10

Vector Quantization

10.1 Overview

By grouping source outputs together and encoding them as a single block, we can obtain efficient lossy as well as lossless compression algorithms. Many of the lossless compression algorithms that we looked at took advantage of this fact. We can do the same with quantization. In this chapter, several quantization techniques that operate on blocks of data are described. We can view these blocks as vectors, hence the name "vector quantization." We will describe several different approaches to vector quantization. We will explore how to design vector quantizers and how these quantizers can be used for compression.

10.2 Introduction

In the last chapter, we looked at different ways of quantizing the output of a source. In all cases the quantizer inputs were scalar values, and each quantizer codeword represented a single sample of the source output. In Chapter 2 we saw that, by taking longer and longer sequences of input samples, it is possible to extract the structure in the source coder output. In Chapter 4 we saw that, even when the input is random, encoding sequences of samples instead of encoding individual samples separately provides a more efficient code. Encoding sequences of samples is more advantageous in the lossy compression framework as well. By "advantageous" we mean a lower distortion for a given rate, or a lower rate for a given distortion. As in the previous chapter, by "rate" we mean the average number of bits per input sample, and the measures of distortion will generally be the mean squared error and the signal-to-noise ratio.

The idea that encoding sequences of outputs can provide an advantage over the encoding of individual samples was first put forward by Shannon, and the basic results in information

theory were all proved by taking longer and longer sequences of inputs. This indicates that a quantization strategy that works with sequences or blocks of output would provide some improvement in performance over scalar quantization. In other words, we wish to generate a representative set of sequences. Given a source output sequence, we would represent it with one of the elements of the representative set.

In vector quantization we group the source output into blocks or vectors. For example, we can treat L consecutive samples of speech as the components of an L-dimensional vector. Or, we can take a block of L pixels from an image and treat each pixel value as a component of a vector of size or dimension L. This vector of source outputs forms the input to the vector quantizer. At both the encoder and decoder of the vector quantizer, we have a set of L-dimensional vectors called the *codebook* of the vector quantizer. The vectors in this codebook, known as *code-vectors*, are selected to be representative of the vectors we generate from the source output. Each code-vector is assigned a binary index. At the encoder, the input vector is compared to each code-vector in order to find the code-vector closest to the input vector. The elements of this code-vector are the quantized values of the source output. In order to inform the decoder about which code-vector was found to be the closest to the input vector, we transmit or store the binary index of the code-vector. Because the decoder has exactly the same codebook, it can retrieve the code-vector given its binary index. A pictorial representation of this process is shown in Figure 10.1.

Although the encoder may have to perform a considerable amount of computations in order to find the closest reproduction vector to the vector of source outputs, the decoding consists of a table lookup. This makes vector quantization a very attractive encoding scheme for applications in which the resources available for decoding are considerably less than the resources available for encoding. For example, in multimedia applications, considerable

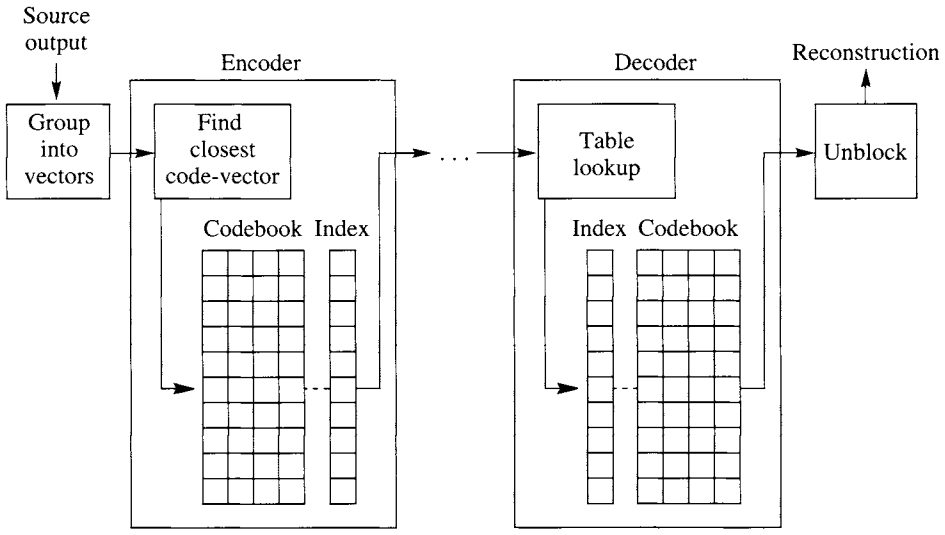

FIGURE 10. 1 **The vector quantization procedure.**

computational resources may be available for the encoding operation. However, if the decoding is to be done in software, the amount of computational resources available to the decoder may be quite limited.

Even though vector quantization is a relatively new area, it has developed very rapidly, and now even some of the subspecialties are broad areas of research. If this chapter we will try to introduce you to as much of this fascinating area as we can. If your appetite is whetted by what is available here and you wish to explore further, there is an excellent book by Gersho and Gray [5] devoted to the subject of vector quantization.

Our approach in this chapter is as follows: First, we try to answer the question of why we would want to use vector quantization over scalar quantization. There are several answers to this question, each illustrated through examples. In our discussion, we assume that you are familiar with the material in Chapter 9. We will then turn to one of the most important elements in the design of a vector quantizer, the generation of the codebook. While there are a number of ways of obtaining the vector quantizer codebook, most of them are based on one particular approach, popularly known as the Linde-Buzo-Gray (LBG) algorithm. We devote a considerable amount of time in describing some of the details of this algorithm. Our intent here is to provide you with enough information so that you can write your own programs for design of vector quantizer codebooks. In the software accompanying this book, we have also included programs for designing codebooks that are based on the descriptions in this chapter. If you are not currently thinking of implementing vector quantization routines, you may wish to skip these sections (Sections 10.4.1 and 10.4.2). We follow our discussion of the LBG algorithm with some examples of image compression using codebooks designed with this algorithm, and then with a brief sampling of the many different kinds of vector quantizers. Finally, we describe another quantization strategy, called trellis-coded quantization (TCQ), which, though different in implementation from the vector quantizers, also makes use of the advantage to be gained from operating on sequences.

Before we begin our discussion of vector quantization, let us define some of the terminology we will be using. The amount of compression will be described in terms of the rate, which will be measured in bits per sample. Suppose we have a codebook of size K, and the input vector is of dimension L. In order to inform the decoder of which code-vector was selected, we need to use $\lceil \log_2 K \rceil$ bits. For example, if the codebook contained 256 code-vectors, we would need 8 bits to specify which of the 256 code-vectors had been selected at the encoder. Thus, the number of bits *per vector* is $\lceil \log_2 K \rceil$ bits. As each code-vector contains the reconstruction values for L source output samples, the number of bits *per sample* would be $\frac{\lceil \log_2 K \rceil}{L}$. Thus, the rate for an L-dimensional vector quantizer with a codebook of size K is $\frac{\lceil \log_2 K \rceil}{L}$. As our measure of distortion we will use the mean squared error. When we say that in a codebook $\mathcal{C}$, containing the K code-vectors $\{Y_i\}$, the input vector X is closest to Y_j, we will mean that

$$\|X - Y_j\|^2 \leq \|X - Y_i\|^2 \qquad \text{for all} Y_i \in \mathcal{C} \tag{10.1}$$

where $X = (x_1 x_2 \cdots x_L)$ and

$$\|X\|^2 = \sum_{i=1}^{L} x_i^2. \tag{10.2}$$

The term *sample* will always refer to a scalar value. Thus, when we are discussing compression of images, a sample refers to a single pixel. Finally, the output points of the quantizer are often referred to as *levels*. Thus, when we wish to refer to a quantizer with K output points or code-vectors, we may refer to it as a K-level quantizer.

10.3 Advantages of Vector Quantization over Scalar Quantization

For a given rate (in bits per sample), use of vector quantization results in a lower distortion than when scalar quantization is used at the same rate, for several reasons. In this section we will explore these reasons with examples (for a more theoretical explanation, see [3, 4, 17]).

If the source output is correlated, vectors of source output values will tend to fall in clusters. By selecting the quantizer output points to lie in these clusters, we have a more accurate representation of the source output. Consider the following example.

Example 10.3.1:

In Example 8.5.1, we introduced a source that generates the height and weight of individuals. Suppose the height of these individuals varied uniformly between 40 and 80 inches, and the weight varied uniformly between 40 and 240 pounds. Suppose we were allowed a total of 6 bits to represent each pair of values. We could use 3 bits to quantize the height and 3 bits to quantize the weight. Thus, the weight range between 40 and 240 pounds would be divided into eight intervals of equal width of 25 and with reconstruction values $\{52, 77, \ldots, 227\}$. Similarly, the height range between 40 and 80 inches can be divided into eight intervals of width five, with reconstruction levels $\{42, 47, \ldots, 77\}$. When we look at the representation of height and weight separately, this approach seems reasonable. But let's look at this quantization scheme in two dimensions. We will plot the height values along the x-axis and the weight values along the y-axis. Note that we are not changing anything in the quantization process. The height values are still being quantized to the same eight different values, as are the weight values. The two-dimensional representation of these two quantizers is shown in Figure 10.2.

From the figure we can see that we effectively have a quantizer output for a person who is 80 inches (6 feet 8 inches) tall and weighs 40 pounds, as well as a quantizer output for an individual whose height is 42 inches but weighs more than 200 pounds. Obviously, these outputs will never be used, as is the case for many of the other outputs. A more sensible approach would be to use a quantizer like the one shown in Figure 10.3, where we take account of the fact that the height and weight are correlated. This quantizer has exactly the same number of output points as the quantizer in Figure 10.2; however, the output points are clustered in the area occupied by the input. Using this quantizer, we can no longer quantize the height and weight separately. We have to consider them as the coordinates of a point in two dimensions in order to find the closest quantizer output point. However, this method provides a much finer quantization of the input.

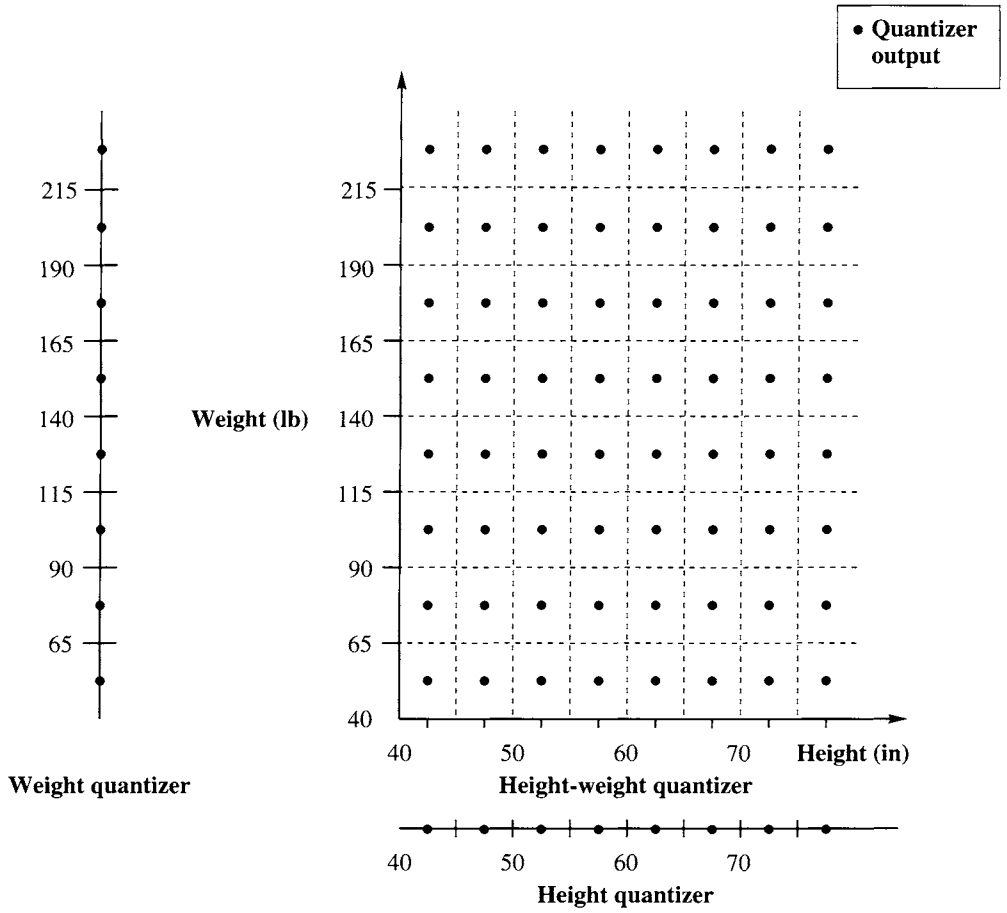

FIGURE 10.2 **The height/weight scalar quantizers when viewed in two dimensions.**

Note that we have not said how we would obtain the locations of the quantizer outputs shown in Figure 10.3. These output points make up the codebook of the vector quantizer, and we will be looking at codebook design in some detail later in this chapter. ♦

We can see from this example that, as in lossless compression, looking at longer sequences of inputs brings out the structure in the source output. This structure can then be used to provide more efficient representations.

We can easily see how structure in the form of correlation between source outputs can make it more efficient to look at sequences of source outputs rather than looking at each sample separately. However, the vector quantizer is also more efficient than the scalar quantizer when the source output values are not correlated. The reason for this is actually

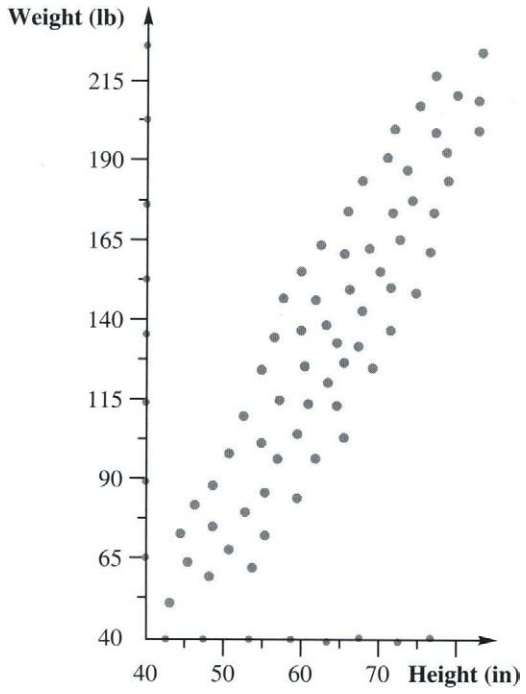

FIGURE 10.3 The height-weight vector quantizer.

quite simple. As we look at longer and longer sequences of source outputs, we are afforded more flexibility in terms of our design. This flexibility in turn allows us to match the design of the quantizer to the source characteristics. Consider the following example.

Example 10.3.2:

Suppose we have to design a uniform quantizer with eight output values for a Laplacian input. Using the information from Table 9.3 in Chapter 9, we would obtain the quantizer shown in Figure 10.4, where Δ is equal to 0.7309. As the input has a Laplacian distribution, the probability of the source output falling in the different quantization intervals is not the same. For example, the probability that the input will fall in the interval $[0, \Delta)$ is 0.3242, while the probability that a source output will fall in the interval $[3\Delta, \infty)$ is 0.0225. Let's look at how this quantizer will quantize two consecutive source outputs. As we did in the previous example, let's plot the first sample along the x-axis and the second sample along the y-axis. We can represent this two-dimensional view of the quantization process as shown in Figure 10.5. Note that, as in the previous example, we have not changed the quantization process; we are simply representing it differently. The first quantizer input, which we have represented in the figure as x_1, is quantized to the same eight possible output values as before. The same is true for the second quantizer input, which we have represented in the

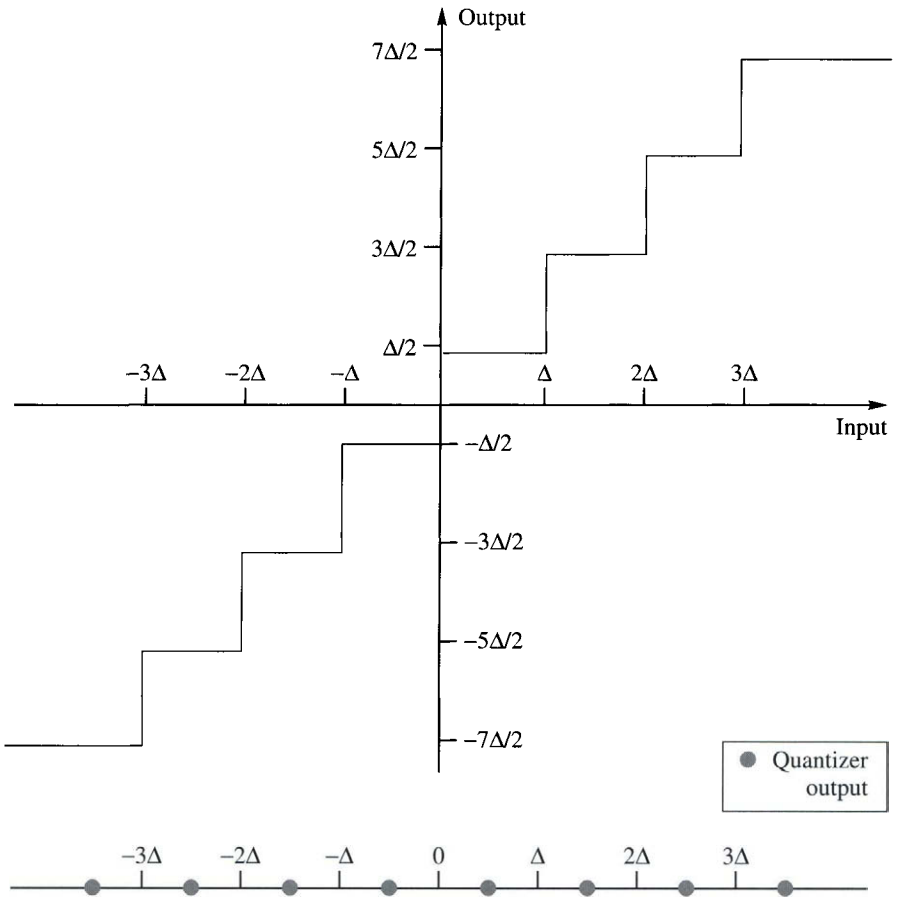

FIGURE 10.4 **Two representations of an eight-level scalar quantizer.**

figure as x_2. This two-dimensional representation allows us to examine the quantization process in a slightly different manner. Each filled-in circle in the figure represents a sequence of two quantizer outputs. For example, the top rightmost circle represents the two quantizer outputs that would be obtained if we had two consecutive source outputs with a value greater than 3Δ. We computed the probability of a single source output greater than 3Δ to be 0.0225. The probability of two consecutive source outputs greater than 2.193 is simply $0.0225 \times 0.0225 = 0.0005$, which is quite small. Given that we do not use this output point very often, we could simply place it somewhere else where it would be of more use. Let us move this output point to the origin, as shown in Figure 10.6. We have now modified the quantization process. Now if we get two consecutive source outputs with values greater than 3Δ, the quantizer output corresponding to the second source output may not be the same as the first source output.

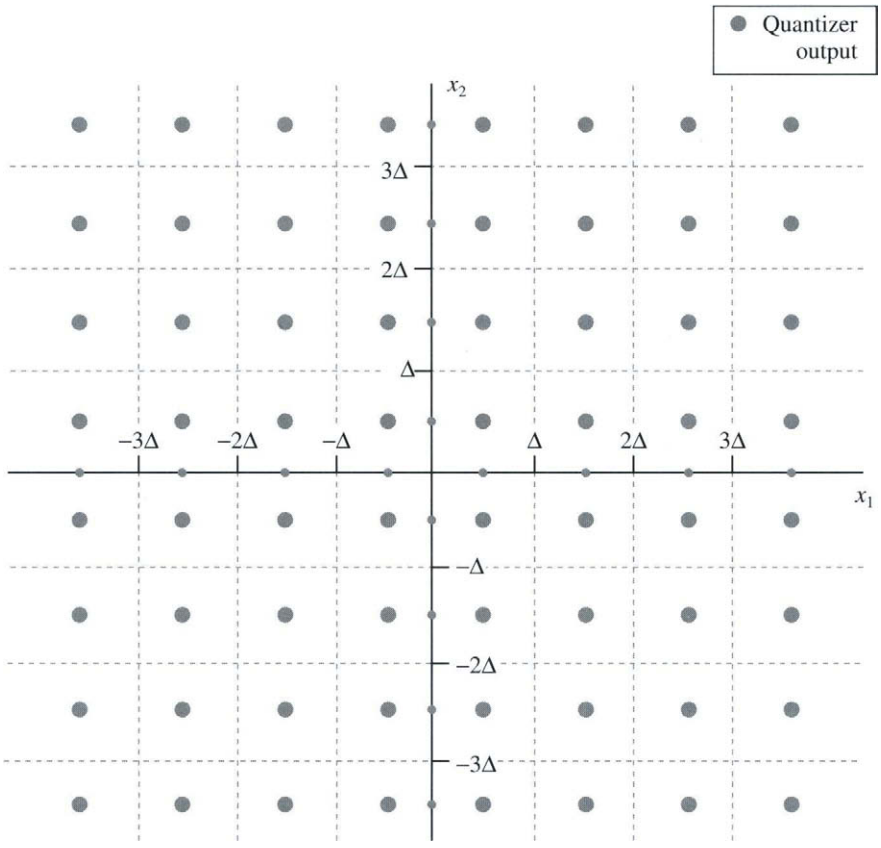

FIGURE 10.5 **Input-output map for consecutive quantization of two inputs using an eight-level scalar quantizer.**

If we compare the rate distortion performance of the two vector quantizers, the SNR for the first vector quantizer is 11.44 dB, which agrees with the result in Chapter 9 for the uniform quantizer with a Laplacian input. The SNR for the modified vector quantizer, however, is 11.73 dB, an increase of about 0.3 dB. Recall that the SNR is a measure of the average squared value of the source output samples and the mean squared error. As the average squared value of the source output is the same in both cases, an increase in SNR means a decrease in the mean squared error. Whether this increase in SNR is significant will depend on the particular application. What is important here is that by treating the source output in groups of two we could effect a positive change with only a minor modification. We could argue that this modification is really not that minor since the uniform characteristic of the original quantizer has been destroyed. However, if we begin with a nonuniform quantizer and modify it in a similar way, we get similar results.

Could we do something similar with the scalar quantizer? If we move the output point at $\frac{7\Delta}{2}$ to the origin, the SNR *drops* from 11.44 dB to 10.8 dB. What is it that permits us to make

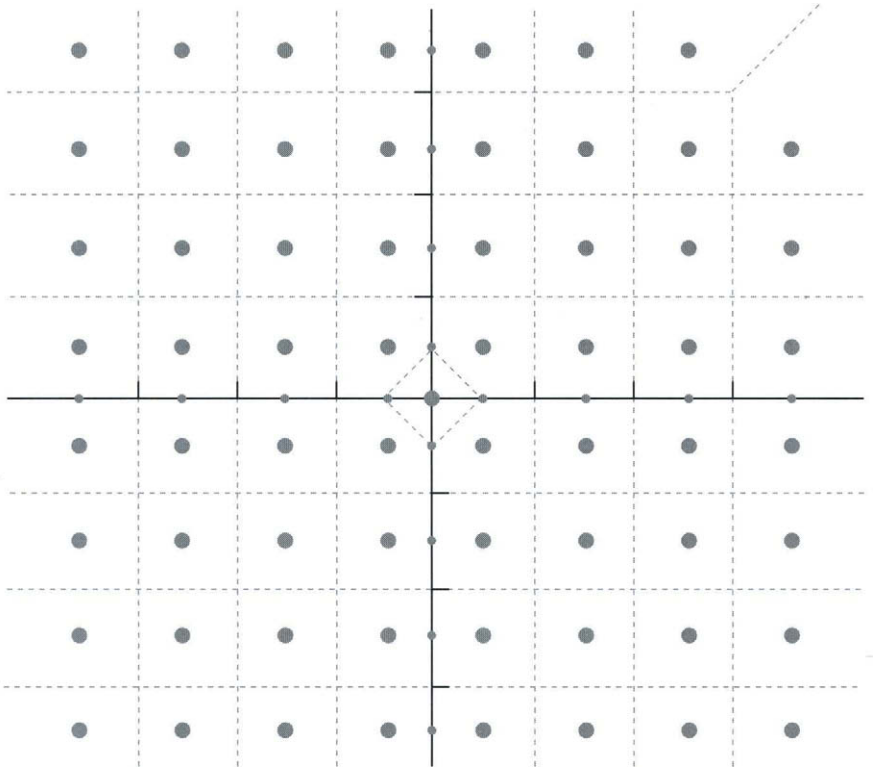

FIGURE 10.6 Modified two-dimensional vector quantizer.

modifications in the vector case, but not in the scalar case? This advantage is caused by the added flexibility we get by viewing the quantization process in higher dimensions. Consider the effect of moving the output point from $\frac{7\Delta}{2}$ to the origin in terms of two consecutive inputs. This one change in one dimension corresponds to moving 15 output points in two dimensions. Thus, modifications at the scalar quantizer level are gross modifications when viewed from the point of view of the vector quantizer. Remember that in this example we have only looked at two-dimensional vector quantizers. As we block the input into larger and larger blocks or vectors, these higher dimensions provide even greater flexibility and the promise of further gains to be made. ♦

In Figure 10.6, notice how the quantization regions have changed for the outputs around the origin, as well as for the two neighbors of the output point that were moved. The decision boundaries between the reconstruction levels can no longer be described as easily as in the case for the scalar quantizer. However, if we know the distortion measure, simply knowing the output points gives us sufficent information to implement the quantization

process. Instead of defining the quantization rule in terms of the decision boundary, we can define the quantization rule as follows:

$$Q(X) = Y_j \quad \text{iff} \quad d(X, Y_j) < d(X, Y_i) \ \forall i \neq j. \tag{10.3}$$

For the case where the input X is equidistant from two output points, we can use a simple tie-breaking rule such as "use the output point with the smaller index." The quantization regions V_j can then be defined as

$$V_j = \{X : d(X, Y_j) < d(X, Y_i) \ \forall i \neq j\}. \tag{10.4}$$

Thus, the quantizer is completely defined by the output points and a distortion measure.

From a multidimensional point of view, using a scalar quantizer for each input restricts the output points to a rectangular grid. Observing several source output values at once allows us to move the output points around. Another way of looking at this is that in one dimension the quantization intervals are restricted to be intervals, and the only parameter that we can manipulate is the size of these intervals. When we divide the input into vectors of some length n, the quantization regions are no longer restricted to be rectangles or squares. We have the freedom to divide the range of the inputs in an infinite number of ways.

These examples have shown two ways in which the vector quantizer can be used to improve performance. In the first case, we exploited the sample-to-sample dependence of the input. In the second case, there was no sample-to-sample dependence; the samples were independent. However, looking at two samples together still improved performance.

These two examples can be used to motivate two somewhat different approaches toward vector quantization. One approach is a pattern-matching approach, similar to the process used in Example 10.3.1, while the other approach deals with the quantization of random inputs. We will look at both of these approaches in this chapter.

10.4 The Linde-Buzo-Gray Algorithm

In Example 10.3.1 we saw that one way of exploiting the structure in the source output is to place the quantizer output points where the source output (blocked into vectors) are most likely to congregate. The set of quantizer output points is called the *codebook* of the quantizer, and the process of placing these output points is often referred to as *codebook design*. When we group the source output in two-dimensional vectors, as in the case of Example 10.3.1, we might be able to obtain a good codebook design by plotting a representative set of source output points and then visually locate where the quantizer output points should be. However, this approach to codebook design breaks down when we design higher-dimensional vector quantizers. Consider designing the codebook for a 16-dimensional quantizer. Obviously, a visual placement approach will not work in this case. We need an automatic procedure for locating where the source outputs are clustered.

This is a familiar problem in the field of pattern recognition. It is no surprise, therefore, that the most popular approach to designing vector quantizers is a clustering procedure known as the k-means algorithm, which was developed for pattern recognition applications.

The k-means algorithm functions as follows: Given a large set of output vectors from the source, known as the *training set*, and an initial set of k representative patterns, assign each element of the training set to the closest representative pattern. After an element is assigned, the representative pattern is updated by computing the centroid of the training set vectors assigned to it. When the assignment process is complete, we will have k groups of vectors clustered around each of the output points.

Stuart Lloyd [115] used this approach to generate the *pdf*-optimized scalar quantizer, except that instead of using a training set, he assumed that the distribution was known. The Lloyd algorithm functions as follows:

1. Start with an initial set of reconstruction values $\left\{y_i^{(0)}\right\}_{i=1}^{M}$. Set $k = 0$, $D^{(0)} = 0$. Select threshold ϵ.
2. Find decision boundaries
$$b_j^{(k)} = \frac{y_{j+1}^{(k)} + y_j^{(k)}}{2} \qquad j = 1, 2, \ldots, M-1.$$
3. Compute the distortion
$$D^{(k)} = \sum_{i=1}^{M} \int_{b_{i-1}^{(k)}}^{b_i^{(k)}} (x - y_i)^2 f_X(x)\,dx.$$
4. If $D^{(k)} - D^{(k-1)} < \epsilon$, stop; otherwise, continue.
5. $k = k + 1$. Compute new reconstruction values
$$y_j^{(k)} = \frac{\int_{b_{j-1}^{(k-1)}}^{b_j^{(k-1)}} x f_X(x)\,dx}{\int_{b_{j-1}^{(k-1)}}^{b_j^{(k-1)}} f_X(x)\,dx}.$$
Go to Step 2.

Linde, Buzo, and Gray generalized this algorithm to the case where the inputs are no longer scalars [125]. For the case where the distribution is known, the algorithm looks very much like the Lloyd algorithm described above.

1. Start with an initial set of reconstruction values $\left\{Y_i^{(0)}\right\}_{i=1}^{M}$. Set $k = 0$, $D^{(0)} = 0$. Select threshold ϵ.
2. Find quantization regions
$$V_i^{(k)} = \{X : d(X, Y_i) < d(X, Y_j)\ \forall j \neq i\} \qquad j = 1, 2, \ldots, M.$$
3. Compute the distortion
$$D^{(k)} = \sum_{i=1}^{M} \int_{V_i^{(k)}} \left\| X - Y_i^{(k)} \right\|^2 f_X(X)\,dX.$$

4. If $\frac{(D^{(k)}-D^{(k-1)})}{D^{(k)}} < \epsilon$, stop; otherwise, continue.

5. $k = k+1$. Find new reconstruction values $\left\{Y_i^{(k)}\right\}_{i=1}^{M}$ that are the centroids of $\left\{V_i^{(k-1)}\right\}$. Go to Step 2.

This algorithm is not very practical because the integrals required to compute the distortions and centroids are over odd-shaped regions in n dimensions, where n is the dimension of the input vectors. Generally, these integrals are extremely difficult to compute, making this particular algorithm more of an academic interest.

Of more practical interest is the algorithm for the case where we have a training set available. In this case, the algorithm looks very much like the k-means algorithm.

1. Start with an initial set of reconstruction values $\left\{Y_i^{(0)}\right\}_{i=1}^{M}$ and a set of training vectors $\{X_n\}_{n=1}^{N}$. Set $k = 0$, $D^{(0)} = 0$. Select threshold ϵ.

2. The quantization regions $\left\{V_i^{(k)}\right\}_{i=1}^{M}$ are given by

$$V_i^{(k)} = \{X_n : d(X_n, Y_i) < d(X_n, Y_j)\ \forall j \neq i\} \qquad i = 1, 2, \ldots, M.$$

   We assume that none of the quantization regions are empty. (Later we will deal with the case where $V_i^{(k)}$ is empty for some i and k.)

3. Compute the average distortion $D^{(k)}$ between the training vectors and the representative reconstruction value.

4. If $\frac{(D^{(k)}-D^{(k-1)})}{D^{(k)}} < \epsilon$, stop; otherwise, continue.

5. $k = k+1$. Find new reconstruction values $\left\{Y_i^{(k)}\right\}_{i=1}^{M}$ that are the average value of the elements of each of the quantization regions $V_i^{(k-1)}$. Go to Step 2.

This algorithm forms the basis of most vector quantizer designs. It is popularly known as the Linde-Buzo-Gray or LBG algorithm, or the generalized Lloyd algorithm (GLA) [125]. Although the paper of Linde, Buzo, and Gray [125] is a starting point for most of the work on vector quantization, the latter algorithm had been used several years prior by Edward E. Hilbert at the NASA Jet Propulsion Laboratories in Pasadena, California. Hilbert's starting point was the idea of clustering, and although he arrived at the same algorithm as described above, he called it the *cluster compression algorithm* [126].

In order to see how this algorithm functions, consider the following example of a two-dimensional vector quantizer codebook design.

Example 10.4.1:

Suppose our training set consists of the height and weight values shown in Table 10.1. The initial set of output points is shown in Table 10.2. (For ease of presentation, we will always round the coordinates of the output points to the nearest integer.) The inputs, outputs, and quantization regions are shown in Figure 10.7.

TABLE 10.1 Training set for designing vector quantizer codebook.

Height	Weight
72	180
65	120
59	119
64	150
65	162
57	88
72	175
44	41
62	114
60	110
56	91
70	172

TABLE 10.2 Initial set of output points for codebook design.

Height	Weight
45	50
75	117
45	117
80	180

The input (44, 41) has been assigned to the first output point; the inputs (56, 91), (57, 88), (59, 119), and (60, 110) have been assigned to the second output point; the inputs (62, 114), and (65, 120) have been assigned to the third output; and the five remaining vectors from the training set have been assigned to the fourth output. The distortion for this assignment is 387.25. We now find the new output points. There is only one vector in the first quantization region, so the first output point is (44, 41). The average of the four vectors in the second quantization region (rounded up) is the vector (58, 102), which is the new second output point. In a similar manner, we can compute the third and fourth output points as (64, 117) and (69, 168). The new output points and the corresponding quantization regions are shown in Figure 10.8. From Figure 10.8, we can see that, while the training vectors that were initially part of the first and fourth quantization regions are still in the same quantization regions, the training vectors (59,115) and (60,120), which were in quantization region 2, are now in quantization region 3. The distortion corresponding to this assignment of training vectors to quantization regions is 89, considerably less than the original 387.25. Given the new assignments, we can obtain a new set of output points. The first and fourth output points do not change because the training vectors in the corresponding regions have not changed. However, the training vectors in regions 2 and 3 have changed. Recomputing the output points for these regions, we get (57, 90) and (62, 116). The final form of the

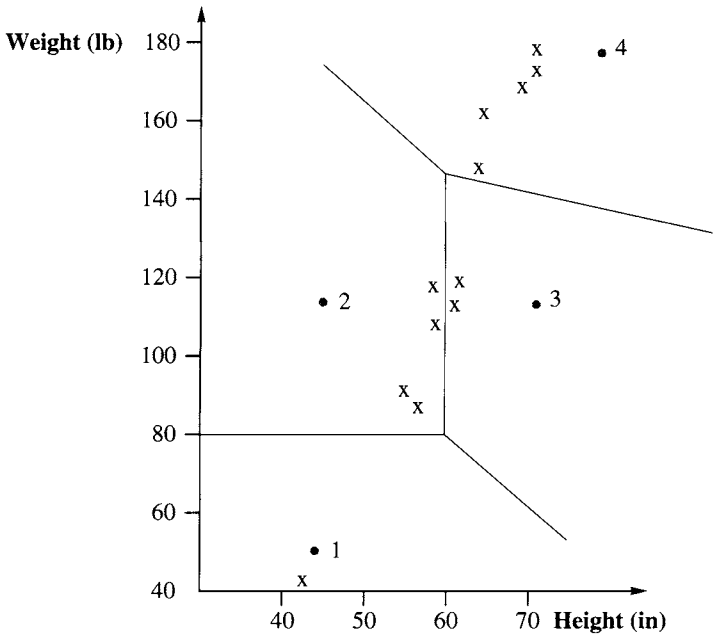

FIGURE 10. 7 **Initial state of the vector quantizer.**

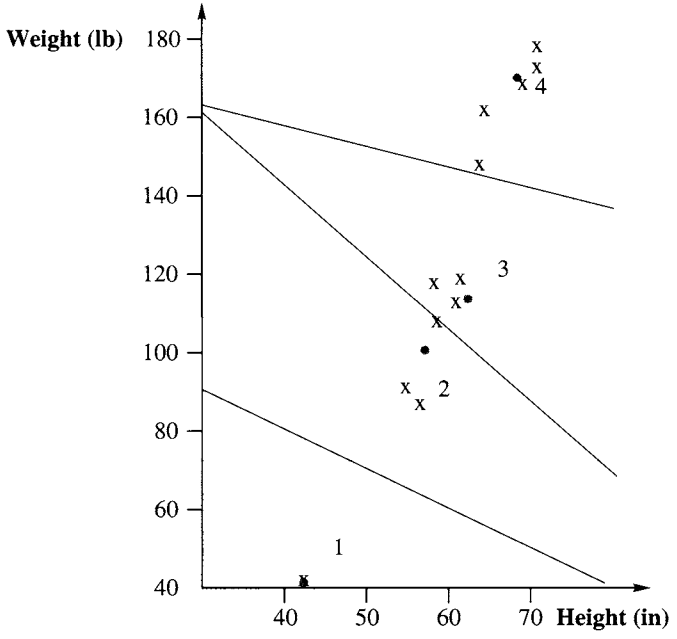

FIGURE 10. 8 **The vector quantizer after one iteration.**

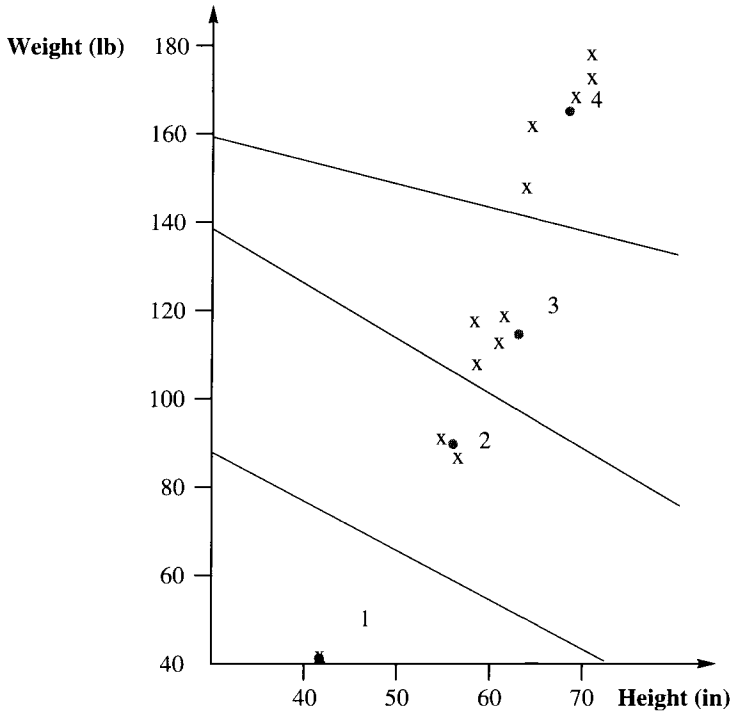

FIGURE 10.9 Final state of the vector quantizer.

quantizer is shown in Figure 10.9. The distortion corresponding to the final assignments is 60.17. ♦

The LBG algorithm is conceptually simple, and as we shall see later, the resulting vector quantizer is remarkably effective in the compression of a wide variety of inputs, both by itself and in conjunction with other schemes. In the next two sections we will look at some of the details of the codebook design process. While these details are important to consider when designing codebooks, they are not necessary for the understanding of the quantization process. If you are not currently interested in these details, you may wish to proceed directly to Section 10.4.3.

10.4.1 Initializing the LBG Algorithm

The LBG algorithm guarantees that the distortion from one iteration to the next will not increase. However, there is no guarantee that the procedure will converge to the optimal solution. The solution to which the algorithm converges is heavily dependent on the initial conditions. For example, if our initial set of output points in Example 10.4 had been those

TABLE 10.3 An alternate initial set of output points.

Height	Weight
75	50
75	117
75	127
80	180

TABLE 10.4 Final codebook obtained using the alternative initial codebook.

Height	Weight
44	41
60	107
64	150
70	172

shown in Table 10.3 instead of the set in Table 10.2, by using the LBG algorithm we would get the final codebook shown in Table 10.4.

The resulting quantization regions and their membership are shown in Figure 10.10. This is a very different quantizer than the one we had previously obtained. Given this heavy dependence on initial conditions, the selection of the initial codebook is a matter of some importance. We will look at some of the better-known methods of initialization in the following section.

Linde, Buzo, and Gray described a technique in their original paper [125] called the *splitting technique* for initializing the design algorithm. In this technique, we begin by designing a vector quantizer with a single output point; in other words, a codebook of size one, or a one-level vector quantizer. With a one-element codebook, the quantization region is the entire input space, and the output point is the average value of the entire training set. From this output point, the initial codebook for a two-level vector quantizer can be obtained by including the output point for the one-level quantizer and a second output point obtained by adding a fixed perturbation vector ϵ. We then use the LBG algorithm to obtain the two-level vector quantizer. Once the algorithm has converged, the two codebook vectors are used to obtain the initial codebook of a four-level vector quantizer. This initial four-level codebook consists of the two codebook vectors from the final codebook of the two-level vector quantizer and another two vectors obtained by adding ϵ to the two codebook vectors. The LBG algorithm can then be used until this four-level quantizer converges. In this manner we keep doubling the number of levels until we reach the desired number of levels. By including the final codebook of the previous stage at each "splitting," we guarantee that the codebook after splitting will be at least as good as the codebook prior to splitting.

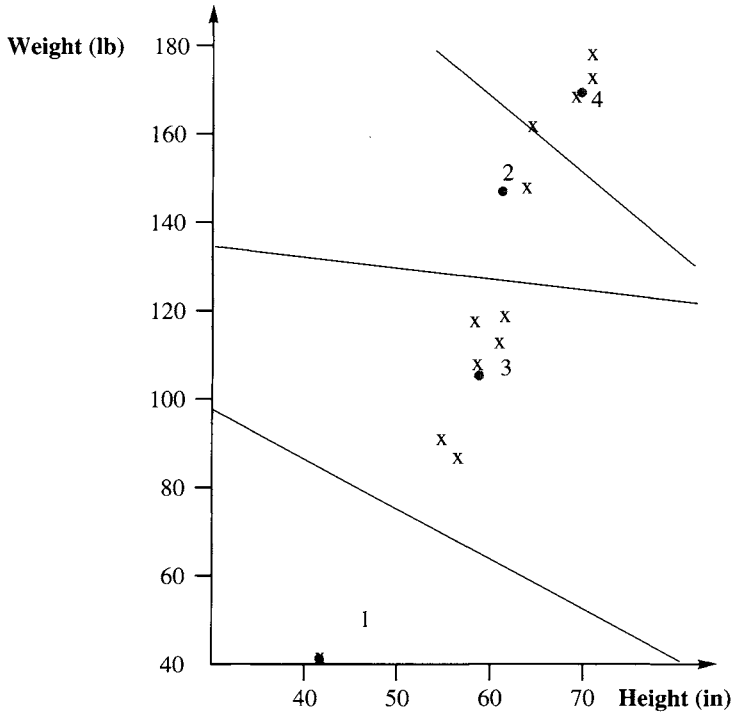

FIGURE 10. 10 Final state of the vector quantizer.

Example 10.4.2:

Let's revisit Example 10.4.1. This time, instead of using the initial codewords used in Example 10.4.1, we will use the splitting technique. For the perturbations, we will use a fixed vector $\epsilon = (10, 10)$. The perturbation vector is usually selected randomly; however, for purposes of explanation it is more useful to use a fixed perturbation vector.

We begin with a single-level codebook. The codeword is simply the average value of the training set. The progression of codebooks is shown in Table 10.5.

The perturbed vectors are used to initialize the LBG design of a two-level vector quantizer. The resulting two-level vector quantizer is shown in Figure 10.11. The resulting distortion is 468.58. These two vectors are perturbed to get the initial output points for the four-level design. Using the LBG algorithm, the final quantizer obtained is shown in Figure 10.12. The distortion is 156.17. The average distortion for the training set for this quantizer using the splitting algorithm is higher than the average distortion obtained previously. However, because the sample size used in this example is rather small, this is no indication of relative merit. ♦

TABLE 10.5 Progression of codebooks using splitting.

Codebook	Height	Weight
One-level	62	127
Initial two-level	62	127
	72	137
Final two-level	58	98
	69	168
Initial four-level	58	98
	68	108
	69	168
	79	178
Final four-level	52	73
	62	116
	65	156
	71	176

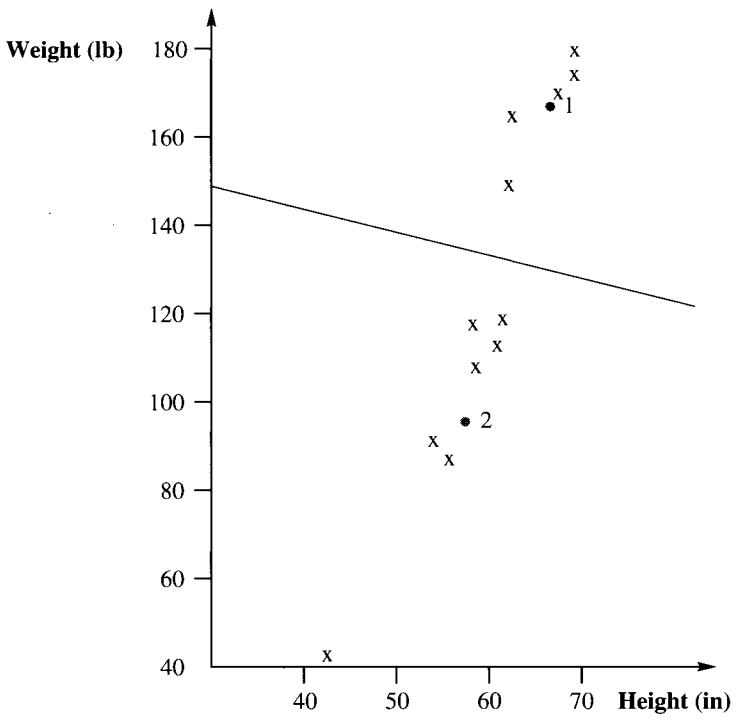

FIGURE 10.11 Two-level vector quantizer using splitting approach.

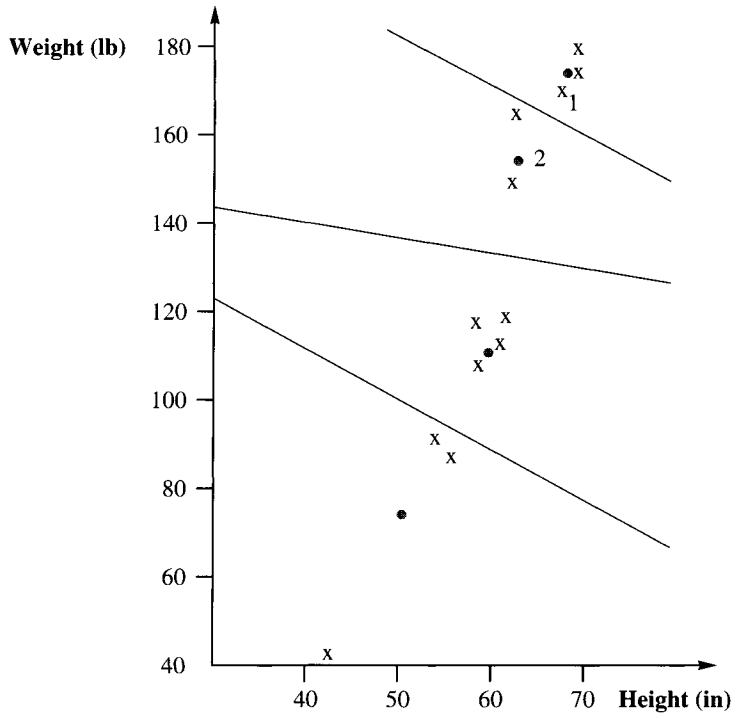

FIGURE 10.12 Final design using the splitting approach.

If the desired number of levels is not a power of two, then in the last step, instead of generating two initial points from each of the output points of the vector quantizer designed previously, we can perturb as many vectors as necessary to obtain the desired number of vectors. For example, if we needed an eleven-level vector quantizer, we would generate a one-level vector quantizer first, then a two-level, then a four-level, and then an eight-level vector quantizer. At this stage, we would perturb only three of the eight vectors to get the eleven initial output points of the eleven-level vector quantizer. The three points should be those with the largest number of training set vectors, or the largest distortion.

The approach used by Hilbert [126] to obtain the initial output points of the vector quantizer was to pick the output points randomly from the training set. This approach guarantees that, in the initial stages, there will always be at least one vector from the training set in each quantization region. However, we can still get different codebooks if we use different subsets of the training set as our initial codebook.

Example 10.4.3:

Using the training set of Example 10.4.1, we selected different vectors of the training set as the initial codebook. The results are summarized in Table 10.6. If we pick the codebook labeled "Initial Codebook 1," we obtain the codebook labeled "Final Codebook 1." This

TABLE 10.6 Effect of using different subsets of the training sequence as the initial codebook.

Codebook	Height	Weight
Initial Codebook 1	72	180
	72	175
	65	120
	59	119
Final Codebook 1	71	176
	65	156
	62	116
	52	73
Initial Codebook 2	65	120
	44	41
	59	119
	57	88
Final Codebook 2	69	168
	44	41
	62	116
	57	90

codebook is identical to the one obtained using the split algorithm. The set labeled "Initial Codebook 2" results in the codebook labeled "Final Codebook 2." This codebook is identical to the quantizer we obtained in Example 10.4.1. In fact, most of the other selections result in one of these two quantizers. ♦

Notice that by picking different subsets of the input as our initial codebook, we can generate different vector quantizers. A good approach to codebook design is to initialize the codebook randomly several times, and pick the one that generates the least distortion in the training set from the resulting quantizers.

In 1989, Equitz [127] introduced a method for generating the initial codebook called the *pairwise nearest neighbor* (PNN) algorithm. In the PNN algorithm, we start with as many clusters as there are training vectors and end with the initial codebook. At each stage, we combine the two closest vectors into a single cluster and replace the two vectors by their mean. The idea is to merge those clusters that would result in the smallest increase in distortion. Equitz showed that when we combine two clusters C_i and C_j, the increase in distortion is

$$\frac{n_i n_j}{n_i + n_j} \| Y_i - Y_j \|^2, \tag{10.5}$$

where n_i is the number of elements in the cluster C_i, and Y_i is the corresponding output point. In the PNN algorithm, we combine clusters that cause the smallest increase in the distortion.

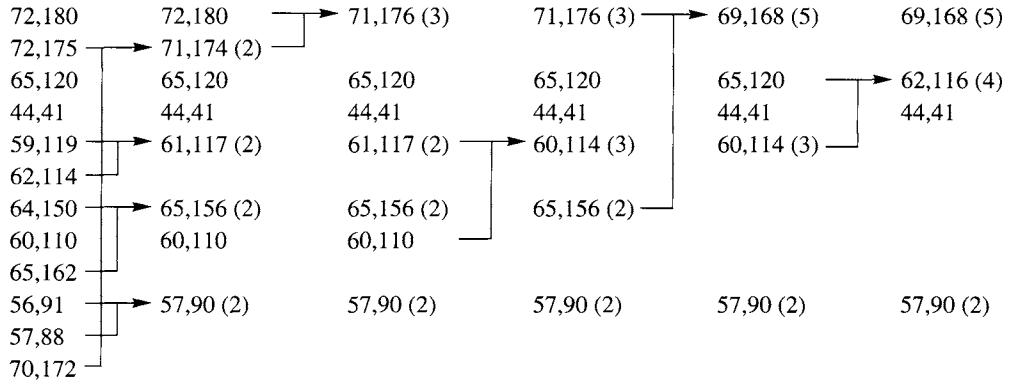

FIGURE 10. 13 Obtaining initial output points using the PNN approach.

Example 10.4.4:

Using the PNN algorithm, we combine the elements in the training set as shown in Figure 10.13. At each step we combine the two clusters that are closest in the sense of Equation (10.5). If we use these values to initialize the LBG algorithm, we get a vector quantizer shown with output points (70, 172), (60, 107), (44, 41), (64, 150), and a distortion of 104.08. ♦

Although it was a relatively easy task to generate the initial codebook using the PNN algorithm in Example 10.4.4, we can see that, as the size of the training set increases, this procedure becomes progressively more time-consuming. In order to avoid this cost, we can use a fast PNN algorithm that does not attempt to find the absolute smallest cost at each step (see [127] for details).

Finally, a simple initial codebook is the set of output points from the corresponding scalar quantizers. In the beginning of this chapter we saw how scalar quantization of a sequence of inputs can be viewed as vector quantization using a rectangular vector quantizer. We can use this rectangular vector quantizer as the initial set of outputs.

Example 10.4.5:

Return once again to the quantization of the height-weight data set. If we assume that the heights are uniformly distributed between 40 and 180, then a two-level scalar quantizer would have reconstruction values 75 and 145. Similarly, if we assume that the weights are uniformly distributed between 40 and 80, the reconstruction values would be 50 and 70. The initial reconstruction values for the vector quantizer are (50, 75), (50, 145), (70, 75), and (70, 145). The final design for this initial set is the same as the one obtained in Example 10.4.1 with a distortion of 60.17. ♦

We have looked at four different ways of initializing the LBG algorithm. Each has its own advantages and drawbacks. The PNN initialization has been shown to result in better designs, producing a lower distortion for a given rate than the splitting approach [127]. However, the procedure for obtaining the initial codebook is much more involved and complex. We cannot make any general claims regarding the superiority of any one of these initialization techniques. Even the PNN approach cannot be proven to be optimal. In practice, if we are dealing with a wide variety of inputs, the effect of using different initialization techniques appears to be insignificant.

10.4.2 The Empty Cell Problem

Let's take a closer look at the progression of the design in Example 10.4.5. When we assign the inputs to the initial output points, no input point gets assigned to the output point at (70, 75). This is a problem because in order to update an output point, we need to take the average value of the input vectors. Obviously, some strategy is needed. The strategy that we actually used in Example 10.4.5 was not to update the output point if there were no inputs in the quantization region associated with it. This strategy seems to have worked in this particular example; however, there is a danger that we will end up with an output point that is never used. A common approach to avoid this is to remove an output point that has no inputs associated with it, and replace it with a point from the quantization region with the most output points. This can be done by selecting a point at random from the region with the highest population of training vectors, or the highest associated distortion. A more systematic approach is to design a two-level quantizer for the training vectors in the most heavily populated quantization region. This approach is computationally expensive and provides no significant improvement over the simpler approach. In the program accompanying this book, we have used the first approach. (To compare the two approaches, see Problem 3.)

10.4.3 Use of LBG for Image Compression

One application for which the vector quantizer described in this section has been extremely popular is image compression. For image compression, the vector is formed by taking blocks of pixels of size $N \times M$ and treating them as an $L = NM$ dimensional vector. Generally, we take $N = M$. Instead of forming vectors in this manner, we could form the vector by taking L pixels in a row of the image. However, this does not allow us to take advantage of the two-dimensional correlations in the image. Recall that correlation between the samples provides the clustering of the input, and the LBG algorithm takes advantage of this clustering.

Example 10.4.6:

Let us quantize the Sinan image shown in Figure 10.14 using a 16-dimensional quantizer. The input vectors are constructed using 4×4 blocks of pixels. The codebook was trained on the Sinan image.

The results of the quantization using codebooks of size 16, 64, 256, and 1024 are shown in Figure 10.15. The rates and compression ratios are summarized in Table 10.7. To see how these quantities were calculated, recall that if we have K vectors in a codebook, we need

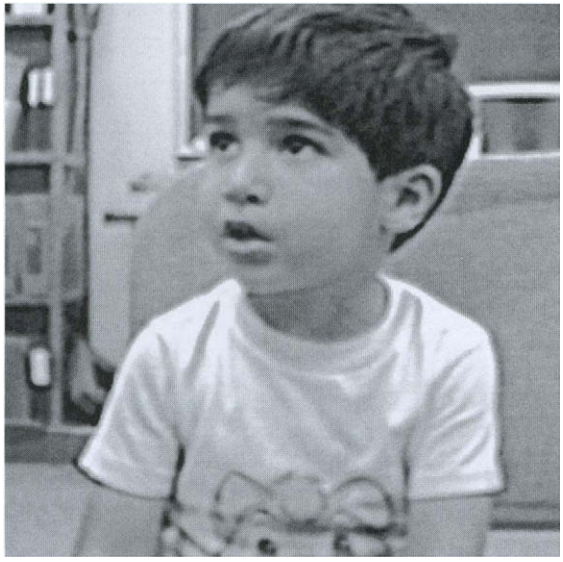

FIGURE 10. 14 Original Sinan image.

$\lceil \log_2 K \rceil$ bits to inform the receiver which of the K vectors is the quantizer output. This quantity is listed in the second column of Table 10.7 for the different values of K. If the vectors are of dimension L, this means that we have used $\lceil \log_2 K \rceil$ bits to send the quantized value of L pixels. Therefore, the rate in bits per pixel is $\frac{\lceil \log_2 K \rceil}{L}$. (We have assumed that the codebook is available to both transmitter and receiver, and therefore we do not have to use any bits to transmit the codebook from the transmitter to the receiver.) This quantity is listed in the third column of Table 10.7. Finally, the compression ratio, given in the last column of Table 10.7, is the ratio of the number of bits per pixel in the original image to the number of bits per pixel in the compressed image. The Sinan image was digitized using 8 bits per pixel. Using this information and the rate after compression, we can obtain the compression ratios.

Looking at the images, we see that reconstruction using a codebook of size 1024 is very close to the original. At the other end, the image obtained using a codebook with 16 reconstruction vectors contains a lot of visible artifacts. The utility of each reconstruction depends on the demands of the particular application. ♦

In this example, we used codebooks trained on the image itself. Generally, this is not the preferred approach because the receiver has to have the same codebook in order to reconstruct the image. Either the codebook must be transmitted along with the image, or the receiver has the same training image so that it can generate an identical codebook. This is impractical because, if the receiver already has the image in question, much better compression can be obtained by simply sending the name of the image to the receiver. Sending the codebook with the image is not unreasonable. However, the transmission of

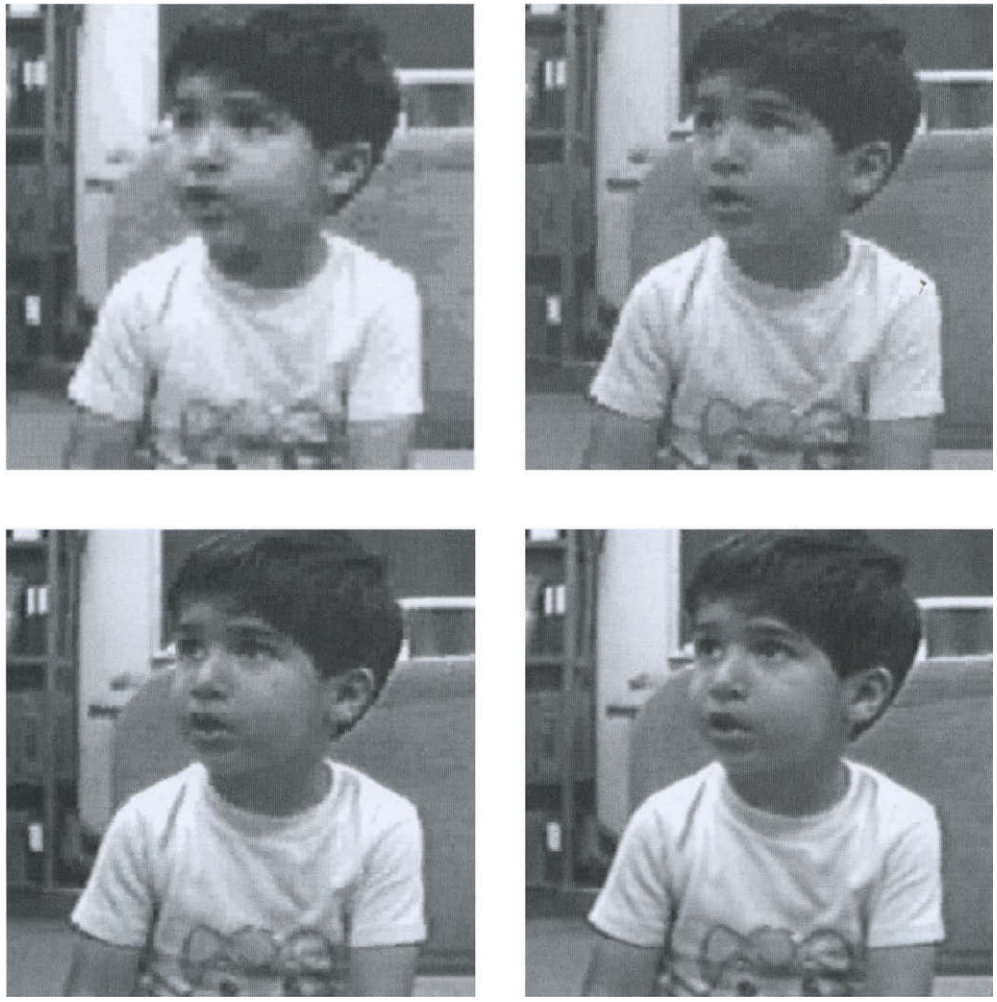

FIGURE 10.15 **Top left: codebook size 16; top right: codebook size 64; bottom left: codebook size 256; bottom right: codebook size 1024.**

TABLE 10.7 **Summary of compression measures for image compression example.**

Codebook Size (# of codewords)	Bits Needed to Select a Codeword	Bits per Pixel	Compression Ratio
16	4	0.25	32:1
64	6	0.375	21.33:1
256	8	0.50	16:1
1024	10	0.625	12.8:1

TABLE 10.8 Overhead in bits per pixel for codebooks of different sizes.

Codebook Size K	Overhead in Bits per Pixel
16	0.03125
64	0.125
256	0.50
1024	2.0

the codebook is overhead that could be avoided if a more generic codebook, one that is available to both transmitter and receiver, were to be used.

In order to compute the overhead, we need to calculate the number of bits required to transmit the codebook to the receiver. If each codeword in the codebook is a vector with L elements and if we use B bits to represent each element, then in order to transmit the codebook of a K-level quantizer we need $B \times L \times K$ bits. In our example, $B = 8$ and $L = 16$. Therefore, we need $K \times 128$ bits to transmit the codebook. As our image consists of 256×256 pixels, the overhead in bits per pixel is $128K/65,536$. The overhead for different values of K is summarized in Table 10.8. We can see that while the overhead for a codebook of size 16 seems reasonable, the overhead for a codebook of size 1024 is over three times the rate required for quantization.

Given the excessive amount of overhead required for sending the codebook along with the vector quantized image, there has been substantial interest in the design of codebooks that are more generic in nature and, therefore, can be used to quantize a number of images. To investigate the issues that might arise, we quantized the Sinan image using four different codebooks generated by the Sena, Sensin, Earth, and Omaha images. The results are shown in Figure 10.16.

As expected, the reconstructed images from this approach are not of the same quality as when the codebook is generated from the image to be quantized. However, this is only true as long as the overhead required for storage or transmission of the codebook is ignored. If we include the extra rate required to encode and transmit the codebook of output points, using the codebook generated by the image to be quantized seems unrealistic. Although using the codebook generated by another image to perform the quantization may be realistic, the quality of the reconstructions is quite poor. Later in this chapter we will take a closer look at the subject of vector quantization of images and consider a variety of ways to improve this performance.

You may have noticed that the bit rates for the vector quantizers used in the examples are quite low. The reason is that the size of the codebook increases exponentially with the rate. Suppose we want to encode a source using R bits per sample; that is, the average number of bits per sample in the compressed source output is R. By "sample" we mean a scalar element of the source output sequence. If we wanted to use an L-dimensional quantizer, we would group L samples together into vectors. This means that we would have RL bits available to represent each vector. With RL bits, we can represent 2^{RL} different output vectors. In other words, the size of the codebook for an L-dimensional R-bits-per-sample quantizer is 2^{RL}. From Table 10.7, we can see that when we quantize an image using 0.25 bits per pixel and 16-dimensional quantizers, we have $16 \times 0.25 = 4$ bits available to represent each

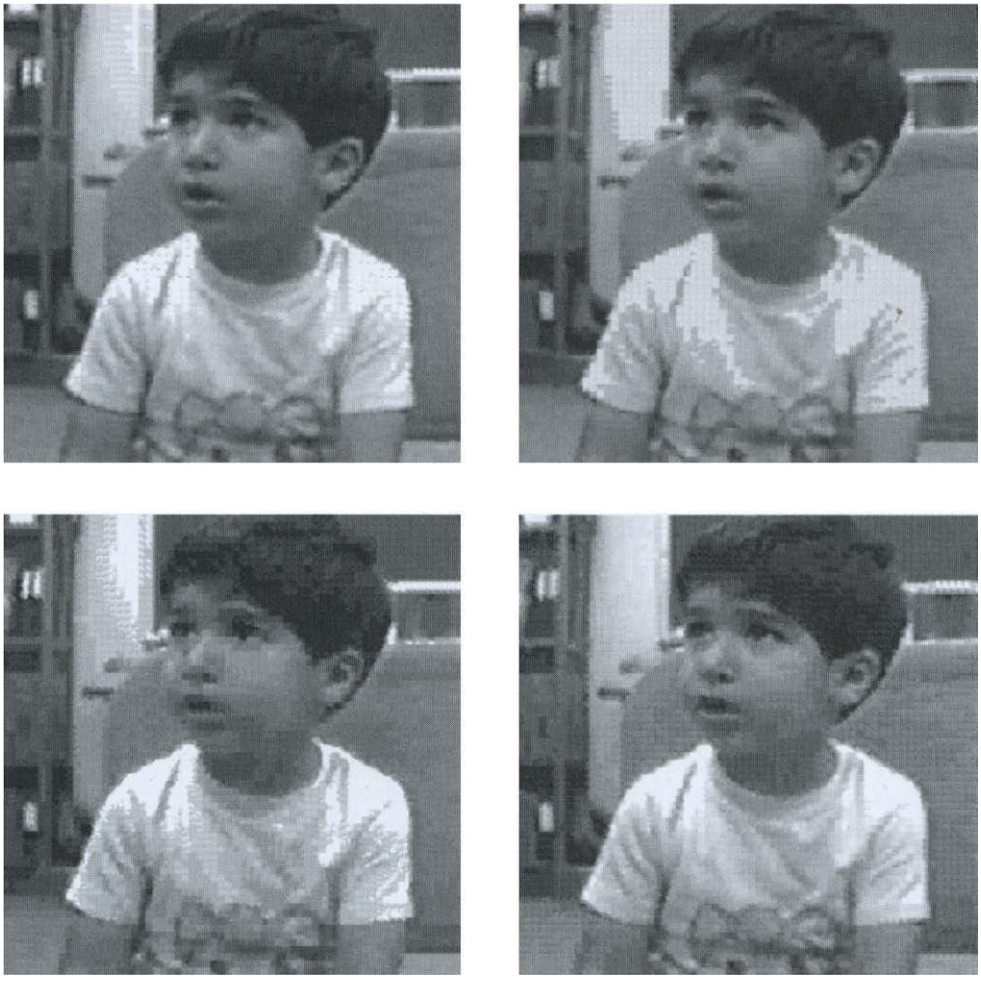

FIGURE 10.16 **Sinan image quantized at the rate of 0.5 bits per pixel. The images used to obtain the codebook were (clockwise from top left) Sensin, Sena, Earth, Omaha.**

vector. Hence, the size of the codebook is $2^4 = 16$. The quantity RL is often called the *rate dimension product*. Note that the size of the codebook grows exponentially with this product.

Consider the problems. The codebook size for a 16-dimensional, 2-bits-per-sample vector quantizer would be $2^{16\times 2}$! (If the source output was originally represented using 8 bits per sample, a rate of 2 bits per sample for the compressed source corresponds to a compression ratio of 4:1.) This large size causes problems both with storage and with the quantization process. To store 2^{32} sixteen-dimensional vectors, assuming that we can store each component of the vector in a single byte, requires $2^{32} \times 16$ bytes—approximately 64 gigabytes of storage. Furthermore, to quantize a single input vector would require over four billion vector

comparisons to find the closest output point. Obviously, neither the storage requirements nor the computational requirements are realistic. Because of this problem, most vector quantization applications operate at low bit rates. In many applications, such as low-rate speech coding, we want to operate at very low rates; therefore, this is not a drawback. However, for applications such as high-quality video coding, which requires higher rates, this is definitely a problem.

There are several approaches to solving these problems. Each entails the introduction of some structure in the codebook and/or the quantization process. While the introduction of structure mitigates some of the storage and computational problems, there is generally a trade-off in terms of the distortion performance. We will look at some of these approaches in the following sections.

10.5 Tree-Structured Vector Quantizers

One way we can introduce structure is to organize our codebook in such a way that it is easy to pick which part contains the desired output vector. Consider the two-dimensional vector quantizer shown in Figure 10.17. Note that the output points in each quadrant are the mirror image of the output points in neighboring quadrants. Given an input to this vector quantizer, we can reduce the number of comparisons necessary for finding the closest output point by using the sign on the components of the input. The sign on the components of the input vector will tell us in which quadrant the input lies. Because all the quadrants are mirror images of the neighboring quadrants, the closest output point to a given input will lie in the same quadrant as the input itself. Therefore, we only need to compare the input to the output points that lie in the same quadrant, thus reducing the number of required comparisons by a factor of four. This approach can be extended to L dimensions, where the signs on the L components of the input vector can tell us in which of the 2^L hyperquadrants the input lies, which in turn would reduce the number of comparisons by 2^L.

This approach works well when the output points are distributed in a symmetrical manner. However, it breaks down as the distribution of the output points becomes less symmetrical.

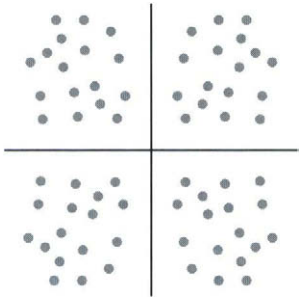

FIGURE 10.17 A symmetrical vector quantizer in two dimensions.

Example 10.5.1:

Consider the vector quantizer shown in Figure 10.18. This is different from the output points in Figure 10.17; we have dropped the mirror image requirement of the previous example. The output points are shown as filled circles, and the input point is the X. It is obvious from the figure that while the input is in the first quadrant, the closest output point is in the fourth quadrant. However, the quantization approach described above will force the input to be represented by an output in the first quadrant.

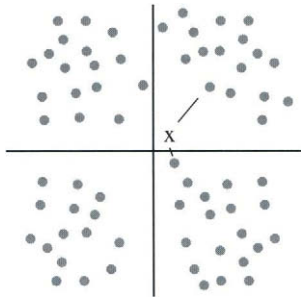

FIGURE 10.18 Breakdown of the method using the quadrant approach.

The situation gets worse as we lose more and more of the symmetry. Consider the situation in Figure 10.19. In this quantizer, not only will we get an incorrect output point when the input is close to the boundaries of the first quadrant, but also there is no significant reduction in the amount of computation required.

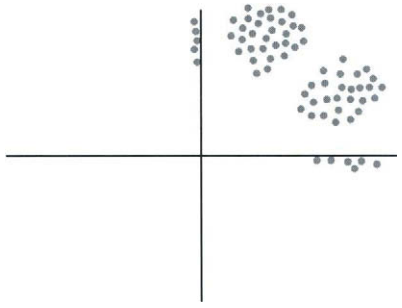

FIGURE 10.19 Breakdown of the method using the quadrant approach.

Most of the output points are in the first quadrant. Therefore, whenever the input falls in the first quadrant, which it will do quite often if the quantizer design is reflective of the distribution of the input, knowing that it is in the first quadrant does not lead to a great reduction in the number of comparisons. ♦

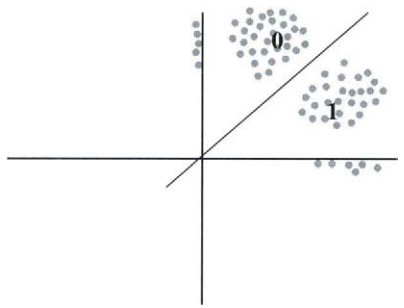

FIGURE 10. 20 Division of output points into two groups.

The idea of using the L-dimensional equivalents of quadrants to partition the output points in order to reduce the computational load can be extended to nonsymmetrical situations, like those shown in Figure 10.19, in the following manner. Divide the set of output points into two groups, *group0* and *group1*, and assign to each group a test vector such that output points in each group are closer to the test vector assigned to that group than to the test vector assigned to the other group (Figure 10.20). Label the two test vectors 0 and 1. When we get an input vector, we compare it against the test vectors. Depending on the outcome, the input is compared to the output points associated with the test vector closest to the input. After these two comparisons, we can discard half of the output points. Comparison with the test vectors takes the place of looking at the signs of the components to decide which set of output points to discard from contention. If the total number of output points is K, with this approach we have to make $\frac{K}{2}+2$ comparisons instead of K comparisons.

This process can be continued by splitting the output points in each group into two groups and assigning a test vector to the subgroups. So *group0* would be split into *group00* and *group01*, with associated test vectors labeled 00 and 01, and *group1* would be split into *group10* and *group11*, with associated test vectors labeled 10 and 11. Suppose the result of the first set of comparisons was that the output point would be searched for in *group1*. The input would be compared to the test vectors 10 and 11. If the input was closer to the test vector 10, then the output points in *group11* would be discarded, and the input would be compared to the output points in *group10*. We can continue the procedure by successively dividing each group of output points into two, until finally, if the number of output points is a power of two, the last set of groups would consist of single points. The number of comparisons required to obtain the final output point would be $2\log K$ instead of K. Thus, for a codebook of size 4096 we would need 24 vector comparisons instead of 4096 vector comparisons.

This is a remarkable decrease in computational complexity. However, we pay for this decrease in two ways. The first penalty is a possible increase in distortion. It is possible at some stage that the input is closer to one test vector while at the same time being closest to an output belonging to the rejected group. This is similar to the situation shown in Figure 10.18. The other penalty is an increase in storage requirements. Now we not only have to store the output points from the vector quantizer codebook, we also must store the test vectors. This means almost a doubling of the storage requirement.

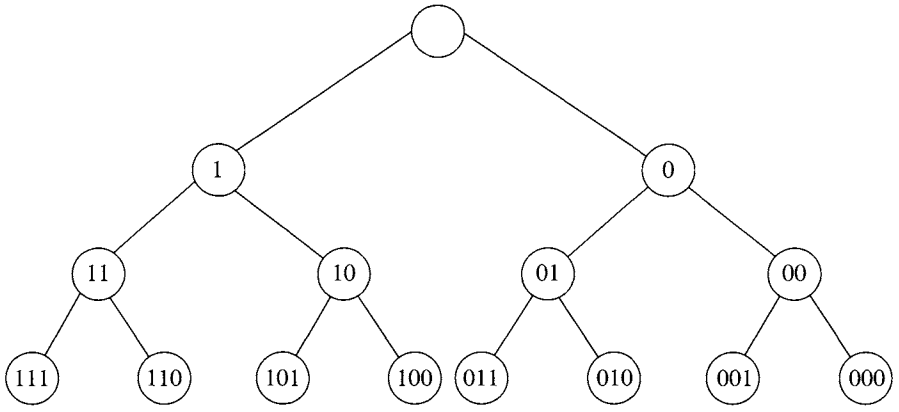

FIGURE 10.21 Decision tree for quantization.

The comparisons that must be made at each step are shown in Figure 10.21. The label inside each node is the label of the test vector that we compare the input against. This tree of decisions is what gives tree-structured vector quantizers (TSVQ) their name. Notice also that, as we are progressing down a tree, we are also building a binary string. As the leaves of the tree are the output points, by the time we reach a particular leaf or, in other words, select a particular output point, we have obtained the binary codeword corresponding to that output point.

This process of building the binary codeword as we progress through the series of decisions required to find the final output can result in some other interesting properties of tree-structured vector quantizers. For instance, even if a partial codeword is transmitted, we can still get an approximation of the input vector. In Figure 10.21, if the quantized value was the codebook vector 5, the binary codeword would be 011. However, if only the first two bits 01 were received by the decoder, the input can be approximated by the test vector labeled 01.

10.5.1 Design of Tree-Structured Vector Quantizers

In the last section we saw how we could reduce the computational complexity of the design process by imposing a tree structure on the vector quantizer. Rather than imposing this structure after the vector quantizer has been designed, it makes sense to design the vector quantizer within the framework of the tree structure. We can do this by a slight modification of the splitting design approach proposed by Linde et al. [125].

We start the design process in a manner identical to the splitting technique. First, obtain the average of all the training vectors, perturb it to obtain a second vector, and use these vectors to form a two-level vector quantizer. Let us label these two vectors 0 and 1, and the groups of training set vectors that would be quantized to each of these two vectors *group0* and *group1*. We will later use these vectors as test vectors. We perturb these output points to get the initial vectors for a four-level vector quantizer. At this point, the design procedure

for the tree-structured vector quantizer deviates from the splitting technique. Instead of using the entire training set to design a four-level vector quantizer, we use the training set vectors in *group0* to design a two-level vector quantizer with output points labeled 00 and 01. We use the training set vectors in *group1* to design a two-level vector quantizer with output points labeled 10 and 11. We also split the training set vectors in *group0* and *group1* into two groups each. The vectors in *group0* are split, based on their proximity to the vectors labeled 00 and 01, into *group00* and *group01*, and the vectors in *group1* are divided in a like manner into the groups *group10* and *group11*. The vectors labeled 00, 01, 10, and 11 will act as test vectors at this level. To get an eight-level quantizer, we use the training set vectors in each of the four groups to obtain four two-level vector quantizers. We continue in this manner until we have the required number of output points. Notice that in the process of obtaining the output points, we have also obtained the test vectors required for the quantization process.

10.5.2 Pruned Tree-Structured Vector Quantizers

Once we have built a tree-structured codebook, we can sometimes improve its rate distortion performance by removing carefully selected subgroups. Removal of a subgroup, referred to as *pruning*, will reduce the size of the codebook and hence the rate. It may also result in an increase in distortion. Therefore, the objective of the pruning is to remove those subgroups that will result in the best trade-off of rate and distortion. Chou, Lookabaugh, and Gray [128] have developed an optimal pruning algorithm called the *generalized BFOS algorithm*. The name of the algorithm derives from the fact that it is an extension of an algorithm originally developed by Brieman, Freidman, Olshen, and Stone [129] for classification applications. (See [128] and [5] for description and discussion of the algorithm.)

Pruning output points from the codebook has the unfortunate effect of removing the structure that was previously used to generate the binary codeword corresponding to the output points. If we used the structure to generate the binary codewords, the pruning would cause the codewords to be of variable length. As the variable-length codes would correspond to the leaves of a binary tree, this code would be a prefix code and, therefore, certainly usable. However, it would not require a large increase in complexity to assign fixed-length codewords to the output points using another method. This increase in complexity is generally offset by the improvement in performance that results from the pruning [130].

10.6 Structured Vector Quantizers

The tree-structured vector quantizer solves the complexity problem, but acerbates the storage problem. We now take an entirely different tack and develop vector quantizers that do not have these storage problems; however, we pay for this relief in other ways.

Example 10.3.1 was our motivation for the quantizer obtained by the LBG algorithm. This example showed that the correlation between samples of the output of a source leads to clustering. This clustering is exploited by the LBG algorithm by placing output points at the location of these clusters. However, in Example 10.3.2, we saw that even when there

is no correlation between samples, there is a kind of probabilistic structure that becomes more evident as we group the random inputs of a source into larger and larger blocks or vectors.

In Example 10.3.2, we changed the position of the output point in the top-right corner. All four corner points have the same probability, so we could have chosen any of these points. In the case of the two-dimensional Laplacian distribution in Example 10.3.2, all points that lie on the contour described by $|x| + |y| = constant$ have equal probability. These are called *contours of constant probability*. For spherically symmetrical distributions like the Gaussian distribution, the contours of constant probability are circles in two dimensions, spheres in three dimensions, and hyperspheres in higher dimensions.

We mentioned in Example 10.3.2 that the points away from the origin have very little probability mass associated with them. Based on what we have said about the contours of constant probability, we can be a little more specific and say that the points on constant probability contours farther away from the origin have very little probability mass associated with them. Therefore, we can get rid of all of the points outside some contour of constant probability without incurring much of a distortion penalty. In addition as the number of reconstruction points is reduced, there is a decrease in rate, thus improving the rate distortion performance.

Example 10.6.1:

Let us design a two-dimensional uniform quantizer by keeping only the output points in the quantizer of Example 10.3.2 that lie on or within the contour of constant probability given by $|x_1| + |x_2| = 5\Delta$. If we count all the points that are retained, we get 60 points. This is close enough to 64 that we can compare it with the eight-level uniform scalar quantizer. If we simulate this quantization scheme with a Laplacian input, and the same step size as the scalar quantizer, that is, $\Delta = 0.7309$, we get an SNR of 12.22 dB. Comparing this to the 11.44 dB obtained with the scalar quantizer, we see that there is a definite improvement. We can get slightly more improvement in performance if we modify the step size. ♦

Notice that the improvement in the previous example is obtained only by restricting the outer boundary of the quantizer. Unlike Example 10.3.2, we did not change the shape of any of the inner quantization regions. This gain is referred to in the quantization literature as *boundary gain*. In terms of the description of quantization noise in Chapter 8, we reduced the overload error by reducing the overload probability, without a commensurate increase in the granular noise. In Figure 10.22, we have marked the 12 output points that belonged to the original 64-level quantizer, but do not belong to the 60-level quantizer, by drawing circles around them. Removal of these points results in an increase in overload probability. We also marked the eight output points that belong to the 60-level quantizer, but were not part of the original 64-level quantizer, by drawing squares around them. Adding these points results in a decrease in the overload probability. If we calculate the increases and decreases (Problem 5), we find that the net result is a decrease in overload probability. This overload probability is further reduced as the dimension of the vector is increased.

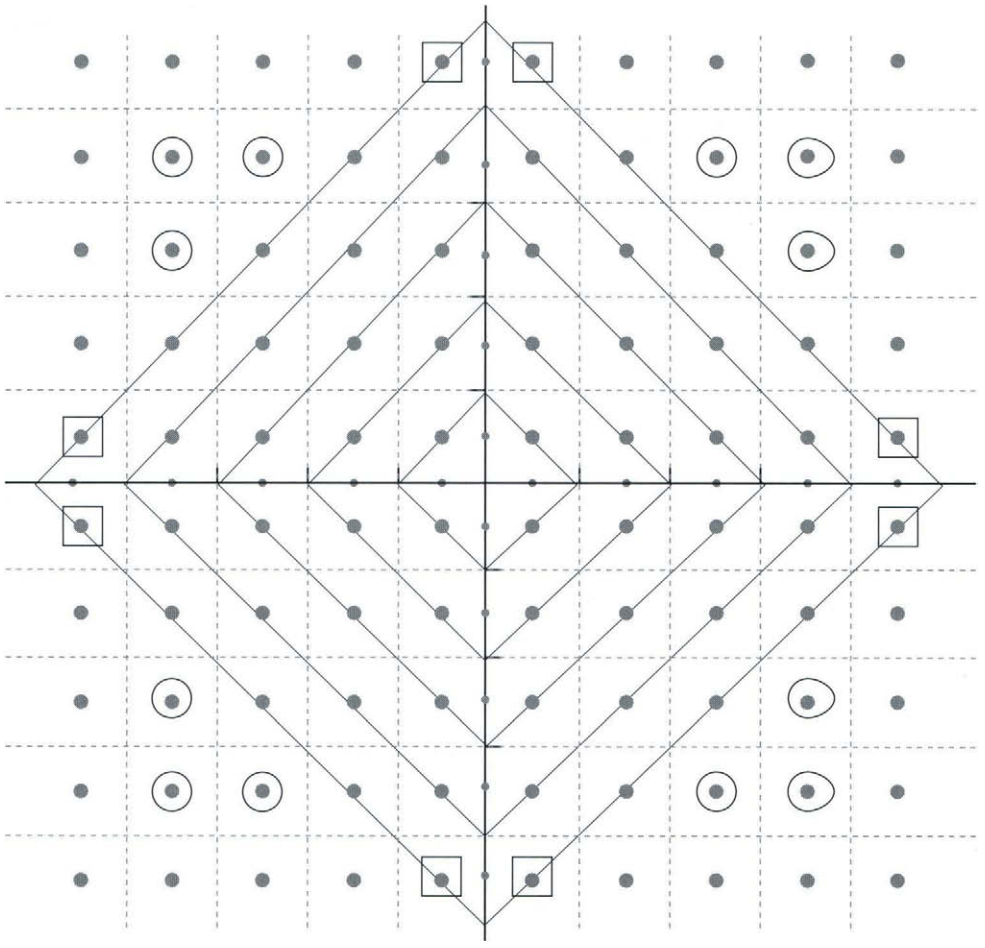

FIGURE 10.22 Contours of constant probability.

10.6.1 Pyramid Vector Quantization

As the dimension of the input vector increases, something interesting happens. Suppose we are quantizing a random variable X with *pdf* $f_X(X)$ and differential entropy $h(X)$. Suppose we block samples of this random variable into a vector $\mathbf{X}$. A result of Shannon's, called the *asymptotic equipartition property* (AEP), states that for sufficiently large L and arbitrarily small ϵ

$$\left| \frac{\log f_{\mathbf{X}}(\mathbf{X})}{L} + h(X) \right| < \epsilon \tag{10.6}$$

for all but a set of vectors with a vanishingly small probability [7]. This means that almost all the L-dimensional vectors will lie on a contour of constant probability given by

$$\left|\frac{\log f_{\mathbf{X}}(\mathbf{X})}{L}\right| = -h(X). \tag{10.7}$$

Given that this is the case, Sakrison [131] suggested that an optimum manner to encode the source would be to distribute 2^{RL} points uniformly in this region. Fischer [132] used this insight to design a vector quantizer called the *pyramid vector quantizer* for the Laplacian source that looks quite similar to the quantizer described in Example 10.6.1. The vector quantizer consists of points of the rectangular quantizer that fall on the hyperpyramid given by

$$\sum_{i=1}^{L} |x_i| = C$$

where C is a constant depending on the variance of the input. Shannon's result is asymptotic, and for realistic values of L, the input vector is generally not localized to a single hyperpyramid.

For this case, Fischer first finds the distance

$$r = \sum_{i=1}^{L} |x_i|.$$

This value is quantized and transmitted to the receiver. The input is normalized by this gain term and quantized using a single hyperpyramid. The quantization process for the shape term consists of two stages: finding the output point on the hyperpyramid closest to the scaled input, and finding a binary codeword for this output point. (See [132] for details about the quantization and coding process.) This approach is quite successful, and for a rate of 3 bits per sample and a vector dimension of 16, we get an SNR value of 16.32 dB. If we increase the vector dimension to 64, we get an SNR value of 17.03. Compared to the SNR obtained from using a nonuniform scalar quantizer, this is an improvement of more than 4 dB.

Notice that in this approach we separated the input vector into a *gain* term and a pattern or *shape* term. Quantizers of this form are called *gain-shape vector quantizers*, or *product code vector quantizers* [133].

10.6.2 Polar and Spherical Vector Quantizers

For the Gaussian distribution, the contours of constant probability are circles in two dimensions and spheres and hyperspheres in three and higher dimensions. In two dimensions, we can quantize the input vector by first transforming it into polar coordinates r and θ:

$$r = \sqrt{x_1^2 + x_2^2} \tag{10.8}$$

and

$$\theta = \tan^{-1}\frac{x_2}{x_1}. \tag{10.9}$$

r and θ can then be either quantized independently [134], or we can use the quantized value of r as an index to a quantizer for θ [135]. The former is known as a polar quantizer; the latter, an unrestricted polar quantizer. The advantage to quantizing r and θ independently is one of simplicity. The quantizers for r and θ are independent scalar quantizers. However, the performance of the polar quantizers is not significantly higher than that of scalar quantization of the components of the two-dimensional vector. The unrestricted polar quantizer has a more complex implementation, as the quantization of θ depends on the quantization of r. However, the performance is also somewhat better than the polar quantizer. The polar quantizer can be extended to three or more dimensions [136].

10.6.3 Lattice Vector Quantizers

Recall that quantization error is composed of two kinds of error, overload error and granular error. The overload error is determined by the location of the quantization regions furthest from the origin, or the boundary. We have seen how we can design vector quantizers to reduce the overload probability and thus the overload error. We called this the boundary gain of vector quantization. In scalar quantization, the granular error was determined by the size of the quantization interval. In vector quantization, the granular error is affected by the size and shape of the quantization interval.

Consider the square and circular quantization regions shown in Figure 10.23. We show only the quantization region at the origin. These quantization regions need to be distributed in a regular manner over the space of source outputs. However, for now, let us simply consider the quantization region at the origin. Let's assume they both have the same area so that we can compare them. This way it would require the same number of quantization regions to cover a given area. That is, we will be comparing two quantization regions of the same "size." To have an area of one, the square has to have sides of length one. As the area of a circle is given by πr^2, the radius of the circle is $\frac{1}{\sqrt{\pi}}$. The maximum quantization error possible with the square quantization region is when the input is at one of the four corners of the square. In this case, the error is $\frac{1}{\sqrt{2}}$, or about 0.707. For the circular quantization region, the maximum error occurs when the input falls on the boundary of the circle. In this case, the error is $\frac{1}{\sqrt{\pi}}$, or about 0.56. Thus, the maximum granular error is larger for the square region than the circular region.

In general, we are more concerned with the average squared error than the maximum error. If we compute the average squared error for the square region, we obtain

$$\int_{\text{Square}} \|X\|^2 \, dX = 0.166\bar{6}.$$

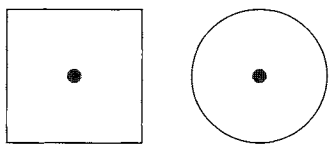

FIGURE 10.23 Possible quantization regions.

For the circle, we obtain

$$\int_{\text{Circle}} \|X\|^2 \, dX = 0.159.$$

Thus, the circular region would introduce less granular error than the square region.

Our choice seems to be clear; we will use the circle as the quantization region. Unfortunately, a basic requirement for the quantizer is that for every possible input vector there should be a unique output vector. In order to satisfy this requirement and have a quantizer with sufficient structure that can be used to reduce the storage space, a union of translates of the quantization region should cover the output space of the source. In other words, the quantization region should *tile* space. A two-dimensional region can be tiled by squares, but it cannot be tiled by circles. If we tried to tile the space with circles, we would either get overlaps or holes.

Apart from squares, other shapes that tile space include rectangles and hexagons. It turns out that the best shape to pick for a quantization region in two dimensions is a hexagon [137].

In two dimensions, it is relatively easy to find the shapes that tile space, then select the one that gives the smallest amount of granular error. However, when we start looking at higher dimensions, it is difficult, if not impossible, to visualize different shapes, let alone find which ones tile space. An easy way out of this dilemma is to remember that a quantizer can be completely defined by its output points. In order for this quantizer to possess structure, these points should be spaced in some regular manner.

Regular arrangements of output points in space are called *lattices*. Mathematically, we can define a lattice as follows:

Let $\{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_L\}$ be L independent L-dimensional vectors. Then the set

$$\mathcal{L} = \left\{ \mathbf{x} : \mathbf{x} = \sum_{i=1}^{L} u_i \mathbf{a}_i \right\} \tag{10.10}$$

is a lattice if $\{u_i\}$ are all integers.

When a subset of lattice points is used as the output points of a vector quantizer, the quantizer is known as a *lattice vector quantizer*. From this definition, the pyramid vector quantizer described earlier can be viewed as a lattice vector quantizer. Basing a quantizer on a lattice solves the storage problem. As any lattice point can be regenerated if we know the basis set, there is no need to store the output points. Further, the highly structured nature of lattices makes finding the closest output point to an input relatively simple. Note that what we give up when we use lattice vector quantizers is the clustering property of LBG quantizers.

Let's take a look at a few examples of lattices in two dimensions. If we pick $a_1 = (1, 0)$ and $a_2 = (0, 1)$, we obtain the integer lattice—the lattice that contains all points in two dimensions whose coordinates are integers.

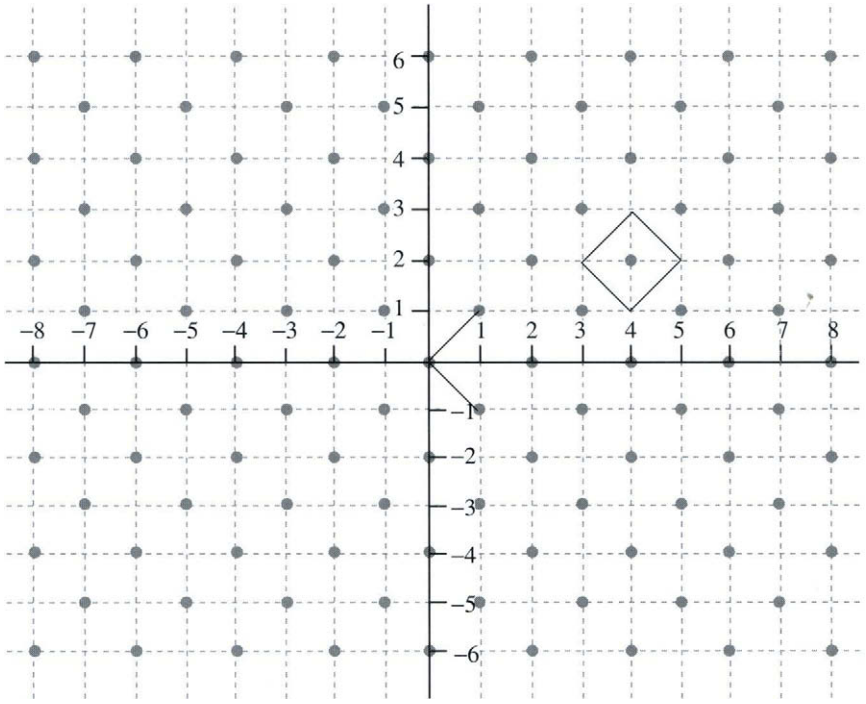

FIGURE 10.24 The D_2 lattice.

If we pick $a_1 = (1, 1)$ and $a_2 = (1, -1)$, we get the lattice shown in Figure 10.24. This lattice has a rather interesting property. Any point in the lattice is given by $na_1 + ma_2$, where n and m are integers. But

$$na_1 + ma_2 = \begin{bmatrix} n+m \\ n-m \end{bmatrix}$$

and the sum of the coefficients is $n+m+n-m = 2n$, which is even for all n. Therefore, all points in this lattice have an even coordinate sum. Lattices with these properties are called *D lattices*.

Finally, if $a_1 = (1, 0)$ and $a_2 = \left(-\frac{1}{2}, \frac{\sqrt{3}}{2}\right)$, we get the hexagonal lattice shown in Figure 10.25. This is an example of an *A lattice*.

There are a large number of lattices that can be used to obtain lattice vector quantizers. In fact, given a dimension L, there are an infinite number of possible sets of L independent vectors. Among these, we would like to pick the lattice that produces the greatest reduction in granular noise. When comparing the square and circle as candidates for quantization regions, we used the integral over the shape of $\|X\|^2$. This is simply the second moment of the shape. The shape with the smallest second moment for a given volume is known to be the circle in two dimensions and the sphere and hypersphere in higher dimensions [138]. Unfortunately, circles and spheres cannot tile space; either there will be overlap or there will

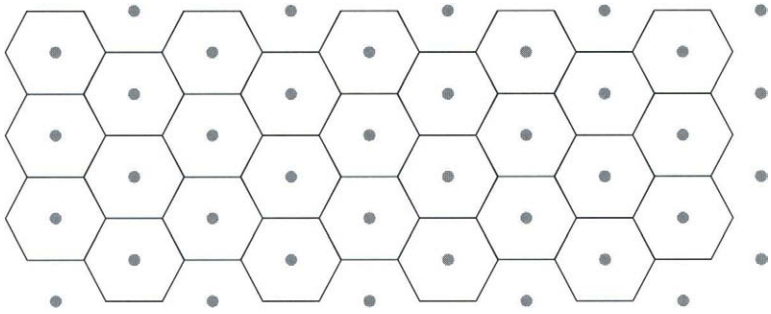

FIGURE 10.25 The A_2 lattice.

be holes. As the ideal case is unattainable, we can try to approximate it. We can look for ways of arranging spheres so that they cover space with minimal overlap [139], or look for ways of packing spheres with the least amount of space left over [138]. The centers of these spheres can then be used as the output points. The quantization regions will not be spheres, but they may be close approximations to spheres.

The problems of sphere covering and sphere packing are widely studied in a number of different areas. Lattices discovered in these studies have also been useful as vector quantizers [138]. Some of these lattices, such as the A_2 and D_2 lattices described earlier, are based on the root systems of Lie algebras [140]. The study of Lie algebras is beyond the scope of this book; however, we have included a brief discussion of the root systems and how to obtain the corresponding lattices in Appendix C.

One of the nice things about root lattices is that we can use their structural properties to obtain fast quantization algorithms. For example, consider building a quantizer based on the D_2 lattice. Because of the way in which we described the D_2 lattice, the size of the lattice is fixed. We can change the size by picking the basis vectors as (Δ, Δ) and $(\Delta, -\Delta)$, instead of $(1, 1)$ and $(1, -1)$. We can have exactly the same effect by dividing each input by Δ before quantization, and then multiplying the reconstruction values by Δ. Suppose we pick the latter approach and divide the components of the input vector by Δ. If we wanted to find the closest lattice point to the input, all we need to do is find the closest integer to each coordinate of the scaled input. If the sum of these integers is even, we have a lattice point. If not, find the coordinate that incurred the largest distortion during conversion to an integer and then find the next closest integer. The sum of coordinates of this new vector differs from the sum of coordinates of the previous vector by one. Therefore, if the sum of coordinates of the previous vector was odd, the sum of the coordinates of the current vector will be even, and we have the closest lattice point to the input.

Example 10.6.2:

Suppose the input vector is given by (2.3, 1.9). Rounding each coefficient to the nearest integer, we get the vector (2, 2). The sum of the coordinates is even; therefore, this is the closest lattice point to the input.

Suppose the input was (3.4, 1.8). Rounding the components to the nearest integer, we get (3, 2). The sum of the components is 5, which is odd. The differences between the components of the input vector and the nearest integer are 0.4 and 0.2. The largest difference was incurred by the first component, so we round it up to the next closest integer, and the resulting vector is (4, 2). The sum of the coordinates is 6, which is even; therefore, this is the closest lattice point. ♦

Many of the lattices have similar properties that can be used to develop fast algorithms for finding the closest output point to a given input [141, 140].

To review our coverage of lattice vector quantization, overload error can be reduced by careful selection of the boundary, and we can reduce the granular noise by selection of the lattice. The lattice also provides us with a way to avoid storage problems. Finally, we can use the structural properties of the lattice to find the closest lattice point to a given input.

Now we need two things: to know how to find the closest *output* point (remember, not all lattice points are output points), and to find a way of assigning a binary codeword to the output point and recovering the output point from the binary codeword. This can be done by again making use of the specific structures of the lattices. While the procedures necessary are simple, explanations of the procedures are lengthy and involved (see [142] and [140] for details).

10.7 Variations on the Theme

Because of its capability to provide high compression with relatively low distortion, vector quantization has been one of the more popular lossy compression techniques over the last decade in such diverse areas as video compression and low-rate speech compression. During this period, several people have come up with variations on the basic vector quantization approach. We briefly look at a few of the more well-known variations here, but this is by no means an exhaustive list. For more information, see [5] and [143].

10.7.1 Gain-Shape Vector Quantization

In some applications such as speech, the dynamic range of the input is quite large. One effect of this is that, in order to be able to represent the various vectors from the source, we need a very large codebook. This requirement can be reduced by normalizing the source output vectors, then quantizing the normalized vector and the normalization factor separately [144, 133]. In this way, the variation due to the dynamic range is represented by the normalization factor or *gain*, while the vector quantizer is free to do what it does best, which is to capture the structure in the source output. Vector quantizers that function in this manner are called *gain-shape vector quantizers*. The pyramid quantizer discussed earlier is an example of a gain-shape vector quantizer.

10.7.2 Mean-Removed Vector Quantization

If we were to generate a codebook from an image, differing amounts of background illumination would result in vastly different codebooks. This effect can be significantly reduced if we remove the mean from each vector before quantization. The mean and the mean-removed vector can then be quantized separately. The mean can be quantized using a scalar quantization scheme, while the mean-removed vector can be quantized using a vector quantizer. Of course, if this strategy is used, the vector quantizer should be designed using mean-removed vectors as well.

Example 10.7.1:

Let us encode the Sinan image using a codebook generated by the Sena image, as we did in Figure 10.16. However, this time we will use a mean-removed vector quantizer. The result is shown in Figure 10.26. For comparison we have also included the reconstructed image from Figure 10.16. Notice the annoying blotches on the shoulder have disappeared. However, the reconstructed image also suffers from more blockiness. The blockiness increases because adding the mean back into each block accentuates the discontinuity at the block boundaries.

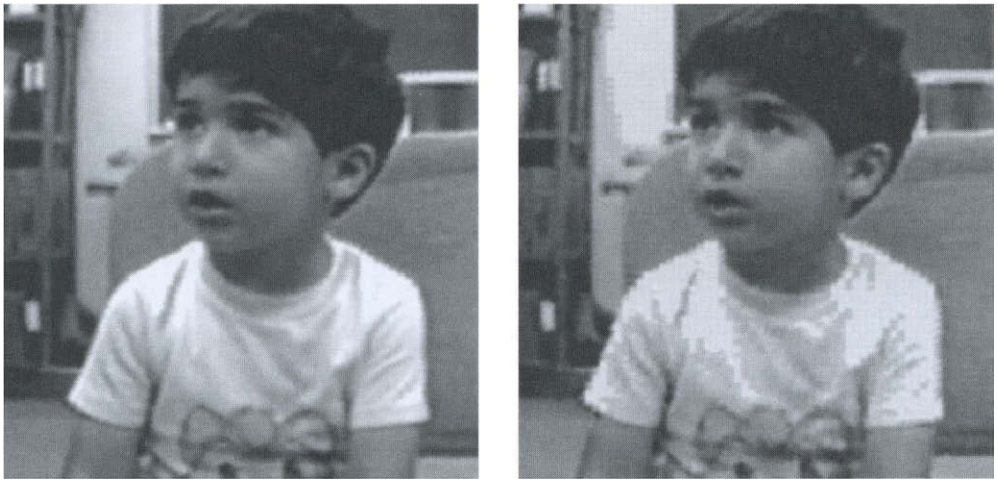

FIGURE 10.26 **Left: Reconstructed image using mean-removed vector quantization and the Sena image as the training set. Right: LBG vector quantization with the Sena image as the training set.**

Each approach has its advantages and disadvantages. Which approach we use in a particular application depends very much on the application. ♦

10.7.3 Classified Vector Quantization

We can sometimes divide the source output into separate classes with different spatial properties. In these cases, it can be very beneficial to design separate vector quantizers for the different classes. This approach, referred to as *classified vector quantization*, is especially useful in image compression, where edges and nonedge regions form two distinct classes. We can separate the training set into vectors that contain edges and vectors that do not. A separate vector quantizer can be developed for each class. During the encoding process, the vector is first tested to see if it contains an edge. A simple way to do this is to check the variance of the pixels in the vector. A large variance will indicate the presence of an edge. More sophisticated techniques for edge detection can also be used. Once the vector is classified, the corresponding codebook can be used to quantize the vector. The encoder transmits both the label for the codebook used and the label for the vector in the codebook [145].

A slight variation of this strategy is to use different kinds of quantizers for the different classes of vectors. For example, if certain classes of source outputs require quantization at a higher rate than is possible using LBG vector quantizers, we can use lattice vector quantizers. An example of this approach can be found in [146].

10.7.4 Multistage Vector Quantization

Multistage vector quantization [147] is an approach that reduces both the encoding complexity and the memory requirements for vector quantization, especially at high rates. In this approach, the input is quantized in several stages. In the first stage, a low-rate vector quantizer is used to generate a coarse approximation of the input. This coarse approximation, in the form of the label of the output point of the vector quantizer, is transmitted to the receiver. The error between the original input and the coarse representation is quantized by the second-stage quantizer, and the label of the output point is transmitted to the receiver. In this manner, the input to the nth-stage vector quantizer is the difference between the original input and the reconstruction obtained from the outputs of the preceding $n-1$ stages. The difference between the input to a quantizer and the reconstruction value is often called the *residual*, and the multistage vector quantizers are also known as *residual vector quantizers* [148]. The reconstructed vector is the sum of the output points of each of the stages. Suppose we have a three-stage vector quantizer, with the three quantizers represented by $\mathbf{Q}_1$, $\mathbf{Q}_2$, and $\mathbf{Q}_3$. Then for a given input $\mathbf{X}$, we find

$$\begin{aligned}\mathbf{Y}_1 &= \mathbf{Q}_1(\mathbf{X})\\ \mathbf{Y}_2 &= \mathbf{Q}_2(\mathbf{X}-\mathbf{Q}_1(\mathbf{X}))\\ \mathbf{Y}_3 &= \mathbf{Q}_3(\mathbf{X}-\mathbf{Q}_1(\mathbf{X})-\mathbf{Q}_2(\mathbf{X}-\mathbf{Q}_1(\mathbf{X}))).\end{aligned} \tag{10.11}$$

The reconstruction $\hat{\mathbf{X}}$ is given by

$$\hat{\mathbf{X}} = \mathbf{Y}_1+\mathbf{Y}_2+\mathbf{Y}_3. \tag{10.12}$$

This process is shown in Figure 10.27.

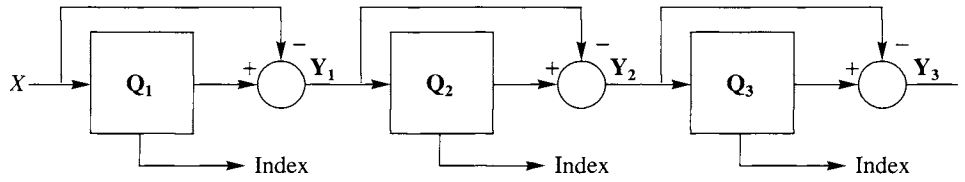

FIGURE 10.27 A three-stage vector quantizer.

If we have K stages, and the codebook size of the nth-stage vector quantizer is L_n, then the effective size of the overall codebook is $L_1 \times L_2 \times \cdots \times L_K$. However, we need to store only $L_1 + L_2 + \cdots + L_K$ vectors, which is also the number of comparisons required. Suppose we have a five-stage vector quantizer, each with a codebook size of 32, meaning that we would have to store 160 codewords. This would provide an effective codebook size of $32^5 = 33,554,432$. The computational savings are also of the same order.

This approach allows us to use vector quantization at much higher rates than we could otherwise. However, at rates at which it is feasible to use LBG vector quantizers, the performance of the multistage vector quantizers is generally lower than the LBG vector quantizers [5]. The reason for this is that after the first few stages, much of the structure used by the vector quantizer has been removed, and the vector quantization advantage that depends on this structure is not available. Details on the design of residual vector quantizers can be found in [148, 149].

There may be some vector inputs that can be well represented by fewer stages than others. A multistage vector quantizer with a variable number of stages can be implemented by extending the idea of recursively indexed scalar quantization to vectors. It is not possible to do this directly because there are some fundamental differences between scalar and vector quantizers. The input to a scalar quantizer is assumed to be *iid*. On the other hand, the vector quantizer can be viewed as a pattern-matching algorithm [150]. The input is assumed to be one of a number of different patterns. The scalar quantizer is used after the redundancy has been removed from the source sequence, while the vector quantizer takes advantage of the redundancy in the data.

With these differences in mind, the recursively indexed vector quantizer (RIVQ) can be described as a two-stage process. The first stage performs the normal pattern-matching function, while the second stage recursively quantizes the residual if the magnitude of the residual is greater than some prespecified threshold. The codebook of the second stage is ordered so that the magnitude of the codebook entries is a nondecreasing function of its index. We then choose an index I that will determine the mode in which the RIVQ operates.

The quantization rule Q, for a given input value $\mathbf{X}$, is as follows:

- Quantize $\mathbf{X}$ with the first-stage quantizer Q_1.
- If the residual $\|\mathbf{X} - \mathbf{Q}_1(\mathbf{X})\|$ is below a specified threshold, then $\mathbf{Q}_1(\mathbf{X})$ is the nearest output level.

- Otherwise, generate $\mathbf{X}_1 = \mathbf{X} - \mathbf{Q}_1(\mathbf{X})$ and quantize using the second-stage quantizer $\mathbf{Q}_2$. Check if the index J_1 of the output is below the index I. If so,

$$\mathbf{Q}(\mathbf{X}) = \mathbf{Q}_1(\mathbf{X}) + \mathbf{Q}_2(\mathbf{X}_1).$$

  If not, form

$$\mathbf{X}_2 = \mathbf{X}_1 - \mathbf{Q}(\mathbf{X}_1)$$

  and do the same for $\mathbf{X}_2$ as we did for $\mathbf{X}_1$.

This process is repeated until for some m, the index J_m falls below the index I, in which case $\mathbf{X}$ will be quantized to

$$\mathbf{Q}(\mathbf{X}) = \mathbf{Q}_1(\mathbf{X}) + \mathbf{Q}_2(\mathbf{X}_1) + \cdots + \mathbf{Q}_2(\mathbf{X}_M).$$

Thus, the RIVQ operates in two modes: when the index J of the quantized input falls below a given index I and when the index J falls above the index I.

Details on the design and performance of the recursively indexed vector quantizer can be found in [151, 152].

10.7.5 Adaptive Vector Quantization

While LBG vector quantizers function by using the structure in the source output, this reliance on the use of the structure can also be a drawback when the characteristics of the source change over time. For situations like these, we would like to have the quantizer adapt to the changes in the source output.

For mean-removed and gain-shape vector quantizers, we can adapt the scalar aspect of the quantizer, that is, the quantization of the mean or the gain using the techniques discussed in the previous chapter. In this section, we look at a few approaches to adapting the codebook of the vector quantizer to changes in the characteristics of the input.

One way of adapting the codebook to changing input characteristics is to start with a very large codebook designed to accommodate a wide range of source characteristics [153]. This large codebook can be ordered in some manner known to both transmitter and receiver. Given a sequence of input vectors to be quantized, the encoder can select a subset of the larger codebook to be used. Information about which vectors from the large codebook were used can be transmitted as a binary string. For example, if the large codebook contained 10 vectors, and the encoder was to use the second, third, fifth, and ninth vectors, we would send the binary string 0110100010, with a 1 representing the position of the codeword used in the large codebook. This approach permits the use of a small codebook that is matched to the local behavior of the source.

This approach can be used with particular effectiveness with the recursively indexed vector quantizer [151]. Recall that in the recursively indexed vector quantizer, the quantized output is always within a prescribed distance of the inputs, determined by the index I. This means that the set of output values of the RIVQ can be viewed as an accurate representation of the inputs and their statistics. Therefore, we can treat a subset of the output set of the previous intervals as our large codebook. We can then use the method described in [153] to

inform the receiver of which elements of the previous outputs form the codebook for the next interval. This method (while not the most efficient) is quite simple. Suppose an output set, in order of first appearance, is $\{p, a, q, s, l, t, r\}$, and the desired codebook for the interval to be encoded is $\{a, q, l, r\}$. Then we would transmit the binary string 0110101 to the receiver. The 1s correspond to the letters in the output set, which would be elements of the desired codebook. We select the subset for the current interval by finding the closest vectors from our collection of past outputs to the input vectors of the current set. This means that there is an inherent delay of one interval imposed by this approach. The overhead required to send the codebook selection is M/N, where M is the number of vectors in the output set and N is the interval size.

Another approach to updating the codebook is to check the distortion incurred while quantizing each input vector. Whenever this distortion is above some specified threshold, a different higher-rate mechanism is used to encode the input. The higher-rate mechanism might be the scalar quantization of each component, or the use of a high-rate lattice vector quantizer. This quantized representation of the input is transmitted to the receiver and, at the same time, added to both the encoder and decoder codebooks. In order to keep the size of the codebook the same, an entry must be discarded when a new vector is added to the codebook. Selecting an entry to discard is handled in a number of different ways. Variations of this approach have been used for speech coding, image coding, and video coding (see [154, 155, 156, 157, 158] for more details).

10.8 Trellis-Coded Quantization

Finally, we look at a quantization scheme that appears to be somewhat different from other vector quantization schemes. In fact, some may argue that it is not a vector quantizer at all. However, the trellis-coded quantization (TCQ) algorithm gets its performance advantage by exploiting the statistical structure exploited by the lattice vector quantizer. Therefore, we can argue that it should be classified as a vector quantizer.

The trellis-coded quantization algorithm was inspired by the appearance of a revolutionary concept in modulation called trellis-coded modulation (TCM). The TCQ algorithm and its entropy-constrained variants provide some of the best performance when encoding random sources. This quantizer can be viewed as a vector quantizer with very large dimension, but a restricted set of values for the components of the vectors.

Like a vector quantizer, the TCQ quantizes sequences of source outputs. Each element of a sequence is quantized using 2^R reconstruction levels selected from a set of 2^{R+1} reconstruction levels, where R is the number of bits per sample used by a trellis-coded quantizer. The 2^R element subsets are predefined; which particular subset is used is based on the reconstruction level used to quantize the previous quantizer input. However, the TCQ algorithm allows us to postpone a decision on which reconstruction level to use until we can look at a sequence of decisions. This way we can select the sequence of decisions that gives us the lowest amount of average distortion.

Let's take the case of a 2-bit quantizer. As described above, this means that we will need 2^3, or 8, reconstruction levels. Let's label these reconstruction levels as shown in Figure 10.28. The set of reconstruction levels is partitioned into two subsets: one consisting

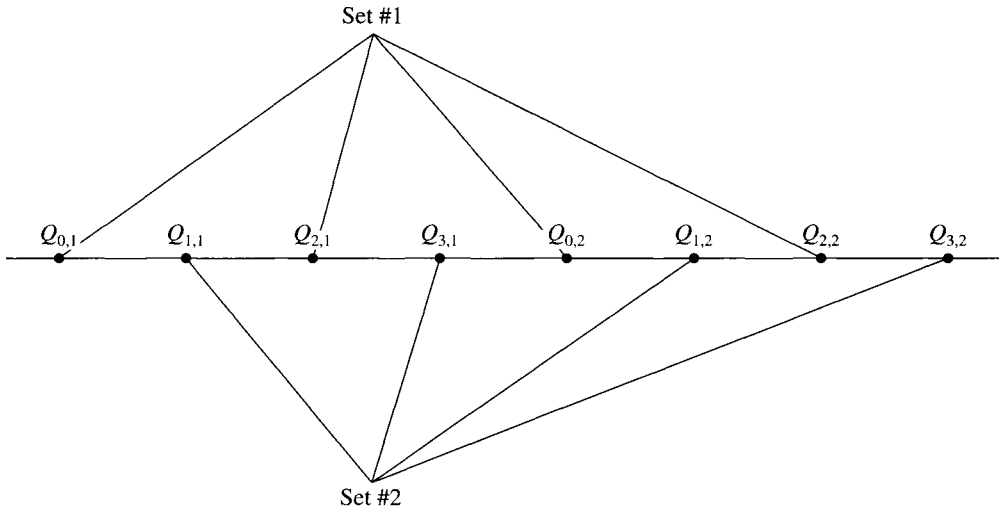

FIGURE 10.28 Reconstruction levels for a 2-bit trellis-coded quantizer.

of the reconstruction values labeled $Q_{0,i}$ and $Q_{2,i}$, and the remainder comprising the second set. We use the first set to perform the quantization if the previous quantization level was one labeled $Q_{0,i}$ or $Q_{1,i}$; otherwise, we use the second set. Because the current reconstructed value defines the subset that can be used to perform the quantization on the next input, sometimes it may be advantageous to actually accept more distortion than necessary for the current sample in order to have less distortion in the next quantization step. In fact, at times it may be advantageous to accept poor quantization for several samples so that several samples down the line the quantization can result in less distortion. If you have followed this reasoning, you can see how we might be able to get lower overall distortion by looking at the quantization of an entire sequence of source outputs. The problem with delaying a decision is that the number of choices increases exponentially with each sample. In the 2-bit example, for the first sample we have four choices; for each of these four choices we have four choices for the second sample. For each of these 16 choices we have four choices for the third sample, and so on. Luckily, there is a technique that can be used to keep this explosive growth of choices under control. The technique, called the *Viterbi algorithm* [159], is widely used in error control coding.

In order to explain how the Viterbi algorithm works, we will need to formalize some of what we have been discussing. The sequence of choices can be viewed in terms of a state diagram. Let's suppose we have four states: S_0, S_1, S_2, and S_3. We will say we are in state S_k if we use the reconstruction levels $Q_{k,1}$ or $Q_{k,2}$. Thus, if we use the reconstruction levels $Q_{0,i}$, we are in state S_0. We have said that we use the elements of Set #1 if the previous quantization levels were $Q_{0,i}$ or $Q_{1,i}$. As Set #1 consists of the quantization levels $Q_{0,i}$ and $Q_{2,i}$, this means that we can go from state S_0 and S_1 to states S_0 and S_2. Similarly, from states S_2 and S_3 we can only go to states S_1 and S_3. The state diagram can be drawn as shown in Figure 10.29.

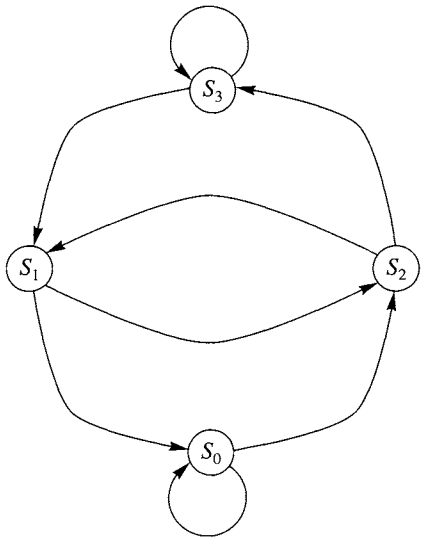

FIGURE 10. 29 State diagram for the selection process.

Let's suppose we go through two sequences of choices that converge to the same state, after which both sequences are identical. This means that the sequence of choices that had incurred a higher distortion at the time the two sequences converged will have a higher distortion from then on. In the end we will select the sequence of choices that results in the lowest distortion; therefore, there is no point in continuing to keep track of a sequence that we will discard anyway. This means that whenever two sequences of choices converge, we can discard one of them. How often does this happen? In order to see this, let's introduce time into our state diagram. The state diagram with the element of time introduced into it is called a *trellis diagram*. The trellis for this particular example is shown in Figure 10.30. At each time instant, we can go from one state to two other states. And, at each step we

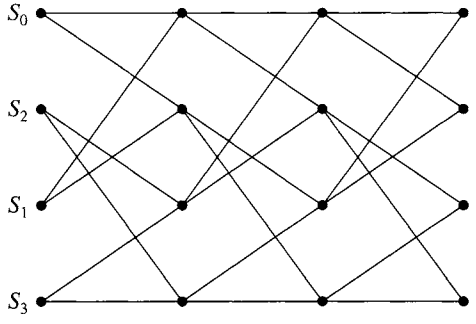

FIGURE 10. 30 Trellis diagram for the selection process.

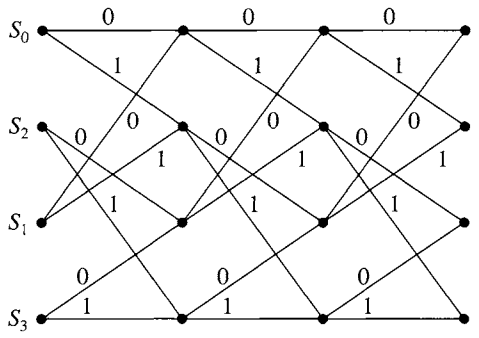

FIGURE 10.31 **Trellis diagram for the selection process with binary labels for the state transitions.**

have two sequences that converge to each state. If we discard one of the two sequences that converge to each state, we can see that, no matter how long a sequence of decisions we use, we will always end up with four sequences.

Notice that, assuming the initial state is known to the decoder, any path through this particular trellis can be described to the decoder using 1 bit per sample. From each state we can only go to two other states. In Figure 10.31, we have marked the branches with the bits used to signal that transition. Given that each state corresponds to two quantization levels, specifying the quantization level for each sample would require an additional bit, resulting in a total of 2 bits per sample. Let's see how all this works together in an example.

Example 10.8.1:

Using the quantizer whose quantization levels are shown in Figure 10.32, we will quantize the sequence of values 0.2, 1.6, 2.3. For the distortion measure we will use the sum of absolute differences. If we simply used the quantization levels marked as Set #1 in Figure 10.28, we would quantize 0.2 to the reconstruction value 0.5, for a distortion of 0.3. The second sample value of 1.6 would be quantized to 2.5, and the third sample value of 2.3 would also be quantized to 2.5, resulting in a total distortion of 1.4. If we used Set #2 to quantize these values, we would end up with a total distortion of 1.6. Let's see how much distortion results when using the TCQ algorithm.

We start by quantizing the first sample using the two quantization levels $Q_{0,1}$ and $Q_{0,2}$. The reconstruction level $Q_{0,2}$, or 0.5, is closer and results in an absolute difference of 0.3. We mark this on the first node corresponding to S_0. We then quantize the first sample using

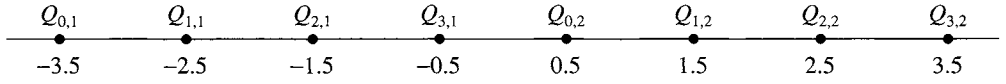

FIGURE 10.32 **Reconstruction levels for a 2-bit trellis-coded quantizer.**

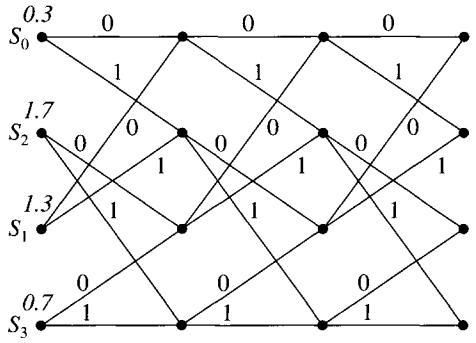

FIGURE 10.33 Quantizing the first sample.

$Q_{1,1}$ and $Q_{1,2}$. The closest reconstruction value is $Q_{1,2}$, or 1.5, which results in a distortion value of 1.3. We mark the first node corresponding to S_1. Continuing in this manner, we get a distortion value of 1.7 when we use the reconstruction levels corresponding to state S_2 and a distortion value of 0.7 when we use the reconstruction levels corresponding to state S_3. At this point the trellis looks like Figure 10.33. Now we move on to the second sample. Let's first quantize the second sample value of 1.6 using the quantization levels associated with state S_0. The reconstruction levels associated with state S_0 are -3.5 and 0.5. The closest value to 1.6 is 0.5. This results in an absolute difference for the second sample of 1.1. We can reach S_0 from S_0 and from S_1. If we accept the first sample reconstruction corresponding to S_0, we will end up with an accumulated distortion of 1.4. If we accept the reconstruction corresponding to state S_1, we get an accumulated distortion of 2.4. Since the accumulated distortion is less if we accept the transition from state S_0, we do so and discard the transition from state S_1. Continuing in this fashion for the remaining states, we end up with the situation depicted in Figure 10.34. The sequence of decisions that have been terminated are shown by an X on the branch corresponding to the particular transition. The accumulated distortion is listed at each node. Repeating this procedure for the third sample value of 2.3, we obtain the

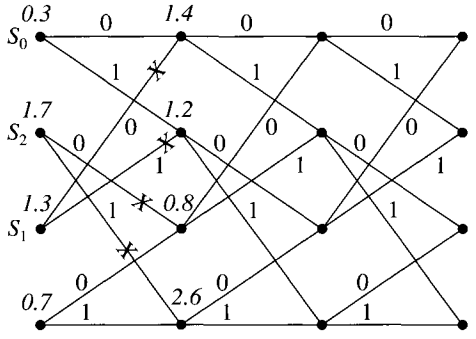

FIGURE 10.34 Quantizing the second sample.

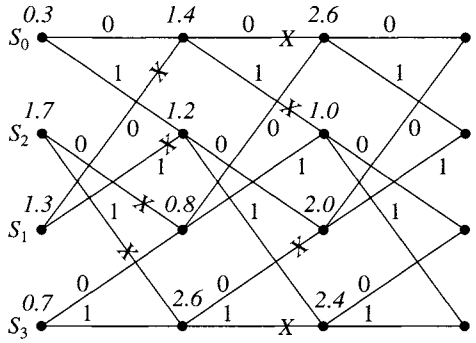

FIGURE 10.35 Quantizing the third sample.

trellis shown in Figure 10.35. If we wanted to terminate the algorithm at this time, we could pick the sequence of decisions with the smallest accumulated distortion. In this particular example, the sequence would be S_3, S_1, S_2. The accumulated distortion is 1.0, which is less than what we would have obtained using either Set #1 or Set #2. ♦

10.9 Summary

In this chapter we introduced the technique of vector quantization. We have seen how we can make use of the structure exhibited by groups, or vectors, of values to obtain compression. Because there are different kinds of structure in different kinds of data, there are a number of different ways to design vector quantizers. Because data from many sources, when viewed as vectors, tend to form clusters, we can design quantizers that essentially consist of representations of these clusters. We also described aspects of the design of vector quantizers and looked at some applications. Recent literature in this area is substantial, and we have barely skimmed the surface of the large number of interesting variations of this technique.

Further Reading

The subject of vector quantization is dealt with extensively in the book *Vector Quantization and Signal Compression*, by A. Gersho and R.M. Gray [5]. There is also an excellent collection of papers called *Vector Quantization*, edited by H. Abut and published by IEEE Press [143].

There are a number of excellent tutorial articles on this subject:

1. "Vector Quantization," by R.M. Gray, in the April 1984 issue of *IEEE Acoustics, Speech, and Signal Processing Magazine* [160].
2. "Vector Quantization: A Pattern Matching Technique for Speech Coding," by A. Gersho and V. Cuperman, in the December 1983 issue of *IEEE Communications Magazine* [150].

3. "Vector Quantization in Speech Coding," by J. Makhoul, S. Roucos, and H. Gish, in the November 1985 issue of the *Proceedings of the IEEE* [161].
4. "Vector Quantization," by P.F. Swaszek, in *Communications and Networks*, edited by I.F. Blake and H.V. Poor [162].
5. A survey of various image-coding applications of vector quantization can be found in "Image Coding Using Vector Quantization: A Review," by N.M. Nasrabadi and R.A. King, in the August 1988 issue of the *IEEE Transactions on Communications* [163].
6. A thorough review of lattice vector quantization can be found in "Lattice Quantization," by J.D. Gibson and K. Sayood, in *Advances in Electronics and Electron Physics* [140].

The area of vector quantization is an active one, and new techniques that use vector quantization are continually being developed. The journals that report work in this area include *IEEE Transactions on Information Theory*, *IEEE Transactions on Communications*, *IEEE Transactions on Signal Processing*, and *IEEE Transactions on Image Processing*, among others.

10.10 Projects and Problems

1. In Example 10.3.2 we increased the SNR by about 0.3 dB by moving the top-left output point to the origin. What would happen if we moved the output points at the four corners to the positions $(\pm\Delta, 0)$, $(0, \pm\Delta)$. As in the example, assume the input has a Laplacian distribution with mean zero and variance one, and $\Delta = 0.7309$. You can obtain the answer analytically or through simulation.
2. For the quantizer of the previous problem, rather than moving the output points to $(\pm\Delta, 0)$ and $(0, \pm\Delta)$, we could have moved them to other positions that might have provided a larger increase in SNR. Write a program to test different (reasonable) possibilities and report on the best and worst cases.
3. In the program `trainvq.c` the empty cell problem is resolved by replacing the vector with no associated training set vectors with a training set vector from the quantization region with the largest number of vectors. In this problem, we will investigate some possible alternatives.

   Generate a sequence of pseudorandom numbers with a triangular distribution between 0 and 2. (You can obtain a random number with a triangular distribution by adding two uniformly distributed random numbers.) Design an eight-level, two-dimensional vector quantizer with the initial codebook shown in Table 10.9.

   (a) Use the `trainvq` program to generate a codebook with 10,000 random numbers as the training set. Comment on the final codebook you obtain. Plot the elements of the codebook and discuss why they ended up where they did.

   (b) Modify the program so that the empty cell vector is replaced with a vector from the quantization region with the largest distortion. Comment on any changes in

TABLE 10.9 Initial codebook for Problem 3.

1	1
1	2
1	0.5
0.5	1
0.5	0.5
1.5	1
2	5
3	3

the distortion (or lack of change). Is the final codebook different from the one you obtained earlier?

(c) Modify the program so that whenever an empty cell problem arises, a two-level quantizer is designed for the quantization region with the largest number of output points. Comment on any differences in the codebook and distortion from the previous two cases.

4. Generate a 16-dimensional codebook of size 64 for the Sena image. Construct the vector as a 4×4 block of pixels, an 8×2 block of pixels, and a 16×1 block of pixels. Comment on the differences in the mean squared errors and the quality of the reconstructed images. You can use the program `trvqsp_img` to obtain the codebooks.

5. In Example 10.6.1 we designed a 60-level two-dimensional quantizer by taking the two-dimensional representation of an 8-level scalar quantizer, removing 12 output points from the 64 output points, and adding 8 points in other locations. Assume the input is Laplacian with zero mean and unit variance, and $\Delta = 0.7309$.

(a) Calculate the increase in the probability of overload by the removal of the 12 points from the original 64.

(b) Calculate the decrease in overload probability when we added the 8 new points to the remaining 52 points.

6. In this problem we will compare the performance of a 16-dimensional pyramid vector quantizer and a 16-dimensional LBG vector quantizer for two different sources. In each case the codebook for the pyramid vector quantizer consists of 272 elements:

- 32 vectors with 1 element equal to $\pm\Delta$, and the other 15 equal to zero, and
- 240 vectors with 2 elements equal to $\pm\Delta$ and the other 14 equal to zero.

The value of Δ should be adjusted to give the best performance. The codebook for the LBG vector quantizer will be obtained by using the program `trvqsp_img` on the source output. You will have to modify `trvqsp_img` slightly to give you a codebook that is not a power of two.

(a) Use the two quantizers to quantize a sequence of 10,000 zero mean unit variance Laplacian random numbers. Using either the mean squared error or the SNR as a measure of performance, compare the performance of the two quantizers.

(b) Use the two quantizers to quantize the Sinan image. Compare the two quantizers using either the mean squared error or the SNR and the reconstructed image. Compare the difference between the performance of the two quantizers with the difference when the input was random.