CHAPTER

Lossless Image Compression

16

Lina J. Karam

Arizona State University

16.1 INTRODUCTION

The goal of lossless image compression is to represent an image signal with the smallest possible number of bits without loss of *any* information, thereby speeding up transmission and minimizing storage requirements. The number of bits representing the signal is typically expressed as an average bit rate (average number of bits per sample for still images, and average number of bits per second for video). The goal of lossy compression is to achieve the best possible fidelity given an available communication or storage bit rate capacity or to minimize the number of bits representing the image signal subject to some allowable loss of information. In this way, a much greater reduction in bit rate can be attained as compared to lossless compression, which is necessary for enabling many real-time applications involving the handling and transmission of audiovisual information. The function of *compression* is often referred to as *coding*, for short.

Coding techniques are crucial for the effective transmission or storage of data-intensive visual information. In fact, a single uncompressed color image or video frame with a medium resolution of 500×500 pixels would require 100 seconds for transmission over an Integrated Services Digital Network (ISDN) link having a capacity of 64,000 bits per second (64 Kbps). The resulting delay is intolerably large considering that a delay as small as 1 to 2 seconds is needed to conduct an interactive "slide show," and a much smaller delay (on the order of 0.1 second) is required for video transmission or playback. Although a CD-ROM device has a storage capacity of a few gigabits, its average data-read throughput is only a few Megabits per second (about 1.2 Mbps to 1.5 Mbps for the common $1\times$ read speed CLV CDs). As a result, compression is essential for the storage and real-time transmission of digital audiovisual information, where large amounts of data must be handled by devices having a limited bandwidth and storage capacity.

Lossless compression is possible because, in general, there is significant redundancy present in image signals. This redundancy is proportional to the amount of correlation among the image data samples. For example, in a natural still image, there is

usually a high degree of spatial correlation among neighboring image samples. Also, for video, there is additional temporal correlation among samples in successive video frames. In color images and multispectral imagery (Chapter 8), there is correlation, known as spectral correlation, between the image samples in the different spectral components.

In lossless coding, the decoded image data should be identical both quantitatively (numerically) and qualitatively (visually) to the original encoded image. Although this requirement preserves exactly the accuracy of representation, it often severely limits the amount of compression that can be achieved to a compression factor of two or three. In order to achieve higher compression factors, perceptually lossless coding methods attempt to remove redundant as well as perceptually irrelevant information; these methods require that the encoded and decoded images be only visually, and not necessarily numerically, identical. In this case, some loss of information is allowed as long as the recovered image is perceived to be identical to the original one.

Although a higher reduction in bit rate can be achieved with lossy compression, there exist several applications that require lossless coding, such as the compression of digital medical imagery and facsimile transmissions of bitonal images. These applications triggered the development of several standards for lossless compression, including the lossless JPEG standard (Section 16.4), facsimile compression standards, and the JBIG and JBIG2 compression standards. More recently, the JPEG2000 standard was developed as a unified compression standard that integrates both lossy and lossless compression into one system for different types of images including continuous-tone, bilevel, text, and compound imagery. Furthermore, lossy coding schemes make use of lossless coding components to minimize the redundancy in the signal being compressed.

This chapter introduces the basics of lossless image coding and presents classical as well as some more recently developed lossless compression methods. This chapter is organized as follows. Section 16.2 introduces basic concepts in lossless image coding. Section 16.3 reviews concepts from information theory and presents classical lossless compression schemes including Huffman, Arithmetic, Lempel-Ziv-Welch (LZW), Elias, and Exp-Golomb codes. Standards for lossless compression are presented in Section 16.4. Section 16.5 introduces more recently developed lossless compression schemes and presents the basics of perceptually lossless image coding.

16.2 BASICS OF LOSSLESS IMAGE CODING

The block diagram of a lossless coding system is shown in Fig. 16.1. The encoder (Fig. 16.1(a)) takes as input an image and generates as output a compressed bitstream. The decoder (Fig. 16.1(b)) takes as input the compressed bitstream and recovers the original uncompressed image. In general, the encoder and decoder can each be each viewed as consisting of three main stages. In this section, only the main elements of the encoder will be discussed since the decoder performs the inverse operations of the encoder. As

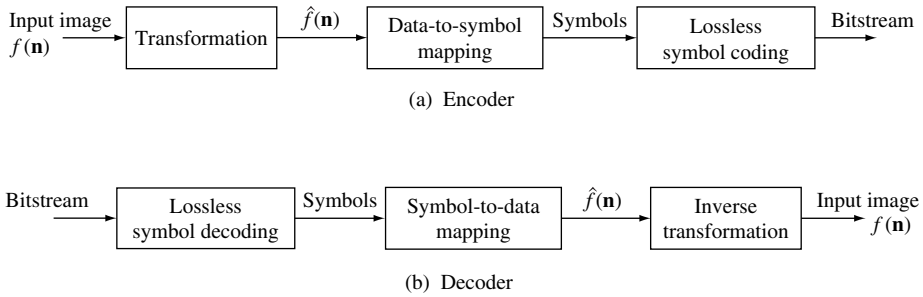

FIGURE 16.1

General lossless coding system.

shown in Fig. 16.1(a), the operations of a lossless image encoder can be grouped into three stages:

1. **Transformation:** This stage applies a reversible (one-to-one) transformation to the input image data. The purpose of this stage is to convert the input image data $f(\mathbf{n})$ into a form $\hat{f}(\mathbf{n})$ that can be compressed more efficiently. For this purpose, the selected transformation can aid in reducing the data correlation (interdependency, redundancy), alter the data statistical distribution, and/or pack a large amount of information into few data samples or subband regions. Typical transformations include differential/predictive mapping, unitary transforms such as the discrete cosine transform (DCT), subband decompositions such as wavelet transforms, and color space conversions such as conversion from the highly correlated RGB representation to the less correlated luminance-chrominance representation. A combination of the above transforms can be used at this stage. For example, an RGB color image can be transformed to its luminance-chrominance representation followed by DCT or subband decomposition followed by predictive/differential mapping. In some applications (e.g., low power), it might be desirable to operate directly on the original data without incurring the additional cost of applying a transformation; in this case, the transformation could be set to the identity mapping.

2. **Data-to-symbol mapping:** This stage converts the image data $\hat{f}(\mathbf{n})$ into entities called *symbols* that can be efficiently coded by the final stage. The conversion into symbols can be done through partitioning and/or run-length coding (RLC), for example.

   The image data can be partitioned into blocks by grouping neighboring data samples together; in this case, each data block is a symbol. Grouping several data units together allows the exploitation of any correlation that might be present between the image data, and may result in higher compression ratios at the expense of increasing the coding complexity. On the other hand, each separate data unit can be taken to be a symbol without any further grouping or partitioning.

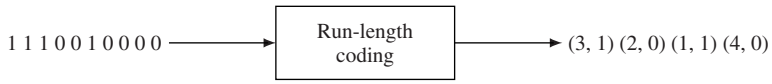

FIGURE 16.2

Illustration of run-length coding for a binary input sequence.

The basic idea behind RLC is to map a sequence of numbers into a sequence of symbol pairs (*run*, *value*), where *value* is the value of a data sample in the input data sequence and *run* or *runlength* is the number of times that data sample is contiguously repeated. In this case, each pair (*run*, *value*) is a symbol. An example illustrating RLC for a binary sequence is shown in Fig. 16.2. Different implementations might use a slightly different format. For example, if the input data sequence has long runs of zeros, some coders such as the JPEG standard (Chapter 17) use *value* to code only the value of the nonzero data samples and *run* to code the number of zeros preceding each nonzero data sample.

Appropriate mapping of the input data into symbols is very important for optimizing the coding. For example, grouping data points into small localized sets, where each set is coded separately as a symbol, allows the coding scheme to adapt to the changing local characteristics of the (transformed) image data. The appropriate data-to-symbol mapping depends on the considered application and the limitations in hardware/software complexity.

3. **Lossless symbol coding:** This stage generates a binary bitstream by assigning binary codewords to the input symbols. Lossless symbol coding is commonly referred to as noiseless coding or just lossless coding since this stage is where the actual lossless coding into the final compressed bitstream is performed. The first two stages can be regarded as preprocessing stages for mapping the data into a form that can be more efficiently coded by this lossless coding stage.

   Lossless compression is usually achieved by using variable-length codewords, where the shorter codewords are assigned to the symbols that occur more frequently. This variable-length codeword assignment is known as *variable-length coding (VLC)* and also as *entropy coding*. Entropy coders, such as Huffman and arithmetic coders, attempt to minimize the average bit rate (average number of bits per symbol) needed to represent a sequence of symbols, based on the probability of symbol occurrence. Universal codes, such as Elias codes [1] and Exp-Golomb codes [2, 3], are variable-length codes that can encode positive integer values into variable-length binary codewords without knowledge of the true probability distribution of the occuring integer values. Entropy coding will be discussed in more detail in Section 16.3. An alternative way to achieve compression is to code *variable-length strings* of symbols using fixed-length binary codewords. This is the basic strategy behind dictionary (Lempel-Ziv) codes, which are also described in Section 16.3.

   The generated lossless code (bitstream) should be uniquely decodable; i.e., the bitstream can be decoded without ambiguity resulting in only one unique sequence

of symbols (the original one). For variable-length codes, unique decodability is achieved by imposing a prefix condition which states that no codeword can be a prefix of another codeword. Codes that satisfy the prefix condition are called *prefix codes* or *instantaneously decodable codes*, and they include Huffman, arithmetic, Elias, and Exp-Golomb codes. Binary prefix codes can be represented as a binary tree, and are also called *tree-structured codes*. For dictionary codes, unique decodability can be easily achieved since the generated code words are fixed length.

Selecting which lossless coding method to use depends on the application and usually involves a tradeoff between several factors including the implementation hardware or software, the allowable coding delay, and the required compression level. Some of the factors that need to be considered when choosing or devising a lossless compression scheme are listed below.

1. **Compression efficiency:** Compression efficiency is usually given in the form of a compression ratio C_R,

$$C_R = \frac{\text{Total size in bits of original input image}}{\text{Total size in bits of compressed bitstream}} = \frac{\text{Total size in bits of encoder input}}{\text{Total size in bits of encoder output}}, \tag{16.1}$$

which compares the size of the original input image data with the size of the generated compressed bitstream. Compression efficiency is also commonly expressed as an average bit rate B in bits per pixel, or *bpp* for short,

$$B = \frac{\text{Total size in bits of compressed bitstream}}{\text{Total number of pixels in original input image}} = \frac{\text{Total size in bits of encoder output}}{\text{Total size in pixels of encoder input}}. \tag{16.2}$$

As discussed in Section 16.3, for lossless coding, the achievable compression efficiency is bounded by the *entropy* of the finite set of symbols generated as the output of Stage 2, assuming these symbols are each coded separately, on a one-by-one basis, by Stage 3.

2. **Coding delay:** The coding delay can be defined as the minimum time required to both encode and decode an input data sample. The coding delay increases with the total number of required arithmetic operations. It also usually increases with an increase in memory requirements since memory usage usually leads to communication delays. Minimizing the coding delay is especially important for real-time applications.

3. **Implementation complexity:** Implementation complexity is measured in terms of the total number of required arithmetic operations and in terms of the memory requirements. Alternatively, implementation complexity can be measured in terms of the required number of arithmetic operations per second and the memory requirements for achieving a given coding delay or real-time performance. For applications that put a limit on power consumption, the implementation complexity would also include a measure of the level of power consumption. Higher

compression efficiency can usually be achieved by increasing the implementation complexity, which would in turn lead to an increase in the coding delay. In practice, it is desirable to optimize the compression efficiency while keeping the implementation requirements as simple as possible. For some applications such as database browsing and retrieval, only a low decoding complexity is needed since the encoding is not performed as frequently as the decoding.

4. **Robustness:** For applications that require transmission of the compressed bitstream in error-prone environments, robustness of the coding method to transmission errors becomes an important consideration.
5. **Scalability:** Scalable encoders generate a layered bitstream embedding a hierarchical representation of the input image data. In this way, the input data can be recovered at different resolutions in a hierarchical manner (scalability in resolution), and the bit rate can be varied depending on the available resources using the same encoded bitstream (scalability in bit rate; the encoding does not have to be repeated to generate the different bit rates). The JPEG 2000 standard (Chapter 17) is an example of a scalable image coder that generates an embedded bitstream and that supports scalability in quality, resolution, spatial location, and image components [4, Chapter 9].

16.3 LOSSLESS SYMBOL CODING

As mentioned in Section 16.2, lossless symbol coding is commonly referred to as lossless coding or lossless compression. The popular lossless symbol coding schemes fall into one of the following main categories:

- Statistical schemes (Huffman, arithmetic): these schemes require knowledge of the source symbol probability distribution; shorter code words are assigned to the symbols with higher probability of occurrence (VLC); a statistical source model (also called probability model) gives the symbol probabilities; the statistical source model can be fixed, in which case the symbol probabilities are fixed, or adaptive, in which case the symbol probabilities are calculated adaptively; sophisticated source models can provide more accurate modeling of the source statistics and, thus, achieve higher compression at the expense of an increase in complexity.
- Dictionary-based schemes (Lempel-Ziv): these schemes do not require a priori knowledge of the source symbol probability distribution; they dynamically construct encoding and decoding tables (called dictionaries) of variable-length symbol strings as they occur in the input data; as the encoding table is constructed, fixed-length binary codewords are generated by indexing into the encoding table.
- Structured universal coding schemes (Elias codes, Exponential-Golomb codes): these schemes generate variable-length code words with a regular structure; they operate on positive integers, which necessitates first mapping the source symbols

to positive integers; no a priori knowledge of the true probability distribution of the integer values is required, but these codes are constructed under the assumption of a monotonically decreasing distribution, which implies that smaller integer values are more probable than larger integer values.

The aforementioned statistical, dictionary-based, and structured universal coders attempt to minimize the average bit rate without incurring any loss in fidelity. Such lossless coding schemes are also referred to as *entropy coding* schemes. The field of information theory gives lower bounds on the achievable bit rates. This section presents the popular classical lossless symbol coding schemes, including Huffman, Arithmetic and Lempel-Ziv coding, in addition to the structured Elias and Exp-Golomb universal codes. In order to gain an insight into how the bit rate minimization is done by these different lossless coding schemes, some important basic concepts from information theory are reviewed first.

16.3.1 Basic Concepts from Information Theory

Information theory makes heavy use of probability theory since information is related to the degree of unpredictability and randomness in the generated messages. In here, the generated messages are the symbols output by Stage 2 (Section 16.2).

An information source is characterized by the set of symbols S it is capable of generating and by the probability of occurrence of these symbols. For the considered lossless image coding application, the information source is a discrete-time, discrete-amplitude source with a finite set of unique symbols; i.e., S consists of a finite number of symbols and is commonly called the *source alphabet*.

Let S consist of N symbols:

$$S = \{s_0, s_1, \ldots, s_{N-1}\}. \tag{16.3}$$

Then the information source outputs a sequence of symbols $\{x_1, x_2, x_3, \ldots, x_i, \ldots\}$ drawn from the set of symbols S, where x_1 is the first output source sample, x_2 is the second output sample, and x_i is the ith output sample from S. At any given time (given by the output sequence index), the probability that the source outputs symbol s_k is $p_k = P(s_k)$, $0 \leq k \leq N-1$. Note that $\sum_{k=0}^{N-1} p_k = 1$ since it is certain that the source outputs only symbols from its alphabet S. The source is said to be *stationary* if its statistics (set of probabilities) do not change with time.

The information associated with a symbol s_k $(0 \leq k \leq N-1)$, also called *self-information*, is defined as

$$I_{s_k} = \log_2(1/p_k) \text{ (bits)} = -\log_2(p_k) \text{ (bits)}. \tag{16.4}$$

From (16.4), it can be seen that $I_k = 0$ if $p_k = 1$ (certain event) and $I_k \to \infty$ if $p_k = 0$ (impossible event). Also, I_k is large when p_k is small (unlikely symbols), as expected.

The information content of the source can be measured using the source *entropy* $H(S)$, which is a measure of the average amount of information per symbol. The source entropy $H(S)$, also known as *first-order entropy* or *marginal entropy*, is defined as the

expected value of the self information and is given by

$$H(S) = E\{I_k\} = -\sum_{k=0}^{N-1} p_k \log_2(p_k) \text{ (bits per symbol).} \tag{16.5}$$

Note that $H(S)$ is maximal if the symbols in S are equiprobable (flat probability distribution), in which case $H(S) = \log_2(N)$ bits per symbol. A skewed probability distribution results in a smaller source entropy.

In the case of memoryless coding, each source symbol is coded separately. For a given lossless code C, let l_k denote the length (number of bits) of the code word assigned to code symbol s_k $(0 \leq k \leq N-1)$. Then, the resulting average bit rate B_C corresponding to code C is

$$B_C = \sum_{k=0}^{N-1} p_k l_k \text{ (bits per symbol).} \tag{16.6}$$

For any uniquely decodable lossless code C, the entropy $H(S)$ is a lower bound on the average bit rate B_C [5]:

$$B_C \geq H(S). \tag{16.7}$$

So, $H(S)$ puts a limit on the achievable average bit rate given that each symbol is coded separately in a memoryless fashion.

In addition, a uniquely decodable prefix code C can always be constructed (e.g., Huffman coding - Section 16.3.3) such that

$$H(S) \leq B_C \leq H(S) + 1. \tag{16.8}$$

An important result which can be used in constructing prefix codes is the Kraft inequality:

$$\sum_{k=1}^{N} 2^{-l_k} \leq 1. \tag{16.9}$$

Every uniquely decodable code has codewords with lengths satisfying the Kraft inequality (16.9), and prefix codes can be constructed with any set of lengths satisfying (16.9) [6].

Higher compression can be achieved by coding a block (subsequence, vector) of M successive symbols jointly. The coding can be done as in the case of memoryless coding by regarding each block of M symbols as one compound symbol $S^{(M)}$ drawn from the alphabet:

$$S^{(M)} = \underbrace{S \times S \times \ldots \times S}_{M \text{ times}}, \tag{16.10}$$

where $\times$ in (16.10) denotes a cartesian product, and the superscript (M) denotes the size of each compound block of symbols. Therefore, $S^{(M)}$ is the set of all possible compound

symbols of the form $[x_1, x_2, \ldots, x_M]$, where $x_i \in S$, $1 \le i \le M$. Since S consists of N symbols, $S^{(M)}$ will contain $L = N^M$ compound symbols:

$$S^{(M)} = \{s_0^{(M)}, s_1^{(M)}, \ldots, s_{L-1}^{(M)}\}; \quad L = N^M. \tag{16.11}$$

The previous results and definitions directly generalize by replacing S with $S^{(M)}$ and replacing the symbol probabilities $p_k = P(s_k)$ $(0 \le k \le N-1)$ with the joint probabilities (compound symbol probabilities) $p_k^{(M)} = P(s_k^{(M)})$ $(0 \le k \le L-1)$. So, the entropy of the set $S^{(M)}$, which is the set of all compound symbols $s_k^{(M)}$ $(0 \le k \le L-1)$ is given by

$$H(S^{(M)}) = -\sum_{k=0}^{L-1} p_k^{(M)} \log_2 (p_k^{(M)}). \tag{16.12}$$

$H(S^{(M)})$ is also called the *Mth-order entropy* of S. If S corresponds to a stationary source (i.e., symbol probabilities do not change over time), $H(S^{(M)})$ is related to the source entropy $H(S)$ as follows [5]:

$$\frac{H(S^{(M)})}{M} \le H(S), \tag{16.13}$$

with equality if and only if the symbols in S are statistically independent (memoryless source). The quantity

$$\lim_{M \to \infty} \frac{H(S^{(M)})}{M} \tag{16.14}$$

is called the *entropy rate* of the source S and gives the average information per output symbol drawn from S. For a stationary source, the limit in (16.14) always exists. Also, from (16.13), the entropy rate is equal to the source entropy for the case of a memoryless source.

As before each output (compound) symbol can be coded separately. For a given lossless code C, if $l_k^{(M)}$ is the length of the codeword assigned to code symbol $s_k^{(M)}$ $(0 \le k \le L-1)$, the resulting average bit rate $B_C^{(M)}$ in code bits per *compound* symbol is

$$B_C^{(M)} = \sum_{k=0}^{L-1} p_k^{(M)} l_k^{(M)} \text{ (bits per } \textit{compound} \text{ symbol)}. \tag{16.15}$$

Also, as before, a prefix code C can be constructed such that

$$H(S^{(M)}) \le B_C^{(M)} \le H(S^{(M)}) + 1, \tag{16.16}$$

where $B_C^{(M)}$ is the resulting average bit rate per *compound* symbol. The desired average bit rate B_C in bits per *source* symbol is equal to $B_C^{(M)}/M$. So, dividing the terms in (16.16) by M, we obtain

$$\frac{H(S^{(M)})}{M} \le B_C \le \frac{H(S^{(M)})}{M} + \frac{1}{M}. \tag{16.17}$$

From (16.17), it follows that, by jointly coding very large blocks of source symbols (M very large), a source code C can be found with an average bit rate B_C approaching monotonically the entropy rate of the source as M goes to infinity. For a memoryless source, (16.17) becomes

$$H(S) \leq B_C \leq H(S) + \frac{1}{M}, \tag{16.18}$$

where $B_C = B_C^{(M)}/M$.

From the discussion above, the statistics of the considered source (given by the symbol probabilities) need to be known in order to compute the lower bounds on the achievable bit rate. In addition, statistical-based entropy coding schemes, such as Huffman (Section 16.3.3) and Arithmetic (Section 16.3.4) coding, require that the source statistics be known or be estimated accurately. In practice, the source statistics can be estimated from the histogram of a set of sample source symbols. For a nonstationary source, the symbol probabilities need to be estimated adaptively since the source statistics change over time.

In the case of block coding where M successive symbols are coded jointly as a compound symbol $s^{(M)}$, the joint probabilities $P(s^{(M)})$ need to be estimated. These joint probabilities are very difficult to calculate and the computational complexity grows exponentially with the block size M, except when the source is memoryless in which case the compound source entropy $H(S^{(M)}) = MH(S)$ based on (16.13).

However, in practice, the memoryless source model is not generally appropriate when compressing digital imagery as various dependencies exist between the image data elements even after the transformation and mapping stages (Fig. 16.1). This is due to the fact that the existing practical image transforms, such as the DCT (Chapter 17) and the discrete wavelet transform (DWT) (Chapter 17), reduce the dependencies but do not eliminate them as they cannot totally decorrelate the real-world image data. Therefore, most state-of-the-art image compression coders model these dependencies by estimating the statistics of a source with memory. The statistics of the source with memory are usually estimated by computing conditional probabilities which provide context-based modeling. Context-based modeling is a key component of context-based entropy coding which has been widely adopted in image compression schemes and is at the core of the JPEG2000 image compression standard. Context-based entropy coding is discussed below in Section 16.3.2.

16.3.2 Context-Based Entropy Coding

Context-based entropy coding is a key component of the JPEG-LS standard [7] and the more recent JPEG2000 standard [4, 8] and has also been adopted as the final entropy coding stage of various other state-of-the-art image and video compression schemes [9, 10]. The contexts provide a model for estimating the probability of each possible message to be coded, and the entropy coder translates the estimated probabilities into bits.

For sources with memory, the dependencies among adjacent symbols can be captured by either estimating joint probabilities or conditional probabilities. As discussed earlier, joint probabilities are difficult to calculate due to increased computational complexity,

memory requirements, and coding delay. These issues can be resolved in practice by estimating conditional probabilities as implied by the following entropy chain rule:

$$H(S_1, S_2, \ldots, S_M) = \sum_{i=1}^{M} H(S_{M-i} | S^{i-1}), \tag{16.19}$$

where $S^{i-1} = S_1, \ldots, S_{i-1}$, and where $S_i, i = 1, \ldots, M$ are random variables that can take on realizations drawn from a finite set such as the set S of (16.3). In this way, the entropy coding can be performed incrementally one symbol at a time.

One way of calculating the conditional probabilities is to estimate those based on a finite number of previous symbols in the data stream. A more common way of estimating the conditional probabilities is to assign each possible realization s^{i-1} of S^{i-1} to some conditioning class; in this latter case, the conditional probability $p(S_i|s^{i-1}) \approx p(S_i|C(s^{i-1}))$, where $C(\bullet)$ is the function that maps each possible realization s^{i-1} to a conditioning context.

Therefore, context-based entropy coding can be separated into 2 parts: a context-based modeler and a coder. At each time instance i, the modeler tries to estimate $p(S_i|C(s^{i-1}))$, the probability of the next symbol to be coded based on the observed context. Because the contexts are formed by the already coded symbols and the entropy coding is lossless, the same context is also available at the decoder, so no side information needs to be transmitted. Given the estimated $\hat{p}(S_i|C(s^{i-1}))$, an ideal entropy coder places $-\log_2(\hat{p}(S_i = s_k|C(s^{i-1})))$ bits onto its output stream if $S_i = s_k$ actually occurs.

The estimated probabilities $\hat{p}(S_i|C(s^{i-1}))$ can be precomputed through training and made available to both the encoder and decoder before the actual coding starts. Alternatively, these estimates can be adaptively computed and updated on the fly based on the past-coded symbols. Practical applications generally require adaptive, online estimation of these probabilities either because sufficient prior statistical knowledge is not available or because these statistics are time-varying. One very natural way of estimating the probabilities online is to count the number of times each symbol has occurred under a certain context. The estimates at the encoder are updated with each encoded symbol, and the estimates at the decoder are updated with each decoded symbol. This universal modeling approach requires no a priori knowledge about the data to be coded, and the coding can be completed in only one pass.

16.3.3 Huffman Coding

In [11], Huffman presented a simple technique for constructing prefix codes which results in an average bit rate satisfying (16.8) when the source symbols are coded separately, or (16.17) in the case of joint M-symbol vector coding. A tighter upper bound on the resulting average bit rate is derived in [6].

The Huffman coding algorithm is based on the following optimality conditions for a prefix code [11]: 1) If $P(s_k) > P(s_j)$ (symbol s_k more probable than symbol s_j, $k \neq j$), then $l_k \leq l_j$, where l_k and l_j are the lengths of the codewords assigned to code symbols s_k and s_j, respectively; 2) If the symbols are listed in the order of decreasing probabilities,

the last two symbols in the ordered list are assigned codewords that have the same length and are alike except for their final bit.

Given a source with alphabet S consisting of N symbols s_k with probabilities $p_k = P(s_k)$ $(0 \leq k \leq (N-1))$, a Huffman code corresponding to source S can be constructed by iteratively constructing a binary tree as follows:

1. Arrange the symbols of S such that the probabilities p_k are in decreasing order; i.e.,

$$p_0 \geq p_1 \geq \ldots \geq p_{(N-1)} \tag{16.20}$$

   and consider the ordered symbols s_k, $0 \leq k \leq (N-1)$ as the leaf nodes of a tree. Let T be the set of the leaf nodes corresponding to the ordered symbols of S.

2. Take the two nodes in T with the smallest probabilities and merge them into a new node whose probability is the sum of the probabilities of these two nodes. For the tree construction, make the new resulting node the "parent" of the two least probable nodes of T by connecting the new node to each of the two least probable nodes. Each connection between two nodes form a "branch" of the tree; so two new branches are generated. Assign a value of 1 to one branch and 0 to the other branch.

3. Update T by replacing the two least probable nodes in T with their "parent" node and reorder the nodes (with their subtrees) if needed. If T contains more than one node, repeat from Step 2; otherwise the last node in T is the "root" node of the tree.

4. The codeword of a symbol $s_k \in S$ $(0 \leq k \leq (N-1))$ can be obtained by traversing the linked path of the tree from the root node to the leaf node corresponding to s_k $(0 \leq k \leq (N-1))$ while reading sequentially the bit values assigned to the tree branches of the traversed path.

The Huffman code construction procedure is illustrated by the example shown in Fig. 16.3 for the source alphabet $S = \{s_0, s_1, s_2, s_3\}$ with symbol probabilities as given in Table 16.1. The resulting symbol codewords are listed in the 3rd column of Table 16.1. For this example, the source entropy is $H(S) = 1.84644$ and the resulting average bit rate is $B_H = \sum_{k=0}^{3} p_k l_k = 1.9$ (bits per symbol), where l_k is the length of the codeword assigned

TABLE 16.1 Example of Huffman code assignment.

Source symbol s_k	Probability p_k	Assigned codeword
s_0	0.1	111
s_1	0.3	10
s_2	0.4	0
s_3	0.2	110

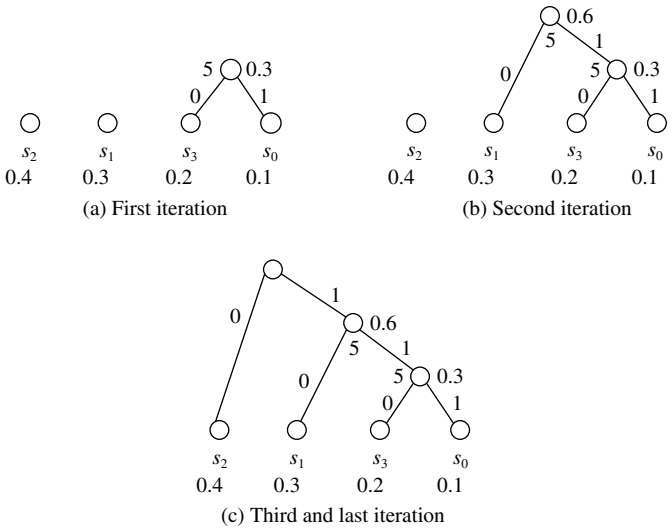

FIGURE 16.3

Example of Huffman code construction for the source alphabet of Table 16.1.

to symbol s_k of S. The symbol codewords are usually stored in a symbol-to-codeword mapping table that is made available to both the encoder and the decoder.

If the symbol probabilities can be accurately computed, the above Huffman coding procedure is optimal in the sense that it results in the minimal average bit rate among all uniquely decodable codes assuming memoryless coding. Note that, for a given source S, more than one Huffman code is possible but they are all optimal in the above sense. In fact another optimal Huffman code can be obtained by simply taking the complement of the resulting binary codewords.

As a result of memoryless coding, the resulting average bit rate is within one bit of the source entropy since integer-length codewords are assigned to each symbol separately. The described Huffman coding procedure can be directly applied to code a group of M symbols jointly by replacing S with $S^{(M)}$ of (16.10). In this case, higher compression can be achieved (Section 16.3.1), but at the expense of an increase in memory and complexity since the alphabet becomes much larger and joint probabilities need to be computed.

While encoding can be simply done by using the symbol-to-codeword mapping table, the realization of the decoding operation is more involved. One way of decoding the bitstream generated by a Huffman code is to first reconstruct the binary tree from the symbol-to-codeword mapping table. Then, as the bitstream is read one bit at a time, the tree is traversed starting at the root until a leaf node is reached. The symbol corresponding to the attained leaf node is then output by the decoder. Restarting at the root of the tree, the above tree traversal step is repeated until all the bitstream is decoded. This decoding method produces a variable symbol rate at the decoder output since the codewords vary in length.

Another way to perform the decoding is to construct a lookup table from the symbol-to-codeword mapping table. The constructed lookup table has $2^{l_{\max}}$ entries, where $l_{\max}$ is the length of the longest codeword. The binary codewords are used to index into the lookup table. The lookup table can be constructed as follows. Let l_k be the length of the codeword corresponding to symbol s_k. For each symbol s_k in the symbol-to-codeword mapping table, place the pair of values (s_k, l_k) in all the table entries, for which the l_k leftmost address bits are equal to the codeword assigned to s_k. Thus there will be $2^{(l_{\max}-l_k)}$ entries corresponding to symbol s_k. For decoding, $l_{\max}$ bits are read from the bitstream. These $l_{\max}$ bits are used to index into the lookup table to obtain the decoded symbol s_k, which is then output by the decoder, and the corresponding codeword length l_k. Then the next table index is formed by discarding the first l_k bits of the current index and appending to the right the next l_k bits that are read from the bitstream. This process is repeated until all the bitstream is decoded. This approach results in a relatively fast decoding and in a fixed output symbol rate. However, the memory size and complexity grows exponentially with $l_{\max}$, which can be very large.

In order to limit the complexity, procedures to construct *constrained-length Huffman codes* have been developed [12]. *Constrained-length Huffman codes* are Huffman codes designed while limiting the maximum allowable codeword length to a specified value $l_{\max}$. The shortened Huffman codes result in a higher average bit rate compared to the unconstrained-length Huffman code.

Since the symbols with the lowest probabilities result in the longest codewords, one way of constructing shortened Huffman codes is to group the low probability symbols into a compound symbol. The low probability symbols are taken to be the symbols in S with a probability $\leq 2^{-l_{\max}}$. The probability of the compound symbol is the sum of the probabilities of the individual low-probability symbols. Then the original Huffman coding procedure is applied to an input set of symbols formed by taking the original set of symbols and replacing the low probability symbols with one compound symbol s_c. When one of the low probability symbols is generated by the source, it is encoded using the codeword corresponding to s_c followed by a second fixed-length binary code word corresponding to that particular symbol. The other "high probability" symbols are encoded as usual by using the Huffman symbol-to-codeword mapping table.

In order to avoid having to send an additional codeword for the low probability symbols, an alternative approach is to use the original unconstrained Huffman code design procedure on the original set of symbols S with the probabilities of the low probability symbols changed to be equal to $2^{-l_{\max}}$. Other methods [12] involve solving a constrained optimization problem to find the optimal codeword lengths l_k $(0 \leq k \leq N-1)$ that minimize the average bit rate subject to the constraints $1 \leq l_k \leq l_{\max}$ $(0 \leq k \leq N-1)$. Once the optimal codeword lengths have been found, a prefix code can be constructed using the Kraft inequality (16.9). In this case the codeword of length l_k corresponding to s_k is given by the l_k bits to the right of the binary point in the binary representation of the fraction $\sum_{1 \leq i \leq k-1} 2^{-l_i}$.

The discussion above assumes that the source statistics are described by a *fixed* (nonvarying) set of source symbol probabilities. As a result, only one fixed set of codewords need to be computed and supplied once to the encoder/decoder. This fixed model fails

if the source statistics vary, since the performance of Huffman coding depends on how accurately the source statistics are modeled. For example, images can contain different data types, such as text and picture data, with different statistical characteristics. Adaptive Huffman coding changes the codeword set to match the locally estimated source statistics. As the source statistics change, the code changes, remaining optimal for the current estimate of source symbol probabilities. One simple way for adaptively estimating the symbol probabilities is to maintain a count of the number of occurrences of each symbol [6]. The Huffman code can be dynamically changed by precomputing offline different codes corresponding to different source statistics. The precomputed codes are then stored in symbol-to-codeword mapping tables that are made available to the encoder and decoder. The code is changed by dynamically choosing a symbol-to-codeword mapping table from the available tables based on the frequencies of the symbols that occurred so far. However, in addition to storage and the run-time overhead incurred for selecting a coding table, this approach requires a priori knowledge of the possible source statistics in order to predesign the codes. Another approach is to dynamically redesign the Huffman code while encoding based on the local probability estimates computed by the provided source model. This model is also available at the decoder, allowing it to dynamically alter its decoding tree or decoding table in synchrony with the encoder. Implementation details of adaptive Huffman coding algorithms can be found in [6, 13].

In the case of context-based entropy coding, the described procedures are unchanged except that now the symbol probabilities $P(s_k)$ are replaced with the symbol conditional probabilities $P(s_k|\text{Context})$ where the context is determined from previously occuring neighboring symbols, as discussed in Section 16.3.2.

16.3.4 Arithmetic Coding

As indicated in Section 16.3.3, the main drawback of Huffman coding is that it assigns an integer-length codeword to each symbol separately. As a result the bit rate cannot be less than one bit per symbol unless the symbols are coded jointly. However, joint symbol coding, which codes a block of symbols jointly as one compound symbol, results in delay and in an increased complexity in terms of source modeling, computation, and memory. Another drawback of Huffman coding is that the realization and the structure of the encoding and decoding algorithms depend on the source statistical model. It follows that any change in the source statistics would necessitate redesigning the Huffman codes and changing the encoding and decoding trees, which can render adaptive coding more difficult.

Arithmetic coding is a lossless coding method which does not suffer from the aforementioned drawbacks and which tends to achieve a higher compression ratio than Huffman coding. However, Huffman coding can generally be realized with simpler software and hardware.

In arithmetic coding, each symbol does not need to be mapped into an integral number of bits. Thus, an average fractional bit rate (in bits per symbol) can be achieved without the need for blocking the symbols into compound symbols. In addition, arithmetic coding allows the source statistical model to be separate from the structure of

the encoding and decoding procedures; i.e., the source statistics can be changed without having to alter the computational steps in the encoding and decoding modules. This separation makes arithmetic coding more attractive than Huffman for adaptive coding.

The arithmetic coding technique is a practical extended version of Elias code and was initially developed by Pasco and Rissanen [14]. It was further developed by Rubin [15] to allow for incremental encoding and decoding with fixed-point computation. An overview of arithmetic coding is presented in [14] with *C* source code.

The basic idea behind arithmetic coding is to map the input sequence of symbols into one single codeword. Symbol blocking is not needed since the codeword can be determined and updated incrementally as each new symbol is input (symbol-by-symbol coding). At any time, the determined codeword uniquely represents all the past occurring symbols. Although the final codeword is represented using an integral number of bits, the resulting average number of bits per symbol is obtained by dividing the length of the codeword by the number of encoded symbols. For a sequence of M symbols, the resulting average bit rate satisfies (16.17) and, therefore, approaches the optimum (16.14) as the length M of the encoded sequence becomes very large.

In the actual arithmetic coding steps, the codeword is represented by a half-open subinterval $[L_c, H_c) \subset [0,1)$. The half-open subinterval gives the set of all codewords that can be used to encode the input symbol sequence, which consists of all past input symbols. So any real number within the subinterval $[L_c, H_c)$ can be assigned as the codeword representing all the past occurring symbols. The selected real codeword is then transmitted in binary form (fractional binary representation, where 0.1 represents 1/2, 0.01 represents 1/4, 0.11 represents 3/4, and so on). When a new symbol occurs, the current subinterval $[L_c, H_c)$ is updated by finding a new subinterval $[L_c^{'}, H_c^{'}) \subset [L_c, H_c)$ to represent the new change in the encoded sequence. The codeword subinterval is chosen and updated such that its length is equal to the probability of occurrence of the corresponding encoded input sequence. It follows that less probable events (given by the input symbol sequences) are represented with shorter intervals and, therefore, require longer codewords since more precision bits are required to represent the narrower subintervals. So the arithmetic encoding procedure constructs, in a hierarchical manner, a code subinterval which uniquely represents a sequence of successive symbols.

In analogy with Huffman where the root node of the tree represents all possible occurring symbols, the interval $[0,1)$ here represents all possible occurring sequences of symbols (all possible messages including single symbols). Also, considering the set of all possible M-symbol sequences having the same length M, the total interval [0,1) can be subdivided into nonoverlapping subintervals such that each M symbol sequence is represented uniquely by one and only one subinterval whose length is equal to its probability of occurrence.

Let S be the source alphabet consisting of N symbols $s_0, \ldots, s_{(N-1)}$. Let $p_k = P(s_k)$ be the probability of symbol s_k, $0 \leq k \leq (N-1)$. Since, initially, the input sequence will consist of the first occurring symbol ($M = 1$), arithmetic coding begins by subdividing the interval [0,1) into N nonoverlapping intervals, where each interval is assigned to a distinct symbol $s_k \in S$ and has a length equal to the symbol probability p_k. Let $[L_{s_k}, H_{s_k})$

TABLE 16.2 Example of code subinterval construction in arithmetic coding.

Source symbol s_k	Probability p_k	Symbol subinterval $[L_{s_k}, H_{s_k})$
s_0	0.1	[0, 0.1)
s_1	0.3	[0.1, 0.4)
s_2	0.4	[0.4, 0.8)
s_3	0.2	[0.8, 1)

denote the interval assigned to symbol s_k, where $p_k = H_{s_k} - L_{s_k}$. This assignment is illustrated in Table 16.2; the same source alphabet and source probabilities as in the example of Fig. 16.3 are used for comparison with Huffman. In practice, the subinterval limits L_{s_k} and H_{s_k} for symbol s_k can be directly computed from the available symbol probabilities and are equal to cumulative probabilities P_k as given below:

$$L_{s_k} = \sum_{i=0}^{k-1} p_k = P_{k-1}; \quad 0 \leq k \leq (N-1), \tag{16.21}$$

$$H_{s_k} = \sum_{i=0}^{k} p_k = P_k; \quad 0 \leq k \leq (N-1). \tag{16.22}$$

Let $[L_c, H_c)$ denote the code interval corresponding to the input sequence which consists of the symbols that occurred so far. Initially, $L_c = 0$ and $H_c = 1$; so the initial code interval is set to $[0, 1)$. Given an input sequence of symbols, the calculation of $[L_c, H_c)$ is performed based on the following encoding algorithm:

1. $L_c = 0$; $H_c = 1$.
2. Calculate code subinterval length,
$$\text{length} = H_c - L_c. \tag{16.23}$$
3. Get next input symbol s_k.
4. Update the code subinterval,
$$L_c = L_c + \text{length} \cdot L_{s_k},$$
$$H_c = L_c + \text{length} \cdot H_{s_k}. \tag{16.24}$$
5. Repeat from Step 2 until all the input sequence has been encoded.

As indicated before, any real number within the final interval $[L_c, H_c)$ can be used as a valid codeword for uniquely encoding the considered input sequence. The binary representation of the selected codeword is then transmitted. The above arithmetic encoding procedure is illustrated in Table 16.3 for encoding the sequence of symbols $s_1\, s_0\, s_2\, s_3\, s_3$. Another representation of the encoding process within the context of the considered

TABLE 16.3 Example of code subinterval construction in arithmetic coding.

Iteration # I	Encoded symbol s_k	Code subinterval $[L_c, H_c)$
1	s_1	[0.1, 0.4)
2	s_0	[0.1, 0.13)
3	s_2	[0.112, 0.124)
4	s_3	[0.1216, 0.124)
5	s_3	[0.12352, 0.124)

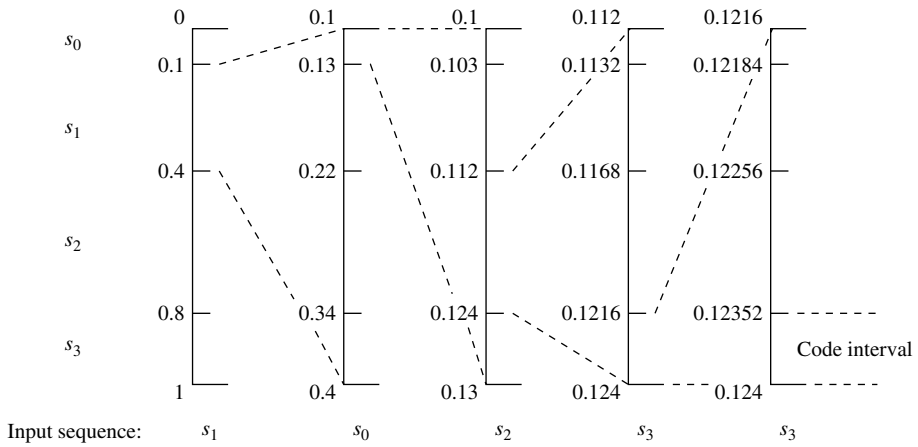

FIGURE 16.4
Arithmetic coding example.

example is shown in Fig. 16.4. Note that arithmetic coding can be viewed as remapping, at each iteration, the symbol subintervals $[L_{s_k}, H_{s_k})$ $(0 \leq k \leq (N-1))$ to the current code subinterval $[L_c, H_c)$. The mapping is done by rescaling the symbol subintervals to fit within $[L_c, H_c)$, while keeping them in the same relative positions. So when the next input symbol occurs, its symbol subinterval becomes the new code subinterval, and the process repeats until all input symbols are encoded.

In the arithmetic encoding procedure, the length of a code subinterval, *length* of (16.23), is always equal to the product of the probabilities of the individual symbols encoded so far, and it monotonically decreases at each iteration. As a result, the code interval shrinks at every iteration. So, longer sequences result in narrower code subintervals which would require the use of high-precision arithmetic. Also, a direct implementation of the presented arithmetic coding procedure produces an output only after all the input symbols have been encoded. Implementations that overcome these problems are

presented in [14, 15]. The basic idea is to begin outputting the leading bit of the result as soon as it can be determined (incremental encoding), and then to shift out this bit (which amounts to scaling the current code subinterval by 2). In order to illustrate how incremental encoding would be possible, consider the example in Table 16.3. At the second iteration, the leading part "0.1" can be output since it is not going to be changed by the future encoding steps. A simple test to check whether a leading part can be output is to compare the leading parts of L_c and H_c; the leading digits that are the same can then be output and they remain unchanged since the next code subinterval will become smaller. For fixed-point computations, overflow and underflow errors can be avoided by restricting the source alphabet size [12].

Given the value of the codeword, arithmetic decoding can be performed as follows:

1. $L_c = 0; H_c = 1$.
2. Calculate the code subinterval length,
$$\text{length} = H_c - L_c.$$
3. Find symbol subinterval $[L_{s_k}, H_{s_k})$ $(0 \le k \le N-1)$ such that
$$L_{s_k} \le \frac{\text{codeword} - L_c}{\text{length}} < H_{s_k}.$$
4. Output symbol s_k.
5. Update code subinterval,
$$L_c = L_c + \text{length} \cdot L_{s_k}$$
$$H_c = L_c + \text{length} \cdot H_{s_k}.$$
6. Repeat from Step 2 until last symbol is decoded.

In order to determine when to stop the decoding (i.e., which symbol is the last symbol), a special end-of-sequence symbol is usually added to the source alphabet S and is handled like the other symbols. In the case when fixed-length blocks of symbols are encoded, the decoder can simply keep a count of the number of decoded symbols and no end-of-sequence symbol is needed. As discussed before, incremental decoding can be achieved before all the codeword bits are output [14, 15].

Context-based arithmetic coding has been widely used as the final entropy coding stage in state-of-the-art image and video compression schemes, including the JPEG-LS and the JPEG2000 standards. The same procedures and discussions hold for context-based arithmetic coding with the symbol probabilities $P(s_k)$ replaced with conditional symbol probabilities $P(s_k|\text{Context})$ where the context is determined from previously occuring neighboring symbols, as discussed in Section 16.3.2. In JPEG2000, context-based adaptive binary arithmetic coding (CABAC) is used with 17 contexts to efficiently code the binary significance, sign, and magnitude refinement information (Chapter 17). Binary arithmetic coding work with a binary (two-symbol) source alphabet, can be

implemented more efficiently than nonbinary arithmetic coders, and has universal application as data symbols from any alphabet can be represented as a sequence of binary symbols [16].

16.3.5 Lempel-Ziv Coding

Huffman coding (Section 16.3.3) and arithmetic coding (Section 16.3.4) require a priori knowledge of the source symbol probabilities or of the source statistical model. In some cases, a sufficiently accurate source model is difficult to obtain, especially when several types of data (such as text, graphics, and natural pictures) are intermixed.

Universal coding schemes do not require a priori knowledge or explicit modeling of the source statistics. A popular lossless universal coding scheme is a dictionary-based coding method developed by Ziv and Lempel in 1977 [17] and known as Lempel-Ziv-77 (LZ77) coding. One year later, Ziv and Lempel presented an alternate dictionary-based method known as LZ78.

Dictionary-based coders dynamically build a coding table (called dictionary) of variable-length symbol strings as they occur in the input data. As the coding table is constructed, fixed-length binary codewords are assigned to the variable-length input symbol strings by indexing into the coding table. In Lempel-Ziv (LZ) coding, the decoder can also dynamically reconstruct the coding table and the input sequence as the code bits are received without any significant decoding delays. Although LZ codes do not explicitly make use of the source probability distribution, they asymptotically approach the source entropy rate for very long sequences [5]. Because of their adaptive nature, dictionary-based codes are ineffective for short input sequences since these codes initially result in a lot of bits being output. Short input sequences can thus result in data expansion instead of compression.

There are several variations of LZ coding. They mainly differ in how the dictionary is implemented, initialized, updated, and searched. Variants of the LZ77 algorithm have been used in many other applications and provided the basis for the development of many popular compression programs such as gzip, winzip, pkzip, and the public-domain Portable Network Graphics (PNG) image compression format.

One popular LZ coding algorithm is known as the LZW algorithm, a variant of the LZ78 algorithm developed by Welch [18]. This is the algorithm used for implementing the *compress* command in the UNIX operating system. The LZW procedure is also incorporated in the popular CompuServe GIF image format, where GIF stands for Graphics Interchange Format. However, the LZW compression procedure is patented, which decreased the popularity of compression programs and formats that make use of LZW. This was one main reason that triggered the development of the public-domain lossless PNG format.

Let S be the source alphabet consisting of N symbols s_k $(1 \leq k \leq N)$. The basic steps of the LZW algorithm can be stated as follows:

1. Initialize the first N entries of the dictionary with the individual source symbols of S, as shown below.

2. Parse the input sequence and find the longest input string of successive symbols w (including the first still unencoded symbol s in the sequence) that has a matching entry in the dictionary.
3. Encode w by outputing the index (address) of the matching entry as the codeword for w.
4. Add to the dictionary the string ws formed by concatenating w and the next input symbol s (following w).
5. Repeat from Step 2 for the remaining input symbols starting with the symbol s, until the entire input sequence is encoded.

Consider the source alphabet $S = \{s_1, s_2, s_3, s_4\}$. The encoding procedure is illustrated for the input sequence $s_1 s_2 s_1 s_2 s_3 s_2 s_1 s_2$. The constructed dictionary is shown in Table 16.4. The resulting code is given by the fixed-length binary representation of the following sequence of dictionary addresses: 1 2 5 3 6 2. The length of the generated binary codewords depends on the maximum allowed dictionary size. If the maximum dictionary size is M entries, the length of the codewords would be $\log_2(M)$ rounded to the next smallest integer.

The decoder constructs the same dictionary (Table 16.4) as the codewords are received. The basic decoding steps can be described as follows:

1. Start with the same initial dictionary as the encoder. Also, initialize w to be the empty string.
2. Get the next "codeword" and decode it by outputing the symbol string sm stored at address "codeword" in dictionary.
3. Add to the dictionary the string ws formed by concatenating the previous decoded string w (if any) and the first symbol s of the current decoded string.
4. Set $w = m$ and repeat from Step 2 until all the codewords are decoded.

TABLE 16.4 Dictionary constructed while encoding the sequence $s_1 s_2 s_1 s_2 s_3 s_2 s_1 s_2$, which is emitted by a source with alphabet $S = \{s_1, s_2, s_3, s_4\}$.

Address	Entry
1	s_1
2	s_2
3	s_3
4	s_4
5	$s_1\, s_2$
6	$s_2\, s_1$
7	$s_1\, s_2\, s_3$
8	$s_3\, s_2$
9	$s_2\, s_1\, s_2$

Address	Entry
1	s_1
2	s_2
3	s_3
⋮	⋮
N	s_N

Note that the constructed dictionary has a prefix property; i.e., every string w in the dictionary has its prefix string (formed by removing the last symbol of w) also in the dictionary. Since the strings added to the dictionary can become very long, the actual LZW implementation exploits the prefix property to render the dictionary construction more tractable. To add a string ws to the dictionary, the LZW implementation only stores the pair of values (c, s), where c is the address where the prefix string w is stored and s is the last symbol of the considered string ws. So the dictionary is represented as a linked list [5, 18].

16.3.6 Elias and Exponential-Golomb Codes

Similar to LZ coding, Elias codes [1] and Exponential-Golomb (Exp-Golomb) codes [2] are universal codes that do not require knowledge of the true source statistics. They belong to a class of structured codes that operate on the set of positive integers. Furthermore, these codes do not require having a finite set of values and can code arbitrary positive integers with an unknown upper bound. For these codes, each codeword can be constructed in a regular manner based on the value of the corresponding positive integer. This regular construction is formed based on the assumption that the probability distribution decreases monotonically with increasing integer values, i.e., smaller integer values are more probable than larger integer values. Signed integers can be coded by remapping them to positive integers. For example, an integer i can be mapped to the odd positive integer $2|i| - 1$ if it is negative, and to the even positive integer $2|i|$ if it is positive. Similarly, other one-to-one mapping can be formed to allow the coding of the entire integer set including zero. Noninteger source symbols can also be coded by first sorting them in the order of decreasing frequency of occurrence and then mapping the sorted set of symbols to the set of positive integers using a one-to-one (bijection) mapping, with smaller integer values being mapped to symbols with a higher frequency of occurrence. In this case, each positive integer value can be regarded as the index of the source symbol to which it is mapped, and can be referred to as the source symbol index or the codeword number or the codeword index.

Elias [1] described a set of codes including alpha (α), beta (β), gamma (γ), gamma′ (γ'), delta (δ), and omega (ω) codes. For a positive integer I, the alpha code $\alpha(I)$ is a unary code that represents the value I with $(I - 1)$ 0's followed by a 1. The last 1 acts as a terminating flag which is also referred to as a comma. For example, $\alpha(1) = 1, \alpha(2) = 01$, $\alpha(3) = 001$, $\alpha(4) = 0001$, and so forth. The beta code of I, $\beta(I)$, is simply the natural binary representation of I with the most significant bit set to 1. For example, $\beta(1) = 1$, $\beta(2) = 10, \beta(3) = 11$, and $\beta(4) = 100$. One drawback of the beta code is that the codewords are not decodable, since it is not a prefix code and it does not contain a way to determine the length of the codewords. Thus the beta code is usually combined with other codes to form other useful codes, such as Elias gamma, gamma′, delta, and omega codes, and Exp-Golomb codes. The Exp-Golomb codes have been incorporated within the H.264/AVC, also known as MPEG-4 Part 10, video coding standard to code different

parameters and data values, including types of macro blocks, indices of reference frames, motion vector differences, quantization parameters, patterns for coded blocks, and others. Details about these codes are given below.

16.3.6.1 Elias Gamma (γ) and Gamma′ (γ') Codes

The Elias γ and γ' codes are variants of each other with one code being a permutation of the other code. The γ' code is also commonly referred to as a γ code.

For a positive integer I, Elias γ' coding generates a binary codeword of the form

$$\gamma'(I) = [(L-1) \text{ zeros}][\beta(I)], \tag{16.25}$$

where $\beta(I)$ is the beta code of I which corresponds to the natural binary representation of I, and L is the length of (number of bits in) the binary codeword $\beta(I)$. L can be computed as $L = (\lfloor \log_2(I) \rfloor + 1)$, where $\lfloor . \rfloor$ denotes rounding to the nearest smaller integer value. For example, $\gamma'(1) = 1, \gamma'(2) = 010, \gamma'(3) = 011$, and $\gamma'(4) = 00100$. In other words, an Elias γ' code can be constructed for a positive integer I using the following procedure:

1. Find the natural binary representation, $\beta(I)$, of I.
2. Determine the total number of bits, L, in $\beta(I)$.
3. The codeword $\gamma'(I)$ is formed as $(L-1)$ zeros followed by $\beta(I)$.

Alternatively, the Elias γ' code can be constructed as the unary alpha code $\alpha(L)$, where L is the number of bits in $\beta(I)$, followed by the last $(L-1)$ bits of $\beta(I)$ (i.e., $\beta(I)$ with the ommission of the most significant bit 1).

An Elias γ' code can be decoded by reading and counting the leading 0 bits until 1 is reached, which gives a count of $L-1$. Decoding then proceeds by reading the following $L-1$ bits and by appending those to 1 in order to get the $\beta(I)$ natural binary code. $\beta(I)$ is then converted into its corresponding integer value.

The Elias γ code of I, $\gamma(I)$, can be obtained as a permutation of the γ' code of I, $\gamma'(I)$, by preceding each bit of the last $L-1$ bits of the $\beta(I)$ codeword with one of the bits of the $\alpha(L)$ codeword, where L is the length of $\beta(I)$. In other words, interleave the first L bits in $\gamma'(I)$ with the last $L-1$ bits by alternating those. For example, $\gamma(1) = 1$, $\gamma(2) = 001$, $\gamma(3) = 011$, and $\gamma(4) = 00001$.

16.3.6.2 Elias Delta (δ) Code

For a positive integer I, Elias δ coding generates a binary codeword of the form:

$$\begin{aligned}\delta(I) &= [(L'-1) \text{ zeros}][\beta(L)][\text{Last } (L-1) \text{ bits of } \beta(I)] \\ &= [\gamma'(L)][\text{Last } (L-1) \text{ bits of } \beta(I)],\end{aligned} \tag{16.26}$$

where $\beta(I)$ and $\beta(L)$ are the beta codes of I and L, respectively, L is the length of the binary codeword $\beta(I)$, and L' is the length of the binary codeword $\beta(L)$. For example,

$\delta(1) = 1, \delta(2) = 0100, \delta(3) = 0101$, and $\delta(4) = 01100$. In other words, Elias δ code can be constructed for a positive integer I using the following procedure:

1. Find the natural binary representation, $\beta(I)$, of I.
2. Determine the total number of bits, L, in $\beta(I)$.
3. Construct the γ' codeword, $\gamma'(L)$, of L, as discussed in Section 16.3.6.1.
4. The codeword $\delta(I)$ is formed as $\gamma'(L)$ followed by the last $(L-1)$ bits of $\beta(I)$ (i.e., $\beta(I)$ without the most significant bit 1).

An Elias δ code can be decoded by reading and counting the leading 0 bits until 1 is reached, which gives a count of $L'-1$. The $L'-1$ bits following the reached 1 bit are then read and appended to the 1 bit, which gives $\beta(L)$ and thus its corresponding integer value L. The next $L-1$ bits are then read and are appended to 1 in order to get $\beta(I)$. $\beta(I)$ is then converted into its corresponding integer value I.

16.3.6.3 Elias Omega (ω) Code

Similar to the previously discussed Elias δ code, the Elias ω code encodes the length L of the beta code, $\beta(I)$ of I, but it does this encoding in a recursive manner.

For a positive integer I, Elias ω coding generates a binary codeword of the form

$$\omega(I) = [\beta(L_N)][\beta(L_{N-1})]\ldots[\beta(L_1)][\beta(L_0)][\beta(I)][0], \tag{16.27}$$

where $\beta(I)$ is the beta code of I, $\beta(L_i)$ is the beta code of L_i, $i = 0,\ldots,N$, and $(L_i + 1)$ corresponds to the length of the codeword $\beta(L_{i-1})$, for $i = 1,\ldots,N$. In (16.27), $L_0 + 1$ corresponds to the length L of the codeword $\beta(I)$. The first codeword $\beta(L_N)$ can only be 10 or 11 for all positive integer values $I > 1$, and the other codewords $\beta(L_i)$, $i = 0,\ldots,N-1$, have lengths greater than two. The Elias omega code is thus formed by recursively encoding the lengths of the $\beta(L_i)$ codewords. The recursion stops when the produced beta codeword has a length of two bits.

An Elias ω code, $\omega(I)$, for a positive integer I can be constructed using the following recursive procedure:

1. Set $R = I$ and set $\omega(I) = [0]$.
2. Set $C = \omega(I)$.
3. Find the natural binary representation, $\beta(R)$, of R.
4. Set $\omega(I) = [\beta(R)][C]$.
5. Determine the length (total number of bits) L_R of $\beta(R)$.
6. If L_R is greater than 2, set $R = L_R - 1$ and repeat from Step 2.
7. If L_R is equal to 2, stop.
8. If L_R is equal to 1, set $\omega(I) = [0]$ and stop.

For example, $\omega(1) = 0$, $\omega(2) = 100$, $\omega(3) = 110$, and $\omega(4) = 101000$.

An Elias ω code can be decoded by initially reading the first three bits. If the third bit is 0, then the first two bits correspond to the beta code of the value of the integer data I, $\beta(I)$. If the third bit is one, then the first two bits correspond to the beta code of a length, whose value indicates the number of bits to be read and placed following the third 1 bit in order to form a beta code. The newly formed beta code corresponds either to a coded length or to the coded data value I depending whether the next following bit is 0 or 1. So the decoding proceeds by reading the next bit following the last formed beta code. If the read bit is 1, the last formed beta code corresponds to the beta code of a length whose value indicated the number of values to read following the read 1 bit. If the read bit is 0, the last formed beta code corresponds to the beta code of I and the decoding terminates.

16.3.6.4 Exponential-Golomb Codes

Exponential-Golomb codes [2] are parameterized structured universal codes that encode nonnegative integers, i.e., both positive integers and zero can be encoded in contrast to the previously discussed Elias codes which do not provide a code for zero.

For a positive integer I, a kth order Exp-Golomb (Exp-Golomb) code generates a binary codeword of the form

$$\begin{aligned} EG_k(I) &= [(L' - 1) \text{ zeros}][(\text{Most significant } (L - k) \text{ bits of } \beta(I)) + 1][\text{Last } k \text{ bits of } \beta(I)] \\ &= [(L' - 1) \text{ zeros}][\beta(1 + I/2^k)][\text{Last } k \text{ bits of } \beta(I)], \end{aligned} \tag{16.28}$$

where $\beta(I)$ is the beta code of I which corresponds to the natural binary representation of I, L is the length of the binary codeword $\beta(I)$, and L' is the length of the binary codeword $\beta(1 + I/2^k)$, which corresponds to taking the first $(L - k)$ bits of $\beta(I)$ and arithmetically adding 1. The length L can be computed as $L = (\lfloor \log_2(I) \rfloor + 1)$, for $I > 0$, where $\lfloor . \rfloor$ denotes rounding to the nearest smaller integer. For $I = 0$, $L = 1$. Similarly, the length L' can be computed as $L' = (\lfloor \log_2(1 + I/2^k) \rfloor + 1)$.

For example, for $k = 0$, $EG_0(0) = 1$, $EG_0(1) = 010$, $EG_0(2) = 011$, $EG_0(3) = 00100$, and $EG_0(4) = 00101$. For $k = 1$, $EG_1(0) = 10$, $EG_1(1) = 11$, $EG_1(2) = 0100$, $EG_1(3) = 0101$, and $EG_1(4) = 0110$.

Note that the Exp-Golomb code with order $k = 0$ of a nonnegative integer I, $EG_0(I)$, is equivalent to the Elias gamma′ code of $I + 1$, $\gamma'(I + 1)$. The zeroth-order ($k = 0$) Exp-Golomb codes are used as part of the H.264/AVC (MPEG-4 Part 10) video coding standard for coding parameters and data values related to macro blocks type, reference frame index, motion vector differences, quantization parameters, patterns for coded blocks, and other values [19].

A kth-order Exp-Golomb code can be decoded by first reading and counting the leading 0 bits until 1 is reached. Let the number of counted 0's be N. The binary codeword $\beta(I)$ is then obtained by reading the next N bits following the 1 bit, appending those read N bits to 1 in order to form a binary beta codeword, subtracting 1 from the formed binary codeword, and then reading and appending the last k bits. The obtained $\beta(I)$ codeword is converted into its corresponding integer value I.

16.4 LOSSLESS CODING STANDARDS

The need for interoperability between various systems have led to the formulation of several international standards for lossless compression algorithms targeting different applications. Examples include the standards formulated by the International Standards Organization (ISO), the International Electrotechnical Commission (IEC), and the International Telecommunication Union (ITU), which was formerly known as the International Consultative Committee for Telephone and Telegraph. A comparison of the lossless still image compression standards is presented in [20].

Lossless image compression standards include lossless JPEG (Chapter 17), JPEG-LS (Chapter 17), which supports lossless and near lossless compression, JPEG2000 (Chapter 17), which supports both lossless and scalable lossy compression, and facsimile compression standards such as the ITU-T Group 3 (T.4), Group 4 (T.6), JBIG (T.82), JBIG2 (T.88), and the Mixed Raster Content (MRC-T.44) standards [21]. While the lossless JPEG, JPEG-LS, and JPEG2000 standards are optimized for the compression of continuous-tone images, the facsimile compression standards are optimized for the compression of bilevel images except for the lastest MRC standard which is targeted for mixmode documents that can contain continuous-tone images in addition to text and line art.

The remainder of this section presents a brief overview of the JBIG, JBIG2, lossless JPEG, and JPEG2000 (with emphasis on lossless compression) standards. It is important to note that the image and video compression standards generally only specify the decoder-compatible bitstream syntax, thus leaving enough room for innovations and flexibility in the encoder and decoder design. The presented coding procedures below are popular standard implementations, but they can be modified as long as the generated bitstream syntax is compatible with the considered standard.

16.4.1 The JBIG and JBIG2 Standards

The JBIG standard (ITU-T Recommendation T.82, 1993) was developed jointly by the ITU and the ISO/IEC with the objective to provide improved lossless compression performance, for both business-type documents and binary halftone images, as compared to the existing standards. Another objective was to support progressive transmission. Grayscale images are also supported by encoding separately each bit plane. Later, the same JBIG committee drafted the JBIG2 standard (ITU-T Recommendation T.88, 2000) which provides improved lossless compression as compared to JBIG in addition to allowing lossy compression of bilevel images.

The JBIG standard consists of a context-based arithmetic encoder which takes as input the original binary image. The arithmetic encoder makes use of a context-based modeler that estimates conditional probabilities based on causal templates. A causal template consists of a set of already encoded neighboring pixels and is used as a context for the model to compute the symbol probabilities. Causality is needed to allow the decoder to recompute the same probabilities without the need to transmit side information.

JBIG supports sequential coding transmission (left to right, top to bottom) as well as progressive transmission. Progressive transmission is supported by using a layered coding scheme. In this scheme, a low resolution initial version of the image (initial layer) is first encoded. Higher resolution layers can then be encoded and transmitted in the order of increasing resolution. In this case the causal templates used by the modeler can include pixels from the previously encoded layers in addition to already encoded pixels belonging to the current layer.

Compared to the ITU Group 3 and Group 4 facsimile compression standards [12, 20], the JBIG standard results in 20% to 50% more compression for business-type documents. For halftone images, JBIG results in compression ratios that are two to five times greater than those obtained from the ITU Group 3 and Group 4 facsimile standards [12, 20].

In contrast to JBIG, JBIG2 allows the bilevel document to be partitioned into three types of regions: 1) text regions, 2) halftone regions, and 3) generic regions (such as line drawings or other components that cannot be classified as text or halftone). Both quality progressive and content progressive representations of a document are supported and are achieved by ordering the different regions in the document. In addition to the use of context-based arithmetic coding (MQ coding as in JBIG), JBIG2 allows also the use of run-length MMR (modified modified relative address designate) Huffman coding as in the Group 4 (ITU-T.6) facsimile standard, when coding the generic regions. Furthermore, JBIG2 supports both lossless and lossy compression. While the lossless compression performance of JBIG2 is slightly better than JBIG, JBIG2 can result in substantial coding improvements if lossy compression is used to code some parts of the bilevel documents.

16.4.2 The Lossless JPEG Standard

The JPEG standard was developed jointly by the ITU and ISO/IEC for the lossy and lossless compression of continuous-tone, color or grayscale, still images [22]. This section discusses very briefly the main components of the lossless mode of the JPEG standard (known as lossless JPEG).

The lossless JPEG coding standard can be represented in terms of the general coding structure of Fig. 16.1 as follows:

- Stage 1: Linear prediction/differential (DPCM) coding is used to form prediction residuals. The prediction residuals usually have a lower entropy than the original input image. Thus higher compression ratios can be achieved.
- Stage 2: The prediction residual is mapped into a pair of symbols (*category*, *magnitude*), where the symbol *category* gives the number of bits needed to encode *magnitude*.
- Stage 3: For each pair of symbols (*category*, *magnitude*), Huffman coding is used to code the symbol *category*. The symbol *magnitude* is then coded using a binary codeword whose length is given by the value *category*. Arithmetic coding can also be used in place of Huffman coding.

Complete details about the lossless JPEG standard and related recent developments, including JPEG-LS [23], are presented in Chapter 17.

16.4.3 The JPEG2000 Standard

JPEG2000 is the latest still image coding standard developed by the JPEG in order to support new features that are demanded by current modern applications and that are not supported by JPEG. Such features include lossy and lossless representations embedded within the same codestream, highly scalable codestreams with different progression orders (quality, resolution, spatial location, and component), region-of-interest (ROI) coding, and support for continuous-tone, bilevel, and compound image coding.

JPEG2000 is divided into 12 different parts featuring different application areas. JPEG2000 Part 1 [24] is the baseline standard and describes the minimal codestrean syntax that must be followed for compliance with the standard. All the other parts should include the features supported by this part. JPEG2000 Part 2 [25] is an extension of Part 1 and supports add-ons to improve the performance, including different wavelet filters with various subband decompositions. A brief overview of the JPEG2000 baseline (Part 1) coding procedure is presented below.

JPEG2000 [24] is a wavelet-based bit plane coding method. In JPEG2000, the original image is first divided into tiles (if needed). Each tile (subimage) is then coded independently. For color images, two optional color transforms, an irreversible color transform and a reversible color transform (RCT) are provided to decorrelate the color image components and increase the compression efficiency. The RCT should be used for lossless compression as it can be implemented using finite precision arithmetic and is perfectly invertible. Each color image component is then coded separately by dividing it first into tiles.

For each tile, the image samples are first shifted in level (if they are unsigned pixel values) such that they form a symmetric distribution of the DWT coefficients for the low-low (LL) subband. JPEG2000 (Part 1) supports two types of wavelet transforms: 1) an irreversible floating point 9/7 DWT [26], and 2) a reversible integer 5/3 DWT [27]. For lossless compression the 5/3 DWT should be used. After DC level shifting and the DWT, if lossy compression is chosen, the transformed coefficients are quantized using a deadzone scalar quantizer [4]. No quantization should be used in the case of lossless compression. The coefficients in each subband are then divided into coding blocks. The usual code block size is 64×64 or 32×32. Each coding block is then independently bit plane coded from the most significant bit plane (MSB) to the least significant bit plane using the embedded block coding with optimal truncation (EBCOT) algorithm [28].

The EBCOT algorithm consists of two coding stages known as tier-1 and tier-2 coding. In the tier-1 coding stage, each bit plane is fractionally coded using three coding passes: significant propagation, magnitude refinement, and cleanup (except the MSB, which is coded using only the cleanup pass). The significance propagation pass codes the significance of each sample based upon the significance of the neighboring eight pixels. The sign coding primitive is applied to code the sign information when a sample is coded for the first time as a nonzero bit plane coefficient. The magnitude refinement pass codes

only those samples that have already become significant. The cleanup pass will code the remaining coefficients that are not coded during the first two passes. The output symbols from each pass are entropy coded using context-based arithmetic coding. At the same time, the rate increase and the distortion reduction associated with each coding pass is recorded. This information is then used by the postcompression rate-distortion (PCRD) optimization (PCRD-opt) algorithm to determine the contribution of each coding block to the different quality layers in the final bitstream. Given the compressed bitstream for each coding block and the rate allocation result, tier-2 coding is performed to form the final coded bitstream. This two-tier coding structure gives great flexibility to the final bitstream formation. By determining how to assemble the sub-bitstreams from each coding block to form the final bitstream, different progression (quality, resolution, position, component) order can be realized. More details about the JPEG2000 standard are given in Chapter 17.

16.5 OTHER DEVELOPMENTS IN LOSSLESS CODING

Several other lossless image coding systems have been proposed [7, 9, 29]. Most of these systems can be described in terms of the general structure of Fig. 16.1, and they make use of the lossless symbol coding techniques discussed in Section 16.3 or variations on those. Among the recently developed coding systems, LOCO-I [7] was adopted as part of the JPEG-LS standard (Chapter 17), since it exhibits the best compression/complexity tradeoff. Context-based, Adaptive, Lossless Image Code (CALIC) [9] achieves the best compression performance at a slightly higher complexity than LOCO-I. Perceptual-based coding schemes can achieve higher compression ratios at a much reduced complexity by removing perceptually-irrelevant information in addition to the redundant information. In this case, the decoded image is required to only be visually, and not necessarily numerically, identical to the original image. In what follows, CALIC and perceptual-based image coding are introduced.

16.5.1 CALIC

CALIC represents one of the best performing practical and general purpose lossless image coding techniques.

CALIC encodes and decodes an image in raster scan order with a single pass through the image. For the purposes of context modeling and prediction, the coding process uses a neighborhood of pixel values taken only from the previous two rows of the image. Consequently, the encoding and decoding algorithms require a buffer that holds only two rows of pixels that immediately precede the current pixel. Figure 16.5 presents a schematic description of the encoding process in CALIC. Decoding is achieved by the reverse process. As shown in Fig. 16.5, CALIC operates in two modes: binary mode and continuous-tone mode. This allows the CALIC system to distinguish between binary and continuous-tone images on a local, rather than a global, basis. This distinction between the two modes is important due to the vastly different compression methodologies

employed within each mode. The former uses predictive coding, whereas the latter codes pixel values directly. CALIC selects one of the two modes depending on whether or not the local neighborhood of the current pixel has more than two distinct pixel values. The two-mode design contributes to the universality and robustness of CALIC over a wide range of images.

In the binary mode, a context-based adaptive ternary arithmetic coder is used to code three symbols, including an escape symbol. In the continuous-tone mode, the system has four major integrated components: prediction, context selection and quantization, context-based bias cancellation of prediction errors, and conditional entropy coding of prediction errors. In the prediction step, a gradient-adjusted prediction $\hat{y}$ of the current pixel y is made. The predicted value $\hat{y}$ is further adjusted via a bias cancellation procedure that involves an error feedback loop of one-step delay. The feedback value is the sample mean of prediction errors $\bar{e}$ conditioned on the current context. This results in an adaptive, context-based, nonlinear predictor $\check{y} = \hat{y} + \bar{e}$. In Fig. 16.5, these operations correspond to the blocks of "context quantization," "error modeling," and the error feedback loop.

The bias corrected prediction error $\check{y}$ is finally entropy coded based on a few estimated conditional probabilities in different conditioning states or coding contexts. A small number of coding contexts are generated by context quantization. The context quantizer partitions prediction error terms into a few classes by the expected error magnitude. The described procedures in relation to the system are identified by the blocks of "context quantization" and "conditional probabilities estimation" in Fig. 16.5. The details of this context quantization scheme in association with entropy coding are given in [9].

CALIC has also been extended to exploit interband correlations found in multiband images like color images, multispectral images, and 3D medical images. Interband CALIC

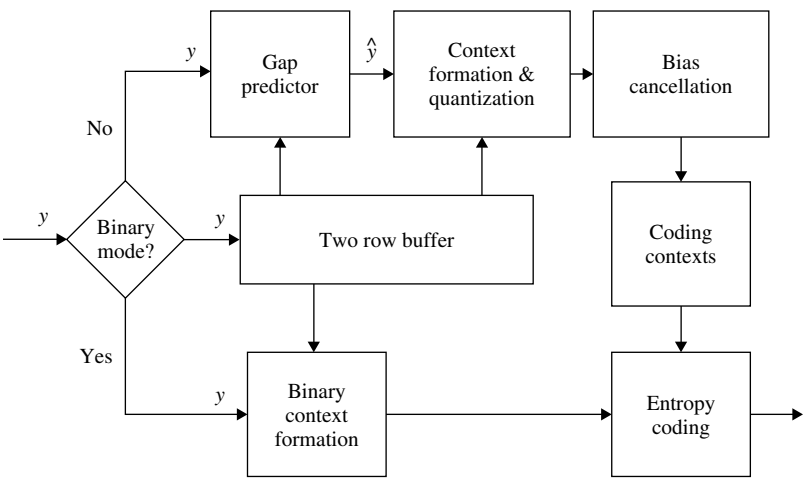

FIGURE 16.5

Schematic description of CALIC (Courtesy of Nasir Memon).

TABLE 16.5 Lossless bit rates with Intraband and Interband CALIC (Courtesy of Nasir Memon).

Image	JPEG-LS	Intraband CALIC	Interband CALIC
band	3.36	3.20	2.72
aerial	4.01	3.78	3.47
cats	2.59	2.49	1.81
water	1.79	1.74	1.51
cmpnd1	1.30	1.21	1.02
cmpnd2	1.35	1.22	0.92
chart	2.74	2.62	2.58
ridgely	3.03	2.91	2.72

can give 10% to 30% improvement over intraband CALIC, depending on the type of image. Table 16.5 shows bit rates achieved with intraband and interband CALIC on a set of multiband images. For the sake of comparison, results obtained with JPEG-LS are also included.

16.5.2 Perceptually Lossless Image Coding

The lossless coding methods presented so far require the decoded image data to be identical both quantitatively (numerically) and qualitatively (visually) to the original encoded image. This requirement usually limits the amount of compression that can be achieved to a compression factor of two or three even when sophisticated adaptive models are used as discussed in Section 16.5.1. In order to achieve higher compression factors, perceptually lossless coding methods attempt to remove redundant as well as perceptually irrelevant information.

Perceptual-based algorithms attempt to discriminate between signal components which are and are not detected by the human receiver. They exploit the spatio-temporal masking properties of the human visual system and establish thresholds of *just-noticeable distortion* (JND) based on psychophysical contrast masking phenomena. The interest is in bandlimited signals because of the fact that visual perception is mediated by a collection of individual mechanisms in the visual cortex, denoted *channels* or *filters*, that are selective in terms of frequency and orientation [30]. Mathematical models for human vision are discussed in Chapter 8.

Neurons respond to stimuli above a certain contrast. The necessary contrast to provoke a response from the neurons is defined as the *detection threshold*. The inverse of the detection threshold is the *contrast sensitivity*. Contrast sensitivity varies with frequency (including spatial frequency, temporal frequency, and orientation) and can be measured using *detection experiments* [31].

In *detection experiments*, the tested subject is presented with test images and needs only to specify whether the target stimulus is visible or not visible. They are used to

derive JND or detection thresholds in the absence or presence of a masking stimulus superimposed over the target. For the image coding application, the input image is the masker and the target (to be masked) is the quantization noise (distortion). JND contrast sensitivity profiles, obtained as the inverse of the measured detection thresholds, are derived by varying the target or the masker contrast, frequency, and orientation. The common signals used in vision science for such experiments are sinusoidal gratings. For image coding, bandlimited subband components are used [31].

Several perceptual image coding schemes have been proposed [31–35]. These schemes differ in the way the perceptual thresholds are computed and used in coding the visual data. For example, not all the schemes account for contrast masking in computing the thresholds. One method called DCTune [33] fits within the framework of JPEG. Based on a model of human perception that considers frequency sensitivity and contrast masking, it designs a fixed DCT quantization matrix (3 quantization matrices in the case of color images) for each image. The fixed quantization matrix is selected to minimize an overall perceptual distortion which is computed in terms of the perceptual thresholds. In such block-based methods, a scalar value can be used for each block or macro block to uniformly scale a fixed quantization matrix in order to account for the variation in available masking (and as a means to control the bit rate) [34]. The quantization matrix and the scalar value for each block need to be transmitted, resulting in additional side information.

The perceptual image coder proposed by Safranek and Johnston [32] works in a subband decomposition setting. Each subband is quantized using a uniform quantizer with a fixed step size. The step size is determined by the JND threshold for uniform noise at the most sensitive coefficient in the subband. The model used does not include contrast masking. A scalar multiplier in the range of 2 to 2.5 is applied to uniformly scale all step sizes in order to compensate for the conservative step size selection and to achieve a good compression ratio.

Higher compression can be achieved by exploiting the varying perceptual characteristics of the input image in a locally-adaptive fashion. Locally-adaptive perceptual image coding requires computing and making use of image-dependent, locally-varying, masking thresholds to adapt the quantization to the varying characteristics of the visual data. However, the main problem in using a locally-adaptive perceptual quantization strategy is that the locally-varying masking thresholds are needed both at the encoder and at the decoder in order to be able to reconstruct the coded visual data. This, in turn, would require sending or storing a large amount of side information, which might lead to data expansion instead of compression. The aforementioned perceptual-based compression methods attempt to avoid this problem by giving up or significantly restricting the local adaptation. They either choose a fixed quantization matrix for the whole image, select one fixed step size for a whole subband, or scale all values in a fixed quantization matrix uniformly.

In [31, 35], locally-adaptive perceptual image coders are presented without the need for side information for the locally-varying perceptual thresholds. This is accomplished by using a low-order linear predictor, at both the encoder and decoder, for estimating the locally available amount of masking. The locally-adaptive perceptual image coding

(a) Original *Lena* image, 8 bpp

(b) Decoded *Lena* image at 0.361 bpp

FIGURE 16.6

Perceptually-lossless image compression [31]. The perceptual thresholds are computed for a viewing distance equal to 6 times the image height.

schemes [31, 35] achieve higher compression ratios (25% improvement on average) in comparison with the nonlocally adaptive schemes [32, 33] with no significant increase in complexity. Figure 16.6 presents coding results obtained by using the locally adaptive perceptual image coder of [31] for the *Lena* image. The original image is represented by 8 bits per pixel (bpp) and is shown in Fig. 16.6(a). The decoded perceptually-lossless image is shown in Fig 16.6(b) and requires only 0.361 bpp (compression ratio $C_R = 22$).

REFERENCES

[1] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory*, IT-21:194–203, 1975.

[2] J. Teuhola. A compression method for clustered bit-vectors. *Inf. Process. Lett.*, 7:308–311, 1978.

[3] J. Wen and J. D. Villasenor. Structured prefix codes for quantized low-shape-parameter generalized Gaussian sources. *IEEE Trans. Inf. Theory*, 45:1307–1314, 1999.

[4] D. S. Taubman and M. W. Marcellin. *JPEG2000: Image Compression Fundamentals, Standards, and Practice*. Kluwer Academic Publishers, Boston, MA, 2002.

[5] R. B. Wells. *Applied Coding and Information Theory for Engineers*. Prentice Hall, New Jersey, 1999.

[6] R. G. Gallager. Variations on a theme by Huffman. *IEEE Trans. Inf. Theory*, IT-24:668–674, 1978.

[7] M. J. Weinberger, G. Seroussi, and G. Sapiro. LOCO-I: a low complexity, context-based, lossless image compression algorithm. In *Data Compression Conference*, 140–149, March 1996.

[8] D. Taubman. Context-based, adaptive, lossless image coding. *IEEE Trans. Commun.*, 45:437–444, 1997.

[9] X. Wu and N. Memon. Context-based, adaptive, lossless image coding. *IEEE Trans. Commun.*, 45:437–444, 1997.

[10] Z. Liu and L. Karam. Mutual information-based analysis of JPEG2000 contexts. *IEEE Trans. Image Process.*, accepted for publication.

[11] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 40: 1098–1101, 1952.

[12] V. Bhaskaran and K. Konstantinides. *Image and Video Compression Standards: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, 1995.

[13] W. W. Lu and M. P. Gough. A fast adaptive Huffman coding algorithm. *IEEE Trans. Commun.*, 41:535–538, 1993.

[14] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30:520–540, 1987.

[15] F. Rubin. Arithmetic stream coding using fixed precision registers. *IEEE Trans. Inf. Theory*, IT-25:672–675, 1979.

[16] A. Said. Arithmetic coding. In K. Sayood, editor, *Lossless Compression Handbook*, Ch. 5, Academic Press, London, UK, 2003.

[17] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, IT-23:337–343, 1977.

[18] T. A. Welch. A technique for high-performance data compression. *Computer*, 17:8–19, 1987.

[19] ITU-T Rec. H.264 (11/2007). Advanced video coding for generic audiovisual services. http://www.itu.int/rec/T-REC-H.264-200711-I/en (Last viewed: June 29, 2008).

[20] R. B. Arps and T. K. Truong. Comparison of international standards for lossless still image compression. *Proc. IEEE*, 82:889–899, 1994.

[21] K. Sayood. Facsimile compression. In K. Sayood, editor, *Lossless Compression Handbook*, Ch. 20, Academic Press, London, UK, 2003.

[22] W. Pennebaker and J. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Rheinhold, New York, 1993.

[23] ISO/IEC JTC1/SC29 WG1 (JPEG/JBIG); ITU Rec. T. 87. Information technology – lossless and near-lossless compression of continuous-tone still images – final draft international standard FDIS14495-1 (JPEG-LS). Tech. Rep., ISO, 1998.

[24] ISO/IEC 15444-1. JPEG2000 image coding system – part 1: core coding system. Tech. Rep., ISO, 2000.

[25] ISO/IEC JTC1/SC20 WG1 N2000. JPEG2000 part 2 final committee draft. Tech. Rep., ISO, 2000.

[26] A. Cohen, I. Daubechies, and J. C. Feaveau. Biorthogonal bases of compactly supported wavelets. *Commun. Pure Appl. Math.*, 45:485–560, 1992.

[27] R. Calderbank, I. Daubechies, W. Sweldens, and B. L. Yeo. Wavelet transforms that map integers to integers. *Appl. Comput. Harmonics Anal.*, 5(3):332–369, 1998.

[28] D. Taubman. High performance scalable image compression with EBCOT. *IEEE Trans. Image Process.*, 9:1151–1170, 2000.

[29] A. Said and W. A. Pearlman. An image multiresolution representation for lossless and lossy compression. *IEEE Trans. Image Process.*, 5:1303–1310, 1996.

[30] L. Karam. An analysis/synthesis model for the human visual based on subspace decomposition and multirate filter bank theory. In *IEEE International Symposium on Time-Frequency and Time-Scale Analysis*, 559–562, October 1992.

[31] I. Hontsch and L. Karam. APIC: Adaptive perceptual image coding based on subband decomposition with locally adaptive perceptual weighting. In *IEEE International Conference on Image Processing*, Vol. 1, 37–40, October 1997.

[32] R. J. Safranek and J. D. Johnston. A perceptually tuned subband image coder with image dependent quantization and post-quantization. In *IEEE ICASSP*, 1945–1948, 1989.

[33] A. B. Watson. DCTune: A technique for visual optimization of DCT quantization matrices for individual images. *Society for Information Display Digest of Technical Papers XXIV*, 946–949, 1993.

[34] R. Rosenholtz and A. B. Watson. Perceptual adaptive JPEG coding. In *IEEE International Conference on Image Processing*, Vol. 1, 901–904, September 1996.

[35] I. Hontsch and L. Karam. Locally-adaptive image coding based on a perceptual target distortion. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2569–2572, May 1998.