. The byte buffer B, the A and C registers, and CT_k are discussed in detail in Annex C of the JPEG 2000 standard, which describes the MQ coder.

2.3.3.3.4.2 Distortion Estimation (J.14.4)

The distortion reduction resulting from encoding each coding pass is calculated. This value is divided by the rate associated with the coding pass to determine the rate-distortion slope. The slope value is used during optimization to decide when the coding pass is included in the overall image codestream. Distortion-reduction estimates can be calculated with two small lookup tables that are independent of the coding pass, bit-plane or subband involved.

Following the notation of the JPEG 2000 standard document, let $w_i \Delta_i^2$ denote the distortion contribution from a single step-size error from a coefficient in code-block B_i. Δ_i^2 is the squared value of the quantization step-size for the relevant subband (the absolute step size, Δ_b, divided by the subband analysis gain, $gain_b$), while w_i is a weighting factor, computed from the L2 norm energy weights (CF_{L2b} and RF_{L2b}) of the subband's wavelet synthesis waveform, and multiplied by a visual weighting (VW_b) if required:

$$w_i = \left(CF_{L2b} \cdot RF_{L2b}\right)^2 \cdot VW_b^{\,2}$$

$$\text{Comment} = w_i \Delta_i^2 = \Delta_i^2$$

Equation 2.31

Visual weightings are input by the user as a set of weights, one for each subband, and these values are squared (per Equation 2.31) prior to multiplying by Δ_i^2. The weights are not coded in the codestream, but are used to adjust distortion estimates and subsequently affect the ordering of coding passes in the codestream. For the recommended VL compression, the same subband weights are used for visual weighting of the base step size, as described in Section 2.3.3.2.2. Note that the distortion estimation modification is necessary for proper visual weighting during rate control even though the quantization was also visually weighted.

Define $v_i[m,n]$ to be the fractional representation of the magnitude of quantized coefficient $[m,n]$. $v_i[m,n]$ contains both the integer value (the quantization index) and the fractional value (the six least significant bits that are stored beyond the quantization step-size accuracy) of the quantized coefficient magnitude. Although these six bits are not explicitly coded, and are unrecoverable by the decoder, they are used to calculate distortion-reduction estimates. Suppose bit-plane p is being coded, where $p = 0$ corresponds to the least significant bit-plane to be coded, and define

$$v_i^p[m,n] = 2^{-p} v_i[m,n] - 2\left\lfloor \frac{2^{-p} v_i[m,n]}{2} \right\rfloor$$

Equation 2.32

Thus $v_i^p[m,n]$ represents the normalized difference between the magnitude of the coefficient and the largest quantization threshold in the previous bit-plane that does not exceed this magnitude. It can be seen that $0 \le v_i^p[m,n] < 2$. Specifically, $v_i^p[m,n]$ can be thought of as having the representation x.yyyyyy...y, where x

is the coefficient value in the current bit-plane, and yyyyyy...y are the subsequent bit-planes. The use of six least significant bit-planes guarantees that for any bit-plane *p*, there are at least six subsequent bit-planes that can be used for distortion-reduction estimation.

For example, suppose the fractional representation of the magnitude of the quantized coefficient is $v_i[m,n]$ = 10001001011011.101111. For bit-plane p=8, application of Equation 2.32 yields $v_i^p[m,n]$ = 0.01011011101111. Likewise, for the least significant bit-plane p=0, $v_i^p[m,n]$ = 1.101111.

If the current coefficient becomes significant in this bit-plane, the distortion reduction can be expressed as:

$$2^{2p} w_i \Delta_i^2 [v_i^p[m,n]^2 - (v_i^p[m,n] - 1.5)^2] = 2^{2p} w_i \Delta_i^2 f_s(v_i^p[m,n])$$

Equation 2.33

Thus, f_s is the normalized MSE decrease for the significance pass:

$$f_s(v_i^p[m,n]) = v_i^p[m,n]^2 - (v_i^p[m,n] - 1.5)^2$$

Equation 2.34

This value assumes a midpoint reconstruction level (the normalized coefficient, originally restricted to the range [0,2), becomes significant and therefore is in the range [1,2), so a midpoint reconstruction has the value 1.5).

Similarly, a coefficient being refined in bit-plane *p* has a distortion reduction expressed as:

$$2^{2p} w_i \Delta_i^2 [(v_i^p[m,n] - 1)^2 - (v_i^p[m,n] - k)^2] = 2^{2p} w_i \Delta_i^2 f_m(v_i^p[m,n])$$

Equation 2.35

where *k* is 1.5 if $v_i^p[m,n] \geq 1$ and *k* is 0.5 otherwise. Note that the subscript *m* in the function *f* stands for 'magnitude', distinguishing it from f_s, and is unrelated to the coefficient location [*m,n*].

Thus, f_m is the normalized MSE decrease for the magnitude refinement pass:

$$f_m(v_i^p[m,n]) = \begin{cases} [(v_i^p[m,n] - 1)^2 - (v_i^p[m,n] - 1.5)^2], & v_i^p[m,n] \geq 1 \\ [(v_i^p[m,n] - 1)^2 - (v_i^p[m,n] - 0.5)^2], & v_i^p[m,n] < 1 \end{cases}$$

Equation 2.36

The reference software uses simple seven-bit (128 entry) look-up tables to store values for f_s and f_m. The current bit-plane plus the first six fractional bits of $v_i^p[m,n]$ can be used as an entry into the table. [Actually, since the significance pass only applies when the current bit-plane is 1, a six-bit table could be used for f_s.] The distortion-reduction estimate over the entire code-block is calculated as:

$$\sum_{m,n} 2^{2p} w_i \Delta_i^2 \hat{f}(v_i^p[m,n]) = 2^{2p} w_i \Delta_i^2 \sum_{m,n} \hat{f}(v_i^p[m,n])$$

Equation 2.37

where the accumulation term $\sum_{m,n} \hat{f}(v_i^p[m,n])$ is summed over all *(m,n)* in the code-block. The function $\hat{f}$ is either f_m or f_s, depending on the coding pass.

Note that the reference software computes the distortion-reduction estimations using floating-point arithmetic. An alternate implementation might scale the output of $\hat{f}$ by 2^{13} so that the tables can be represented with a 16-bit integer, and then the summation over the entire code-block may be computed in a 32-bit accumulator without risk of overflow.

The look-up tables contain exact distortion-reduction values for the respective seven bit inputs. The look-up table values become estimates of the actual distortion reduction when the coefficient being coded has more than six fractional bit-planes of data in its representation as $v_i^p[m,n]$. Only the first six fractional bits are used to access the look-up table, and hence the returned value is only an approximation of the actual distortion-reduction.

2.3.3.3.4.2.1 Distortion Estimation Modifications for Reversible Transforms (Annex J.14.4.2)

When a reversible wavelet transform is used, the wavelet coefficients have integer values and midpoint reconstruction makes no sense for distortion estimations involving the final bit-plane. For the final bit-plane, the distortion estimation formulas for significance and refinement coding can be modified to account for the fact that there will be no quantizer error and hence no distortion remaining after the final bit-plane is coded. For significance coding, the distortion reduction in the final bit-plane is given by:

$$2^{2p} w_i \Delta_i^2 v_i^p[m,n]^2$$

Equation 2.38

For magnitude refinement coding, the distortion reduction in the final bit-plane is given by:

$$2^{2p} w_i \Delta_i^2 (v_i^p[m,n] - 1)^2$$

Equation 2.39

2.3.3.3.4.3 Computation of the Rate-Distortion Convex Hull (Annex J.14.3)

After a code-block has been compressed, each coding pass has associated with it a rate and a rate-distortion (RD) slope value. The RD-slope value is a floating point value obtained by dividing the weighted distortion-reduction estimate by the rate. The post-compression rate-distortion optimization algorithm uses these RD-slope values to determine when to include coding passes in the codestream. Specifically, when forming a layer, an RD-slope threshold is used to determine how many coding passes from each code-block to include in the current layer. In order to efficiently determine this contribution, the set of valid coding pass truncation points for a code-block is first computed, prior to the optimization process, by forming the convex hull of the rate-distortion curve for that code-block. Coding passes that lie on the convex hull form the set of valid truncation points. These valid truncation points comprise the set of allowable points at which to truncate the contribution of a code-block to a layer. Coding passes that do not lie on the convex hull are not valid truncation points, and each is grouped with the next valid truncation point. The RD-slope associated with valid truncation points is modified to represent the overall RD-slope for all coding passes since the last valid truncation point. This approach results in a set of RD-slopes for the valid truncation points which are monotonically decreasing, thus making it easy for the optimization algorithm to determine how many coding passes to include in a layer, given a certain RD-slope threshold. The layer formation algorithm is discussed in detail in the following section (Section 2.3.3.4).

Figure 2.36 illustrates the convex hull of an example rate-distortion curve. Only those coding passes that lie on the convex hull are valid truncation points. In this example, coding pass 2 is grouped with coding pass 3. Similarly, coding pass 5 is grouped with coding pass 6.

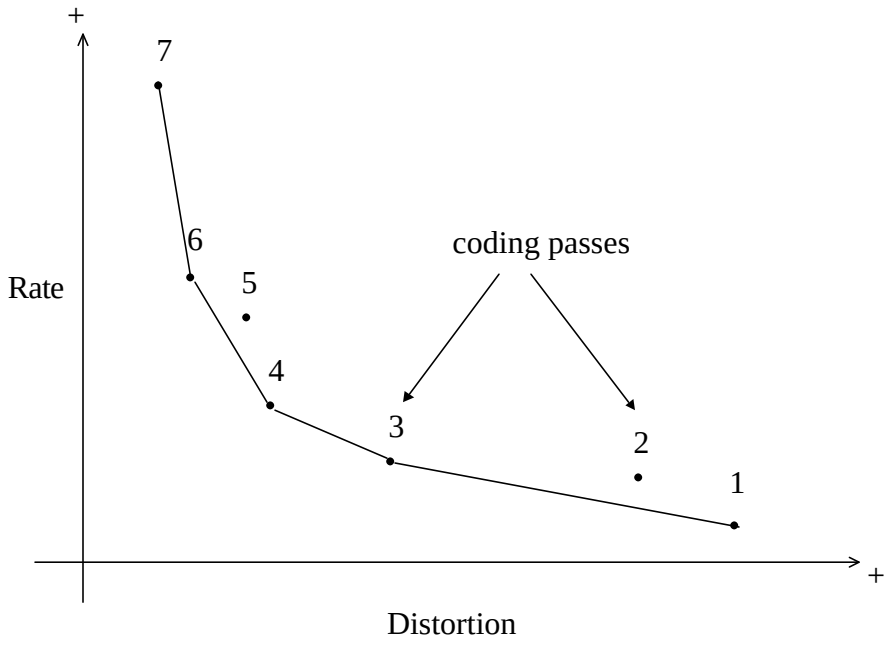

Figure 2.36. Convex hull of rate-distortion curve.

The algorithm for determining the valid truncation points for a code-block is given below. Let B_i be the code-block, and let M be the number of compressed coding passes for B_i. For k ranging from 1 to M, let R_i^k be the total rate associated with coding passes 1 through k, and let D_i^k be the total weighted distortion-reduction estimate from coding passes 1 through k. Let $N_i = \{0,1,2,...,M\}$. This is the set of all truncation points, and will be trimmed to become the set of valid truncation points. [Note that 0 and M, corresponding to including no coding passes, and including all coding passes, cannot be removed from the set N_i in the following algorithm.]

1) Set $p = 0$. Define $R_i^0 = 0$ and $D_i^0 = 0$.
2) For $q = 1$ to M
   If $q \in N_i$
      Set $\Delta R_i^q = R_i^q - R_i^p$
      Set $\Delta D_i^q = D_i^q - D_i^p$
      Set $S_i^q = \Delta D_i^q / \Delta R_i^q$
      If $p \neq 0$ and $S_i^q > S_i^p$ then remove p from N_i and go to step 1)
      Else set $p = q$.

Once the set of valid truncation points is determined, the modified rate and RD-slope value for each valid truncation point is recomputed as the overall rate and RD-slope since the previous valid truncation point.

2.3.3.4 Layer Formation (Annex B.8)

The compressed data of a code-block can be represented as a single array of bytes. This data is distributed across one or more layers in the codestream. Each layer consists of a certain number of consecutive coding passes from each code-block contained in a tile. The number of coding passes contained in a layer may vary from code-block to code-block. For any given code-block, the coding passes must appear in sequential order both within and across layers. Each successive layer can be thought of as improving the quality of each code-block.

For each tile, layers are formed to achieve target bit rates. Rate-distortion optimization is used to achieve the maximum distortion reduction for the allowed rate. For the given target bit rate, it is necessary to decide which coding passes to include in that layer, so as to maximize distortion reduction. This is done by finding the smallest RD-slope threshold, t , such that the total rate required to include all coding passes with RD-slopes greater than or equal to t is less than the allowed rate.

Layers are formed based on target bit rates. The targets are to be met independently for each tile. Thus the layer formation algorithm is independent for each tile, and it may be the case that each tile has a different threshold t at which a certain target bit rate is achieved. Each target bit rate refers to packet bytes only. Packets are discussed in detail in the following section. The main header and tile-part headers are excluded from any layer formation rate calculations. Thus the actual achieved bit rate for a certain layer may be slightly higher than the target, once the header information is considered. Tables containing the actual target bit rates are available in other documentation.

The final layer contains all remaining bits – that is, it contains all coding passes that have not previously been included in the codestream for one of the first 18 layers. For visually lossless compression a maximum per-tile bit rate of 4.3 bpp (TBR) will truncate this final layer if necessary. Essentially, this final layer is implemented as a rate-controlled layer with a target of 4.3 bpp. Due to careful selection of the quantization base step size, however, most tiles will be able to include all remaining bits prior to hitting this final truncation point. For numerically lossless compression all coding passes are included in the codestream and there is no ceiling on the per-tile bit rate of the final layer.

It is possible that the entropy of the data in a tile is small enough such that all of the tile's code-block coding passes are coded using fewer bytes than available. In this case, the final layer containing coding pass data may be smaller than the target. Additionally, any remaining layers still exist, but contain no coding pass data. Each code-block is marked as contributing zero coding passes to each of the extra layers.

For each tile and each layer, a binary search algorithm is used to determine the optimal RD-slope threshold t . The layer comprises all coding passes of all code-blocks in the tile which have not been included in a previous layer and whose RD-slope value is greater than or equal to t . The RD-slope values are converted to a 16-bit exponent/mantissa representation. The first bit is 0, the next six bits are used to represent the unsigned (positive) exponent, and the remaining nine bits are used to represent the mantissa. Using the exponent/mantissa notation, the RD-slope is represented as:

$$2^{E}(1+\frac{M}{2^{9}})$$

Equation 2.40

The six-bit exponent, *E*, can take integer values from [0,63]. The nine-bit mantissa, *M*, can take integer values from $[0,2^{9}-1]$. Thus the 16-bit exponent/mantissa notation can represent a range of RD-slopes from 1 to nearly 2^{64}. As will be seen below, this 16-bit logarithmic re-mapping of the RD-slopes allows the iterative optimization algorithm to converge on the correct threshold t more quickly than a simple floating point linear representation of the RD-slopes. This re-mapping does incur a small loss of precision in the RD-slope values. For each value of the exponent *E* in the exponent/mantissa notation, only nine additional bits of precision are used in the mantissa *M*. This loss of precision does not significantly affect the performance of the optimization algorithm.

To convert floating-point RD-slopes to exponent/mantissa notation, the following algorithm is used:

1) Set E=0
2) While E < 63

   If RD-slope $\leq 2 - 2^{-10}$ then break from while loop
   Else Set RD-slope = RD-slope / 2 and Set E = E + 1
3) If RD-slope > $2 - 2^{-10}$ then Set RD-slope = $2 - 2^{-10}$.
4) M = $\lfloor .5 + (RDslope - 1)2^9 \rfloor$
5) If M < 0 Set M = 0
6) If M > $2^9 - 1$ Set M = $2^9 - 1$.

During the conversion of RD-slope values to exponent/mantissa notation, a conservative estimate of the rate used for certain thresholds is formed. Specifically, for each exponent value, *E*, a counter array tracks how many coding pass bytes would be included if the layer had a boundary at the threshold corresponding to the exponent/mantissa pair (*E*,0). To compute the conservative estimates, the exponent, $\hat{E}$, of each re-mapped RD-slope value is observed, and all counters with an exponent $E \leq \hat{E}$ are incremented by the associated rate. Note that the conservative rate estimates are computed independently for each tile. The actual rate required for each threshold would be higher, as header information is required in addition to the compressed coding pass bytes, but the conservative estimate can be used to decrease the initial range of the binary search to find the optimal threshold t, and thus decrease computational complexity.

The iterative binary search algorithm for determining the optimal RD-slope threshold, t, at which to form the layer boundary is described below.

1) Set *hi_threshold* = *previous_layer_threshold* (or if this is the first layer, set it equal to maximum possible exponent/mantissa pair ($63,2^9-1$))
2) Set *lo_threshold* = exponent/mantissa pair (*E*,0), where *E* is the largest exponent value for which the conservative rate estimation is higher than the available rate.
3) Set $t = \left\lfloor \frac{hi\_threshold + lo\_threshold}{2} \right\rfloor$
4) While *hi_threshold* > *lo_threshold*

   Calculate rate necessary to encode layer using threshold t
   If rate necessary > available rate

      Set *lo_threshold* = t + 1

   Else

      Set *hi_threshold* = t

   Set $t = \left\lfloor \frac{hi\_threshold + lo\_threshold}{2} \right\rfloor$

At the conclusion of the algorithm, t is the optimal RD-slope threshold.

There are several comments to be made regarding this algorithm. The RD-slope values contained in *lo_threshold* and *hi_threshold* are stored in 16-bit exponent/mantissa notation, but when used in arithmetic or logical operations, are treated as regular integers (i.e. integers in the range $[0,2^{15}-1]$, since the first bit is always 0). This means that the average taken in steps 3) and 4) is not in general the arithmetic average of the two thresholds. This is not a problem, however, as the exponent/mantissa representation is an increasing function with respect to the standard integer representation (i.e. if a and b are each 16-bits, and a>b when a,b are interpreted as integers, then a>b when they are interpreted as exponent/mantissa notation as well). Thus the average operation will still converge on the optimal threshold. By representing the wide range of possible RD-slope values using only 16 bits, the binary search is guaranteed to converge in no more than 16 iterations. The algorithm terminates upon finding the smallest value of the threshold t for which the required rate is less than the available rate.

The above algorithm must also allow the possibility that a layer will have the same threshold as the previous layer. This happens in the case that the differential in bit rate between one layer and the next is too small to contain the coding passes included by decreasing the threshold at all.

As each potential threshold t is being tested, it is necessary to simulate the cost of forming a layer at this threshold. Note that the target bit rates are given in terms of coding pass information only, and do not include the cost of any main or tile-part header information. Thus when calculating the rate required to code a layer at a certain threshold, only the cost required to encode the coding passes is included. This cost is calculated as a sum of the cost of individual packets. Essentially, a packet contains compressed coding pass data from one layer of one precinct of one resolution of one tile-component, as well as a packet header, which is needed to properly decode the coding pass data. Packet headers are included in the rate required to form a layer at a certain threshold, and the construction of packet headers must be simulated for each potential threshold in order to calculate an accurate rate. Packets are discussed in detail in the following section.

In order to minimize the execution time necessary to converge on the optimal threshold t for a given layer, note that for each potential threshold, it is not necessary to calculate the exact layer cost once it is known that the cost is higher than the available rate. Also, conservative rate estimates can be used to start the iterative binary search algorithm with an accurate value for *lo_threshold*, allowing for quick convergence to the optimal threshold.

Once all layers have been formed, the following data must be retained in temporary storage for each code-block: how many coding passes are included in each layer, and how many bytes (rate) are necessary for each layer to code those coding passes. This information is used in the formation of packet headers. The RD-slope values are discarded at this stage.

2.3.3.5 Packet Formation (Annex B.9)

In addition to the grouping of data into code-blocks, precincts and layers as discussed previously, JPEG 2000 also organizes compressed data into structures called packets. A packet is a segment of the codestream that contains a packet header and the compressed image data from one layer of one precinct of one resolution of one tile-component (packet body).

The concept of a packet is central to the layer formation algorithm discussed in the previous section. For each potential RD-slope threshold examined during the binary search algorithm, it is necessary to simulate the cost of forming a layer at that threshold. This cost is the rate necessary to form the packets for that layer. For example, a system-specific image is JPEG 2000 compressed using a five-level wavelet decomposition with maximal precincts and 19 quality layers. For each layer formed, there are six associated packets that contribute to the cost of that layer – there are six resolutions, and only one precinct per resolution. Thus, there will be a total of 6 x 19 = 114 packets per tile. When the layer formation algorithm calculates the cost of forming a layer at a certain threshold, it is calculating the cost of the packets comprising that layer. This cost is the sum of both the packet headers and the packet bodies. The cost of the packet bodies at a certain threshold is easy to compute. It is simply the sum of the cost of the coding pass data, for which stored rate values have already been pre-computed. The cost of a packet header, however, must be determined by simulating its construction at the given threshold.

The compressed image data of a packet must occur contiguously as one unit in the codestream. Although JPEG 2000 does allow packet headers to be grouped separately in PPM or PPT marker segments, in this system's implementation a packet header will appear in the codestream immediately preceding the packet body (compressed image data).

The packet header is byte-aligned in the codestream. It must begin at the beginning of a byte, and it comprises a whole number of bytes. Similarly, the packet body is byte-aligned. This is clear from the fact that it contains only code-block coding pass contributions, and each code-block contribution is a whole number of bytes. Consequently, the total packet is byte-aligned as well.

Within a packet body, the code-block contributions appear in a specific order. Since a packet contains data from only one resolution, it will either contain data from just the LL band, or from all the subbands (HL, LH, HH) from a particular resolution. For packets containing data from all three (HL, LH, HH) subbands, the code-block data contributions occur one subband at a time. Code-block data from the HL subband appears first, followed by code-block data from the LH subband, followed by data from the HH subband. Within any subband, the code-block data appears in raster-scan order, restricted to the relevant precinct. The number of coding passes contributed by each code-block is determined during the RD optimization process.

Figure 2.37 illustrates how code-block coding passes may be distributed among different layers. The code-blocks in this example correspond to a two-level wavelet decomposition of a 256 x 256 tile using 64 x 64 code-blocks and maximal precincts. In this example, the outer resolution does not contribute to the first layer– each code-block from 1HL, 1LH, and 1HH contributes zero coding passes to the first layer.

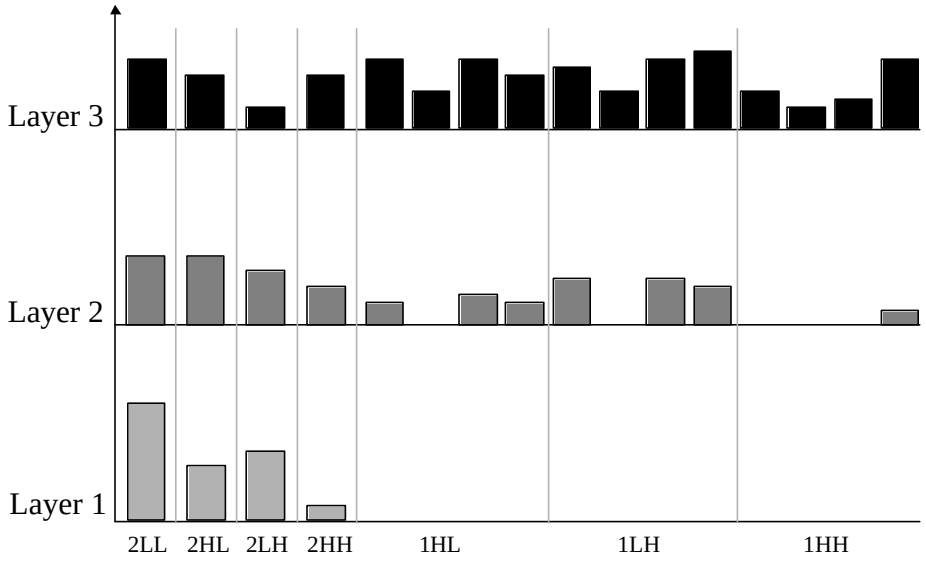

Figure 2.37. Distribution of code-block coding passes among different layers.

For a specific code-block, the entire compressed coding pass data can be viewed as a single stream of bytes, distributed among one or more packets. The coding passes must occur in sequential order in the overall codestream. Each time a code-block contributes to a packet, it is necessary only to determine where in the corresponding code-block compressed data bit stream the most recent contribution from this code-block terminated, and include the next set of bytes starting from that point.

A code-block contribution to a packet may not end with a 0xFF byte. In this case, the 0xFF byte is moved to the subsequent packet that contains data from this code-block, or dropped if no such packet exists. The corresponding rates associated with the affected coding passes must be adjusted accordingly. The rate of the final coding pass included in the current packet is decreased by one, and the rate of the next coding pass, if any, is increased by one.

Whereas the arithmetic encoding of the bit-planes is referred to as Tier 1 (T1) coding, the packetization of the compressed data and encoding of the packet header is referred to as Tier 2 (T2) coding. The majority of the computational complexity of a JPEG 2000 encoder resides in T1 coding. Codestream packets can be parsed, shuffled and reordered using only T2 decoding, and thus JPEG 2000 codestreams can be rearranged with minimal computational complexity.

2.3.3.6 Packet Headers (Annex B.10)

The packet header contains the signaling information necessary to properly parse and decode the packet body. It contains five types of information:

1) Zero-length packet (Annex B.10.3)
2) Code-block inclusion (Annex B.10.4)
3) Zero bit-plane information (Annex B.10.5)
4) Number of coding passes (Annex B.10.6)
5) Length of compressed image data from each code-block (Annex B.10.7)

2.3.3.6.1 Bit Stuffing Routine (Annex B.10.1)

Bits are packed into header bytes from most significant bit (MSB) to least significant bit (LSB). When a byte is completed, it is appended to the packet header. An 0xFF byte must be immediately followed by a zero bit placed in the MSB of the following byte, even if this occurs at the end of the packet header and requires an extra byte to be appended to the header. The last byte of the packet header is bit-stuffed with '0's as necessary.

2.3.3.6.2 Tag Trees (Annex B.10.2)

Two types of data are represented in the packet header using a data structure called tag trees. One tag tree is used to represent the layer number in which each of the code-blocks is first present. A second tag tree is used to represent the number of initially zero bit-planes for each code-block. Although each tag tree represents information for all code-blocks in the precinct, it is not encoded all at once in the packet header. The tag tree coding is distributed such that as each code-block has its header information encoded, only enough information about the tag trees is encoded to allow that current code-block to be correctly signaled for the current packet.

Since only enough information about the tag trees is included in a packet header to correctly parse the coding pass data for that packet, it is the case that tag trees are coded gradually over one or more packets. Thus the state of each tag tree must be stored after each packet, so that the next related packet (same precinct, resolution and tile-component, but next layer) can continue coding the tag trees where the previous packet left off. During the RD optimization algorithm, each time an optimal threshold is found for a layer boundary, the tag trees associated with each packet for that layer must be updated and stored to allow simulations for future layers to correctly compute packet header costs. The procedure for encoding tag trees is described below.

In general, a tag-tree is a hierarchical method of representing a two-dimensional array of non-negative integers. In JPEG 2000 packet headers, this two-dimensional array represents a precinct within a subband, with each element corresponding to a code-block within that precinct.

From the original 2-D array, a tree structure is created that contains a 2-D array for each reduced resolution, where each element in the reduced resolution stores the minimum of the (up to) four pixels in the corresponding 2x2 neighborhood in the higher resolution.

Mathematically, this can be described as follows. Let $q_n()$ be the 2-D array at resolution level n. Then, the next reduced resolution is $q_{n-1}(x,y) = \min\{q_n(2x,2y), q_n(2x+1,2y), q_n(2x,2y+1), q_n(2x+1,2y+1)\}$. The procedure is repeated recursively until the lowest resolution, $q_0()$, is generated that contains a single element.

First, consider a simple case where all information for the tag tree is coded at once. (As will be described later, the JPEG 2000 use of tag trees is not this straightforward.) Once the reduced resolution tree is generated, the original array is processed in raster order (i.e. left to right and then top to bottom). For each element in the original array, start at the root node $q_0(0,0)$, and traverse the appropriate path in the tree to the branch node in the original array. For any of these intermediate nodes that have not already been coded, calculate the difference between this node and its parent, and code a 0 bit for each increment, followed by a single 1 bit to terminate that node. I.e., 1 = no increment necessary; 01 = increment by 1; 001 = increment by 2; etc. Assume an initial value of 0 for the "parent"

of the root node. Continue for each element in the original array, skipping any intermediate node that has already been coded.

Pseudocode for this simple procedure to code all tag tree information at once is as follows:

1) Let $N = \lfloor \log_2(\max\{\#rows, \#cols\}) \rfloor + 1$
2) For n = N-2 to 0 step –1
   For x = 0 to $\left\lceil \frac{\#cols}{2^{N-1-n}} \right\rceil$ -1
      For y = 0 to $\left\lceil \frac{\#rows}{2^{N-1-n}} \right\rceil$ -1
         $q_n(x,y) = \min\{q_{n+1}(2x,2y), q_{n+1}(2x+1,2y), q_{n+1}(2x,2y+1), q_{n+1}(2x+1,2y+1)\}$
3) For i = 0 to #cols – 1
   For j = 0 to #rows – 1
      For n = 0 to N-1
         Let x = $\left\lfloor \frac{i}{2^{N-1-n}} \right\rfloor$
         Let y = $\left\lfloor \frac{j}{2^{N-1-n}} \right\rfloor$
         If $q_n(x,y)$ has not already been coded
            If n = 0
               Let d = $q_n(x,y)$
            Else
               Let d = $q_n(x,y) - q_{n-1}(\left\lfloor \frac{x}{2} \right\rfloor, \left\lfloor \frac{y}{2} \right\rfloor)$
            Code d "0" bits followed by a single "1" bit

Consider the example from Annex B.10.2. Here, we are interested in coding the following two-dimensional array of non-negative integers:

Original array (branches): $q_3()$

1	3	2	3	2	3
2	2	1	4	3	2
2	2	2	2	1	2

To start, we build the hierarchy of arrays for the reduced-resolution levels. Based on the maximum dimension of the array (width = 6), we know that we need 3 reductions to get to a 1x1 root node, so we will call the original array values $q_3(x,y)$. I.e. $q_3(0,0) = 1$; $q_3(1,0) = 3$; $q_3(2,0) = 2$. The first reduced resolution is created by taking the minimum value of each 2x2 neighborhood. I.e., $q_2(0,0) = \min\{q_3(0,0), q_3(1,0), q_3(0,1), q_3(1,1)\} = \min\{1, 3, 2, 2\} = 1$; $q_2(1,0) = \min\{q_3(2,0), q_3(3,0), q_3(2,1), q_3(3,1)\} = \min\{2, 3, 1, 4\} = 1$; etc. Since there are an odd number of rows, the bottom row of the reduced resolution is calculated on 2x1 neighborhoods. E.g., $q_2(0,1) = \min\{q_3(0,2), q_3(1,2)\} = \min\{2, 2\} = 2$.

First reduced resolution: $q_2()$

1	1	2
2	2	1

Continuing this pattern for the next reduced resolution, $q_1(0,0) = \min\{q_2(0,0), q_2(1,0), q_2(0,1), q_2(1,1)\} = \min\{1, 1, 2, 2\} = 1$. Since there are an odd number of columns, the rightmost column of the reduced resolution is calculated on a 1x2 neighborhood: $q_1(1,0) = \min\{(q_2(2,0), q_2(2,1)\} = \min\{2, 1\} = 1$.

Second reduced resolution: $q_1()$

1	1

Finally, the last reduced resolution is simply $q_0(0,0) = \min\{q_1(0,0), q_1(1,0)\} = \min\{1, 1\} = 1$.

Lowest resolution (root): $q_0()$

1

Once the reduced resolution arrays are created, we can code each element in the original array (q_3). The array is coded in raster order, so we start with $q_3(0,0)$. To get to $q_3(0,0)$, the hierarchy is traversed from root to branch in the following order: $q_0(0,0)$ -> $q_1(0,0)$ -> $q_2(0,0)$ -> $q_3(0,0)$. We start off with an initial value of 0. At node $q_0(0,0)$, we have the value 1, so our initial value of 0 must be incremented by 1. Thus, we code a single 0 bit followed by a 1. At $q_1(0,0)$, we have the value 1, which is the same as that at the parent node $q_0(0,0)$, so we don't code any 0 bits – only a 1. At $q_2(0,0)$, we again have the value 1, so the code is simply a 1 bit. Finally, at $q_3(0,0)$, we have the value 1, so again there is no incrementing and we simply code a 1. Thus, the entire code for $q_3(0,0)$ is 01111, and this includes coding of the intermediate nodes $q_0(0,0)$, $q_1(0,0)$, and $q_2(0,0)$.

Next, we need to code $q_3(1,0)$. The hierarchy is traversed in the following order: $q_0(0,0)$ -> $q_1(0,0)$ -> $q_2(0,0)$ -> $q_3(1,0)$. Now, we already have coded $q_0(0,0)$, $q_1(0,0)$, and $q_2(0,0)$, so all that remains is to code $q_3(1,0)$ based on the knowledge that its parent at $q_2(0,0)$ is 1. To code $q_3(1,0) = 3$, we need to increment $q_2(0,0)$ by 2, so we code two 0 bits followed by a 1, and the code for $q_3(1,0)$ is 001.

Next, we need to code $q_3(2,0)$. The hierarchy is traversed in the following order: $q_0(0,0)$ -> $q_1(0,0)$ -> $q_2(1,0)$ -> $q_3(2,0)$. We already have coded $q_0(0,0)$ and $q_1(0,0)$, so we begin with $q_2(1,0)$. Since $q_2(1,0) = q_1(0,0)$, we simply code a 1. Then, $q_3(2,0) = 2 = q_2(1,0) + 1$, so we code a single 0 followed by a 1. Thus, the code for $q_3(2,0)$ is 101.

If we continue this process for the rest of the array, the results are as follows:

Intermediate nodes	Leaf nodes	Code
$q_0(0,0) = 1$		01
$q_1(0,0) = 1$		1
$q_2(0,0) = 1$		1
	$q_3(0,0) = 1$	1
	$q_3(1,0) = 3$	001
$q_2(1,0) = 1$		1
	$q_3(2,0) = 2$	01
	$q_3(3,0) = 3$	001
$q_1(1,0) = 1$		1
$q_2(2,0) = 2$		01
	$q_3(4,0) = 2$	1
	$q_3(5,0) = 3$	01
	$q_3(0,1) = 2$	01
	$q_3(1,1) = 2$	01
	$q_3(2,1) = 1$	1
	$q_3(3,1) = 4$	0001
	$q_3(4,1) = 3$	01
	$q_3(5,1) = 2$	1
$q_2(0,1) = 2$		01
	$q_3(0,2) = 2$	1
	$q_3(1,2) = 2$	1
$q_2(1,1) = 2$		01
	$q_3(2,2) = 2$	1
	$q_3(3,2) = 2$	1
$q_2(2,1) = 1$		1
	$q_3(4,2) = 1$	1
	$q_3(5,2) = 2$	01

Thus, we are able to fully encode this 6x3 array of non-negative integers using a total of 44 bits. Note that this example does not fully explain the use of tag trees for JPEG 2000, which will be described next.

JPEG 2000 allows tag trees to be partially coded at any given time, and the resulting bit order in the code stream is not strictly the same as the full coding example above. While each layer processes code-blocks in a strict raster order, it may be the case that information for a certain code-block may be needed in an earlier layer than that for a code-block with a lower raster index, and in this case code bits related to the higher raster index code-block will appear earlier in the code stream. Furthermore, with JPEG 2000, at any given time the tag tree is only coded to the minimum extent necessary to answer a relevant question about whether the value at a leaf is at least some quantity. To avoid redundancy, each tag tree must maintain state information identifying the current value of each node in the tree and whether or not that node's value has been locked in (i.e. a 1 has been coded).

The pseudocode for partial tag tree coding is provided below. Assume we start with a tag tree created using steps 1 and 2 of the previous procedure for a fully coded tag tree. For each node $q_n(x,y)$, let V(n,x,y) be the current tag tree value, and L(n,x,y) be a flag indicating whether the value of this node has been locked in (1) or not (0). Initially set V(n,x,y) = 0 and L(n,x,y) = 0 for all n, x, and y. We define a procedure T(x,y,t) that, given the state stored in V and L, will code all of the partial tag tree information necessary to answer the question "is $q_{N-1}(x,y) >= t$?" as follows:

For n = 0 to N-1 *# for each level of the tag tree, starting at the root*

Let x' = $\left\lfloor \frac{x}{2^{N-1-n}} \right\rfloor$

Let y' = $\left\lfloor \frac{y}{2^{N-1-n}} \right\rfloor$

 # if node's current value < parent's value, set node's value = parent's value

If (n > 0) and (V(n,x',y') < V(n-1, $\left\lfloor \frac{x'}{2} \right\rfloor$, $\left\lfloor \frac{y'}{2} \right\rfloor$))

V(n,x',y') = V(n-1, $\left\lfloor \frac{x'}{2} \right\rfloor$, $\left\lfloor \frac{y'}{2} \right\rfloor$)

 # while node's current value < threshold, and node is not locked

While (V(n,x',y') < t) and (L(n,x',y') = 0)

 # if node's current value < node's final value, increment node's current value

If V(n,x',y') < qn(x',y')

Set V(n,x',y') = V(n,x',y') + 1

Emit a "0" bit

 # else lock the current value of the node

Else

Set L(n,x',y') = 1

Emit a "1" bit

 # if current node's value >= threshold, we know leaf is >= threshold so we are done

If V(n,x',y') >= t

Return TRUE

 # else if we finished processing the leaf node, we know that leaf is < threshold

Else if n = N-1

Return FALSE

 # otherwise, continue to the next node on the path, i.e. the next iteration of the for-loop

Note that although intuitive, the algorithm above may not be the most efficient method for encoding tag trees. Any valid implementation, though, must achieve the same bit stream as this algorithm.

2.3.3.6.3 Zero-Length Packet (Annex B.10.3)

The first information included in the packet header is a single bit to indicate if the packet has a length of zero (i.e., is empty). A '0' indicates an empty packet. In this case the packet header is one byte long (the rest of the byte is padded with zeros) and the packet body is zero bytes. The empty packet symbol may be used by bit-stream parsers to effectively discard packets from an existing bit-stream without interfering with global packet ordering constraints. It is not typically used when a packet constructed using the RD optimization algorithm happens to contain no coding passes, as may happen when two target bit rates are too close. Nor is it typically used when a packet contains no coding passes because all possible coding passes appeared in previous packets, as may occur if target bit rates for layers exist that are higher than the entropy of the wavelet coefficients. In these cases, a '1' bit is used to signal a non-empty packet, and the remainder of the packet header is used to indicate that there are no coding passes present in the packet.

The recommended JPEG 2000 encoder will always set the first bit of the packet header to '1'. In the case where empty packets for a precinct are later followed by non-empty packets for this precinct, it may actually be inefficient to code a '0', because the tag tree information for these empty packets will need to be transmitted with the later packets rather than incrementally taking advantage of the unused 7 bits in the packet header (due to byte alignment). Although there is no such issue in handling empty packets when all possible coding passes appeared in previous packets, for simplicity we implement a consistent policy where the first bit is always '1'.

The remainder of the packet header contains some information for each code-block in the relevant precinct. This information appears in the same order as the compressed data in the packet body. All header information for the HL

subband appears first, followed by header information for the LH subband, followed by information for the HH subband. Within each subband, header information is included for the relevant code-blocks (those in the current precinct) in raster-scan order, with all header information for the first code-block encoded first, and so forth.

2.3.3.6.4 Code-Block Inclusion (Annex B.10.4)

For each code-block, the first information that is coded in the packet header is whether or not data from this code-block is included in the current layer. If the code-block has already been included in any previous packet, this information is simply coded using a 0 or 1 to indicate that the code-block is or is not included in the current layer. If the code-block has not previously been included, a more complicated method using a tag tree is used to exploit redundancy.

Each precinct maintains a separate tag tree for each subband it covers. The values in the tag tree are the number of the layer (starting with 0) in which each code-block is first included. Although each bit associated with the full code of a tag tree will be used at some point in the (non-truncated) codestream, only those bits that are required for determining if the code-block is included in the current layer are placed in the packet header for this layer. If some of the tag tree is already known from previous code-blocks or previous layers, it is not repeated. Furthermore, the path to the current leaf node is only coded to the point where it can be inferred whether the code-block is included or not in this layer. It may not be necessary to finish coding the entire branch to the leaf, and it may happen that a node is only partially coded during the current layer.

For instance, suppose we are processing layer 2 for a code-block which is not included until layer 6. Furthermore, suppose that the parent of this node in the tag tree has a final value of 4, and that its parent has already been fully coded for a value of 2 (while processing a previous code-block). If we increment the current code-block's parent from 2 to 3, we already know that the current code-block cannot be included in layer 2. Thus, we would simply code a 0 to increment the current value of the parent, and we would move on to the next code-block in the precinct. As other code-blocks and layers are processed, eventually the parent node will be incremented to a final value of 4 (via additional code-bits 01), and then the leaf node will be incremented to a final value of 6 (via code-bits 001), but these bits will be sent one at a time on an as-needed basis. The decoder knows to stop reading tag tree bits as soon as it has the information needed to determine if the current code-block is included in the current layer.

2.3.3.6.5 Zero Bit-Plane Information (Annex B.10.5)

During entropy coding, the all-zero most significant bit-planes are skipped, and coding passes start at the most significant bit-plane with a non-zero element. When a code-block is included for the first time, it is necessary to signal the number of zero bit-planes that were skipped, so that the decoder may determine the significance of the first bit-plane coded for that code-block.

The number of missing most significant bit-planes is coded in the packet header with a separate tag tree for each precinct, in the same manner as the code-block inclusion information. Processing the zero bit-planes tag tree is intuitively a little more straightforward, though, since the full path to the code-block leaf is immediately required when that code-block is included for the first time. When applying the partial tag tree coding algorithm in Section 2.3.3.6.2, simply perform the procedure T with threshold $t = \infty$ to force coding all the way to the leaf.

2.3.3.6.6 Number of Coding Passes (Annex B.10.6)

The number of coding passes included from each code-block is signaled using Huffman-style codewords from Table 2.10 (Table B-4):

Table 2.10 Codewords for the number of coding passes for each code-block

Number of coding passes	Codeword in packet header
1	0
2	10
3	1100
4	1101
5	1110
6 – 36	1 1110 0000 – 1 1111 1110
37 – 164	1111 1111 1000 0000 – 1111 1111 1111 1111

2.3.3.6.7 Length of Compressed Image Data from Each Code-Block (Annex B.10.7)

Since the recommended JPEG 2000 encoder does not use arithmetic coding bypass, each code-block contribution will contain a single codeword segment. The packet header encodes the length of the codeword segment using a binary number. The number of bits in this binary number is calculated using Equation 2.41 (Equation B.19):

$$bits = Lblock + \lfloor \log_2(\text{coding passes added}) \rfloor$$

Equation 2.41

Consider the example in Annex B.10.7.1. Suppose that in successive layers a code-block has 6 bytes, 31 bytes, 44 bytes, and 134 bytes respectively. Further assume that the number of coding passes is 1, 9, 2, and 5. Initially, the value of *Lblock* is 3. When processing the first layer, we need to encode the value of 6 (110), which requires at least 3 bits. Based on Equation 2.41, if we keep *Lblock* at its initial value of 3, we would use $3 + \log_2 1 = 3$ bits. This is sufficient, so we do not increment *Lblock*, and the code word is 0110, where the leading zero delimits the fact that *Lblock* is not incremented. When processing the second layer, we need to encode the value of 31 (11111), which requires at least 5 bits. Based on Equation 2.41, with *Lblock* of 3, we use $3 + \log_2 9 = 6$ bits (excluding the *Lblock* delimiter), so the code word is 0011111. When processing the third layer, we need to encode the value of 44 (101100), which requires at least 6 bits. If *Lblock* were to remain 3, we would have only $3 + \log_2 2 = 4$ bits, so we need to increment *Lblock* by 2. We do this by prepending two ones prior to the zero delimiter, and the code word is 110101100. Finally, when processing the fourth layer, we need to encode the value of 134 (10000110), which requires at least 8 bits. Since *Lblock* is now 5, we would have $5 + \log_2 5 = 7$ bits, so we need to further increment *Lblock* by 1. So, we prepend a single one prior to the zero delimiter, and the code word is 1010000110.

2.3.3.6.8 Order of Information Within Packet Header (Annex B.10.8)

The overall packet header construction is outlined below:

1) CODE one bit to indicate zero or non-zero packet
2) If non-zero
    For each subband (LL, or HL, LH and HH)
        For all code-blocks in the subband, confined to the relevant precinct, in raster-scan order
            CODE code-block inclusion bits (if not previously included, use tag tree, otherwise single bit)
            If code-block included
                If first-time inclusion
                    CODE zero bit-planes information
                CODE number of coding passes
                CODE length of coding pass data
3) Bit stuff to byte boundary with '0's.

Consider the example of packet header construction from Annex B.10.8 (Figure B-13 and Table B-5). We consider a single precinct with six code-blocks. Code-block 0,0 is first included in layer 0; it has 3 missing most-significant (zero) bit-planes; in layer 0 it contains 4 bytes in 3 coding passes, and in layer 1 it contains 10 bytes in 3 coding passes. Code-block 1,0 is first included in layer 0; it has 4 missing bit-planes; in layer 0 it contains 4 bytes in 2 coding passes, and in layer 1 it contains no data. Code-block 2,0 is first included in layer 2; it has 7 missing bit-planes. (For this example, we will only consider layers 0 and 1.) Code-block 0,1 is first included in layer 2; it has 3 missing bit-planes. Code-block 1,1 is first included in layer 1; it has 3 missing bit-planes; in layer 1 it contains 1 byte in 1 coding pass. Code-block 2,1 is first included in layer 1; it has 6 missing bit-planes; in layer 1 it contains 2 bytes in 1 coding pass. This information, and the resulting tag trees for the inclusion information and zero bit-planes, is detailed below.

Inclusion information

0	0	2
2	1	1

Zero bit-planes

3	4	7
3	3	6

# of coding passes (layer 0)

3	2	–
–	–	–

Length information (layer 0)

4	4	–
–	–	–

Inclusion tag tree

0	1

0

Zero bit-planes tag tree

3	6

3

# of coding passes (layer 1)

3	–	–
–	1	1

Length information (layer 1)

10	–	–
–	1	2

The resulting bit-stream is provided in Table B-5 and is repeated here for convenience. Each step in this table will be further described below.

Packet for layer 0

Step #	Bit stream (in order)	Derived meaning
1	1	Packet non-zero in length
2	111	Code-block 0,0 included for the first time (partial inclusion tag tree)
3	000111	Code-block 0,0 insignificant for 3 bit-planes
4	1100	Code-block 0,0 has 3 coding passes included
5	0	Code-block 0,0 length indicator is unchanged
6	0100	Code-block 0,0 has 4 bytes, 4 bits are used, 3 + floor($\log_2$ 3)
7	1	Code-block 1,0 included for the first time (partial inclusion tag tree)
8	01	Code-block 1,0 insignificant for 4 bit-planes
9	10	Code-block 1,0 has 2 coding passes included
10	10	Code-block 1,0 length indicator increased by 1 bit (3 to 4)
11	00100	Code-block 1,0 has 4 bytes, 5 bits are used, 4 + floor($\log_2$ 2) (Note that while this is a legitimate entry, it is not minimal in code length)
12	0	Code-block 2,0 not yet included (partial tag tree)
13	0	Code-block 0,1 not yet included
14	0	Code-block 1,1 not yet included
15		Code-block 2,1 no yet included (no data needed, already conveyed by partial tag tree for code-block 2,0)
16	…	Packet header data for the other subbands, packet data

Packet for layer 1

Step #	Bit stream (in order)	Derived meaning
17	1	Packet non-zero in length
18	1	Code-block 0,0 included again
19	1100	Code-block 0,0 has 3 coding passes included
20	0	Code-block 0,0 length indicator is unchanged
21	1010	Code-block 0,0 has 10 bytes, 4 bits are used, 3 + floor($\log_2$ 3)
22	0	Code-block 1,0 not included in this layer
23	10	Code-block 2,0 not yet included
24	0	Code-block 0,1 not yet included
25	1	Code-block 1,1 included for the first time
26	1	Code-block 1,1 insignificant for 3 bit-planes
27	0	Code-block 1,1 has 1 coding pass included
28	0	Code-block 1,1 length information is unchanged
29	001	Code-block 1,1 has 1 byte, 3 bits are used, 3 + floor($\log_2$ 1)
30	1	Code-block 2,1 included for the first time
31	00011	Code-block 2,1 insignificant for 6 bit-planes
32	0	Code-block 2,1 has 1 coding pass included
33	0	Code-block 2,1 length indicator is unchanged
34	010	Code-block 2,1 has 2 bytes, 3 bits are used, 3 + floor($\log_2$ 1)
35	…	Packet header data for the other subbands, packet data

Further explanation of each step in this coding example follows:

1) First, we create the packet for layer 0. The first bit in each packet is used to indicate that the packet is not zero length.
2) We process each code-block in the precinct in raster order, so the first code-block we consider is 0,0. This code-block is included for the first time in this layer, so we use the inclusion tag tree to signal this fact. Starting at the root of the tag tree, we visit all nodes on a direct line to the leaf for code-block 0,0. Since we have not yet

coded any partial information for the tag tree, we need to code the root node $q_0(0,0)$, the intermediate node $q_1(0,0)$, and the leaf node $q_2(0,0)$. Each of these three nodes is 0, so there is no need to increment the default states of any of these nodes, and each node is locked in by coding a 1. Thus, the code word is 111.

3) Since code-block 0,0 is included in this layer for the first time, we need to code the number of zero bit-planes that were skipped. For this, we use the zero bit-planes tag tree, which is a separate tag tree from the inclusion information tag tree, with its own state. In this tag tree, we have not coded any partial information yet, so we start at the root and code all nodes on the branch to the leaf for code-block 0,0. The root $q_0(0,0)$ is 3, but its initial state was 0, so we need to increment by 3; i.e. we code three 0s followed by a 1 to lock in the value. The intermediate node $q_1(0,0)$ is also 3, so there is no increment to its parent's value, and we code the 1 to lock in its value. The leaf node $q_2(0,0)$ is also 3, so again we simply code a 1. Thus, the code word is 000111.
4) Since code-block 0,0 is included in this layer, we must code the number of coding passes and the length of the compressed image data. The number of coding passes is coded using the look-up table in Table 2.10 (Table B-4). For 3 coding passes, the code-word is 1100.
5) The length of the compressed image data is 4 bytes, i.e. 100 in binary. This requires at least 3 bits. The current value of *Lblock* for code-block 0,0 is unchanged from its initial value of 3, and the number of bits used is calculated to be 4, using Equation 2.41 (Equation B.19). We first code a single 0 to indicate that the value of *Lblock* was unchanged.
6) Then, we code the data length as a binary number using 4 bits, and the codeword is 0100.
7) Now, we proceed in raster order to code-block 1,0. It is included for the first time in this layer, so we use the tag tree to code this information. Starting from the root, both $q_0(0,0)$ and $q_1(0,0)$ have been locked in and coded as 0, and thus are not coded at this point in time. The leaf $q_2(1,0)$ is also 0, so its state does not need to be incremented above that of its parent $q_1(0,0)$. So, we simply code a 1 to lock in the value.
8) Since code-block 1,0 is included in this layer for the first time, we need to code the number of zero bit-planes that were skipped, using the zero bit-planes tag tree. In this tag tree, we have already locked in and coded $q_0(0,0)$ and $q_1(0,0)$ with the value 3. The leaf $q_2(1,0)$ has a value of 4, so we need to increment by one (i.e. code a single 0) and lock in the value (i.e. code a 1). Thus, the code word is 01.
9) The number of coding passes is coded using the look-up table in Table 2.10 (Table B-4). For 2 coding passes, the code-word is 10.
10) The length of the compressed image data is 4 bytes, i.e. 100 in binary. This requires at least 3 bits. In a minimal code-length implementation, there would be no need to change the value of *Lblock* for code-block 1,0 at this point in time. In such a minimal code-length implementation, we would code a 0 to indicate that *Lblock* is unchanged followed by 0100 to code the value 4 using 4 bits. In this example, however, *Lblock* is prematurely incremented by 1 (from 3 to 4), so we code a single 1 for the increment followed by a 0; i.e. the codeword is 10. Note that although not optimal, this premature incrementing of *Lblock* is legitimate, and a decoder must be able to handle it.
11) Using the value of *Lblock* = 4, we calculate that we need to use 5 bits to represent the data length, so the codeword is 00100.
12) Next, we proceed to code-block 2,0. It has not yet been included, and will not be included until layer 2, so we must use the tag tree to indicate this fact. The rule for the inclusion tag tree is that only the information that is necessary at this point in time is to be coded. At this point in time, all we need to know is that code-block 2,0 is not included in layer 0. It does not matter whether it will be included in layer 1 or layer 2. Starting at the root node $q_0(0,0)$, we have already locked in the value of 0. The next node in the path to the leaf is the intermediate node $q_1(1,0)$. If we increment the current state of $q_1(1,0)$ from 0 to 1, we know that all its children must have a value of at least 1. This is all we need to know that code-block 2,0 is not included in layer 0, so this is all that we will code at this point in time. I.e., code a single 0 to increment $q_1(1,0)$. At this time, we do not code the 1 to lock in the value of $q_1(1,0)$ as 1.
13) Next, we proceed to code-block 0,1. It has not yet been included, so we use the tag tree to indicate this. We have already locked in its parent at $q_1(0,0)$ with the value 0, so we are now coding the leaf node $q_2(0,1)$. All we need to do at this point in time is increment its current state by 1 to indicate that it is not included until at least layer 1. While processing the current layer, we do not need to increment this node all the way to its final value of 2. Note that there is no option here – it would be illegal to code any more information than necessary for this layer, as the decoder would interpret the extra bits as part of the codeword for the next code-block. So, we code a single 0.
14) Code-block 1,1 is also not yet included, so we use the tag tree. Again, we have already locked in its parent at $q_1(0,0)$ with the value 0, so all we need to do at this time is increment $q_2(1,1)$ by 1 to indicate that it is not included until at least layer 1. So, the codeword is also a single 0.

15) Code-block 2,1 is also not yet included, so we use the tag tree. Its parent at $q_1(1,0)$ has already been incremented to a current value of 1, so we already know that $q_2(2,1)$ is at least 1 and therefore is not included in layer 0. This is all we need to know right now, so no information is coded for this code-block in this layer.
16) After completing this subband of this precinct, the other subbands of this precinct (if applicable) are processed in a similar manner using their own sets of tag trees. Once the packet header is complete for all the subbands of this precinct, the compressed image data for this layer's packet is appended to the codestream.
17) Next, we consider the packet for layer 1 of the same precinct. All tag tree states have remained from the processing performed on the packet header for layer 0. We start the new packet by coding a 1 to indicate that the packet is not zero-length.
18) Now, we again visit all code-blocks in the precinct in raster order. Starting with code-block 0,0, we see that there is additional data included in this layer. Since code-block 0,0 was already included, we do not use the tag tree, but simply code a 1 to indicate that it is included again in this layer.
19) Since this is not the first time that code-block 0,0 was included, we are not concerned with zero bit-planes, and we proceed directly to the number of coding passes. From the look-up table in Table 2.10 (Table B-4), the codeword for 3 coding passes is 1100.
20) The length of the compressed image data is 10 bytes, i.e. 1010 in binary. With the current value of *Lblock* for code-block 0,0 equal to 3, we calculate that we would use 4 bits. Since this is sufficient, there is no need to increment *Lblock*, and we code a 0 to indicate this fact.
21) Using 4 bits, we code the length as 1010.
22) Code-block 1,0 is not included in layer 1. Since it was previously included, we do not use the tag tree, but simply code a 0 to indicate that it is not included again in this layer.
23) Code-block 2,0 is still not yet included. Referring to the inclusion tag tree, we previously (in step 12) had incremented its parent at $q_1(1,0)$ to 1 but did not lock in its value. Knowing that $q_1(1,0)$ is at least 1 and therefore $q_2(2,0)$ is at least 1 is not sufficient to know if code-block 2,0 is included in layer 1, so we need to continue coding this branch of the tag tree. The next step is to lock in the current value of 1 for node $q_1(1,0)$ with a code of 1. Then, we can increment the current value for node $q_2(2,0)$ from 1 to 2 with a code of 0. Remember that at this time it is not necessary to lock in the value of node $q_2(2,0)$ – as long as we know that its value is at least 2, we know that this code-block is not included in layer 1. Thus, the codeword is 10.
24) Code-block 0,1 is also not yet included. Referring to the inclusion tag tree, we previously (in step 13) had incremented the current value of $q_2(0,1)$ to 1, and now we need to increment it once more to 2. Again, it is not yet necessary to lock in the value as 2 – just knowing that the value is at least 2 implies that this code-block is not included in layer 1. Thus, the codeword is simply 0.
25) Code-block 1,1 is included for the first time in layer 1. Referring to the inclusion tag tree, we previously (in step 14) had incremented $q_2(1,1)$ to 1, and now we need to lock it in as a final value using the codeword 1.
26) Since code-block 1,1 is included for the first time, we need to code the zero bit-planes using the associated tag tree. We have already coded and locked in a value of 3 for $q_1(0,0)$, so we just need to lock in this value of 3 for $q_2(1,1)$. Thus, the codeword is simply 1.
27) From the look-up table in Table 2.10 (Table B-4), the codeword for 1 coding pass is 0.
28) There is no need to increment *Lblock* for code-block 1,1, so we code this with a 0.
29) Based on *Lblock* of 3, we calculate to use 3 bits, so the codeword for a compressed image data length of 1 byte is 001.
30) Code-block 2,1 is included for the first time in layer 1. Referring to the inclusion tag tree, we previously (in step 23) locked in the parent node $g_1(1,0)$ with a value of 1. Now, we must lock in the leaf node at $q_2(2,1)$ with its current value of 1, so we code a 1.
31) Since code-block 2,1 is included for the first time, we need to code the zero bit-planes using the associated tag tree. Starting from the root and following the path to this code-block, we see that $q_0(0,0)$ was already locked in with the value of 3, but $q_1(1,0)$ has not been processed at all. We need to increment $q_1(1,0)$ from 3 to 6, which requires coding three 0s, and we lock this value in by coding a 1. Then, we proceed to $g_2(2,1)$, where we simply need to lock in the current value of 6 (same as its parent), by coding a 1. Thus, the codeword is 00011.
32) From the look-up table in Table 2.10 (Table B-4), the codeword for 1 coding pass is 0.
33) There is no need to increment *Lblock* for code-block 2,1, so we code this with a 0.
34) Based on *Lblock* of 3, we calculate to use 3 bits, so the codeword for a compressed image data length of 2 bytes is 010.
35) Finally, after completing this subband of this precinct, the other subbands of this precinct (if applicable) are processed for layer 1 in a similar manner using their own sets of tag trees. Once the packet header is complete

for all the subbands of this precinct, the compressed image data for this layer's packet is appended to the codestream.

2.3.3.7 Tile Codestream Formation (Annex B.11)

JPEG 2000 allows the tile codestream data to be split into one or more tile-parts. Each tile part has associated with it a tile-part header, which is a group of markers and marker segments at the beginning of the tile-part in the codestream that describes the tile-part coding parameters. The tile-part header is followed by the packets contained in the tile-part. In the recommended implementation, only one tile-part is used for each tile. Thus, the tile codestream contains one tile-part header, followed by all of the tile data packets ordered in LRCP progression (see Section 2.3.3.8). The details regarding the structure of the tile-part headers, including the ordering of the marker segments and system-specific parameters, are discussed in other documentation.

2.3.3.8 Progression Order (Annex B.12)

Once all packets for a tile have been formed, they must be organized within the codestream. JPEG 2000 allows five ordering schemes that can be selected to control the sequence in which packets occur in a tile codestream. A single ordering scheme can be used to order all the packets, or the ordering scheme can be changed as often as desired within the codestream. For this implementation of JPEG 2000, the tile codestream is formed using one progression ordering scheme, which is signaled in the COD marker of the main header (see Section 2.3.3.9). The progression order is layer-resolution-component-position (LRCP), and is described by a series of nested for loops:

```
for each layer = 1,2,3,...
        for each resolution = 1,2,3,...
                for each component = 1,2,3,...
                        for each position (precinct) = 1,2,3,...
                                Include corresponding packet
```

Note that this simplifies for images with only one component and when maximal precincts are used (such that each subband has only one precinct). In this case there is only one packet per layer-resolution, and, for a specific tile, the number of packets per layer is equal to the number of resolutions. This knowledge can be used to quickly locate a specific packet within the tile codestream.

Given an LRCP ordering scheme, the packets corresponding to any particular resolution are distributed throughout the tile codestream, and thus to obtain a lower resolution version of the tile, it is necessary to jump around within the codestream to grab the desired packets and skip the unwanted packets. This parsing is made easier with the *a priori* knowledge that each layer-resolution has only one packet.

2.3.3.9 Image Codestream Formation (Annex A)

The image codestream is formed by merging the tile-part codestreams and including a main header and end of codestream (EOC) symbol. The tile codestreams (each containing one tile-part) are included in raster-scan order. The main header is a group of markers and marker segments at the beginning of the codestream that describes the image parameters and coding parameters that apply (unless overridden by later markers) to every tile and tile-component. The details regarding the structure of the main header, including the ordering of the marker segments and system-specific parameters, are discussed in other documentation.

[1] It should be noted that the requirement for visually lossless quality is a more stringent requirement than simply "being as good as the 4.3 DPCM". Thus, the overall achieved bit rates for imagery are likely to be higher than the estimates from this study.