Data Compression

Khalid Sayood
University of Nebraska, Lincoln

I. INTRODUCTION

Data compression involves the development of a compact representation of information. Most representations of information contain large amounts of redundancy. Redundancy can exist in various forms. It may exist in the form of correlation: spatially close pixels in an image are generally also close in value. The redundancy might be due to context: the number of possibilities for a particular letter in a piece of English text is drastically reduced if the previous letter is a *q*. It can be probabilistic in nature: the letter *e* is much more likely to occur in a piece of English text than the letter *q*. It can be a result of how the information-bearing sequence was generated: voiced speech has a periodic structure. Or, the redundancy can be a function of the user of the information: when looking at an image we cannot see above a certain spatial frequency; therefore, the high-frequency information is redundant for this application. Redundancy is defined by the *Merriam-Webster Dictionary* as "the part of the message that can be eliminated without the loss of essential information." Therefore, one aspect of data compression is redundancy removal. Characterization of redundancy involves some form of modeling. Hence, this step in the compression process is also known as modeling. For historical reasons another name applied to this process is decorrelation.

After the redundancy removal process the information needs to be encoded into a binary representation. At this stage we make use of the fact that if the information is represented using a particular alphabet some letters may occur with higher probability than others. In the coding step we use shorter code words to represent letters that occur more frequently, thus lowering the average number of bits required to represent each letter.

Compression in all its forms exploits structure, or redundancy, in the data to achieve a compact representation. The design of a compression algorithm involves understanding the types of redundancy present in the data and then developing strategies for exploiting these redundancies to obtain a compact representation of the data. People have come up with many ingenious ways of characterizing and using the different types of redundancies present in different kinds of technologies from the telegraph to the cellular phone and digital movies.

One way of classifying compression schemes is by the model used to characterize the redundancy. However, more popularly, compression schemes are divided into two main groups: lossless compression and lossy compression. Lossless compression preserves all the information in the data being compressed, and the reconstruction is identical to the original data. In lossy compression some of the information contained in the original data is irretrievably lost. The loss in information is, in some sense, a payment for achieving higher levels of compression. We begin our examination of data compression schemes by first looking at lossless compression techniques.

II. LOSSLESS COMPRESSION

Lossless compression involves finding a representation which will exactly represent the source. There should be no loss of information, and the decompressed, or

reconstructed, sequence should be identical to the original sequence. The requirement that there be no loss of information puts a limit on how much compression we can get. We can get some idea about this limit by looking at some concepts from information theory.

A. Information and Entropy

In one sense it is impossible to denote anyone as the parent of data compression: people have been finding compact ways of representing information since the dawn of recorded history. One could argue that the drawings on the cave walls are representations of a significant amount of information and therefore qualify as a form of data compression. Significantly less controversial would be the characterizing of the Morse code as a form of data compression. Samuel Morse took advantage of the fact that certain letters such as *e* and *a* occur more frequently in the English language than *q* or *z* to assign shorter code words to the more frequently occurring letters. This results in lower average transmission time per letter. The first person to put data compression on a sound theoretical footing was Claude E. Shannon (1948). He developed a quantitative notion of information that formed the basis of a mathematical representation of the communication process.

Suppose we conduct a series of independent experiments where each experiment has outcomes A_1, A_2, . . ., A_M. Shannon associated with each outcome a quantity called *self information,* defined as

$$i(A_k) = \log \frac{1}{P(A_k)}$$

The units of self-information are bits, nats, or Hartleys, depending on whether the base of the logarithm is 2 *e,* or 10. The average amount of information associated with the experiment is called the *entropy H:*

$$H = E[i(A)] = \sum_{k=1}^{M} i(A_k)P(A_k) = \sum_{k=1}^{M} P(A_k) \log \frac{1}{P(A_k)}$$

$$= -\sum_{k=1}^{M} P(A_k) \log P(A_k) \quad (1)$$

Suppose the "experiment" is the generation of a letter by a source, and further suppose the source generates each letter independently according to some probability model. Then H as defined in Eq. (1) is called the entropy of the source. Shannon showed that if we compute the entropy in bits (use logarithm base 2) the entropy is the smallest average number of bits needed to represent the source output.

We can get an intuitive feel for this connection between the entropy of a source and the average number of bits needed to represent its output (denoted by *rate*) by performing a couple of experiments.

Suppose we have a source which puts out one of four letters $\{A_1, A_2, A_3, A_4\}$. In order to ascertain the outcome of the experiment we are allowed to ask a predetermined sequence of questions which can be answered with a *yes* or a *no.* Consider first the case of a source which puts out each letter with equal probability, that is,

$$P(A_k) = \frac{1}{4} \qquad k = 1,2,3,4$$

From Eq. (1) the entropy of this source is two bits. The sequence of questions can be represented as a flowchart as shown in Fig. 1. Notice that we need to ask two questions in order to determine the output of the source. Each answer can be represented by a single bit (1 for *yes* and 0 for *no*); therefore, we need two bits to represent each output of the source. Now consider a

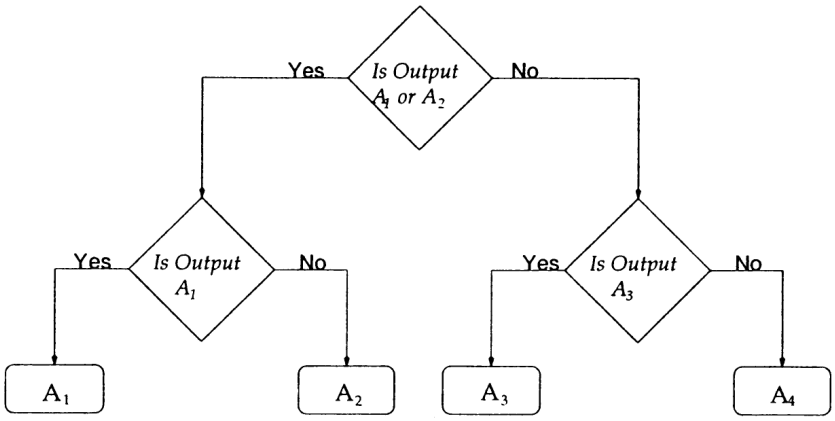

Figure 1 Sequence of questions for equiprobable source.

slightly different situation in which the source puts out the four different letters with different probabilities:

$$P(A_1) = \frac{1}{2}, \quad P(A_2) = \frac{1}{4}, \quad P(A_3) = \frac{1}{8}, \quad P(A_4) = \frac{1}{8}$$

The entropy of this source is 1.75 bits.

Armed with the information about the source, we construct a different sequence of questions shown in Fig. 2. Notice that when the output of the source is A_1 we need ask only one question. Because $P(A_1) = \frac{1}{2}$ on the average this will happen half the time. If the source output is A_2 we need to ask two questions. This will happen a quarter of the time. If the source output is A_3 or A_4 we will need to ask three questions. This too will happen a quarter of the time. Thus, half the time we can represent the output of the source with one bit, a quarter of the time with two bits, and another quarter of the time with three bits. Therefore, on the average we will need 1.75 bits to represent the output of the source. We should note that if our information about the frequency with which the letters occur is wrong we will end up using more bits than if we had used the question sequence of Fig. 1. We can easily see this effect by switching the probabilities of A_1 and A_2 and keeping the questioning strategy of Fig. 2.

Both these examples demonstrate the link between average information, or entropy of the source, and the average number of bits, or the rate, required to represent the output of the source. They also demonstrate the need for taking into account the probabilistic structure of the source when creating a representation. Note that incorrect estimates of the probability will substantially decrease the compression efficiency of the procedure.

The creation of a binary representation for the source output, or the generation of a *code* for a source, is the topic of the next section.

B. Coding

In the second example of the previous section the way we obtained an efficient representation was to use fewer bits to represent letters that occurred more frequently—the same idea that Samuel Morse had. It is a simple idea, but in order to use it we need an algorithm for systematically generating variable length code words for a given source. David Huffman (1951) created such an algorithm for a class project. We describe this algorithm below. Another coding algorithm which is fast gaining popularity is the arithmetic coding algorithm. We will also describe the arithmetic coding algorithm in this section.

1. Huffman Coding

Huffman began with two rather obvious conditions on the code and then added a third that allowed for the construction of the code. The conditions were:

1. The codes corresponding to the higher probability letters could not be longer than the code words associated with the lower probability letters.

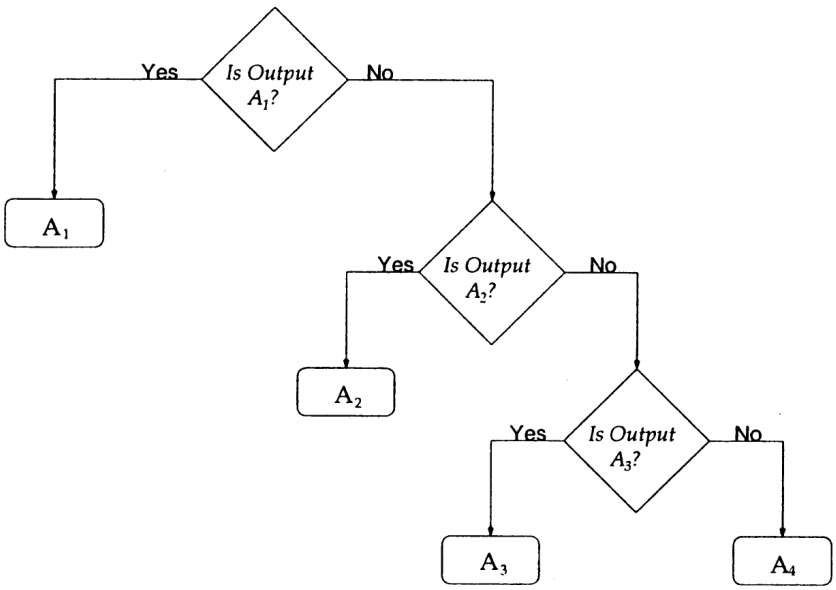

Figure 2 Sequence of questions for second source.

2. The two lowest probability letters had to have code words of the same length.

He added a third condition to generate a practical compression scheme.

3. The two lowest probability letters have codes that are identical except for the last bit.

It is easy to visualize these conditions if we think of the code words as the path through a binary tree: a zero bit denoting a left branch and a one bit denoting a right branch. The two lowest probability letters would then be the two leaves of a node at the lowest level of the tree. We can consider the parent node of these leaves as a letter in a reduced alphabet which is obtained as a combination of the two lowest probability symbols. The probability of the letter corresponding to this node would be the sum of the probabilities of the two individual letters. Now, we can find the two lowest probability symbols of the reduced alphabet and treat them as the two leaves of a common node. We can continue in this fashion until we have completed the tree.

We can see this process best through an example.

Example 1: Huffman Coding

Suppose we have a source alphabet with five letters $\{a_1,a_2,a_3,a_4,a_5\}$ with probabilities of occurrence $P(a_1) = 0.15$, $P(a_2) = 0.04$, $P(a_3) = 0.26$, $P(a_4) = 0.05$, and $P(a_5) = 0.50$. We can calculate the entropy to be

$$H = -\sum_{i=1}^{5} P(a_i) \log P(a_i) = 1.817684 \text{ bits}$$

If we sort the probabilities in descending order we can see that the two letters with the lowest probabilities are a_2 and a_4. These will become the leaves on the lowest level of the binary tree. The parent node of these leaves will have a probability of 0.09. If we consider the parent node as a letter in a reduced alphabet it will be one of the two letters with the lowest probability: the other one being a_1. Continuing in this manner we get the binary tree shown in Fig. 3. The code is

a_1 110
a_2 1111
a_3 10
a_4 1110
a_5 0

It can be shown that, for a sequence of independent letters, or a memoryless source, the rate of the Huffman code will always be within one bit of the entropy.

$$H \leq R \leq 1$$

In fact we can show that (Gallagher, 1978) if p_{max} is the largest probability in the probability model, then

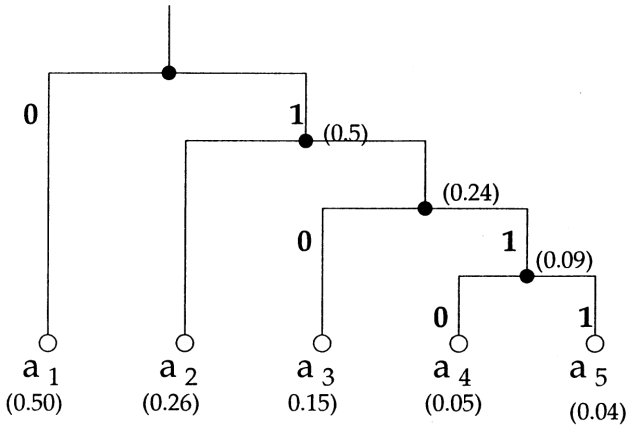

Figure 3 Huffman code for the five letter alphabet.

for $p_{max} < 0.5$ the upper bound for the Huffman code is $H(\mathcal{S}) + p_{max}$, while for $p_{max} \geq 0.5$, the upper bound is $H(\mathcal{S}) + p_{max} + 0.086$. If instead of coding each letter separately we group the letters into blocks containing n letters, then the rate is guaranteed to be even closer to the entropy of the source. Using our looser bound we can show that the bounds on the average length of the code will be

$$H \leq R \leq \frac{1}{n}$$

However, blocking letters together means an exponential increase in the size of the alphabet. Therefore, in many situations this particular approach is not very practical.

2. Arithmetic Coding

Practical arithmetic coding came into existence in 1976 through the work of Risannen (1976) and Pasco (1976). However, the basic ideas of arithmetic coding have their origins in the original work of Shannon (1948). (For a brief history see Sayood, 2000). Arithmetic coding relies on the fact that there are an uncountably infinite number of numbers between 0 and 1 (or any other nonzero interval on the real number line). Therefore, we can assign a unique number from the unit interval to any sequence of symbols from a finite alphabet. We can then encode the entire sequence with this single number which acts as a label or tag for this sequence. In other words, this number is a code for this sequence: a binary representation of this number is a binary code for this sequence. Because this tag is unique, in theory, given the tag the decoder can reconstruct the entire sequence. In order to implement this idea we need a mapping from

the sequence to the tag and an inverse mapping from the tag to the sequence.

One particular mapping is the cumulative density function of the sequence. Let us view the sequence to be encoded as the realization of a sequence of random variables $\{X_1, X_2, \cdots\}$ and represent the set of all possible realizations which have nonzero probability of occurrence in lexicographic order by $\{\mathbf{X}_i\}$. Given a particular realization

$$\mathbf{X}_k = x_{k,1}, x_{k,2}, \cdots$$

the cumulative distribution function $F_{\mathbf{X}}(x_{k,1}, x_{k,2}, \cdots)$ is a number between 0 and 1. Furthermore, as we are only dealing with sequences with nonzero probability, this number is unique to the sequence $\mathbf{X}_k$. In fact, we can uniquely assign the half-open interval $[F_{\mathbf{X}}(\mathbf{X}_{k-1}), F_{\mathbf{X}}(\mathbf{X}_{k-1}))$ to the sequence $\mathbf{X}_k$. Any number in this interval, which we will refer to as the tag interval for $\mathbf{X}_k$, can be used as a label or tag for the sequence $\mathbf{X}_k$. The arithmetic coding algorithm is essentially the calculation of the end points of this tag interval. Before we describe the algorithm for computing these end points let us look at an example for a sequence of manageable length.

Suppose we have an *iid* sequence with letters from an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$. The probability of occurrence of the letters are given by $p_0 = 0.3$, $p_1 = 0.1$, $p_2 = 0.2$, and $p_4 = 0.4$. Ordering the letters from the smallest to the largest index, we have $F_X(a_1) = 0.3$, $F_X(a_2) = 0.4$, $F_X(a_3) = 0.6$, and $F_X(a_4) = 1.0$. We will find the tag interval for the sequence a_4, a_1, a_2, a_4. We begin with a sequence that consists of a single letter a_4. Given that $F_X(a_4) = 1.0$ and $F_X(a_3) = 0.6$, the tag interval is $[0.6,1.0)$. Now consider the two-letter sequence a_4, a_1. If we impose a lexicographic ordering,

$$\begin{aligned} &F_{X_1,X_2}(X_1 = a_4, X_2 = a_1) \\ &= \sum_{i=1}^{3} \sum_{j=1}^{4} Pr[X_1 = a_i, X_2 = a_j] + Pr[X_1 = a_4, X_2 = a_1] \\ &= F_X(a_3) + Pr[X_1 = a_4, X_2 = a_1] \\ &= 0.6 + 0.4 \times 0.3 = 0.72 \end{aligned}$$

The two-letter sequence prior to a_4, a_1 in the lexicographic ordering is a_3, a_4. We can compute $F_{X1,X2}(X_1 = a_3, X_2 = a_4) = 0.6$; therefore, the tag interval is $(0.6,0.72]$. Another way of obtaining the tag interval would be to partition the single letter tag interval $[0.6,1)$ in the same proportions as the partitioning of the unit interval. As in the case of the unit interval, the first subpartition would correspond to the letter a_1, the second subpartition would correspond to the letter a_2, and so on. As the second letter of the sequence under consideration is a_1, the tag interval would be the first subinterval, which is $[0.6,0.72)$. Note that the tag interval for the two-letter sequence is wholly contained in the tag interval corresponding to the first letter. Furthermore, note that the size of the interval is $p_4 p_1 = 0.12$. Continuing in this manner for the three-letter sequence, we can compute $F_{X1,X2,X3}(a_4, a_1, a_2) = 0.672$ and $F_{X1,X2,X3}(a_4, a_1, a_1) = 0.648$. Thus, the tag interval is $(0.648,0.672]$. Again notice that the tag interval for the three letter sequence is entirely contained within the tag interval for the two-letter sequence, and the size of the interval has been reduced to $p_4 p_1 p_2 = 0.024$. This progression is shown in Fig. 4. If we represent the upper limit of the tag interval at time n by $u^{(n)}$ and the lower limit by $l^{(n)}$, we can show that

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)}) F_X(x_n - 1) \quad (2)$$

$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)}) F_X(x_n). \quad (3)$$

Notice that as the sequence length increases, the tag interval is always a subset of the previous tag interval. However, the number of digits required to represent the tag intervals increases as well. This was one of the problems with the initial formulation of arithmetic coding. Another was the fact that you could not send the code until the last element of the sequence was encoded. Both these problems can be partially resolved by noting that if we use a binary alphabet, once the interval is complete in either the upper or lower half of the unit interval the most significant digits of the upper and lower limits of the tag interval are identical. Furthermore, there is no possibility of the tag interval migrating from the half of the unit interval in which it is contained to the other half of the unit interval. Therefore, we can send the most significant bit of the upper and lower intervals as the tag for the sequence. At the same time we can shift the most significant bit out of the upper and lower limits, effectively doubling the size of the tag interval. Thus, each time the tag interval is trapped in either the upper or lower half of the unit interval, we obtain one more bit of the code and we expand the interval. This way we prevent the necessity for increasing precision *as long as the tag interval resides entirely within the top or bottom half of the unit interval.* We can also start transmitting the code before the entire sequence has been encoded. We have to use a slightly more complicated strategy when the tag interval straddles the midpoint of the unit interval. We leave the details to Sayood (2000).

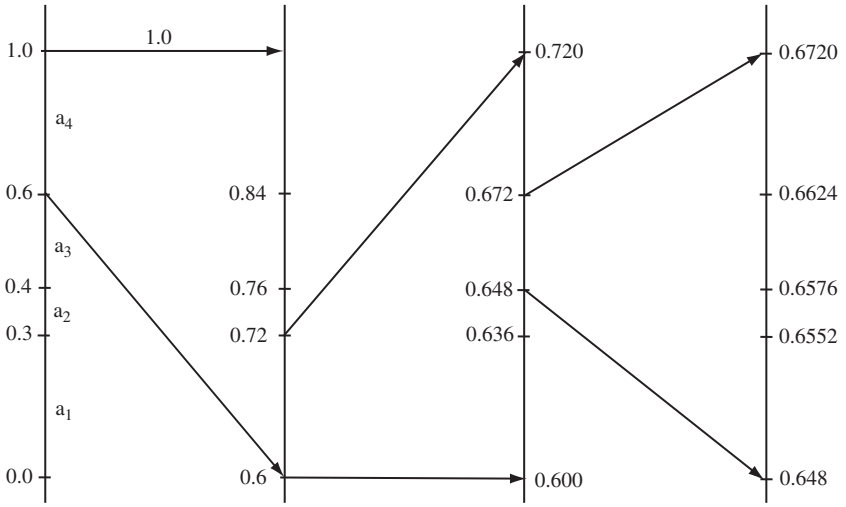

Figure 4 Narrowing of the tag interval for the sequence $a_4a_0a_2a_4$.

C. Sources with Memory

Throughout our discussion we have been assuming that each symbol in a sequence occurs independently of the previous sequence. In other words, there is no *memory* in the sequence. Most sequences of interest to us are not made up of independently occurring symbols. In such cases an assumption of independence would give us an incorrect estimate of the probability of occurrence of that symbol, thus leading to an inefficient code. By taking account of the dependencies in the sequence we can significantly improve the amount of compression available. Consider the letter u in a piece of English text. The frequency of occurrence of the letter u occurring in a piece of English text is approximately 0.018. If we had to encode the letter u and we used this value as our estimate of the probability of the letter, we would need approximately $\lfloor \log_2 \frac{1}{0.018} \rfloor = 6$ bits. However, if we knew the previous letter was q our estimate of the probability of the letter u would be substantially more than 0.018 and thus encoding u would require fewer bits. Another way of looking at this is by noting that if we have an accurate estimate of the probability of occurrence of particular symbols we can obtain a much more efficient code. If we know the preceding letter(s) we can obtain a more accurate estimate of the probabilities of occurrence of the letters that follow. Similarly, if we look at pixel values in a natural image and assume that the pixels occur independently, then our estimate of the probability of the values that each pixel could take on would be approximately the same (and small). If we took into account the values of the neighboring pixels, the probability that the pixel under consideration would have a similar value would be quite high. If, in fact, the pixel had a value close to its neighbors, encoding it in this context would require many fewer bits than encoding it independent of its neighbors (Memon and Sayood, 1995).

Shannon (1948) incorporated this dependence in the definition of entropy in the following manner. Define

$$G_n = -\sum_{i_1=m}^{i_1=m}\sum_{i_2=m}^{i_2=m}\cdots\sum_{i_n=1}^{i_n=m} P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n) \log P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n)$$

where $\{X_1, X_2, \ldots, X_n\}$ is a sequence of length n generated by a source $\mathcal{S}$. Then the entropy of the source is defined as

$$H(\mathcal{S}) = \lim_{n\to\infty} \frac{1}{n} G_n$$

There are a large number of compression schemes that make use of this dependence.

D. Context-Based Coding

The most direct approach to using this dependence is to code each symbol based on the probabilities provided by the context. If we are to sequentially encode the sequence $x_1, x_2, \ldots, x_n$, we need $-\log \Pi_{i=0}^{n-1} p(x_{i+1}|x_1, x_2, \ldots, x_n)$ bits (Weinberger *et al.*, 1995). The history of the sequence makes up its context. In prac-

tice, if we had these probabilities available they could be used by an arithmetic coder to code the particular symbol. To use the entire history of the sequence as the context is generally not feasible. Therefore, the context is made up of some subset of the history. It generally consists of those symbols that our knowledge of the source tells us will have an affect on the probability of the symbol being encoded and which are available to both the encoder and the decoder. For example, in the binary image coding standard JBIG, the context may consist of the pixels in the immediate neighborhood of the pixel being encoded along with a pixel some distance away which reflects the periodicity in half-tone images. Some of the most popular context-based schemes today are *ppm* (prediction with partial match) schemes.

1. Prediction with Partial Match

In general, the larger the size of a context, the higher the probability that we can accurately predict the symbol to be encoded. Suppose we are encoding the fourth letter of the sequence *their*. The probability of the letter *i* is approximately 0.053. It is not substantially changed when we use the single letter context *e,* as *e* is often followed by most letters of the alphabet. In just this paragraph it has been followed by *d, i, l, n, q, r, t,* and *x*. If we increase the context to two letters *he,* the probability of *i* following *he* increases. It increases even more when we go to the three-letter context *the*. Thus, the larger the context, the better off we usually are. However, all the contexts and the probabilities associated with the contexts have to be available to both the encoder and the decoder. The number of contexts increases exponentially with the size of the context. This puts a limit on the size of the context we can use. Furthermore, there is the matter of obtaining the probabilities relative to the contexts. The most efficient approach is to obtain the frequency of occurrence in each context from the past history of the sequence. If the history is not very large, or the symbol is an infrequent one, it is quite possible that the symbol to be encoded has not occurred in this particular context.

The *ppm* algorithm initially proposed by Cleary and Witten (1984) starts out by using a large context of predetermined size. If the symbol to be encoded has not occurred in this context, an escape symbol is sent and the size of the context is reduced. For example, when encoding *i* in the example above we could start out with a context size of three and use the context *the*. If *i* has not appeared in this context, we would send an escape symbol and use the second order context *he*. If *i* has not appeared in this context either, we reduce the context size to one and look to see how often *i* has occurred in the context of *e*. If this is zero, we again send an escape and use the zero order context. The zero order context is simply the frequency of occurrence of *i* in the history. If *i* has never occurred before, we send another escape symbol and encode *i,* assuming an equally likely occurrence of each letter of the alphabet.

We can see that if the symbol has occurred in the longest context, it will likely have a high probability and will require few bits. However, if it has not, we pay a price in the shape of the escape symbol. Thus, there is a tradeoff here. Longer contexts may mean higher probabilities or a long sequence of escape symbols. One way out of this has been proposed by Cleary and Teahan (1997) under the name *ppm**. In *ppm** we look at the longest context that has appeared in the history. It is very likely that this context will be followed by the symbol we are attempting to encode. If so, we encode the symbol with very few bits. If not, we drop to a relatively short context length and continue with the standard *ppm* algorithm. There are a number of variants of the *ppm** algorithm, with the most popular being *ppmz* developed by Bloom. It can be found at *http://www.cbloom.com/src/ppmz.html.*

E. Predictive Coding

If the data we are attempting to compress consists of numerical values, such as images, using context-based approaches directly can be problematic. There are several reasons for this. Most context-based schemes exploit exact reoccurrence of patterns. Images are usually acquired using sensors that have a small amount of noise. While this noise may not be perceptible, it is sufficient to reduce the occurrence of exact repetitions of patterns. A simple alternative to using the context approach is to generate a prediction for the value to be encoded and encode the prediction error. If there is considerable dependence among the values with high probability, the prediction will be close to the actual value and the prediction error will be a small number. We can encode this small number, which as it occurs with high probability will require fewer bits to encode. An example of this is shown in Fig. 5. The plot on the left of Fig. 5 is the histogram of the pixel values of the *sensin* image, while the plot on the right is the histogram of the differences between neighboring pixels. We can see that the small differences occur much more frequently and therefore can be encoded using fewer bits. As

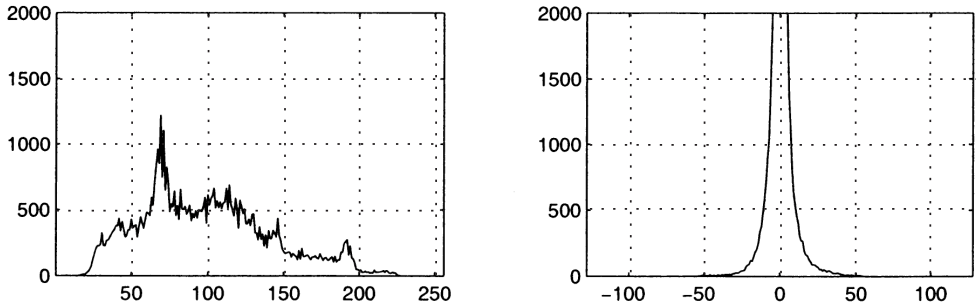

Figure 5 Histograms of pixel values in the *sensin* image and of the differences between neighboring pixels.

opposed to this, the actual pixel values have a much more uniform distribution.

Because of the strong correlation between pixels in a neighborhood, predictive coding has been highly effective for image compression. It is used in the current state-of-the-art algorithm CALIC (Wu and Memon, 1996) and forms the basis of JPEG-LS, which is the standard for lossless image compression.

1. JPEG-LS

The JPEG-LS standard uses a two-stage prediction scheme followed by a context-based coder to encode the difference between the pixel value and the prediction. For a given pixel the prediction is generated in the following manner. Suppose we have a pixel with four neighboring pixels as shown in Fig. 6. The initial prediction X is obtained as

```
if NW ≥ max(W,N)
X = max(W,N)
else
{
    if NW ≤ min(W,N)
    X = min(W,N)
    else
    X = W + N − NW
}
```

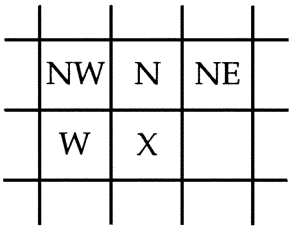

Figure 6 Labeling of pixels in the neighborhood of a pixel to be encoded.

This prediction is then refined by using an estimate of how much the prediction differed from the actual value in similar situations in the past. The "situation" is characterized by an activity measure which is obtained using the differences of the pixels in the neighborhood. The differences $NE - N$, $N - NW$, and $NW - W$ are compared against thresholds which can be defined by the user and a number between 0 and 364 and a *SIGN* parameter which takes on the values $+1$ and -1. The sign parameter is used to decide whether the correction should be added or subtracted from the original prediction. The difference between the pixel value and its prediction is mapped into the range of values occupied by the pixel and encoded using Golomb codes. For details, see Sayood (2000) and Weinberger *et al.* (1998).

2. Burrows-Wheeler Transform

The Burrows-Wheeler transform (BWT) uses the memory in a sequence in a somewhat different manner than either of the two techniques described previously. In this technique the entire sequence to be encoded is read in and all possible cyclic shifts of the sequence are generated. These are then sorted in lexicographic order. The last letter of each sorted cyclically shifted sequence is then transmitted along with the location of the original sequence in the sorted list. The sorting results in long sequences of identical letters in the transmitted sequence. This structure can be used to provide a very efficient representation of the transmitted sequence. The easiest way to understand the BWT algorithm is through an example.

Example: BWT

Suppose we wish to encode the sequence *RINTINTIN*. We first obtain all cyclical shifts of this sequence and then sort them in lexicographic order as shown in Fig. 7.

We transmit the string consisting of the last letters in the lexicographically ordered set and the position

R	I	N	T	I	N	T	I	N	I	N	R	I	N	T	I	N	T
N	R	I	N	T	I	N	T	I	I	N	T	I	N	R	I	N	T
I	N	R	I	N	T	I	N	T	I	N	T	I	N	T	I	N	R
T	I	N	R	I	N	T	I	N	N	R	I	N	T	I	N	T	I
N	T	I	N	R	I	N	T	I	N	T	I	N	R	I	N	T	I
I	N	T	I	N	R	I	N	T	N	T	I	N	T	I	N	R	I
T	I	N	T	I	N	R	I	N	R	I	N	T	I	N	R	I	N
N	T	I	N	T	I	N	R	I	T	I	N	R	I	N	T	I	N
I	N	T	I	N	T	I	N	R	T	I	N	T	I	N	R	I	N

Figure 7 Cyclically shifted versions of the original sequence and lexicographically order set of the cyclical shifts.

of the original string in the lexicographically ordered set. For this example the transmitted sequence would be the string *TTRIIINNN* and the index 7. Notice that the string to be transmitted contains relatively long strings of the same letter. If our example string had been realistically long, the runs of identical letters would have been correspondingly long. Such a string is easy to compress. The method of choice for BWT is move-to-front coding (Burrows and Wheeler, 1994; Nelson, 1996; Sayood, 2000).

Once the string and the index have been received, we decode them by working backwards. We know the last letter in the string is its seventh element which is *N*. To find the letter before *N* we need to generate the string containing the first letters of the lexicographically ordered set. This can be easily obtained as *IIINNNRTT* by lexicographically ordering the received sequence. We will refer to this sequence of first letters as the $\mathcal{F}$ sequence and the sequence of last letters as the $\mathcal{L}$ sequence. Note that given a particular string and its cyclic shift, the last letter of the string becomes the first letter of the cyclically shifted version, and the last letter of the cyclically shifted version is the letter prior to the last letter in the original sequence. Armed with this fact and the knowledge of where the original sequence was in the lexicographically ordered set, we can decode the received sequence. We know the original sequence was seventh in the lexicographically ordered set; therefore, the last letter of the sequence has to be *N*. This is the first *N* in $\mathcal{L}$. Looking in $\mathcal{F}$ we see that the first *N* appears in the fourth location. The fourth element in $\mathcal{L}$ is *I*. Therefore, the letter preceding *N* in our decoded sequence is *I*. This *I* is the first *I* in $\mathcal{L}$; therefore, we look for the first *I* in $\mathcal{F}$. The first *I* occurs in the first location. The letter in the first location in $\mathcal{L}$ is *T*. Therefore, the decoded sequence becomes *TIN*. Continuing in this fashion we can decode the entire sequence.

The BWT is the basis for the compression utilities *bzip* and *bzip2*. For more details on BWT compression, see Burrows and Wheeler (1994), Nelson and Gailly (1996), and Sayood (2000).

F. Dictionary Coding

One of the earliest forms of compression is to create a dictionary of commonly occurring patterns which is available to both the encoder and the decoder. When this pattern occurs in a sequence to be encoded it can be replaced by the index of its entry in the dictionary. A problem with this approach is that a dictionary that works well for one sequence may not work well for another sequence. In 1977 and 1978, Jacob Ziv and Abraham Lempel (1977, 1978) described two different ways of making the dictionary adapt to the sequence being encoded. These two approaches form the basis for many of the popular compression programs of today.

1. LZ77

The 1977 algorithm commonly referred to as LZ77 uses the past of the sequence as the dictionary. Whenever a pattern recurs within a predetermined window, it is replaced by a pointer to the beginning of its previous occurrence and the length of the pattern. Consider the following (admittedly weird) sequence:

a mild fork for miles vorkosigan

We repeat this sentence in Fig. 8 with recurrences replaced by pointers. If there were sufficient recurrences of long patterns, we can see how this might result in a compressed representation. In Fig. 8 we can see that the "compressed" representation consists of both pointers and fragments of text that have not been encountered previously. We need to inform the decoder when a group of bits is to be interpreted as a pointer and when it is to be interpreted as text. There are a number of different ways in which this can be done. The original LZ77 algorithm used a sequence of triples <*o*,*l*,*c*> to encode the source output, where *o* is the offset or distance to the previous recurrence, *l* is the length of the pattern, and *c* corresponds to the character following the recurrence. In this approach when a symbol was encountered for the first time (in the encoding window) the values for *o* and *l* were set to 0. Storer and Syzmanski (1982) suggested using a one bit flag to differentiate between pointers and symbols which had not occurred previously in the coding

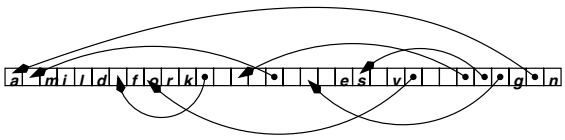

Figure 8 Illustration for LZ77.

window. In their variant, commonly known as LZSS, the one bit flag is followed by either a pair (<*o,l*>) or the code word of the new symbol. There are a number of variants of both the LZ77 and the LZSS algorithms which are the basis for several popularly used compression utilities. Included among them are *gzip* and *PNG* (portable network graphics).

2. LZ78

The 1978 algorithm requires actually building a dynamic dictionary based on the past symbols in the sequence. The sequence is encoded using pairs of code words <*i,c*>, where i is the index to a pattern in the dictionary and c is the code word of the character following the current symbol. The most popular variant of the LZ78 algorithm is the LZW algorithm developed by Terry Welch (1984). In this variation the dictionary initially contains all the individual letters of the source alphabet. The encoder operates on the sequence by collecting the symbols in the sequence into a pattern p until such time as the addition of another symbol α will result in a pattern which is not contained in the dictionary. The index to p is then transmitted, the pattern p concatenated with α is added as the next entry in the dictionary, and a new pattern p is begun with α as the first symbol.

Variants of the LZW algorithm are used in the UNIX *compress* command as part of the graphic interchange format (GIF) and for the modem protocol v. 42bis.

III. LOSSY COMPRESSION

The requirement that no information be lost in the compression process puts a limit on the amount of compression we can obtain. The lowest number of bits per sample is the entropy of the source. This is a quantity over which we generally have no control. In many applications this requirement of no loss is excessive. For example, there is high-frequency information in an image which cannot be perceived by the human visual system. It makes no sense to preserve this information for images that are destined for human consumption. Similarly, when we listen to sampled speech we cannot perceive the exact numerical value of each sample. Therefore, it makes no sense to expend coding resources to preserve the exact value of each speech sample. In short, there are numerous applications in which the preservation of all information present in the source output is not necessary. For these applications we relax the requirement that the reconstructed signal be identical to the original. This allows us to create compression schemes that can provide a much higher level of compression. However, it should be kept in mind that we generally pay for higher compression by increased information loss. Therefore, we measure the performance of the compression system using two metrics. We measure the amount of compression as before; however, we also measure the amount of distortion introduced by the loss of information. The measure of distortion is generally some variant of the mean squared error. If at all possible, it is more useful to let the application define the distortion.

In this section we describe a number of compression techniques that allow loss of information, hence the name lossy compression. We begin with a look at quantization which, in one way or another, is at the heart of all lossy compression schemes.

A. Quantization

Quantization is the process of representing the output of a source with a large (possibly infinite) alphabet with a small alphabet. It is a many-to-one mapping and therefore irreversible. Quantization can be performed on a sample-by-sample basis or it can be performed on a group of samples. The former is called scalar quantization and the latter is called vector quantization. We look at each in turn.

1. Scalar Quantization

Let us, for the moment, assume that the output alphabet of the source is all or some portion of the real number line. Thus, the size of the source alphabet is infinite. We would like to represent the source output using a finite number of code words M. The quantizer consists of two processes: an encoding process that maps the output of the source into one of the M code words, and a decoding process that maps each code word into a reconstruction value. The encoding process can be viewed as a partition of the source alphabet, while the decoding process consists of obtaining representation values for each partition. An example for a quantizer with an alphabet size of four is shown in Fig. 9. If the quantizer output alphabet includes 0, the quantizer is called a *midtread* quantizer. Otherwise, it is called a *midrise* quantizer.

The simplest case is when the partitions are of the same size. Quantizers with same size partitions are called *uniform quantizers*. If the source is uniformly distributed between $-A$ and A, the size of the partition or quantization interval Δ is $2A/M$. The reconstruction values are located in the middle of each quanti-

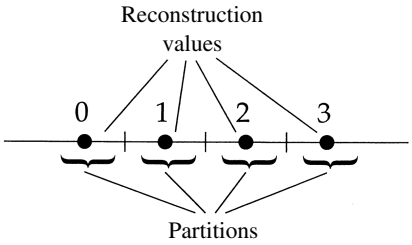

Figure 9 Quantizer with alphabet size of four.

zation interval. If we define the difference between the input x and the output $Q(x)$ to be the quantization noise, we can show that the variance of the quantization noise in this situation is $\Delta^2/12$. If the distribution of the source output is other than uniform, we can optimize the value of Δ for the particular distribution (Gersho and Gray, 1991; Max, 1960).

Often, the distribution of the quantizer input is a peaked distribution modeled as a Gaussian or Laplacian distribution. If the encoder is going to use a fixed length code, that is, each quantizer output is encoded using the same number of bits, we can get a lower average distortion if we use smaller partitions corresponding to the lower value inputs. Such a quantizer is called a *nonuniform quantizer* and is specified by the boundary values of the partition b_i and the reconstruction levels y_i. If we know the probability density function $f_X(x)$, the boundary and reconstruction values for an M-level quantizer which minimizes the mean squared error can be obtained by iteratively solving the following equations.

$$y_j = \frac{\int_{bj-1}^{bj} x f_X(x)\,dx\}}{\int_{bj-1}^{bj} fX(x)\,dx} \tag{4}$$

$$b_j = \frac{y_{j+1} + y_j}{2} \tag{5}$$

Rather than changing the size of the quantization intervals, we can also implement a nonuniform quantizer as shown in Fig. 10. The high-probability input region is "spread out" so as to make use of multiple quantization intervals. The mapping is reversed after the quantizer. For a companding function $c(x)$ and a source which lies between $\pm x_{max}$, the variance of the quantization noise is

$$\sigma_q^2 = \frac{x_{max}^2}{3M^2} \int_{-x_{max}}^{x_{max}} \frac{f_X(x)}{(c'(x))^2}\,dx. \tag{6}$$

If a variable length code such as a Huffman code or arithmetic coding is used to encode the output of the quantizer, Gish and Pierce (1968) showed that the optimum quantizer is a uniform quantizer which covers the entire range of the source output. If the range is large and our desired distortion is small, the number of quantizer levels can become quite large. In these situations we can use a quantizer with a limited output alphabet called a *recursively indexed quantizer* (Sayood and Na, 1992).

The JPEG algorithm uses a set of uniform scalar quantizers for quantizing the coefficients used to represent the image. The quantizer levels are then encoded using a variable length code.

2. Vector Quantization

The idea of representing groups of samples rather than individual samples has been present since Shannon's original papers (1948). There are several advantages to representing sequences. Consider the samples of the signal shown in Fig. 11. The values vary approximately between -4 and 4. We could quantize these samples with an eight-level scalar quantizer with $\Delta = 1$. So the reconstruction values would be $\{\pm\frac{1}{2}, \pm\frac{3}{2}, \pm\frac{5}{2}, \pm\frac{7}{2}\}$. If we were to use a fixed length code we would need three bits to represent the eight quantizer outputs. If we look at the output of the quantizer in pairs, we get 64 possible reconstruction values. These are represented by the larger filled circles in Fig. 12. However, if we plot the samples of the signal as pairs (as in Fig. 13) we see that the samples are clustered along a line in the first and third quadrants. This is due to the fact that there is a high degree of correlation between neighboring samples, which means that in two dimensions the samples will cluster

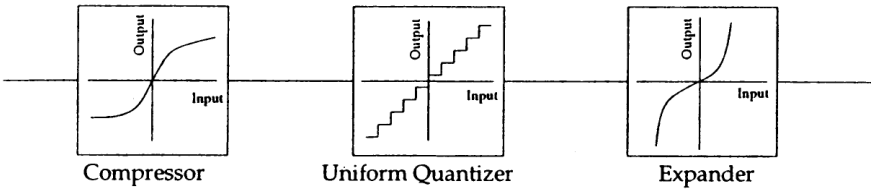

Figure 10 Companded quantization.

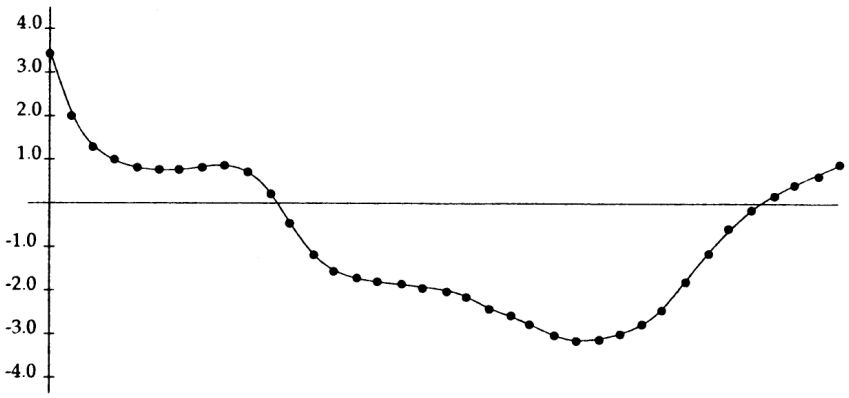

Figure 11 Samples of a signal.

around the $y = x$ line. Looking from a two-dimensional point of view, it makes much more sense to place all the 64 output points of the quantizer close to the $y = x$ line. For a fixed length encoding, we would need six bits to represent the 64 different quantizer outputs. As each quantizer output is a representation of two samples, we would end up with three bits per sample. Therefore, for the same number of bits, we would get a more accurate representation of the input and, therefore, incur less distortion. We pay for this decrease in distortion in several different ways. The first is through an increase in the complexity of the encoder. The scalar quantizer has a very simple encoder. In the case of the two-dimensional quantizer, we need to block the input samples into "vectors" and then compare them against all the possible quantizer output values. For three bits per sample and two dimensions this translates to 64 possible compares. However, for the same number of bits and a block size, or vector dimension, of 10, the number of quantizer outputs would be $2^{3\times10}$ which is 1,073,741,824! As it generally requires a large block size to get the full advantage of a vector quantizer, this means that the rate at which a vector quantizer (VQ) operates (i.e., bits per sample) is usually quite low.

The second way we pay is because of the fact that the quantizer becomes tuned to our assumptions about the source. For example, if we put all our quan-

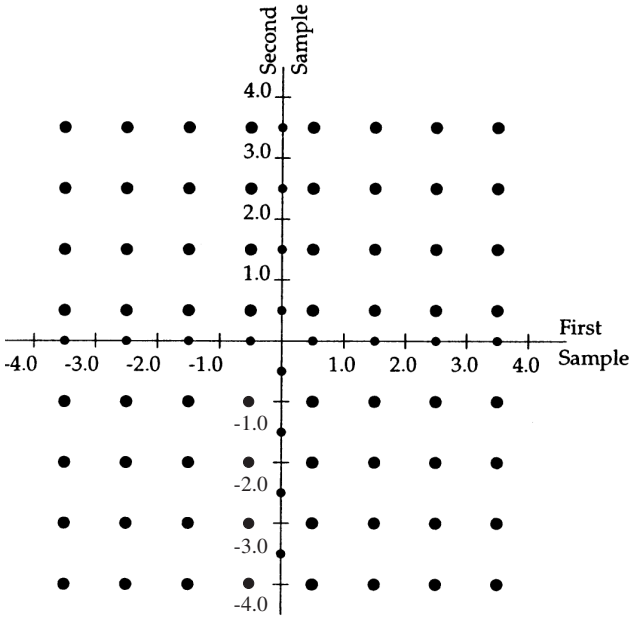

Figure 12 Two-dimensional view of an eight-level scalar quantizer.

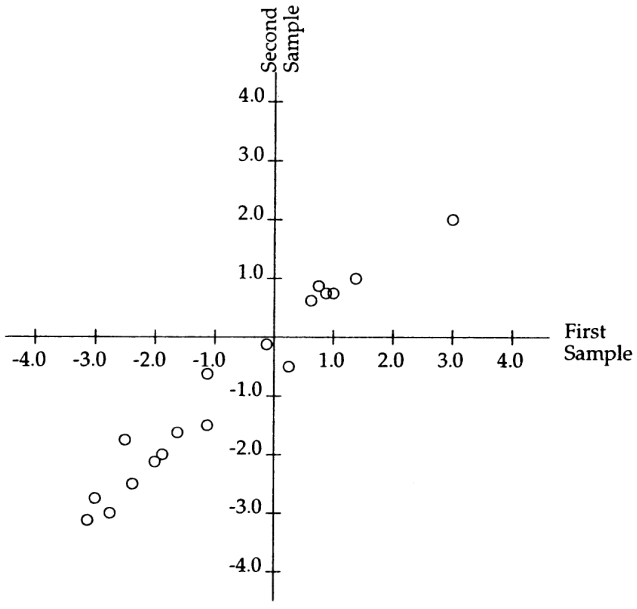

Figure 13 Two-dimensional view of an eight-level scalar quantizer.

tizer output values along the $y = x$ line and then we started getting input vectors which lay in the second or fourth quadrants, we would end up with substantial distortion in the reconstruction.

The operation of a VQ can be summarized as follows (see Fig. 14). Both the encoder and the decoder have copies of the VQ codebook. The codebook consists of the quantizer reconstruction vectors. The encoder blocks the input into an N sample vector and finds the vector in the codebook which is closest (usually in the Euclidean sense) to the input. The encoder then sends the index of the closest match. The decoder, upon receiving the index, performs a table lookup and obtains the reconstruction vector.

The VQ codebook is usually obtained using a clustering algorithm popularized by Linde, Buzo, and Gray (1980). It is generally referred to by the initials of the three authors (LBG). The LBG algorithm obtains a nearest neighbor partition of the source output space by making use of a training set. Selection of the training set can be an important aspect of the design of the VQ as it embodies our assumptions about the source output. Details of the LBG algorithm can be found in a number of places (see Gersho and Gray, 1991; Linde, Buzo, and Gray, 1980; Sayood, 2000).

There are a number of variations on the basic VQ described above. The most well known is the tree structured vector quantizer (TSVQ). Details on these can be found in Gersho and Gray (1991).

B. Predictive Coding

If we have a sequence with sample values that vary slowly as in the signal shown in Fig. 11, knowledge of the previous samples gives us a lot of information about the current sample. This knowledge can be used in a number of different ways. One of the earliest attempts at exploiting this redundancy was in the development of differential pulse code modulation (DPCM) (Cutler, 1952). A version of DPCM is the algorithm used in the International Telecommunication Union (ITU) standard G.726 for speech coding.

The DPCM system consists of two blocks as shown in Fig. 15. The function of the predictor is to obtain an estimate of the current sample based on the *reconstructed* values of the past sample. The difference between this estimate, or prediction, and the actual value is quantized, encoded, and transmitted to the receiver. The decoder generates an estimate identical to the encoder, which is then added on to generate the reconstructed value. The requirement that the prediction algorithm use only the reconstructed values is to ensure that the prediction at both the encoder and the decoder are identical. The reconstructed values used by the predictor, and the prediction algorithm, are dependent on the nature of the data being encoded. For example, for speech coding the predictor often uses the immediate past several values of the sequence, along with a sample that is a pitch period away, to form the prediction. In image compression the predictor may use the same set of pixels used by the JPEG-LS algorithm to form the prediction.

The predictor generally used in a DPCM scheme is a linear predictor. That is, the predicted value is obtained as a weighted sum of past reconstructed values.

$$p_n = \sum_{i \in \mathcal{I}} a_i x_i$$

where $\mathcal{I}$ is an index set corresponding to the samples to be used for prediction. The coefficients a_i are generally referred to as the predictor coefficients. If we assume the source output to be wide sense stationary, the predictor coefficients can be obtained as a solution of the discrete form of the Weiner-Hopf equations

$$A = R^{-1}P$$

where A is an $M \times 1$ vector of predictor coefficients, R is the $M \times M$ autocorrelation matrix of the set of

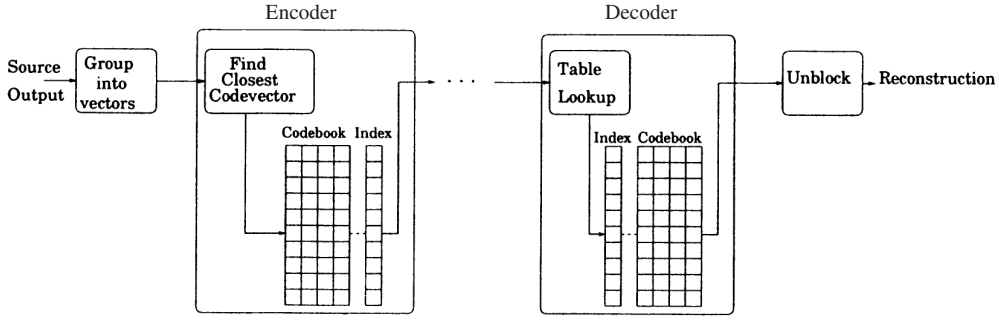

Figure 14 Operation of a vector quantizer.

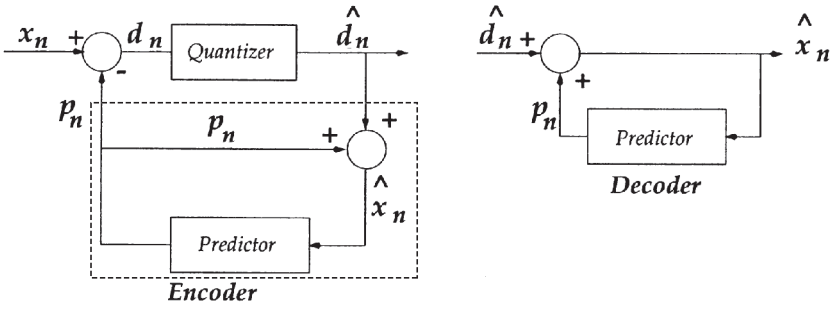

Figure 15 Block diagram of a DPCM system.

samples used to form the prediction, and P is the $M \times 1$ vector of autocorrelation coefficients between the elements of the index set and the value to be estimated.

Both the quantizer and the predictor in the DPCM system can be adaptive. The most common form of adaptivity for DPCM in speech coding is based on the reconstruction values. This allows both the encoder and the decoder, in the absence of channel errors, to adapt using the same information.

In Fig. 16 we show the results of encoding the *sensin* image using a simple fixed predictor and a recursively indexed quantizer with entropy coding.

C. Transform Coding

Transform coding first became popular in the early 1970s as a way of performing vector quantization (Huang and Schultheiss, 1963). Transform coding consists of three steps. The data to be compressed is divided into blocks, and the data in each block is transformed to a set of coefficients. The transform is selected to compact most of the energy into as few coefficients as possible. The coefficients are then quantized with different quantizers for each coefficient. Finally, the quantizer labels are encoded.

Transform coding generally involves linear transforms. Consider the situation where the data is blocked into vectors of length M. The transform coefficients can be obtained by multiplying the data with a transform matrix of dimension $M \times M$.

$$\theta = AX$$

where X is a vector of size $M \times 1$ containing the data, and θ is the $M \times 1$ vector of transform coefficients. The data can be recovered by taking the inverse transform:

$$X = A^{-1}\theta$$

In most transforms of interest the transform matrix is unitary, that is,

$$A^{-1} = A^T$$

If the data is stationary we can show that the optimum transform in terms of providing the most energy com-

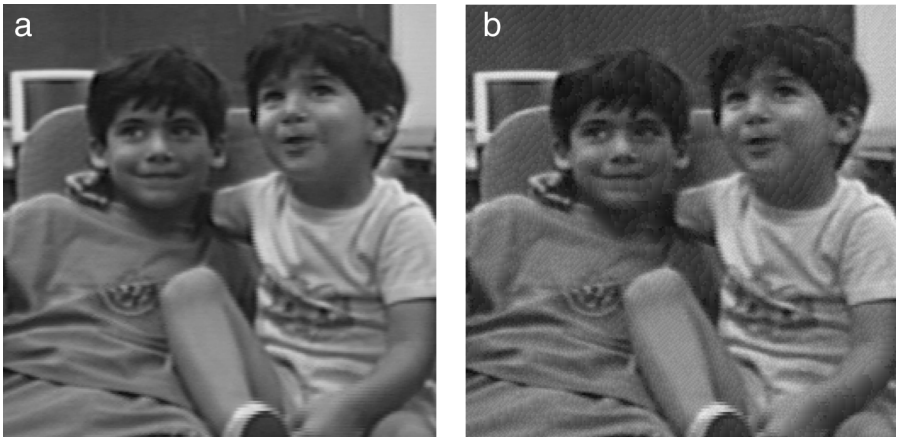

Figure 16 (a) The original *sensin* image and (b) the *sensin* image coded at one bit per pixel using DPCM and the recursively indexed quantizer.

paction is the Karhunen-Loeve transform (KLT). The KLT transform matrix is constructed from the eigenvectors of the autocorrelation matrix. Therefore, the transform is data dependent. Different sets of data might require different transform matrices. In practice, unless the size of the transform is small relative to the data, it generally costs too many bits to send the transform matrix to the decoder for the KLT to be feasible. The practical alternative to the KLT has been the discrete cosine transform (DCT). The DCT provides comparable compression to the KLT for most sources and has a fixed transform matrix whose elements are given by:

$$[\mathbf{A}]_{i,j} = \begin{cases} \sqrt{\frac{1}{M}} \cos \frac{(2j+1)i\pi}{2M} & i = 0,\ j = 0,1,\cdots,\ M-1 \\ \sqrt{\frac{2}{M}} \cos \frac{(2j+1)i\pi}{2M} & i = 1,2,\cdots,\ M-1,\ j = 0,1,\cdots,\ M-1 \end{cases} \tag{7}$$

The rows of the transform matrix are shown in graphical form in Fig. 17. We can see that the rows represent signals of increasing frequency. Thus, the DCT breaks the signal up into its frequency components. As most natural sources, such as speech, have higher low-frequency content the lower order coefficients usually contain most of the information about a particular block. We obtain compression by discarding those coefficients which contain little or no energy.

When coding two-dimensional sources such as images, most transform coding schemes use separable transforms. These are transforms which can be implemented by first taking the transform of the rows and then taking the transform of the columns (or vice versa). The popular JPEG image compression standard uses a separable 8×8 DCT as its transform. The image is divided into 8×8 blocks. These blocks are then transformed to obtain 8×8 blocks of coefficients.

The coefficients have different statistical characteristics and may be of differing importance to the end user. Therefore, they are quantized using different quantizers. In image compression the lower order coefficients are more important in terms of human perception and are therefore quantized using quantizers with smaller step sizes (and hence less quantization noise) than the higher order, higher frequency coefficients. A sample set of step sizes recommended by the JPEG committee (Pennebaker and Mitchell, 1993) is shown in Fig. 18.

Each quantized value $Q(\theta_{i,j})$ is represented by a label

$$l_{i,j} = \left\lfloor \frac{\theta_{ij}}{Q_{ij}} + .5 \right\rfloor$$

where Q_{ij} is the step size in the ith row and jth column and $\lfloor\ \rfloor$ indicates truncation to an integer value. The reconstruction is obtained from the label by multiplying it with the step size.

The final step involves coding the quantization levels. The JPEG image compression standard uses a

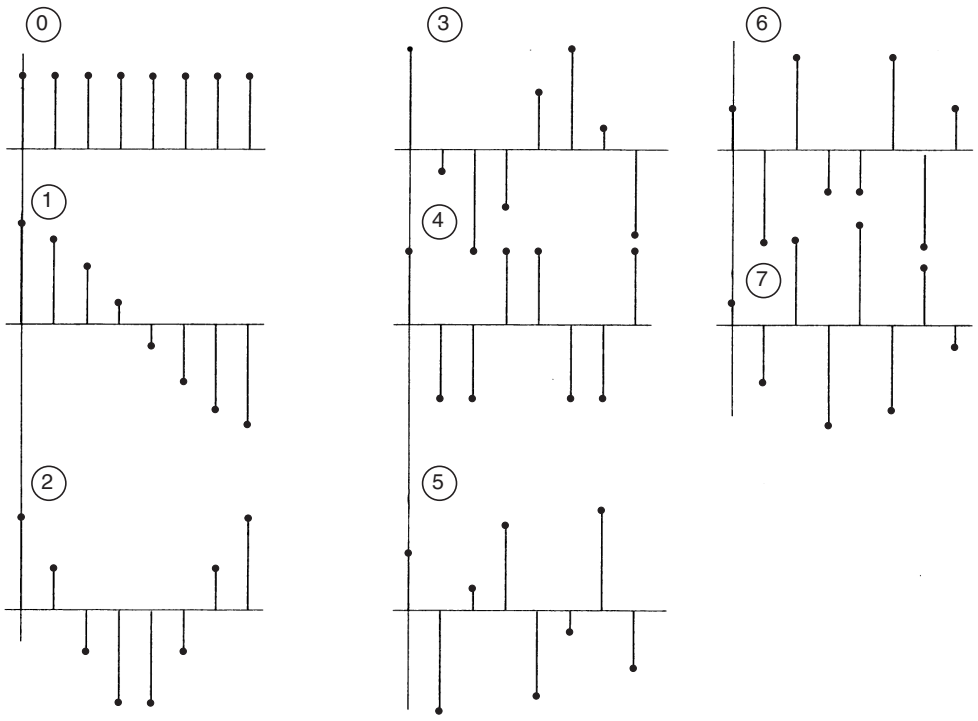

Figure 17 Basis set for the DCT. The numbers in the circles correspond to the row of the transform matrix.

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

Figure 18 Sample quantization table.

minor variation of a scheme by Chen and Pratt (1984). The lowest order coefficient, commonly referred to as the DC coefficient, has been found to be correlated from block to block. It is therefore coded differentially. The remaining coefficients are scanned in zigzag order as shown in Fig. 19. As most of the higher frequency coefficients are generally small and the step sizes used to quantize them are relatively large, most of the labels at the tail end of the scan are zero. We take advantage of this fact by sending an end of block symbol after the last nonzero label on the scan. Thus, a large number of coefficients are represented using a single code word. We represent the remaining coefficients using a code which is indexed by the magnitude of the coefficient and the number of zero valued labels preceding it. Details of the coding can be found in a number of places including Pennebaker and Mitchell (1993) and Sayood (2000).

In Fig. 20 we show the *sensin* image coded at 0.5 bits per pixel using JPEG.

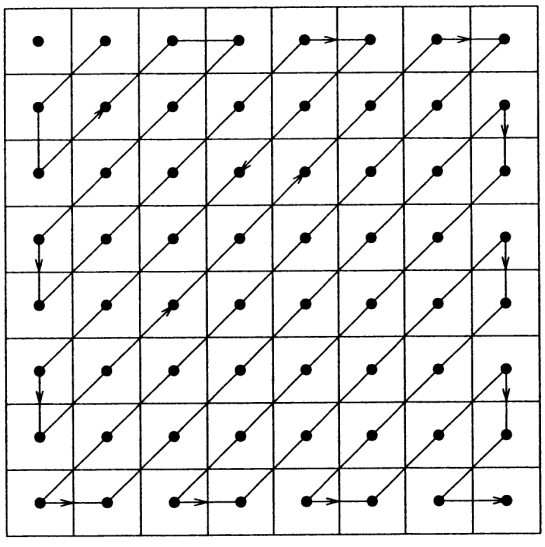

Figure 19 The zigzag scanning pattern for an 8×8 transform.

D. Subband/Wavelet Coding

Transform coding at low rates tends to give the reconstructed image a blocky appearance. The image in Fig. 21 has been coded at 0.25 bits per pixel using the JPEG algorithm. The blocky appearance is clearly apparent. This has led to the increasing popularity of subband and wavelet-based schemes. The implementation for both subband and wavelet-based schemes is similar. The input is filtered through a bank of filters, called the *analysis filterbank*. The filters cover the entire frequency range of the signal. As the bandwidth of each filter is only a fraction of the bandwidth of the original signal, the Nyquist criterion dictates that the number of samples required at the output of the filter be less than the number of samples per second required at the input of the filter. The output of the filters is subsampled or *decimated* and encoded. The decimated output values are quantized, and the quantization labels are encoded. At the decoder, after the received samples are decoded they are *upsampled* by the insertion of zeros between the received samples and filtered using a bank of reconstruction filters. A two-band subband coding scheme is shown in Fig. 22. The major components of the design of subband coding schemes are the selection of the filters and the encoding method used for the subbands. In order to determine the latter, it may be necessary to allocate a predetermined bit budget between the various bands.

Notice that in the system shown in Fig. 22 if the filters are not ideal filters then at least one of the two analysis filters will have a bandwidth greater than half the bandwidth of the source output. If the source is initially sampled at the Nyquist rate, then when we subsample by two that particular filter output will effectively be sampled at less the Nyquist rate, thus introducing aliasing. One of the objectives of the design of the analysis and synthesis filterbanks is to remove the effect of aliasing.

If we ignore the quantization and coding for the moment, we can determine conditions on the analysis and synthesis filterbanks such that the reconstruction is a delayed version of the source output. If we represent the samples of the source sequence by $x(n)$ and the reconstruction sequence by $\hat{x}(n)$, then this requirement known as the *perfect reconstruction* (*PR*) requirement can be written as

$$\hat{x}(n) = cx(n - n_0) \tag{8}$$

where c is a constant. In terms of the Z transforms of the sequences, we can write the PR requirement as

$$\hat{X}(z) = cz^{-n_0}X(z) \tag{9}$$

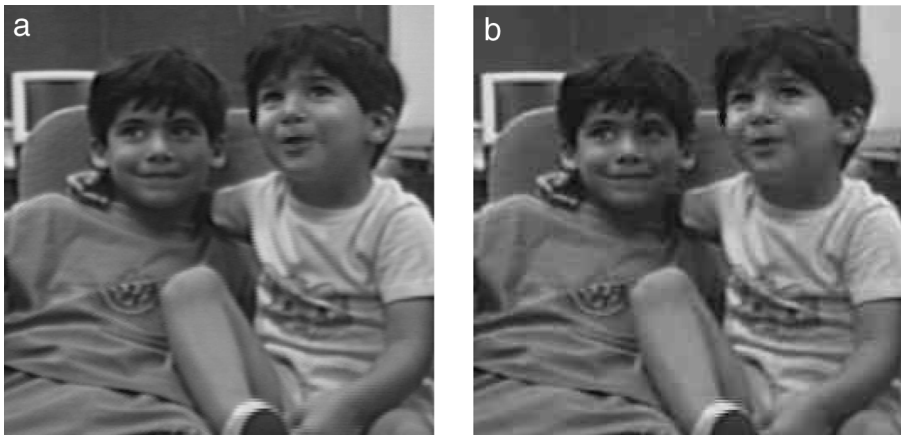

Figure 20 (a) The original *sensin* image and (b) the *sensin* image coded at 0.5 bits per pixel using JPEG.

Let $H_1(z)$ and $H_2(z)$ be the transfer functions for the analysis filters and $K_1(z)$ and $K_2(z)$ be the transfer functions of the synthesis filters. Then we can show (Sayood, 2000) that

$$X(z) = \frac{1}{2}[H_1(z)K_1(z) + H_2(z)K_2(z)]X(z) + \frac{1}{2}[H_1(-z)K_1(z) + H_2(-z)K_2(z)]X(-z) \quad (10)$$

Examining this equation we can see that in order for the perfect reconstruction condition to be satisfied, we need

$$H_1(-z)K_1(z) + H_2(-z)K_2(z) = 0 \quad (11)$$

$$H_1(z)K_1(z) + H_2(z)K_2(z) = cz^{-n_0} \quad (12)$$

The first equation is satisfied if we pick the synthesis filters as

$$K_1(z) = -H_2(z) \quad (13)$$

$$K_2(z) = H_1(-z) \quad (14)$$

To satisfy the second equation we can select $H_2(z)$ as (Mintzer, 1985; Smith and Barnwell, 1984)

$$H_2(z) = z^{-N}H_1(-z^{-1}) \quad (15)$$

Thus, all four filters can be expressed in terms of one prototype filter. We can show that this prototype filter should have an impulse response satisfying

$$\sum_{k=0}^{N} h_k h_{k+2n} = \delta_n \quad (16)$$

We can arrive at the same requirement on the filter coefficients using a wavelet formulation. Once we have obtained the coefficients for the filters, the compression approach using wavelets is similar. After the source output has been decomposed the next step is the quantization and coding of the coefficients. The two most popular approaches to quantization and coding for image compression are the embedded zerotree (EZW) (Shapiro, 1993) and the set partitioning in hierarchical trees (SPIHT) (Said and Pearlman, 1996) approaches.

Both these approaches make use of the fact that there is a relationship between the various subbands. A natural image is essentially a low-pass signal. Therefore, most of the energy in a wavelet or subband decomposition is concentrated in the LL band. One effect of this is that if the coefficient representing a particular pixel in the LL band is less than a specified threshold, the coefficients corresponding to the same pixel in the other bands will also have a magnitude smaller than that threshold. Thus, during coding we can scan the coefficients in the LL band first and compare them against a sequence of decreasing

Figure 21 The *sensin* image coded at 0.25 bits per pixel using JPEG.

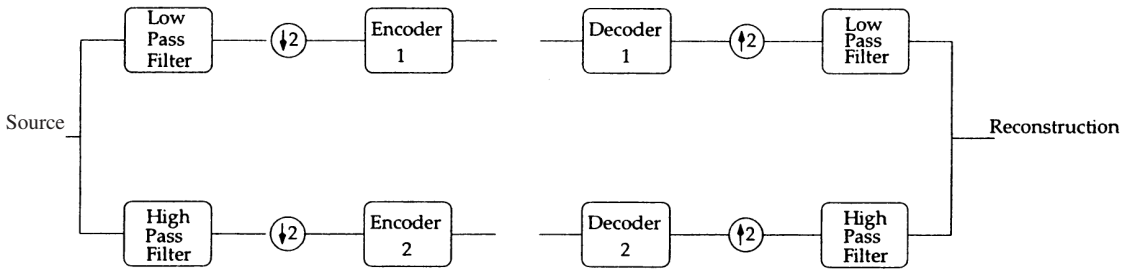

Figure 22 A two-band subband coding scheme.

thresholds. If the coefficient is less than the threshold we can check to see if the corresponding coefficients in the other bands are also less than this threshold. This information is then transmitted to the decoder. If the coefficients in the other band are also less than the threshold this is a highly efficient code. Note that the efficiency is dependent on the image being low pass. For more high-pass images, such as remotely sensed images, this strategy is not very effective.

In Fig. 23 we have the *sensin* image coded at rates of 0.5 bits per pixel and 0.25 bits per pixel using the SPIHT algorithm. Comparing the 0.25 bits per pixel reconstruction to Fig. 21 we can see the absence of blockiness. However, there are different artifacts that have taken the place of the blockiness. Neither reconstruction is very good at this rate.

A descendant of these techniques, known as EBCOT (Taubman, 2000), is the basis for the new JPEG 2000 image compression standard. Detailed information about these techniques can be found in Said and Pearlman (1996), Sayood (2000), and Shapiro (1993).

E. Analysis-Synthesis Schemes

When possible, one of the most effective means of compression is to transmit instructions on how to reconstruct the source rather than transmitting the source samples. In order to do this we should have a fairly good idea about how the source samples were generated. One particular source for which this is true is human speech.

Human speech can be modeled as the output of a linear filter which is excited by either white noise or a periodic input or a combination of the two. One of the earliest modern compression algorithms made use of this fact to provide a very high compression of speech. The technique, known as linear predictive coding, has its best known embodiment in the (now outdated) U.S. Government standard LPC-10. Some of the basic aspects of this standard are still alive, albeit in modified form in today's standards.

The LPC-10 standard assumes a model of speech pictured in Fig. 24. The speech is divided into frames. Each frame is classified as voiced or unvoiced. For the

Figure 23 The *sensin* image coded at (a) 0.5 bits per pixel and (b) 0.25 bits per pixel using SPIHT.

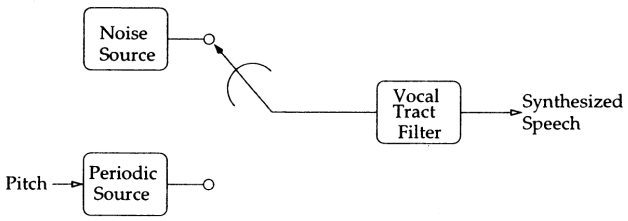

Figure 24 Speech synthesis model used by LPC-10.

voiced speech the pitch period for the speech sample is extracted. The parameters of the vocal tract filter are also extracted and quantized. All this information is sent to the decoder. The decoder synthesizes the speech samples y_n as

$$y_n = \sum_{i=1}^{M} b_i y_{n-i} + G\epsilon_n \tag{17}$$

where $\{b_i\}$ is the coefficient of the vocal tract filter. The input to the filter, the sequence $\{\epsilon_n\}$, is either the output of a noise generator or a periodic pulse train, where the period of the pulse train is the pitch period.

Since the introduction of the LPC-10 standard there has been a considerable increase in the sophistication of speech coders. In code excited linear prediction (CELP) the vocal tract filter is excited by elements of an excitation codebook. The entries of the codebook are used as input to a vocal filter of the form

$$y_n = \sum_{i=1}^{10} b_i y_{n-i} + \beta y_{n-P} + G\epsilon_n \tag{18}$$

where P is the pitch period. The synthesized speech is compared with the actual speech, and the codebook entry that provides the closest perceptual match is selected. The index for this entry is sent to the decoder along with the vocal tract filter parameters.

Mixed excitation linear prediction (MELP) uses a somewhat more complex approach to generating the excitation signal. The input is subjected to a multiband voicing analysis using five filters. The results of the analysis are used with a complex pitch detection strategy to obtain a rich excitation signal.

F. Video Compression

Currently, the source that requires the most resources in terms of bits and, therefore, has benefitted the most from compression is video. We can think of video as a sequence of images. With this view video compression becomes repetitive image compression and we can compress each frame separately. This is the point of view adopted by M-JPEG, or motion JPEG, in which each frame is compressed using the JPEG algorithm.

However, we know that in most video sequences there is a substantial amount of correlation between frames. It is much more efficient to send differences between the frames rather than the frames themselves. This idea is the basis for several international standards in video compression. In the following we briefly look at some of the compression algorithms used in these standards. Note that the standards contain much more than just the compression algorithms.

The ITU H.261 and its descendant ITU H.263[1] are international standards developed by the ITU, which is a part of the United Nations organization. A block diagram of the H.261 video coding algorithm is shown in Fig. 25. The image is divided into blocks of size 8 $\times$ 8. The previous frame is used to predict the values of the pixels in the block being encoded. As the objects in each frame may have been offset from the previous frame, the block in the identical location is not always used. Instead the block of size 8 $\times$ 8 in the previous frame which is closest to the block being encoded in the current frame is used as the predicted value. In order to reduce computations the search area for the closest match is restricted to lie within a prescribed region around the location of the block being encoded. This form of prediction is known as *motion compensated prediction*. The offset of the block used for prediction from the block being encoded is referred to as the *motion vector* and is transmitted to the decoder. The loop filter is used to prevent sharp transitions in the previous frame from generating high frequency components in the difference.

The difference is encoded using transform coding. The DCT is used followed by uniform quantization. The DC coefficient is quantized using a scalar quantizer with a step size of 8. The other coefficients are quantized with 1 of 31 other quantizers, all of which are midtread quantizers, with step sizes between 2 and 62. The selection of the quantizer depends in part on the availability of transmission resources. If higher compression is needed (fewer bits available), a larger step size is selected. If less compression is acceptable, a smaller step size is selected. The quantization labels are scanned in a zigzag fashion and encoded in a manner similar to (though not the same as) JPEG.

[1]Originally published in 1996; an update published in 1998 is commonly referred to as H.263+.

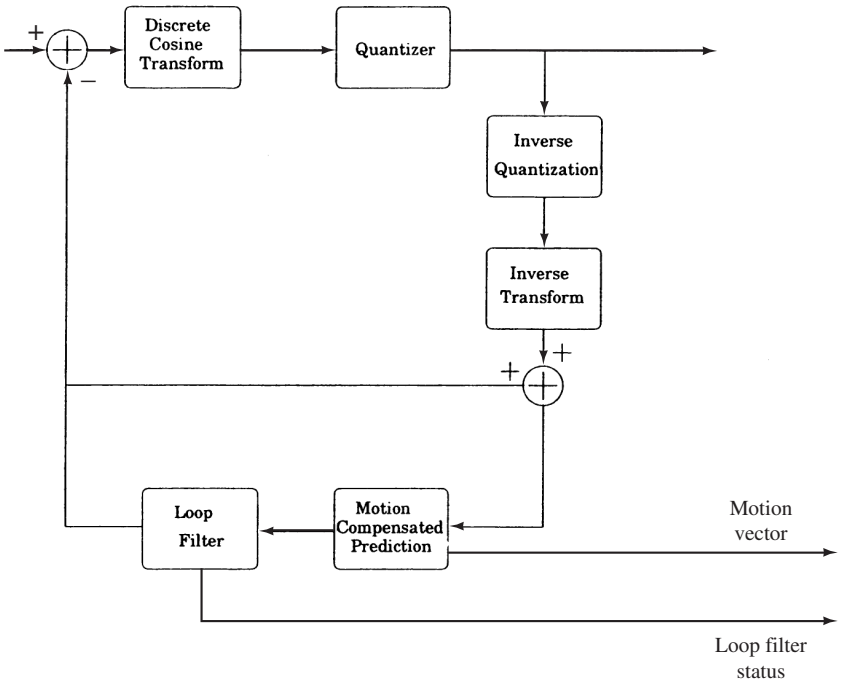

Figure 25 Block diagram of the ITU-T H.261 video compression algorithm.

The coding algorithm for ITU-T H.263 is similar to that used for H.261 with some improvements. The improvements include:

- Better motion compensated prediction
- Better coding
- Increased number of formats
- Increased error resilience

There are a number of other improvements that are not essential for the compression algorithm. As mentioned before, there are two versions of H.263. As the earlier version is a subset of the latter one; we only describe the later version.

The prediction is enhanced in H.263 in a number of ways. By interpolating between neighboring pixels in the frame being searched, a "larger image" is created. This essentially allows motion compensation displacement of half pixel steps rather than integer number of pixels. There is an unrestricted motion vector mode that allows references to areas outside the picture, where the outside areas are generated by duplicating the pixels at the image boundaries. Finally, in H.263+ the prediction can be generated by a frame that is not the previous frame. An independent segment decoding mode allows the frame to be broken into segments where each segment can be decoded independently. This prevents error propagation and also allows for greater control over quality of regions in the reconstruction. The H.263 standard also allows for bidirectional prediction.

As the H.261 algorithm was designed for videoconferencing, there was no consideration given to the need for random access. The MPEG standards incorporate this need by requiring that at fixed intervals a frame of an image be encoded without reference to past frames. The MPEG-1 standard defines three different kinds of frames: *I* frames, *P* frames, and *B* frames. An *I* frame is coded without reference to previous frames, that is, no use is made of prediction from previous frames. The use of the *I* frames allows random access. If such frames did not exist in the video sequence, then to view any given frame we would have to decompress all previous frames, as the reconstruction of each frame would be dependent on the prediction from previous frames. The use of periodic *I* frames is also necessary if the standard is to be used for compressing television programming. A viewer should have the ability to turn on the television (and the MPEG decoder) at more or less any time during the broadcast. If there are periodic *I* frames available, then the decoder can start decoding from the first *I* frame. Without the redundancy removal provided via prediction the compression obtained with *I* frames is less than would have been possible if prediction had been used.

The *P* frame is similar to frames in the H.261 standard in that it is generated based on prediction from previous reconstructed frames. The similarity is closer to H.263, as the MPEG-1 standard allows for half pixel shifts during motion compensated prediction.

The *B* frame was introduced in the MPEG-1 standard to offset the loss of compression efficiency occasioned by the *I* frames. The *B* frame is generated using prediction from the previous *P* or *I* frame and the nearest future *P* or *I* frame. This results in extremely good prediction and a high level of compression. The *B* frames are not used to predict other frames, therefore, the *B* frames can tolerate more error. This also permits higher levels of compression.

The various frames are organized together in a *group of pictures* (GOP). A GOP is the smallest random access unit in the video sequence. Therefore, it has to contain at least one *I* frame. Furthermore, the first *I* frame in a GOP is either the first frame of the GOP or is preceded by *B* frames which use motion compensated prediction only from this *I* frame. A possible GOP is shown in Fig. 26. Notice that in order to reconstruct frames 2, 3, and 4, which are *B* frames, we need to have the *I* and *P* frames. Therefore, the order in which these frames are transmitted is different from the order in which they are displayed.

The MPEG-2 standard extends the MPEG-1 standard to higher bit rates, bigger picture sizes, and interlaced frames. Where MPEG-1 allows half pixel displacements, the MPEG-2 standard requires half pixel displacements for motion compensation. Furthermore, the MPEG-2 standard contains several additional modes of prediction. A full description of any of these standards is well beyond the scope of this article. For details the readers are referred to the standards ISO/IEC IS 11172, 13818, and 14496 and books Gibson *et al.* (1998), Mitchell *et al.* (1997), and Sayood (2000).

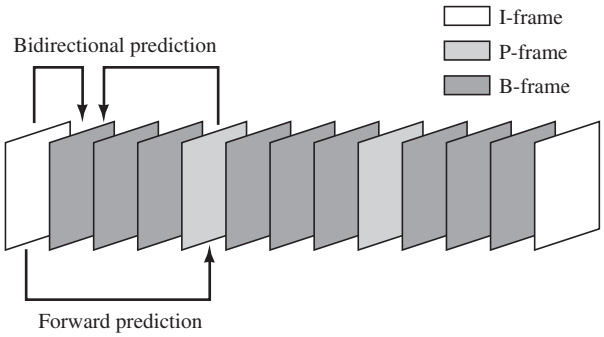

Figure 26 A possible arrangement for a GOP.

IV. FURTHER INFORMATION

We have described a number of compression techniques. How they compare relative to each other depends on the performance criteria, which in turn depend on the application. An excellent resource for people interested in comparisons between the multitude of compression programs available for different applications is the *Archive Compression Test* maintained by Jeff Gilchrist at http://act.by.net. Another excellent resource on the Internet is the data compression page maintained by Mark Nelson at http://dogma.net/DataCompression/. This page contains links to many other data compression resources, including programs and a set of informative articles by Mark Nelson. Programs implementing some of the techniques described here can also be obtained at ftp://ftp.mkp.com/pub/Sayood/.

SEE ALSO THE FOLLOWING ARTICLES

Desktop Publishing • Electronic Data Interchange • Error Detecting and Correcting Codes • Multimedia

BIBLIOGRAPHY

Burrows, M., and Wheeler, D. J. (1994). A Block Sorting Data Compression Algorithm. Technical Report SRC 124, Digital Systems Research Center.

Chen, W.-H., and Pratt, W. K. (March 1984). Scene Adaptive Coder. IEEE Trans. Communications. COM-32:225–232.

Cleary, J. G., and Witten, I. H. (1984). Data compression using adaptive coding and partial string matching. IEEE Trans. Communications. 32(4):396–402.

Cleary, J. G., and Teahan, W. J. (February 1997). Unbounded length contexts for PPM. Computer Journal, 40:30–74.

Cutler, C. C. (July 29, 1952). Differential Quantization for Television Signals. U.S. Patent 2 605 361.

Gallagher, R. G. (November 1978). Variations on a theme by Huffman. IEEE Trans. Information Theory. IT-24(6):668–674.

Gersho, A., and Gray, R. M. (1991). *Vector Quantization and Signal Compression.* Dordrecht/Norwell, MA: Kluwer Academic.

Gibson, J. D., Berger, T., Lookabaugh, T., Lindbergh, D., and Baker, R. (1998). *Digital Compression for Multimedia: Principles and Standards.* San Mateo, CA: Morgan Kaufmann.

Gish, H., and Pierce, J. N. (September 1968). Asymptotically efficient quantization. IEEE Trans. Information Theory. IT-14:676–683.

Huang, J.-Y., and Schultheiss, P. M. (September 1963). Block quantization of correlated gaussian random variables. IEEE Trans. Communication Systems. CS-11:289–296.

Huffman, D. A. (1951). A method for the construction of minimum redundancy codes. Proc. IRE. 40:1098–1101.

ISO/IECIS 11172. Information Technology—Coding of Moving

Pictures and Associated Audio for Digital Storage Media Up To About 1.5 Mbits/s.

ISO/IECIS 13818. Information Technology—Generic Coding of Moving Pictures and Associated Audio Information.

ISO/IECIS 14496. Coding of Moving Pictures and Audio.

Linde, Y., Buzo, A., and Gray, R. M. (January 1980). An algorithm for vector quantization design. IEEE Trans. Communications. COM-28:84–95.

Max, J. (January 1960). Quantizing for minimum distortion. IRE Trans. Information Theory. IT-6:7–12.

Memon, N. D., and Sayood, K. (1995). Lossless Image Compression: A Comparitive Study. In *Proceedings SPIE Conference on Electronic Imaging*. SPIE.

Mintzer, F. (June 1985). Filters for distortion-free two-band multirate filter banks. *IEEE Trans. Acoustics, Speech, and Signal Processing*. ASSP-33:626–630.

Mitchell, J. L., Pennebaker, W. B., Fogg, C. E., and LeGall, D. J. (1997). *MPEG Video Compression Standard*. London/New York: Chapman & Hall.

Nelson, M. (September 1996). Data compression with the Burrows-Wheeler transform. Dr. Dobbs Journal.

Nelson, M., and Gailly, J.-L. (1996). *The Data Compression Book*. California: M&T Books.

Pasco, R. (1976). Source Coding Algorithms for Fast Data Compression. Ph.D. thesis, Stanford University.

Pennebaker, W. B., and Mitchell, J. L. (1993). *JPEG Still Image Data Compression Standard*. New York: Van Nostrand-Reinhold.

Rissanen, J. J. (May 1976). Generalized Kraft inequality and arithmetic coding. IBM J. Research and Development. 20:198–203.

Said, A., and Pearlman, W. A. (June 1996). A new fast and efficient coder based on set partitioning in hierarchical trees. *IEEE Trans. Circuits and Systems for Video Technologies*. 243–250.

Sayood, K. (2000). *Introduction to Data Compression, Second Edition*. San Mateo, CA: Morgan Kauffman/Academic Press.

Sayood, K., and Na, S. (November 1992). Recursively indexed quantization of memoryless sources. IEEE Trans. Information Theory. IT-38:1602–1609.

Shannon, C. E. (1948). A mathematical theory of communication. Bell System Technical J. 27:379–423,623–656.

Shapiro, J. M. (December 1993). Embedded image coding using zerotrees of wavelet coefficients. IEEE Trans. Signal Processing. SP-41:3445–3462.

Smith, M. J. T., and Barnwell, T. P., III. (1984). A Procedure for Designing Exact Reconstruction Filter Banks for Tree Structured Subband Coders. In *Proceedings IEEE International Conference on Acoustics Speech and Signal Processing*. IEEE.

Storer, J. A., and Syzmanski, T. G. (1982). Data compression via textual substitution. J. ACM. 29:928–951.

Taubman, D. (July 2000). High performance scalable image motion compression with EBCOT. IEEE Trans. Image Processing. IP-9:1158–1170.

Weinberger, M., Seroussi, G., and Sapiro, G. (November 1998). The LOCO-I Lossless Compression Algorithm: Principles and Standardization into JPEG-LS. Technical Report HPL-98-193, Hewlett-Packard Laboratory.

Weinberger, M. J., Rissanen, J. J., and Arps, R. (1995). Applications of Universal Context Modeling to Lossless Compression of Gray-Scale Images. In *Proc. Asilomar Conference on Signals, Systems and Computers,* pp. 229–233. IEEE.

Welch, T. A. (June 1984). A technique for high-performance data compression. IEEE Computer. 8–19.

Wu, X., and Memon, N. D. (May 1996). CALIC—A context based adaptive lossless image coding scheme. IEEE Trans. Communications.

Ziv, J., and Lempel, A. (May 1977). A universal algorithm for data compression. IEEE Trans. Information Theory. IT-23(3):337–343.

Ziv, J., and Lempel, A. (September 1978). Compression of individual sequences via variable-rate coding. IEEE Trans. Information Theory. IT-24(5):530–536.