Chapter 1

Channel Coding and Viterbi Algorithm

There are two major categories of channel codes: block and convolutional. This chapter deals mainly with convolutional coding. A *linear block code is* described by two integers, n and k, and a generator matrix or polynomial. The integer k is the number of data bits that form an input to a block encoder. The integer n is the total number of bits in the associated codeword out of the encoder. A characteristic of linear block codes is that each codeword n-tuple is uniquely determined by the input message k-tuple. The ratio *kln is* called the *rate of the code* — a measure of the amount of added redundancy. A *convolutional code is* described by three integers, n, k, and K, where the ratio k/n has the same code rate significance (information per coded bit) that it has for block codes; however, n does *not* define a block or codeword length as it does for block codes. The integer K is a parameter known as the *constraint length;* it represents the number of k-tuple stages in the encoding shift register. An important characteristic of convolutional codes, different from block codes, is that the encoder has

memory — the *n*-tuple emitted by the convolutional encoding procedure is not only a function of an input *k*-tuple, but is also a function of the previous K - 1 input *k*-tuples. In practice, *n* and *k* are small integers and K is varied to control the redundancy.

1.1 CONVOLUTIONAL ENCODING

Figure 1.1 represents the block diagram of a convolutional encode/decode and modulator/demodulator part in a typical communication link. The input message source is denoted by the sequence $\mathbf{m} = m_1, m_2, \ldots\ldots\ldots, m_i, \ldots\ldots$, where each m_i represents a binary digit (bit). We shall assume that each m_i is equally likely to be a one or a zero, and independent from digit to digit. Being independent, the bit sequence lacks any redundancy; that is, knowledge about bit m_i gives no information about m_j $(i \neq j)$. The encoder transforms each sequence $\mathbf{m}$ into a unique codeword sequence $\mathbf{U} = \mathbf{G}(\mathbf{m})$. Even though the sequence $\mathbf{m}$ uniquely defines the sequence **U**, a key feature of convolutional codes is that a given *k*-tuple

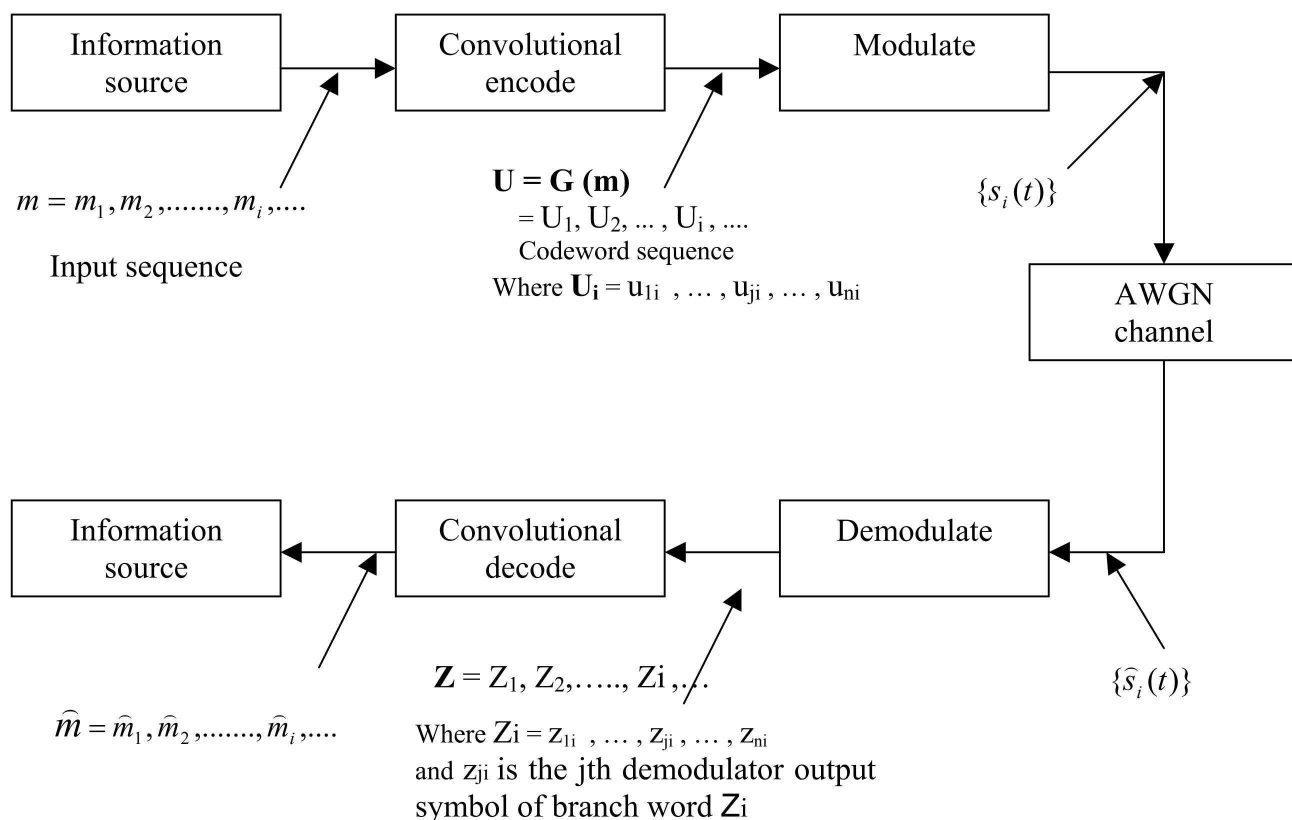

Figure 1.1 Encode/decode and modulator/demodulator part in a typical communication link.

within **m** does *not* uniquely define its associated *n*-tuple within **U** since the encoding of each *k*-tuple is *not only* a function of that *k*-tuple but is also a function of the K - 1 input *k*-tuples that precede it. The sequence **U** can be partitioned into a sequence of branch words: **U** = U_1, U_2, ... , U_i, ... Each branch word U_i is made up of binary *code symbols,* often called *channel symbols, channel bits,* or *coded bits;* unlike the input message bits the code symbols are not independent.

In a typical communication application, the codeword sequence **U** modulates a waveform *s(t)*. During transmission, the waveform *s(t)* is corrupted by noise, resulting in a received waveform *s(t)* and a demodulated sequence **Z** = Z_1, Z_2, . . . , Z_i , . . . , as indicated in Figure 1.1. The task of the decoder is to produce an estimate $\widehat{m} = \widehat{m}_1, \widehat{m}_2, ..., \widehat{m}_i,$ of the original message sequence, using the received sequence **Z** together with a priori knowledge of the encoding procedure.

A general convolutional encoder, shown in figure 1.2, is mechanized with a *k*K-stage shift register and *n* modulo-2 adders, where K is the constraint length. The constraint length

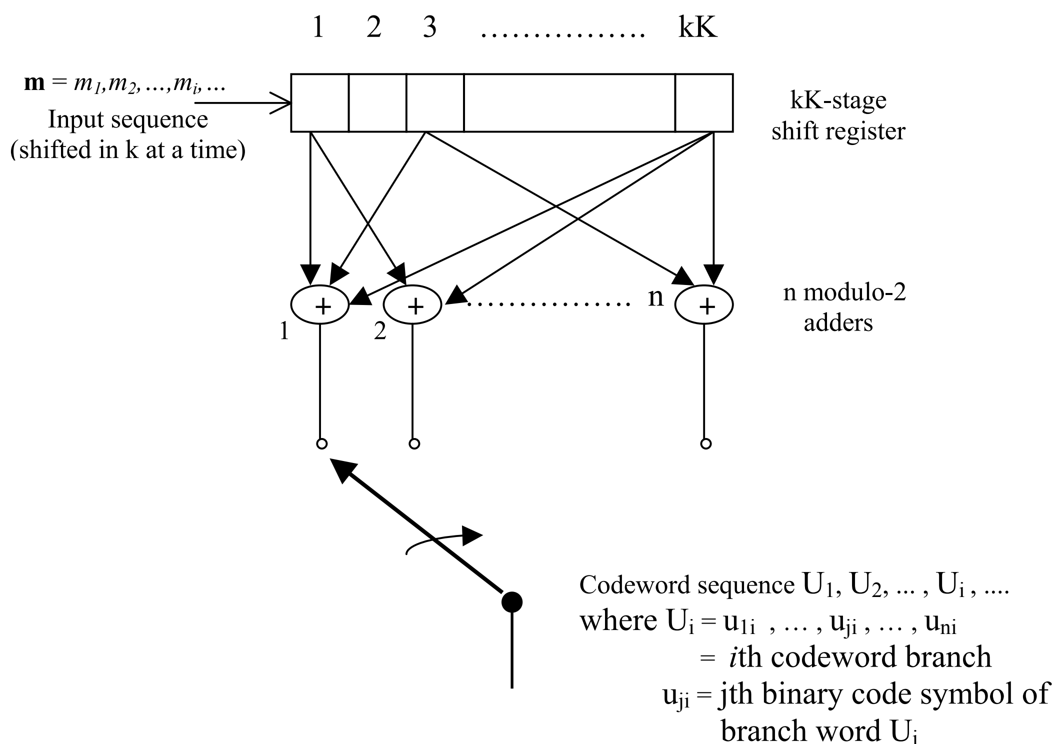

Figure 1.2 Convolutional encoder with constraint length K and rate *k/n.*

represents the number of *k*-bit shifts over which a single information bit can influence the

encoder output. At each unit of time, k bits are shifted into the first k stages of the register; all bits in the register are shifted k stages to the right, and the outputs of the n adders are sequentially sampled to yield the binary code symbols or coded bits. These code symbols are then used by the modulator to specify the waveforms to be transmitted over the channel. Since there are n coded bits for each input group of k message bits, the code rate is k/n message bit per coded bit, where $k < n$.

We shall consider only the most commonly used binary convolutional encoders for which $k = 1$, that is, those encoders in which the message bits are shifted into the encoder one bit at a time, although generalization to higher-order alphabets is straightforward [1, 2]. For the $k = 1$ encoder, at the ith unit of time, message bit m_i is shifted into the first shift register stage; all previous bits in the register are shifted one stage to the right, and as in the more general case, the outputs of the n adders are sequentially sampled and transmitted. Since there are n coded bits for each message bit, the code rate is $1/n$. The n code symbols occurring at time t_i comprise the ith branch word, $U_i = u_{1i}, \ldots, u_{ji}, \ldots, u_{ni}$, where u_{ji} ($j = 1, 2, \ldots, n$) is the jth code symbol belonging to the ith branch word. Note that for the rate $1/n$ encoder, the kK-stage shift register can be referred to simply as a K-stage register, and the constraint length K, which was expressed in units of k-tuple stages, can be refereed to as constraint length in units of bits.

1.2 CONVOLUTIONAL ENCODER PRESENTATION

To describe a convolutional code, one needs to characterize the encoding function **G (m)**, so that given an input sequence **m**, one call readily compute the output sequence **U**. Several methods are used for representing *a* convolutional encoder, the most popular being the *connection pictorial, connection vectors or polynomials,* the *state diagram* the *tree diagram* and the *trellis diagram.* Some of them will be discussed below.

1.2.1 Connection Representation

We shall use the convolutional encoder, shown in Figure 1.3, as a model for discussing convolutional encoders. The figure illustrates a (2, 1) convolutional encoder with constraint length K = 3. There are $n = 2$ modulo-2 adders; thus he code rate k/n is 1/2. At each input bit time, a bit is shifted into the leftmost stage and the bits in the register are shifted one position to the right. Next, the output switch samples the output of each modulo-2 adder (i.e., first the upper adder, then the lower adder), thus forming the code symbol pair making up the branch word associated with the bit just inputted. The sampling is repealed for each inputted bit. The choice of connections between the adders and the stages of the register gives rise to the characteristics of the code. Any change in the choice of connections results in a different code. The connections are, of course, not chosen or changed arbitrarily. The problem of choosing connections to yield good distance properties is complicated and has not been solved in general; however, good codes have been found by computer search for all constraint lengths less than about 20.

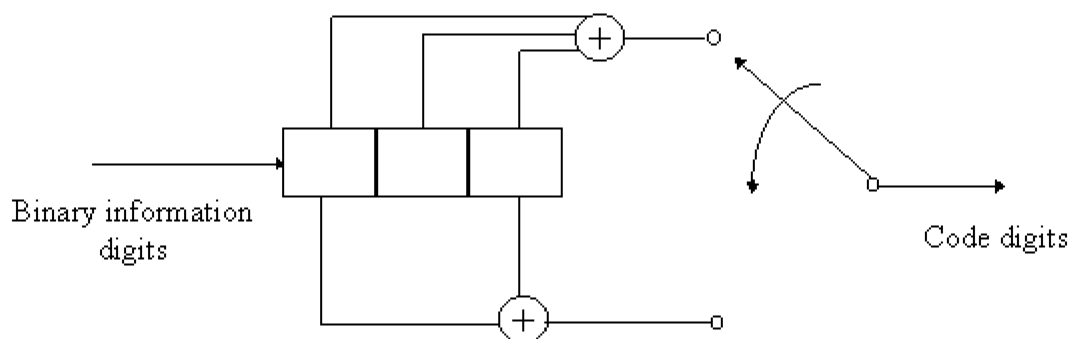

Figure 1.3 A simple rate ½ convolutional code encoder.
(The rectangular box represents one element of a serial shift register.)

Unlike a block code that has a fixed word length *n,* a convolutional code has no particular block size. However, convolutional codes are often forced into a block structure by *periodic truncation.* This requires a number of zero bits to be appended to the end of the input data

sequence, for the purpose of clearing or *flushing* the encoding shift register of the data bits. Since the added zeros carry no information, the *effective code rate* falls below *k/n.* To keep the code rate close to *kln,* the truncation period is generally made as long as practical.

One way to represent the encoder is to specify a set of n *connection vectors,* one for each of the n modulo-2 adders. Each vector has dimension K and describes the connection of the encoding shift register to that modulo-2 adder. A one in the ith position of the vector indicates that the corresponding stage in the shift register is connected to the modulo-2 adder, and a zero in a given position indicates that no connection exists between the stage and the modulo-2 adder. For the encoder example in Figure 1.3, we call write the connection vector g_1 for the upper connections find g_2 for the lower connections as follows:

$$g_1 = 1\ 1\ 1$$

$$g_2 = 1\ 0\ 1$$

Consider that a message vector m = 1 0 1 is convohltionally encoded with the encoder shown in Figure 1.3. The three message bits are inputted, one at a time, at times t_1, t_2, and t_3, as shown in Figure 1.4. Subsequently, (K - 1) = 2 zeros are inputted at times t_4 and t_5 to flush the register and thus ensure that the tail end of the message is shifted the full length of the register. The output sequence is seen to be 1110001011, where the leftmost symbol represents the earliest transmission. The entire output sequence, including the code symbols as a result of flushing, are needed to decode the message. To flush the message from the encoder requires one less zero than the number of stages in the register, or K - I flush bits. Another zero input is shown at time t_6, for the reader to verify that the corresponding branch word output is then 00.

1.2.2 Impulse Response of the Encoder

We can approach the encoder in terms of its *impulse response*—that is, the response of the encoder to a single "one" bit that moves through it. Consider he contents of the register in Figure 1.3 as a one moves through it.

Register contents	Branch word	
	u_1	u_2
1 0 0	1	1
0 1 0	1	0
0 0 1	1	1

Input sequence: 1 0 1

Output sequence: 1 1 1 0 1 1

The output sequence for the input "one" is called the impulse response of the encoder. Then for the input sequence **m** = 1 0 1, the output may be found by the superposition or the *lineal addition* of the time-shifted input "impulses" as follows:

Input **m**	Output				
1	1 1	1 0	1 1		
0		0 0	0 0	0 0	
1			1 1	1 0	1 1
Modulo-2 sum:	1 1	1 0	0 0	1 0	1 1

Observe that this is the same output as that obtained in Figure 1.4, demonstrating that *convolutional codes* are linear. It is from this property of generating the output by the linear addition of time shifted impulses, or the convolution of the input sequence w*i*th the impulse response of the encoder, that we derive thc name *convolutional encoder.* Often, this encoder characterization is presented in terms of an infinite-order generator matrix [6].

Notice that the *effective code rate* for the foregoing example with 3-bit input sequence and 10-bit output sequence is *k/n* = 3/10—quite a bit less than the rate 1/2 that might have been expected from the knowledge that each input data bit yields a pair of output channel bits. The reason for the disparity is that the final data bit into the encoder needs to be shifted through the encoder. All of the output channel bits are needed in the decoding process. If the message had been longer, say 300 bits, the output codeword sequence would contain 604

bits, resulting in a code rate of 300/604—much closer to 1/2.

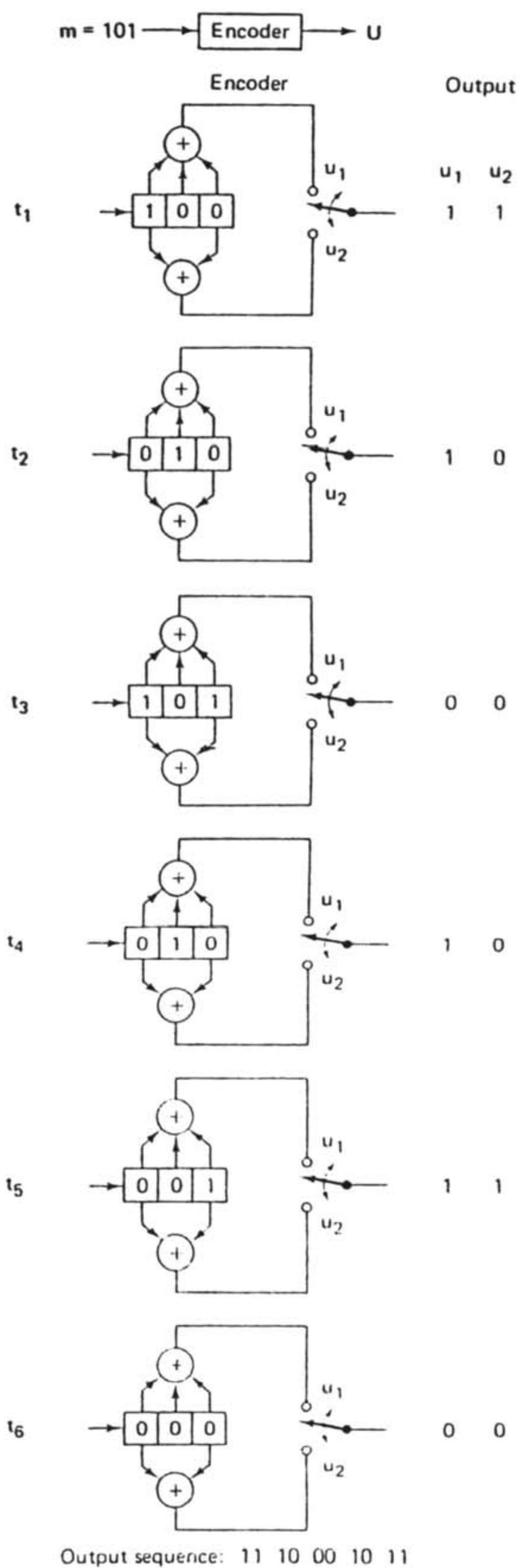

Figure 1.4 Convolutional encoding a message sequence with a rate ½ , K = 3 encoder.

1.2.2.1 Polynomial Presentation

Sometimes, the encoder connections are characterized by *generator polynomials*. We can represent a convolutional encoder with a set of *n* generator polynomials, one for each of the *n* modulo-2 adders. Each polynomial is of degree K - 1 or less and describes the connection of the encoding shift register to that modulo-2 adder, much the same way that a connection vector does. The coefficient of each term in the (K – 1)-degree polynomial is either 1 or 0, depending on whether a connection exists or does not exist between the shift register and the modulo-2 adder in question- For the encoder example in Figure 1.3, we can write the generator polynomial $g_1(X)$ for the upper connections and $g_2(X)$ for the lower connections as follows:

$$g_1(X) = 1 + X + X^2$$
$$g_2(X) = 1 + X^2$$

where the lowest-order term in the polynomial corresponds to the input stags of the register. The output sequence is found as follows:

$$\mathbf{U(X) = m(X)\, g_1(X) \text{ interlaced with } m(X) g_2(X)}$$

First, express the message vector **m** = 1 0 1 as a polynomial—that is, $\mathbf{m}(X) = 1 + X^2$. We shall again assume the use of zeros following the message bits, to flush the register. Then the output polynomial, **U (*X*)**, or the output sequence, U, of the Figure 1.3 encoder can be found for the input message **m** as follows:

$$m(X)g_1(X) = (1 + X^2)(1 + X + X^2) = 1 + X + X^3 + X^4$$
$$m(X)g_2(X) = (1 + X^2)(1 + X^2) = 1 + X^4$$

$$m(X)g1(X) = 1 + X + 0X^2 + X^3 + X^4$$
$$m(X)g1(X) = 1 + 0X + 0X^2 + X^3 + X^4$$

$$U(X) = (1,1) + (1,0)X + (0,0)X^2 + (1,0)X^3 + (1,1)X^4$$

In this example we started with another point of view—that the convolutional encoder can be

treated as a set of *cyclic code shift registers.* We represented the encoder *with polynomial generators* as used for describing cyclic codes. However, we arrived at the same output sequence as in Figure 1.4 and the same output sequence as the impulse response treatment of the preceding section. For a good presentation of convolutional code structure in the context of linear sequential circuits.

1.2.3 State Representation and the state diagram

The state of a rate l/*n* convolutional encoder is defined as the contents of the rightmost K-1 stages (see Figure 1.3). Knowledge of the state together with Knowledge of the next input is necessary and sufficient to determine the next output. Let the state of the encoder at time, t_i, be defined as $X_i = m_{i-1}\ ,\ m_{i-2}\ ,\ \ldots\ldots\ ,\ m_{i-k+1}$. The ith codeword branch, $U_{i,}$ is completely determined by state X_i and the present input bit *m,;* thus the state X_i represents the past history of the encoder in determining the encoder output. The encoder state is said to be ***Markov,*** in the sense that the probability, ***P*** $(X_{i+1}|X_i\ ,\ X_{i-1}\ ,\ .\ .\ .,\ X_0)$, of being n state $X_{i+\ 1}$, given all previous states, depends only on the most recent state, of,; that is, the probability is equal to ***P*** $(X_{i+1}|X_i\ )$.One way to represent simple encoders is with a *state diagram* such a representation for the encoder in Figure 1.3 is shown in Figure 1.5. The states, shown in the boxes of the diagram, represent the possible contents of the rightmost K - 1 stages of the register, and the paths between the states represent the output branch words resulting from such state transitions. The states of the register are designated a = 00, b = 10, c = 01, and d = 11; the diagram shown in Figure 1.5 illustrates all the state transitions that are possible for the encoder in Figure 1.3. There are *only two transitions* emanating from each state corresponding to the two possible input bits. Next to each path between states is written the output branch word associated with the state transitions. In drawing the path, we use the convention that a solid line denotes a path associated with an input bit, zero, and a dashed line denotes a path associated with an input bit,one. Notice that it is *not possible* in a single transition to move from a given state to *any arbitrary state. As* a consequence of shifting-in one bit at a time, there are only two possible state transitions that the register can make at each bit time- For example, if the present encoder state is 00, the only *possibilities* for the state at the next shift

are 00 or 10.

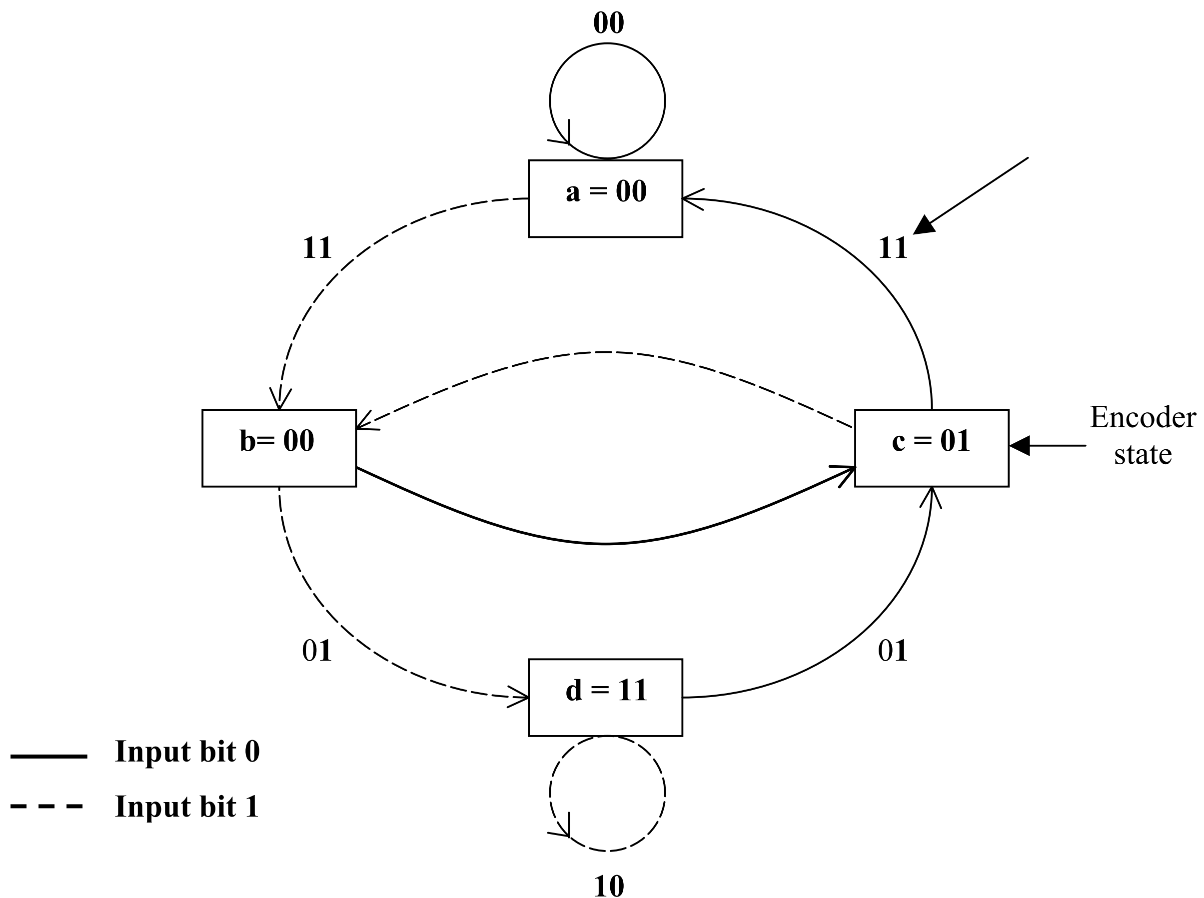

Figure 1.5 Encoder state diagram (rate ½ , K = 3)

1.2.4 The Tree Diagram

Although the state diagram completely characterizes the encoder, one cannot easily use it for tracking the encoder transitions as a function of time since the diagram cannot represent time history. The tree diagram adds the *dimension of time to* the state diagram. The tree diagram for the convolutional encoder shown in Figure 1.3 is illustrated in Figure 1.5. At each successive input bit time the encoding procedure can be described by traversing the diagram from left to right, each tree branch describing an output branch word.

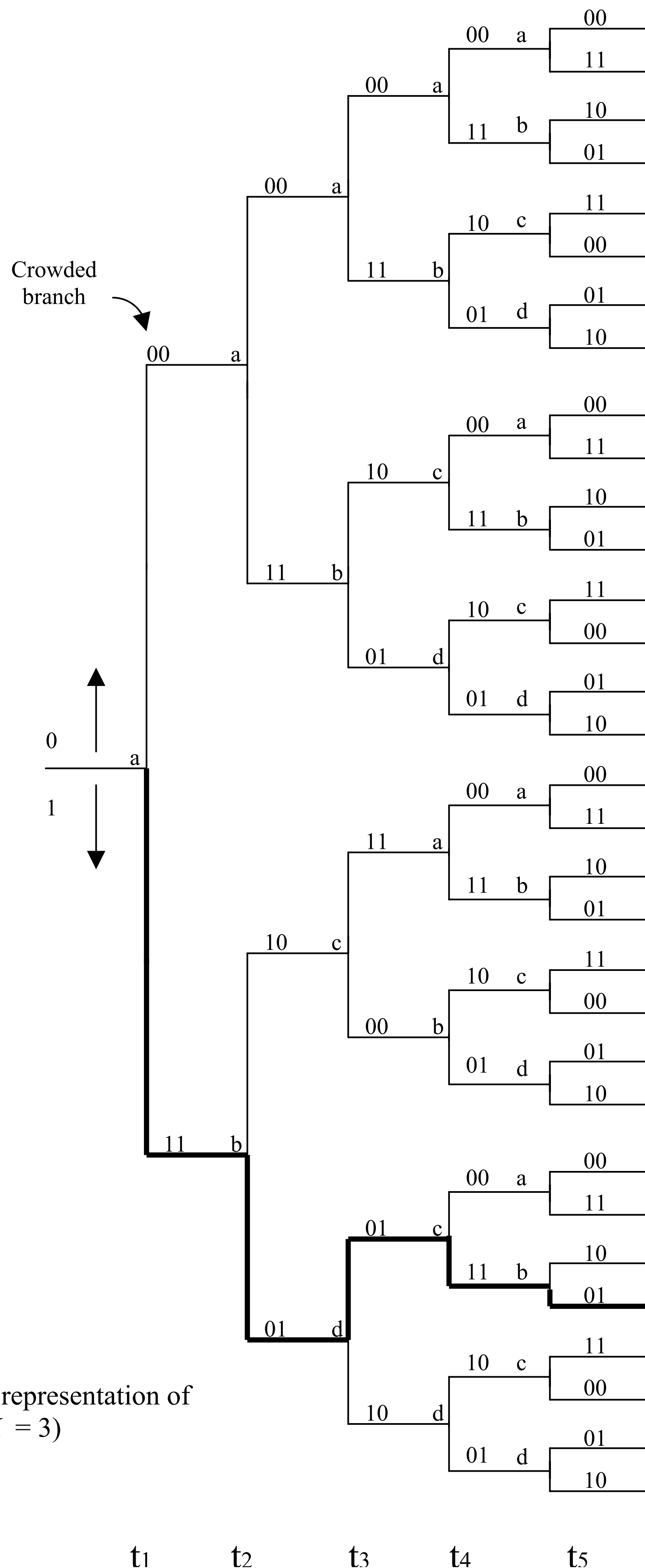

Figure 1.6 Tree representation of encoder (rate ½, K = 3)

The branching rule for finding a codeword sequence is as follows: if the input bit is a zero, its associated branch word is found by moving to the next rightmost branch in the upward direction. If the input bit is a one its branch word is found by moving to the next rightmost branch in the downward direction. Assuming that the initial contents of the encoder is all zeros, the diagram shows that if the first input bit is a zero, the output branch word is 00 and, if the first input bit is a one, the output branch word is 11. Similarly, if the first input bit is a one and the second input bit is a zero, the second output branch word is 10. Or, if the first input bit is a one and the second input bit is a one, the second output branch word is 01. Following this procedure we see that the input sequence 11011 traces the heavy line drawn on the tree diagram in Figure 1.5. This path corresponds to the following output codeword sequence: 1101010001.

The added dimension of time in the tree diagram (compared to the state diagram) allows one to dynamically describe the encoder as a function of a particular input sequence. However, can you see one problem in trying to use a tree diagram for describing a sequence of any length? The numbers of branches increases as a function of 2^L, where L is the number of bits in the input sequence. You would quickly run out of paper, and patience.

1.2.5 The Trillis Diagram

Observation of the Figure 1.6 tree diagram shows that for this example, the structure repeats itself at time t4, after the third branching (in general, the tree structure *repeats after-K branchings,* where K is the constraint length). We label each node in the tree of Figure 1.6 to correspond to the four possible states in the shift register, as follows: a = 00, b = 10, c = 01, and d = 11. The first branching of the tree structure, at time t_1, produces a pair of nodes labeled *a* and b. At each successive branching the number of nodes double The second branching, at time t_2. Results in four nodes labeled *a, b, c,* and d. After the *third* branching there are a total of eight nodes; two of them are labeled *a,* two are labeled *b,* two are labeled c, and two are labeled d.We can see that all branches emanating from two nodes of the same state generate identical branch word sequences. From this point on, the upper and the lower halves of the tree are identical. The reason for this should be obvious from examination of the encoder in Figure

1.3. As the fourth input bit enters the encoder on the left, the first input bit is ejected on the right and no longer influences the output branch words. Consequently, the input sequences 100 x y . . . and 000 x y… , where the leftmost bit is the earliest bit, generate tl1c same branch words after the (K = 3)rd branching. This means that any two nodes having the same state label, al the same time t_i, can be merged since all succeeding paths will be indistinguishably. If we do this to the tree structure of Figure 1.6, we obtain another diagram called the trellis. The *trellis dia*gram, by exploiting the repetitive structure, provides a more manageable encoder description than does the tree diagram. The trellis diagram for the convolutional encoder of Figure 6.3 is shown in Figure 6.7.

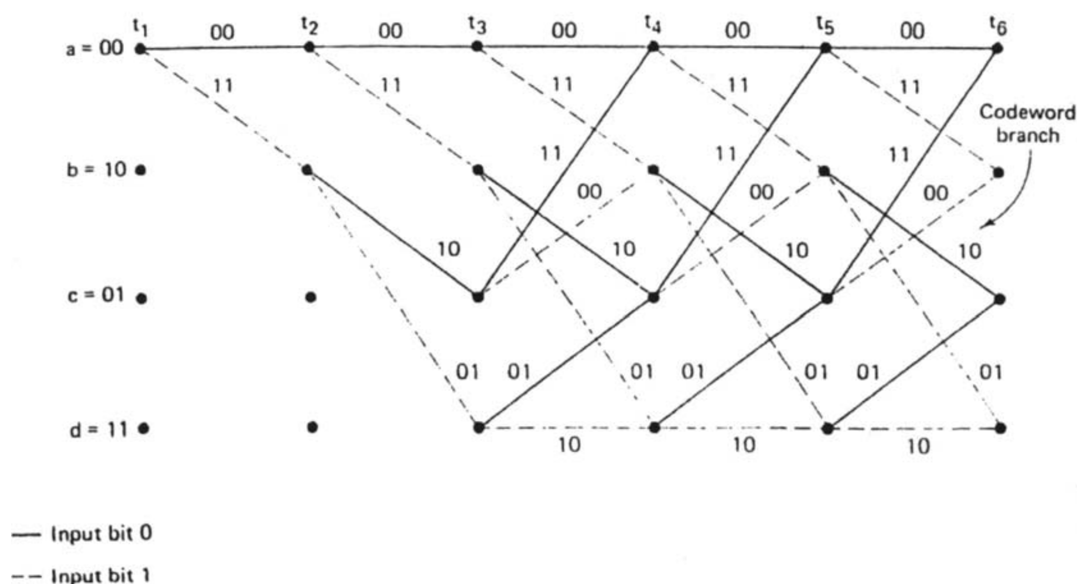

Figure 1.7 Encoder trellis diagram (rate ½ , K = 3)

In drawing the trellis diagram, we use the same convention that we introduced with the state diagram—that a solid line denotes the output generated by an input bit, zero, and a dashed line denotes the output generated by an input bit, one. The nodes of the trellis characterize the encoder states; the first row nodes correspond to the state a = 00, the second and subsequent rows correspond to the states b = 10, c = 01, and d = 11. At each unit of time the trellis requires 2^{K-1} nodes to represent the 2^{K-1} possible encoder states. The trellis in our example assumes a fixed periodic structure after trellis depth 3 is reached (at time t_4). In the

general case, the fixed structure prevails after depth K is reached. After this point, each of the states can be entered from either of two preceding states. Also, each of the states can transition to one of two states. Of the two outgoing branches, one corresponds to an input bit zero and the other corresponds to an input bit one. On Figure 1.7 the output branch words corresponding to the state transitions appear as labels on the trellis branches.

1.3 FORMATION OF THE CONVOLUTIONAL DECODING PROBLEM

1.3.1 Maximum Likelihood Decoding

If all input message sequences are equally likely, a decoder that achieves the minimum probability of error is one that compares the conditional probabilities, also called the *likelihood functions,* $\boldsymbol{P}(\mathbf{Z}|\mathbf{U}^{(m)})$, where $\mathbf{Z}$ is the received sequence and $\mathbf{U}^{(m)}$ is one of the possible transmitted sequences, and chooses the maximum. The decoder chooses $\mathbf{U}^{(m)}$ if

$$\boldsymbol{P}(\mathbf{Z}|\mathbf{U}^{(m')}) = \max \boldsymbol{P}(\mathbf{Z}|\mathbf{U}^{(m)}) \qquad \text{for all } \mathbf{U}^{(m)} \qquad (1.1)$$

The maximum likelihood concept, as stated in Equation (1.1), is a fundamental development of decision theory, it is the formalization of a “common-sense” way to make decisions when there is statistical knowledge of the possibilities. In the binary demodulation there were only two equally likely possible signals, $s_1(t)$ or $s_2(t)$, that might have been transmitted. Therefore, to make the binary maximum likelihood decision. Given a received signal, meant only to decide that $s_1(t)$ was transmitted if

$$p(z|s_1) > p(z|s_2)$$

Otherwise, to decide that $s_2(t)$ was transmitted. The parameter z represents z(t), the receiver output at a symbol duration time t = T. However, when applying maximum likelihood to the convolutional decoding problem, there is typically a *multitude* of possible codeword

sequences that might have been transmitted. To be specific, an L-bit codeword sequence is a member of a set of 2^L possible sequences. Therefore, in the maximum likelihood context, we can say that the decoder chooses a particular $\mathbf{U}^{(m')}$ as the transmitted sequence if the likelihood $\boldsymbol{P}$ $(\mathbf{Z}|\mathbf{U}^{(m')})$ is greater than the likelihood *of* all the other possible transmitted sequences. Such an optimal decoder, which minimizes the error probability (for he case where all transmitted sequences are equally likely), is known as a *maximum likelihood decoder*. The likelihood functions are given or computed from he specifications of the channel.

We will assume that the noise is additive white Gaussian with zero mean and the channel is *memoryless* which means that the noise affects each code symbol *independently* of all the other symbols. For a convolutional code of rate $1/n$, we can therefore express the likelihood, $\boldsymbol{P}$ $(\mathbf{Z}|\mathbf{U}^{(m)})$ as follows:

$$P(Z \mid U^{(m)}) = \prod_{i=1}^{\infty} P(Z_i \mid U_i^{(m)}) = \prod_{i=1}^{\infty}\prod_{j=1}^{n} P(z_{ji} \mid u_{ji}^{(m)}) \tag{2.2}$$

where Z_i is the ith branch of the received sequence $\mathbf{Z}$. $U_i^{(m)}$ is the ith branch of a particular codeword sequence $U_i^{(m)}$, z_{ji} the jth code symbol of Z_i, and $u_{ji}^{(m)}$ the jth code symbol of $U_i^{(m)}$, each branch comprising n code symbols. The decoder problem consists of choosing a path through the trellis of Figure 1.7 (each possible path defines a codeword) such that

$$\prod_{i=1}^{\infty}\prod_{j=1}^{n} P(z_{ji} \mid u_{ji}^{(m)}) \qquad \text{is maximized} \tag{3.2}$$

Generally, it is computationally more convenient to use the logarithm of the likelihood function since this permits the summation, instead of the multiplication of terms. We are able to use this transformation because the logarithm is a monotonically increasing function and thus will not alter the final result in our codeword selection. We can define the log-likelihood function $\gamma_U(m)$ as

$$\gamma_U(m) = \log P(Z \mid U^{(m)}) = \sum_{i=1}^{\infty} \log P(Z_i \mid U_i^{(m)}) = \sum_{i=1}^{\infty}\sum_{j=1}^{n} \log P(z_{ji} \mid u_{ji}) \tag{1.4}$$

The decoder problem now consists of choosing a path through the tree of Figure 1.6 or the

trellis of Figure 1.7 such that $\gamma_U(m)$ is maximized. For the decoding of convolutional codes, either the tree or the trellis structure can be used. In the tree representation of the code, the fact that the paths remerge is ignored. Since the number of possible sequences for an L-symbol-long sequence is 2^L, maximum likelihood decoding of an L-bit-long received sequence, using a tree diagram, requires the “brute force” or exhaustive comparison of 2^L accumulated log-likelihood metrics, representing all the possible different codewords that could have been transmitted. Hence it is not practical to consider maximum likelihood decoding with a tree structure. It is shown in a later section that with the use of the trellis representation of the code, it is possible to configure a decoder which can discard the paths that could not possibly be candidates for the maximum likelihood sequence. The decoded path is chosen from some reduced set of *surviving paths.* Such a decoder is still optimum in the sense that the decoded path is the same as the decoded path obtained from a "brute force" maximum likelihood decoder, but the early rejection of unlikely paths reduces the decoding complexity.

There are several algorithms that yield *approximate* solutions to the maximum likelihood decoding problem, including sequential and threshold. Each of these algorithms is suited to certain special applications, but are all suboptimal. In contrast, the *Viterbi decoding algorithm* performs maximum likelihood decoding and is therefore optimal. This does not imply that the *Viterbi algorithm* is best for every application; there are severe constraints imposed by hardware complexity. The Viterbi algorithm is considered in Sections 1.3.3 and 1.3.4.

1.3.2 Channel Models: Hard versus Soft Decisions

Before specifying an algorithm that will determine the maximum likelihood decision, let us describe the channel. The codeword sequence $\mathbf{U}^{(m)}$, made up of branch words, with each branch word comprised of n code symbols, can be considered tan endless stream, as opposed to a block code, in which the source data and their codewords are partitioned into precise block sizes. The codeword sequence shown in Figure 1.1 emanates from the convolutional

encoder and enters the modulator, where the code symbols are transformed into signal

waveforms The modulation may be baseband (e.g., pulse waveforms) or bandpass (e.g., PSK or FSK). In general, l symbols at a lime, where l is an integer, are mapped into signal waveforms $s_i(t)$, where i = 1, 2, . . ., M = 2^l. When l = 1, the modulator maps each code symbol into a binary waveform. The channel over which the waveform is transmitted is assumed to corrupt the signal with Gaussian noise. When the corrupted signal is received, it is first processed by the demodulator and then by the decoder.

Consider that a binary signal, transmitted over a symbol interval (0, T), is represented by $s_1(t)$ for a binary one and $s_2(t)$ for a binary zero. The received signal is $r(t) = s_i(t) + n(t),$ where n(t) is a zero-mean Gaussian noise process. The detection of $r(t)$ is described in terms of two basic steps. In the first step, the received waveform is reduced to a single number. $z(T)= a_i + n_o$, where *a, is* the signal component of $z(T)$ and n_o is the noise component. The noise component, n_o, is a zero-mean *Gaussian random variable,* and thus *z(T) is* a *Gaussian random variable* with a mean of either a_1 or a_2 depending on whether a binary one or binary zero was sent. In the second step of the detection process a decision was made as to which signal was transmitted, on the basis of comparing $z(T)$ to a threshold. The conditional probabilities of $z(T)$, $p(z|s_1)$ and $p(z|s_2)$ are shown in Figure 1.8, labeled likelihood of s_1 and likelihood of s_2. The demodulator in Figure 1.1, converts the set of time-ordered random variables. $\{z(T)\}$, into a code sequence, **Z**, and passes it on to the decoder. The demodulator

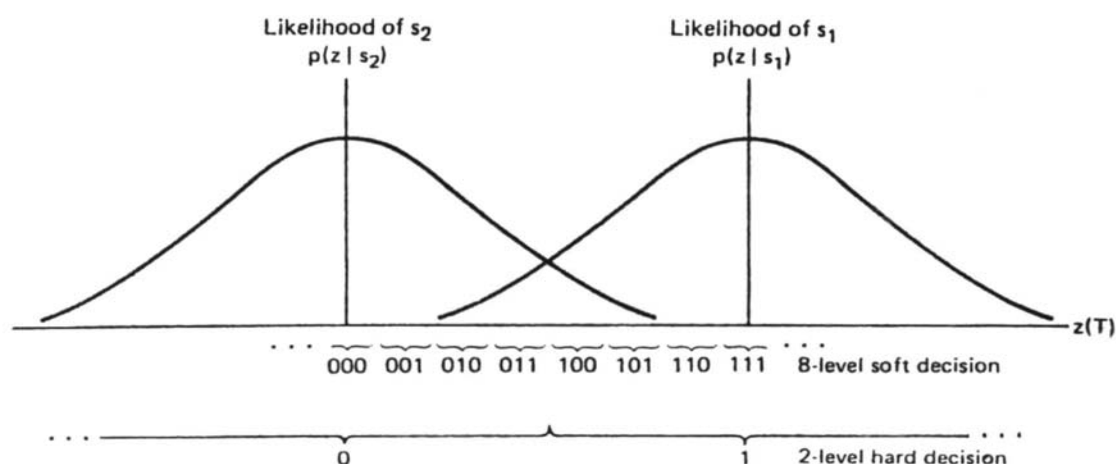

Figure 1.8 Hard and soft decoding decisions.

output can be configured in a variety of ways. It can be implemented to make a *firm of hard decision* as to whether $z(T)$ represents a zero or a one. In this case, the output of the

demodulator is quantized to two levels, zero and one, and fed into the decoder. Since the decoder operates on the hard decisions made by the demodulator, the decoding is called *hard-decision decoding*.

The demodulator can also be configured to feed the decoder with a *quantized value of z(T) greater that two levels,* or with an unquantized or analog value of *z(T)*. Such an implementation furnishes the decoder with more information than is provided in the hard-decision case. When the quantization level of the demodulator output is greater than two, the decoding is called *soft-decision decoding.* Eight levels (3-bits) of quantization are illustrated on the abscissa of Figure 1.8. When the demodulator sends a hard binary decision to the decoder, it sends it a single binary symbol. When the demodulator sends a soft binary decision, quantized to eight levels, it sends the decoder a 3-bit word describing an interval along *z(T)*. In effect, sending such a 3-bit word in place of a single binary symbol is equivalent to sending the decoder a *measure* of *confidence* along with the code symbol. Referring to Figure 1.8, if the demodulator sends 1 1 1 to the decoder, this is tantamount to declaring the code symbol to be a one with very high confidence, while sending a 1 0 0 is tantamount to declaring the code symbol to be a one with very low confidence. It should be clear that ultimately, every message decision out of the decoder must be a hard decision; otherwise, one might see computer printouts that read: "think it's a 1," "think it's a 0," and so on. The idea behind the demodulator *not making hard decisions* and sending more data (soft decisions) to the decoder can be thought of as an interim step to provide the decoder with more information, which the decoder then uses for recovering the message sequence (with better error performance than it could in the case of hard decision decoding).

For a Gaussian channel, eight-level quantization results in a performance improvement of approximately 2 dB in required signal-to-noise ratio compared to two-level quantization. This means that eight-level soft-decision decoding can provide the same probability of bit error as that of hard-decision decoding, but requires 2 dB less E_b/N_o for the same performance. Analog (or infinite-level quantization) results in a 2.2-dB performance improvement over two-level quantization; therefore, *eight-level quantization* results in a loss of approximately 0.2 dB

compared to infinitely fine quantization. For this reason, quantization to more than eight levels can yield little performance improvement. What price is paid for such improved soft-decision-decoder performance? In the case of hard decision decoding, a single bit is used

to describe each code symbol, while for eight-level quantized soft-decision decoding 3 bits are used to describe each code symbol; therefore, three times the amount of data must be handled during the decoding process. Hence the price paid for soft-decision decoding is an increase in required memory size at the decoder (and possibly a speed penalty).

Block decoding algorithms and convolutional decoding algorithms have been devised to operate with hard or soft decisions. However, soft-decision decoding is generally not used with block codes because it is considerably more difficult than hard-decision decoding to implement. The most prevalent use of soft-decision decoding is with the *Viterbi Convolutional decoding algorithm,* since with Viterbi decoding, soft decisions represent only a trivial increase in computation.

1.3.2.1 Binary Symmetric Channel

A binary symmetric channel (**BSC**) is a discrete memoryless that has binary input and output alphabets and symmetric transition probabilities. It can be described by the conditional probabilities

$$\begin{aligned} \boldsymbol{P}(0|1) &= \boldsymbol{P}(1|0) = p \\ \boldsymbol{P}(1|1) &= \boldsymbol{P}(0|0) = 1-p \end{aligned} \qquad (1.5)$$

as illustrated in Figure 1.9. The probability that an output symbol will differ from the input symbol is p, and the probability that the output symbol will be identical to the input symbol is (1 - p). The **BSC** is an example of a *hard decision channel,* which means that, even though continuous-valued signals may be received by the demodulator, a **BSC** allows only firm decisions such that each demodulator output symbol, z_{ji}, as shown in Figure 1.1, consists of one of two binary values. The indexing of z_{ji} pertains to the jth code symbol of the ith branch word. Z_i. The demodulator then feeds the sequence $\mathbf{Z} = \{Z_i\}$ to the decoder.

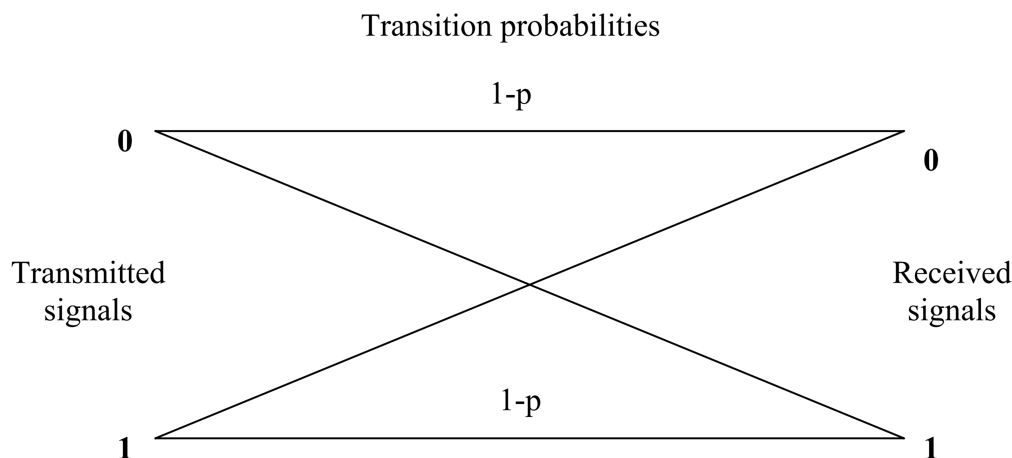

Figure 1.9 Formulation of the convolutional Decoding Problem

Let $\mathbf{U}^{(m)}$ be a transmitted codeword over a **BSC** with symbol error probability *p*, and let **Z** be the corresponding received decoder sequence. As noted' previously, a maximum likelihood decoder chooses the codeword $\mathbf{U}^{(m')}$ which > maximizes the likelihood, $\boldsymbol{P}(\mathbf{Z}|\mathbf{U}^{(m)})$ or its logarithm. For a **BSC**, this is equivalent to choosing the codeword, $\mathbf{U}^{(m')}$, that is closest in *Hamming distance* to **Z**. Thus *Hamming distance* is an appropriate metric to describe the distance or closeness of fit between $\mathbf{U}^{(m)}$ and **Z**. From all the possible transmitted sequences, $\mathbf{U}^{(m)}$, the decoder chooses the $\mathbf{U}^{(m')}$ sequence for which the distance to **Z** is minimum.

Suppose that $\mathbf{U}^{(m)}$ and **Z** are each *L*-bit-long sequences and that they differ in d_m positions [i.e., the Hamming distance between Used and **Z** is d_m]. Then, since the channel is assumed memoryless, the probability that this $\mathbf{U}^{(m)}$ was transformed to the specific received **Z** at distance d_m from it can be written

$$\boldsymbol{P}\,(\mathbf{Z}|\mathbf{U}^{(m)}) = p^{dm}\,(1-p)^{L-dm} \tag{1.6}$$

And the log-likelihood function is

$$\text{Log}\,\boldsymbol{P}\,(\mathbf{Z}|\mathbf{U}^{(m)}) = -\,d_m\ \log\{\,(1-p)/p\,\} + L\log(1-p) \tag{1.7}$$

If we compute this quantity for each possible transmitted sequence, the second term will be constant in each case. Assuming that $p < 0.5$, we can express Equation (1.7) as

$$\text{Log}\, \boldsymbol{P}\,(\mathbf{Z}|\mathbf{U}^{(m)}) = -A\, d_m - B \tag{1.8}$$

where A and B are positive constants. Therefore, choosing the codeword $\mathbf{U}^{(m')}$ such that the Hamming distance, d_m , to the received sequence **Z** is minimized corresponds to *maximizing the likelihood or log likelihood metrics.* Consequently, over a **BSC**, the log-likelihood metric is conveniently replaced by the Hamming distance, and a maximum likelihood decoder will choose, in the tree or trellis diagram, the path whose corresponding sequence, $\mathbf{U}^{(m')}$, is at the minimum *Hamming distance* to the received sequence **Z**.

1.3.2.2 Gaussian Channel

For a Gaussian channel, each demodulator output symbol, z_{ji}, as shown in Figure 1. 1, is a value from a continuous alphabet. The symbol z_{ji} cannot be labeled as a correct or incorrect detection decision. Sending the decoder such sop decisions can be viewed as sending a family of conditional probabilities of the different symbols. It can be shown that maximizing $\boldsymbol{P}(\mathbf{Z}|\mathbf{U}^{(m)})$ is equivalent to maximizing the inner product between the codeword sequence, $\mathbf{U}^{(m)}$ (consisting of binary symbols), and the analog-valued received sequence, **Z**. Thus the decoder chooses the codeword $\mathbf{U}^{(m')}$ if it maximizes

$$\sum_{i=1}^{\infty}\sum_{j=1}^{n} z_{ji} u_{ji}^{(m)} \tag{1.9}$$

This is equivalent to choosing the codeword User that is closest in *Euclidean distance* to **Z**. Even though the hard- and soft-decision channels require different metrics, the concept of choosing the codeword $\mathbf{U}^{(m')}$ that is closest to the received sequence, **Z**. is the same in both cases. To implement the maximization of Equation (1.9) exactly, the decoder would have to

be able to handle analog-valued arithmetic operations. This is impractical because the decoder is generally implemented digitally. Thus it is necessary to quantize the received symbols z_{ji}. Equation (1.9) is the discrete version of correlating an input received waveform, *r(t)*, with a

reference waveform, $s_i(t)$. The quantized Gaussian channel, typically referred to as a *soft-decision channel, is* the channel model assumed for the soft-decision decoding described earlier.

1.3.3 The Viterbi Convolutional Decoding Algorithm

The Viterbi decoding algorithm was discovered and analyzed by Viterbi in 1967. The Viterbi algorithm essentially performs maximum likelihood decoding; however, it reduces the computational load by taking advantage of the special structure in the code trellis. The advantage of Viterbi decoding, compared with brute-force decoding, is that the complexity of a Viterbi decoder is not a function of the number of symbols in the codeword sequence. The algorithm involves calculating a *measure of similarity, or distance* between the received signal, at time t_i, and all the trellis paths entering each state at time t_i. The Viterbi algorithm removes from consideration those trellis paths that could not possibly be candidates for the maximum likelihood choice. When two paths enter the same state, the one having the best metric is chosen; this path is called the *surviving path.* This selection of surviving paths is performed for all the states. The decoder continues in this way to advance deeper into the trellis, making decisions by eliminating the least likely paths. The early rejection of the unlikely paths reduces the decoding complexity. In 1969, Omura demonstrated that the Viterbi algorithm is, in fact, maximum likelihood. Note that the goal of selecting the optimum path can be expressed, equivalently, as choosing the codeword with the *maximum likelihood metric* or as choosing the codeword with the *minimum distance metric.*

1.3.4 Path Memory and Synchronization

The storage requirements of the Viterbi decoder grow exponentially with constraint length K. For a code with rate $1/n$, the decoder retains a set of 2^{K-1} paths after each decoding step. With high probability, these paths will not be mutual disjoint very far back from the present

decoding depth. All of the 2^{K-1} paths tend to have a common stem, which eventually branches to the various states. Thus if the decoder stores enough of the history of the 2^{K-1} paths, the oldest bits on all paths will be the same. A simple decoder implementation, then, contains *a fixed amount of path history* and outputs the oldest bit on an arbitrary path each time it steps one level deeper into the trellis. The amount of path storage required, u, is

$$u = h\,2^{k-1} \tag{1.10}$$

where *h is* the length of the information bit path history per state. A refinement, which minimizes the value of *h, uses* the oldest bit on the most likely path as the decoder output, instead of the oldest bit on au arbitrary path. It has been demonstrated that a value of *h* of 4 or 5 times the code constraint length is sufficient for near-optimum decoder performance. The storage requirement, In is the basic limitation on the implementation of Viterbi decoders. The current state of the art admits decoders to a constraint length of about K = 10. Efforts to increase coding gain by further increasing constraint length are met by the exponential increased in memory requirements (and complexity) that follows from Equation (1.10).

Branch word synchronization is the process of determining the beginning of e a branch word in the received sequence. Such synchronization can take place without new information being added to the transmitted symbol stream because the received data appear to have an excessive error rate when not synchronized. Therefore, a simple way of accomplishing synchronization is to monitor some concomitant indication of this large error rate, that is, the rate at which the path d metrics are increasing or the rate at which the surviving paths in the trellis merge. The monitored parameters are compared to a threshold, and synchronization is then adjusted accordingly.