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

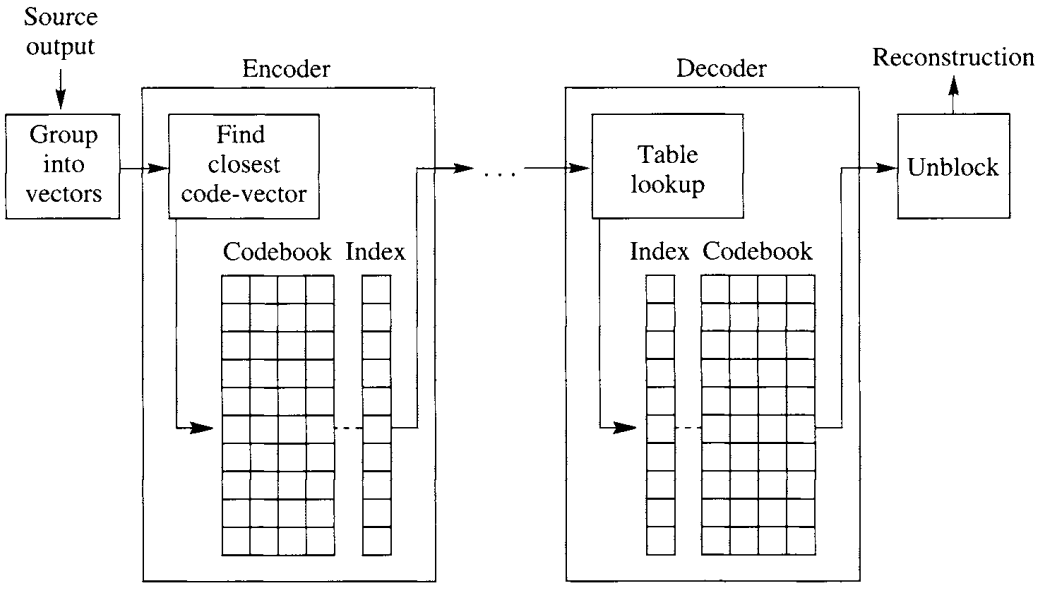

FIGURE 10.1 **The vector quantization procedure.**

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.

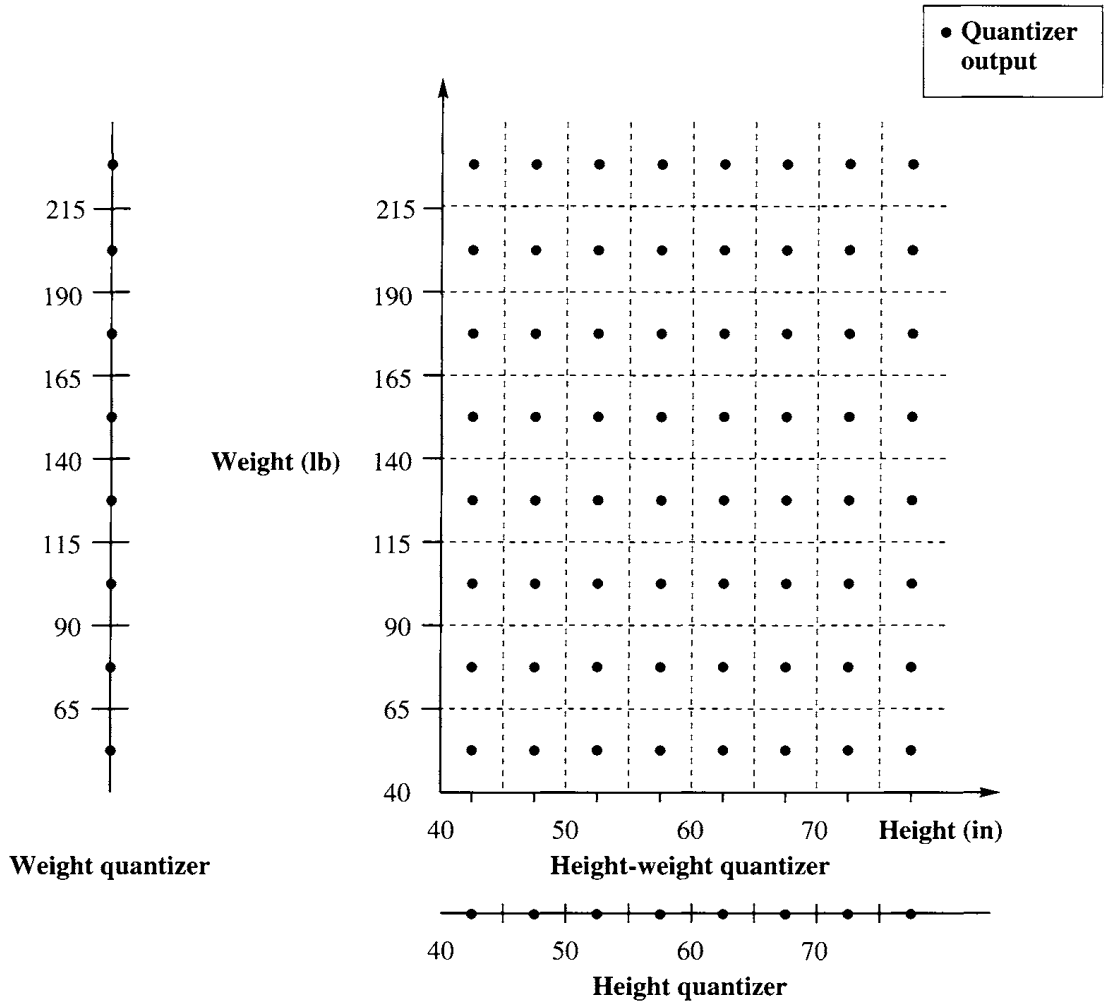

FIGURE 10.2 **The height/weight scalar quantizers when viewed in two dimensions.**

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

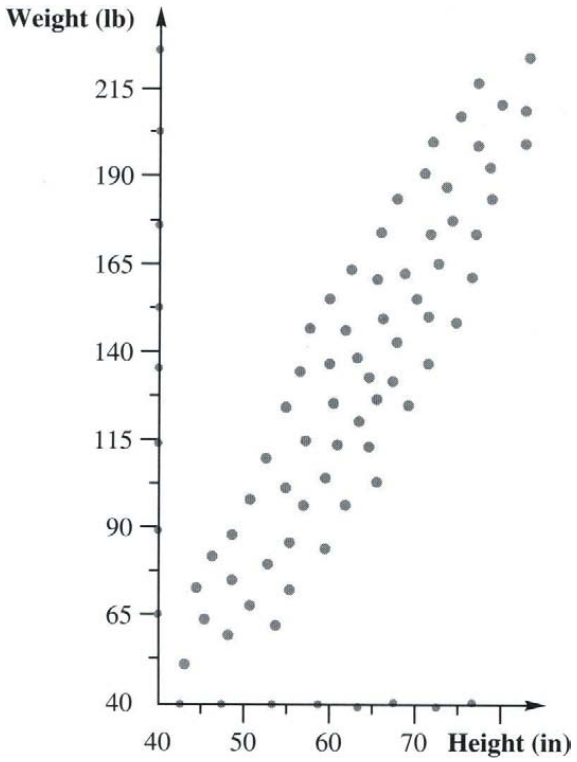

FIGURE 10.3 **The height-weight vector quantizer.**

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

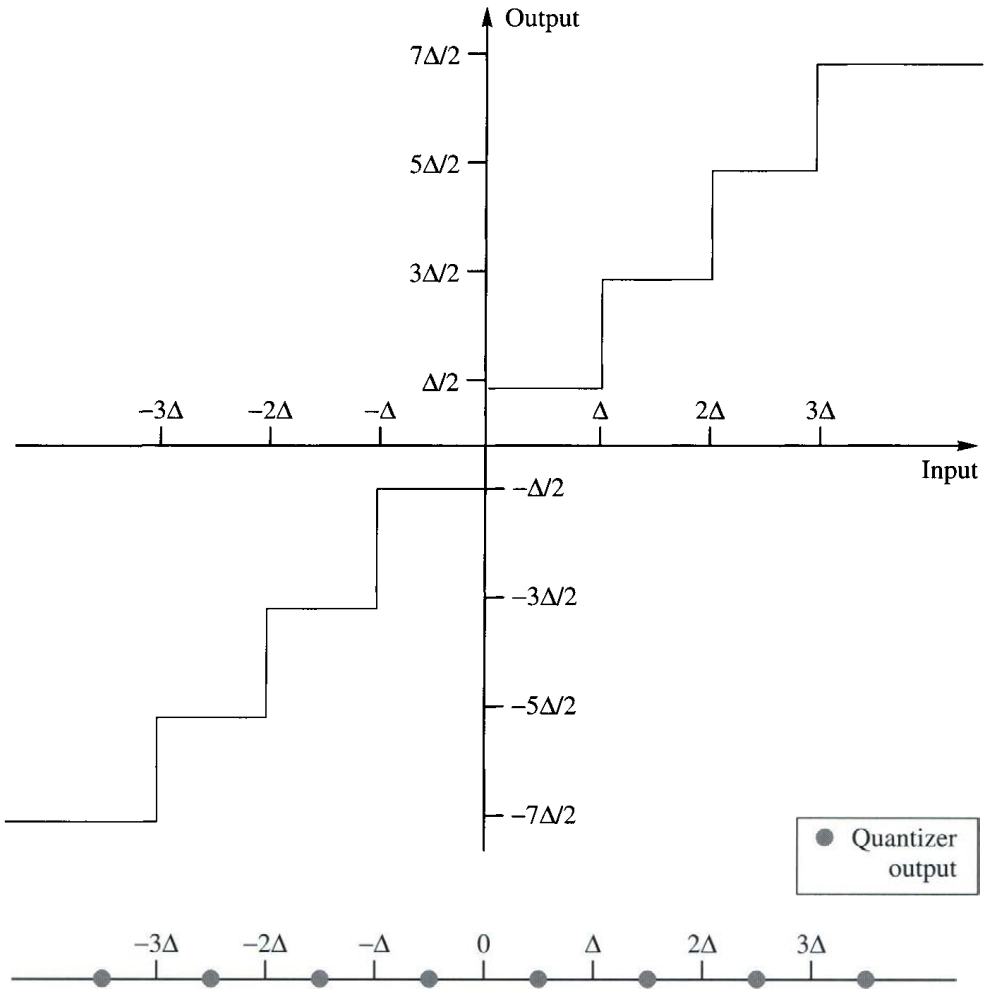

FIGURE 10.4 **Two representations of an eight-level scalar quantizer.**

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.

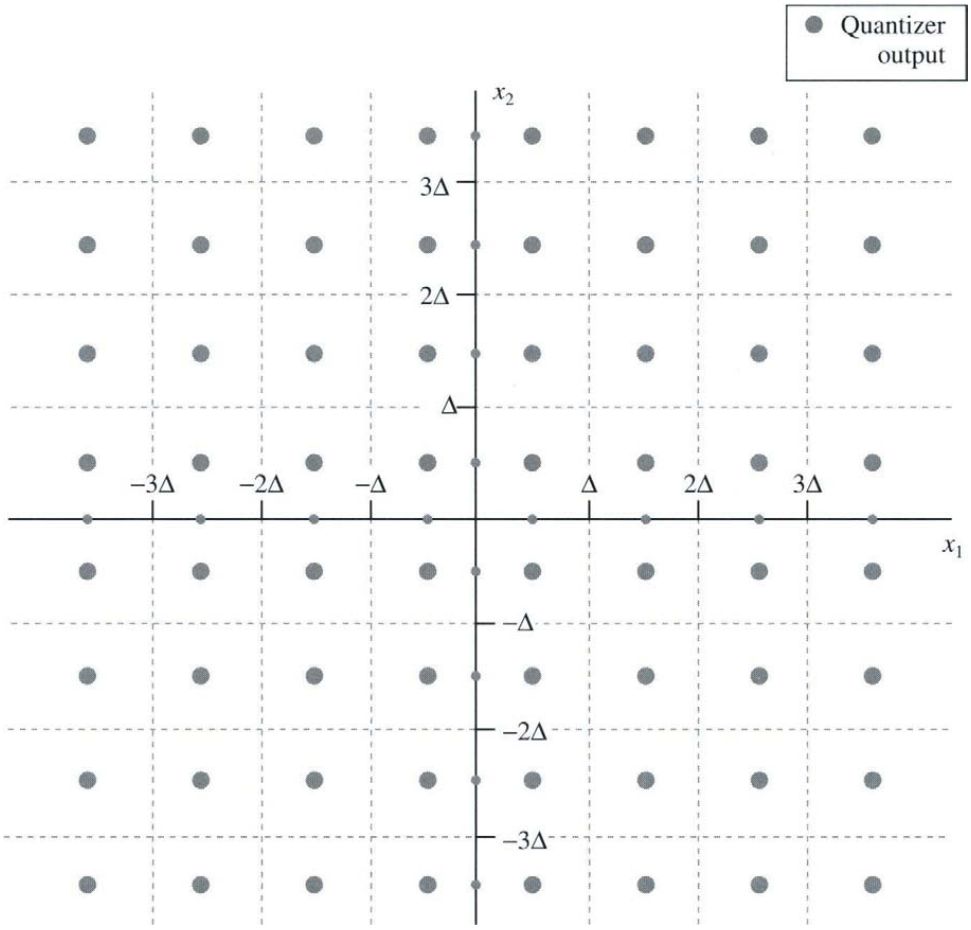

FIGURE 10.5 **Input-output map for consecutive quantization of two inputs using an eight-level scalar quantizer.**

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

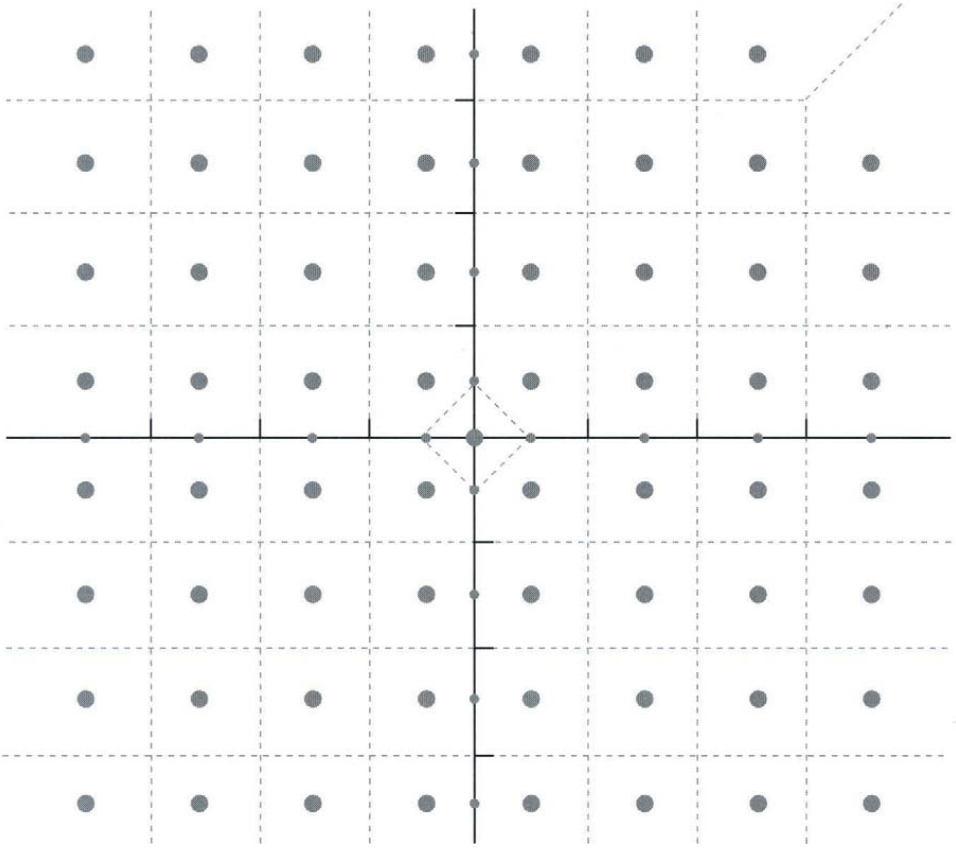

FIGURE 10.6 Modified two-dimensional vector quantizer.

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

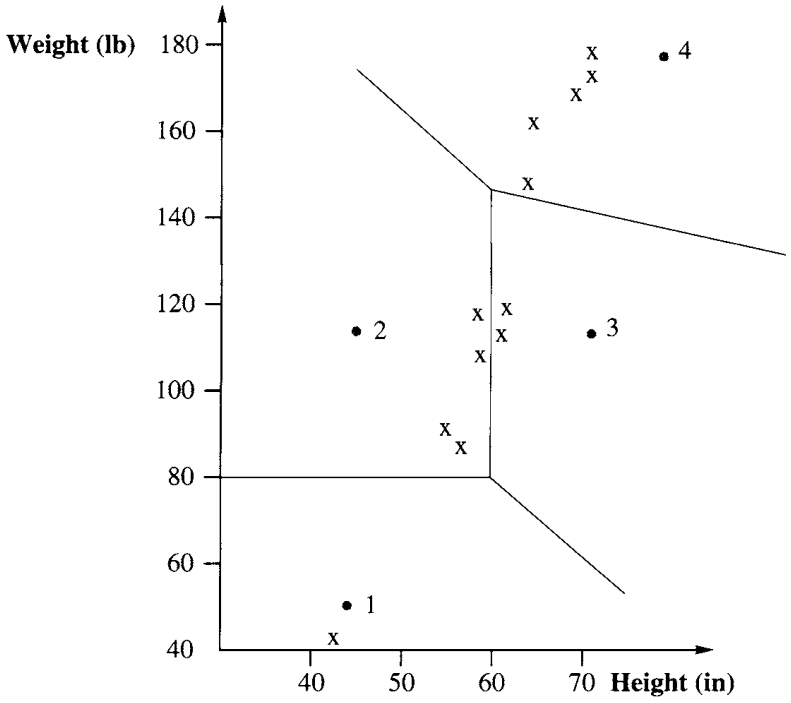

FIGURE 10. 7 **Initial state of the vector quantizer.**

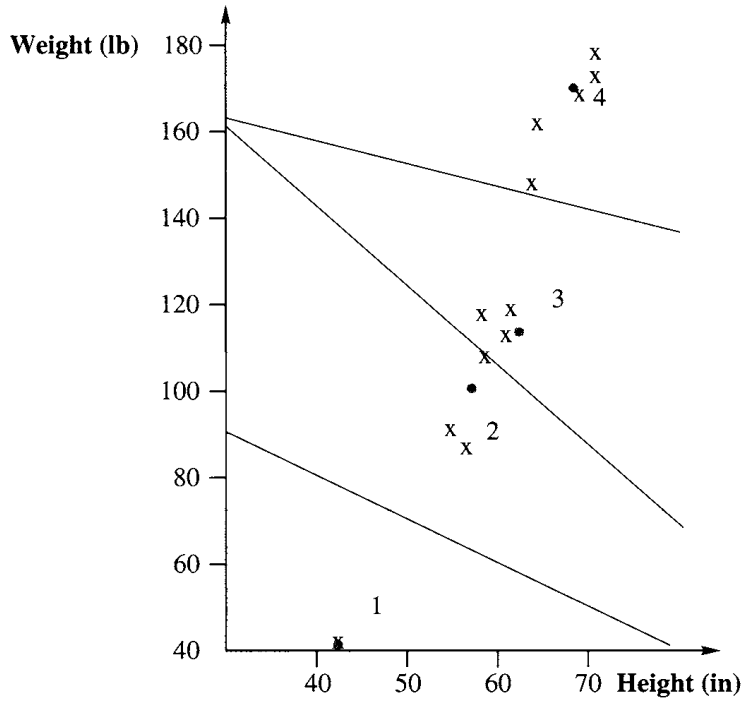

FIGURE 10. 8 **The vector quantizer after one iteration.**

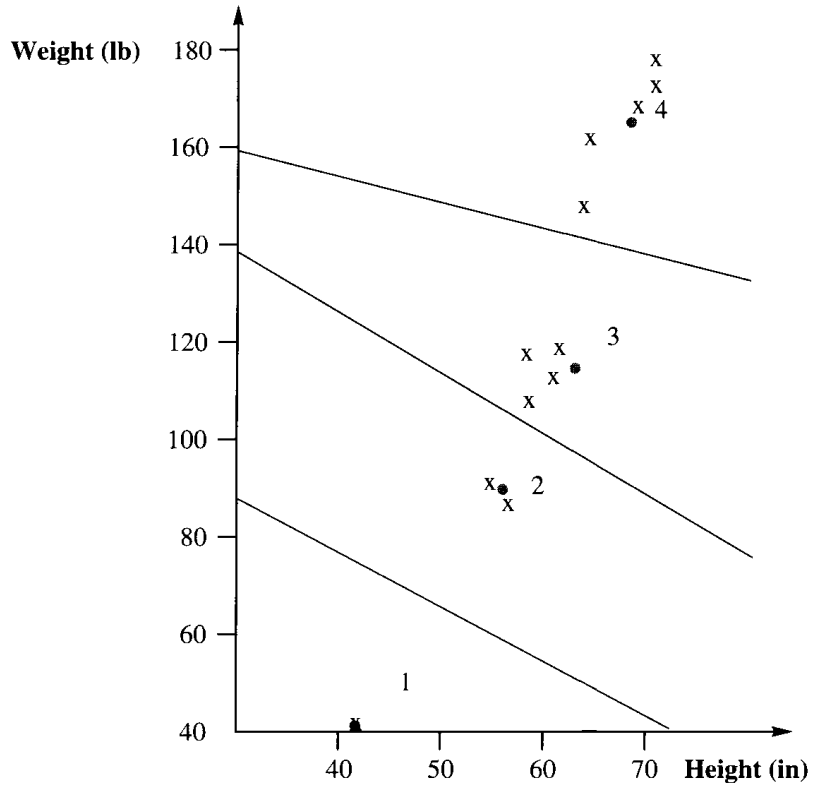

FIGURE 10.9 Final state of the vector quantizer.

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.

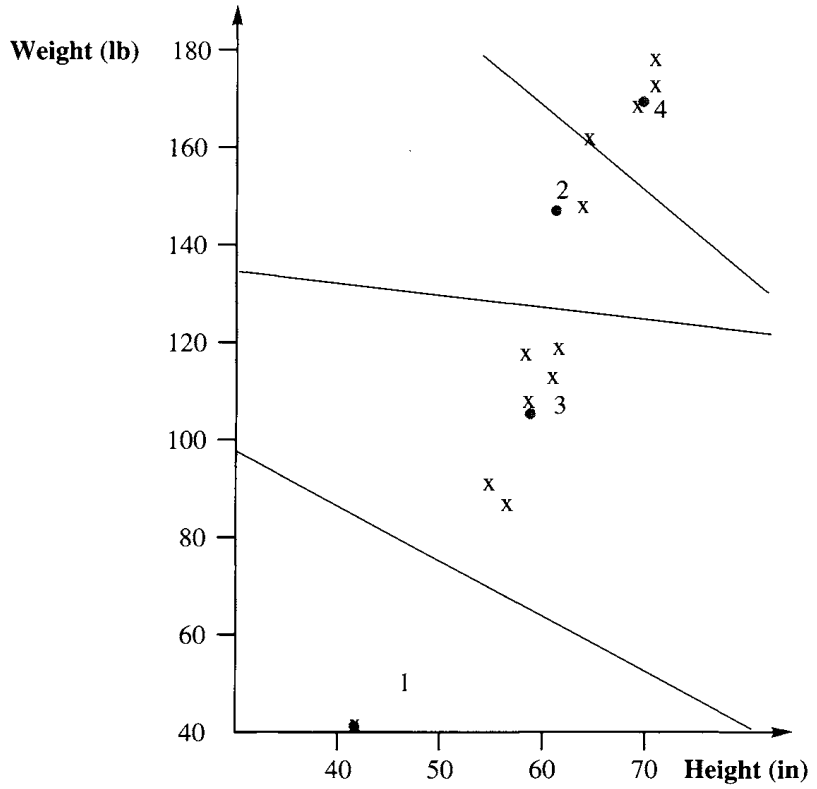

FIGURE 10.10 Final state of the vector quantizer.

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

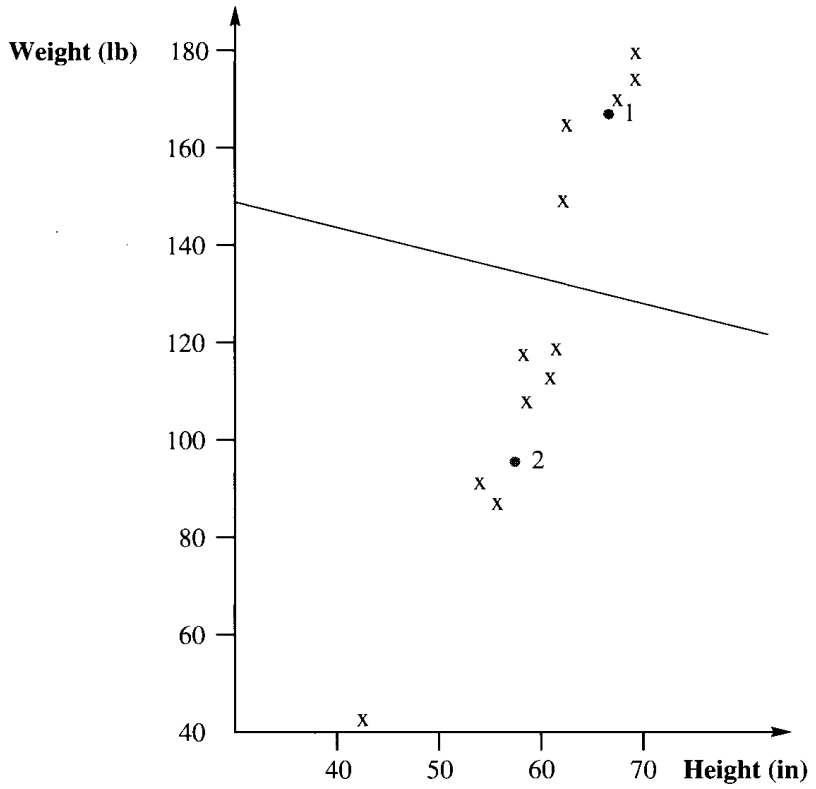

FIGURE 10.11 Two-level vector quantizer using splitting approach.

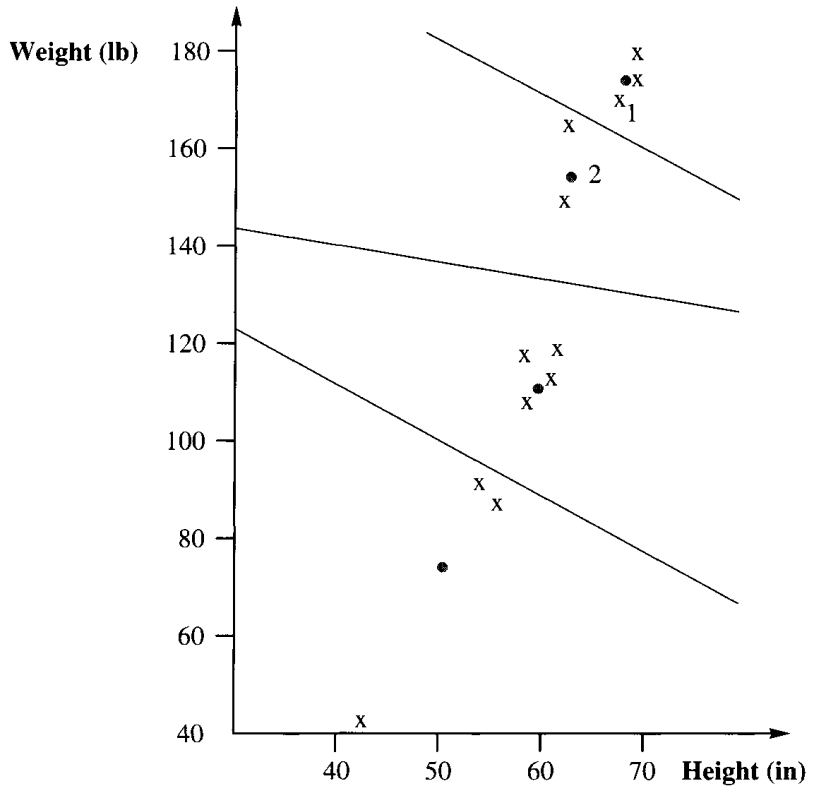

FIGURE 10.12 Final design using the splitting approach.

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.

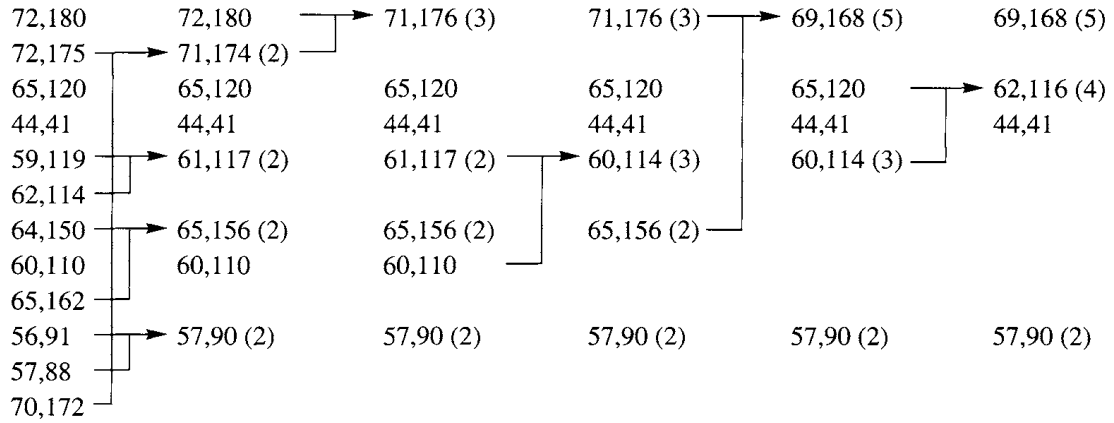

FIGURE 10. 13 Obtaining initial output points using the PNN approach.

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

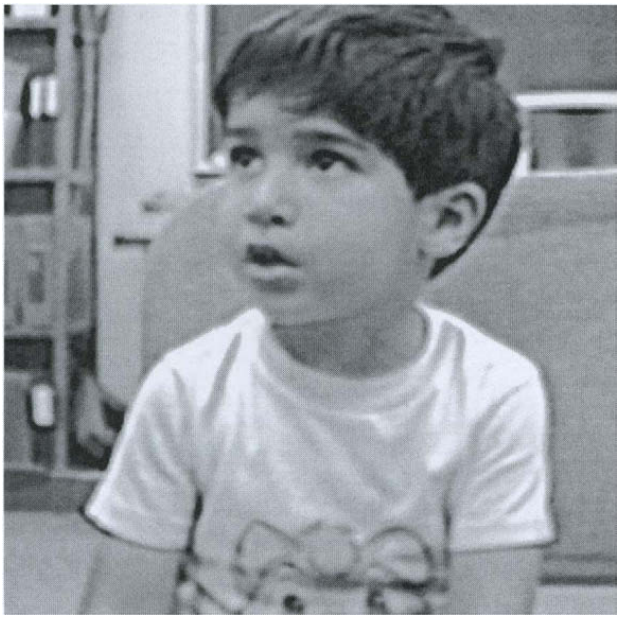

FIGURE 10. 14 Original Sinan image.

$\lceil \log_2 K \rceil$ 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

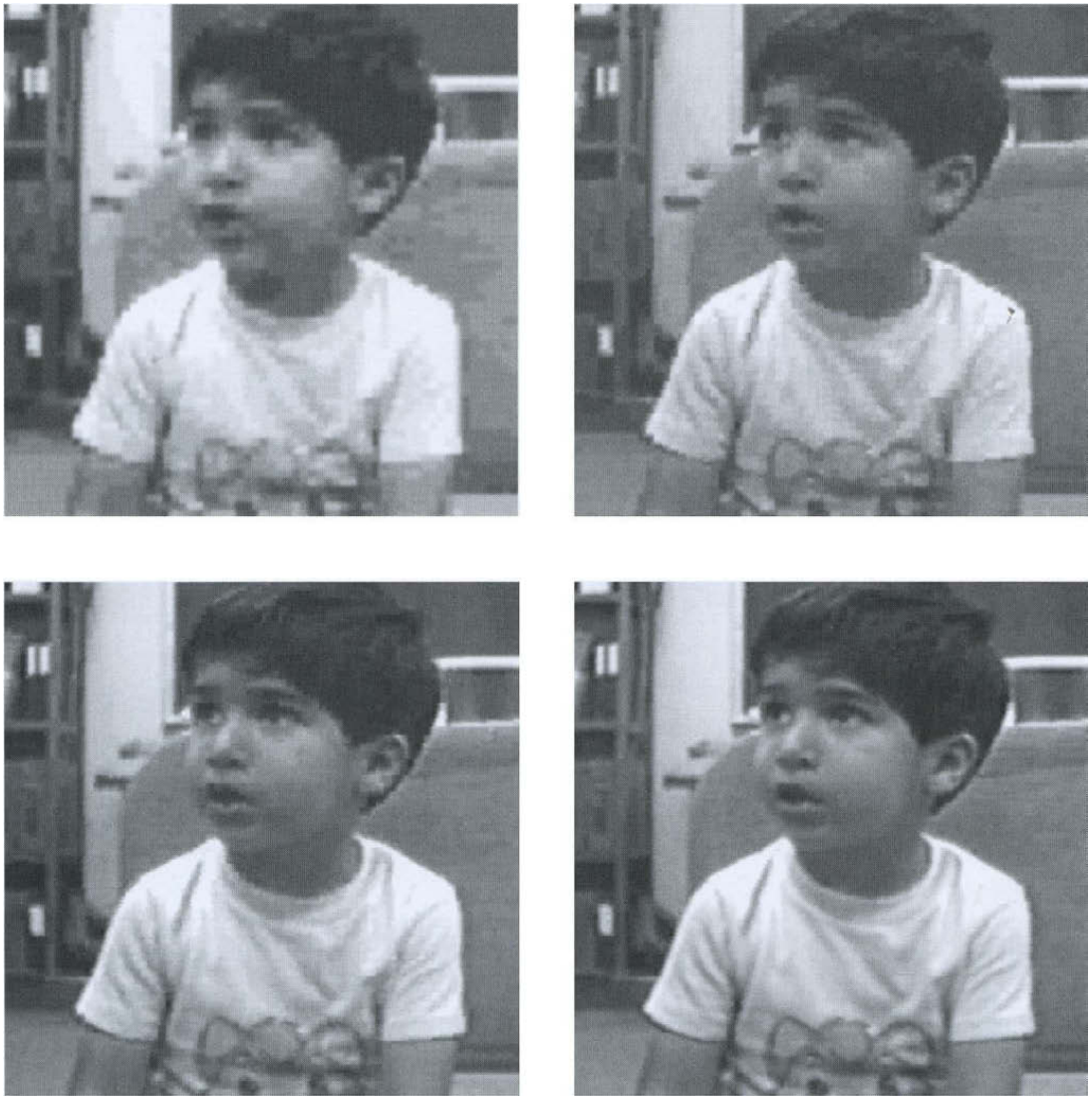

FIGURE 10.15 **Top left: codebook size 16; top right: codebook size 64; bottom left: codebook size 256; bottom right: codebook size 1024.**

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

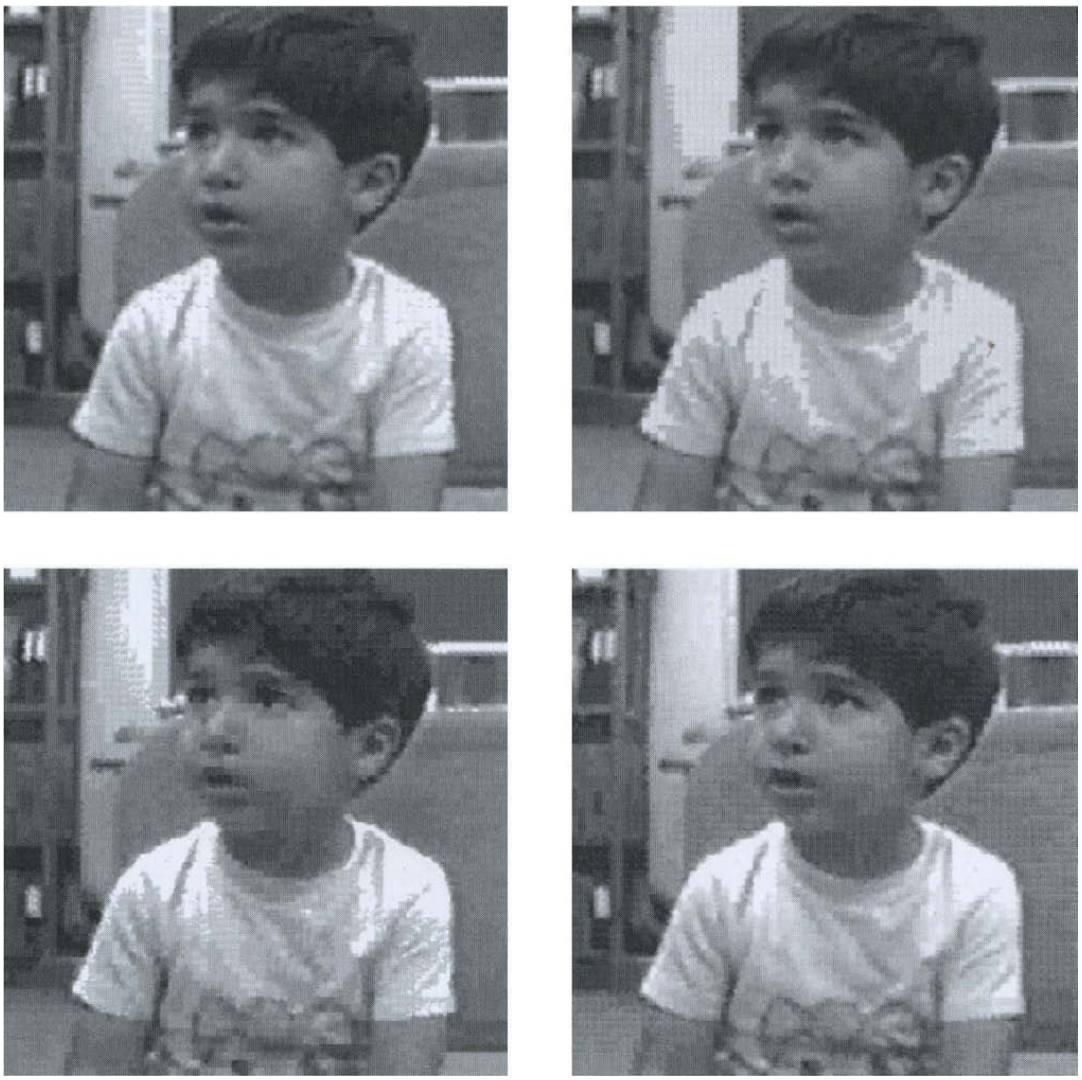

FIGURE 10.16 **Sinan image quantized at the rate of 0.5 bits per pixel. The images used to obtain the codebook were (clockwise from top left) Sensin, Sena, Earth, Omaha.**

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.

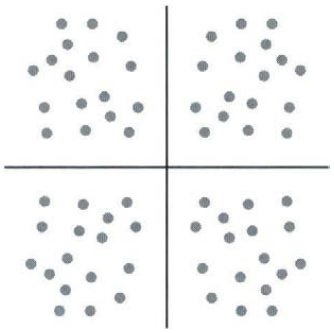

FIGURE 10.17 **A symmetrical vector quantizer in two dimensions.**