Image Compression Standards

9

Recent years have seen an explosion in the availability of digital images, because of the increase in numbers of digital imaging devices such as smartphones, webcams, digital cameras, and scanners. The need to efficiently process and store images in digital form has motivated the development of many image compression *standards* for various applications and needs. In general, standards have greater longevity than particular programs or devices and therefore warrant careful study. In this chapter, we examine some current standards and demonstrate how topics presented in Chaps. 7 and 8 are applied in practice.

We first explore the standard JPEG definition, used in most images, then go on to look at the wavelet-based JPEG2000 standard. Two other standards, JPEG-LS—aimed particularly at a lossless JPEG, outside the main JPEG standard—and JBIG, for bi-level image compression, are included for completeness.

9.1 The JPEG Standard

JPEG is an image compression standard developed by the *Joint Photographic Experts Group*. It was formally accepted as an international standard in 1992 [1].

JPEG consists of a number of steps, each of which contributes to compression. We will look at the motivation behind these steps, then take apart the algorithm piece by piece.

9.1.1 Main Steps in JPEG Image Compression

As we know, unlike one-dimensional audio signals, a digital image $f(i, j)$ is not defined over the time domain. Instead, it is defined over a *spatial domain*—that is, an image is a function of the two dimensions i and j (or, conventionally, x and y). The 2D DCT is used as one step in JPEG, to yield a frequency response that is a function $F(u, v)$ in the *spatial frequency domain*, indexed by two integers u and v.

Z.-N. Li et al., *Fundamentals of Multimedia*,
Texts in Computer Science, DOI: 10.1007/978-3-319-05290-8_9,

JPEG is a lossy image compression method. The effectiveness of the DCT transform coding method in JPEG relies on three major observations:

Observation 1. Useful image contents change relatively slowly across the image—that is, it is unusual for intensity values to vary widely several times in a small area—for example, in an 8×8 image block. Spatial frequency indicates how many times pixel values change across an image block. The DCT formalizes this notion with a measure of how much the image contents change in relation to the number of cycles of a cosine wave per block.

Observation 2. Psychophysical experiments suggest that humans are much less likely to notice the loss of very high-spatial frequency components than lower frequency components.

JPEG's approach to the use of DCT is basically to reduce high-frequency contents and then efficiently code the result into a bitstring. The term *spatial redundancy* indicates that much of the information in an image is repeated: if a pixel is red, then its neighbor is likely red also. Because of Observation 2 above, the DCT coefficients for the lowest frequencies are most important. Therefore, as frequency gets higher, it becomes less important to represent the DCT coefficient accurately. It may even be safely set to zero without losing much perceivable image information.

Clearly, a string of zeros can be represented efficiently as the length of such a run of zeros, and compression of bits required is possible. Since we end up using fewer numbers to represent the pixels in blocks, by removing some location-dependent information, we have effectively removed spatial redundancy.

JPEG works for both color and grayscale images. In the case of color images, such as YCbCr, the encoder works on each component separately, using the same routines. If the source image is in a different color format, the encoder performs a color-space conversion to YCbCr. As discussed in Chap. 5, the chrominance images Cr and Cb are subsampled: JPEG uses the 4:2:0 scheme, making use of another observation about vision:

Observation 3. Visual acuity (accuracy in distinguishing closely spaced lines) is much greater for gray ("black and white") than for color. We simply cannot see much change in color if it occurs in close proximity—think of the blobby ink used in comic books. This works simply because our eye sees the black lines best, and our brain just pushes the color into place. In fact, ordinary broadcast TV makes use of this phenomenon to transmit much less color information than gray information.

When the JPEG image is needed for viewing, the three compressed component images can be decoded independently and eventually combined. For the color channels, each pixel must be first enlarged to cover a 2×2 block. Without loss of generality, we will simply use one of them—for example, the Y image, in the description of the compression algorithm below.

Figure 9.1 shows a block diagram for a JPEG encoder. If we reverse the arrows in the figure, we basically obtain a JPEG decoder. The JPEG encoder consists of the following main steps:

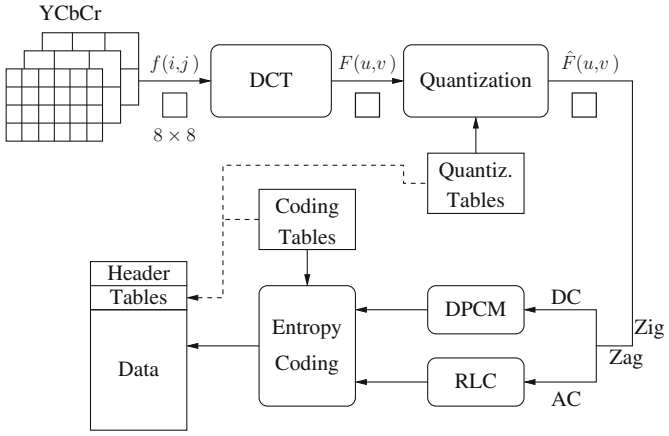


Fig. 9.1 Block diagram for JPEG encoder

- Transform RGB to YCbCr and subsample color
- Perform DCT on image blocks
- Apply Quantization
- Perform Zigzag ordering and run-length encoding
- Perform Entropy coding.

DCT on Image Blocks

Each image is divided into 8×8 blocks. The 2D DCT (Eq. 8.17) is applied to each block image $f(i, j)$, with output being the DCT coefficients $F(u, v)$ for each block. The choice of a small block size in JPEG is a compromise reached by the committee: a number larger than 8 would have made accuracy at low frequencies better, but using 8 makes the DCT (and IDCT) computation very fast.

Using blocks at all, however, has the effect of isolating each block from its neighboring context. This is why JPEG images look choppy ("blocky") when the user specifies a high *compression ratio*—we can see these blocks. (And in fact removing such "blocking artifacts" is an important concern of researchers.)

To calculate a particular $F(u, v)$, we select the basis image in Fig. 8.9 that corresponds to the appropriate u and v and use it in Eq. 8.17 to derive one of the frequency responses $F(u, v)$.

Quantization

The quantization step in JPEG is aimed at reducing the total number of bits needed for a compressed image [2]. It consists of simply dividing each entry in the frequency

Table 9.1 The luminance quantization table

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Table 9.2 The chrominance quantization table

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

space block by an integer, then rounding:

$$\hat{F}(u, v) = \text{round}\left(\frac{F(u, v)}{Q(u, v)}\right) \tag{9.1}$$

Here, $F(u, v)$ represents a DCT coefficient, $Q(u, v)$ is a *quantization matrix* entry, and $\hat{F}(u, v)$ represents the *quantized DCT coefficients* JPEG will use in the succeeding entropy coding.

The default values in the 8×8 quantization matrix $Q(u, v)$ are listed in Tables 9.1 and 9.2 for luminance and chrominance images, respectively. These numbers resulted from psychophysical studies, with the goal of maximizing the compression ratio while minimizing perceptual losses in JPEG images. The following should be apparent:

- Since the numbers in $Q(u, v)$ are relatively large, the magnitude and variance of $\hat{F}(u, v)$ are significantly smaller than those of $F(u, v)$. We'll see later that $\hat{F}(u, v)$ can be coded with many fewer bits. *The quantization step is the main source for loss in JPEG compression.*
- The entries of $Q(u, v)$ tend to have larger values toward the lower right corner. This aims to introduce more loss at the higher spatial frequencies—a practice supported by Observations 1 and 2.

We can handily change the compression ratio simply by multiplicatively *scaling* the numbers in the $Q(u, v)$ matrix. In fact, the *quality factor* (qf), a user choice offered in every JPEG implementation, can be specified. It is usually in the range of 1..100, where $qf = 100$ corresponds to the highest quality compressed images and $qf = 1$ the lowest quality. The relationship between qf and the $scaling\_factor$ is as below:

```
// qf is the user-selected compression quality
// Q is the default Quantization Matrix
```

```
// Qx is the scaled Quantization Matrix
// Q1 is a Quantization Matrix which is all 1's

if qf >= 50
    scaling_factor = (100-qf)/50;
else
    scaling_factor = (50/qf);
end
if scaling_factor != 0    // if qf is not 100
    Qx = round( Q*scaling_factor );
else
    Qx = Q1;              // no quantization
end
Qx = uint8(Qx);           // max is clamped to 255 for qf=1
```

As an example, when $qf = 50$, $scaling\_factor$ will be 1. The resulting Q values will be equal to the table entries. When $qf = 10$, the $scaling\_factor$ will be 5. The resulting Q values will be five times the table entry values. For $qf = 100$, the table entries simply become all 1 values, meaning no quantization from this source. Very low quality factors, like $qf = 1$, are a special case: if indeed $qf = 1$ then the $scaling\_factor$ will be 50, and the quantization matrix will contain very large numbers. However, because of the type-cast to `uint8` in a typical implementation, the table entries go their effective maximum value of 255. Realistically, almost for all applications, qf should not be less than 10.

JPEG also allows custom quantization tables to be specified and put in the header; it is interesting to use low-constant or high-constant values such as $Q \equiv 2$ or $Q \equiv 128$ to observe the basic effects of Q on visual artifacts.

Figures 9.2 and 9.3 shows some results of JPEG image coding and decoding on the test image *Lena*. Only the luminance image (Y) is shown. Also, the lossless coding steps after quantization are not shown, since they do not affect the quality/loss of the JPEG images. These results show the effect of compression and decompression applied to a relatively smooth block in the image and a more textured (higher-frequency-content) block, respectively.

Suppose $f(i, j)$ represents one of the 8×8 blocks extracted from the image, $F(u, v)$ the DCT coefficients, and $\hat{F}(u, v)$ the quantized DCT coefficients. Let $\tilde{F}(u, v)$ denote the dequantized DCT coefficients, determined by simply multiplying by $Q(u, v)$, and let $\tilde{f}(i, j)$ be the reconstructed image block. To illustrate the quality of the JPEG compression, especially the loss, the error $\epsilon(i, j) = f(i, j) - \tilde{f}(i, j)$ is shown in the last row in Figs. 9.2 and 9.3.

In Fig. 9.2, an image block (indicated by a black box in the image) is chosen at the area where the luminance values change smoothly. Actually, the left side of the block is brighter, and the right side is slightly darker. As expected, except for the DC and the first few AC components, representing low spatial frequencies, most of the DCT coefficients $F(u, v)$ have small magnitudes. This is because the pixel values in this block contain few high-spatial-frequency changes.

An explanation of a small implementation detail is in order. The range of 8-bit luminance values $f(i, j)$ is $[0, 255]$. In the JPEG implementation, each Y value is

An 8×8 block from the Y image of 'Lena'

200	202	189	188	189	175	175	175
200	203	198	188	189	182	178	175
203	200	200	195	200	187	185	175
200	200	200	200	197	187	187	187
200	205	200	200	195	188	187	175
200	200	200	200	200	190	187	175
205	200	199	200	191	187	187	175
210	200	200	200	188	185	187	186

$f(i,j)$

515	65	-12	4	1	2	-8	5
-16	3	2	0	0	-11	-2	3
-12	6	11	-1	3	0	1	-2
-8	3	-4	2	-2	-3	-5	-2
0	-2	7	-5	4	0	-1	-4
0	-3	-1	0	4	1	-1	0
3	-2	-3	3	3	-1	-1	3
-2	5	-2	4	-2	2	-3	0

$F(u,v)$

32	6	-1	0	0	0	0	0
-1	0	0	0	0	0	0	0
-1	0	1	0	0	0	0	0
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$\hat{F}(u,v)$

512	66	-10	0	0	0	0	0
-12	0	0	0	0	0	0	0
-14	0	16	0	0	0	0	0
-14	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$\tilde{F}(u,v)$

199	196	191	186	182	178	177	176
201	199	196	192	188	183	180	178
203	203	202	200	195	189	183	180
202	203	204	203	198	191	183	179
200	201	202	201	196	189	182	177
200	200	199	197	192	186	181	177
204	202	199	195	190	186	183	181
207	204	200	194	190	187	185	184

$\tilde{f}(i,j)$

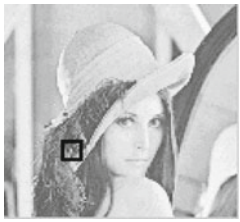
1	6	-2	2	7	-3	-2	-1
-1	4	2	-4	1	-1	-2	-3
0	-3	-2	-5	5	-2	2	-5
-2	-3	-4	-3	-1	-4	4	8
0	4	-2	-1	-1	-1	5	-2
0	0	1	3	8	4	6	-2
1	-2	0	5	1	1	4	-6
3	-4	0	6	-2	-2	2	2

$\epsilon(i,j) = f(i,j) - \tilde{f}(i,j)$

Fig. 9.2 JPEG compression for a smooth image block

first reduced by 128 by simply subtracting 128 before encoding. The idea here is to turn the Y component into a zero-mean image, the same as the chrominance images. As a result, we do not waste any bits coding the mean value. (Think of an 8×8 block with intensity values ranging from 120 to 135.) Using $f(i, j)-128$ in place of $f(i, j)$ will not affect the output of the AC coefficients—it alters only the DC coefficient.After decoding, the subtracted 128 will be added back onto the Y values.

In Fig. 9.3, the image block chosen has rapidly changing luminance. Hence, many more AC components have large magnitudes (including those toward the lower right



Another 8×8 block from the Y image of 'Lena'

70	70	100	70	87	87	150	187
85	100	96	79	87	154	87	113
100	85	116	79	70	87	86	196
136	69	87	200	79	71	117	96
161	70	87	200	103	71	96	113
161	123	147	133	113	113	85	161
146	147	175	100	103	103	163	187
156	146	189	70	113	161	163	197

$f(i,j)$

-80	-40	89	-73	44	32	53	-3
-135	-59	-26	6	14	-3	-13	-28
47	-76	66	-3	-108	-78	33	59
-2	10	-18	0	33	11	-21	1
-1	-9	-22	8	32	65	-36	-1
5	-20	28	-46	3	24	-30	24
6	-20	37	-28	12	-35	33	17
-5	-23	33	-30	17	-5	-4	20

$F(u,v)$

-5	-4	9	-5	2	1	1	0
-11	-5	-2	0	1	0	0	-1
3	-6	4	0	-3	-1	0	1
0	1	-1	0	1	0	0	0
0	0	-1	0	0	1	0	0
0	-1	1	-1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$\hat{F}(u,v)$

-80	-44	90	-80	48	40	51	0
-132	-60	-28	0	26	0	0	-55
42	-78	64	0	-120	-57	0	56
0	17	-22	0	51	0	0	0
0	0	-37	0	0	109	0	0
0	-35	55	-64	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$\tilde{F}(u,v)$

70	60	106	94	62	103	146	176
85	101	85	75	102	127	93	144
98	99	92	102	74	98	89	167
132	53	111	180	55	70	106	145
173	57	114	207	111	89	84	90
164	123	131	135	133	92	85	162
141	159	169	73	106	101	149	224
150	141	195	79	107	147	210	153

$\tilde{f}(i,j)$

0	10	-6	-24	25	-16	4	11
0	-1	11	4	-15	27	-6	-31
2	-14	24	-23	-4	-11	-3	29
4	16	-24	20	24	1	11	-49
-12	13	-27	-7	-8	-18	12	23
-3	0	16	-2	-20	21	0	-1
5	-12	6	27	-3	2	14	-37
6	5	-6	-9	6	14	-47	44

$\epsilon(i,j) = f(i,j) - \tilde{f}(i,j)$

Fig. 9.3 JPEG compression for a textured image block

corner, where u and v are large). Notice that the error $\epsilon(i, j)$ is also larger now than in Fig. 9.2—JPEG does introduce more loss if the image has quickly changing details.

An easy-to-use JPEG demo written in Java, linked from this section of the text website, is available for you to try.

Preparation for Entropy Coding

We have so far seen two of the main steps in JPEG compression: DCT and quantization. The remaining small steps shown in the block diagram in Fig. 9.1 all lead

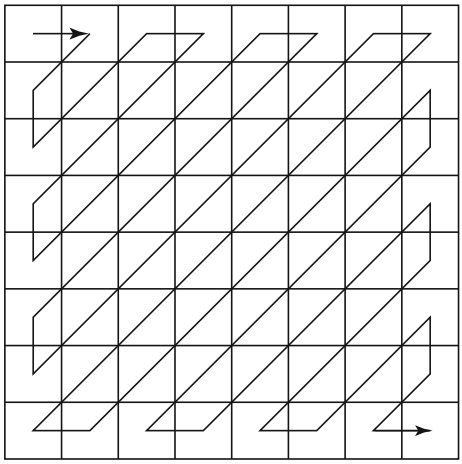


Fig. 9.4 Zigzag scan in JPEG

up to *entropy coding* of the quantized DCT coefficients. These additional data compression steps are lossless. Interestingly, the DC and AC coefficients are treated quite differently before entropy coding: run-length encoding on ACs versus DPCM on DCs.

Run-Length Coding on AC Coefficients

Notice in Fig. 9.2 the many zeros in $\hat{F}(u, v)$ after quantization is applied. *Run-length Coding* (*RLC*) (or *Run-length Encoding*, *RLE*) is therefore useful in turning the $\hat{F}(u, v)$ values into sets {*#-zeros-to-skip, next nonzero value*}. RLC is even more effective when we use an addressing scheme, making it most likely to hit a long run of zeros: a *zigzag scan* turns the 8×8 matrix $\hat{F}(u, v)$ into a *64-vector*, as Fig. 9.4 illustrates. After all, most image blocks tend to have small high-spatial-frequency components, which are zeroed out by quantization. Hence the zigzag scan order has a good chance of concatenating long runs of zeros. For example, $\hat{F}(u, v)$ in Fig. 9.2 will be turned into

$$(32, 6, -1, -1, 0, -1, 0, 0, 0, -1, 0, 0, 1, 0, 0, \ldots, 0)$$

with three runs of zeros in the middle and a run of 51 zeros at the end.

The RLC step replaces values by a pair `(RUNLENGTH, VALUE)` for each run of zeros in the AC coefficients of $\hat{F}$, where `RUNLENGTH` is the number of zeros in the run and `VALUE` is the next nonzero coefficient. To further save bits, a special pair (0,0) indicates the end-of-block after the last nonzero AC coefficient is reached. In the above example, not considering the first (DC) component, we will thus have

$$(0, 6)(0, -1)(0, -1)(1, -1)(3, -1)(2, 1)(0, 0)$$

Differential Pulse Code Modulation on DC Coefficients

The DC coefficients are coded separately from the AC ones. Each 8×8 image block has only one DC coefficient. The values of the DC coefficients for various blocks could be large and different, because the DC value reflects the average intensity of each block, but consistent with Observation 1 above, the DC coefficient is unlikely to change drastically within a short distance. This makes DPCM an ideal scheme for coding the DC coefficients.

If the DC coefficients for the first five image blocks are 150, 155, 149, 152, 144, DPCM would produce 150, 5, −6, 3, −8, assuming the predictor for the ith block is simply $d_i = DC_i - DC_{i-1}$, and $d_0 = DC_0$. We expect DPCM codes to generally have smaller magnitude and variance, which is beneficial for the next *entropy coding* step.

It is worth noting that unlike the run-length coding of the AC coefficients, which is performed on each individual block, DPCM for the DC coefficients in JPEG is carried out on the entire image at once.

Entropy Coding

The DC and AC coefficients finally undergo an entropy coding step. Below, we will discuss only the basic (or *baseline*[1]) entropy coding method, which uses Huffman coding and supports only 8-bit pixels in the original images (or color image components).

Let us examine the two entropy coding schemes, using a variant of Huffman coding for DCs and a slightly different scheme for ACs.

Huffman Coding of DC Coefficients

Each DPCM-coded DC coefficient is represented by a pair of symbols `(SIZE, AMPLITUDE)`, where `SIZE` indicates how many bits are needed for representing the coefficient and `AMPLITUDE` contains the actual bits.

Table 9.3 illustrates the size category for the different possible amplitudes. Notice that DPCM values could require more than 8 bits and could be negative values. The one's-complement scheme is used for negative numbers—that is, binary code 10 for 2, 01 for −2; 11 for 3, 00 for −3; and so on. In the example we are using, codes 150, 5, −6, 3, −8 will be turned into

$$(8, 10010110), (3, 101), (3, 001), (2, 11), (4, 0111)$$

In the JPEG implementation, `SIZE` is Huffman coded and is hence a variable-length code. In other words, `SIZE 2` might be represented as a single bit (0 or 1) if

[1] The JPEG standard allows both Huffman coding and Arithmetic coding; both are entropy coding methods. It also supports both 8-bit and 12-bit pixel sizes.

Table 9.3 Baseline entropy coding details—size category

Size	Amplitude
1	−1, 1
2	3, −2, 2, 3
3	−7 .. −4, 4 .. 7
4	−15 .. −8, 8 .. 15
.	.
.	.
.	.
10	−1023 .. −512, 512 .. 1023

it appeared most frequently. In general, smaller `SIZE`s occur much more often—the entropy of `SIZE` is low. Hence, deployment of Huffman coding brings additional compression. After encoding, a custom Huffman table can be stored in the JPEG image header; otherwise, a default Huffman table is used.

On the other hand, `AMPLITUDE` is not Huffman coded. Since its value can change widely, Huffman coding has no appreciable benefit.

Huffman Coding of AC Coefficients

Recall we said that the AC coefficients are run-length coded and are represented by pairs of numbers `(RUNLENGTH, VALUE)`. However, in an actual JPEG implementation, `VALUE` is further represented by `SIZE` and `AMPLITUDE`, as for the DCs. To save bits, `RUNLENGTH` and `SIZE` are allocated only 4 bits each and squeezed into a single byte—let us call this *Symbol 1*. *Symbol 2* is the `AMPLITUDE` value; its number of bits is indicated by `SIZE`:

Symbol 1: `(RUNLENGTH, SIZE)`
Symbol 2: `(AMPLITUDE)`

The 4-bit `RUNLENGTH` can represent only zero-runs of length 0 to 15. Occasionally, the zero run-length exceeds 15; then a special extension code, (15, 0), is used for Symbol 1. In the worst case, three consecutive (15, 0) extensions are needed before a normal terminating Symbol 1, whose `RUNLENGTH` will then complete the actual runlength. As in DC, Symbol 1 is Huffman coded, whereas Symbol 2 is not.

9.1.2 JPEG Modes

The JPEG standard supports numerous modes (variations). Some of the commonly used ones are:

- Sequential Mode
- Progressive Mode
- Hierarchical Mode
- Lossless Mode.

Sequential Mode

This is the default JPEG mode. Each graylevel image or color image component is encoded in a single left-to-right, top-to-bottom scan. We implicitly assumed this mode in the discussions so far. The "Motion JPEG" video codec uses Baseline Sequential JPEG, applied to each image frame in the video.

Progressive Mode

Progressive JPEG delivers low quality versions of the image quickly, followed by higher quality passes, and has become widely supported in web browsers. Such multiple scans of images are of course most useful when the speed of the communication line is low. In Progressive Mode, the first few scans carry only a few bits and deliver a rough picture of what is to follow. After each additional scan, more data is received, and image quality is gradually enhanced. The advantage is that the user end has a choice whether to continue receiving image data after the first scan(s).

Progressive JPEG can be realized in one of the following two ways. The main steps (DCT, quantization, etc.) are identical to those in Sequential Mode.

Spectral selection: This scheme takes advantage of the *spectral* (spatial frequency spectrum) characteristics of the DCT coefficients: the higher AC components provide only detail information.

Scan 1: Encode DC and first few AC components, e.g., AC1,AC2.
Scan 2: Encode a few more AC components, e.g., AC3, AC4, AC5.
⋮
Scan k: Encode the last few ACs, e.g., AC61, AC62, AC63.

Successive approximation: Instead of gradually encoding spectral bands, all DCT coefficients are encoded simultaneously, but with their most significant bits (MSBs) first.

Scan 1: Encode the first few MSBs, e.g., Bits 7, 6, 5, and 4.
Scan 2: Encode a few more less-significant bits, e.g., Bit 3.
⋮
Scan m: Encode the least significant bit (LSB), Bit 0.

Hierarchical Mode

As its name suggests, Hierarchical JPEG encodes the image in a hierarchy of several different resolutions. The encoded image at the lowest resolution is basically a compressed low-pass-filtered image, whereas the images at successively higher resolutions provide additional details (differences from the lower-resolution images). Similar to Progressive JPEG, Hierarchical JPEG images can be transmitted in multiple passes with progressively improving quality.

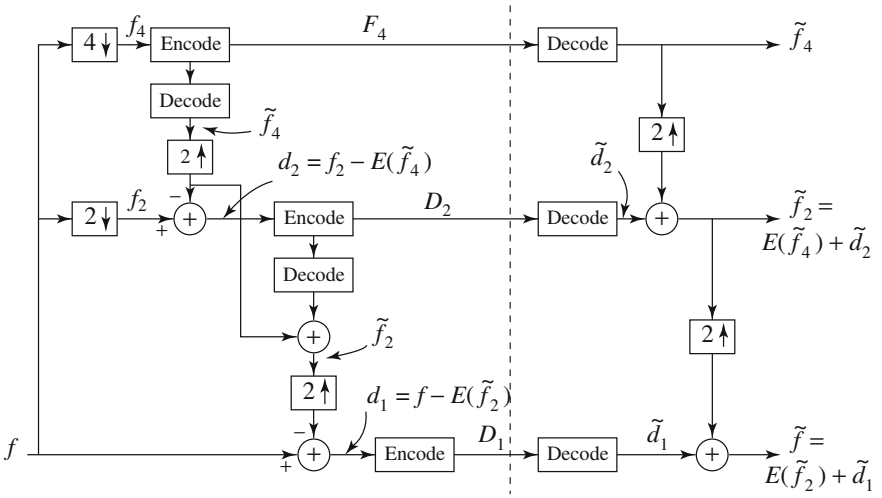


Fig. 9.5 Block diagram for Hierarchical JPEG

Figure 9.5 illustrates a three-level hierarchical JPEG encoder and decoder (separated by the dashed line in the figure).

Algorithm 9.1 (Three-Level Hierarchical JPEG Encoder).

1. **Reduction of image resolution**. Reduce resolution of the input image f (e.g., 512×512) by a factor of 2 in each dimension to obtain f_2 (e.g., 256×256). Repeat this to obtain f_4 (e.g., 128×128).
2. **Compress low-resolution image f_4**. Encode f_4 using any other JPEG method (e.g., Sequential, Progressive) to obtain F_4.
3. **Compress difference image d_2**. (a) Decode F_4 to obtain $\tilde{f}_4$. Use any interpolation method to expand $\tilde{f}_4$ to be of the same resolution as f_2 and call it $E(\tilde{f}_4)$. (b) Encode difference $d_2 = f_2 - E(\tilde{f}_4)$ using any other JPEG method (e.g., Sequential, Progressive) to generate D_2.
4. **Compress difference image d_1**. (a) Decode D_2 to obtain $\tilde{d}_2$; add it to $E(\tilde{f}_4)$ to get $\tilde{f}_2 = E(\tilde{f}_4) + \tilde{d}_2$, which is a version of f_2 after compression and decompression. (b) Encode difference $d_1 = f - E(\tilde{f}_2)$ using any other JPEG method (e.g., Sequential, Progressive) to generate D_1.

Algorithm 9.2 (Three-Level Hierarchical JPEG Decoder).

1. **Decompress the encoded low-resolution image F_4**. Decode F_4 using the same JPEG method as in the encoder, to obtain $\tilde{f}_4$.
2. **Restore image $\tilde{f}_2$ at the intermediate resolution**. Use $E(\tilde{f}_4) + \tilde{d}_2$ to obtain $\tilde{f}_2$.
3. **Restore image $\tilde{f}$ at the original resolution**. Use $E(\tilde{f}_2) + \tilde{d}_1$ to obtain $\tilde{f}$.

It should be pointed out that at step 3 in the encoder, the difference d_2 is not taken as $f_2 - E(f_4)$ but as $f_2 - E(\tilde{f}_4)$. Employing $\tilde{f}_4$ has its overhead, since an additional decoding step must be introduced on the encoder side, as shown in the figure.

So, is it necessary? It is, because the *decoder* never has a chance to see the original f_4. The restoration step in the decoder uses $\tilde{f}_4$ to obtain $\tilde{f}_2 = E(\tilde{f}_4) + \tilde{d}_2$. Since $\tilde{f}_4 \neq f_4$ when a lossy JPEG method is used in compressing f_4, the encoder must use $\tilde{f}_4$ in $d_2 = f_2 - E(\tilde{f}_4)$ to avoid unnecessary error at decoding time. This kind of decoder-encoder step is typical in many compression schemes. In fact, we have seen it in Sect. 6.3.5. It is present simply because the decoder has access only to encoded, not original, values.

Similarly, at step 4 in the encoder, d_1 uses the difference between f and $E(\tilde{f}_2)$, not $E(f_2)$.

Lossless Mode

Lossless JPEG is a very special case of JPEG which indeed has no loss in its image quality. As discussed in Chap. 7, however, it employs only a simple differential coding method, involving no transform coding. It is rarely used, since its compression ratio is very low compared to other, lossy modes. On the other hand, it meets a special need, and the subsequently developed JPEG-LS standard is specifically aimed at lossless image compression (see Sect. 9.3).

9.1.3 A Glance at the JPEG Bitstream

Figure 9.6 provides a hierarchical view of the organization of the bitstream for JPEG images. Here, a *frame* is a picture, a *scan* is a pass through the pixels (e.g., the red component), a *segment* is a group of blocks, and a *block* consists of 8×8 pixels. Examples of some header information are:

- **Frame header**
  - Bits per pixel
  - (Width, height) of image
  - Number of components
  - Unique ID (for each component)
  - Horizontal/vertical sampling factors (for each component)
  - Quantization table to use (for each component).
- **Scan header**
  - Number of components in scan
  - Component ID (for each component)
  - Huffman/Arithmetic coding table (for each component).

9.2 The JPEG2000 Standard

The JPEG standard is no doubt the most successful and popular image format to date. The main reason for its success is the quality of its output for relatively good compression ratio. However, in anticipating the needs and requirements of next-generation

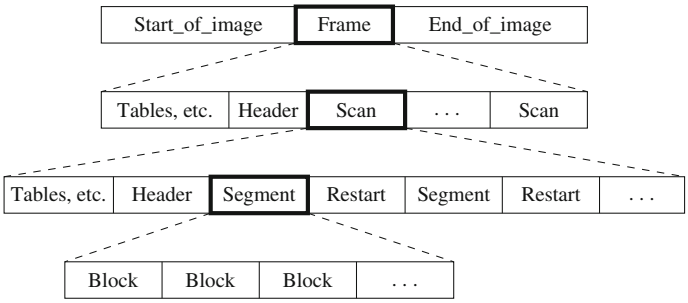


Fig. 9.6 JPEG bitstream

imagery applications, the JPEG committee also defined a new standard: JPEG2000. The main part, the core coding system, is specified in ISO/IEC 15444-1 [3].

The new JPEG2000 standard [4–6] aims to provide not only a better rate-distortion tradeoff and improved subjective image quality but also additional functionalities the current JPEG standard lacks. In particular, the JPEG2000 standard addresses the following problems:

- **Low bitrate compression.** The current JPEG standard offers excellent rate-distortion performance at medium and high bitrates. However, at bitrates below 0.25 bpp, subjective distortion becomes unacceptable. This is important if we hope to receive images on our web-enabled ubiquitous devices, such as web-aware wristwatches, and so on.
- **Lossless and lossy compression.** Currently, no standard can provide superior lossless compression and lossy compression in a single bitstream.
- **Large images.** The new standard will allow image resolutions greater than 64 k × 64 k without tiling. It can handle image sizes up to $2^{32} - 1$.
- **Single decompression architecture.** The current JPEG standard has 44 modes, many of which are application-specific and not used by the majority of JPEG decoders.
- **Transmission in noisy environments.** The new standard will provide improved error resilience for transmission in noisy environments such as wireless networks and the Internet.
- **Progressive transmission.** The new standard provides seamless quality and resolution scalability from low to high bitrates. The target bitrate and reconstruction resolution need not be known at the time of compression.
- **Region-of-interest coding.** The new standard permits specifying *Regions of Interest* (*ROI*), which can be coded with better quality than the rest of the image. We might, for example, like to code the face of someone making a presentation with more quality than the surrounding furniture.
- **Computer-generated imagery.** The current JPEG standard is optimized for natural imagery and does not perform well on computer-generated imagery.

- **Compound documents.** The new standard offers metadata mechanisms for incorporating additional nonimage data as part of the file. This might be useful for including text along with imagery, as one important example.

In addition, JPEG2000 is able to handle up to 256 channels of information, whereas the current JPEG standard is able to handle only three color channels. Such huge quantities of data are routinely produced in satellite imagery.

Consequently, JPEG2000 is designed to address a variety of applications, such as the Internet, color facsimile, printing, scanning, digital photography, remote sensing, mobile applications, medical imagery, digital library, e-commerce, and so on. The method looks ahead and provides the power to carry out remote browsing of large compressed images.

The JPEG2000 standard operates in two coding modes: DCT-based and wavelet-based. The DCT-based coding mode is offered for backward compatibility with the current JPEG standard and implements baseline JPEG. All the new functionalities and improved performance reside in the wavelet-based mode.

9.2.1 Main Steps of JPEG2000 Image Compression*

The main compression method used in JPEG2000 is the *(Embedded Block Coding with Optimized Truncation) algorithm* (*EBCOT*), designed by Taubman [7]. In addition to providing excellent compression efficiency, EBCOT produces a bitstream with a number of desirable features, including quality and resolution scalability and *random access*.

The basic idea of EBCOT is the partition of each sub-band LL, LH, HL, HH produced by the wavelet transform into small blocks called *code blocks*. Each code block is coded independently, in such a way that no information for any other block is used.

A separate, scalable bitstream is generated for each code block. With its block-based coding scheme, the EBCOT algorithm has improved error resilience. The EBCOT algorithm consists of three steps:

1. Block coding and bitstream generation
2. Postcompression rate-distortion (PCRD) optimization
3. Layer formation and representation.

Block Coding and Bitstream Generation

Each sub-band generated for the 2D discrete wavelet transform is first partitioned into small code blocks, typically 64×64, or other size no less than 32×32. Then the EBCOT algorithm generates a highly scalable bitstream for each code block B_i. The bitstream associated with B_i may be independently truncated to any member of a predetermined collection of different lengths R_i^n, with associated distortion D_i^n.

For each code block B_i (see Fig. 9.7), let $s_i[\mathbf{k}] = s_i[k_1, k_2]$ be the 2D sequence of small code blocks of sub-band samples, with k_1 and k_2 the row and column index. (With this definition, the horizontal high-pass sub-band HL must be transposed so that k_1 and k_2 will have meaning consistent with the other sub-bands. This transposition

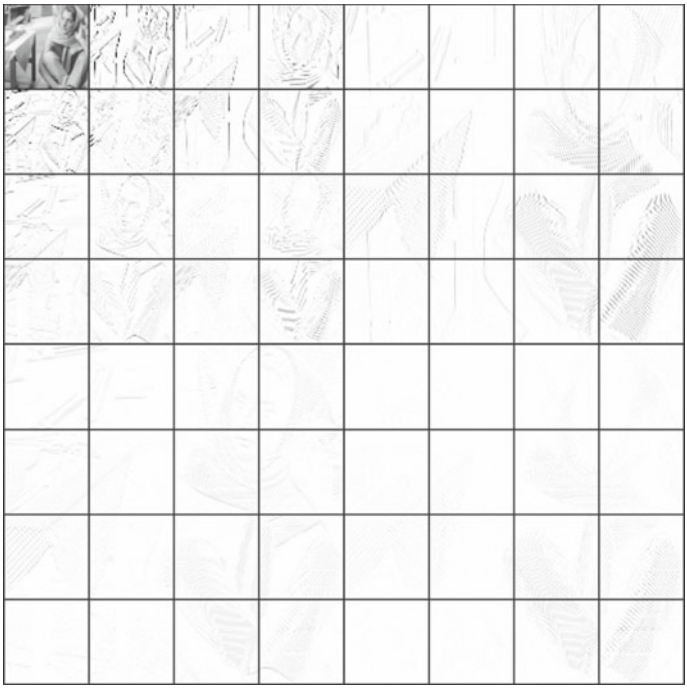


Fig. 9.7 Code block structure of EBCOT

means that the HL sub-band can be treated in the same way as the LH, HH, and LL sub-bands and use the same context model.)

The algorithm uses a *dead-zone* quantizer shown in Fig. 9.8—a double-length region straddling 0. Let $\chi_i[\mathbf{k}] \in \{-1, 1\}$ be the sign of $s_i[\mathbf{k}]$ and let $\nu_i[\mathbf{k}]$ be the quantized magnitude. Explicitly, we have

$$\nu_i[\mathbf{k}] = \frac{\|s_i[\mathbf{k}]\|}{\delta_{\beta_i}} \tag{9.2}$$

where δ_{β_i} is the step size for sub-band β_i, which contains code block B_i. Let $\nu_i^p[\mathbf{k}]$ be the pth bit in the binary representation of $\nu_i[\mathbf{k}]$, where $p = 0$ corresponds to the least significant bit, and let $p_i^{\max}$ be the maximum value of p such that $\nu_i^{p_i^{\max}}[\mathbf{k}] \neq 0$ for at least one sample in the code block.

The encoding process is similar to that of a bitplane coder, in which the most significant bit $\nu_i^{p_i^{\max}}[\mathbf{k}]$ is coded first for all samples in the code block, followed by the next most significant bit $\nu_i^{p_i^{(\max-1)}}[\mathbf{k}]$, and so on, until all bitplanes have been coded. In this way, if the bitstream is truncated, then some samples in the code block may be missing one or more least significant bits. This is equivalent to having used a coarser dead-zone quantizer for these samples.

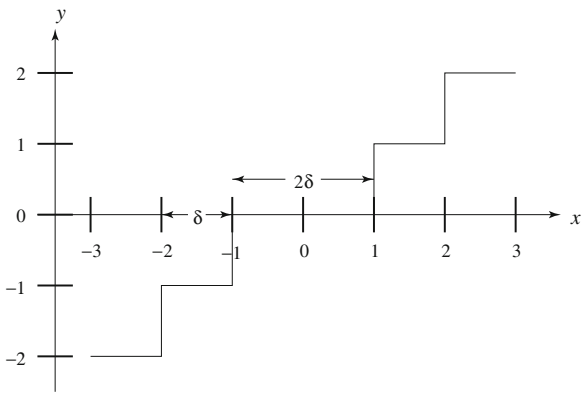


Fig. 9.8 Dead-zone quantizer. The length of the dead-zone is 2δ. Values inside the dead-zone are quantized to 0

In addition, it is important to exploit the previously encoded information about a particular sample and its neighboring samples. This is done in EBCOT by defining a binary valued state variable $\sigma_i[\mathbf{k}]$, which is initially 0 but changes to 1 when the relevant sample's first nonzero bitplane $v_i^p[\mathbf{k}] = 1$ is encoded. This binary state variable is referred to as the *significance* of a sample.

Section 8.8 introduces the zerotree data structure as a way of efficiently coding the bitstream for wavelet coefficients. The underlying observation behind the zerotree data structure is that significant samples tend to be clustered, so that it is often possible to dispose of a large number of samples by coding a single binary symbol.

EBCOT takes advantage of this observation; however, with efficiency in mind, it exploits the clustering assumption only down to relatively large sub-blocks of size 16×16. As a result, each code block is further partitioned into a two-dimensional sequence of sub-blocks $B_i[\mathbf{j}]$. For each bitplane, explicit information is first encoded that identifies sub-blocks containing one or more significant samples. The other sub-blocks are bypassed in the remaining coding phases for that bitplane.

Let $\sigma^p(B_i[\mathbf{j}])$ be the significance of sub-block $B_i[\mathbf{j}]$ in bitplane p. The significance map is coded using a quad-tree. The tree is constructed by identifying the sub-blocks with leaf nodes—that is, $B_i^0[\mathbf{j}] = B_i[\mathbf{j}]$. The higher levels are built using recursion: $B_i^t[\mathbf{j}] = \cup_{\mathbf{z}\in\{0,1\}^2} B_i^{t-1}[2\mathbf{j}+\mathbf{z}]$, $0 \le t \le T$. The root of the tree represents the entire code block: $B_i^T[\mathbf{0}] = \cup_{\mathbf{j}} B_i[\mathbf{j}]$.

The significance of the code block is identified one quad level at a time, starting from the root at $t = T$ and working toward the leaves at $t = 0$. The significance values are then sent to an arithmetic coder for entropy coding. Significance values that are *redundant* are skipped. A value is taken as redundant if any of the following conditions is met:

- The parent is insignificant.
- The current quad was already significant in the previous bitplane.
- This is the last quad visited among those that share the same significant parent, and the other siblings are insignificant.

Table 9.4 Context assignment for the zero coding primitive

Label	LL, LH and HL sub-bands			HH sub-band	
	$h_i[\mathbf{k}]$	$v_i[\mathbf{k}]$	$d_i[\mathbf{k}]$	$d_i[\mathbf{k}]$	$h_i[\mathbf{k}]+v_i[\mathbf{k}]$
0	0	0	0	0	0
1	0	0	1	0	1
2	0	0	>1	0	>1
3	0	1	x	1	0
4	0	2	x	1	1
5	1	0	0	1	>1
6	1	0	>0	2	0
7	1	>0	x	2	>0
8	2	x	x	>2	x

EBCOT uses four different coding primitives to code new information for a single sample in a bitplane p, as follows:

- **Zero coding.** This is used to code $v_i^p[\mathbf{k}]$, given that the quantized sample satisfies $v_i[\mathbf{k}] < 2^{p+1}$. Because the sample statistics are measured to be approximately Markovian, the significance of the current sample depends on the values of its eight immediate neighbors. The significance of these neighbors can be classified into three categories:
  - **Horizontal.** $h_i[\mathbf{k}] = \sum_{z\in\{1,-1\}} \sigma_i[k_1+z, k_2]$, with $0 \le h_i[\mathbf{k}] \le 2$
  - **Vertical.** $v_i[\mathbf{k}] = \sum_{z\in\{1,-1\}} \sigma_i[k_1, k_2+z]$, with $0 \le v_i[\mathbf{k}] \le 2$
  - **Diagonal.** $d_i[\mathbf{k}] = \sum_{z_1,z_2\in\{1,-1\}} \sigma_i[k_1+z_1, k_2+z_2]$, with $0 \le d_i[\mathbf{k}] \le 4$.

  The neighbors outside the code block are considered to be insignificant, but note that sub-blocks are not at all independent. The 256 possible neighborhood configurations are reduced to the nine distinct context assignments listed in Table 9.4.
- **Run-length coding.** The run-length coding primitive is aimed at producing runs of the 1-bit significance values, as a prelude for the arithmetic coding engine. When a horizontal run of insignificant samples having insignificant neighbors is found, it is invoked instead of the zero coding primitive. Each of the following four conditions must be met for the run-length coding primitive to be invoked:
  - Four consecutive samples must be insignificant.
  - The samples must have insignificant neighbors.
  - The samples must be within the same sub-block.
  - The horizontal index k_1 of the first sample must be even.

  The last two conditions are simply for efficiency. When four symbols satisfy these conditions, one special bit is encoded instead, to identify whether any sample in the group is significant in the current bitplane (using a separate context model). If any of the four samples becomes significant, the index of the first such sample is sent as a 2-bit quantity.

Table 9.5 Context assignments for the sign coding primitive

Label	$\hat{\chi}_i[\mathbf{k}]$	$\bar{h}_i[\mathbf{k}]$	$\bar{v}_i[\mathbf{k}]$
4	1	1	1
3	1	0	1
2	1	−1	1
1	−1	1	0
0	1	0	0
1	1	−1	0
2	−1	1	−1
3	−1	0	−1
4	−1	−1	−1

- **Sign coding.** The sign coding primitive is invoked at most once for each sample, immediately after the sample makes a transition from being insignificant to significant during a zero coding or run-length coding operation. Since it has four horizontal and vertical neighbors, each of which may be insignificant, positive, or negative, there are $3^4 = 81$ different context configurations. However, exploiting both horizontal and vertical symmetry and assuming that the conditional distribution of $\chi_i[\mathbf{k}]$, given any neighborhood configuration, is the same as that of $-\chi_i[\mathbf{k}]$, the number of contexts is reduced to 5. Let $\bar{h}_i[\mathbf{k}]$ be 0 if both horizontal neighbors are insignificant, 1 if at least one horizontal neighbor is positive, or -1 if at least one horizontal neighbor is negative (and $\bar{v}_i[\mathbf{k}]$ is defined similarly). Let $\hat{\chi}_i[\mathbf{k}]$ be the sign prediction. The binary symbol coded using the relevant context is $\chi_i[\mathbf{k}] \cdot \hat{\chi}_i[\mathbf{k}]$. Table 9.5 lists these context assignments.
- **Magnitude refinement.** This primitive is used to code the value of $v_i^p[\mathbf{k}]$, given that $v_i[\mathbf{k}] \geq 2^{p+1}$. Only three context models are used for the magnitude refinement primitive. A second state variable $\tilde{\sigma}_i[\mathbf{k}]$ is introduced that changes from 0 to 1 after the magnitude refinement primitive is first applied to $s_i[\mathbf{k}]$. The context models depend on the value of this state variable: $v_i^p[\mathbf{k}]$ is coded with context 0 if $\tilde{\sigma}[\mathbf{k}] = h_i[\mathbf{k}] = v_i[\mathbf{k}] = 0$, with context 1 if $\tilde{\sigma}_i[\mathbf{k}] = 0$ and $h_i[\mathbf{k}] + v_i[\mathbf{k}] \neq 0$, and with context 2 if $\tilde{\sigma}_i[\mathbf{k}] = 1$.

To ensure that each code block has a finely embedded bitstream, the coding of each bitplane p proceeds in four distinct passes, ($\mathcal{P}_1^p$) to ($\mathcal{P}_4^p$):

- **Forward-significance-propagation pass** ($\mathcal{P}_\mathbf{1}^\mathbf{p}$). The sub-block samples are visited in scanline order. Insignificant samples and samples that do not satisfy the neighborhood requirement are skipped. For the LH, HL, and LL sub-bands, the neighborhood requirement is that at least one of the horizontal neighbors has to be significant. For the HH sub-band, the neighborhood requirement is that at least one of the four diagonal neighbors must be significant. For significant samples that pass the neighborhood requirement, the zero coding and run-length coding primitives are invoked as appropriate, to determine whether the sample first becomes significant in bitplane p. If so, the sign coding primitive is invoked to encode the

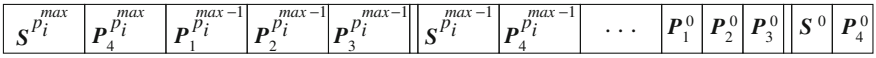


Fig. 9.9 Appearance of coding passes and quad-tree codes in each block's embedded bitstream

sign. This is called the forward-significance-propagation pass, because a sample that has been found to be significant helps in the new significance determination steps that propagate in the direction of the scan.

- **Reverse-significance-propagation pass** ($\mathcal{P}_{\mathbf{2}}^{\mathbf{p}}$). This pass is identical to $\mathcal{P}_1^p$, except that it proceeds in the reverse order. The neighborhood requirement is relaxed to include samples that have at least one significant neighbor in any direction.
- **Magnitude refinement pass** ($\mathcal{P}_{\mathbf{3}}^{\mathbf{p}}$). This pass encodes samples that are already significant but that have not been coded in the previous two passes. Such samples are processed with the magnitude refinement primitive.
- **Normalization pass** ($\mathcal{P}_{\mathbf{4}}^{\mathbf{p}}$). The value $v_i^p[\mathbf{k}]$ of all samples not considered in the previous three coding passes is coded using the sign coding and run-length coding primitives, as appropriate. If a sample is found to be significant, its sign is immediately coded using the sign coding primitive.

Figure 9.9 shows the layout of coding passes and quad-tree codes in each block's embedded bitstream. $\mathcal{S}^p$ denotes the quad-tree code identifying the significant sub-blocks in bitplane p. Notice that for any bitplane p, $\mathcal{S}^p$ appears just before the final coding pass $\mathcal{P}_4^p$, not the initial coding pass $\mathcal{P}_1^p$. This implies that sub-blocks that become significant for the first time in bitplane p are ignored until the final pass.

Post Compression Rate-Distortion Optimization

After all the sub-band samples have been compressed, a *post compression rate distortion* (*PCRD*) step is performed. The goal of PCRD is to produce an optimal truncation of each code block's independent bitstream such that distortion is minimized, subject to the bit-rate constraint. For each truncated embedded bitstream of code block B_i having rate $R_i^{n_i}$, the overall distortion of the reconstructed image is (assuming distortion is additive)

$$D = \sum_i D_i^{n_i} \tag{9.3}$$

where $D_i^{n_i}$ is the distortion from code block B_i having truncation point n_i. For each code block B_i, distortion is computed by

$$D_i^n = w_{b_i}^2 \sum_{\mathbf{k} \in B_i} (\hat{s}_i^n[\mathbf{k}] - s_i[\mathbf{k}])^2 \tag{9.4}$$

where $s_i[\mathbf{k}]$ is the 2D sequence of sub-band samples in code block B_i and $\hat{s}_i^n[\mathbf{k}]$ is the quantized representation of these samples associated with truncation point n. The value $w_{b_i}^2$ is the L_2 norm of the wavelet basis function for the sub-band b_i that contains code block B_i.

The optimal selection of truncation points n_i can be formulated into a minimization problem subject to the following constraint:

$$R = \sum_i R_i^{n_i} \leq R^{\max} \tag{9.5}$$

where $R^{\max}$ is the available bit-rate. For some λ, any set of truncation points $\{n_i^\lambda\}$ that minimizes

$$(D(\lambda) + \lambda R(\lambda)) = \sum_i \left(D_i^{n_i^\lambda} + \lambda R_i^{n_i^\lambda} \right) \tag{9.6}$$

is optimal in the rate-distortion sense. Thus, finding the set of truncation points that minimizes Eq. (9.6) with total rate $R(\lambda) = R^{\max}$ would yield the solution to the entire optimization problem.

Since the set of truncation points is discrete, it is generally not possible to find a value of λ for which $R(\lambda)$ is exactly equal to $R^{\max}$. However, since the EBCOT algorithm uses relatively small code blocks, each of which has many truncation points, it is sufficient to find the smallest value of λ such that $R(\lambda) \leq R^{\max}$.

It is easy to see that each code block B_i can be minimized independently. Let $\mathcal{N}_i$ be the set of feasible truncation points and let $j_1 < j_2 < \cdots$ be an enumeration of these feasible truncation points having corresponding distortion-rate slopes given by the ratios

$$S_i^{j_k} = \frac{\Delta D_i^{j_k}}{\Delta R_i^{j_k}} \tag{9.7}$$

where $\Delta R_i^{j_k} = R_i^{j_k} - R_i^{j_{k-1}}$ and $\Delta D_i^{j_k} = D_i^{j_k} - D_i^{j_{k-1}}$. It is evident that the slopes are strictly decreasing, since the operational distortion-rate curve is convex and strictly decreasing. The minimization problem for a fixed value of λ is simply the trivial selection

$$n_i^\lambda = \max \left\{ j_k \in \mathcal{N}_i | S_i^{j_k} > \lambda \right\} \tag{9.8}$$

The optimal value λ^* can be found using a simple bisection method operating on the distortion-rate curve. A detailed description of this method can be found in [8].

Layer Formation and Representation

The EBCOT algorithm offers both resolution and quality scalability, as opposed to other well-known scalable image compression algorithms such as EZW and SPIHT, which offer only quality scalability. This functionality is achieved using a layered bitstream organization and a two-tiered coding strategy.

The final bitstream EBCOT produces is composed of a collection of quality layers. The quality layer $\mathcal{Q}_1$ contains the initial $R_i^{n_i^1}$ bytes of each code block B_i and the other layers $\mathcal{Q}_q$ contain the incremental contribution $L_i^q = R_i^{n_i^q} - R^{n_i^{q-1}} \geq 0$ from

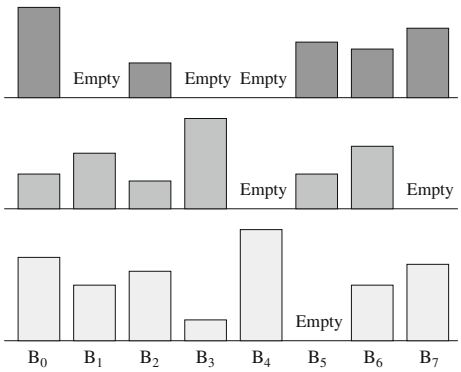


Fig. 9.10 Three quality layers with eight blocks each

code block B_i. The quantity n_i^q is the truncation point corresponding to the rate-distortion threshold λ_q selected for the qth quality layer. Figure 9.10 illustrates the layered bitstream (after [7]).

Along with these incremental contributions, auxiliary information such as the length L_i^q, the number of new coding passes $N_i^q = n_i^q - n_i^{q-1}$, the value $p_i^{\max}$ when B_i makes its first nonempty contribution to quality layer $\mathcal{Q}_q$, and the index q_i of the quality layer to which B_i first makes a nonempty contribution must be explicitly stored. This auxiliary information is compressed in the second-tier coding engine. Hence, in this two-tiered architecture, the first tier produces the embedded block bitstreams, while the second encodes the block contributions to each quality layer.

The focus of this subsection is the second-tier processing of the auxiliary information accompanying each quality layer. The second-tier coding engine handles carefully the two quantities that exhibit substantial interblock redundancy. These two quantities are $p_i^{\max}$ and the index q_i of the quality layer to which B_i first makes a nonempty contribution.

The quantity q_i is coded using a separate embedded quad-tree code within each sub-band. Let $B_i^0 = B_i$ be the leaves and B_i^T be the root of the tree that represents the entire sub-band. Let $q_i^t = \min\{q_j | B_j \subset B_i^t\}$ be the index of the first layer in which any code block in quad B_i^t makes a nonempty contribution. A single bit identifies whether $q_i^t > q$ for each quad at each level t, with redundant quads omitted. A quad is redundant if either $q_i^t < q - 1$ or $q_j^{t+1} > q$ for some parent quad B_j^{t+1}.

The other redundant quantity to consider is $p_i^{\max}$. It is clear that $p_i^{\max}$ is irrelevant until the coding of the quality layer $\mathcal{Q}_q$. Thus, any unnecessary information concerning $p_i^{\max}$ need not be sent until we are ready to encode $\mathcal{Q}_q$. EBCOT does this using a modified embedded quad-tree driven from the leaves rather than from the root.

Let B_i^t be the elements of the quad-tree structure built on top of the code blocks B_i from any sub-band, and let $p_i^{\max,t} = \max\{p_j^{\max} | B_j \subset B_i^t\}$. In addition, let $B_{i_t}^t$ be the ancestor of quads from which B_i descends and let P be a value guaranteed to be larger than $p_i^{\max}$ for any code block B_i. When code block B_i first contributes to the bitstream in quality layer $\mathcal{Q}_q$, the value of $p_i^{\max} = p_{i_0}^{\max,0}$ is coded using the following algorithm:

- For $p = P-1, P-2, \ldots, 0$
  - Send binary digits to identify whether $p_{i_t}^{\max,t} < p$. The redundant bits are skipped.
  - If $p_i^{\max} = p$, then stop.

The redundant bits are those corresponding to the condition $p_{i_t}^{\max,t} < p$ that can be inferred either from ancestors such that $p_{i_{t+1}}^{\max,t+1} < p$ or from the partial quad-tree code used to identify $p_j^{\max}$ for a different code block B_j.

9.2.2 Adapting EBCOT to JPEG2000

JPEG2000 uses the EBCOT algorithm as its primary coding method. However, the algorithm is slightly modified to enhance compression efficiency and reduce computational complexity.

To further enhance compression efficiency, as opposed to initializing the entropy coder using equiprobable states for all contexts, the JPEG2000 standard makes an assumption of highly skewed distributions for some contexts, to reduce the model adaptation cost for typical images. Several small adjustments are made to the original algorithm to further reduce its execution time.

First, a low complexity arithmetic coder that avoids multiplications and divisions, known as the MQ coder [9], replaces the usual arithmetic coder used in the original algorithm. Furthermore, JPEG2000 does not transpose the HL sub-band's code blocks. Instead, the corresponding entries in the zero coding context assignment map are transposed.

To ensure a consistent scan direction, JPEG2000 combines the forward- and reverse-significance-propagation passes into a single forward-significance-propagation pass with a neighborhood requirement equal to that of the original reverse pass. In addition, reducing the sub-block size to 4×4 from the original 16×16 eliminates the need to explicitly code sub-block significance. The resulting probability distribution for these small sub-blocks is highly skewed, so the coder behaves as if all sub-blocks are significant.

The cumulative effect of these modifications is an increase of about 40 % in software execution speed, with an average loss of about 0.15 dB relative to the original algorithm.

9.2.3 Region-of-Interest Coding

A significant feature of the new JPEG2000 standard is the ability to perform region-of-interest (ROI) coding. Here, particular regions of the image may be coded with better quality than the rest of the image or the background. The method is called MAXSHIFT, a scaling-based method that scales up the coefficients in the ROI so that they are placed into higher bitplanes. During the embedded coding process, the resulting bits are placed in front of the non-ROI part of the image. Therefore, given

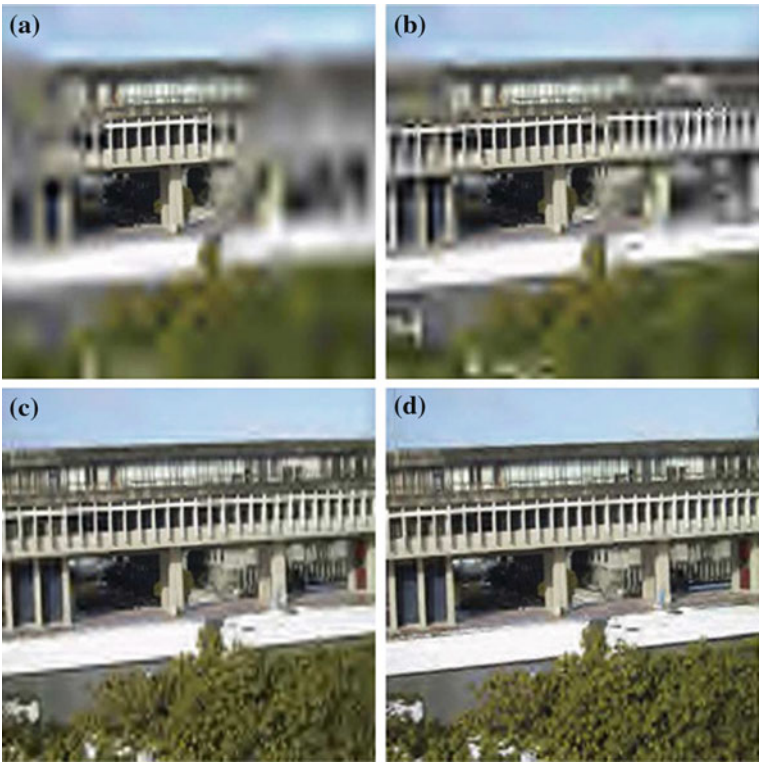


Fig. 9.11 Region-of-interest (ROI) coding of an image with increasing bit-rate using a circularly shaped ROI: **a** 0.4 bpp; **b** 0.5 bpp; **c** 0.6 bpp; **d** 0.7 bpp

a reduced bitrate, the ROI will be decoded and refined before the rest of the image. As a result of these mechanisms, the ROI will have much better quality than the background.

One thing to note is that regardless of scaling, full decoding of the bitstream will result in reconstruction of the entire image with the highest fidelity available. Figure 9.11 demonstrates the effect of region-of-interest coding as the target bitrate of the sample image is increased.

9.2.4 Comparison of JPEG and JPEG2000 Performance

After studying the internals of the JPEG2000 compression algorithm, a natural question that comes to mind is: how well does JPEG2000 perform compared to other well-known standards, in particular JPEG? Many comparisons have been made between JPEG and other well-known standards, so here we compare JPEG2000 only to the popular JPEG.

Various criteria, such as computational complexity, error resilience, compression efficiency, and so on, have been used to evaluate the performance of systems. Since our main focus is on the compression aspect of the JPEG2000 standard, here we simply compare compression efficiency. (Interested readers can refer to [6,10] for comparisons using other criteria.)

Given a fixed bitrate, let us compare quality of compressed images quantitatively by the PSNR: for color images, the PSNR is calculated based on the average of the mean square error of all the RGB components. Also, we visually show results for both JPEG2000 and JPEG compressed images, so that you can make your own qualitative assessment. We perform a comparison for three categories of images: natural, computer generated, and medical, using three images from each category. The test images used are shown on the textbook web site.

For each image, we compress using JPEG and JPEG2000, at four bitrates: 0.25 bpp, 0.5 bpp, 0.75 bpp, and 1.0 bpp. Figure 9.12 shows plots of the average PSNR of the images in each category against bitrate. We see that JPEG2000 substantially outperforms JPEG in all categories.

For a qualitative comparison of the compression results, let us choose a single image and show decompressed output for the two algorithms using a low bitrate (0.75 bpp) and the lowest bitrate (0.25 bpp). From the results in Fig. 9.13, it should be obvious that images compressed using JPEG2000 show significantly fewer visual artifacts.

9.3 The JPEG-LS Standard

Generally, we would likely apply a lossless compression scheme to images that are critical in some sense, say medical images of a brain, or perhaps images that are difficult or costly to acquire. A scheme in competition with the lossless mode provided in JPEG2000 is the JPEG-LS standard, specifically aimed at lossless encoding [11]. The main advantage of JPEG-LS over JPEG2000 is that JPEG-LS is based on a low complexity algorithm. JPEG-LS is part of a larger ISO effort aimed at better compression of medical images.

JPEG-LS is in fact the current ISO/ITU standard for lossless or "near lossless" compression of continuous-tone images. The core algorithm in JPEG-LS is called *LOw COmplexity LOssless COmpression for Images* (*LOCO-I*), proposed by Hewlett-Packard [11]. The design of this algorithm is motivated by the observation that complexity reduction is often more important overall than any small increase in compression offered by more complex algorithms.

LOCO-I exploits a concept called *context modeling*. The idea of context modeling is to take advantage of the structure in the input source—conditional probabilities of what pixel values follow from each other in the image. This extra knowledge is called the *context*. If the input source contains substantial structure, as is usually the case, we could potentially compress it using fewer bits than the 0th-order entropy.

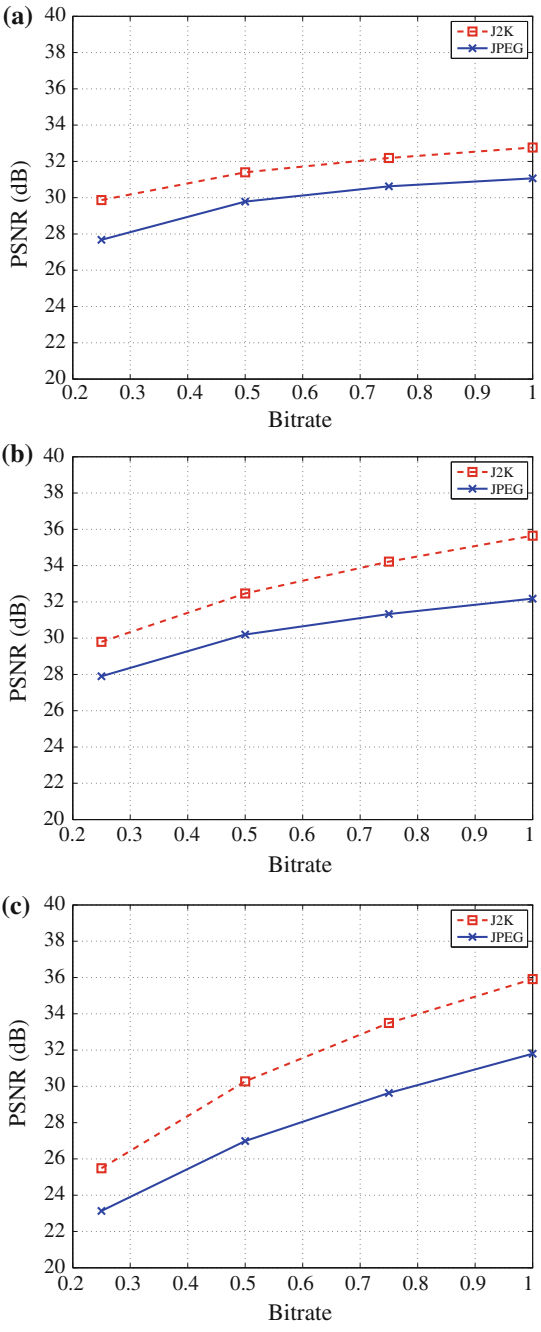


Fig. 9.12 Performance comparison for JPEG and JPEG2000 on different image types: **a** Natural images; **b** Computer-generated images; **c** Medical images

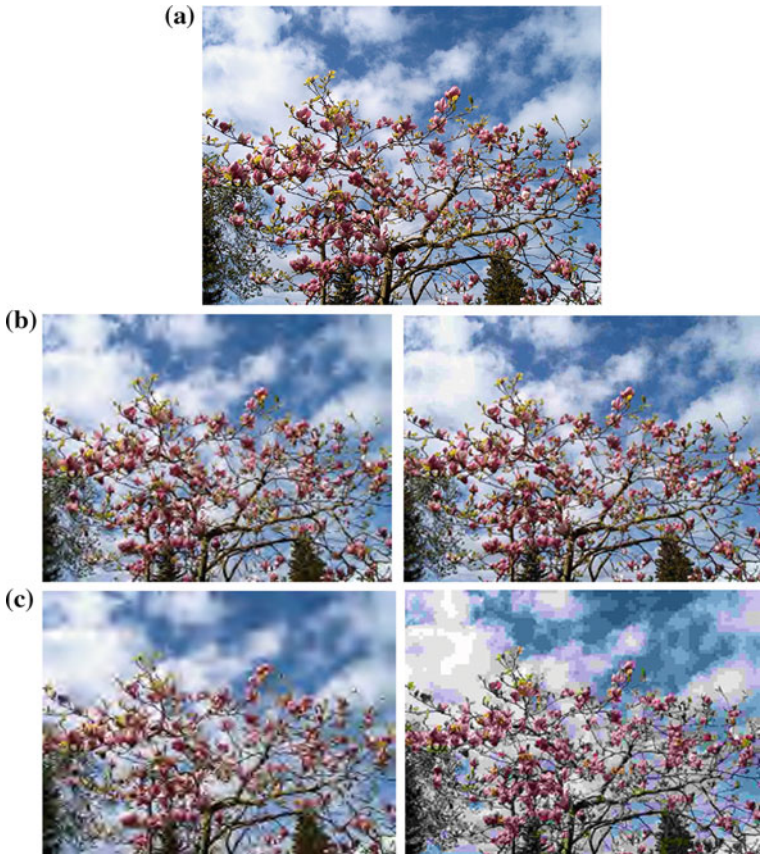


Fig. 9.13 Comparison of JPEG and JPEG2000: **a** original image; **b** JPEG (*left*) and JPEG2000 (*right*) images compressed at 0.75 bpp; **c** JPEG (*left*) and JPEG2000 (*right*) images compressed at 0.25 bpp

As a simple example, suppose we have a binary source with $P(0) = 0.4$ and $P(1) = 0.6$. Then the 0th-order entropy $H(S) = -0.4\log_2(0.4) - 0.6\log_2(0.6) = 0.97$. Now suppose we also know that this source has the property that if the previous symbol is 0, the probability of the current symbol being 0 is 0.8, and if the previous symbol is 1, the probability of the current symbol being 0 is 0.1.

If we use the previous symbol as our *context*, we can divide the input symbols into two sets, corresponding to context 0 and context 1, respectively. Then the entropy of each of the two sets is

$$H(S_1) = -0.8\log_2(0.8) - 0.2\log_2(0.2) = 0.72$$
$$H(S_2) = -0.1\log_2(0.1) - 0.9\log_2(0.9) = 0.47$$

The average bit-rate for the entire source would be $0.4 \times 0.72 + 0.6 \times 0.47 = 0.57$, which is substantially less than the 0th-order entropy of the entire source in this case.

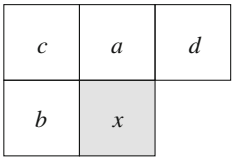


Fig. 9.14 JPEG-LS context model

LOCO-I uses a context model shown in Fig. 9.14. In raster scan order, the context pixels a, b, c, and d all appear before the current pixel x. Thus, this is called a *causal context*.

LOCO-I can be broken down into three components:

- **Prediction.** Predicting the value of the next sample x' using a causal template
- **Context determination.** Determining the context in which x' occurs
- **Residual coding.** Entropy coding of the prediction residual conditioned by the context of x'.

9.3.1 Prediction

A better version of prediction can use an adaptive model based on a calculation of the local edge direction. However, because JPEG-LS is aimed at low complexity, the LOCO-I algorithm instead uses a fixed predictor that performs primitive tests to detect vertical and horizontal edges. The fixed predictor used by the algorithm is given as follows:

$$\hat{x}' = \begin{cases} \min(a, b) & c \geq \max(a, b) \\ \max(a, b) & c \leq \min(a, b) \\ a + b - c & \text{otherwise} \end{cases} \tag{9.9}$$

It is easy to see that this predictor switches between three simple predictors. It outputs a when there is a vertical edge to the left of the current location; it outputs b when there is a horizontal edge above the current location; and finally it outputs $a + b - c$ when the neighboring samples are relatively smooth.

9.3.2 Context Determination

The context model that conditions the current prediction error (the *residual*) is indexed using a three-component context vector $\mathbf{Q} = (q_1, q_2, q_3)$, whose components are

$$\begin{aligned} q_1 &= d - b \\ q_2 &= b - c \\ q_3 &= c - a \end{aligned} \tag{9.10}$$

These differences represent the local gradient that captures the local smoothness or edge contents that surround the current sample. Because these differences can

potentially take on a wide range of values, the underlying context model is huge, making the context modeling approach impractical. To solve this problem, parameter reduction methods are needed.

An effective method is to quantize these differences so that they can be represented by a limited number of values. The components of $\mathbf{Q}$ are quantized using a quantizer with decision boundaries $-T, \cdots, -1, 0, 1, \cdots, T$. In JPEG-LS, $T = 4$. The context size is further reduced by replacing any context vector $\mathbf{Q}$ whose first element is negative by $-\mathbf{Q}$. Therefore, the number of different context states is $\frac{(2T+1)^3+1}{2} = 365$ in total. The vector $\mathbf{Q}$ is then mapped into an integer in [0, 364].

9.3.3 Residual Coding

For any image, the prediction residual has a finite size, α. For a given prediction $\hat{x}$, the residual ε is in the range $-\hat{x} \leq \varepsilon < \alpha - \hat{x}$. Since the value $\hat{x}$ can be generated by the decoder, the dynamic range of the residual ε can be reduced modulo α and mapped into a value between $-\lfloor \frac{\alpha}{2} \rfloor$ and $\lceil \frac{\alpha}{2} \rceil - 1$.

It can be shown that the error residuals follow a *two-sided geometric distribution* (TSGD). As a result, they are coded using adaptively selected codes based on *Golomb codes*, which are optimal for sequences with geometric distributions [12].

9.3.4 Near-Lossless Mode

The JPEG-LS standard also offers a near-lossless mode, in which the reconstructed samples deviate from the original by no more than an amount δ. The main lossless JPEG-LS mode can be considered a special case of the near-lossless mode with $\delta = 0$. Near-lossless compression is achieved using quantization: residuals are quantized using a uniform quantizer having intervals of length $2\delta + 1$. The quantized values of ε are given by

$$Q(\varepsilon) = \text{sign}(\varepsilon) \left\lfloor \frac{|\varepsilon| + \delta}{2\delta + 1} \right\rfloor \tag{9.11}$$

Since δ can take on only a small number of integer values, the division operation can be implemented efficiently using lookup tables. In near-lossless mode, the prediction and context determination step described previously are based on the quantized values only.

9.4 Bi-level Image Compression Standards

As more and more documents are handled in electronic form, efficient methods for compressing bi-level images (those with only 1-bit, black-and-white pixels) are sometimes called for. A familiar example is fax images. Algorithms that take advantage of the binary nature of the image data often perform better than generic image compression algorithms. Earlier facsimile standards, such as G3 and G4, use simple

models of the structure of bi-level images. Each scanline in the image is treated as a run of black-and-white pixels. However, considering the neighboring pixels and the nature of data to be coded allows much more efficient algorithms to be constructed. This section examines the JBIG standard and its successor, JBIG2, as well as the underlying motivations and principles for these two standards.

9.4.1 The JBIG Standard

JBIG is the coding standard recommended by the Joint Bi-level Image Processing Group for binary images. This lossless compression standard is used primarily to code scanned images of printed or handwritten text, computer-generated text, and facsimile transmissions. It offers progressive encoding and decoding capability, in the sense that the resulting bitstream contains a set of progressively higher resolution images. This standard can also be used to code grayscale and color images by coding each bitplane independently, but this is not the main objective.

The JBIG compression standard has three separate modes of operation: *progressive*, *progressive-compatible sequential*, and *single-progression sequential*. The progressive-compatible sequential mode uses a bitstream compatible with the progressive mode. The only difference is that the data is divided into *strips* in this mode.

The single-progression sequential mode has only a single lowest resolution layer. Therefore, an entire image can be coded without any reference to other higher-resolution layers. Both these modes can be viewed as special cases of the progressive mode. Therefore, our discussion covers only the progressive mode.

The JBIG encoder can be decomposed into two components:

- Resolution-reduction and differential-layer encoder
- Lowest resolution layer encoder.

The input image goes through a sequence of resolution-reduction and differential-layer encoders. Each is equivalent in functionality, except that their input images have different resolutions. Some implementations of the JBIG standard may choose to recursively use one such physical encoder. The lowest resolution image is coded using the lowest resolution-layer encoder. The design of this encoder is somewhat simpler than that of the resolution-reduction and differential-layer encoders, since the resolution-reduction and deterministic-prediction operations are not needed.

9.4.2 The JBIG2 Standard

While the JBIG standard offers both lossless and progressive (lossy to lossless) coding abilities, the lossy image produced by this standard has significantly lower quality than the original, because the lossy image contains at most only one-quarter of the number of pixels in the original image. By contrast, the JBIG2 standard is explicitly designed for lossy, lossless, and lossy to lossless image compression. The design goal for JBIG2 aims not only at providing superior lossless compression

performance over existing standards but also at incorporating lossy compression at a much higher compression ratio, with as little visible degradation as possible.

A unique feature of JBIG2 is that it is both *quality progressive* and *content progressive*. By quality progressive, we mean that the bitstream behaves similarly to that of the JBIG standard, in which the image quality progresses from lower to higher (or possibly lossless) quality. On the other hand, content progressive allows different types of image data to be added progressively. The JBIG2 encoder decomposes the input bi-level image into regions of different attributes and codes each separately, using different coding methods.

As in other image compression standards, only the JBIG2 bitstream, and thus the decoder, is explicitly defined. As a result, any encoder that produces the correct bitstream is "compliant," regardless of the actions it actually takes. Another feature of JBIG2 that sets it apart from other image compression standards is that it is able to represent multiple pages of a document in a single file, enabling it to exploit interpage similarities.

For example, if a character appears on one page, it is likely to appear on other pages as well. Thus, using a dictionary-based technique, this character is coded only once instead of multiple times for every page on which it appears. This compression technique is somewhat analogous to video coding, which exploits interframe redundancy to increase compression efficiency.

JBIG2 offers content progressive coding and superior compression performance through *model-based coding*, in which different models are constructed for different data types in an image, realizing additional coding gain.

Model-Based Coding

The idea behind model-based coding is essentially the same as that of context-based coding. From the study of the latter, we know we can realize better compression performance by carefully designing a context template and accurately estimating the probability distribution for each context. Similarly, if we can separate the image content into different categories and derive a model specifically for each, we are much more likely to accurately model the behavior of the data and thus achieve higher compression ratio.

In the JBIG style of coding, adaptive and model templates capture the structure within the image. This model is general, in the sense that it applies to all kinds of data. However, being general implies that it does not explicitly deal with the structural differences between text and halftone data that comprise nearly all the contents of bi-level images. JBIG2 takes advantage of this by designing custom models for these data types.

The JBIG2 specification expects the encoder to first segment the input image into regions of different data types, in particular, text and halftone regions. Each region is then coded independently, according to its characteristics.

Text Region Coding

Each text region is further segmented into pixel blocks containing connected black pixels. These blocks correspond to characters that make up the content of this region. Then, instead of coding all pixels of each character, the bitmap of one representative instance of this character is coded and placed into a *dictionary*. For any character to be coded, the algorithm first tries to find a match with the characters in the dictionary. If one is found, then both a pointer to the corresponding entry in the dictionary and the position of the character on the page are coded. Otherwise, the pixel block is coded directly and added to the dictionary. This technique is referred to as *pattern matching and substitution* in the JBIG2 specification.

However, for *scanned* documents, it is unlikely that two instances of the same character will match pixel by pixel. In this case, JBIG2 allows the option of including refinement data to reproduce the original character on the page. The refinement data codes the current character using the pixels in the matching character *in the dictionary*. The encoder has the freedom to choose the refinement to be exact or lossy. This method is called *soft pattern matching*.

The numeric data, such as the index of matched character in the dictionary and the position of the characters on the page, are either bitwise or Huffman encoded. Each bitmap for the characters in the dictionary is coded using JBIG-based techniques.

Halftone-Region Coding

The JBIG2 standard suggests two methods for halftone image coding. The first is similar to the context-based arithmetic coding used in JBIG. The only difference is that the new standard allows the context template to include as many as 16 template pixels, four of which may be adaptive.

The second method is called *descreening*. This involves converting back to grayscale and coding the grayscale values. In this method, the bi-level region is divided into blocks of size $m_b \times n_b$. For an $m \times n$ bi-level region, the resulting grayscale image has dimension $m_g = \lfloor (m + (m_b - 1))/m_b \rfloor$ by $n_g = \lfloor (n + (n_b - 1))/n_b \rfloor$. The grayscale value is then computed to be the sum of the binary pixel values in the corresponding $m_b \times n_b$ block. The bitplanes of the grayscale image are coded using context-based arithmetic coding. The grayscale values are used as indices into a dictionary of halftone bitmap patterns. The decoder can use this value to index into this dictionary, to reconstruct the original halftone image.

Preprocessing and Postprocessing

JBIG2 allows the use of lossy compression but does not specify a method for doing so. From the decoder point of view, the decoded bitstream is lossless with respect to the image encoded by the encoder, although not necessarily with respect to the

original image. The encoder may modify the input image in a preprocessing step, to increase coding efficiency. The preprocessor usually tries to change the original image to lower the code length in a way that does not generally affect the image's appearance. Typically, it tries to remove noisy pixels and smooth out pixel blocks.

Postprocessing, another issue not addressed by the specification, can be especially useful for halftones, potentially producing more visually pleasing images. It is also helpful to tune the decoded image to a particular output device, such as a laser printer.

9.5 Exercises

1. You are given a computer cartoon picture and a photograph. If you have a choice of using either JPEG compression or GIF, which compression would you apply for these two images? Justify your answer.
2. Suppose we view a decompressed 512×512 JPEG image but use only the *color* part of the stored image information, not the luminance part, to decompress. What does the 512×512 color image look like? Assume JPEG is compressed using a 4:2:0 scheme.
3. An X-ray photo is usually a graylevel image. It often has low contrast and low graylevel intensities, i.e., all intensity values are in the range of $[a, b]$, where a and b are positive integers, much less than the maximum intensity value 255 if it is an 8-bit image. In order to enhance the appearance of this image, a simple "stretch" operation can be used to convert all original intensity values f_0 to f:

$$f = \frac{255}{b-a} \cdot (f_0 - a).$$

   For simplicity, assuming $f_0(i, j)$ and $f(i, j)$ are 8×8 images:
   (a) If the DC value for the original image f_0 is m, what is the DC value for the stretched image f?
   (b) If one of the AC values $F_0(2, 1)$ for the original image f_0 is n, what is the $F(2, 1)$ value for the stretched image f?
4. (a) JPEG uses the Discrete Cosine Transform (DCT) for image compression.
      i. What is the value of F(0, 0) if the image $f(i, j)$ is as below?
      ii. Which AC coefficient $|F(u, v)|$ is the largest for this $f(i, j)$? Why? Is this $F(u, v)$ positive or negative? Why?

20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20
80	80	80	80	80	80	80	80
80	80	80	80	80	80	80	80
140	140	140	140	140	140	140	140
140	140	140	140	140	140	140	140
200	200	200	200	200	200	200	200
200	200	200	200	200	200	200	200

(b) Show in detail how a three-level hierarchical JPEG will encode the image above, assuming that

i. The encoder and decoder at all three levels use Lossless JPEG.
ii. *Reduction* simply averages each 2×2 block into a single pixel value.
iii. *Expansion* duplicates the single pixel value four times.

5. In JPEG, the Discrete Cosine Transform is applied to 8×8 blocks in an image. For now, let's call it DCT-8. Generally, we can define a DCT-N to be applied to $N \times N$ blocks in an image. DCT-N is defined as:

$$F_N(u, v) = \frac{2C(u)C(v)}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \cos \frac{(2i+1)u\pi}{2N} \cos \frac{(2j+1)v\pi}{2N} f(i, j)$$

$$C(\xi) = \begin{cases} \frac{\sqrt{2}}{2} & \text{for } \xi = 0 \\ 1 & \text{otherwise} \end{cases}$$

Given $f(i, j)$ as below, show your work for deriving all pixel values of $F_2(u, v)$. (That is, show the result of applying DCT-2 to the image below.)

$$\begin{matrix} 100 & -100 & 100 & -100 & 100 & -100 & 100 & -100 \\ 100 & -100 & 100 & -100 & 100 & -100 & 100 & -100 \\ 100 & -100 & 100 & -100 & 100 & -100 & 100 & -100 \\ 100 & -100 & 100 & -100 & 100 & -100 & 100 & -100 \\ 100 & -100 & 100 & -100 & 100 & -100 & 100 & -100 \\ 100 & -100 & 100 & -100 & 100 & -100 & 100 & -100 \\ 100 & -100 & 100 & -100 & 100 & -100 & 100 & -100 \\ 100 & -100 & 100 & -100 & 100 & -100 & 100 & -100 \end{matrix}$$

6. According to the DCT-N definition above, $F_N(1)$ and $F_N(N-1)$ are the AC coefficients representing the lowest and highest spatial frequencies, respectively.
   (a) It is known that $F_{16}(1)$ and $F_8(1)$ *do not* capture the same (lowest) frequency response in image filtering. Explain why.
   (b) Do $F_{16}(15)$ and $F_8(7)$ capture the same (highest) frequency response?
7. (a) How many principal modes does JPEG have? What are their names?
   (b) In the hierarchical model, explain briefly why we must include an encode/decode cycle on the coder side before transmitting difference images to the decode side.
   (c) What are the two methods used to decode only part of the information in a JPEG file, so that the image can be coarsely displayed quickly and iteratively increased in quality?
8. Could we use wavelet-based compression in ordinary JPEG? How?
9. We decide to create a new image compression standard based on JPEG, for use with images that will be viewed by an alien species. What part of the JPEG workflow would we likely have to change?
10. Unlike EZW, EBCOT does not explicitly take advantage of the spatial relationships of wavelet coefficients. Instead, it uses the PCRD optimization approach. Discuss the rationale behind this approach.

11. Is the JPEG2000 bitstream SNR scalable? If so, explain how it is achieved using the EBCOT algorithm.
12. Implement transform coding, quantization, and hierarchical coding for the encoder and decoder of a three-level Hierarchical JPEG. Your code should include a (minimal) graphical user interface for the purpose of demonstrating your results. You do not need to implement the entropy (lossless) coding part; optionally, you may include any publicly available code for it.

References

1. W.B. Pennebaker, J.L. Mitchell, *The JPEG Still Image Data Compression Standard* (Van Nostrand Reinhold, New York, 1992)
2. V. Bhaskaran, K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, 2nd edn. (Kluwer Academic Publishers, Boston, 1997)
3. ISO/IEC 15444–1, *Information Technology - JPEG 2000 Image Coding System: Core Coding System*. ISO/IEC, (2004)
4. D.S. Taubman, M.W. Marcellin, *JPEG2000: Image Compression Fundamentals* (Kluwer Academic Publishers, Standards and Practice, 2002)
5. M. Rabbani, R. Joshi, An Overview of the JPEG 2000 Still Image Compression Standard. Signal Processing: Image Communication **17**, 3–48 (2002)
6. P. Schelkens, A. Skodras, T. Ebrahimi (eds.). The JPEG 2000 Suite. (Wiley, 2009)
7. D. Taubman, High performance scalable image compression with EBCOT. IEEE Trans. Image Process. **9**(7), 1158–1170 (2000)
8. K. Ramachandran, M. Vetterli, Best wavelet packet basis in a rate-distortion sense. IEEE Trans. Image Process. **2**, 160–173 (1993)
9. I. Ueno, F. Ono, T. Yanagiya, T. Kimura, M. Yoshida. Proposal of the Arithmetic Coder for JPEG2000. ISO/IEC JTC1/SC29/WG1 N1143, (1999)
10. D. Santa-Cruz, R. Grosbois, T. Ebrahimi, JPEG 2000 Performance Evaluation and Assessment. Signal Process.: Image Commun. **17**, 113–130 (2002)
11. M. Weinberger, G. Seroussi, G. Sapiro, The LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS. Technical Report HPL-98-193R1, Hewlett-Packard Technical Report, (1998)
12. N. Merhav, G. Seroussi, M.J. Weinberger, Optimal prefix codes for sources with two-sided geometric distributions. IEEE Trans. on Inf. Theory **46**(1), 121–135 (2000)