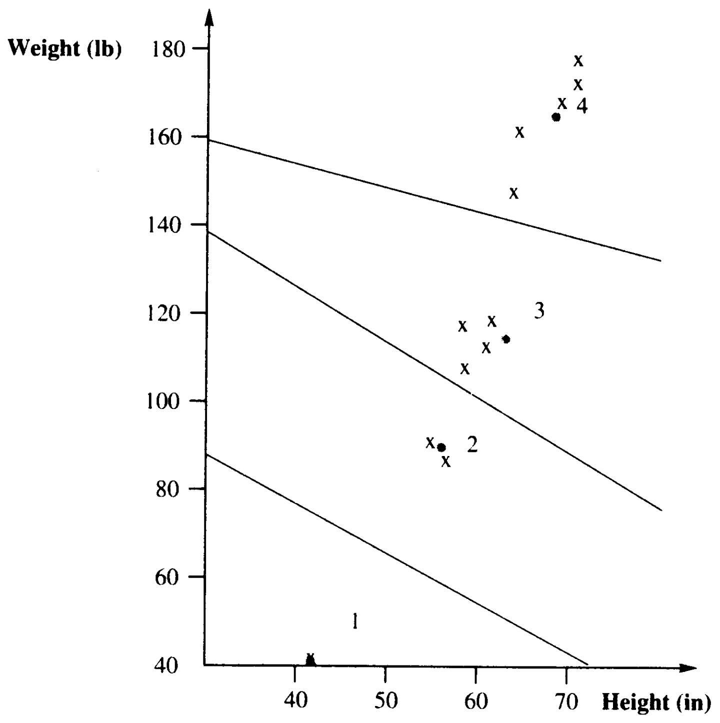

FIGURE 10. 9 **Final state of the vector quantizer.**

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.

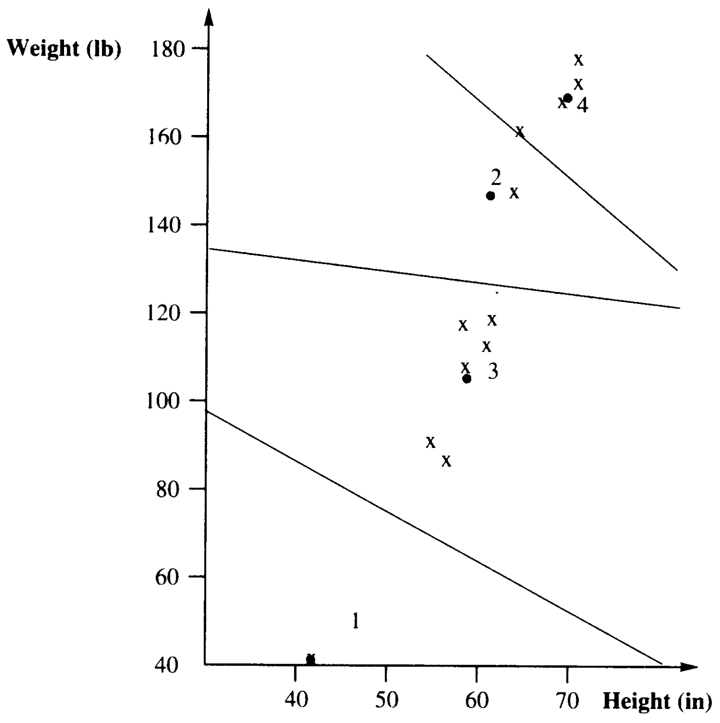

FIGURE 10. 10 **Final state of the vector quantizer.**

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

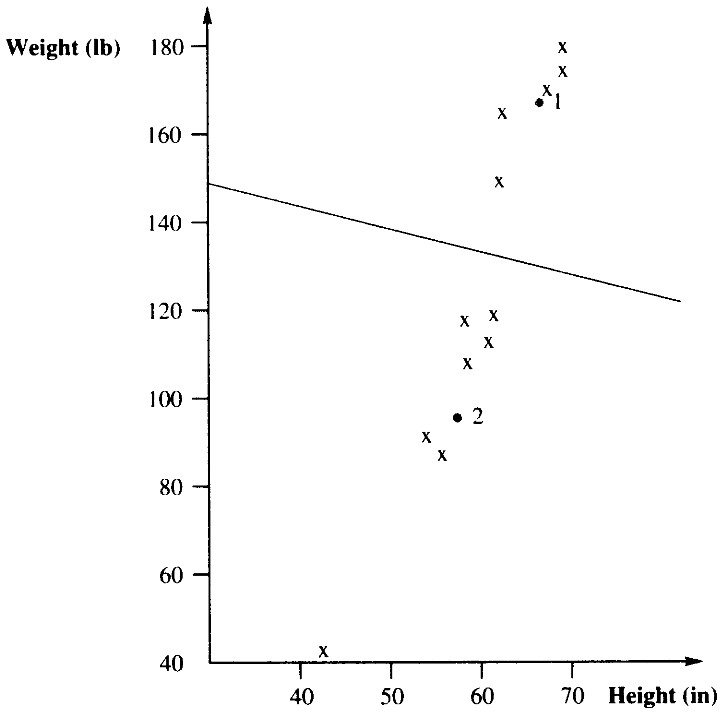

FIGURE 10. 11 Two-level vector quantizer using splitting approach.

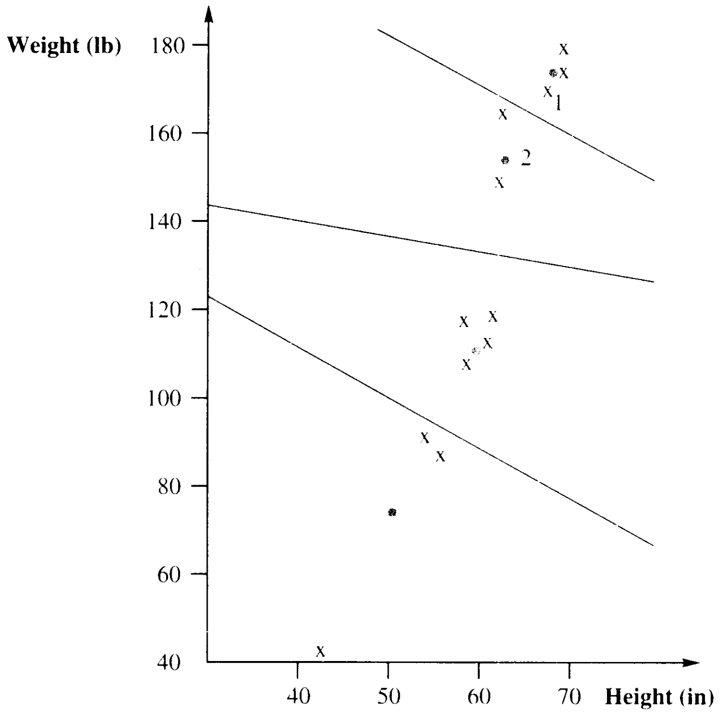

FIGURE 10. 12 Final design using the splitting approach.

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.

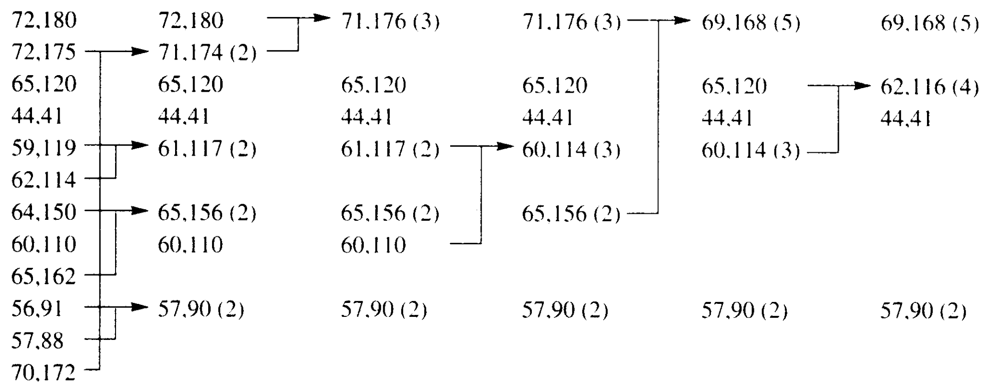

FIGURE 10. 13 **Obtaining initial output points using the PNN approach.**

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

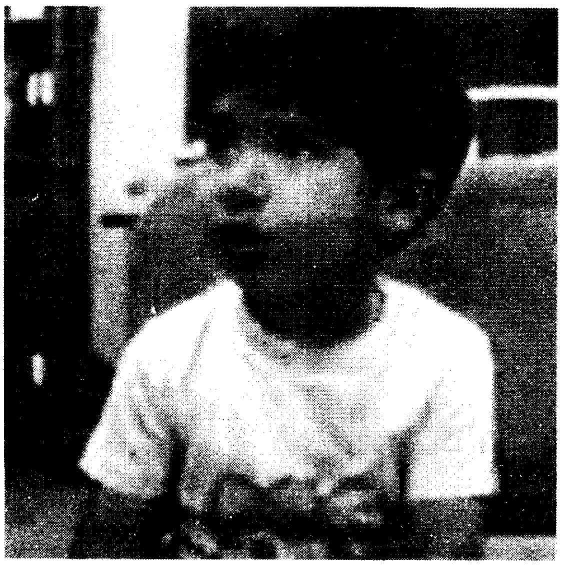

FIGURE 10.14 Original Sinan image.

$\lceil \log_2 K \rceil$ 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

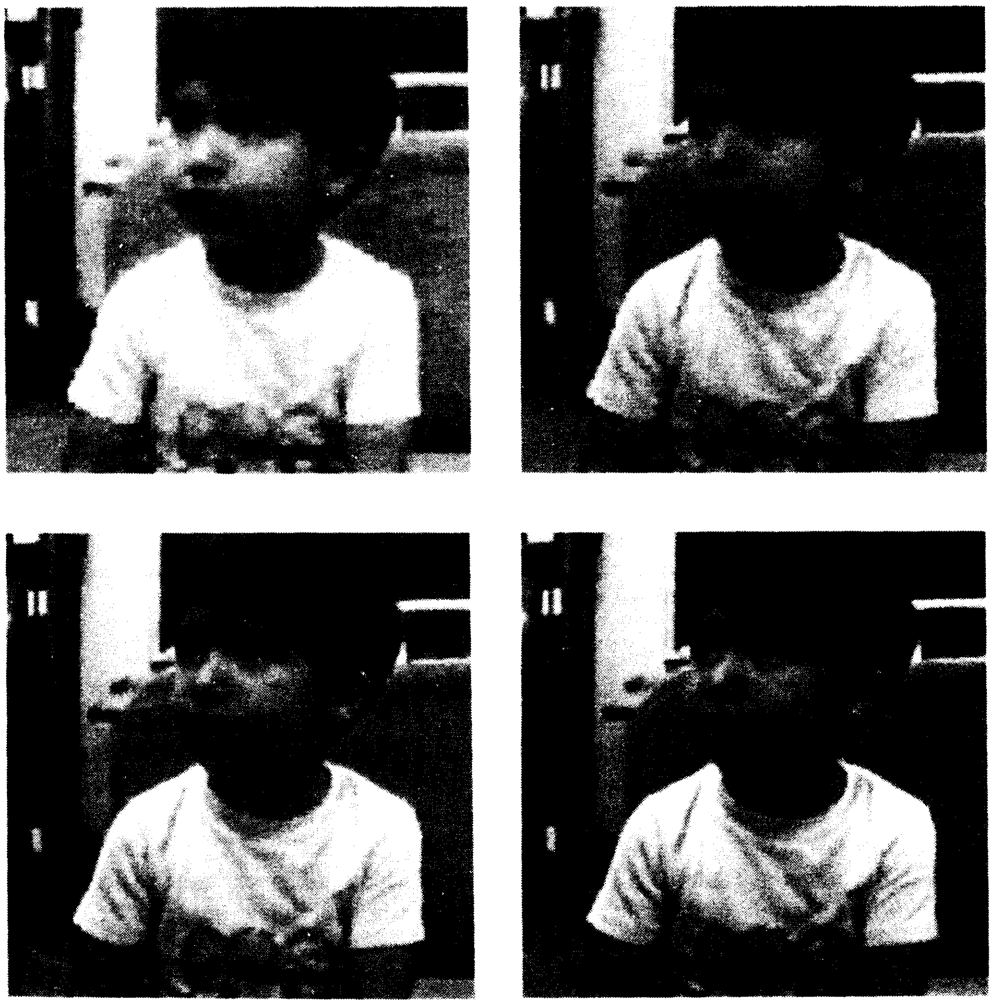

FIGURE 10.15 Top left: codebook size 16; top right: codebook size 64; bottom left: codebook size 256; bottom right: codebook size 1024.

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

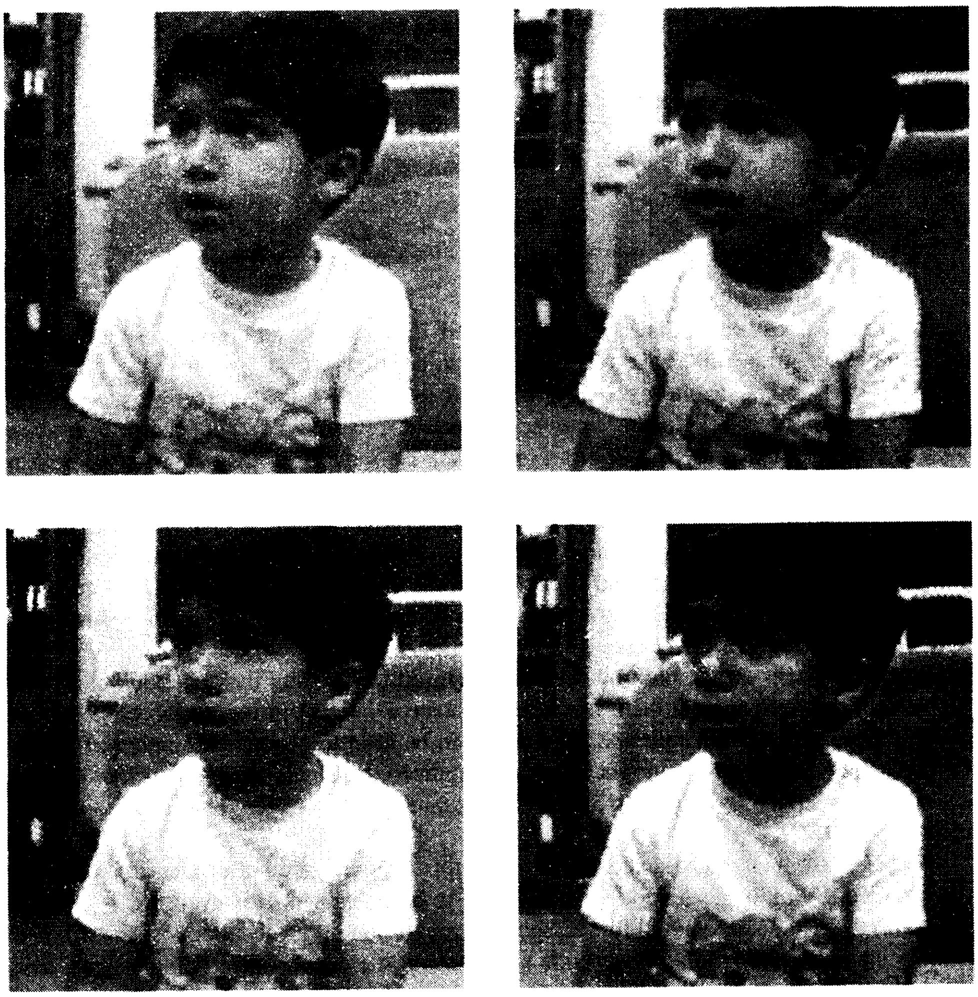

FIGURE 10.16 **Sinan image quantized at the rate of 0.5 bits per pixel. The images used to obtain the codebook were (clockwise from top left) Sensin, Sena, Earth, Omaha.**

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.

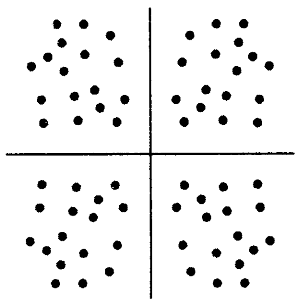

FIGURE 10. 17 A symmetrical vector quantizer in two dimensions.

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.

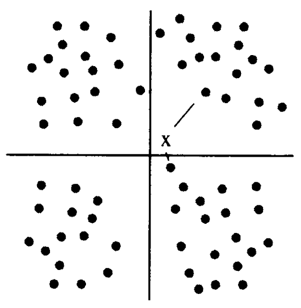

FIGURE 10. 18 Breakdown of the method using the quadrant approach.

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.

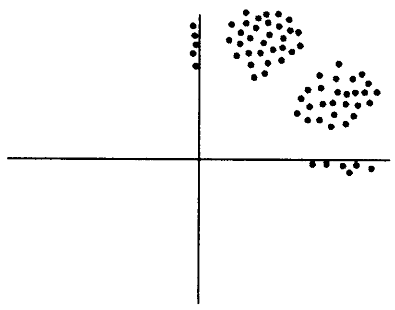

FIGURE 10. 19 Breakdown of the method using the quadrant approach.

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. ♦

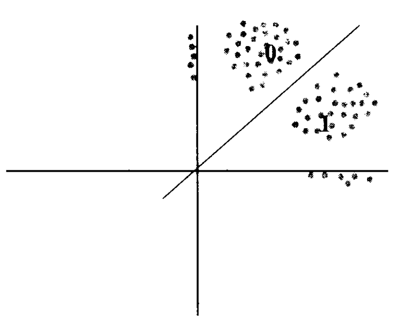

FIGURE 10. 20 Division of output points into two groups.

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.

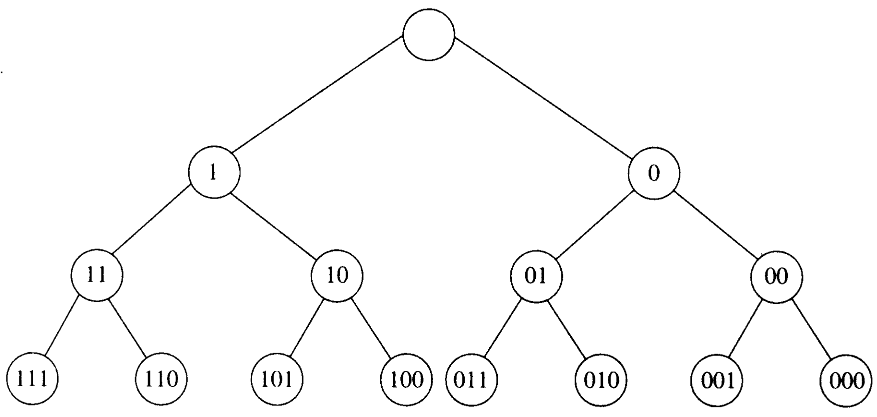

FIGURE 10.21 Decision tree for quantization.

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.

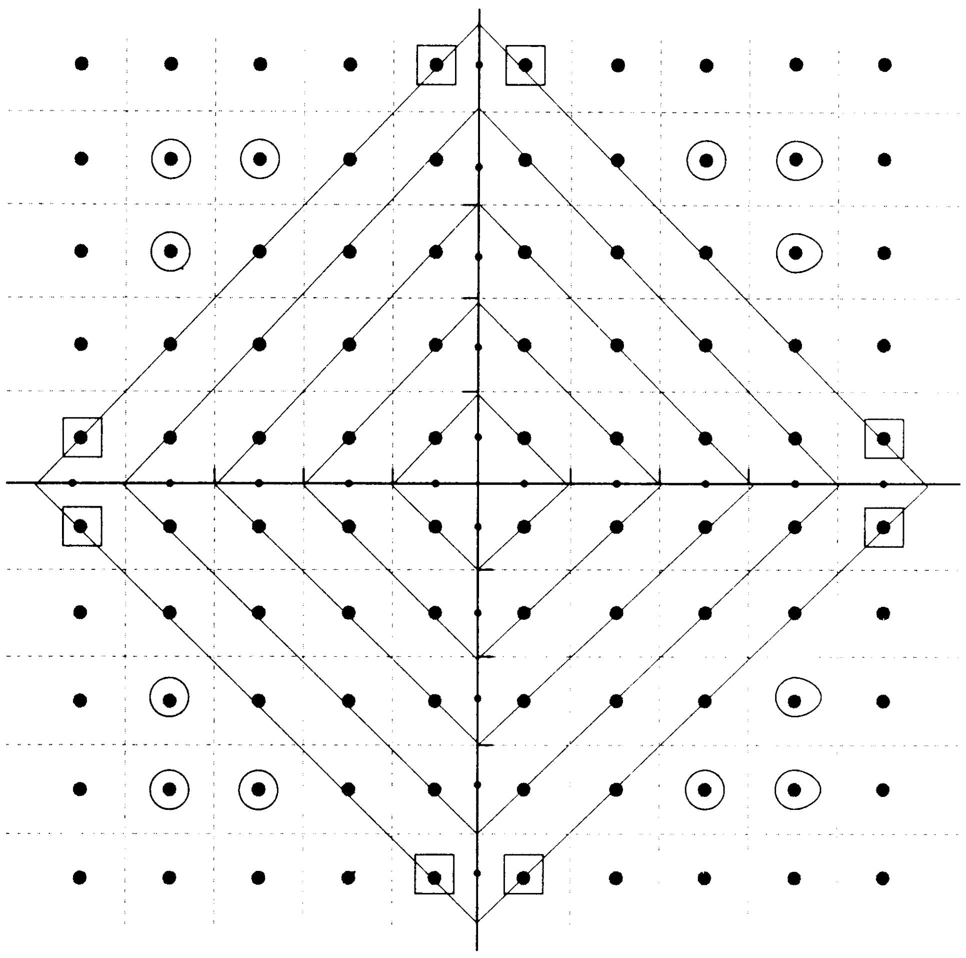

FIGURE 10. 22 Contours of constant probability.

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

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

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

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

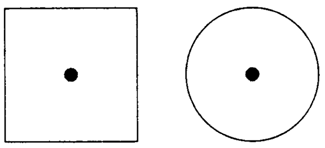

FIGURE 10. 23 Possible quantization regions.

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.

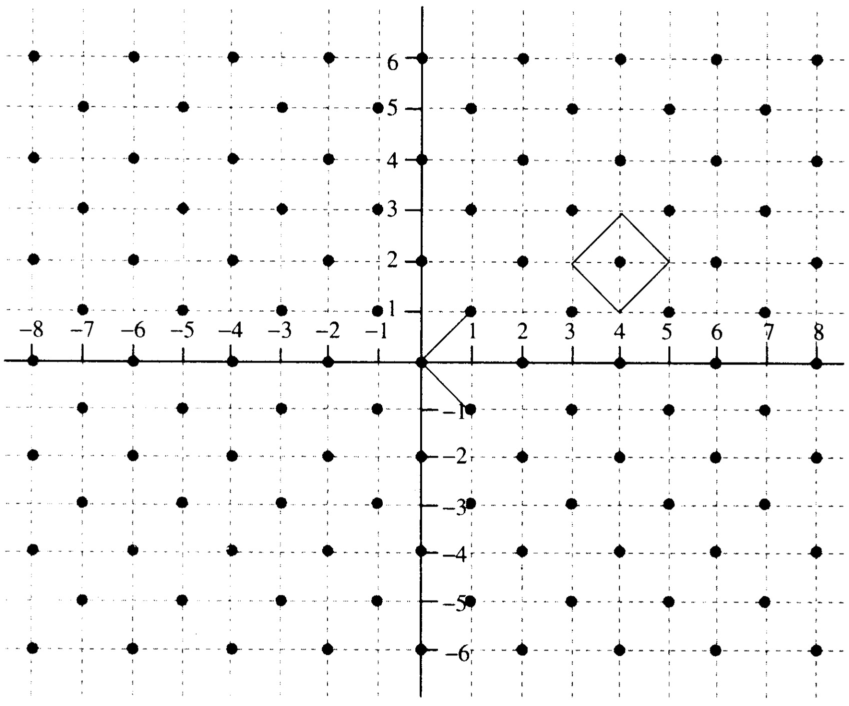

FIGURE 10.24 **The D_2 lattice.**

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

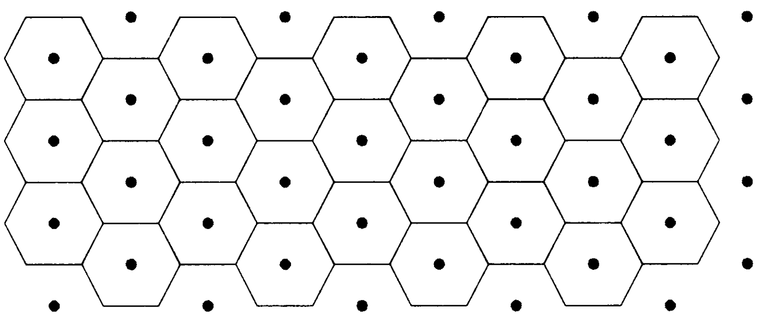

FIGURE 10. 25 **The A_2 lattice.**

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.

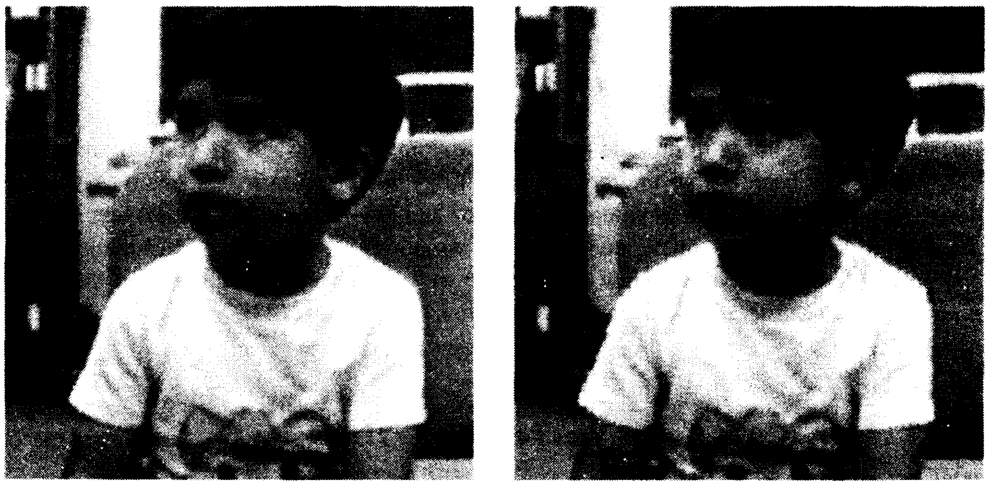

FIGURE 10. 26 **Left: Reconstructed image using mean-removed vector quantization and the Sena image as the training set. Right: LBG vector quantization with the Sena image as the training set.**

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.

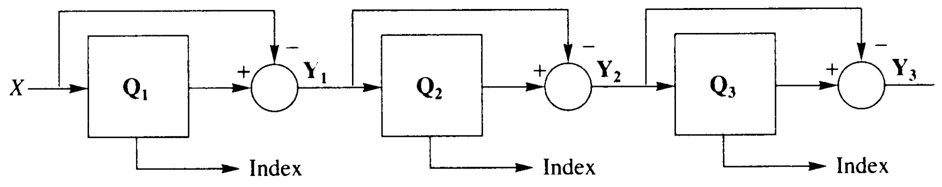

FIGURE 10. 27 A three-stage vector quantizer.

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

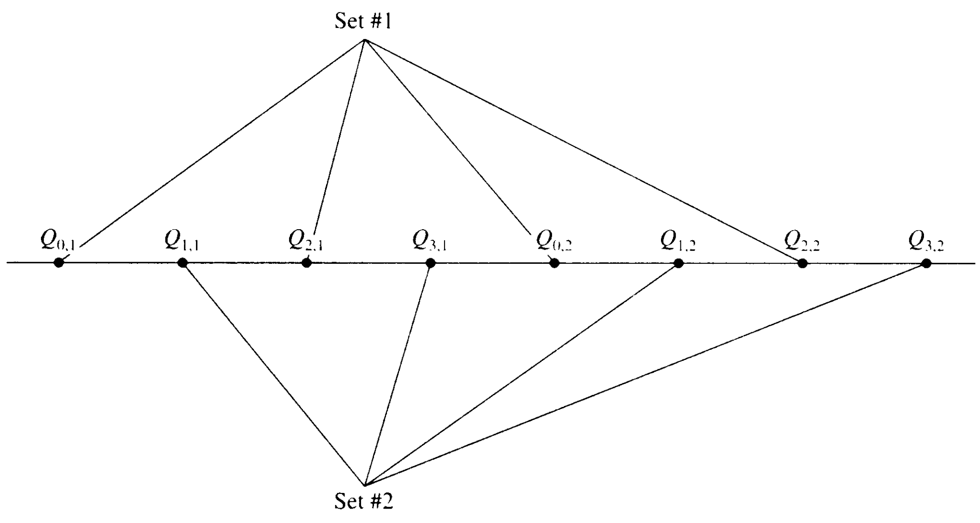

FIGURE 10. 28 Reconstruction levels for a 2-bit trellis-coded quantizer.

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.

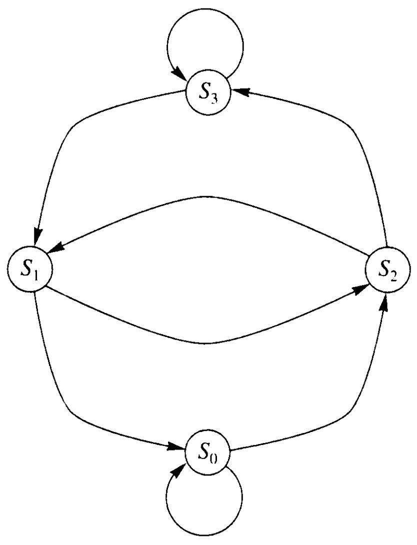

FIGURE 10.29 **State diagram for the selection process.**

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

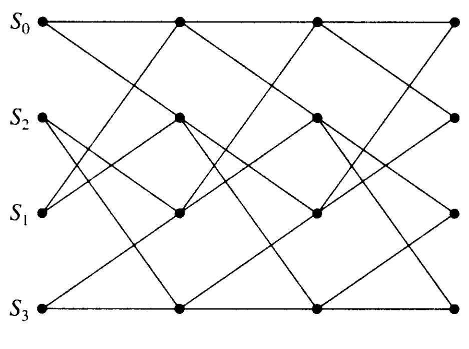

FIGURE 10.30 **Trellis diagram for the selection process.**

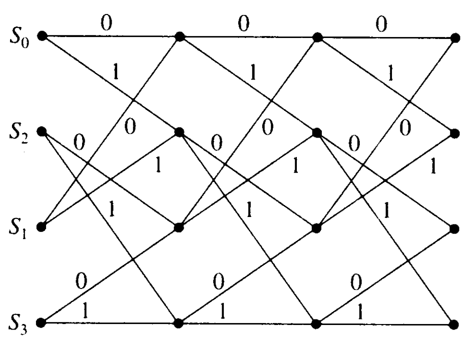

FIGURE 10.31 Trellis diagram for the selection process with binary labels for the state transitions.

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

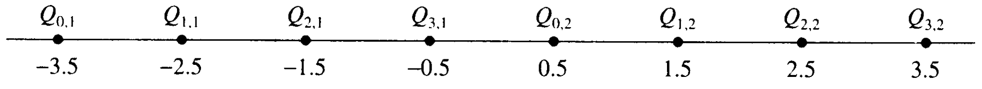

FIGURE 10.32 Reconstruction levels for a 2-bit trellis-coded quantizer.

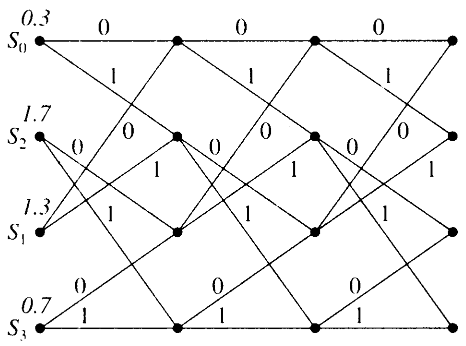

FIGURE 10. 33 Quantizing the first sample.

$Q_{1,1}$ 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

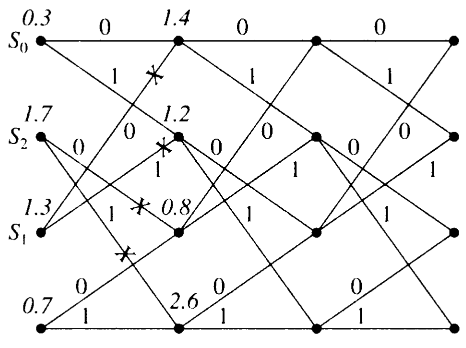

FIGURE 10. 34 Quantizing the second sample.

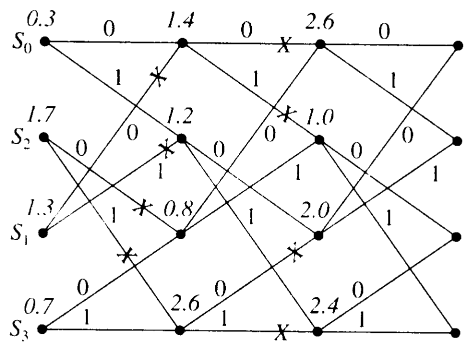

FIGURE 10. 35 Quantizing the third sample.

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.

Differential Encoding

11.1 Overview

Sources such as speech and images have a great deal of correlation from sample to sample. We can use this fact to predict each sample based on its past and only encode and transmit the differences between the prediction and the sample value. Differential encoding schemes are built around this premise. Because the prediction techniques are rather simple, these schemes are much easier to implement than other compression schemes. In this chapter, we will look at various components of differential encoding schemes and study how they are used to encode sources—in particular, speech. We will also look at a widely used international differential encoding standard for speech encoding.

11.2 Introduction

In the last chapter we looked at vector quantization—a rather complex scheme requiring a significant amount of computational resources—as one way of taking advantage of the structure in the data to perform lossy compression. In this chapter, we look at a different approach that uses the structure in the source output in a slightly different manner, resulting in a significantly less complex system.

When we design a quantizer for a given source, the size of the quantization interval depends on the variance of the input. If we assume the input is uniformly distributed, the variance depends on the dynamic range of the input. In turn, the size of the quantization interval determines the amount of quantization noise incurred during the quantization process.

In many sources of interest, the sampled source output $\{x_n\}$ does not change a great deal from one sample to the next. This means that both the dynamic range and the variance of the sequence of differences $\{d_n = x_n - x_{n-1}\}$ are significantly smaller than that of the source output sequence. Furthermore, for correlated sources the distribution of d_n is highly peaked at zero. We made use of this skew, and resulting loss in entropy, for the lossless compression of images in Chapter 7. Given the relationship between the variance of the quantizer input and the incurred quantization error, it is also useful, in terms of lossy compression, to look at ways to encode the difference from one sample to the next rather than encoding the actual sample value. Techniques that transmit information by encoding differences are called *differential encoding techniques*.

Example 11.2.1:

Consider the half cycle of a sinusoid shown in Figure 11.1 that has been sampled at the rate of 30 samples per cycle. The value of the sinusoid ranges between 1 and -1. If we wanted to quantize the sinusoid using a uniform four-level quantizer, we would use a step size of 0.5, which would result in quantization errors in the range $[-0.25, 0.25]$. If we take the sample-to-sample differences (excluding the first sample), the differences lie in the range $[-0.2, 0.2]$. To quantize this range of values with a four-level quantizer requires a step size of 0.1, which results in quantization noise in the range $[-0.05, 0.05]$.

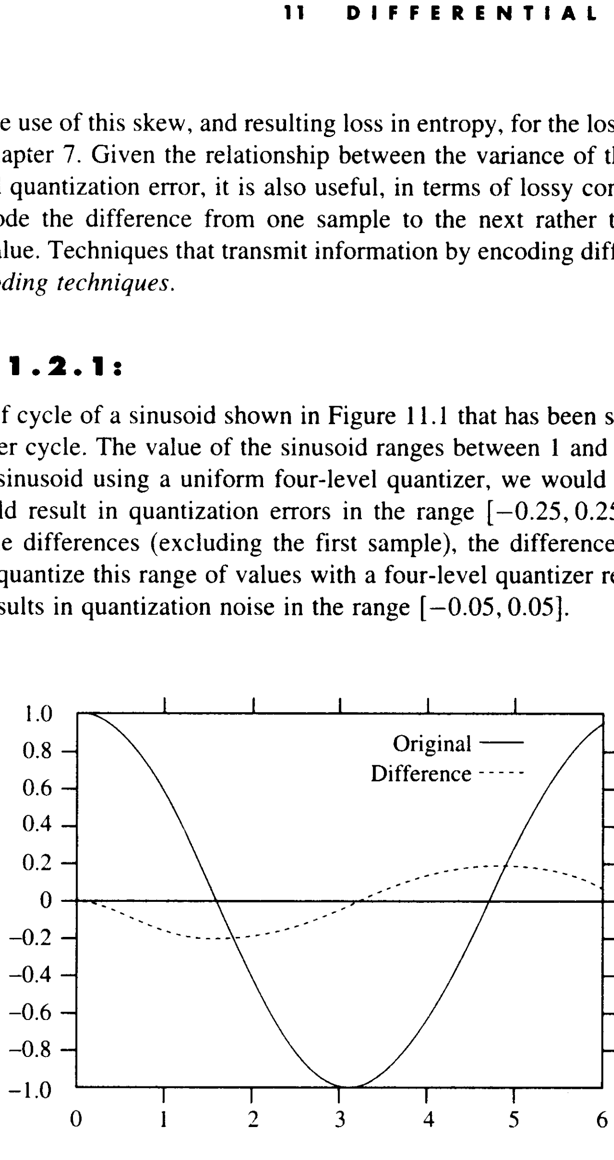

FIGURE 11.1 Sinusoid and sample-to-sample differences. ♦

The sinusoidal signal in the previous example is somewhat contrived. However, if we look at some of the real-world sources that we want to encode, we see that the dynamic range that contains most of the differences is significantly smaller than the dynamic range of the source output.

Example 11.2.2:

Figure 11.2 is the histogram of the Sinan image. Notice that the pixel values vary over almost the entire range of 0 to 255. To represent these values exactly, we need 8 bits per

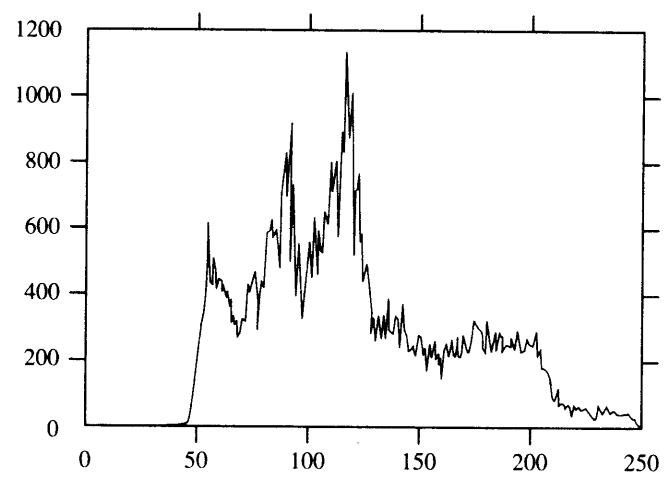

FIGURE 11.2 **Histogram of the Sinan image.**

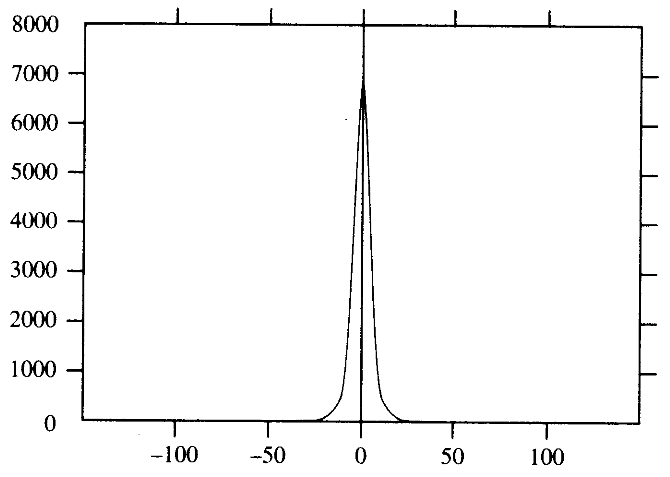

FIGURE 11.3 **Histogram of pixel-to-pixel differences of the Sinan image.**

pixel. To represent these values in a lossy manner to within an error in the least significant bit, we need 7 bits per pixel. Figure 11.3 is the histogram of the differences.

More than 99% of the pixel values lie in the range -31 to 31. Therefore, if we were willing to accept distortion in the least significant bit, for more than 99% of the difference values we need 5 bits per pixel rather than 7. In fact, if we were willing to have a small percentage of the differences with a larger error, we could get by with 4 bits for each difference value. ♦

In both examples, we have shown that the dynamic range of the differences between samples is substantially less than the dynamic range of the source output. In the following sections we describe encoding schemes that take advantage of this fact to provide improved compression performance.

11.3 The Basic Algorithm

Although it takes fewer bits to encode differences than it takes to encode the original pixel, we have not said whether it is possible to recover an acceptable reproduction of the original sequence from the quantized difference value. When we were looking at lossless compression schemes, we found that if we encoded and transmitted the first value of a sequence, followed by the encoding of the differences between samples, we could losslessly recover the original sequence. Unfortunately, a strictly analogous situation does not exist for lossy compression.

Example 11.3.1:

Suppose a source puts out the sequence

$$6.2\ 9.7\ 13.2\ 5.9\ 8\ 7.4\ 4.2\ 1.8$$

We could generate the following sequence by taking the difference between samples (assume that the first sample value is zero):

$$6.2\ 3.5\ 3.5\ -7.3\ 2.1\ -0.6\ -3.2\ -2.4$$

If we losslessly encoded these values, we could recover the original sequence at the receiver by adding back the difference values. For example, to obtain the second reconstructed value, we add the difference 3.5 to the first received value 6.2 to obtain a value of 9.7. The third reconstructed value can be obtained by adding the received difference value of 3.5 to the second reconstructed value of 9.7, resulting in a value of 13.2, which is the same as the third value in the original sequence. Thus, by adding the nth received difference value to the $(n-1)$th reconstruction value, we can recover the original sequence exactly.

Now let us look at what happens if these difference values are encoded using a lossy scheme. Suppose we had a seven-level quantizer with output values $-6, -4, -2, 0, 2, 4, 6$. The quantized sequence would be

$$6\ 4\ 4\ -6\ 2\ 0\ -4\ -2$$

If we follow the same procedure for reconstruction as we did for the lossless compression scheme, we get the sequence

$$6\ 10\ 14\ 8\ 10\ 10\ 6\ 4$$

The difference or error between the original sequence and the reconstructed sequence is

$$0.2\ -0.3\ -0.8\ -2.1\ -2\ -2.6\ -1.8\ -2.2$$

Notice that initially the magnitudes of the error are quite small (0.2, 0.3). As the reconstruction progresses, the magnitudes of the error become significantly larger (2.6, 1.8, 2.2). ♦

To see what is happening, consider a sequence $\{x_n\}$. A difference sequence $\{d_n\}$ is generated by taking the differences $x_n - x_{n-1}$. This difference sequence is quantized to obtain the sequence $\{\hat{d}_n\}$:

$$\hat{d}_n = Q[d_n] = d_n + q_n$$

where q_n is the quantization error. At the receiver, the reconstructed sequence $\{\hat{x}_n\}$ is obtained by adding $\hat{d}_n$ to the previous reconstructed value $\hat{x}_{n-1}$:

$$\hat{x}_n = \hat{x}_{n-1} + \hat{d}_n.$$

Let us assume that both transmitter and receiver start with the same value x_0, that is, $\hat{x}_0 = x_0$. Follow the quantization and reconstruction process for the first few samples:

$$d_1 = x_1 - x_0 \tag{11.1}$$
$$\hat{d}_1 = Q[d_1] = d_1 + q_1 \tag{11.2}$$
$$\hat{x}_1 = x_0 + \hat{d}_1 = x_0 + d_1 + q_1 = x_1 + q_1 \tag{11.3}$$
$$d_2 = x_2 - x_1 \tag{11.4}$$
$$\hat{d}_2 = Q[d_2] = d_2 + q_2 \tag{11.5}$$
$$\hat{x}_2 = \hat{x}_1 + \hat{d}_2 = x_1 + q_1 + d_2 + q_2 \tag{11.6}$$
$$= x_2 + q_1 + q_2. \tag{11.7}$$

Continuing this process, at the nth iteration we get

$$\hat{x}_n = x_n + \sum_{k=1}^{n} q_k. \tag{11.8}$$

We can see that the quantization error accumulates as the process continues. Theoretically, if the quantization error process is zero mean, the errors will cancel each other out in the long run. In practice, often long before that can happen, the finite precision of the machines causes the reconstructed value to overflow.

Notice that the encoder and decoder are operating with different pieces of information. The encoder generates the difference sequence based on the original sample values, while the decoder adds back the quantized difference onto a distorted version of the original signal. We can solve this problem by forcing both encoder and decoder to use the same information during the differencing and reconstruction operations. The only information available to the receiver about the sequence $\{x_n\}$ is the reconstructed sequence $\{\hat{x}_n\}$. As this information is also available to the transmitter, we can modify the differencing operation to use the reconstructed value of the previous sample, instead of the previous sample itself, that is,

$$d_n = x_n - \hat{x}_{n-1}. \tag{11.9}$$

Using this new differencing operation, let's repeat our examination of the quantization and reconstruction process. We again assume that $\hat{x}_0 = x_0$.

$$\begin{aligned} d_1 &= x_1 - x_0 && (11.10)\\ \hat{d}_1 &= Q[d_1] = d_1 + q_1 && (11.11)\\ \hat{x}_1 &= x_0 + \hat{d}_1 = x_0 + d_1 + q_1 = x_1 + q_1 && (11.12)\\ d_2 &= x_2 - \hat{x}_1 && (11.13)\\ \hat{d}_2 &= Q[d_2] = d_2 + q_2 && (11.14)\\ \hat{x}_2 &= \hat{x}_1 + \hat{d}_2 = \hat{x}_1 + d_2 + q_2 && (11.15)\\ &= x_2 + q_2 && (11.16) \end{aligned}$$

At the nth iteration we have

$$\hat{x}_n = x_n + q_n, \qquad (11.17)$$

and there is no accumulation of the quantization noise. In fact, the quantization noise in the nth reconstructed sequence is the quantization noise incurred by the quantization of the nth difference. The quantization error for the difference sequence is substantially less than the quantization error for the original sequence. Therefore, this procedure leads to an overall reduction of the quantization error. If we are satisfied with the quantization error for a given number of bits per sample, then we can use fewer bits with a differential encoding procedure to attain the same distortion.

Example 11.3.2:

Let us try to quantize and then reconstruct the sinusoid of Example 11.2.1 using the two different differencing approaches. Using the first approach, we get a dynamic range of

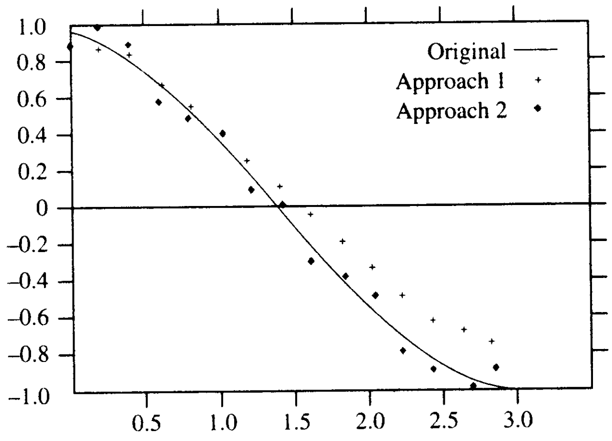

FIGURE 11.4 **Sinusoid and reconstructions.**

differences from -0.2 to 0.2. Therefore, we use a quantizer step size of 0.1. In the second approach, the differences lie in the range $[-0.4, 0.4]$. In order to cover this range, we use a step size in the quantizer of 0.2. The reconstructed signals are shown in Figure 11.4.

Notice in the first case that the reconstruction diverges from the signal as we process more and more of the signal. Although the second differencing approach uses a larger step size, this approach provides a more accurate representation of the input. ♦

A block diagram of the differential encoding system as we have described it to this point is shown in Figure 11.5. We have drawn a dotted box around the portion of the encoder that mimics the decoder. The encoder must mimic the decoder in order to obtain a copy of the reconstructed sample used to generate the next difference.

We would like our difference value to be as small as possible. For this to happen, given the system we have described to this point, $\hat{x}_{n-1}$ should be as close to x_n as possible. However, $\hat{x}_{n-1}$ is the reconstructed value of x_{n-1}; therefore, we would like $\hat{x}_{n-1}$ to be close to x_{n-1}. Unless x_{n-1} is always very close to x_n, some function of past values of the reconstructed sequence can often provide a better prediction of x_n. We will look at some of these *predictor* functions later in this chapter. For now, let's modify Figure 11.5 and replace the delay block with a predictor block to obtain our basic differential encoding system as shown in Figure 11.6. The output of the predictor is the prediction sequence $\{p_n\}$ given by

$$p_n = f(\hat{x}_{n-1}, \hat{x}_{n-2}, \ldots, \hat{x}_0). \tag{11.18}$$

This basic differential encoding system is known as the differential pulse code modulation (DPCM) system. The DPCM system was developed at Bell Laboratories a few years after World War II [164]. It is most popular as a speech-encoding system and is widely used in telephone communications.

As we can see from Figure 11.6, the DPCM system consists of two major components, the predictor and the quantizer. The study of DPCM is basically the study of these two components. In the following sections, we will look at various predictor and quantizer designs and see how they function together in a differential encoding system.

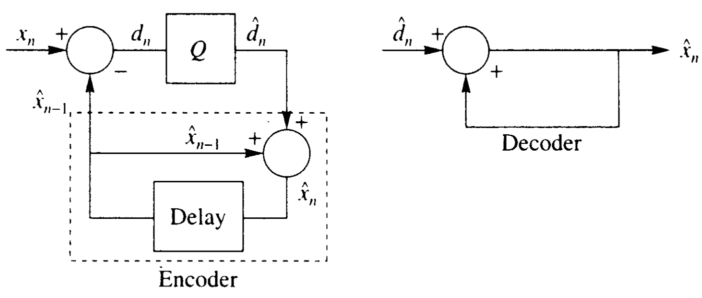

FIGURE 11.5 A simple differential encoding system.

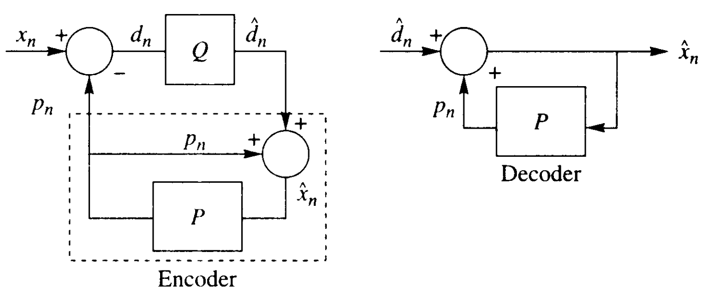

FIGURE 11.6 The basic algorithm.

11.4 Prediction in DPCM

Differential encoding systems like DPCM gain their advantage by the reduction in the variance and dynamic range of the difference sequence. How much the variance is reduced depends on how well the predictor can predict the next symbol based on the past reconstructed symbols. In this section we will mathematically formulate the prediction problem. The analytical solution to this problem will give us one of the more widely used approaches to the design of the predictor. In order to follow this development, some familiarity with the mathematical concepts of expectation and correlation is needed. These concepts are described in Appendix A.

Define σ_d^2, the variance of the difference sequence, as

$$\sigma_d^2 = E[(x_n - p_n)^2] \tag{11.19}$$

where $E[]$ is the expectation operator. As the predictor outputs p_n are given by (11.18), the design of a good predictor is essentially the selection of the function $f(\cdot)$ that minimizes σ_d^2. One problem with this formulation is that $\hat{x}_n$ is given by

$$\hat{x}_n = x_n + q_n$$

and q_n depends on the variance of d_n. Thus, by picking $f(\cdot)$, we affect σ_d^2, which in turn affects the reconstruction $\hat{x}_n$, which then affects the selection of $f(\cdot)$. This coupling makes an explicit solution extremely difficult for even the most well-behaved source [165]. As most real sources are far from well behaved, the problem becomes computationally intractable in most applications.

We can avoid this problem by making an assumption known as the *fine quantization assumption*. We assume that quantizer step sizes are so small that we can replace $\hat{x}_n$ by x_n, and therefore

$$p_n = f(x_{n-1}, x_{n-2}, \ldots, x_0). \tag{11.20}$$

Once the function $f(\cdot)$ has been found, we can use it with the reconstructed values $\hat{x}_n$ to obtain p_n. If we now assume that the output of the source is a stationary process, from the study of random processes [166], we know that the function that minimizes σ_d^2

is the conditional expectation $E[x_n \mid x_{n-1}, x_{n-2}, \ldots, x_0]$. Unfortunately, the assumption of stationarity is generally not true, and even if it were, finding this conditional expectation requires the knowledge of nth-order conditional probabilities, which would generally not be available.

Given the difficulty of finding the best solution, in many applications we simplify the problem by restricting the predictor function to be linear. That is, the prediction p_n is given by

$$p_n = \sum_{i=1}^{N} a_i \hat{x}_{n-i}. \tag{11.21}$$

The value of N specifies the order of the predictor. Using the fine quantization assumption, we can now write the predictor design problem as follows: Find the $\{a_i\}$ so as to minimize σ_d^2.

$$\sigma_d^2 = E\left[\left(x_n - \sum_{i=1}^{N} a_i x_{n-i}\right)^2\right] \tag{11.22}$$

where we assume that the source sequence is a realization of a real valued wide sense stationary process. Take the derivative of σ_d^2 with respect to each of the a_i and set this equal to zero. We get N equations and N unknowns:

$$\frac{\delta\sigma_d^2}{\delta a_1} = -2E\left[\left(x_n - \sum_{i=1}^{N} a_i x_{n-i}\right)x_{n-1}\right] = 0 \tag{11.23}$$

$$\frac{\delta\sigma_d^2}{\delta a_2} = -2E\left[\left(x_n - \sum_{i=1}^{N} a_i x_{n-i}\right)x_{n-2}\right] = 0 \tag{11.24}$$

$$\vdots \quad \vdots$$

$$\frac{\delta\sigma_d^2}{\delta a_N} = -2E\left[\left(x_n - \sum_{i=1}^{N} a_i x_{n-i}\right)x_{n-N}\right] = 0. \tag{11.25}$$

Taking the expectations, we can rewrite these equations as

$$\sum_{i=1}^{N} a_i R_{xx}(i-1) = R_{xx}(1) \tag{11.26}$$

$$\sum_{i=1}^{N} a_i R_{xx}(i-2) = R_{xx}(2) \tag{11.27}$$

$$\vdots \quad \vdots$$

$$\sum_{i=1}^{N} a_i R_{xx}(i-N) = R_{xx}(N) \tag{11.28}$$

where $R_{xx}(k)$ is the autocorrelation function of x_n:

$$R_{xx}(k) = E[x_n x_{n+k}]. \tag{11.29}$$

We can write these equations in matrix form as

$$\mathbf{RA} = \mathbf{P} \tag{11.30}$$

where

$$\mathbf{R} = \begin{bmatrix} R_{xx}(0) & R_{xx}(1) & R_{xx}(2) & \cdots & R_{xx}(N-1) \\ R_{xx}(1) & R_{xx}(0) & R_{xx}(1) & \cdots & R_{xx}(N-2) \\ R_{xx}(2) & R_{xx}(1) & R_{xx}(0) & \cdots & R_{xx}(N-3) \\ \vdots & \vdots & & & \vdots \\ R_{xx}(N-1) & R_{xx}(N-2) & R_{xx}(N-3) & \cdots & R_{xx}(0) \end{bmatrix} \tag{11.31}$$

$$\mathbf{A} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_N \end{bmatrix} \tag{11.32}$$

$$\mathbf{P} = \begin{bmatrix} R_{xx}(1) \\ R_{xx}(2) \\ R_{xx}(3) \\ \vdots \\ R_{xx}(N) \end{bmatrix} \tag{11.33}$$

where we have used the fact that $R_{xx}(-k) = R_{xx}(k)$ for real valued wide sense stationary processes. These equations are referred to as the discrete form of the Wiener-Hopf equations. If we know the autocorrelation values $\{R_{xx}(k)\}$ for $k = 0, 1, \ldots, N$, then we can find the predictor coefficients as

$$\mathbf{A} = \mathbf{R}^{-1}\mathbf{P}. \tag{11.34}$$

Example 11.4.1:

For the speech sequence shown in Figure 11.7, let us find predictors of orders one, two, and three and examine their performance. We begin by estimating the autocorrelation values from the data. Given M data points, we use the following average to find the value for $R_{xx}(k)$:

$$R_{xx}(k) = \frac{1}{M-k} \sum_{i=1}^{M-k} x_i x_{i+k}. \tag{11.35}$$

Using these autocorrelation values, we obtain the following coefficients for the three different predictors. For $N = 1$, the predictor coefficient is $a_1 = 0.66$; for $N = 2$, the coefficients are $a_1 = 0.596$, $a_2 = 0.096$; and for $N = 3$, the coefficients are $a_1 = 0.577$, $a_2 = -0.025$, and $a_3 = 0.204$. We used these coefficients to generate the residual sequence. In order to see the reduction in variance, we computed the ratio of the source output variance to the variance of

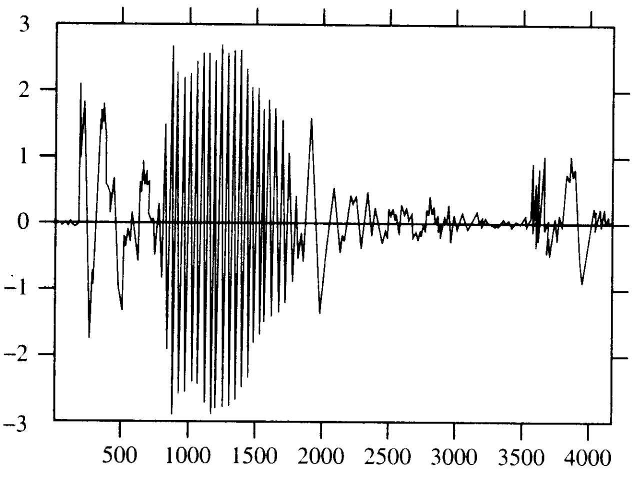

FIGURE 11.7 **A segment of speech: a male speaker saying the word "test."**

the residual sequence. For comparison, we also computed this ratio for the case where the residual sequence is obtained by taking the difference of neighboring samples. The sample-to-sample differences resulted in a ratio of 1.63. Compared to this, the ratio of the input variance to the variance of the residuals from the first-order predictor was 2.04. With a second-order predictor, this ratio rose to 3.37, and with a third-order predictor, the ratio was 6.28.

The residual sequence for the third-order predictor is shown in Figure 11.8. Notice that although there has been a reduction in the dynamic range, there is still substantial structure

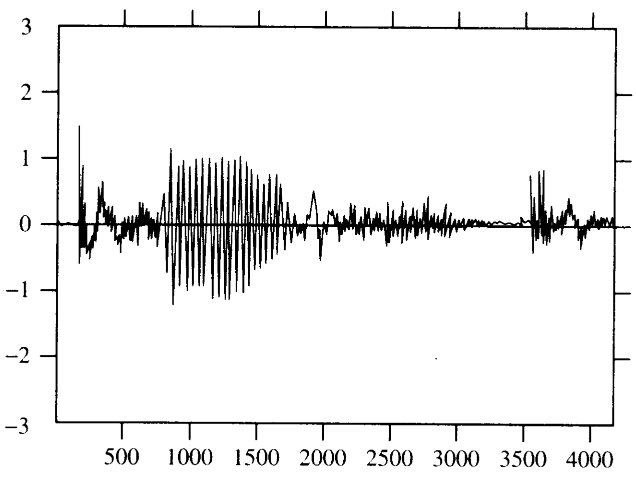

FIGURE 11.8 **The residual sequence using a third-order predictor.**

in the residual sequence, especially in the range of samples from about the 700th sample to the 2000th sample. We will look at ways of removing this structure when we discuss speech coding.

Let us now introduce a quantizer into the loop and look at the performance of the DPCM system. For simplicity, we will use a uniform quantizer. If we look at the histogram of the residual sequence, we find that it is highly peaked. Therefore, we will assume that the input to the quantizer will be Laplacian. We will also adjust the step size of the quantizer based on the variance of the residual. The step sizes provided in Chapter 9 are based on the assumption that the quantizer input has a unit variance. It is easy to show that when the variance differs from unity, the optimal step size can be obtained by multiplying the step size for a variance of one with the standard deviation of the input. Using this approach for a four-level Laplacian quantizer, we obtain step sizes of 0.75, 0.59, and 0.43 for the first-, second-, and third-order predictors, and step sizes of 0.3, 0.4, and 0.5 for an eight-level Laplacian quantizer. We measure the performance using two different measures; the signal-to-noise ratio (SNR) and the signal-to-prediction-error ratio. These are defined as follows:

$$\text{SNR(dB)} = \frac{\sum_{i=1}^{M} x_i^2}{\sum_{i=1}^{M}(x_i - \hat{x}_i)^2} \tag{11.36}$$

$$\text{SPER(dB)} = \frac{\sum_{i=1}^{M} x_i^2}{\sum_{i=1}^{M}(x_i - p_i)^2}. \tag{11.37}$$

The results are tabulated in Table 11.1. For comparison we have also included the results when no prediction is used; that is, we directly quantize the input. Notice the large difference between using a first-order predictor and a second-order predictor, and then the relatively minor increase when going from a second-order predictor to a third-order predictor. This is fairly typical when using a fixed quantizer.

Finally, let's take a look at the reconstructed speech signal. The speech coded using a third-order predictor and an eight-level quantizer is shown in Figure 11.9. Although the reconstructed sequence looks like the original, notice that there is significant distortion in areas where the source output values are small. This is because in these regions the input to the quantizer is close to zero. Because the quantizer does not have a zero output level,

TABLE 11.1 Performance of DPCM system with different predictors and quantizers.

Quantizer	Predictor Order	SNR (dB)	SPER (dB)
Four-level	None	2.43	0
	1	3.37	2.65
	2	8.35	5.9
	3	8.74	6.1
Eight-level	None	3.65	0
	1	3.87	2.74
	2	9.81	6.37
	3	10.16	6.71

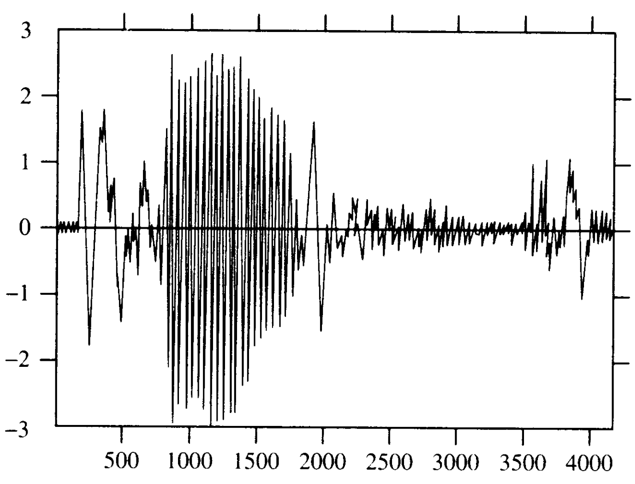

FIGURE 11. 9 **The reconstructed sequence using a third-order predictor and an eight-level uniform quantizer.**

the output of the quantizer flips between the two inner levels. If we listened to this signal, we would hear a hissing sound in the reconstructed signal.

The speech signal used to generate this example is contained among the data sets accompanying this book in the file `testm.raw`. The function `readau.c` can be used to read the file. You are encouraged to reproduce the results in this example and listen to the resulting reconstructions. ♦

If we look at the speech sequence in Figure 11.7, we can see that there are several distinct segments of speech. Between sample number 700 and sample number 2000, the speech looks periodic. Between sample number 2200 and sample number 3500, the speech is low amplitude and noiselike. Given the distinctly different characteristics in these two regions, it would make sense to use different approaches to encode these segments. Some approaches to dealing with these issues are specific to speech coding, and we will discuss these approaches when we specifically discuss encoding speech using DPCM. However, the problem is also much more widespread than when encoding speech. A general response to the nonstationarity of the input is the use of adaptation in prediction. We will look at some of these approaches in the next section.

11.5 Adaptive DPCM

As DPCM consists of two main components, the quantizer and the predictor, making DPCM adaptive means making the quantizer and the predictor adaptive. Recall that we can adapt a system based on its input or output. The former approach is called forward adaptation; the latter, backward adaptation. In the case of forward adaptation, the parameters of the system are updated based on the input to the encoder, which is not available to the decoder. Therefore, the updated parameters have to be sent to the decoder as side information. In the case of backward adaptation, the adaptation is based on the output of the encoder. As this output is also available to the decoder, there is no need for transmission of side information.

In cases where the predictor is adaptive, especially when it is backward adaptive, we generally use adaptive quantizers (forward or backward). The reason for this is that the backward adaptive predictor is adapted based on the quantized outputs. If for some reason the predictor does not adapt properly at some point, this results in predictions that are far from the input, and the residuals will be large. In a fixed quantizer, these large residuals will tend to fall in the overload regions with consequently unbounded quantization errors. The reconstructed values with these large errors will then be used to adapt the predictor, which will result in the predictor moving further and further from the input.

The same constraint is not present for quantization, and we can have adaptive quantization with fixed predictors.

11.5.1 Adaptive Quantization in DPCM

In forward adaptive quantization, the input is divided into blocks. The quantizer parameters are estimated for each block. These parameters are transmitted to the receiver as side information. In DPCM, the quantizer is in a feedback loop, which means that the input to the quantizer is not conveniently available in a form that can be used for forward adaptive quantization. Therefore, most DPCM systems use backward adaptive quantization.

The backward adaptive quantization used in DPCM systems is basically a variation of the backward adaptive Jayant quantizer described in Chapter 9. In Chapter 9, the Jayant algorithm was used to adapt the quantizer to a stationary input. In DPCM, the algorithm is used to adapt the quantizer to the local behavior of nonstationary inputs. Consider the speech segment shown in Figure 11.7 and the residual sequence shown in Figure 11.8. Obviously, the quantizer used around the 3000th sample should not be the same quantizer that was used around the 1000th sample. The Jayant algorithm provides an effective approach to adapting the quantizer to the variations in the input characteristics.

Example 11.5.1:

Let's encode the speech sample shown in Figure 11.7 using a DPCM system with a backward adaptive quantizer. We will use a third-order predictor and an eight-level quantizer. We will also use the following multipliers [110]:

$$M_0 = 0.90, \quad M_1 = 0.90, \quad M_2 = 1.25, \quad M_3 = 1.75.$$

The results are shown in Figure 11.10. Notice the region at the beginning of the speech sample and between the 3000th and 3500th sample, where the DPCM system with the fixed quantizer had problems. Because the step size of the adaptive quantizer can become quite small, these regions have been nicely reproduced. However, right after this region, the speech output has a larger spike than the reconstructed waveform. This is an indication that the quantizer is not expanding rapidly enough. This can be remedied by increasing the

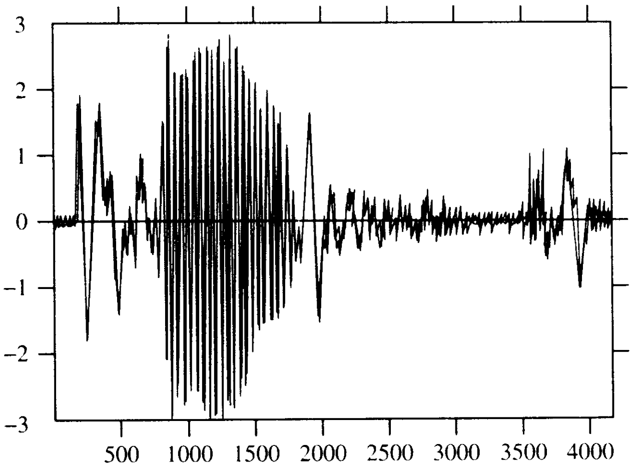

FIGURE 11.10 **The reconstructed sequence using a third-order predictor and an eight-level Jayant quantizer.**

value of M_3. The program used to generate this example is dpcm_aqb. You can use this program to study the behavior of the system for different configurations. ♦

11.5.2 Adaptive Prediction in DPCM

The equations used to obtain the predictor coefficients were derived based on the assumption of stationarity. However, we see from Figure 11.7 that this assumption is not true. In the speech segment shown in Figure 11.7, different segments have different characteristics. This is true for most sources we deal with; while the source output may be locally stationary over any significant length of the output, the statistics may vary considerably. In this situation, it is better to adapt the predictor to match the local statistics. This adaptation can be forward adaptive or backward adaptive.

DPCM with Forward Adaptive Prediction (DPCM-APF)

In forward adaptive prediction, the input is divided into segments or blocks. In speech coding this block consists of about 16 ms of speech. At a sampling rate of 8000 samples per second, this corresponds to 128 samples per block [123, 167]. In image coding, we use an 8×8 block [168].

The autocorrelation coefficients are computed for each block. The predictor coefficients are obtained from the autocorrelation coefficients and quantized using a relatively high-rate quantizer. If the coefficient values are to be quantized directly, we need to use at least 12 bits per coefficient [123]. This number can be reduced considerably if we represent the predictor coefficients in terms of *parcor coefficients*; we will describe how to obtain the parcor coefficients in Chapter 17. For now, let's assume that the coefficients can be transmitted with an expenditure of about 6 bits per coefficient.

In order to estimate the autocorrelation for each block, we generally assume that the sample values outside each block are zero. Therefore, for a block length of M, the autocorrelation function for the lth block would be estimated by

$$R_{xx}^{(l)}(k) = \frac{1}{M-k} \sum_{i=(l-1)M+1}^{lM-k} x_i x_{i+k} \tag{11.38}$$

for k positive, or

$$R_{xx}^{(l)}(k) = \frac{1}{M+k} \sum_{i=(l-1)M+1-k}^{lM} x_i x_{i+k} \tag{11.39}$$

for k negative. Notice that $R_{xx}^{(l)}(k) = R_{xx}^{(l)}(-k)$, which agrees with our initial assumption.

DPCM with Backward Adaptive Prediction (DPCM-APB)

Forward adaptive prediction requires that we buffer the input. This introduces delay in the transmission of the speech. As the amount of buffering is small, the use of forward adaptive prediction when there is only one encoder and decoder is not a big problem. However, in the case of speech, the connection between two parties may be several links, each of which may consist of a DPCM encoder and decoder. In such tandem links, the amount of delay can become large enough to be a nuisance. Furthermore, the need to transmit side information makes the system more complex. In order to avoid these problems, we can adapt the predictor based on the output of the encoder, which is also available to the decoder. The adaptation is done in a sequential manner [169, 167].

In our derivation of the optimum predictor coefficients, we took the derivative of the statistical average of the squared prediction error or residual sequence. In order to do this, we had to assume that the input process was stationary. Let us now remove that assumption and try to figure out how to adapt the predictor to the input algebraically. To keep matters simple, we will start with a first-order predictor and then generalize the result to higher orders.

For a first-order predictor, the value of the residual squared at time n would be given by

$$d_n^2 = (x_n - a_1 \hat{x}_{n-1})^2. \tag{11.40}$$

If we could plot the value of d_n^2 against a_1, we would get a graph similar to the one shown in Figure 11.11. Let's take a look at the derivative of d_n^2 as a function of whether the current value of a_1 is to the left or right of the optimal value of a_1—that is, the value of a_1 for which d_n^2 is minimum. When a_1 is to the left of the optimal value, the derivative is negative. Furthermore, the derivative will have a larger magnitude when a_1 is further away from the optimal value. If we were asked to adapt a_1, we would add to the current value of a_1. The amount to add would be large if a_1 was far from the optimal value, and small if a_1 was close to the optimal value. If the current value was to the right of the optimal value, the derivative would be positive, and we would subtract some amount from a_1 to adapt it. The

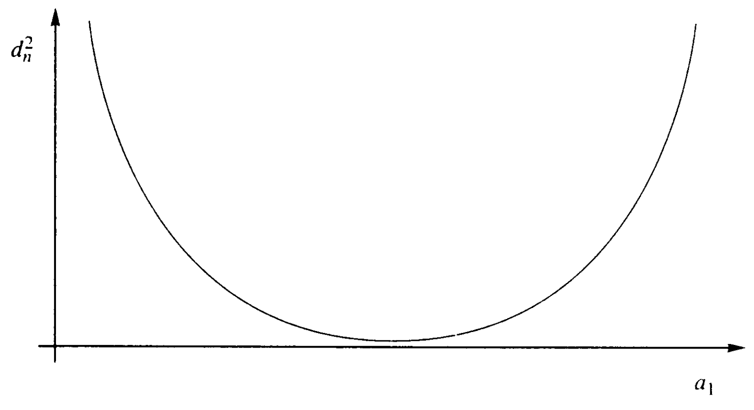

FIGURE 11.11 A plot of the residual squared versus the predictor coefficient.

amount to subtract would be larger if we were further from the optimal, and as before, the derivative would have a larger magnitude if a_1 were further from the optimal value.

At any given time, in order to adapt the coefficient at time $n+1$, we add an amount proportional to the magnitude of the derivative with a sign that is opposite to that of the derivative of d_n^2 at time n:

$$a_1^{(n+1)} = a_1^{(n)} - \alpha \frac{\delta d_n^2}{\delta a_1} \tag{11.41}$$

where α is some proportionality constant.

$$\frac{\delta d_n^2}{\delta a_1} = -2(x_n - a_1\hat{x}_{n-1})\hat{x}_{n-1} \tag{11.42}$$

$$= -2d_n\hat{x}_{n-1}. \tag{11.43}$$

Substituting this into (11.41), we get

$$a_1^{(n+1)} = a_1^{(n)} + \alpha d_n \hat{x}_{n-1} \tag{11.44}$$

where we have absorbed the 2 into α. The residual value d_n is available only to the encoder. Therefore, in order for both the encoder and decoder to use the same algorithm, we replace d_n by $\hat{d}_n$ in (11.44) to obtain

$$a_1^{(n+1)} = a_1^{(n)} + \alpha \hat{d}_n \hat{x}_{n-1}. \tag{11.45}$$

Extending this adaptation equation for a first-order predictor to an Nth-order predictor is relatively easy. The equation for the squared prediction error is given by

$$d_n^2 = \left(x_n - \sum_{i=1}^{N} a_i \hat{x}_{n-i}\right)^2. \tag{11.46}$$

Taking the derivative with respect to a_j will give us the adaptation equation for the jth predictor coefficient:

$$a_j^{(n+1)} = a_j^{(n)} + \alpha \hat{d}_n \hat{x}_{n-j}. \tag{11.47}$$

We can combine all N equations in vector form to get

$$\mathbf{A}^{(n+1)} = \mathbf{A}^{(n)} + \alpha \hat{d}_n \hat{X}_{n-1} \tag{11.48}$$

where

$$\hat{X}_n = \begin{bmatrix} \hat{x}_n \\ \hat{x}_{n-1} \\ \vdots \\ \hat{x}_{n-N+1} \end{bmatrix}. \tag{11.49}$$

This particular adaptation algorithm is called the least mean squared (LMS) algorithm [170].

11.6 Delta Modulation

A very simple form of DPCM that has been widely used in a number of speech-coding applications is the delta modulator (DM). The DM can be viewed as a DPCM system with a 1-bit (two-level) quantizer. With a two-level quantizer with output values $\pm\Delta$, we can only represent a sample-to-sample difference of Δ. If, for a given source sequence, the sample-to-sample difference is often very different from Δ, then we may incur substantial distortion. One way to limit the difference is to sample more often. In Figure 11.12 we see a signal that has been sampled at two different rates. The lower-rate samples are shown by open circles, while the higher-rate samples are represented by $+$. It is apparent that the lower-rate samples are further apart in value.

The rate at which a signal is sampled is governed by the highest frequency component of a signal. If the highest frequency component in a signal is W, then in order to obtain an exact reconstruction of the signal, we need to sample it at least at twice the highest frequency, or $2W$. In systems that use delta modulation, we usually sample the signal at much more than twice the highest frequency. If F_s is the sampling frequency, then the ratio of F_s to $2W$ can range from almost 1 to almost 100 [123]. The higher sampling rates are used for high-quality A/D converters, while the lower rates are more common for low-rate speech coders.

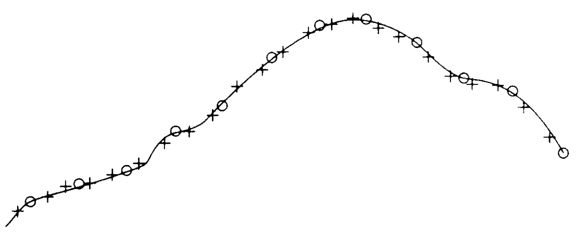

FIGURE 11.12 A signal sampled at two different rates.

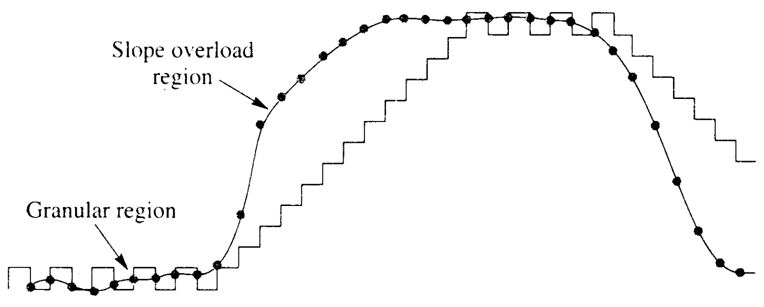

FIGURE 11.13 **A source output sampled and coded using delta modulation.**

If we look at a block diagram of a delta modulation system, we see that, while the block diagram of the encoder is identical to that of the DPCM system, the standard DPCM decoder is followed by a filter. The reason for the existence of the filter is evident from Figure 11.13, where we show a source output and the unfiltered reconstruction. The samples of the source output are represented by the filled circles. As the source is sampled at several times the highest frequency, the staircase shape of the reconstructed signal results in distortion in frequency bands outside the band of frequencies occupied by the signal. The filter can be used to remove these spurious frequencies.

The reconstruction shown in Figure 11.13 was obtained with a delta modulator using a fixed quantizer. Delta modulation systems that use a fixed step size are often referred to as linear delta modulators. Notice that the reconstructed signal shows one of two behaviors. In regions where the source output is relatively constant, the output alternates up or down by Δ; these regions are called the *granular regions*. In the regions where the source output rises or falls fast, the reconstructed output cannot keep up; these regions are called the *slope overload regions*. If we want to reduce the granular error, we need to make the step size Δ small. However, this will make it more difficult for the reconstruction to follow rapid changes in the input. In other words, it will result in an increase in the overload error. To avoid the overload condition, we need to make the step size large so that the reconstruction can quickly catch up with rapid changes in the input. However, this will increase the granular error.

One way to avoid this impasse is to adapt the step size to the characteristics of the input, as shown in Figure 11.14. In quasi-constant regions, make the step size small in order to reduce the granular error. In regions of rapid change, increase the step size in order to reduce overload error. There are various ways of adapting the delta modulator to the local characteristics of the source output. We describe two of the more popular ways here.

11.6.1 Constant Factor Adaptive Delta Modulation (CFDM)

The objective of adaptive delta modulation is clear: increase the step size in overload regions and decrease it in granular regions. The problem lies in knowing when the system is in each of these regions. Looking at Figure 11.13, we see that in the granular region the output of

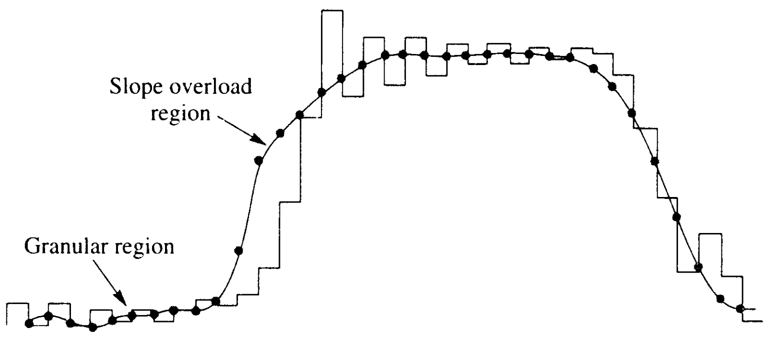

FIGURE 11.14 A source output sampled and coded using adaptive delta modulation.

the quantizer changes sign with almost every input sample; in the overload region, the sign of the quantizer output is the same for a string of input samples. Therefore, we can define an overload or granular condition based on whether the output of the quantizer has been changing signs. A very simple system [171] uses a history of one sample to decide whether the system is in overload or granular condition and whether to expand or contract the step size. If s_n denotes the sign of the quantizer output $\hat{d}_n$,

$$s_n = \begin{cases} 1 & \text{if } \hat{d}_n > 0 \\ -1 & \text{if } \hat{d}_n < 0 \end{cases} \tag{11.50}$$

the adaptation logic is given by

$$\Delta_n = \begin{cases} M_1 \Delta_{n-1} & s_n = s_{n-1} \\ M_2 \Delta_{n-1} & s_n \neq s_{n-1} \end{cases} \tag{11.51}$$

where $M_1 = \frac{1}{M_2} = M > 1$. In general, $M < 2$.

By increasing the memory, we can improve the response of the CFDM system. For example, if we looked at two past samples, we could decide that the system was moving from overload to granular condition if the sign had been the same for the past two samples and then changed with the current sample:

$$s_n \neq s_{n-1} = s_{n-2}. \tag{11.52}$$

In this case it would be reasonable to assume that the step size had been expanding previously and, therefore, needed a sharp contraction. If

$$s_n = s_{n-1} \neq s_{n-2}, \tag{11.53}$$

then it would mean that the system was probably entering the overload region, while

$$s_n = s_{n-1} = s_{n-2} \tag{11.54}$$

would mean the system was in overload and the step size should be expanded rapidly.

For the encoding of speech, the following multipliers M_i are recommended by [172] for a CFDM system with two-sample memory:

$$s_n \neq s_{n-1} = s_{n-2} \qquad M_1 = 0.4 \tag{11.55}$$

$$s_n \neq s_{n-1} \neq s_{n-2} \qquad M_2 = 0.9 \tag{11.56}$$

$$s_n = s_{n-1} \neq s_{n-2} \qquad M_3 = 1.5 \tag{11.57}$$

$$s_n = s_{n-1} = s_{n-2} \qquad M_4 = 2.0. \tag{11.58}$$

The amount of memory can be increased further with a concurrent increase in complexity. The space shuttle used a delta modulator with a memory of seven [173].

11.6.2 Continuously Variable Slope Delta Modulation

The CFDM systems described use a rapid adaptation scheme. For low-rate speech coding, it is more pleasing if the adaptation is over a longer period of time. This slower adaptation results in a decrease in the granular error and generally an increase in overload error. Delta modulation systems that adapt over longer periods of time are referred to as *syllabically* companded. A popular class of syllabically companded delta modulation systems is the continuously variable slope delta modulation systems.

The adaptation logic used in CVSD systems is as follows [123]:

$$\Delta_n = \beta\Delta_{n-1} + \alpha_n\Delta_0 \tag{11.59}$$

where β is a number less than but close to one, and α_n is equal to one if J of the last K quantizer outputs were of the same sign. That is, we look in a window of length K to obtain the behavior of the source output. If this condition is not satisfied, then α_n is equal to zero. Standard values for J and K are $J = 3$ and $K = 3$.

11.7 Speech Coding

Differential encoding schemes are immensely popular for speech encoding. They are used in the telephone system, voice messaging, and multimedia applications, among others. Adaptive DPCM is a part of several international standards (ITU-T G.721, ITU G.723, ITU G.726, ITU-T G.722), which we will look at here and in later chapters.

Before we do that, let's take a look at one issue specific to speech coding. In Figure 11.7, we see that there is a segment of speech that looks highly periodic. We can see this periodicity if we plot the autocorrelation function of the speech segment (Figure 11.15).

The autocorrelation peaks at a lag value of 47 and multiples of 47. This indicates a periodicity of 47 samples. This period is called the *pitch period*. The predictor we originally designed did not take advantage of this periodicity, as the largest predictor was a third-order predictor, and this periodic structure takes 47 samples to show up. We can take advantage of this periodicity by constructing an outer prediction loop around the basic DPCM structure

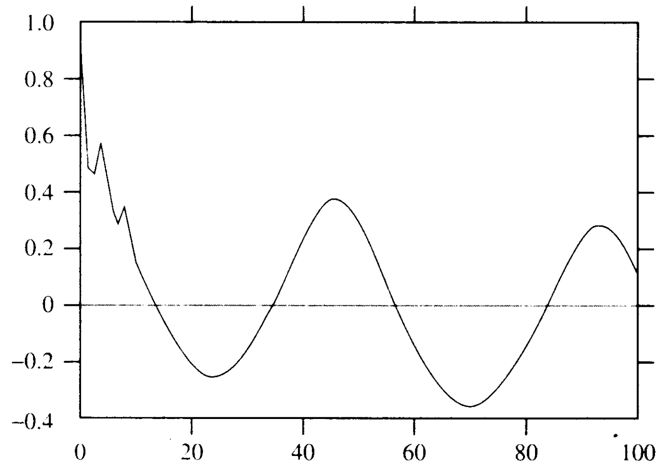

FIGURE 11.15 **Autocorrelation function for** `test.snd`.

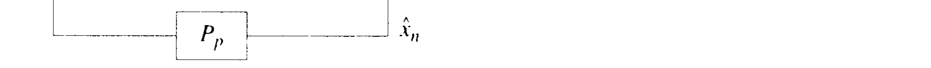

FIGURE 11.16 **The DPCM structure with a pitch predictor.**

as shown in Figure 11.16. This can be a simple single coefficient predictor of the form $b\hat{x}_{n-\tau}$, where τ is the pitch period. Using this system on `testm.raw`, we get the residual sequence shown in Figure 11.17. Notice the decrease in amplitude in the periodic portion of the speech.

Finally, remember that we have been using mean squared error as the distortion measure in all of our discussions. However, perceptual tests do not always correlate with the mean squared error. The level of distortion we perceive is often related to the level of the speech signal. In regions where the speech signal is of higher amplitude, we have a harder time perceiving the distortion, but the same amount of distortion in a different frequency band might be very perceptible. We can take advantage of this by shaping the quantization error so that most of the error lies in the region where the signal has a higher amplitude. This variation of DPCM is called *noise feedback coding* (NFC) (see [123] for details).

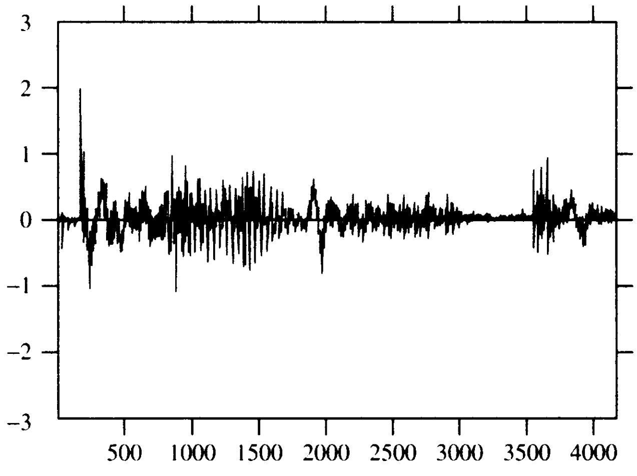

FIGURE 11.17 The residual sequence using the DPCM system with a pitch predictor.

11.7.1 G.726

The International Telecommunications Union has published recommendations for a standard ADPCM system, including recommendations G.721, G.723, and G.726. G.726 supersedes G.721 and G.723. In this section we will describe the G.726 recommendation for ADPCM systems at rates of 40, 32, 24, and 16 kbits.

The Quantizer

The recommendation assumes that the speech output is sampled at the rate of 8000 samples per second, so the rates of 40, 32, 24, and 16 kbits per second translate 5 bits per sample, 4 bits per sample, 3 bits per sample, and 2 bits per sample. Comparing this to the PCM rate of 8 bits per sample, this would mean compression ratios of 1.6:1, 2:1, 2.67:1, and 4:1. Except for the 16 kbits per second system, the number of levels in the quantizer are $2^{nb} - 1$, where nb is the number of bits per sample. Thus, the number of levels in the quantizer is odd, which means that for the higher rates we use a midtread quantizer.

The quantizer is a backward adaptive quantizer with an adaptation algorithm that is similar to the Jayant quantizer. The recommendation describes the adaptation of the quantization interval in terms of the adaptation of a scale factor. The input d_k is normalized by a scale factor α_k. This normalized value is quantized, and the normalization removed by multiplying with α_k. In this way the quantizer is kept fixed and α_k is adapted to the input. Therefore, for example, instead of expanding the step size, we would increase the value of α_k.

The fixed quantizer is a nonuniform midtread quantizer. The recommendation describes the quantization boundaries and reconstruction values in terms of the log of the scaled input. The input-output characteristics for the 24 kbit system are shown in Table 11.2. An output value of $-\infty$ in the table corresponds to a reconstruction value of 0.

TABLE 11.2 Recommended input-output characteristics of the quantizer for 24-kbits-per-second operation.

Input Range	Label	Output
$\log_2 \frac{d_k}{\alpha_k}$	$\lvert I_k \rvert$	$\log_2 \frac{d_k}{\alpha_k}$
$[2.58, \infty)$	3	2.91
$[1.70, 2.58)$	2	2.13
$[0.06, 1.70)$	1	1.05
$(-\infty, -0.06)$	0	$-\infty$

The adaptation algorithm is described in terms of the logarithm of the scale factor

$$y(k) = \log_2 \alpha_k. \tag{11.60}$$

The adaptation of the scale factor α or its log $y(k)$ depends on whether the input is speech or speechlike, where the sample-to-sample difference can fluctuate considerably, or whether the input is voice-band data, which might be generated by a modem, where the sample-to-sample fluctuation is quite small. In order to handle both these situations, the scale factor is composed of two values, a *locked* slow scale factor for when the sample-to-sample differences are quite small, and an *unlocked* value for when the input is more dynamic:

$$y(k) = a_l(k)y_u(k-1) + (1 - a_l(k))y_l(k-1). \tag{11.61}$$

The value of $a_l(k)$ depends on the variance of the input. It will be close to one for speech inputs and close to zero for tones and voice band data.

The unlocked scale factor is adapted using the Jayant algorithm with one slight modification. If we were to use the Jayant algorithm, the unlocked scale factor could be adapted as

$$\alpha_u(k) = \alpha_{k-1} M[I_{k-1}] \tag{11.62}$$

where $M[\cdot]$ is the multiplier. In terms of logarithms, this becomes

$$y_u(k) = y(k-1) + \log M[I_{k-1}]. \tag{11.63}$$

The modification consists of introducing some memory into the adaptive process so that the encoder and decoder converge following transmission errors:

$$y_u(k) = (1-\epsilon)y(k-1) + \epsilon W[I_{k-1}] \tag{11.64}$$

where $W[\cdot] = \log M[\cdot]$, and $\epsilon = 2^{-5}$.

The locked scale factor is obtained from the unlocked scale factor through

$$y_l(k) = (1-\gamma)y_l(k-1) + \gamma y_u(k), \qquad \gamma = 2^{-6}. \tag{11.65}$$

The Predictor

The recommended predictor is a backward adaptive predictor that uses a linear combination of the past two reconstructed values as well as the six past quantized differences to generate the prediction

$$p_k = \sum_{i=1}^{2} a_i^{(k-1)} \hat{x}_{k-i} + \sum_{i=1}^{6} b_i^{(k-1)} \hat{d}_{k-i}. \tag{11.66}$$

The set of predictor coefficients is updated using a simplified form of the LMS algorithm.

$$a_1^{(k)} = (1-2^{-8})a_1^{(k-1)} + 3\times 2^{-8}\text{sgn}[z(k)]\text{sgn}[z(k-1)] \tag{11.67}$$

$$\begin{aligned} a_2^{(k)} &= (1-2^{-7})a_2^{(k-1)} + 2^{-7}(\text{sgn}[z(k)]\text{sgn}[z(k-2)] \\ &\quad -f\left(a_1^{(k-1)}\text{sgn}[z(k)]\text{sgn}[z(k-1)]\right)\Big) \end{aligned} \tag{11.68}$$

where

$$z(k) = \hat{d}_k + \sum_{i=1}^{6} b_i^{(k-1)} \hat{d}_{k-i} \tag{11.69}$$

$$f(\beta) = \begin{cases} 4\beta & |\beta| \le \frac{1}{2} \\ 2\text{sgn}(\beta) & |\beta| > \frac{1}{2}. \end{cases} \tag{11.70}$$

The coefficients $\{b_i\}$ are updated using the following equation:

$$b_i^{(k)} = (1-2^{-8})b_i^{(k-1)} + 2^{-7}\text{sgn}[\hat{d}_k]\text{sgn}[\hat{d}_{k-i}]. \tag{11.71}$$

Notice that in the adaptive algorithms we have replaced products of reconstructed values and products of quantizer outputs with products of their signs. This is computationally much simpler and does not lead to any significant degradation of the adaptation process. Furthermore, the values of the coefficients are selected such that multiplication with these coefficients can be accomplished using shifts and adds. The predictor coefficients are all set to zero when the input moves from tones to speech.

11.8 Image Coding

We saw in Chapter 7 that differential encoding provided an efficient approach to the lossless compression of images. The case for using differential encoding in the lossy compression of images has not been made as clearly. In the early days of image compression, both differential encoding and transform coding were popular forms of lossy image compression. At the current time differential encoding has a much more restricted role as part of other compression strategies. Several currently popular approaches to image compression decompose the image into lower and higher frequency components. As low-frequency signals have high sample-to-sample correlation, several schemes use differential encoding to compress the low-frequency components. We will see this use of differential encoding when we look at subband- and wavelet-based compression schemes and, to a lesser extent, when we study transform coding.

For now let us look at the performance of a couple of stand-alone differential image compression schemes. We will compare the performance of these schemes with the performance of the JPEG compression standard.

Consider a simple differential encoding scheme in which the predictor $p[j,k]$ for the pixel in the jth row and the kth column is given by

$$p[j,k] = \begin{cases} \hat{x}[j,k-1] & \text{for } k > 0 \\ \hat{x}[j-1,k] & \text{for } k = 0 \text{ and } j > 0 \\ 128 & \text{for } j = 0 \text{ and } k = 0 \end{cases}$$

where $\hat{x}[j,k]$ is the reconstructed pixel in the jth row and kth column. We use this predictor in conjunction with a fixed four-level uniform quantizer and code the quantizer output using an arithmetic coder. The coding rate for the compressed image is approximately 1 bit per pixel. We compare this reconstructed image with a JPEG-coded image at the same rate in Figure 11.18. The signal-to-noise ratio for the differentially encoded image is 22.33 dB (PSNR 31.42 dB) and for the JPEG-encoded image is 32.52 dB (PSNR 41.60 dB), a difference of more than 10 dB!

However, this is an extremely simple system compared to the JPEG standard, which has been fine-tuned for encoding images. Let's make our differential encoding system slightly more complicated by replacing the uniform quantizer with a recursively indexed quantizer and the predictor by a somewhat more complicated predictor. For each pixel (except for the boundary pixels) we compute the following three values:

$$\begin{aligned} p_1 &= 0.5 \times \hat{x}[j-1,k] + 0.5 \times \hat{x}[j,k-1] \\ p_2 &= 0.5 \times \hat{x}[j-1,k-1] + 0.5 \times \hat{x}[j,k-1] \\ p_3 &= 0.5 \times \hat{x}[j-1,k-1] + 0.5 \times \hat{x}[j-1,k] \end{aligned} \qquad (11.72)$$

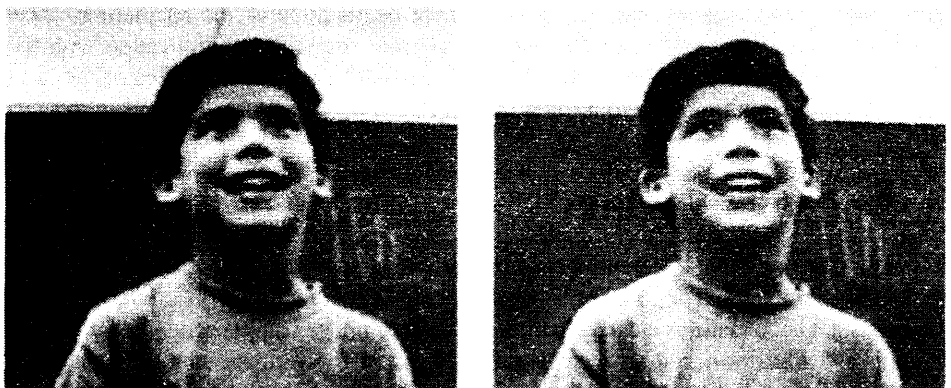

FIGURE 11.18 **Left: Reconstructed image using differential encoding at 1 bit per pixel. Right: Reconstructed image using JPEG at 1 bit per pixel.**

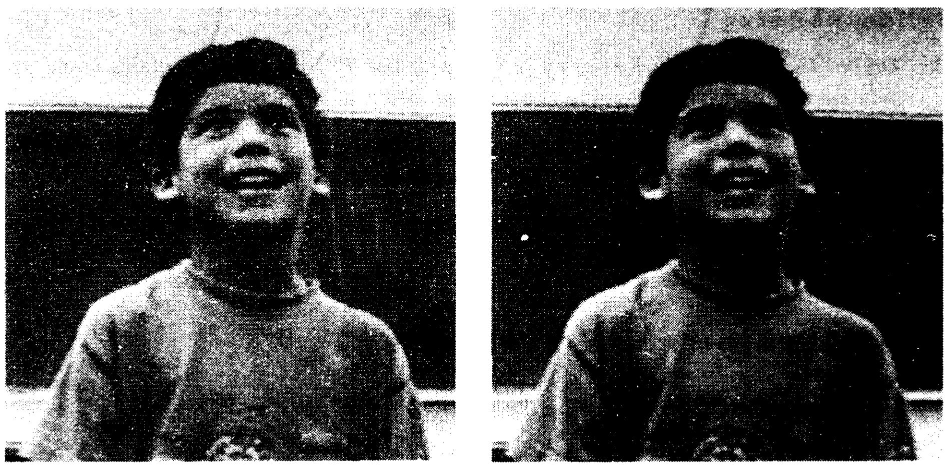

FIGURE 11.19 **Left: Reconstructed image using differential encoding at 1 bit per pixel using median predictor and recursively indexed quantizer. Right: Reconstructed image using JPEG at 1 bit per pixel.**

then obtain the predicted value as

$$p[j,k] = \text{median}\{p_1, p_2, p_3\}.$$

For the boundary pixels we use the simple prediction scheme. At a coding rate of 1 bit per pixel, we obtain the image shown in Figure 11.19. For reference we show it next to the JPEG-coded image at the same rate. The signal-to-noise ratio for this reconstruction is 29.20dB (PSNR 38.28 dB). We have made up two-thirds of the difference using some relatively minor modifications. We can see that it might be feasible to develop differential encoding schemes that are competitive with other image compression techniques. Therefore, it makes sense not to dismiss differential encoding out of hand when we need to develop image compression systems.

11.9 Summary

In this chapter we described some of the more well-known differential encoding techniques. Although differential encoding does not provide compression as high as vector quantization, it is very simple to implement. This approach is especially suited to the encoding of speech, where it has found broad application. The DPCM system consists of two main components, the quantizer and the predictor. We spent a considerable amount of time discussing the quantizer in Chapter 9, so most of the discussion in this chapter focused on the predictor. We have seen different ways of making the predictor adaptive, and looked at some of the improvements to be obtained from source-specific modifications to the predictor design.

Further Reading

1. *Digital Coding of Waveforms*, by N.S. Jayant and P. Noll [123], contains some very detailed and highly informative chapters on differential encoding.
2. "Adaptive Prediction in Speech Differential Encoding Systems," by J.D. Gibson [167], is a comprehensive treatment of the subject of adaptive prediction.
3. A real-time video coding system based on DPCM has been developed by NASA. Details can be found in [174].

11.10 Projects and Problems

1. Generate an AR(1) process using the relationship

$$x_n = 0.9 \times x_{n-1} + \epsilon_n$$

where ϵ_n is the output of a Gaussian random number generator (this is option 2 in `rangen`).

(a) Encode this sequence using a DPCM system with a one-tap predictor with predictor coefficient 0.9 and a three-level Gaussian quantizer. Compute the variance of the prediction error. How does this compare with the variance of the input? How does the variance of the prediction error compare with the variance of the $\{\epsilon_n\}$ sequence?

(b) Repeat using predictor coefficient values of 0.5, 0.6, 0.7, 0.8, and 1.0. Comment on the results.

2. Generate an AR(5) process using the following coefficients: 1.381, 0.6, 0.367, −0.7, 0.359.

(a) Encode this with a DPCM system with a 3-bit Gaussian nonuniform quantizer and a first-, second-, third-, fourth-, and fifth-order predictor. Obtain these predictors by solving (11.30). For each case compute the variance of the prediction error and the SNR in dB. Comment on your results.

(b) Repeat using a 3-bit Jayant quantizer.

3. DPCM can also be used for encoding images. Encode the Sinan image using a one-tap predictor of the form

$$\hat{x}_{i,j} = a \times x_{i,j-1}$$

and a 2-bit quantizer. Experiment with quantizers designed for different distributions. Comment on your results.

4. Repeat the image-coding experiment of the previous problem using a Jayant quantizer.
5. DPCM-encode the Sinan, Elif, and bookshelf1 images using a one-tap predictor and a four-level quantizer followed by a Huffman coder. Repeat using a five-level quantizer. Compute the SNR for each case, and compare the rate distortion performances.

6. We want to DPCM-encode images using a two-tap predictor of the form

$$\hat{x}_{i,j} = a \times x_{i,j-1} + b \times x_{i-1,j}$$

and a four-level quantizer followed by a Huffman coder. Find the equations we need to solve to obtain coefficients a and b that minimize the mean squared error.

7. **(a)** DPCM-encode the Sinan, Elif, and bookshelf1 images using a two-tap predictor and a four-level quantizer followed by a Huffman coder.

   (b) Repeat using a five-level quantizer. Compute the SNR and rate (in bits per pixel) for each case.

   (c) Compare the rate distortion performances with the one-tap case.

   (d) Repeat using a five-level quantizer. Compute the SNR for each case, and compare the rate distortion performances using a one-tap and two-tap predictor.

Mathematical Preliminaries for Transforms, Subbands, and Wavelets

12.1 Overview

In this chapter we will review some of the mathematical background necessary for the study of transforms, subbands, and wavelets. The topics include Fourier series, Fourier transforms, and their discrete counterparts. We will also look at sampling and briefly review some linear system concepts.

12.2 Introduction

The roots of many of the techniques we will study can be found in the mathematical literature. Therefore, in order to understand the techniques, we will need some mathematical background. Our approach in general will be to introduce the mathematical tools just prior to when they are needed. However, there is a certain amount of background that is required for most of what we will be looking at. In this chapter we will present only that material that is a common background to all the techniques we will be studying. Our approach will be rather utilitarian; more sophisticated coverage of these topics can be found in [175]. We will be introducing a rather large number of concepts, many of which depend on each other. In order to make it easier for you to find a particular concept, we will identify the paragraph in which the concept is first introduced.

We will begin our coverage with a brief introduction to the concept of vector spaces, and in particular the concept of the inner product. We will use these concepts in our description of Fourier series and Fourier transforms. Next is a brief overview of linear systems, then a look at the issues involved in sampling a function. Finally, we will revisit the Fourier concepts in the context of sampled functions and provide a brief introduction to Z-transforms. Throughout, we will try to get a physical feel for the various concepts.

12.3 Vector Spaces

The techniques we will be using to obtain compression will involve manipulations and decompositions of (sampled) functions of time. In order to do this we need some sort of mathematical framework. This framework is provided through the concept of vector spaces.

We are very familiar with vectors in two- or three-dimensional space. An example of a vector in two-dimensional space is shown in Figure 12.1. This vector can be represented in a number of different ways: we can represent it in terms of its magnitude and direction, or we can represent it as a weighted sum of the unit vectors in the x and y directions, or we can represent it as an array whose components are the coefficients of the unit vectors. Thus, the vector **v** in Figure 12.1 has a magnitude of 5 and an angle of 36.86 degrees,

$$\mathbf{v} = 4u_x + 3u_y$$

and

$$\mathbf{v} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}.$$

We can view the second representation as a decomposition of V into simpler building blocks, namely, the *basis vectors*. The nice thing about this is that any vector in two dimensions can be decomposed in exactly the same way. Given a particular vector **A** and a

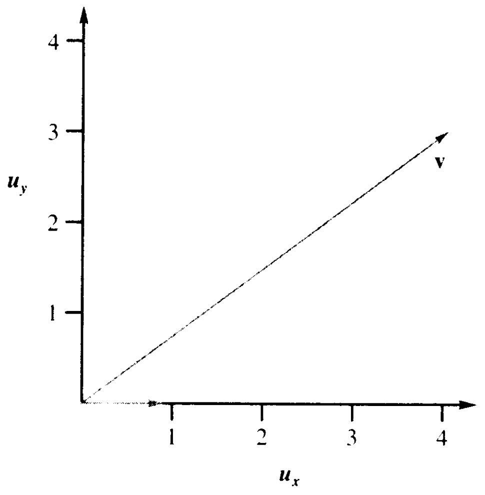

FIGURE 12.1 **A vector.**

basis set (more on this later), decomposition means finding the coefficients with which to weight the unit vectors of the basis set. In our simple example it is easy to see what these coefficients should be. However, we will encounter situations where it is not a trivial task to find the coefficients that constitute the decomposition of the vector. We therefore need some machinery to extract these coefficients. The particular machinery we will use here is called the *dot product* or the *inner product*.

12.3.1 Dot or Inner Product

Given two vectors **a** and **b** such that

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

the inner product between **a** and **b** is defined as

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2.$$

Two vectors are said to be *orthogonal* if their inner product is zero. A set of vectors is said to be orthogonal if each vector in the set is orthogonal to every other vector in the set. The inner product between a vector and a unit vector from an orthogonal basis set will give us the coefficient corresponding to that unit vector. It is easy to see that this is indeed so. We can write u_x and u_y as

$$u_x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad u_y = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

These are obviously orthogonal. Therefore, the coefficient a_1 can be obtained by

$$\mathbf{a} \cdot u_x = a_1 \times 1 + a_2 \times 0 = a_1$$

and the coefficient of u_y can be obtained by

$$\mathbf{a} \cdot u_y = a_1 \times 0 + a_2 \times 1 = a_2.$$

The inner product between two vectors is in some sense a measure of how "similar" they are, but we have to be a bit careful in how we define "similarity." For example, consider the vectors in Figure 12.2. The vector **a** is closer to u_x than to u_y. Therefore $\mathbf{a} \cdot u_x$ will be greater than $\mathbf{a} \cdot u_y$. The reverse is true for **b**.

12.3.2 Vector Space

In order to handle not just two- or three-dimensional vectors but general sequences and functions of interest to us, we need to generalize these concepts. Let us begin with a more general definition of vectors and the concept of a vector space.

> A *vector space* consists of a set of elements called vectors that have the operations of vector addition and scalar multiplication defined on them. Furthermore, the results of these operations are also elements of the vector space.

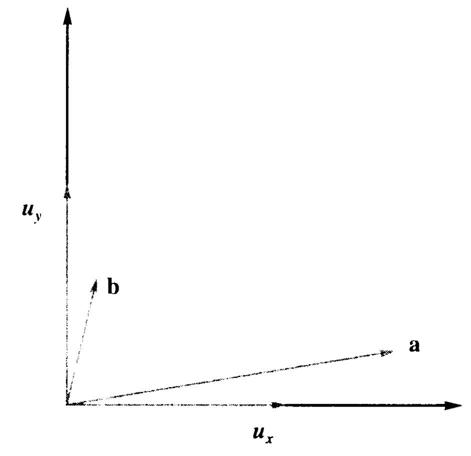

FIGURE 12.2 Example of different vectors.

By *vector addition* of two vectors, we mean the vector obtained by the pointwise addition of the components of the two vectors. For example, given two vectors **a** and **b**:

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{12.1}$$

the vector addition of these two vectors is given as

$$\mathbf{a}+\mathbf{b} = \begin{bmatrix} a_1+b_1 \\ a_2+b_2 \\ a_3+b_3 \end{bmatrix}. \tag{12.2}$$

By *scalar multiplication*, we mean the multiplication of a vector with a real or complex number. For this set of elements to be a vector space it has to satisfy certain axioms.

Suppose V is a vector space; $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are vectors; and α and β are scalars. Then the following axioms are satisfied:

1. $\mathbf{x}+\mathbf{y} = \mathbf{y}+\mathbf{x}$ (commutativity).
2. $(\mathbf{x}+\mathbf{y})+\mathbf{z} = \mathbf{x}+(\mathbf{y}+\mathbf{z})$ and $(\alpha\beta)\mathbf{x} = \alpha(\beta\mathbf{x})$ (associativity).
3. There exists an element θ in V such that $\mathbf{x}+\theta = \mathbf{x}$ for all $\mathbf{x}$ in V. θ is called the additive identity.
4. $\alpha(\mathbf{x}+\mathbf{y}) = \alpha\mathbf{x}+\alpha\mathbf{y}$, and $(\alpha+\beta)\mathbf{x} = \alpha\mathbf{x}+\beta\mathbf{x}$ (distributivity).
5. $1\cdot\mathbf{x} = \mathbf{x}$, and $0\cdot\mathbf{x} = \theta$.
6. For every $\mathbf{x}$ in V, there exists a $(-\mathbf{x})$ such that $\mathbf{x}+(-\mathbf{x}) = \theta$.

A simple example of a vector space is the set of real numbers. In this set zero is the additive identity. We can easily verify that the set of real numbers with the standard operations of addition and multiplication obey the axioms stated above. See if you can verify that the set of real numbers is a vector space. One of the advantages of this exercise is to emphasize the fact that a vector is more than a line with an arrow at its end.

Example 12.3.1:

Another example of a vector space that is of more practical interest to us is the set of all functions $f(t)$ with finite energy. That is,

$$\int_{-\infty}^{\infty} |f(t)|^2 \, dt < \infty. \tag{12.3}$$

Let's see if this set constitutes a vector space. If we define additions as pointwise addition and scalar multiplication in the usual manner, the set of functions $f(t)$ obviously satisfies axioms 1, 2, and 4.

- If $f(t)$ and $g(t)$ are functions with finite energy, and α is a scalar, then the functions $f(t)+g(t)$ and $\alpha f(t)$ also have finite energy.
- If $f(t)$ and $g(t)$ are functions with finite energy, then $f(t)+g(t)=g(t)+f(t)$ (axiom 1).
- If $f(t)$, $g(t)$, and $h(t)$ are functions with finite energy, and α and β are scalars, then $(f(t)+g(t))+h(t)=f(t)+(g(t)+h(t))$ and $(\alpha\beta)f(t)=\alpha(\beta f(t))$ (axiom 2).
- If $f(t)$, $g(t)$, and $h(t)$ are functions with finite energy, and α is a scalar, then $\alpha(f(t)+g(t))=\alpha f(t)+\alpha g(t)$ and $(\alpha+\beta)f(t)=\alpha f(t)+\beta f(t)$ (axiom 4).

Let us define the additive identity function $\theta(t)$ as the function that is identically zero for all t. This function satisfies the requirement of finite energy, and we can see that axioms 3 and 5 are also satisfied. Finally, if a function $f(t)$ has finite energy, then from Equation (12.3), the function $-f(t)$ also has finite energy, and axiom 6 is satisfied. Therefore, the set of all functions with finite energy constitutes a vector space. This space is denoted by $L_2(f)$, or simply L_2. ♦

12.3.3 Subspace

A *subspace* S of a vector space V is a subset of V whose members satisfy all the axioms of the vector space and has the additional property that if $\mathbf{x}$ and $\mathbf{y}$ are in S, and α is a scalar, then $\mathbf{x}+\mathbf{y}$ and $\alpha\mathbf{x}$ are also in S.

Example 12.3.2:

Consider the set S of continuous bounded functions on the interval [0, 1]. Then S is a subspace of the vector space L_2. ♦

12.3.4 Basis

One way we can generate a subspace is by taking linear combinations of a set of vectors. If this set of vectors is *linearly independent*, then the set is called a *basis* for the subspace.

> A set of vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots\}$ is said to be linearly independent if no vector of the set can be written as a linear combination of the other vectors in the set.

A direct consequence of this definition is the following theorem:

Theorem *A set of vectors* $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ *is linearly independent if and only if the expression* $\sum_{i=1}^{N} \alpha_i \mathbf{x}_i = \theta$ *implies that* $\alpha_i = 0$ *for all* $i = 1, 2, \ldots, N$.

Proof The proof of this theorem can be found in most books on linear algebra [175]. □

The set of vectors formed by all possible linear combinations of vectors from a linearly independent set **X** forms a vector space (Problem 1). The set **X** is said to be the *basis* for this vector space. The basis set contains the smallest number of linearly independent vectors required to represent each element of the vector space. More than one set can be the basis for a given space.

Example 12.3.3:

Consider the vector space consisting of vectors $[ab]^T$, where a and b are real numbers. Then the set

$$\mathbf{X} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}$$

forms a basis for this space, as does the set

$$\mathbf{X} = \left\{ \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}.$$

In fact, any two vectors that are not scalar multiples of each other form a basis for this space. ♦

The number of basis vectors required to generate the space is called the *dimension* of the vector space. In the previous example the dimension of the vector space is two. The dimension of the space of all continuous functions on the interval [0, 1] is infinity.

Given a particular basis, we can find a representation with respect to this basis for any vector in the space.

Example 12.3.4:

If $\mathbf{a} = [34]^T$, then

$$\mathbf{a} = 3\begin{bmatrix}1\\0\end{bmatrix} + 4\begin{bmatrix}0\\1\end{bmatrix}$$

and

$$\mathbf{a} = 4\begin{bmatrix}1\\1\end{bmatrix} + (-1)\begin{bmatrix}1\\0\end{bmatrix}$$

so the representation of **a** with respect to the first basis set is (3, 4), and the representation of **a** with respect to the second basis set is (4, −1). ♦

In the beginning of this section we had described a mathematical machinery for finding the components of a vector that involved taking the dot product or inner product of the vector to be decomposed with basis vectors. In order to use the same machinery in more abstract vector spaces we need to generalize the notion of inner product.

12.3.5 Inner Product—Formal Definition

An inner product between two vectors **x** and **y**, denoted by $\langle \mathbf{x}, \mathbf{y}\rangle$, associates a scalar value with each pair of vectors. The inner product satisfies the following axioms:

1. $\langle \mathbf{x}, \mathbf{y}\rangle = \langle \mathbf{y}, \mathbf{x}\rangle^*$, where * denotes complex conjugate.
2. $\langle \mathbf{x}+\mathbf{y}, \mathbf{z}\rangle = \langle \mathbf{x}, \mathbf{z}\rangle + \langle \mathbf{y}, \mathbf{z}\rangle$.
3. $\langle \alpha\mathbf{x}, \mathbf{y}\rangle = \alpha\langle \mathbf{x}, \mathbf{y}\rangle$.
4. $\langle \mathbf{x}, \mathbf{x}\rangle \geq 0$, with equality if and only if $\mathbf{x} = \theta$. The quantity $\sqrt{\langle \mathbf{x}, \mathbf{x}\rangle}$ denoted by $\|\mathbf{x}\|$ is called the *norm* of **x** and is analogous to our usual concept of distance.

12.3.6 Orthogonal and Orthonormal Sets

As in the case of Euclidean space, two vectors are said to be *orthogonal* if their inner product is zero. If we select our basis set to be orthogonal (that is, each vector is orthogonal to every other vector in the set) and further require that the norm of each vector be one (that is, the basis vectors are unit vectors), such a basis set is called an *orthonormal basis set*. Given an orthonormal basis, it is easy to find the representation of any vector in the space in terms of the basis vectors using the inner product. Suppose we have a vector space S_N with an orthonormal basis set $\{\mathbf{x}_i\}_{i=1}^N$. Given a vector **y** in the space S_N, by definition of the basis set we can write **y** as a linear combination of the vectors $\mathbf{x}_i$:

$$\mathbf{y} = \sum_{i=1}^{N} \alpha_i \mathbf{x}_i.$$

To find the coefficient α_k, we find the inner product of both sides of this equation with $\mathbf{x}_k$:

$$\langle \mathbf{y}, \mathbf{x}_k \rangle = \sum_{i=1}^{N} \alpha_i \langle \mathbf{x}_i, \mathbf{x}_k \rangle.$$

Because of orthonormality,

$$\langle \mathbf{x}_i, \mathbf{x}_k \rangle = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$$

and

$$\langle \mathbf{y}, \mathbf{x}_k \rangle = \alpha_k.$$

By repeating this with each $\mathbf{x}_i$, we can get all the coefficients α_i. Note that in order to use this machinery, the basis set has to be orthonormal.

We now have sufficient information in hand to begin looking at some of the well-known techniques for representing functions of time. This was somewhat of a crash course in vector spaces, and you might, with some justification, be feeling somewhat dazed. Basically, the important ideas that we would like you to remember are the following:

- Vectors are not simply points in two- or three-dimensional space. In fact, functions of time can be viewed as elements in a vector space.
- Collections of vectors that satisfy certain axioms make up a vector space.
- All members of a vector space can be represented as linear, or weighted, combinations of the basis vectors (keep in mind that you can have many different basis sets for the same space). If the basis vectors have unit magnitude and are orthogonal, they are known as an *orthonormal basis set.*
- If a basis set is orthonormal, the weights, or coefficients, can be obtained by taking the inner product of the vector with the corresponding basis vector.

In the next section we use these concepts to show how we can represent periodic functions as linear combinations of sines and cosines.

12.4 Fourier Series

The representation of periodic functions in terms of a series of sines and cosines was discovered by Jean Baptiste Joseph Fourier. Although he came up with this idea in order to help him solve equations describing heat diffusion, this work has since become indispensable in the analysis and design of systems. The work was awarded the grand prize for mathematics in 1812 and has been called one of the most revolutionary contributions of the last century. A very readable account of the life of Fourier and the impact of his discovery can be found in [176].

Fourier showed that any periodic function, no matter how awkward looking, could be represented as the sum of smooth, well-behaved sines and cosines. Given a periodic function $f(t)$ with period T,

$$f(t) = f(t+nT) \qquad n = \pm 1, \pm 2, \ldots$$

we can write $f(t)$ as

$$f(t) = a_0 + \sum_{n=1}^{\infty} a_n \cos nw_0 t + \sum_{n=1}^{\infty} b_n \sin nw_0 t, \qquad w_0 = \frac{2\pi}{T}. \tag{12.4}$$

This form is called the *trigonometric Fourier series representation* of $f(t)$.

A more useful form of the Fourier series representation from our point of view is the exponential form of the Fourier series:

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{jnw_0 t}. \tag{12.5}$$

We can easily move between the exponential and trigonometric representations by using Euler's identity

$$e^{j\phi} = \cos\phi + j\sin\phi$$

where $j = \sqrt{-1}$.

In the terminology of the previous section, all periodic functions with period T form a vector space. The complex exponential functions $\{e^{jn\omega_0 t}\}$ constitute a basis for this space. The parameters $\{c_n\}_{n=-\infty}^{\infty}$ are the representations of a given function $f(t)$ with respect to this basis set. Therefore, by using different values of $\{c_n\}_{n=-\infty}^{\infty}$, we can build different periodic functions. If we wanted to inform somebody what a particular periodic function looked like, we could send the values of $\{c_n\}_{n=-\infty}^{\infty}$ and they could synthesize the function.

We would like to see if this basis set is orthonormal. If it is, we want to be able to obtain the coefficients that make up the Fourier representation using the approach described in the previous section. In order to do all this, we need a definition of the inner product on this vector space. If $f(t)$ and $g(t)$ are elements of this vector space, the inner product is defined as

$$\langle f(t), g(t)\rangle = \frac{1}{T}\int_{t_0}^{t_0+T} f(t)g(t)^* dt \tag{12.6}$$

where t_0 is an arbitrary constant and * denotes complex conjugate. For convenience we will take t_0 to be zero.

Using this inner product definition, let us check to see if the basis set is orthonormal.

$$\langle e^{jn\omega_0 t}, e^{jm\omega_0 t}\rangle = \frac{1}{T}\int_0^T e^{jn\omega_0 t} e^{-jm\omega_0 t} dt \tag{12.7}$$

$$= \frac{1}{T}\int_0^T e^{j(n-m)\omega_0 t} dt \tag{12.8}$$

When $n = m$, Equation (12.7) becomes the norm of the basis vector, which is clearly one. When $n \neq m$, let us define $k = n - m$. Then

$$\langle e^{jn\omega_0 t}, e^{jm\omega_0 t} \rangle = \frac{1}{T}\int_0^T e^{jk\omega_0 t} dt \tag{12.9}$$

$$= \frac{1}{jk\omega_0}(e^{jk\omega_0 T} - 1) \tag{12.10}$$

$$= \frac{1}{jk\omega_0}(e^{jk2\pi} - 1) \tag{12.11}$$

$$= 0 \tag{12.12}$$

where we have used the facts that $\omega_0 = \frac{2\pi}{T}$ and

$$e^{jk2\pi} = \cos(2k\pi) + j\sin(2k\pi) = 1.$$

Thus, the basis set is orthonormal.

Using this fact, we can find the coefficient c_n by taking the inner product of $f(t)$ with the basis vector $e^{jn\omega_0 t}$:

$$c_n = \langle f(t), e^{jn\omega_0 t} \rangle = \frac{1}{T}\int_0^T f(t) e^{jn\omega_0 t}\, dt. \tag{12.13}$$

What do we gain from obtaining the Fourier representation $\{c_n\}_{n=-\infty}^{\infty}$ of a function $f(t)$? Before we answer this question, let us examine the context in which we generally use Fourier analysis. We start with some signal generated by a source. If we wish to look at how this signal changes its amplitude over a period of time (or space), we represent it as a function of time $f(t)$ (or a function of space $f(x)$). Thus, $f(t)$ (or $f(x)$) is a representation of the signal that brings out how this signal varies in time (or space). The sequence $\{c_n\}_{n=-\infty}^{\infty}$ is a different representation of the same signal. However, this representation brings out a different aspect of the signal. The basis functions are sinusoids that differ from each other in how fast they fluctuate in a given time interval. The basis vector $e^{2j\omega_0 t}$ fluctuates twice as fast as the basis vector $e^{j\omega_0 t}$. The coefficients of the basis vectors $\{c_n\}_{n=-\infty}^{\infty}$ give us a measure of the different amounts of fluctuation present in the signal. Fluctuation of this sort is usually measured in terms of frequency. A frequency of 1 Hz denotes the completion of one period in one second, a frequency of 2 Hz denotes the completion of two cycles in one second, and so on. Thus, the coefficients $\{c_n\}_{n=-\infty}^{\infty}$ provide us with a frequency profile of the signal: how much of the signal changes at the rate of $\frac{\omega_0}{2\pi}$ Hz, how much of the signal changes at the rate of $\frac{2\omega_0}{2\pi}$ Hz, and so on. This information cannot be obtained by looking at the time representation $f(t)$. On the other hand, the use of the $\{c_n\}_{n=-\infty}^{\infty}$ representation tells us little about how the signal changes with time. Each representation emphasizes a different aspect of the signal. The ability to view the same signal in different ways helps us to better understand the nature of the signal, and thus develop tools for manipulation of the signal. Later, when we talk about wavelets, we will look at representations that provide information about both the time profile and the frequency profile of the signal.

The Fourier series provides us with a frequency representation of *periodic* signals. However, many of the signals we will be dealing with are not periodic. Fortunately, the Fourier series concepts can be extended to nonperiodic signals.

12.5 Fourier Transform

Consider the function $f(t)$ shown in Figure 12.3. Let us define a function $f_P(t)$ as

$$f_P(t) = \sum_{n=-\infty}^{\infty} f(t - nT) \tag{12.14}$$

where $T > t_1$. This function, which is obviously periodic ($f_P(t+T) = f_P(t)$), is called the *periodic extension* of the function $f(t)$. Because the function $f_P(t)$ is periodic, we can define a Fourier series expansion for it:

$$c_n = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f_P(t)e^{-jn\omega_0 t}dt \tag{12.15}$$

$$f_P(t) = \sum_{n=-\infty}^{\infty} c_n e^{jn\omega_0 t}. \tag{12.16}$$

Define

$$C(n, T) = c_n T$$

and

$$\Delta\omega = \omega_0,$$

and let us slightly rewrite the Fourier series equations:

$$C(n, T) = \int_{-\frac{T}{2}}^{\frac{T}{2}} f_P(t)e^{-jn\Delta\omega t}dt \tag{12.17}$$

$$f_P(t) = \sum_{n=-\infty}^{\infty} \frac{C(n, T)}{T} e^{jn\Delta\omega t}. \tag{12.18}$$

We can recover $f(t)$ from $f_P(t)$ by taking the limit of $f_P(t)$ as T goes to infinity. Because $\Delta\omega = \omega_0 = \frac{2\pi}{T}$, this is the same as taking the limit as $\Delta\omega$ goes to zero. As $\Delta\omega$ goes to zero, $n\Delta\omega$ goes to a continuous variable ω. Therefore,

$$\lim_{T\to\infty \Delta\omega\to 0} \int_{-\frac{T}{2}}^{\frac{T}{2}} f_P(t)e^{-jn\Delta\omega t}dt = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt. \tag{12.19}$$

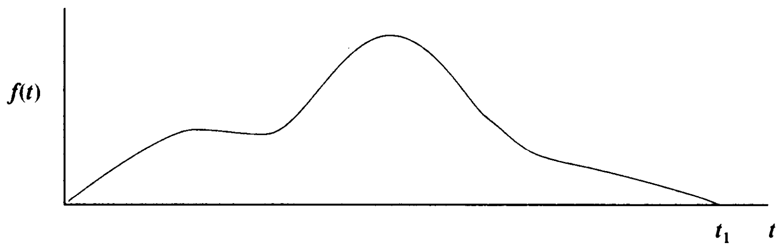

FIGURE 12.3 A function of time.

From the right-hand side, we can see that the resulting function is a function only of ω. We call this function the Fourier transform of $f(t)$, and we will denote it by $F(\omega)$. To recover $f(t)$ from $F(w)$, we apply the same limits to Equation (12.18):

$$f(t) \lim_{T\to\infty} f_P(t) = \lim_{T\to\infty\Delta\omega\to 0} \sum_{n=-\infty}^{\infty} C(n, T)\frac{\Delta\omega}{2\pi} e^{jn\Delta\omega t} \tag{12.20}$$

$$= \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{j\omega t}d\omega. \tag{12.21}$$

The equation

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt \tag{12.22}$$

is generally called the *Fourier transform*. The function $F(\omega)$ tells us how the signal fluctuates at different frequencies. The equation

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(w)e^{j\omega t}d\omega \tag{12.23}$$

is called the *inverse Fourier transform*, and it shows us how we can construct a signal using components that fluctuate at different frequencies. We will denote the operation of the Fourier transform by the symbol $\mathcal{F}$. Thus, in the preceding, $F(\omega) = \mathcal{F}[f(t)]$.

There are several important properties of the Fourier transform, three of which will be of particular use to us. We state them here and leave the proof to the problems (Problems 2, 3, and 4).

12.5.1 Parseval's Theorem

The Fourier transform is an energy-preserving transform; that is, the total energy when we look at the time representation of the signal is the same as the total energy when we look at the frequency representation of the signal. This makes sense because the total energy is a physical property of the signal and should not change when we look at it using different representations. Mathematically, this is stated as

$$\int_{-\infty}^{\infty} |f(t)|^2 = \frac{1}{2\pi}\int_{-\infty}^{\infty} |F(\omega)|^2 d\omega. \tag{12.24}$$

The $\frac{1}{2\pi}$ factor is a result of using units of radians (ω) for frequency instead of Hertz (f). If we substitute $\omega = 2\pi f$ in Equation (12.24), the 2π factor will go away. This property applies to any vector space representation obtained using an orthonormal basis set.

12.5.2 Modulation Property

If $f(t)$ has the Fourier transform $F(\omega)$, then the Fourier transform of $f(t)e^{j\omega_0 t}$ is $F(w - w_0)$. That is, multiplication with a complex exponential in the time domain corresponds to a shift in the frequency domain. As a sinusoid can be written as a sum of complex exponentials, multiplication of $f(t)$ by a sinusoid will also correspond to shifts of $F(\omega)$. For example,

$$\cos(\omega_0 t) = \frac{e^{j\omega_0 t} + e^{-j\omega_0 t}}{2}.$$

Therefore,

$$\mathcal{F}[f(t)\cos(\omega_0 t)] = \frac{1}{2}\left(F(\omega - \omega_0) + F(\omega + \omega_0)\right).$$

12.5.3 Convolution Theorem

When we examine the relationships between the input and output of linear systems, we will encounter integrals of the following forms:

$$f(t) = \int_{-\infty}^{\infty} f_1(\tau) f_2(t-\tau)\, d\tau$$

or

$$f(t) = \int_{-\infty}^{\infty} f_1(t-\tau) f_2(\tau)\, d\tau.$$

These are called convolution integrals. The convolution operation is often denoted as

$$f(t) = f_1(t) \otimes f_2(t).$$

The convolution theorem states that if $F(\omega) = \mathcal{F}[f(t)] = \mathcal{F}[f_1(t) \otimes f_2(t)]$, $F_1(\omega) = \mathcal{F}[f_1(t)]$, and $F_2(\omega) = \mathcal{F}[f_2(t)]$, then

$$F(\omega) = F_1(\omega) F_2(\omega).$$

We can also go in the other direction. If

$$F(\omega) = F_1(\omega) \otimes F_2(\omega) = \int F_1(\sigma) F_2(\omega - \sigma)\, d\sigma$$

then

$$f(t) = f_1(t) f_2(t).$$

As mentioned earlier, this property of the Fourier transform is important because the convolution integral relates the input and output of linear systems, which brings us to one of the major reasons for the popularity of the Fourier transform. We have claimed that the Fourier series and Fourier transform provide us with an alternative frequency profile of a signal. Although sinusoids are not the only basis set that can provide us with a frequency profile, they do, however, have an important property that helps us study linear systems, which we describe in the next section.

12.6 Linear Systems

A linear system is a system that has the following two properties:

- **Homogeneity:** Suppose we have a linear system L with input $f(t)$ and output $g(t)$:

$$g(t) = L[f(t)].$$

  If we have two inputs, $f_1(t)$ and $f_2(t)$, with corresponding outputs, $g_1(t)$ and $g_2(t)$, then the output of the sum of the two inputs is simply the sum of the two outputs:

$$L[f_1(t) + f_2(t)] = g_1(t) + g_2(t).$$

- **Scaling:** Given a linear system L with input $f(t)$ and output $g(t)$, if we multiply the input with a scalar α, then the output will be multiplied by the same scalar:

$$L[\alpha f(t)] = \alpha L[f(t)] = \alpha g(t).$$

The two properties together are referred to as *superposition.*

12.6.1 Time Invariance

Of specific interest to us are linear systems that are *time invariant.* A time-invariant system has the property that the shape of the response of this system does not depend on the time at which the input was applied. If the response of a linear system L to an input $f(t)$ is $g(t)$,

$$L[f(t)] = g(t),$$

and we delay the input by some interval t_0, then if L is a time-invariant system, the output will be $g(t)$ delayed by the same amount:

$$L[f(t - t_0)] = g(t - t_0). \tag{12.25}$$

12.6.2 Transfer Function

Linear time-invariant systems have a very interesting (and useful) response when the input is a sinusoid. If the input to a linear system is a sinusoid of a certain frequency ω_0, then the output is also a sinusoid of the same frequency that has been scaled and delayed; that is,

$$L[\cos(\omega_0 t)] = \alpha \cos(\omega_0(t - t_d))$$

or in terms of the complex exponential

$$L[e^{j\omega_0 t}] = \alpha e^{j\omega_0(t-t_d)}.$$

Thus given a linear system, we can characterize its response to sinusoids of a particular frequency by a pair of parameters, the gain α and the delay t_d. In general, we use the phase $\phi = \omega_0 t_d$ in place of the delay. The parameters α and ϕ will generally be a function of the frequency, so in order to characterize the system for all frequencies, we will need a pair of functions $\alpha(\omega)$ and $\phi(\omega)$. As the Fourier transform allows us to express the signal as coefficients of sinusoids, given an input $f(t)$, all we need to do is, for each frequency ω, multiply the Fourier transform of $f(t)$ with some $\alpha(\omega)e^{j\phi(\omega)}$, where $\alpha(\omega)$ and $\phi(\omega)$ are the gain and phase terms of the linear system for that particular frequency.

This pair of functions $\alpha(\omega)$ and $\phi(\omega)$ constitute the *transfer function* of the linear time-invariant system $H(\omega)$:

$$H(\omega) = |H(\omega)|\, e^{j\phi(\omega)}$$

where $|H(\omega)| = \alpha(\omega)$.

Because of the specific way in which a linear system responds to a sinusoidal input, given a linear system with transfer function $H(\omega)$, input $f(t)$, and output $g(t)$, the Fourier transforms of the input and output $F(\omega)$ and $G(\omega)$ are related by

$$G(w) = H(\omega)F(\omega).$$

Using the convolution theorem, $f(t)$ and $g(t)$ are related by

$$g(t) = \int_{-\infty}^{\infty} f(\tau)h(t-\tau)d\tau$$

or

$$g(t) = \int_{-\infty}^{\infty} f(t-\tau)h(\tau)d\tau$$

where $H(\omega)$ is the Fourier transform of $h(t)$.

12.6.3 Impulse Response

To see what $h(t)$ is, let us look at the input-output relationship of a linear time-invariant system from a different point of view. Let us suppose we have a linear system L with input $f(t)$. We can obtain a staircase approximation $f_S(t)$ to the function $f(t)$, as shown in Figure 12.4:

$$f_S(t) = \sum f(n\Delta t)\mathrm{rect}\left(\frac{t-n\Delta t}{\Delta t}\right) \tag{12.26}$$

where

$$\mathrm{rect}\left(\frac{t}{T}\right) = \begin{cases} 1 & |t| < \frac{T}{2} \\ 0 & \text{otherwise.} \end{cases} \tag{12.27}$$

The response of the linear system can be written as

$$L[f_S(t)] = L\left[\sum f(n\Delta t)\mathrm{rect}\left(\frac{t-n\Delta t}{\Delta t}\right)\right] \tag{12.28}$$

$$= L\left[\sum f(n\Delta t)\frac{\mathrm{rect}\left(\frac{t-n\Delta t}{\Delta t}\right)}{\Delta t}\Delta t\right]. \tag{12.29}$$

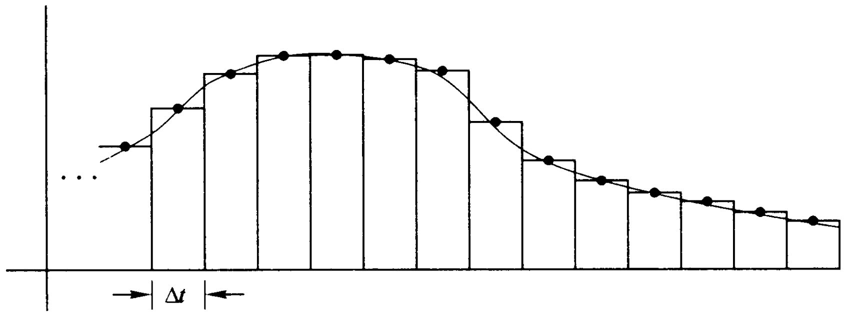

FIGURE 12.4 **A function of time.**

For a given value of Δt, we can use the superposition property of linear systems to obtain

$$L[f_S(t)] = \sum f(n\Delta t) L\left[\frac{\text{rect}(\frac{t-n\Delta t}{\Delta t})}{\Delta t}\right]\Delta t. \tag{12.30}$$

If we now take the limit as Δt goes to zero in this equation, on the left-hand side $f_S(t)$ will go to $f(t)$. To see what happens on the right-hand side of the equation, first let's look at the effect of this limit on the function $\text{rect}(\frac{t}{\Delta t})/\Delta t$. As Δt goes to zero, this function becomes narrower and taller. However, at all times the integral of this function is equal to one. The limit of this function as Δt goes to zero is called the *Dirac delta function*, or *impulse function*, and is denoted by $\delta(t)$:

$$\lim_{\Delta t \to 0} \frac{\text{rect}(\frac{t-n\Delta t}{\Delta t})}{\Delta t} = \delta(t). \tag{12.31}$$

Therefore,

$$L[f(t)] = \lim_{\Delta t \to 0} L[f_S(t)] = \int f(\tau) L[\delta(t-\tau)]d\tau. \tag{12.32}$$

Denote the response of the system L to an impulse, or the *impulse response*, by $h(t)$:

$$h(t) = L[\delta(t)]. \tag{12.33}$$

Then, if the system is also time invariant,

$$L[f(t)] = \int f(\tau)h(t-\tau)d\tau. \tag{12.34}$$

Using the convolution theorem, we can see that the Fourier transform of the impulse response $h(t)$ is the transfer function $H(\omega)$.

The Dirac delta function is an interesting function. In fact, it is not clear that it is a function at all. It has an integral that is clearly one, but at the only point where it is not zero, it is undefined! One property of the delta function that makes it very useful is the *sifting* property:

$$\int_{t_1}^{t_2} f(t)\delta(t - t_0)dt = \begin{cases} f(t_0) & t_1 \leq t_0 \leq t_2 \\ 0 & \text{otherwise.} \end{cases} \tag{12.35}$$

12.6.4 Filter

The linear systems of most interest to us will be systems that permit certain frequency components of the signal to pass through, while attenuating all other components of the signal. Such systems are called *filters*. If the filter allows only frequency components below a certain frequency W Hz to pass through, the filter is called a *low-pass filter*. The transfer function of an ideal low-pass filter is given by

$$H(\omega) = \begin{cases} e^{-j\alpha\omega} & |\omega| < 2\pi W \\ 0 & \text{otherwise.} \end{cases} \tag{12.36}$$

This filter is said to have a *bandwidth* of W Hz. The magnitude of this filter is shown in Figure 12.5. A low-pass filter will produce a smoothed version of the signal by blocking higher-frequency components that correspond to fast variations in the signal.

A filter that attenuates the frequency components below a certain frequency W and allows the frequency components above this frequency to pass through is called a *high-pass filter*. A high-pass filter will remove slowly changing trends from the signal. Finally, a signal that lets through a range of frequencies between two specified frequencies, say, W_1 and W_2, is called a *band-pass filter*. The bandwidth of this filter is said to be $W_2 - W_1$ Hz. The magnitude of the transfer functions of an ideal high-pass filter and an ideal band-pass filter with bandwidth W are shown in Figure 12.6. In all the ideal filter characteristics, there is a sharp transition between the *passband* of the filter (the range of frequencies that are not attenuated) and the *stopband* of the filter (those frequency intervals where the signal is completely attenuated). Real filters do not have such sharp transitions, or *cutoffs*.

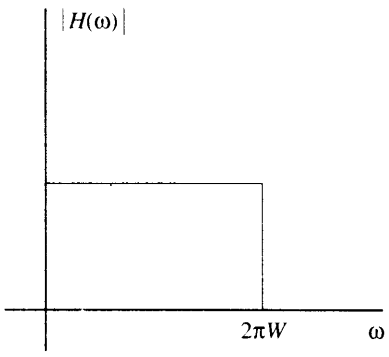

FIGURE 12.5 **Magnitude of the transfer function of an ideal low-pass filter.**

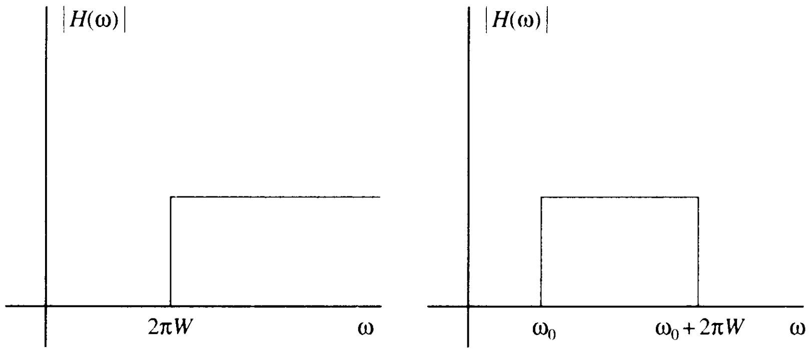

FIGURE 12.6 **Magnitudes of the transfer functions of ideal high-pass (left) and ideal band-pass (right) filters.**

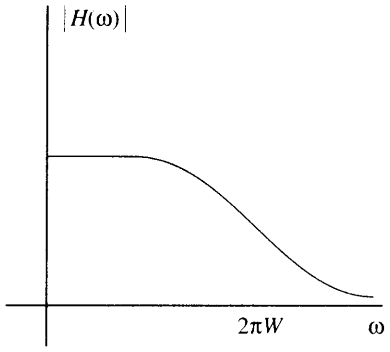

FIGURE 12.7 **Magnitude of the transfer functions of a realistic low-pass filter.**

The magnitude characteristics of a more realistic low-pass filter are shown in Figure 12.7. Notice the more gentle rolloff. But when the cutoff between stopband and passband is not sharp, how do we define the bandwidth? There are several different ways of defining the bandwidth. The most common way is to define the frequency at which the magnitude of the transfer function is $1/\sqrt{2}$ of its maximum value (or the magnitude squared is 1/2 of its maximum value) as the cutoff frequency.

12.7 Sampling

In 1928 Harry Nyquist at Bell Laboratories showed that if we have a signal whose Fourier transform is zero above some frequency W Hz, it can be accurately represented using $2W$ equally spaced samples per second. This very important result, known as the *sampling theorem*, is at the heart of our ability to transmit analog waveforms such as speech and video using digital means. There are several ways to prove this result. We will use the results presented in the previous section to do so.

12.7.1 Ideal Sampling—Frequency Domain View

Let us suppose we have a function $f(t)$ with Fourier transform $F(\omega)$, shown in Figure 12.8, which is zero for ω greater than $2\pi W$. Define the periodic extension of $F(\omega)$ as

$$F_P(\omega) = \sum_{n=-\infty}^{\infty} F(\omega - n\sigma_0), \qquad \sigma_0 = 4\pi W. \tag{12.37}$$

The periodic extension is shown in Figure 12.9. As $F_P(\omega)$ is periodic, we can express it in terms of a Fourier series expansion:

$$F_P(\omega) = \sum_{n=-\infty}^{\infty} c_n e^{jn\frac{1}{2W}\omega}. \tag{12.38}$$

The coefficients of the expansion $\{c_n\}_{n=-\infty}^{\infty}$ are then given by

$$c_n = \frac{1}{4\pi W}\int_{-2\pi W}^{2\pi W} F_P(\omega) e^{-jn\frac{1}{2W}\omega} d\omega. \tag{12.39}$$

However, in the interval $(-2\pi W, 2\pi W)$, $F(\omega)$ is identical to $F_P(\omega)$; therefore,

$$c_n = \frac{1}{4\pi W}\int_{-2\pi W}^{2\pi W} F(\omega) e^{-jn\frac{1}{2W}\omega} d\omega. \tag{12.40}$$

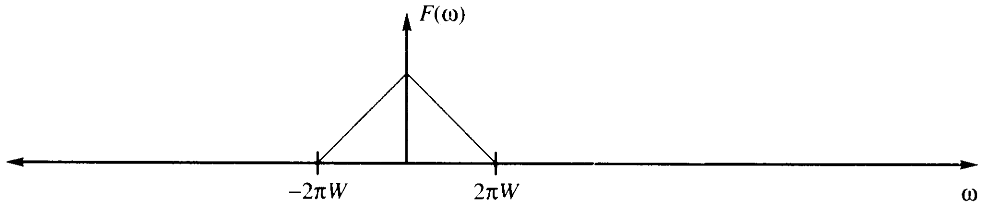

FIGURE 12.8 **A function $F(\omega)$.**

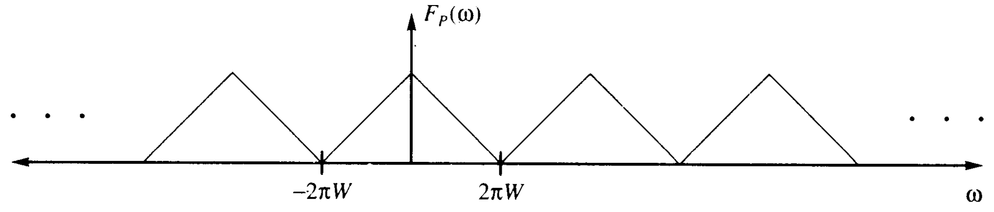

FIGURE 12.9 **The periodic extension $F_P(\omega)$.**

The function $F(\omega)$ is zero outside the interval $(-2\pi W, 2\pi W)$, so we can extend the limits to infinity without changing the result:

$$c_n = \frac{1}{2W}\left[\frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{-jn\frac{1}{2W}\omega}d\omega\right]. \tag{12.41}$$

The expression in brackets is simply the inverse Fourier transform evaluated at $t = \frac{n}{2W}$; therefore,

$$c_n = \frac{1}{2W}f\left(\frac{n}{2W}\right). \tag{12.42}$$

Knowing $\{c_n\}_{n=-\infty}^{\infty}$ and the value of W, we can reconstruct $F_P(\omega)$. Because $F_P(\omega)$ and $F(\omega)$ are identical in the interval $(-2\pi W, 2\pi W)$, therefore knowing $\{c_n\}_{n=-\infty}^{\infty}$, we can also reconstruct $F(\omega)$ in this interval. But $\{c_n\}_{n=-\infty}^{\infty}$ are simply the samples of $f(t)$ every $\frac{1}{2W}$ seconds, and $F(\omega)$ is zero outside this interval. Therefore, given the samples of a function $f(t)$ obtained at a rate of $2W$ samples per second, we should be able to exactly reconstruct the function $f(t)$.

Let us see how we can do this:

$$\begin{aligned}
f(t) &= \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{-j\omega t}d\omega & (12.43)\\
&= \frac{1}{2\pi}\int_{-2\pi W}^{2\pi W} F(\omega)e^{-j\omega t}d\omega & (12.44)\\
&= \frac{1}{2\pi}\int_{-2\pi W}^{2\pi W} F_P(\omega)e^{-j\omega t}d\omega & (12.45)\\
&= \frac{1}{2\pi}\int_{-2\pi W}^{2\pi W} \sum_{n=-\infty}^{\infty} c_n e^{jn\frac{1}{2W}\omega}e^{-j\omega t}d\omega & (12.46)\\
&= \frac{1}{2\pi}\sum_{n=-\infty}^{\infty} c_n \int_{-2\pi W}^{2\pi W} e^{j\omega(t-\frac{n}{2W})}d\omega. & (12.47)
\end{aligned}$$

Evaluating the integral and substituting for c_n from Equation (12.42), we obtain

$$f(t) = \sum_{n=-\infty}^{\infty} f\left(\frac{n}{2W}\right)\text{Sinc}\left[2W\left(t-\frac{n}{2W}\right)\right] \tag{12.48}$$

where

$$\text{Sinc}[x] = \frac{\sin(\pi x)}{\pi x}. \tag{12.49}$$

Thus, given samples of $f(t)$ taken every $\frac{1}{2W}$ seconds, or, in other words, samples of $f(t)$ obtained at a rate of $2W$ samples per second, we can reconstruct $f(t)$ by interpolating between the samples using the Sinc function.

12.7.2 Ideal Sampling—Time Domain View

Let us look at this process from a slightly different point of view, starting with the sampling operation. Mathematically, we can represent the sampling operation by multiplying the function $f(t)$ with a train of impulses to obtain the sampled function $f_S(t)$:

$$f_S(t) = f(t) \sum_{n=-\infty}^{\infty} \delta(t-nT), \qquad T < \frac{1}{2W}. \tag{12.50}$$

To obtain the Fourier transform of the sampled function, we use the convolution theorem:

$$\mathcal{F}\left[f(t) \sum_{n=-\infty}^{\infty} \delta(t-nT)\right] = \mathcal{F}[f(t)] \otimes \mathcal{F}\left[\sum_{n=-\infty}^{\infty} \delta(t-nT)\right]. \tag{12.51}$$

Let us denote the Fourier transform of $f(t)$ by $F(\omega)$. The Fourier transform of a train of impulses in the time domain is a train of impulses in the frequency domain (Problem 5):

$$\mathcal{F}\left[\sum_{n=-\infty}^{\infty} \delta(t-nT)\right] = \sigma_0 \sum_{n=-\infty}^{\infty} \delta(w-n\sigma_0) \qquad \sigma_0 = \frac{2\pi}{T}. \tag{12.52}$$

Thus, the Fourier transform of $f_S(t)$ is

$$F_S(\omega) = F(\omega) \otimes \sum_{n=-\infty}^{\infty} \delta(w-n\sigma_0) \tag{12.53}$$

$$= \sum_{n=-\infty}^{\infty} F(\omega) \otimes \delta(w-n\sigma_0) \tag{12.54}$$

$$= \sum_{n=-\infty}^{\infty} F(\omega-n\sigma_0) \tag{12.55}$$

where the last equality is due to the sifting property of the delta function.

Pictorially, for $F(\omega)$ as shown in Figure 12.8, $F_S(\omega)$ is shown in Figure 12.10. Note that if T is less than $\frac{1}{2W}$, σ_0 is greater than $4\pi W$, and as long as σ_0 is greater than $4\pi W$, we can recover $F(\omega)$ by passing $F_S(\omega)$ through an ideal low-pass filter with bandwidth W Hz ($2\pi W$ radians).

What happens if we do sample at a rate less than $2W$ samples per second (that is, σ_0 is less than $4\pi W$)? Again we can see the results most easily in a pictorial fashion. The result

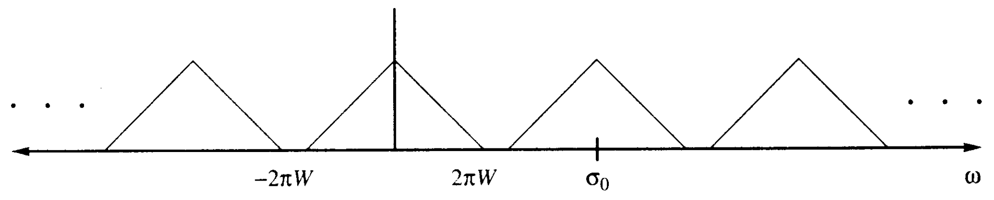

FIGURE 12.10 Fourier transform of the sampled function.

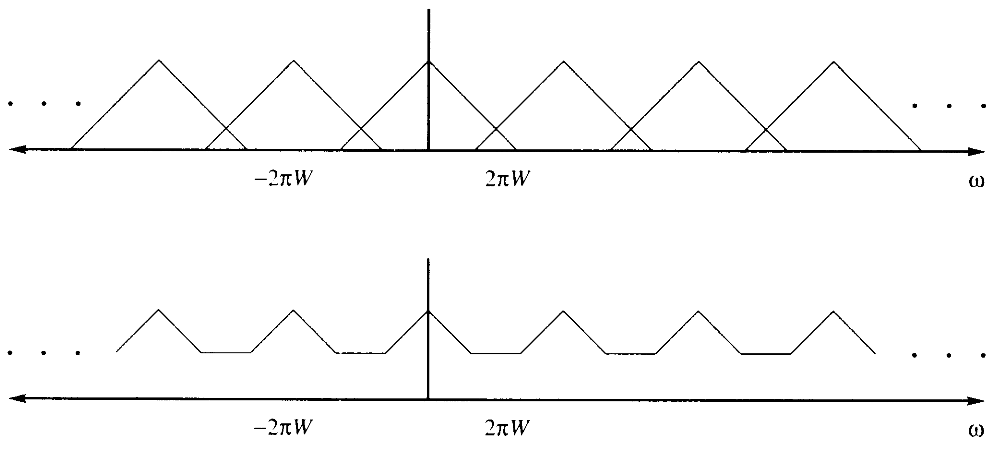

FIGURE 12.11 **Effect of sampling at a rate less than 2W samples per second.**

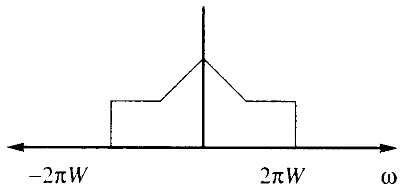

FIGURE 12.12 **Aliased reconstruction.**

for σ_0 equal to $3\pi W$ is shown in Figure 12.11. Filtering this signal through an ideal low-pass filter, we get the distorted signal shown in Figure 12.12. Therefore, if σ_0 is less than $4\pi W$, we can no longer recover the signal $f(t)$ from its samples. This distortion is known as *aliasing*. In order to prevent aliasing, it is useful to filter the signal prior to sampling using a low-pass filter with a bandwidth less than half the sampling frequency.

Once we have the samples of a signal, sometimes the actual times they were sampled at are not important. In these situations we can normalize the sampling frequency to unity. This means that the highest frequency component in the signal is at 0.5 Hz, or π radians. Thus, when dealing with sampled signals, we will often talk about frequency ranges of $-\pi$ to π.

12.8 Discrete Fourier Transform

The procedures that we gave for obtaining the Fourier series and transform were based on the assumption that the signal we were examining could be represented as a continuous function of time. However, for the applications that we will be interested in, we will primarily be dealing with samples of a signal. To obtain the Fourier transform of nonperiodic signals, we started from the Fourier series and modified it to take into account the nonperiodic nature of the signal. To obtain the discrete Fourier transform (DFT), we again start from the Fourier series. We begin with the Fourier series representation of a sampled function, the discrete Fourier series.

Recall that the Fourier series coefficients of a periodic function $f(t)$ with period T is given by

$$c_k = \frac{1}{T}\int_0^T f(t)e^{jkw_0t}dt. \tag{12.56}$$

Suppose instead of a continuous function, we have a function sampled N times during each period T. We can obtain the coefficients of the Fourier series representation of this sampled function as

$$F_k = \frac{1}{T}\int_0^T f(t)\sum_{n=0}^{N-1}\delta\left(t-\frac{n}{N}T\right)e^{jkw_0t}dt \tag{12.57}$$

$$= \frac{1}{T}\sum_{n=0}^{N-1} f\left(\frac{n}{N}T\right)e^{j\frac{2\pi kn}{N}} \tag{12.58}$$

where we have used the fact that $w_0 = \frac{2\pi}{T}$, and we have replaced c_k by F_k. Taking $T = 1$ for convenience and defining

$$f_n = f\left(\frac{n}{N}\right),$$

we get the coefficients for the discrete Fourier series (DFS) representation:

$$F_k = \sum_{n=0}^{N-1} f_n e^{j\frac{2\pi kn}{N}}. \tag{12.59}$$

Notice that the sequence of coefficients $\{F_k\}$ is periodic with period N.

The Fourier series representation was given by

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{jn\omega_0 t}. \tag{12.60}$$

Evaluating this for $t = \frac{n}{N}T$, we get

$$f_n = f(\frac{n}{N}T) = \sum_{k=-\infty}^{\infty} c_k e^{j\frac{2\pi kn}{N}}. \tag{12.61}$$

Let us write this in a slightly different form:

$$f_n = \sum_{k=0}^{N-1}\sum_{l=-\infty}^{\infty} c_{k+lN}e^{j\frac{2\pi n(k+lN)}{N}} \tag{12.62}$$

but

$$e^{j\frac{2\pi n(k+lN)}{N}} = e^{j\frac{2\pi kn}{N}}e^{j2\pi nl} \tag{12.63}$$

$$= e^{j\frac{2\pi kn}{N}}. \tag{12.64}$$

Therefore,

$$f_n = \sum_{k=0}^{N-1} e^{j\frac{2\pi kn}{N}} \sum_{l=-\infty}^{\infty} c_{k+lN}. \tag{12.65}$$

Define

$$\bar{c}_k = \sum_{l=-\infty}^{\infty} c_{k+lN}. \tag{12.66}$$

Clearly, $\bar{c}_k$ is periodic with period N. In fact, we can show that $\bar{c}_k = \frac{1}{N}F_k$ and

$$f_n = \frac{1}{N}\sum_{k=0}^{N-1} F_k e^{j\frac{2\pi kn}{N}}. \tag{12.67}$$

Obtaining the discrete Fourier transform from the discrete Fourier series is simply a matter of interpretation. We are generally interested in the discrete Fourier transform of a finite-length sequence. If we assume that the finite-length sequence is one period of a periodic sequence, then we can use the DFS equations to represent this sequence. The only difference is that the expressions are only valid for one "period" of the "periodic" sequence.

The DFT is an especially powerful tool because of the existence of a fast algorithm, called appropriately the *fast Fourier transform* (FFT), that can be used to compute it.

12.9 Z-Transform

In the previous section we saw how to extend the Fourier series to use with sampled functions. We can also do the same with the Fourier transform. Recall that the Fourier transform was given by the equation

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt. \tag{12.68}$$

Replacing $f(t)$ with its sampled version, we get

$$F(\omega) = \int_{-\infty}^{\infty} f(t) \sum_{n=-\infty}^{\infty} \delta(t-nT)e^{-j\omega t}dt \tag{12.69}$$

$$= \sum_{n=-\infty}^{\infty} f_n e^{-j\omega nT} \tag{12.70}$$

where $f_n = f(nT)$. This is called the discrete time Fourier transform. The Z-transform of the sequence $\{f_n\}$ is a generalization of the discrete time Fourier transform and is given by

$$F(z) = \sum_{n=-\infty}^{\infty} f_n z^{-n} \tag{12.71}$$

where

$$z = e^{\sigma T + j\omega T}. \tag{12.72}$$

Notice that if we let σ equal zero, we get the original expression for the Fourier transform of a discrete time sequence. We denote the Z-transform of a sequence by

$$F(z) = \mathcal{Z}[f_n].$$

We can express this another way. Notice that the magnitude of z is given by

$$|z| = e^{\sigma T}.$$

Thus, when σ equals zero, the magnitude of z is one. Because z is a complex number, the magnitude of z is equal to one on the unit circle in the complex plane. Therefore, we can say that the Fourier transform of a sequence can be obtained by evaluating the Z-transform of the sequence on the unit circle. Notice that the Fourier transform thus obtained will be periodic, which we expect because we are dealing with a sampled function. Further, if we assume T to be one, ω varies from $-\pi$ to π, which corresponds to a frequency range of -0.5 to 0.5 Hz. This makes sense because, by the sampling theorem, if the sampling rate is one sample per second, the highest frequency component that can be recovered is 0.5 Hz.

For the Z-transform to exist—in other words, for the power series to converge—we need to have

$$\sum_{n=-\infty}^{\infty} |f_n z^{-n}| < \infty.$$

Whether this inequality holds will depend on the sequence itself and the value of z. The values of z for which the series converges are called the *region of convergence* of the Z-transform. From our earlier discussion, we can see that for the Fourier transform of the sequence to exist, the region of convergence should include the unit circle. Let us look at a simple example.

Example 12.9.1:

Given the sequence

$$f_n = a^n u[n]$$

where $u[n]$ is the unit step function

$$u[n] = \begin{cases} 1 & n \geq 0 \\ 0 & n < 0 \end{cases} \tag{12.73}$$

the Z-transform is given by

$$F(z) = \sum_{n=0}^{\infty} a^n z^{-n} \tag{12.74}$$

$$= \sum_{n=0}^{\infty} (az^{-1})^n. \tag{12.75}$$

This is simply the sum of a geometric series. As we confront this kind of sum quite often, let us briefly digress and obtain the formula for the sum of a geometric series.

Suppose we have a sum

$$S_{mn} = \sum_{k=m}^{n} x^k = x^m + x^{m+1} + \cdots + x^n \tag{12.76}$$

then

$$xS_{mn} = x^{m+1} + x^{m+2} + \cdots + x^{n+1}. \tag{12.77}$$

Subtracting Equation (12.77) from Equation (12.76), we get

$$(1-x)S_{mn} = x^m - x^{n+1}$$

and

$$S_{mn} = \frac{x^m - x^{n+1}}{1-x}.$$

If the upper limit of the sum is infinity, we take the limit as n goes to infinity. This limit exists only when $|x| < 1$.

Using this formula, we get the Z-transform of the $\{f_n\}$ sequence as

$$F(z) = \frac{1}{1-az^{-1}}, \qquad |az^{-1}| < 1 \tag{12.78}$$

$$= \frac{z}{z-a}, \qquad |z| > |a|. \tag{12.79}$$

♦

In this example the region of convergence is the region $|z| > a$. For the Fourier transform to exist, we need to include the unit circle in the region of convergence. In order for this to happen, a has to be less than one.

Using this example, we can get some other Z-transforms that will be useful to us.

Example 12.9.2:

In the previous example we found that

$$\sum_{n=0}^{\infty} a^n z^{-n} = \frac{z}{z-a}, \qquad |z| > |a|. \tag{12.80}$$

If we take the derivative of both sides of the equation with respect to a, we get

$$\sum_{n=0}^{\infty} na^{n-1} z^{-n} = \frac{z}{(z-a)^2}, \qquad |z| > |a|. \tag{12.81}$$

Thus,

$$\mathcal{Z}[na^{n-1}u[n]] = \frac{z}{(z-a)^2}, \qquad |z| > |a|.$$

If we differentiate Equation (12.80) m times, we get

$$\sum_{n=0}^{\infty} n(n-1)\cdots(n-m+1)a^{n-m} = \frac{m!z}{(z-a)^{m+1}}.$$

In other words,

$$\mathcal{Z}\left[\binom{n}{m}a^{n-m}u[n]\right] = \frac{z}{(z-a)^{m+1}}. \tag{12.82}$$

♦

In these examples the Z-transform is a ratio of polynomials in z. For sequences of interest to us, this will generally be the case, and the Z-transform will be of the form

$$F(z) = \frac{N(z)}{D(z)}.$$

The values of z for which $F(z)$ is zero are called the *zeros* of $F(z)$; the values for which $F(z)$ is infinity are called the *poles* of $F(z)$. For finite values of z, the poles will occur at the roots of the polynomial $D(z)$.

The inverse Z-transform is formally given by the contour integral

$$\frac{1}{2\pi j}\oint_C F(z)z^{n-1}dz$$

where the integral is over the counterclockwise contour C, and C lies in the region of convergence. This integral can be difficult to evaluate directly; therefore, in most cases we use alternative methods for finding the inverse Z-transform.

12.9.1 Tabular Method

The inverse Z-transform has been tabulated for a number of interesting cases (see Table 12.1). If we can write $F(z)$ as a sum of these functions

$$F(z) = \sum \alpha_i F_i(z)$$

TABLE 12.1 Some Z-transform pairs.

$\{f_n\}$	$F(z)$
$a^n u[n]$	$\frac{z}{z-a}$
$nTu[n]$.	$\frac{Tz^{-1}}{(1-z^{-1})^2}$
$\sin(\alpha nT)$	$\frac{(\sin \alpha nT)z^{-1}}{1-2\cos(\alpha T)z^{-1}+z^{-2}}$
$\cos(\alpha nT)$	$\frac{(\cos \alpha nT)z^{-1}}{1-2\cos(\alpha T)z^{-1}+z^{-2}}$

then the inverse Z-transform is given by

$$f_n = \sum \alpha_i f_{i,n}$$

where $F_i(z) = \mathcal{Z}[\{f_{i,n}\}]$.

Example 12.9.3:

$$F(z) = \frac{z}{z-0.5} + \frac{2z}{z-0.3}$$

From our earlier example we know the inverse Z-transform of $z/(z-a)$. Using that, the inverse Z-transform of $F(z)$ is

$$f_n = 0.5^n u[n] + 2(0.3)^n u[n].$$

♦

12.9.2 Partial Fraction Expansion

In order to use the tabular method, we need to be able to decompose the function of interest to us as a sum of simpler terms. The partial fraction expansion approach does exactly that when the function is a ratio of polynomials in z.

Suppose $F(z)$ can be written as a ratio of polynomials $N(z)$ and $D(z)$. For the moment let us assume that the degree of $D(z)$ is greater than the degree of $N(z)$, and that all the roots of $D(z)$ are distinct (distinct roots are referred to as simple roots); that is,

$$F(z) = \frac{N(z)}{(z-z_1)(z-z_2)\cdots(z-z_L)}. \tag{12.83}$$

Then we can write $F(z)/z$ as

$$\frac{F(z)}{z} = \sum_{i=1}^{L} \frac{A_i}{z-z_i}. \tag{12.84}$$

If we can find the coefficients A_i, then we can write $F(z)$ as

$$F(z) = \sum_{i=1}^{L} \frac{A_i z}{z-z_i}$$

and the inverse Z-transform will be given by

$$f_n = \sum_{i=1}^{L} A_i z_i^n u[n].$$

The question then becomes one of finding the value of the coefficients A_i. This can be simply done as follows: Suppose we want to find the coefficient A_k. Multiply both sides of Equation (12.84) by $(z - z_k)$. Simplifying this we obtain

$$\frac{F(z)(z-z_k)}{z} = \sum_{i=1}^{L} \frac{A_i(z-z_k)}{z-z_i} \tag{12.85}$$

$$= A_k + \sum_{\substack{i=1 \\ i \neq k}}^{L} \frac{A_i(z-z_k)}{z-z_i}. \tag{12.86}$$

Evaluating this equation at $z = z_k$, all the terms in the summation go to zero and

$$A_k = \left.\frac{F(z)(z-z_k)}{z}\right|_{z=z_k} \tag{12.87}$$

Example 12.9.4:

Let us use the partial fraction expansion method to find the inverse Z-transform of

$$F(z) = \frac{6z^2 - 9z}{z^2 - 2.5z + 1}.$$

Then

$$\frac{F(z)}{z} = \frac{1}{z}\frac{6z^2 - 9z}{z^2 - 2.5z + 1} \tag{12.88}$$

$$= \frac{6z - 9}{(z-0.5)(z-2)}. \tag{12.89}$$

We want to write $F(z)/z$ in the form

$$\frac{F(z)}{z} = \frac{A_1}{z-0.5} + \frac{A_2}{z-2}.$$

Using the approach described above, we obtain

$$A_1 = \left.\frac{(6z-9)(z-0.5)}{(z-0.5)(z-2)}\right|_{z=0.5} \tag{12.90}$$

$$= 4 \tag{12.91}$$

$$A_2 = \left.\frac{(6z-9)(z-2)}{(z-0.5)(z-2)}\right|_{z=2} \tag{12.92}$$

$$= 2. \tag{12.93}$$

Therefore,

$$F(z) = \frac{4z}{z-0.5} + \frac{2z}{z-2}$$

and

$$f_n = [4(0.5)^n + 2(2)^n]u[n]. \qquad \blacklozenge$$

The procedure becomes slightly more complicated when we have repeated roots of $D(z)$. Suppose we have a function

$$F(z) = \frac{N(z)}{(z-z_1)(z-z_2)^2}.$$

The partial fraction expansion of this function is

$$\frac{F(z)}{z} = \frac{A_1}{z-z_1} + \frac{A_2}{z-z_2} + \frac{A_3}{(z-z_2)^2}.$$

The values of A_1 and A_3 can be found as shown previously:

$$A_1 = \left.\frac{F(z)(z-z_1)}{z}\right|_{z=z_1} \tag{12.94}$$

$$A_3 = \left.\frac{F(z)(z-z_2)^2}{z}\right|_{z=z_2}. \tag{12.95}$$

However, we run into problems when we try to evaluate A_2. Let's see what happens when we multiply both sides by $(z-z_2)$:

$$\frac{F(z)(z-z_2)}{z} = \frac{A_1(z-z_2)}{z-z_1} + A_2 + \frac{A_3}{z-z_2}. \tag{12.96}$$

If we now evaluate this equation at $z = z_2$, the third term on the right-hand side becomes undefined. In order to avoid this problem, we first multiply both sides by $(z-z_2)^2$ and take the derivative with respect to z prior to evaluating the equation at $z = z_2$:

$$\frac{F(z)(z-z_2)^2}{z} = \frac{A_1(z-z_2)^2}{z-z_1} + A_2(z-z_2) + A_3. \tag{12.97}$$

Taking the derivative of both sides with respect to z, we get

$$\frac{d}{dz}\frac{F(z)(z-z_2)^2}{z} = \frac{2A_1(z-z_2)(z-z_1) - A_1(z-z_2)^2}{(z-z_1)^2} + A_2. \tag{12.98}$$

If we now evaluate the expression at $z = z_2$, we get

$$A_2 = \left.\frac{d}{dz}\frac{F(z)(z-z_2)^2}{z}\right|_{z=z_2} \tag{12.99}$$

Generalizing this approach, we can show that if $D(z)$ has a root of order m at some z_k, that portion of the partial fraction expansion can be written as

$$\frac{F(z)}{z} = \frac{A_1}{z - z_k} + \frac{A_2}{(z - z_k)^2} + \cdots + \frac{A_m}{(z - z_k)^m} \tag{12.100}$$

and the lth coefficient can be obtained as

$$A_l = \frac{1}{(m-l)!} \frac{d^{(m-l)}}{dz^{(m-l)}} \frac{F(z)(z - z^k)^m}{z} \bigg|_{z=z_k} \tag{12.101}$$

Finally, let us drop the requirement that the degree of $D(z)$ be greater or equal to the degree of $N(z)$. When the degree of $N(z)$ is greater than the degree of $D(z)$, we can simply divide $N(z)$ by $D(z)$ to obtain

$$F(z) = \frac{N(z)}{D(z)} = Q(z) + \frac{R(z)}{D(z)} \tag{12.102}$$

where $Q(z)$ is the quotient and $R(z)$ is the remainder of the division operation. Clearly, $R(z)$ will have degree less than $D(z)$.

To see how all this works together, consider the following example.

Example 12.9.5:

Let us find the inverse Z-transform of the function

$$F(z) = \frac{2z^4 + 1}{2z^3 - 5z^2 + 4z - 1}. \tag{12.103}$$

The degree of the numerator is greater than the degree of the denominator, so we divide once to obtain

$$F(z) = z + \frac{5z^3 - 4z^2 + z + 1}{2z^3 - 5z^2 + 4z - 1}. \tag{12.104}$$

The inverse Z-transform of z is δ_{n-1}, where δ_n is the discrete delta function defined as

$$\delta_n = \begin{cases} 1 & n = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{12.105}$$

Let us call the remaining ratio of polynomials $F_1(z)$. We find the roots of the denominator of $F_1(z)$ as

$$F_1(z) = \frac{5z^3 - 4z^2 + z + 1}{2(z - 0.5)(z - 1)^2}. \tag{12.106}$$

Then

$$\frac{F_1(z)}{z} = \frac{5z^3 - 4z^2 + z + 1}{2z(z-0.5)(z-1)^2} \tag{12.107}$$

$$= \frac{A_1}{z} + \frac{A_2}{z-0.5} + \frac{A_3}{z-1} + \frac{A_4}{(z-1)^2}. \tag{12.108}$$

Then

$$A_1 = \left.\frac{5z^3 - 4z^2 + z + 1}{2(z-0.5)(z-1)^2}\right|_{z=0} = -1 \tag{12.109}$$

$$A_2 = \left.\frac{5z^3 - 4z^2 + z + 1}{2z(z-1)^2}\right|_{z=0.5} = 4.5 \tag{12.110}$$

$$A_4 = \left.\frac{5z^3 - 4z^2 + z + 1}{2z(z-0.5)}\right|_{z=1} = 3. \tag{12.111}$$

To find A_3, we take the derivative with respect to z, then set $z = 1$:

$$A_3 = \frac{d}{dz}\left[\frac{5z^3 - 4z^2 + 2z + 1}{2z(z-0.5)}\right]\Bigg|_{z=1} = -3. \tag{12.112}$$

Therefore,

$$F_1(z) = -1 + \frac{4.5z}{z-0.5} - \frac{3z}{z-1} + \frac{3z}{(z-1)^2} \tag{12.113}$$

and

$$f_{1,n} = -\delta_n + 4.5(0.5)^n u[n] - 3u[n] + 3nu[n] \tag{12.114}$$

and

$$f_n = \delta_{n-1} - \delta_n + 4.5(0.5)^n u[n] - (3-3n)u[n]. \tag{12.115}$$

♦

12.9.3 Long Division

If we could write $F(z)$ as a power series, then from the Z-transform expression the coefficients of z^{-n} would be the sequence values f_n.

Example 12.9.6:

Let's find the inverse z-transform of

$$F(z) = \frac{z}{z-a}.$$