6 Trellis Codes

In contrast with the block coding procedures of Chapter 5, trellis encoders generate code symbols for transmission utilizing a sequential finite-state machine driven by the information sequence, perhaps an arbitrarily long sequence. This encoding process will install the key properties of memory and redundancy into the coded symbol stream, as we have seen for block codes. Decoding these codes then amounts to sequentially observing a corrupted version of the output of this system and attempting to infer the input sequence. From a formal perspective, there is no need to block the message into segments of some specific length.

A generic description of a trellis encoder is presented in Figure 6.0.1.[1] Every shift time, indexed by j, a vector of k input symbols (usually bits), designated $\mathbf{u}_j$, is presented to the finite-state encoder. We designate this vector by $\mathbf{u}_j = \left(u_j^{(0)}, u_j^{(1)}, \ldots, u_j^{(k-1)}\right)$, symbols that may have been extracted from an original serial symbol stream written in the form $\left(\ldots, u_j^{(0)}, u_j^{(1)}, \ldots, u_j^{(k-1)}, \ldots, u_{j+1}^{(0)}, u_{j+1}^{(1)}, \ldots\right)$. Corresponding to every new input vector, the encoder produces a vector of n code symbols, $\mathbf{x}_j = \left(x_j^{(0)}, x_j^{(1)}, \ldots, x_j^{(n-1)}\right)$, usually members of the same alphabet, with $n > k$, thus inducing redundancy. We define the input memory (also called the **memory order** m) of the encoder to be the number of

[1]The structure shown is feedback free, or in feed-forward form. Eventually, this will be generalized to allow output feedback.

previous input vectors that, together with the current input $\mathbf{u}_j$, define the current output $\mathbf{x}_j$. Thus, in Figure 6.0.1, the encoder possesses m (vector) delay cells.

The n code symbols produced each shift time may be interfaced to a modulator in various ways. Early applications typically involved binary coding, wherein the n binary symbols were serialized and presented to a binary modulator, say a PSK modulator. We could alternatively equate the output symbols with a character in a larger field and produce a single M-ary symbol from the larger set for each clock interval. The method preferred depends on the application, as we shall see.

We may observe a certain similarity with block coding, particularly in comparison of Figure 6.0.1 with Figure 5.0.1. Specifically, if we set $m = 0$, then the trellis encoding device produces n code symbols strictly defined by k current input symbols, which is the description of a generic block encoder. Thus, we might view trellis coding as a generalization of block coding, where the encoding function is allowed to depend on $m \geq 1$ input blocks prior to the current block. However, in typical block coding practice, n and k would normally be rather large, whereas in trellis coding n and k are typically small, in the range of 1 to 8. Thus, the power of trellis codes derives not from making n and k large, but from adopting a larger memory order m. (There is a contrasting view that puts block coding at the head of the class—once the input sequence to a trellis coder is terminated, as with packetized data, then the entire mapping from input sequences to output sequences can be viewed as a long block code. There is no need, however, to become concerned about the precedence of either category of codes.)

The earliest appearance of such codes is found in Elias [1], who formulated the codes as an alternative to then existing block-structured codes. Elias named the class of codes ***convolutional codes***, since these original codes were linear mappings from input to output sequences obtained by a discrete-time, finite-alphabet convolution of the input with an encoder's impulse response. Other early names were ***recurrent codes*** and ***tree codes***, since the code paths are tied to a tree-structured graph. Presently, the more general name ***trellis codes*** is used to incorporate these classical codes, as well as newer nonlinear modulation/coding approaches that still maintain a finite-state machine description. The term trellis is due to Forney [2], who saw the association of codewords with paths in a regular directed graph reminiscent of a garden trellis.

Our presentation in this chapter begins with the simplest and earliest codes in this class, the convolutional codes. Concepts are introduced through the binary codes, after which extension to nonbinary convolutional codes is simple. We will then present and analyze in detail the performance of the most popular decoding algorithm for these codes, due to Viterbi [3], which is capable of implementing maximum likelihood decoding for general memoryless channels. We will then study the performance analysis of maximum likelihood decoding on AWGN and Rayleigh fading channels. Suboptimal decoding procedures (sequential decoding and threshold decoding), which actually predated the maximum likelihood algorithm, are also described.

The discussion then shifts to two newer classes of trellis codes: the trellis-coded signal-space codes introduced by Ungerboeck [4] and the family of signaling techniques known as continuous-phase modulation, or CPM for short [5]. Both have been practically important in coding for bandwidth-constrained channels, once thought not to be a proper domain for channel coding.

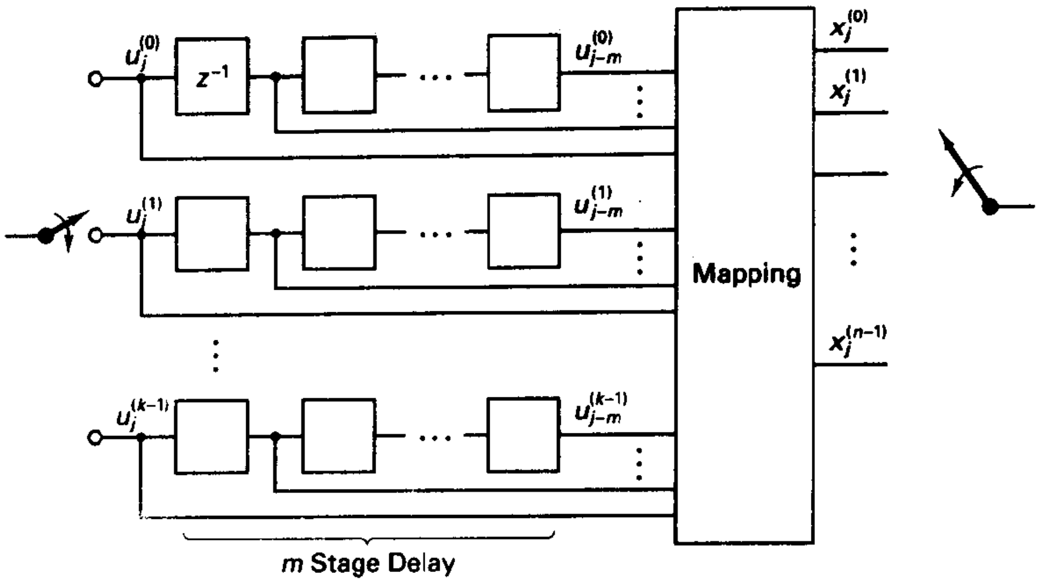


Figure 6.0.1 General trellis encoder with k inputs, n outputs, memory order m.

6.1 DESCRIPTION OF CONVOLUTIONAL CODES

In terms of actual applications of coding techniques to date, convolutional codes (in fact binary convolutional codes) have perhaps been the most noteworthy. This stems from the existence of several decoding possibilities that provide a range of complexity versus performance options and, most importantly, a maximum likelihood sequence decoding algorithm that is readily instrumentable for short memory codes. Ancillary benefits such as simple decoder synchronization have also made convolutional codes a popular choice in practice. Our treatment of convolutional codes begins with the binary case, and the notational framework follows that of Lin and Costello [6].

6.1.1 Binary Convolutional Codes

Several binary convolutional encoders are depicted in Figure 6.1.1.[2] Each encoder is viewed as containing k parallel delay lines, having K_i, $i = 0, 1, \ldots, k-1$, delay cells, respectively, into which we shift message bits k at a time, in parallel. Notice that we allow the delay lines, or shift registers, to have different lengths, which complicates the notation, but this situation is a practical reality. In fact, the ith input line may encounter no delay cells, and $K_i = 0$ in that case. Figure 6.1.1e presents an example of such an encoder. Without loss of generality, we can order the register lengths such that $K_0 \le K_1 \le \cdots \le K_{k-1}$.

Upon each shift of the encoder, $n > k$ output bits are produced by some Boolean function operating on the entire set of inputs residing in the various shift registers. For

[2] Each of these encoders has some claim to optimality under various conditions, as we will progressively develop.

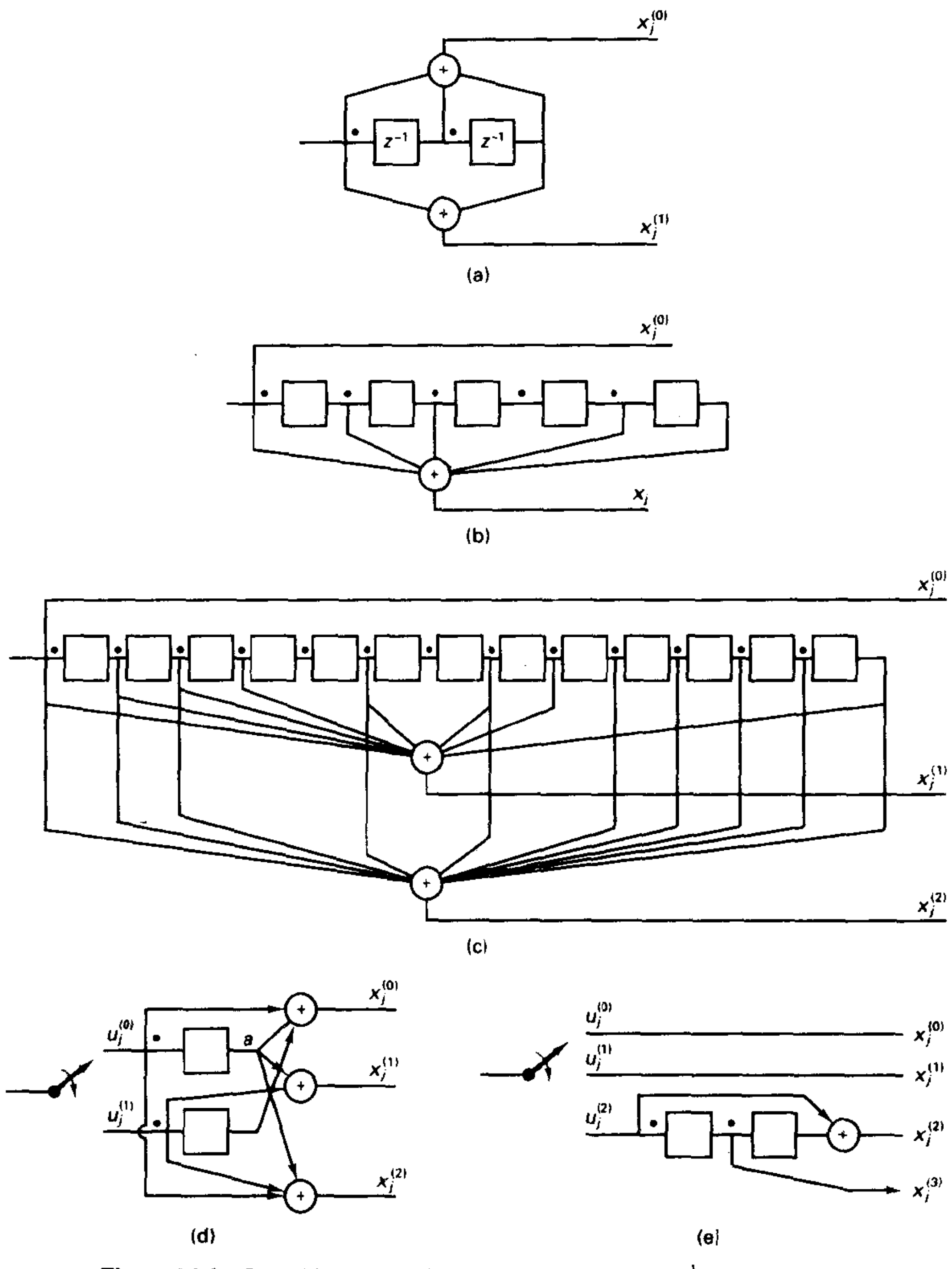


Figure 6.1.1 Some binary convolutional encoders. (a) $R = \frac{1}{2}$, nonsystematic, $m = 2$ encoder; (b) $R = \frac{1}{2}$ systematic, $m = 5$ encoder; (c) $R = \frac{1}{3}$, systematic, $m = 13$ encoder; (d) $R = \frac{2}{3}$, nonsystematic, $m = 1$ encoder; (e) $R = \frac{3}{4}$, nonsystematic, $m = 2$ encoder.

binary convolutional codes, these functions are simple modulo-2 sums involving designated message bits, making the code *linear* over the binary field. Thus, the superposition property holds for the input/output relation, and the all-zeros sequence is a member of every convolutional code. Obviously, some adder connection choices are superior to others, which raises the code design issue, but we will postpone this for now.

The encoders presented in Figure 6.1.1 are ***feedback free***, meaning that the outputs are defined purely in terms of a finite number of consecutive input vectors, as in a finite-impulse-response digital filter. Equivalent forms employing output feedback in the computation are possible, as discussed later in the chapter.

Consistent with the earlier notion of encoder memory, we define the ***memory order*** of the convolutional encoder as

$$m = \max_i K_i, \tag{6.1.1}$$

since a given vector of outputs $\mathbf{x}_j$ depends on the newest input vector $\mathbf{u}_j$ and m previous input k-tuples. It has become conventional to refer to a convolutional encoder with n outputs, k inputs, and memory order m as an ***(n, k, m) convolutional encoder***, although this is not a complete specification of the code.

Another memory-related parameter is the ***encoding constraint length,*** n_E, of the code[3]

$$n_E = n\left[\max_i K_i + 1\right] = n(m+1). \tag{6.1.2}$$

n_E may be interpreted as the maximum span of output bits in a serialized stream that may be influenced, or constrained by, an input bit. Alternatively, n_E can be interpreted as the effective encoding delay measured in channel symbols. As such, n_E plays a similar memory role for convolutional codes that the block-length parameter, n, does for block codes.

Next, we define the ***rate*** of the convolutional code as $R = k/n$ (this rate is dimensionless, that is, message bits/code bit). Typical code rates in practice are $\frac{1}{3}$, $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$, and $\frac{7}{8}$. The normalized code redundancy, $(n-k)/n = 1 - R$, increases as the rate decreases, and relative to an uncoded transmission system using the *same* modulator set, the bandwidth of the coded system typically increases by a factor of $1/R$.

Finally, some of the encoders of Figure 6.1.1 are ***systematic***. This requires that, as for block codes, the information sequence appear explicitly in the coded symbol stream. For example, in Figures 6.1.1b and 6.1.1c we have that $x_j^{(0)} = u_j^{(0)}$. The remaining encoders are nonsystematic. In the case of block codes, we determined that every linear code was equivalent to a code in systematic form, and so systematic-form block codes are the normal operational choice. For convolutional codes, this is not true: ***nonsystematic feedback-free encoders*** are generally preferred, at least in conjunction with maximum likelihood decoding, for reasons tied to the fact that decoding delay may be greater than the effective delay of the encoding device, n_E. This will be further developed as decoding algorithms are presented. It is known, however, that every nonsystematic convolutional encoder has an equivalent code realized by a systematic encoder with output feedback [2],

[3]Beware that there are at least two other prevailing definitions of constraint length, one equivalent to $m+1$ and the other equivalent to $k(m+1)$.

so the choice might really be between systematic encoders with and without feedback. Notice that we are careful to distinguish between encoders and codes; several encoders may produce the same set of code sequences.

To more completely specify convolutional encoders (and their corresponding codes), we return to their convolutional nature and note that the vector of adder outputs $\mathbf{x}_j$ can be represented as a summation of linear block encodings involving $m + 1$ consecutive message blocks, or frames, as shown in Figure 6.1.2. We can compactly represent the entire input/output relation as a convolution operation by using vector notation for input k-tuples and output n-tuples, respectively:

$$\mathbf{x}_j = \sum_{i=0}^{m} \mathbf{u}_{j-i}\mathbf{G}_i, \tag{6.1.3}$$

where $\mathbf{G}_i, i = 0, 1, \ldots, m$, is a $k \times n$ matrix specifying the linear contribution of the ith oldest input vector in the encoding register to the output vector $\mathbf{x}_j$ at a given time j. [In the binary case, these are matrices of 1's and 0's denoting connection or no connection, respectively, of a message bit to an adder, but in the general case, which we shall shortly consider, the matrix entries are in GF(q) and represent a multiplication prior

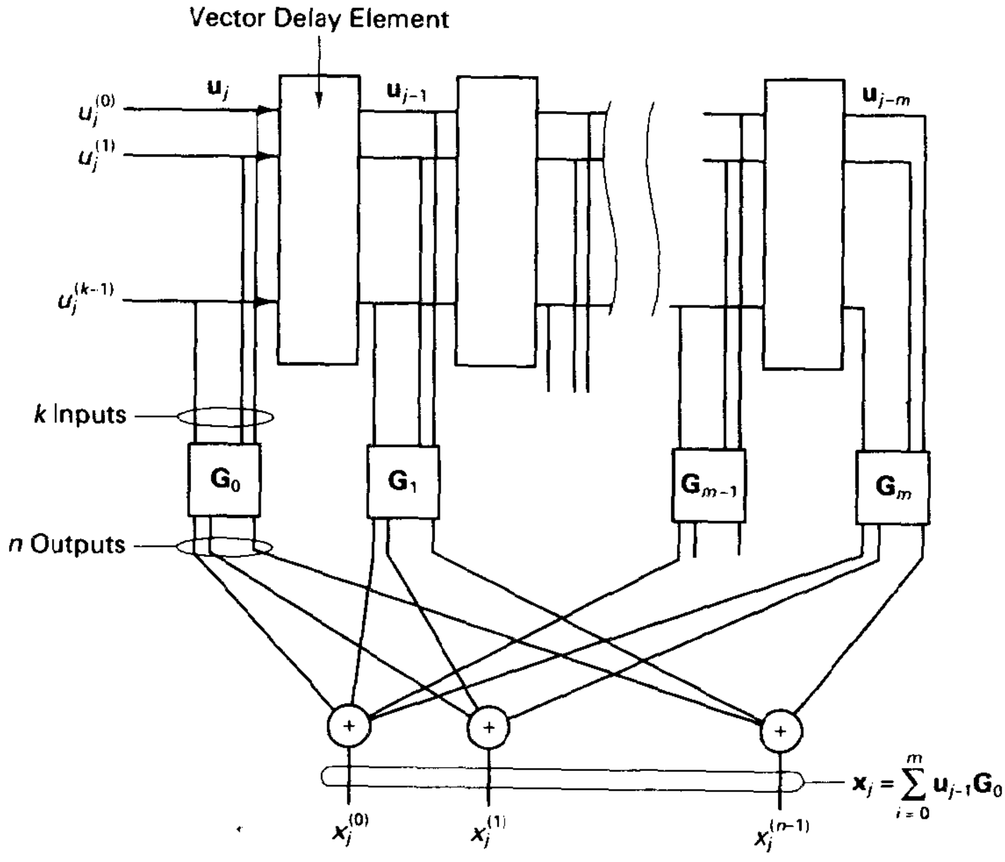


Figure 6.1.2 Representation of convolutional encoder as vector convolution operation.

to final addition]. Specification of the m matrices, $\mathbf{G}_i$, provides a complete description of a linear convolutional encoder.

We may again establish a connection with block coding: clearly if $m = 0$, (6.1.3) reduces to

$$\mathbf{x}_j = \mathbf{u}_j \mathbf{G}_0, \tag{6.1.4}$$

which is the block encoding relation for linear codes, (5.2.1).

To more explicitly describe the $\mathbf{G}_i$ matrices, we may envision a discrete-time impulse response relating the ith input line to the pth output line. We calculate this response by determining the output of the pth adder to the injection of a single 1 symbol on the ith input line, with all other bits zeroed. This impulse response can have duration at most $m + 1$ time units, and we shall denote this impulse response by

$$\mathbf{g}_i^{(p)} = \left(g_{i,0}^{(p)}, g_{i,1}^{(p)}, \ldots, g_{i,m}^{(p)}\right). \tag{6.1.5a}$$

Equivalently, we could express this information in polynomial form by

$$g_i^{(p)}(D) = g_{i,0}^{(p)} + g_{i,1}^{(p)}D + \cdots + g_{i,m}^{(p)}D^m. \tag{6.1.5b}$$

For example, in the rate 2/3 encoder of Figure 6.1.1d, inspection shows that the top input line (denoted as $i = 0$) has an impulse response to the top ($p = 0$) output line given by $(1, 1)$; hence $\mathbf{g}_0^{(0)} = (1, 1)$ and $g_0^{(0)}(D) = 1 + D$. The set of kn impulse responses provides an alternative complete characterization of the convolutional encoder.

The $\mathbf{G}_i$ matrices involved in (6.1.3) are related to these impulse responses by

$$\mathbf{G}_i = \begin{bmatrix} g_{0,i}^{(0)} & g_{0,i}^{(1)} & \cdots & g_{0,i}^{(n-1)} \\ g_{1,i}^{(0)} & g_{1,i}^{(1)} & \cdots & g_{1,i}^{(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,i}^{(0)} & g_{k-1,i}^{(1)} & \cdots & g_{k-1,i}^{(n-1)} \end{bmatrix}, \qquad i = 0, 1, \ldots, m. \tag{6.1.6}$$

Three examples will help clarify the concepts just introduced.

Example 6.1 $R = \frac{1}{2}$, $m = 2$ Code (the Almost Universal Example)

Referring to Figure 6.1.1a, we find a single register having two delay cells,[4] so the memory order is $m = 2$ and the encoding constraint length is $n_E = n(m+1) = 6$. The nonsystematic encoder produces two output symbols according to

$$x_j^{(0)} = u_j + u_{j-1} + u_{j-2}, \qquad x_j^{(1)} = u_j + u_{j-2} \tag{6.1.7}$$

so that $\mathbf{g}_0^{(0)} = (1, 1, 1)$ and $\mathbf{g}_0^{(1)} = (1, 0, 1)$. This in turn implies that the vector convolution relation (6.1.3) becomes

$$\mathbf{x}_j = \sum_{i=0}^{2} \mathbf{u}_{j-i}\mathbf{G}_i \tag{6.1.8a}$$

with

$$\mathbf{G}_0 = [1 \quad 1], \qquad \mathbf{G}_1 = [1 \quad 0], \qquad \mathbf{G}_2 = [1 \quad 1]. \tag{6.1.8b}$$

[4]Some texts would draw this as a register holding three bits.

In tables of encoders, the generator information in the form of $\mathbf{g}_i^{(p)}$ is often expressed in octal representation for compactness. For example, the vector $\mathbf{g}_0^{(0)} = (1, 1, 1)$ would be represented as 7_8, while $\mathbf{g}_0^{(1)} = 5_8$. Still another notation is in polynomial form; the first generator would be listed as $g_0^{(0)}(D) = 1 + D + D^2$.

If the input message is $(u_0, u_1, \ldots) = (110000\ldots)$ and the encoder is initialized with all-zeros contents, then simple computation using (6.1.8) will show[5] that the output code vector sequence is

$$\mathbf{x}_0 = (1, 1), \quad \mathbf{x}_1 = (0, 1), \quad \mathbf{x}_2 = (0, 1), \quad \mathbf{x}_3 = (1, 1), \quad \mathbf{x}_j = (0, 0), \quad j \geq 4. \tag{6.1.9}$$

These bits might be serialized into the output stream $(110101110000\ldots)$ or perhaps, to conserve transmission bandwidth, the two bits produced each unit of time could be mapped to a single QPSK signal.

Example 6.2 $R = \frac{2}{3}$, $m = 1$ **Code**

In the encoder of Figure 6.1.1d, we present $k = 2$ bits per shift time to two registers, each having one delay cell. Consequently, we say the memory order is $m = 1$, and the encoding constraint length is $n_E = 3 \cdot 2 = 6$, from (6.1.2). Study of Figure 6.1.1d shows that the $kn = 6$ impulse responses describing the influence of input bits on various output bits are

$$\begin{aligned} \mathbf{g}_0^{(0)} &= (11), & \mathbf{g}_0^{(1)} &= (01), & \mathbf{g}_0^{(2)} &= (11), \\ \mathbf{g}_1^{(0)} &= (01), & \mathbf{g}_1^{(1)} &= (10), & \mathbf{g}_1^{(2)} &= (10). \end{aligned} \tag{6.1.10}$$

Expressed in (right-justified) octal form, these are 3, 1, 3 and 1, 2, 2, respectively.

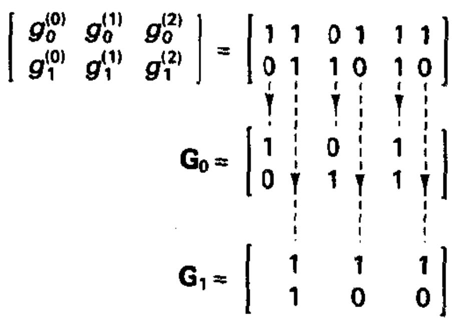
We could also represent the action of the encoder in vector convolution form:

$$\mathbf{x}_j = \sum_{i=0}^{l} \mathbf{u}_{j-i} \mathbf{G}_i, \tag{6.1.11a}$$

where

$$\mathbf{G}_0 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \quad \mathbf{G}_1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{6.1.11b}$$

We remark that the lth row of $\mathbf{G}_i$ is simply the connection vector from the ith stage of the lth register, $l = 0, 1, \ldots, k-1$, to the n adders. Thus, the first row of $\mathbf{G}_0$, (101), implies that the current input bit of the first (top) register is connected to adders 0 and 2, but not to adder 1. A helpful mnemonic for constructing the $\mathbf{G}_i$ matrices is to envision stacking the $\mathbf{g}_i^{(p)}$ vectors into a k by $(m+1)n$ array as shown in Figure 6.1.3; then recognize the $\mathbf{G}_i$ matrices as being obtained by extracting from this array columns spaced by $m + 1$ units.



$\begin{bmatrix} g_0^{(0)} & g_0^{(1)} & g_0^{(2)} \\ g_1^{(0)} & g_1^{(1)} & g_1^{(2)} \end{bmatrix} = \begin{bmatrix} 1\,1 & 0\,1 & 1\,1 \\ 0\,1 & 1\,0 & 1\,0 \end{bmatrix}$
$\mathbf{G}_0 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$
$\mathbf{G}_1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$

Figure 6.1.3 Construction of $\mathbf{G}_i$ matrices from an array of encoder impulse responses, $R = \frac{2}{3}$ encoder of Figure 6.1.1d.

[5]The reader should also verify that this sequence emerges from the device in Figure 6.1.1a.

Example 6.3 $R = \frac{3}{4}$, $m = 2$ **Encoder**

Figure 6.1.1e depicts an $R = \frac{3}{4}$ encoder, which is almost systematic. By writing out the 12 impulse responses, each of length $m + 1 = 3$, and then putting these in array form and extracting columns, we can determine that the $\mathbf{G}_i$ matrices are

$$\mathbf{G}_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad \mathbf{G}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{G}_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{6.1.12}$$

The sparseness of these matrices is reflection of the limited influence of the upper two input lines on the output.

An alternative representation of the input/output relation for a convolutional encoder is provided through the polynomial representation. Specifically, we may write the input sequence as the (vector) polynomial

$$\mathbf{u}(D) = \begin{bmatrix} u^{(0)}(D) \\ u^{(1)}(D) \\ \vdots \\ u^{(k-1)}(D) \end{bmatrix} \tag{6.1.13}$$

where $u^{(i)}(D)$ is the polynomial representation for the sequence on the ith input line. Similarly, we may denote the output (vector) sequence in polynomial form by $\mathbf{x}(D)$. By defining the ***system transfer matrix*** $\mathbf{G}(D)$ as

$$\mathbf{G}(D) = \begin{bmatrix} g_0^{(0)}(D), & g_0^{(1)}(D), & \cdots & g_0^{(n-1)}(D) \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1}^{(0)}(D), & g_{k-1}^{(1)}(D), & \cdots & g_{k-1}^{(n-1)}(D) \end{bmatrix}, \tag{6.1.14}$$

we may then express the input/output relation as

$$\mathbf{x}(D) = \mathbf{u}(D)\mathbf{G}(D). \tag{6.1.15}$$

Thus, for the encoder of Figure 6.1.1a, the response to the input sequence 110000... may be obtained by

$$\mathbf{x}(D) = [1 + D][1 + D + D^2, 1 + D^2] = [1 + D^3, 1 + D + D^2 + D^3], \tag{6.1.16}$$

which corresponds to the two adder output sequences 100100... and 111100.... When multiplexed together, the encoder serial output would be the weight-6 sequence 1101011100..., as earlier calculated.

When $k > 1$, a redrawing of the encoder provides an equivalent single shift register implementation, as shown in Figure 6.1.4a, wherein *bits are shifted k at a time*. Alternatively, we may assume that the input bits are shifted as usual, but that the output vector is computed only every kth shift time. The number of delay elements in this register is $km - 1$, or, if we prefer, the circuit can be regarded as a shift register holding a total of km bits. Figure 6.1.4b illustrates this alternative version of the encoder of Figure 6.1.1e. The message bits residing in this register can be denoted $(u_j^{(0)}, u_j^{(1)}, \ldots, u_j^{(k-1)}, u_{j-1}^{(0)}, u_{j-1}^{(1)}, \ldots)$, and so on. Notice that certain register cells in this representation perhaps hold *don't-care bits* not involved in computation

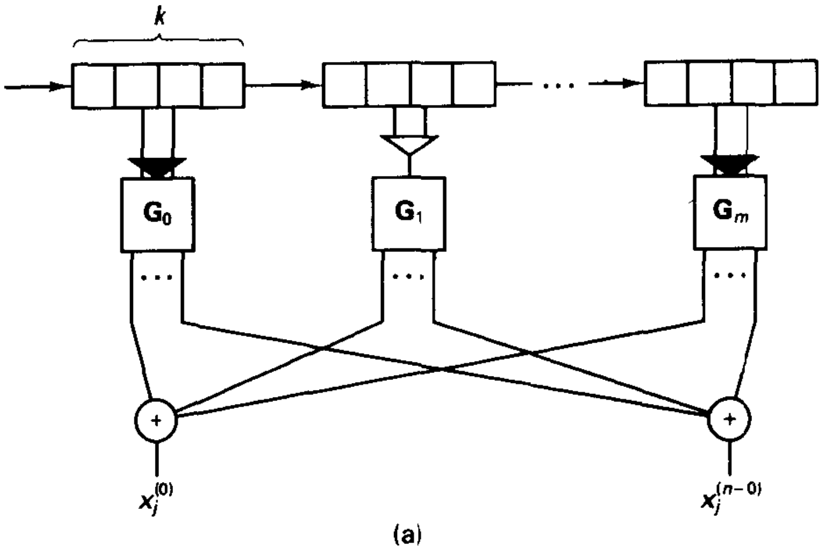


Figure 6.1.4a Single shift register realization of convolutional encoder.

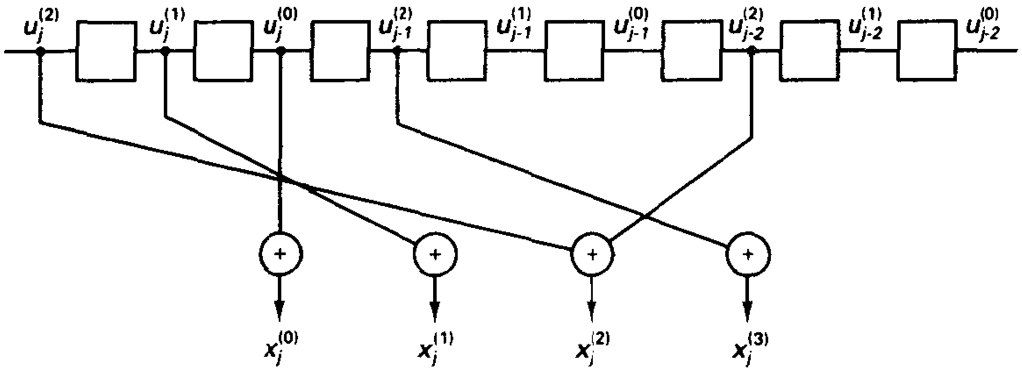


Figure 6.1.4b Single register implementation of encoder of Figure 6.1.1e. Note that outputs are computed every three shift times.

of the current or future encoder outputs. Both representations are common in the literature, and implementation aspects differ little between the parallel or serial register options.

To encode a message sequence, we typically agree to initialize either form of the encoder with 0's and begin shifting the message bits into the encoder k bits at a time. We will assume that the code bits are serialized and modulated using any of a number of binary signaling techniques. In principle, the input message and the output sequence can be arbitrarily long, although typically the encoder is reinitialized periodically by flushing the registers with zero-symbol inputs, say at the initiation of a new packet of data.

Now consider an input vector sequence of finite length L, denoted by $\tilde{\mathbf{u}}_L = (\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{L-1})$. The initial output vectors are $\mathbf{x}_0 = \mathbf{u}_0\mathbf{G}_0$ and $\mathbf{x}_1 = \mathbf{u}_1\mathbf{G}_0 + \mathbf{u}_0\mathbf{G}_1$, and so on. By employing the convolution relation in (6.1.3) and writing the output sequence as a vector (of vectors) $\tilde{\mathbf{x}}_L = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{L-1})$, we have that

$$\tilde{\mathbf{x}}_L = \tilde{\mathbf{u}}_L \mathbf{G}_L, \tag{6.1.17}$$

where $\mathbf{G}_L$ is the $kL \times nL$ matrix (of submatrices) formed according to

$$\mathbf{G}_L = \begin{bmatrix} \mathbf{G}_0 & \mathbf{G}_1 & \mathbf{G}_2 & \cdots & \mathbf{G}_m & 0 & \cdots & 0 \\ 0 & \mathbf{G}_0 & \mathbf{G}_1 & \cdots & \mathbf{G}_{m-1} & \mathbf{G}_m & \cdots & 0 \\ 0 & 0 & \mathbf{G}_0 & \ddots & \vdots & \mathbf{G}_{m-1} & \cdots & \mathbf{G}_m \\ 0 & 0 & 0 & \cdots & \mathbf{G}_0 & \vdots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \mathbf{G}_0 & \cdots & \mathbf{G}_1 \\ 0 & 0 & 0 & \cdots & \cdots & 0 & \cdots & \mathbf{G}_0 \end{bmatrix}_{kL \times nL} \tag{6.1.18}$$

Notice the special banded structure of this generator matrix, wherein rows of the matrix are merely rightshifts of rows above, and the "bandwidth" is $m + 1$ submatrices. Equivalently, if we realize that each submatrix is $k \times n$, we see that the column bandwidth of the generator matrix is $n_E = n(m + 1)$, precisely the encoding constraint length.

Equation (6.1.17) defines a linear relation between input sequences of length kL and output sequences of length nL. [We may even extend this relation to semiinfinite sequences by extending the finite generator matrix in (6.1.18) to a semiinfinite banded matrix.] As mentioned, message transmission with convolutional codes is normally accomplished with a terminating suffix of m k-tuples of 0's to return the encoder to the initial condition; this terminating string affords roughly the same error protection to the last message symbol as to earlier symbols. In this situation, a precise definition of code rate would be $R = Lk/(L + m)n$, which is slightly smaller than our adopted definition of rate of a convolutional code. However, the usual application entails $L \gg m$, whence the true rate approaches the adopted definition, $R = k/n$.

Once a convolutional encoding cycle is terminated in this manner, it is apparent that we have formed a $[(L + m)n, Lk]$ block code, with generator matrix specified by a version of (6.1.18) having L rows and $L + m$ columns (of matrix entries), each row being a single-place right shift of the row above. However, the description of trellis codes and decoding algorithms does not depend strongly on this relationship to block codes. In fact, the banded structure of the encoding matrix suggests that finite memory decoders are feasible for processing arbitrarily long message sequences.

We can now begin to appreciate the convolutional code design problem. Given parameters (R, m), we are interested in making code sequences as distinct as possible, and for now we shall invoke the Hamming distance as a measure of separation between codewords. Since convolutional codes are linear, minimum distance is identical to minimum (nonzero) weight of vectors in the row space of $\mathbf{G}_L$, as for block codes. The difference encountered here is that we have no intrinsic block length to examine and generally must consider the distance structure for sequences of arbitrary length L. Distance descriptions for convolutional codes will be taken up again in Section 6.2. First, however, we generalize our discussion to nonbinary codes and then introduce the essential notions of state diagrams and trellises.

6.1.2 Nonbinary Convolutional Codes

For certain applications, especially those where noncoherent detection is to be performed on fading or jammed channels, it may be advantageous to employ convolutionally coded nonbinary transmission with orthogonal waveforms. The information-theoretic justification for this was supplied in Chapter 4, and we have already encountered the utility of nonbinary block codes, notably Reed–Solomon codes, in Chapter 5.

It is certainly possible to produce coded q-ary ($q > 2$) sequences by treating the outputs of a binary rate k/n convolutional encoder as a single q-ary symbol, with $q = 2^n$. For example, a convolutional encoder for 8-ary signaling could be produced from an $R = \frac{1}{3}$ binary encoder by mapping binary 3-tuples onto GF(8) symbols in any one-to-one manner. However, such a procedure does not directly optimize the nonbinary code's Hamming distance properties; equivalently, good binary Hamming distance codes do not necessarily translate into good q-ary codes by direct mapping.

A rate $R = k/n$ q-ary convolutional encoder of memory order m is formed by the following:

1. k parallel q-ary registers having $K_i, i = 0, 1, \ldots, k-1$, delay elements, where $m = \max K_i$
2. n GF(q) adders
3. Specified multiplier coefficients in GF(q) describing the weighting each message symbol contributes to the pth adder output. As with binary codes, our definition of the encoding constraint length of the code is $n_E = n(m+1)$.

In practical terms, low-rate codes of the form $R = 1/n$, that is, having $k = 1$, are of primary interest, so that typically we shall deal with a single q-ary register of total length $m+1$ (or with m delay cells). Again, the n code symbols at time j are expressed by a convolution relation

$$\mathbf{x}_j = \sum_{i=0}^{m} \mathbf{u}_{j-i}\mathbf{G}_i, \tag{6.1.19}$$

except now the multiplier coefficients in $\mathbf{G}_i$ are members of the field GF(q). For message vector sequences of length L (or serial length kL), the input/output relation is identical to that of (6.1.17) provided that the entries in vectors and matrices are generalized to GF(q).

Example 6.4 Convolutional Codes Over GF(2^k) with $R = \frac{1}{2}$ and Memory Order $m = 1$

In Figure 6.1.5 we illustrate an encoder general to the field GF(2^k), $k > 1$, having rate $\frac{1}{2}$ and $m = 1$. Codes of this form were initially introduced by Viterbi [7] and named ***dual-k codes***, k referring to the size of the *binary* input vector and dual referring to the fact that code symbols are defined by two consecutive input symbols, or two consecutive binary k-tuples.

The element α is taken as primitive in GF(2^k). Several equivalent encoders producing the same set of codeword sequences could be obtained by scaling all multipliers by some nonzero field element.

The encoder could be specified by

$$\mathbf{G}_0 = [1, 1], \qquad \mathbf{G}_1 = [1, \alpha] \tag{6.1.20}$$

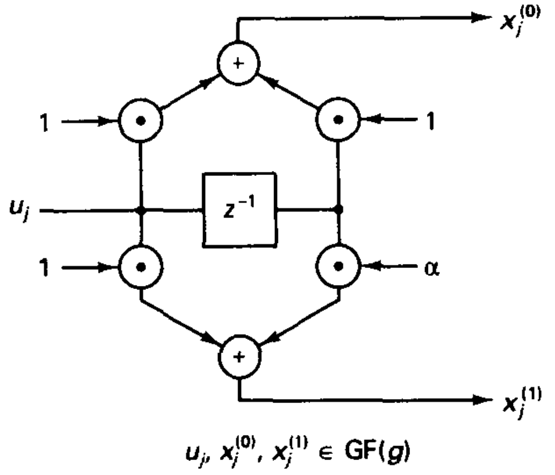


Figure 6.1.5 $R = \frac{1}{2}$ dual-k encoder over GF(q).

or by $\mathbf{g}_0^{(0)} = [1, 1], \mathbf{g}_0^{(1)} = [1, \alpha]$, or in polynomial form by $g_0^{(0)}(D) = 1 + D$, $g_0^{(1)}(D) = 1 + \alpha D$.

A variation on this structure, due to Trumpis [8] is formed by producing a subcode of a general q-ary convolutional code, wherein the input sequence is binary [a subfield of GF(q) assuming that q is a power of 2 and thus the multiplications indicated are well-posed]. Trumpis codes were described originally as having a single binary input and a single ($n = 1$) q-ary output, with $q > 2$, but the concept is easily extended to lower-rate encoders with $n > 1$. The corresponding codes may be viewed as subcodes of q-ary convolutional codes with the same multiplier taps, corresponding to codewords attached to *binary* messages in the larger code. Figure 6.1.6 illustrates a memory-order-2 code for GF(4) having a single input *bit* and a single output symbol per shift time. In such cases the notion of code rate becomes muddled; it is best to explicitly say that the rate is one information bit per q-ary code symbol.

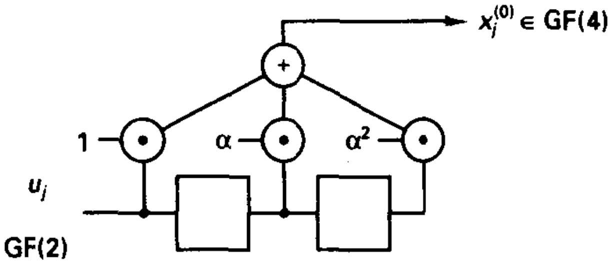


Figure 6.1.6 Binary-to-4-ary convolutional encoder, $R = 1$ bit/code symbol, $m = 2$.

6.1.3 Parity Check Matrices

Convolutional codes also have parity check matrices, denoted $\mathbf{H}$, whose property is that for any encoded sequence $\tilde{\mathbf{x}}$,

$$\tilde{\mathbf{x}}\mathbf{H}^T = \mathbf{0}. \tag{6.1.21}$$

where $\mathbf{0}$ denotes a sequence of zeros of arbitrary length. The parity check matrix plays a less prominent role in the description of convolutional codes, however, and will be of interest to us only in the description of feedback decoders, which produce a syndrome of the error pattern and, in sliding window fashion, correct the received symbols bit by bit. We will argue that for such decoders, systematic encoders are just as powerful as nonsystematic encoders, and thus we will describe parity check matrices for systematic encoders.

With a systematic encoder, the generator matrix is of the form

$$\mathbf{G} = \begin{bmatrix} \mathbf{I}, \mathbf{P}_0 & \mathbf{0}, \mathbf{P}_1 & \mathbf{0}, \mathbf{P}_2 & \cdots \\ & \mathbf{I}, \mathbf{P}_0 & \mathbf{0}, \mathbf{P}_1 & \cdots \\ & & \mathbf{0}, \mathbf{P}_0 & \cdots \\ & & & \ddots \end{bmatrix}, \tag{6.1.22}$$

where the matrices $\mathbf{P}_i$ are $k \times n$ in size, denoting how the parity bits of the code stream are generated. The corresponding parity check matrix is of the form

$$\mathbf{H} = \begin{bmatrix} -\mathbf{P}_0^T, \mathbf{I} & -\mathbf{P}_1^T, \mathbf{0} & -\mathbf{P}_2^T, \mathbf{0} & \cdots \\ & -\mathbf{P}_0^T, \mathbf{I} & -\mathbf{P}_1^T, \mathbf{0} & \cdots \\ & & -\mathbf{P}_0^T, \mathbf{I} & \cdots \\ & & & \ddots \end{bmatrix} \tag{6.1.23}$$

Note that this produces the desired result $\mathbf{G}\mathbf{H}^T = \mathbf{0}$. Both $\mathbf{G}$ and $\mathbf{H}$ matrices are semi-infinite, upper-triangular matrices here, but once the message is terminated to L symbols, the matrices become finite dimensional.

Example 6.5 Parity Check Matrix for Encoder of Figure 6.1.1b

For the systematic $R = \frac{1}{2}$, $m = 5$ encoder shown in Figure 6.1.1b, the first row of the generator matrix is $(1, 1|0, 1|0, 1|0, 0|0, 1|0, 1|\ldots)$, and thus the parity check matrix will be of the banded form (6.1.23) with the first row given by $(1, 1|1, 0|1, 0|0, 0|1, 0|1, 0|\ldots)$. The circuit producing the syndrome sequence from a binary received sequence $\mathbf{r}_j = x_j + e_j$ is shown in Figure 6.1.7.

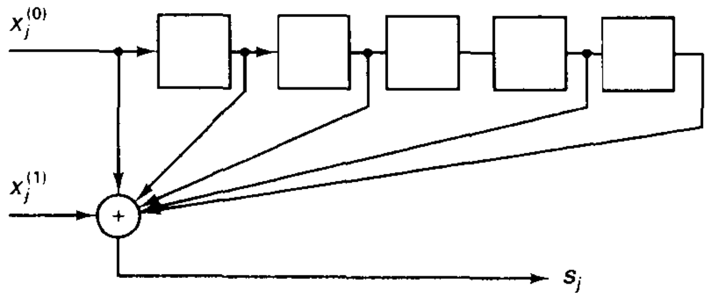


Figure 6.1.7 Syndrome calculation for encoder of Figure 6.1.1b.

6.1.4 Inverse Circuits

For nonsystematic encoders, a pertinent question is whether we can recover the input sequence $\mathbf{u}(D)$ from the code stream, in the absence of errors. This is obviously a desirable property of an encoding system and is in fact a possible question for linear block codes as well, although readily answered in the affirmative if $\mathbf{G}$ has rank k. Here the issues are more subtle; among other things we would like the inverse circuit to have finite, especially small, memory so that the propagation of any possible errors would be limited.

We let $\mathbf{G}^{-1}(D)$ be the system matrix representing a feed-forward (or feedback-free) inverting circuit, which operates on the code stream $\mathbf{x}(D)$. Such a system will be said to be an ***inverse system*** if

$$\mathbf{x}(D)\mathbf{G}^{-1}(D) = \mathbf{u}(D)\mathbf{G}(D)\mathbf{G}^{-1}(D) = \mathbf{u}(D)D^{l}. \tag{6.1.24}$$

That is, it will recover the original message sequence with a delay of l shift times.

Massey and Sain [9] showed that a feed-forward inverse system with delay l exists for rate $R = 1/n$ convolutional codes if, and only if, the impulse response polynomials $\mathbf{g}^{(i)}(D)$ contain no common factor other than D^{l}. This has been generalized to rate k/n codes as follows: a feed-forward inverse exists if and only if all the C_k^n determinants of $k \times k$ submatrices of the system matrix $\mathbf{G}(D)$ contain only D^l as a common factor. This result holds for nonbinary convolutional codes as well.

Example 6.6 Inverse Circuit for a Memory-3 Encoder

Suppose that the encoder is an $R = 1/2$ nonsystematic code specified by $g^{(0)}(D) = 1 + D + D^3$ and $g^{(1)}(D) = 1 + D + D^2 + D^3$. These two impulse response polynomials have no common factor other than $D^0 = 1$, and thus a feed-forward inverse circuit with zero delay exists. The system function for this inverse is

$$\mathbf{G}^{-1}(D) = \begin{bmatrix} 1 + D + D^3 \\ D^2 + D^3 \end{bmatrix} \tag{6.1.25}$$

and the realization is shown in Figure 6.1.8. Message recovery can be verified by writing out the equations defining the code bit streams and substituting in (6.1.24). Actually, simpler delay-1 and delay-2 inverse systems exist, as shown.

We will not be strongly interested in how to realize the inverse circuit—sometimes it is easy by inspection, by writing out the constraint equations and solving, or referring to procedures in [2]. The main importance of this notion of inverses is in determining whether an encoder is catastrophic, as discussed in Section 6.2.

6.1.5 State Diagrams and Trellises

A trellis encoder, by design, corresponds to a k-input, n-output finite-state machine, and we now proceed to develop this formally. Emerging from this is the key concept of a code trellis and, subsequently, an optimal decoding algorithm.

As with any state variable description of a system, we define a ***state vector***, σ_j, to be a (minimal) description of the system at time j that provides an exact specification of future outputs, given future inputs. In essence, we need to find a collection of internal variables that allows unique description of the system evolution over time. For trellis codes as described above, σ_j will lie in a finite state-space denoted by Σ, having size S.

We can abstractly represent the evolution of states and the outputs of the system through a state transition equation and an output equation involving two mappings $f(\cdot)$

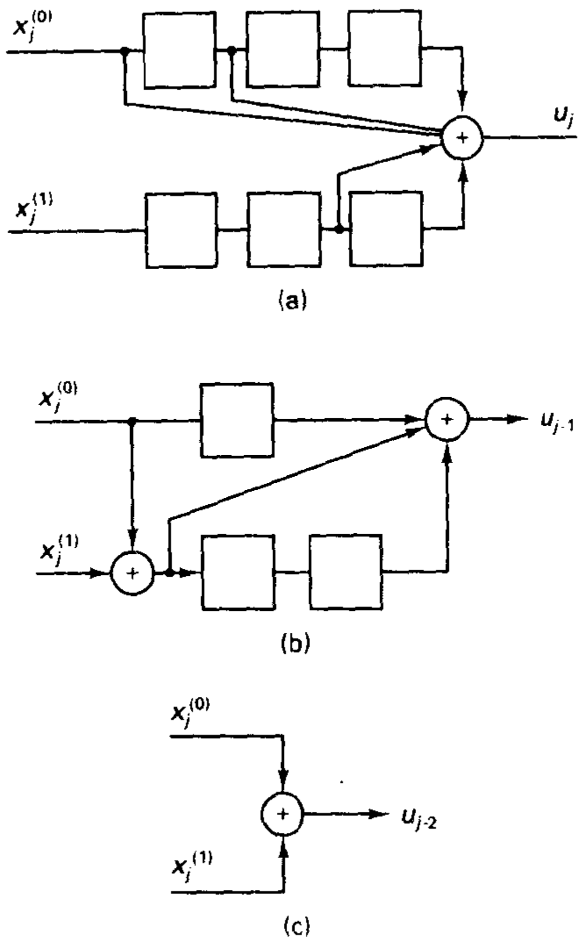


Figure 6.1.8 Inverse circuits for memory-3, $R = \frac{1}{2}$ encoder with $\mathbf{g}^0 = (1101)$, $\mathbf{g}^1 = (1111)$.

and $g(\cdot)$, respectively:

$$\begin{aligned} \sigma_j &= f(\sigma_{j-1}, \mathbf{u}_j) : \quad \text{state-transition rule,} \\ \mathbf{x}_j &= g(\sigma_{j-1}, \mathbf{u}_j) : \quad \text{output rule.} \end{aligned} \tag{6.1.26}$$

These two relations together also imply the code vector $\mathbf{x}_j$ is uniquely specified by the state-transition $\sigma_{j-1} \to \sigma_j$.

As for dynamical systems in general, there is no unique choice of state vector. The obvious (and standard) choice for feed-forward convolutional encoders is to employ a sufficient number of message symbols residing in the encoding register(s). We will be able to easily identify for any feed-forward encoder like those of Figure 6.1.1 a set of ν symbol positions, which together with arbitrary future input vectors, allow us to exactly specify the output vectors $\mathbf{x}_j$ for all j. The number of symbol positions that is necessary (and sufficient) for this specification is

$$\nu = \sum_{i=0}^{k-1} K_i, \tag{6.1.27}$$

sometimes called the ***total memory*** of the encoder. Thus, the state vector can be expressed as

$$\begin{aligned}\boldsymbol{\sigma}_j &= (\sigma_{j_0}, \sigma_{j_1}, \ldots, \sigma_{j_{\nu-1}})\\ &= \left(u_j^{(k-1)}, \ldots, u_{j-K_{k-1}}^{(k-1)}, u_j^{(k-2)}, \ldots, u_{j-K_{k-2}}^{(k-2)}, \ldots, u_j^{(0)}, \ldots, u_{j-K_0}^{(0)}\right).\end{aligned} \tag{6.1.28}$$

which is just an ordered collection of bits appearing on the input lines to the encoder.

The size of the state space is $S = q^{\nu}$, since each element of the state vector is a q-ary information symbol, and all combinations of such symbols are reachable states. Thus, in (6.1.26), the relation $f(\cdot)$, once the input $\mathbf{u}_j$ is specified, maps a set of size q^{ν} onto itself, while $g(\cdot)$ maps the state space Σ onto a set of size q^n.

In Figure 6.1.1 the elements of the state vector for the various binary codes have been identified with a dot. The order in which we put these into a state vector, $\boldsymbol{\sigma}_j$, is not mathematically important, but a naturally ordered choice corresponding to position in the original serial stream is typically used. Thus, in the encoder of Figure 6.1.1a, we define

$$\boldsymbol{\sigma}_j = (u_j, u_{j-1}), \tag{6.1.29a}$$

while for the $R = \frac{2}{3}$ encoder of Figure 6.1.1d, we define the state as

$$\boldsymbol{\sigma}_j = \left(u_j^{(0)}, u_j^{(1)}\right) \tag{6.1.29b}$$

These are both four-state encoders, and we often refer to the corresponding codes as four-state codes. Similarly, the state for the q-ary codes of Example 6.4 will be taken as $\boldsymbol{\sigma}_j = u_j$, the single q-ary symbol residing at the left-hand end of the encoding register; the dual-k encoders thereby have $S = q^1$ states.

An alternative means of labeling states represents each state as an integer in a q-ary number system; that is, we label a state by S_p if

$$p = \sigma_{j_0} q^{\nu-1} + \sigma_{j_1} q^{\nu-2} + \cdots + \sigma_{j_{\nu-1}}. \tag{6.1.30}$$

Thus, S_0 will frequently be referred to as the *all-zeros* state of the encoder.

According to (6.1.26), since there are q^k input vectors at each shift time, every state $\boldsymbol{\sigma}_j = S_p$ must transition to one of q^k next states, simultaneously producing n code symbols. The evolution of the finite-state system over time is conveniently summarized by a ***state-transition diagram***, shown in Figure 6.1.9 for the code of Example 6.1 having R=1/2 and $m = \nu = 2$. Here, each state may transition to two next states, and each such transition is indicated by an arc, or edge, in the graph. Arcs are labeled by strings representing the input sequence and output sequence on the given transition. The encoder diagrams of Figure 6.1.1 and state-transition diagrams contain the same information, and either provides a complete description of a convolutional encoder.

It is possible that a given state can transition to a certain next state under the action of more than one input vector, and this *parallel transition* happens precisely when certain components of the input sequence are not contained in the definition of the state vector. (An example of this is found in Figure 6.1.1e, where each of the four states will transition to only two distinct next states.) In general, if there are k' input symbol lines that have no memory cells assigned them, and thus these k' lines do not affect the state vector, then the state-transition diagram will exhibit parallelism of order $q^{k'}$ in its transitions between states. Such codes are typically not of interest for applications involving q-

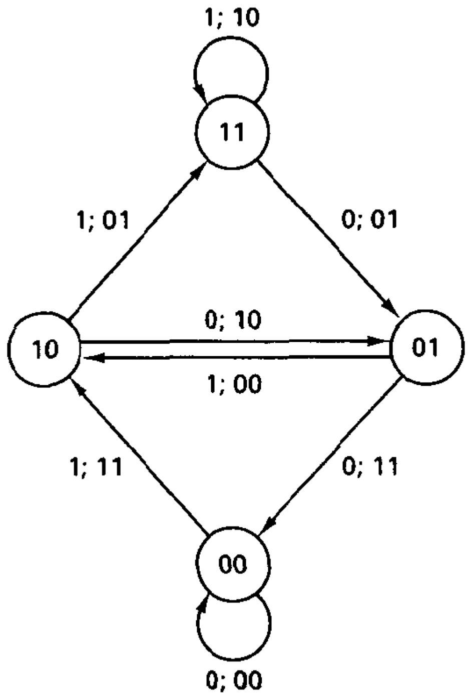


Figure 6.1.9 State transition diagram for $R = \frac{1}{2}$, $m = 2$ code.

ary symmetric channels, since the minimum Hamming distance between distinct code sequences is usually inferior to that achievable when all input lines influence the state. However, for trellis coding on the band-limited Gaussian channel (to be discussed later in the chapter), such codes are frequently optimal in the Euclidean distance sense when properly combined with modulation.

As a related comment, it should be clear that distinct single-step state transitions are capable of producing equivalent output code symbols. For example, in Figure 6.1.9, the transitions $\sigma_j = (00) \rightarrow \sigma_{j+1} = (00)$ and $\sigma_j = (01) \rightarrow \sigma_{j+1} = (10)$ both produce the same output pair, $\mathbf{x}_{j+1} = (00)$. Over time, the corresponding histories will differ.

Now suppose that we initialize the encoder in the all-zeros state (or any other known state for that matter) and track the code sequences produced by various input sequences. After one step, there are q^k state sequences, each with a distinct code sequence, after two steps, q^{2k} sequences, and so on. These can be conveniently represented in the form of a regular graph known as a ***trellis diagram***, which has $S = q^\nu$ states, or nodes, each with q^k branches entering and exiting. Time is understood to increase from left to right in trellis diagrams. The trellis diagram is full after the first m stages and thereafter replicates itself. We label each branch with an information string and a code string in the form $(\mathbf{u}_j \mid \mathbf{x}_j)$. Figure 6.1.10 provides the trellis for the $R = \frac{1}{2}$, $m = 2$ code. Readers new to this material should try to reproduce parts of this diagram, in particular verifying certain state transitions and code sequences.

We may use the trellis diagram to trace the state routing associated with any given message sequence and to determine the associated code sequence as well. For example, with the $R = \frac{1}{2}$, $m = 2$ binary code, the message 110000... produces the trajectory shown in bold lining on Figure 6.1.10, and by reading the code symbols attached to

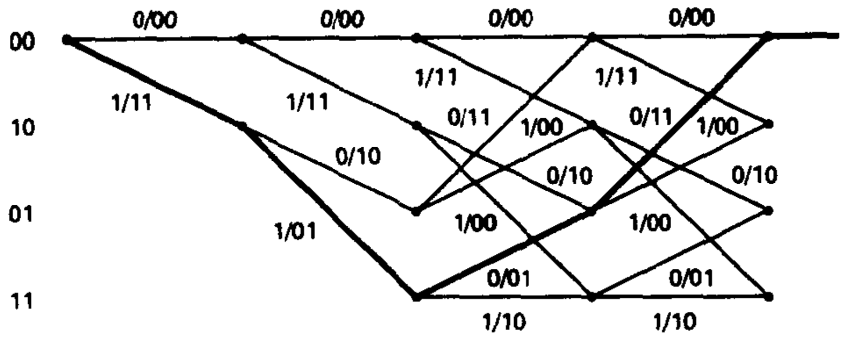


Figure 6.1.10 Trellis for four levels, $R = \frac{1}{2}$, $\nu = 2$ code. Heavy line denotes route for message $\mathbf{u} = (11000\ldots)$.

each branch, we find that the code sequence is $(110101110000\ldots)$, as earlier claimed in Example 6.1.

The concept of trellis ***splits and merges*** will become crucial in our study of maximum likelihood decoding. Specifically, we will be interested in the event that the encoder follows a certain path in the trellis, but that the decoder selects another incorrect path that at some stage departs from the correct path and later becomes common with the transmitted path. Obviously, the code symbols attached to these two message sequences will differ only over the unmerged span, and for a memoryless channel, only this interval of time is useful in discriminating between the two corresponding messages. If we let

$$m_0 = \min_i K_i, \tag{6.1.31}$$

then it is easily seen that two paths that split, or diverge, in their state sequence at time j can remerge *as early as* time $j + m_0 + 1$. (*These events will be called shortest detours.*) As a corollary of this, whenever parallel transitions exist in the trellis, that is, $k' > 0$, then $m_0 = 0$, and the shortest error events are one-step events.

Example 6.7 State and Trellis Diagram for $R = \frac{2}{3}$, $m = 1$ Encoder

Figure 6.1.11 depicts the state diagram and trellis for the $R = \frac{2}{3}$, $m = 1$ encoder of Figure 6.1.1d. Since two bits in the encoding register constitute the state vector, we have $\nu = 2$. We again designate the states by binary strings, that is, $\sigma_j = (u_j^{(0)}, u_j^{(1)})$, or using $S_0, S_1, \ldots, S_3$. Notice that because $k = 2$ bits enter at each update time each state may transition to all four states in one time step, and thus the trellis diagram is fully connected. The diagrams of Figure 6.1.11 are purposefully left incomplete for the reader to complete.

For a general trellis code, we recognize that there are precisely q^{kL} routes from any starting state through a trellis of length L levels, each possessing a unique code sequence of length Ln. (Again, we typically follow the message with a terminating string of m 0's so that the number of routes is still q^{kL}, but the code sequences are of length $n(L + m)$.) This should make it obvious that brute-force maximum likelihood decoding that exhaustively evaluates the likelihood of every trellis route is totally impractical for reasonably sized messages. Fortunately, such complexity can be avoided, as seen in Section 6.3.

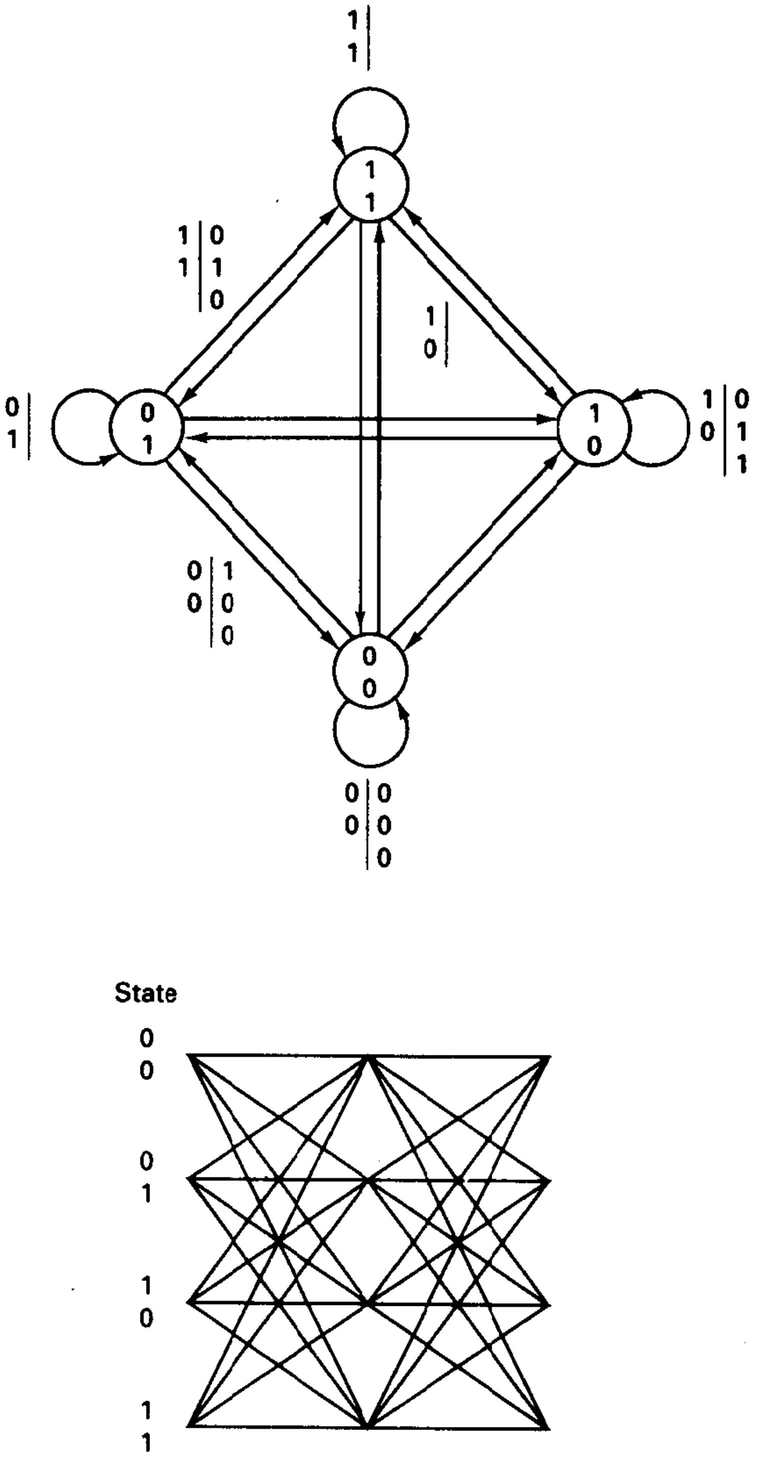


Figure 6.1.11 Partial state transition diagram and trellis diagram for $R = \frac{2}{3}$, $m = 1$, $\nu = 2$ code.

6.2 HAMMING DISTANCE MEASURES FOR CONVOLUTIONAL CODES; VARIOUS GOOD CODES

6.2.1 Distance Definitions

As for block codes, the Hamming distance structure of a convolutional code is of primary importance to system performance when the modulator/channel/demodulator produces a q-ary symmetric channel. In other situations, Hamming distance will generally remain a relevant measure for the code. Because convolutional codes are linear codes, study of the Hamming distance structure reverts to study of the weight spectrum of the code. For convolutional codes, however, there is no intrinsic block length over which to measure distance (or weight), and we can plainly see that the minimum distance between codewords corresponding to input sequences of length kL symbols (or L shift operations of the encoder) will depend in general on L.

More specifically, consider the encoder to be initialized at time $j = 0$ in the zero state; that is, $\sigma_0 = (0000\ldots0)$, also denoted S_0. With no loss of generality, we consider the all-zeros sequence to be the transmitted message and seek to find the minimum weight among all sequences for which the message vector is not $\mathbf{0}$ in position $j = 0$. Thus, we define

$$d_c(L) = \min_{\tilde{\mathbf{u}}_L : \mathbf{u}_0 \neq \mathbf{0}} \operatorname{wt}(\tilde{\mathbf{x}}_L), \qquad L = 1, 2, \ldots, \tag{6.2.1}$$

as the ***column distance function*** of the code,[6] where again $\tilde{\mathbf{u}}_L = (\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{L-1})$ is a message sequence, and $\tilde{\mathbf{x}}_L$ is the corresponding output vector sequence as defined by (6.1.17). Inspection of (6.1.18) reveals that $d_c(L)$ is just the minimum weight produced by the first L columns (of submatrices) of $\mathbf{G}$.

A plot of the column distance function versus L, shown for a typical code in Figure 6.2.1, is nondecreasing in the depth parameter L, since the Hamming weight of sequences is a sum of nonnegative quantities. It is, however, certainly possible for the column distance function at depth L to equal that at depth $L - 1$; that is, the distance profile can have plateaus in the function.

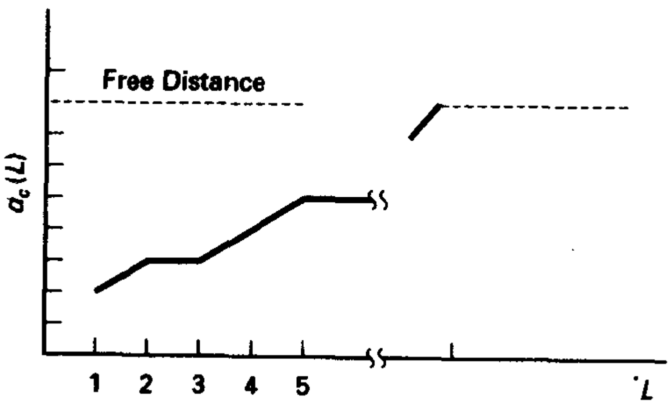


Figure 6.2.1 Typical column distance function showing minimum weight as function of trellis depth.

[6]The term *distance profile* is also used to designate this function.

The distance profile $d_c(L)$ will gradually climb until it reaches a value above which it no longer increases. We define this distance to be the ***free distance***, d_f, of the convolutional code:

$$d_f = \lim_{L\to\infty} d_c(L) \tag{6.2.2}$$

The limit will exist since the column distance function is nondecreasing, and the distance is upper-bounded by the maximum weight obtainable from an input vector of the form $\tilde{\mathbf{u}}_L = (\mathbf{u}_0 \neq \mathbf{0}, \mathbf{0}, \mathbf{0}, \ldots)$, that is, $(m+1)n$. Thus, d_f is the minimum Hamming weight produced among all nonzero input sequences of arbitrary length, whether remergent with the all-zeros path or not. Usually, excepting cases discussed shortly, this free distance will correspond to the minimum weight produced by a path that splits and remerges with the all-zeros path, since we would expect sequences that do not remerge to keep accruing distance. Thus, we can normally take the latter as a definition of free distance:

$$d_f = \min_{\tilde{\mathbf{u}}_L \in I} \text{wt}(\tilde{\mathbf{x}}_L), \tag{6.2.3}$$

where I is the set of message sequences having a nonzero input vector in the first position and with m trailing $\mathbf{0}$ input vectors. This class of sequences covers the error paths diverging from the all-zeros path and remerging at some later time. There can be multiple error events having this minimum weight; usually, some of these are shortest-length detours in the trellis.

Free distance is the fundamental limitation on code performance, for if we imagine a decoder with arbitrarily long memory that could evaluate likelihoods of *all* sequences, free distance is the minimum distance between any transmitted sequence and any other hypothesized sequence. We also desire that the number of sequences having this minimal distance and the associated number of nonzero information symbols be small as well.

Another parameter gleaned from the column distance function is the ***minimum distance***, $d_{\min}$, of the code, defined as the column distance evaluated at depth $m+1$, that is, $d_c(m+1)$. The terminology unfortunately leaves room for confusion and traces to early decoders for convolutional codes that used a sliding window of observations of length $m+1$ blocks, for which the minimum Hamming distance between all sequences over this window length was the primary determinant of performance. The ideas will be taken up in Section 6.5 under feedback decoding.

Example 6.8 Distance Parameters for Code of Example 6.1

For the $R = \frac{1}{2}$ code of Example 6.1, it is readily seen that the only length-1 input that differs from the all-zeros sequence, $\tilde{\mathbf{u}}_L = \mathbf{u}_0 = 1$, produces Hamming weight of 2. (This is because both mod-2 adders include this bit as an input.) Both input sequences of the form $(1x)$ produce Hamming weight of 3. Furthermore, the input sequence 101 also produces weight 3 at depth $L = 3$, so the distance profile does not climb between $L = 2$ and $L = 3$. Further study of the distance profile produces the result of Figure 6.2.2. Notice that the free distance is $d_f = 5$ for this code, which incidentally is the largest attainable among all binary codes of $R = \frac{1}{2}$ and $\nu = 2$.

Notice also that $d_{\min} = d_c(4) = 4$ for this encoder; an implication is that use of a decoder with delay, or memory, of 4 units of time is not adequate to exploit the complete distance between pairs of sequences.

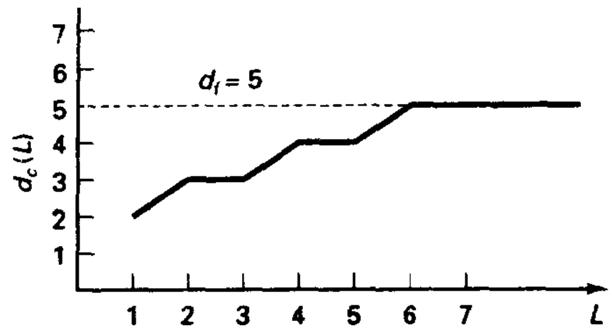


Figure 6.2.2 Distance profile for $R = \frac{1}{2}$, $\nu = 2$ code with $d_f = 5$.

We may determine the column distance function in several ways. First, we can trace all required paths of length L in the trellis, although this is really only a one-time learning exercise. Second, we may consider the generator matrix truncated to width L and determine the minimum weight among vectors in the row space of $\mathbf{G}_L$. Finally, as in Example 6.8, we can simply inject various input sequences into the encoder and measure the weight of the output. We will find a more efficient method later in the chapter, following a discussion of optimal decoding.

There is an important issue attached to whether all sequences remaining unmerged with the all-zeros path continue to grow in distance. It is possible that this will not be the case, and such encoders are known as ***catastrophic***. Catastrophic encoders have loops in their state-transition diagrams containing nonzero information symbols and that do not visit the state S_0, but that accumulate zero code symbol weight. The term catastrophic is apt, for it refers to the potential for a decoder to be diverted onto an incorrect trellis path by a finite-length span of channel disturbance, and, even with a subsequent error-free channel, remain on this incorrect (and nonremergent) path indefinitely. This occurs because the incorrect path is at zero incremental distance from the correct path and thus survives in the decoder's evaluation indefinitely, since it appears perfectly normal! This infinite error propagation possibility must be avoided in the design of a code. As shown in [9], catastrophic error propagation is precisely tied to the lack of a feed-forward inverse system, and so the tests cited in the previous section for existence of an inverse provide tests for catastrophicity: with $\mathbf{G}(D)$ denoting the transfer function matrix of the encoder, which again is a $k \times n$ matrix of polynomial impulse responses, if the determinants of the submatrices of size $k \times k$ formed from this larger matrix have greatest common divisor $D^l, l \geq 0$, the encoder is noncatastrophic. Fortunately, the fraction of encoders that are catastrophic is rather small, but a possible design must be checked nonetheless. Systematic feedback-free encoders are automatically noncatastrophic, which can be argued from the definition or by applying the greatest-common-divisor test.

Example 6.9 A Catastrophic $R = \frac{1}{2}, \nu = 2$ Encoder

Consider the encoder of Figure 6.2.3, where we also show the state-transition diagram. It may be seen that the self-loop at the state $\sigma = (11)$ has weight zero, and the code is thus catastrophic. Equivalently, both encoder polynomials $1 + D$ and $D + D^2$ possess common factor $1 + D$. The corresponding distance profile defined by (6.2.1) reaches a plateau at $L = 2$ and never grows further, due to the input path $(11111\ldots)$. If we merely checked all input sequences that remerge with the all-zeros sequence, that is, they return to the zero-state-after some number of shifts, we would conclude from (6.2.3) that $d_f = 4$. However,

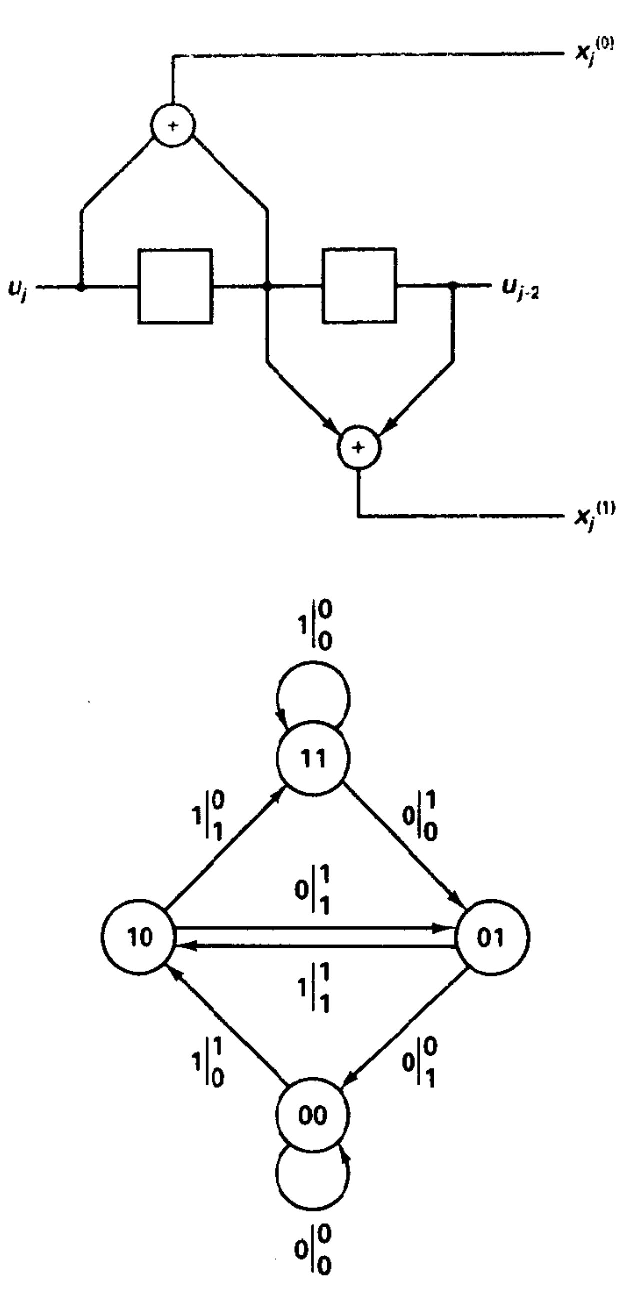


Figure 6.2.3 Catastrophic $R = \frac{1}{2}$, $\nu = 2$ encoder and state transition diagram.

the performance of any decoder, even a maximum likelihood decoder with infinite delay, will be limited by these paths with distance 2. In any case, the code should not be employed in an application with long messages due to the potential for arbitrarily long error propagation.

For noncatastrophic encoders, sequences on trellis paths that remain *nonremergent* with the all-zeros path steadily accumulate Hamming distance, and this eventually will exceed d_f defined by (6.2.3). For example, with the $R = \frac{1}{2}$, $m = 2$ encoder of Figure 6.1.1a, the input sequence (11111111...) never remerges with the all-zeros path in the trellis, and the distance increments by one unit for each input. The depth at which *all* still-unmerged sequences *exceed* d_f is known as the ***decision depth***, N_D of the code.

For the code of Example 6.1, it may be found that the decision depth is 8, since the input sequence (10101010xxxxx) does not accumulate weight 6 until after eight levels. Decision depth plays a role in determining the proper amount of delay needed by a (virtually) maximum likelihood decoder.

Having characterized the distance behavior of convolutional codes, we can now formulate definitions of optimal codes. Foremost in practical importance is the free distance, and we say a code is ***optimal free distance (OFD)*** if a certain code has the largest free distance among all codes of identical rate, total memory ν, and alphabet size. [It is possible that several codes will have the same free distance, in which case we would select based on the number of sequences having weight d_f, and if ties still remain, on the behavior of distance beyond d_f (see Exercise 6.2.3).]

Free distance is the principal figure-of-merit when maximum likelihood sequence decoding is employed, as demonstrated in Section 6.4. However, this decoding technique is only feasible for reasonably small total memory ν. Suboptimal decoding techniques are more influenced by other aspects of the distance profile. Specifically, feedback decoders estimate the message sequence in symbol-by-symbol fashion, with a delay of $L_D \geq m+1$ stages in the decoder, and the important distance measure is the column distance at this depth, $d_c(L_D)$. For sequential decoding procedures, which amount to nonexhaustive examination of the trellis, the amount of decoder computation (which is variable in these algorithms) is minimized, on average, by codes having largest rate of growth in the distance profile. An ***optimal distance profile (ODP)*** code is one whose distance profile is superior to that of all other codes of equivalent rate and memory order. Rather than compare two encoders' complete column distance functions, it is customary to examine only the distance function to depth $m+1$. A given code with profile $\{d_c(j)\}$ is said to be superior to another code with profile $\{d_c'(j)\}$ if for some depth $1 \leq p \leq m+1$

$$d_c(j) = d_c'(j), \qquad 0 \leq j \leq p, \tag{6.2.4}$$

$$d_c(j) > d_c'(j), \qquad j = p. \tag{6.2.5}$$

This definition of dominance reflects the fact that early growth in the distance profile is important. Figure 6.2.4 illustrates profiles for three codes A, B, and C. Code A is

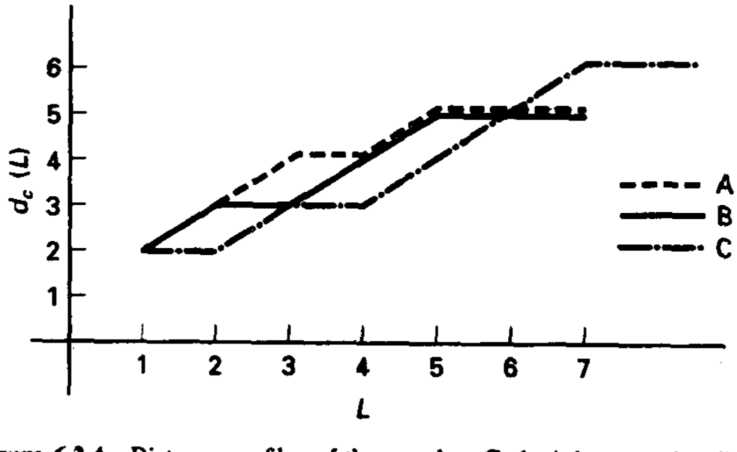


Figure 6.2.4 Distance profiles of three codes. Code A has superior distance profile, but code C has greater free distance.

preferred to code B under the ODP criterion, that in turn is preferred to code C. Note that a code that is superior in the ODP sense may actually be inferior in the OFD sense. We will return to ODP codes at the end of this section.

6.2.2 Bounds on Free Distance

Just as for block codes, we may be interested in bounds on distance properties of codes, in case we either do not have available time to search for the absolutely best code or need to know how good (or bad) a certain code is. Upper bounds on free distance for convolutional codes are rather easily obtained. A simple upper bound is

$$d_f \le (m+1)n. \tag{6.2.6}$$

The bound follows from the fact that an input message with a single nonzero vector, that is, $\mathbf{u}_0 \ne \mathbf{0}$, can produce at most $(m+1)n$ output symbols that are nonzero due to the encoder's input memory of m shifts. This simple bound is never tight for binary codes (except for the degenerate case $m = 0$), but is frequently tight for nonbinary convolutional codes, as will be seen.

A better upper bound is obtained by appeal to the Plotkin bound of Section 5.3 to upper-bound the minimum distance for length-L messages, followed by m 0's. The free distance is then obtained by minimizing over L. This argument was first made by Heller [10] for binary convolutional codes. Consider input messages with $L \ge 1$ vectors, such that $\mathbf{u}_0 \ne 0$ and the input message produces a remerger with the all-zeros path at depth $L+m$. Correspondingly, consider the set of codewords of length $n(L+m)$ symbols generated by a matrix of the form (6.1.17). Recall that in this linear code each alphabet symbol will appear in each position of the codeword an equal number of times in a listing of all codewords (Exercise 5.2.2). The total weight of all codewords of message length L will then be exactly $q^{Lk}[n(L+m)(q-1)/q)]$, and the average weight of the nonzero codewords is an upper bound on the minimum weight of nonzero codewords. Thus, we have that

$$d_f \le \min_{L=1,2,\ldots} \left\lfloor \frac{q^{Lk-1}}{q^{Lk}-1}(L+m)n(q-1) \right\rfloor . \tag{6.2.7}$$

We have indicated with the floor function the fact that we should take the integer part of nonintegral results, since Hamming distance must be integer valued. (It is possible to further tighten the bound in some cases, as shown by Heller [10].)

This upper bound is a function of all the code parameters q, k, n, and m and is shown in Figure 6.2.5 for $R = \frac{1}{2}$ binary and 8-ary codes as a function of memory order m. Also shown for comparison is the bound of (6.2.6), which is surprisingly tight for 8-ary codes, but never tight for binary codes. This behavior is fundamentally linked with the ability of encoders over larger fields to avoid 0's in the output.

Further discussion of upper and lower bounds on free distance of convolutional codes may be found in Costello [11] and Heller [10], where it is shown that systematic codes are inferior to nonsystematic codes in the free distance sense. This supremacy of nonsystematic codes (in the free distance sense) is due to the decoder's ability to evaluate codewords with delay exceeding that of the encoder constraint length, n_E. This is further developed in Exercise 6.2.13.

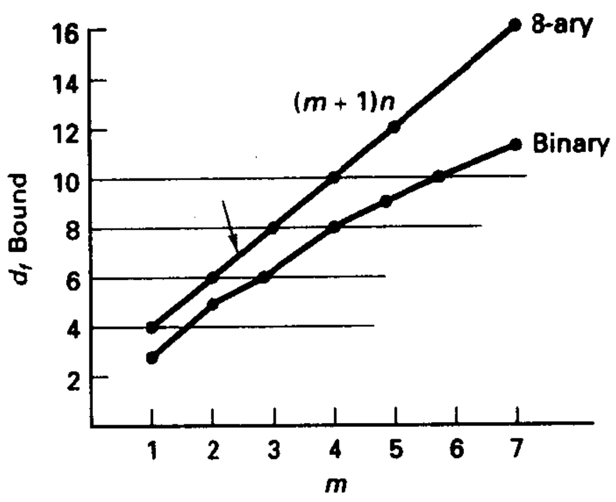


Figure 6.2.5 Upper bounds to d_f for $R = \frac{1}{2}$ binary and 8-ary codes.

6.2.3 Optimal Free Distance Codes

In this section, we summarize some known results obtained from computer search for convolutional codes that are optimal in the Hamming distance sense under the criteria of free distance. In contrast with the development of block codes where good codes have generally been determined from algebraic constructions, location of good convolutional codes has been notably devoid of theory; all the codes reported here have been obtained by computer search.

We reiterate that free distance is the principal figure of merit when maximum likelihood decoding is performed, and we emphasize that Hamming distance is the distance measure here, which will be the appropriate choice whenever we utilize binary modulation, such as binary PSK, or more generally when symmetry of the modulation set and channel implies that maximizing Hamming distance is the obvious criterion. Such a case is when q-ary orthogonal signaling is utilized, combined with q-ary convolutional coding. Because of the orthogonal signal-space constellation, Hamming distance on the q-ary alphabet is still the relevant distance measure. It might seem that such is always the case, but consider the design of codes for an 8-PSK constellation for the AWGN channel. Euclidean distance between codeword sequences is the important distance here, and codes designed for maximal Hamming distance do not directly produce best Euclidean distance codes. We will return to this important issue later.

Tables 6.1 through 6.4 provide data for binary OFD codes with $R = \frac{1}{3}$, $R = \frac{1}{2}$, $R = \frac{2}{3}$, and $R = \frac{3}{4}$ with various values of ν or, equivalently, for various state complexities. Code generators are presented for several form encoders in octal format, right justified, along with the free distance and the upper bound on free distance of (6.2.7). We remark that the bound of (6.2.7) is usually achievable, but the $(m+1)n$ bound is never achieved. Data for these tables are extracted from references [12] through [15]. Convolutional codes with rates closer to 1 are of practical interest as well. These are typically realized by puncturing a lower-rate code, as discussed in Section 6.2.4. Lower-rate codes, should they be of interest, can be obtained by repeating code symbols produced by these encoders, although these generally are slightly suboptimal. For example, an $R = \frac{1}{4}$ code is

TABLE 6.1 $R = \frac{1}{3}$ OFD CODES

ν	Generators	d_f	Bound
2	5, 7, 7	8	8
3	13, 15, 17	10	10
4	25, 33, 37	12	12
5	47, 53, 75	13	13
6	133, 145, 175	15	15

TABLE 6.2 $R = \frac{1}{2}$ OFD CODES

ν	Generators	d_f	Bound
1	2, 3	3	3
2	5, 7	5	5
3	15, 17	6	6
4	23, 35	7	8
5	53, 75	8	9
6	133, 171	10	10
7	247, 371	10	11

TABLE 6.3 $R = \frac{2}{3}$ OFD CODES

ν	Generators	d_f	Bound
1	2, 3, 6	2	2
2	6, 15, 17	3	4
3	15, 22, 33	4	4
4	27, 72, 75	5	6
5	55, 112, 177	6	6

TABLE 6.4 $R = \frac{3}{4}$ OFD CODES

ν	Generators	d_f	Bound
3	2, 13, 14, 15	2	2
4	7, 14, 32, 36	3	3
5	23, 25, 47, 61	4	4
6	45, 106, 127, 172	4	4
7	45, 124, 216, 357	5	6

produced by repeating the two symbols of a rate $\frac{1}{2}$ code, and the corresponding free distance doubles.

Generally, when we speak of an $R = k/n$ code, we presume that k and n are relatively prime, but a code accepting 2 bits per interval and producing 4 coded bits would also have rate $\frac{1}{2}$. Lee [16] has shown that in some cases better free distance is attainable, for a given rate and state complexity, when k and n are not relatively prime. Furthermore, a maximal free distance code is always within the category of encoders having unit memory, that is, one k-tuple of parallel delay elements. An illustration, taken up in the exercises, is that a 16-state encoder with $k = 4$ and $n = 8$ has $d_f = 8$, which is one unit better than the best 16-state code listed in Table 6.2. The comparison of complexities is, however, questionable; the 16-state trellis for the $R = \frac{4}{8}$ code has full connectivity between states, and when computation per bit decoded is analyzed, the standard design is preferred, despite the equal state complexity. It is nonetheless interesting that OFD convolutional codes are remarkably similar to short block codes, having one block of memory.

Perhaps the preeminent case of practical interest has been the $R = \frac{1}{2}$ case. In Table 6.5 we list additional information for codes of memory order $\nu = 2, 4$, and 6, specifically the total *information weight* of all nonzero sequences having code weight d_f, $d_f + 1$, and so on.[7] For example, the $\nu = 6$, $R = \frac{1}{2}$ code has 36 total information 1's on all paths of weight 10 (the free distance), and so on. This information weight spectrum will subsequently become important in performance evaluation, although we will generally not need to directly tabulate the spectrum. Conan [17] provides similar information for $R = \frac{2}{3}$ and $R = \frac{3}{4}$.

Ryan and Wilson [18] have presented OFD q-ary convolutional codes ($q = 4$, 8, and 16) for small memory orders (where maximum likelihood decoding is feasible) and for $R = \frac{1}{2}$ and $R = \frac{1}{3}$. Again, these codes are the result of computer search, which for larger alphabets becomes progressively more time consuming. Table 6.6 shows a listing of the information weight spectra (incorporating free distance information) of the $R = \frac{1}{2}$ and $R = \frac{1}{3}$ codes for 4-ary and 8-ary codes; many encoders are equivalent, and only a representative set of encoder tap connections is presented. In all

TABLE 6.5 INFORMATION WEIGHT DATA FOR $R = \frac{1}{2}$ CODES OF TABLE 6.2

w	$\nu = 2$	$\nu = 4$	$\nu = 6$
5	1	0	0
6	4	0	0
7	12	4	0
8	32	12	0
9	80	20	0
10	192	72	36
11	448	225	0
12	1024	500	211
13	2304	1324	0
14	5120	3680	1,404
15	11264	8967	0
16	24576	22270	11,633

[7] Information extracted from Michelson and Levesque, *Error Control Techniques for Digital Communication*, Wiley-Interscience, New York, 1985.

TABLE 6.6A $R = \frac{1}{2}$ CONVOLUTIONAL CODES OVER GF(4)

$m = 2$		$m = 3$		$m = 4$	
$1\ 1\ 1$		$1\ 1\ 1\ \alpha$		$1\ 1\ 1\ \alpha^2\ \alpha$	
$1\ \alpha\ 1$		$1\ \alpha\ 1\ \alpha^2$		$1\ \alpha\ 1\ \alpha\ \alpha^2$	
w	$C(w)$	w	$C(w)$	w	$C(w)$
6	9	8	39	9	12
7	30	9	42	10	39

TABLE 6.6B $R = \frac{1}{3}$ CONVOLUTIONAL CODES OVER GF(4)

$m = 2$		$m = 3$		$m = 4$	
$1\ 1\ 1$		$1\ 1\ 1\ \alpha$		$1\ 1\ 1\ \alpha\ \alpha$	
$1\ \alpha\ 1$		$1\ \alpha\ 1\ \alpha^2$		$1\ \alpha\ 1\ \alpha\ \alpha^2$	
$1\ \alpha^2\ 1$		$1\ \alpha^2\ 1\ 1$		$1\ \alpha^2\ \alpha\ \alpha\ 1$	
w	$C(w)$	w	$C(w)$	w	$C(w)$
9	3	12	21	14	18
10	18	13	0	15	3

TABLE 6.6C $R = \frac{1}{2}$ CONVOLUTIONAL CODES OVER GF(8)

$m = 2$		$m = 3$		$m = 4$	
$1\ 1\ \alpha^4$		$1\ 1\ \alpha^5\ 1$		$1\ 1\ \alpha^4\ \alpha^3\ \alpha$	
$1\ \alpha\ \alpha^4 1$		$1\ \alpha\ \alpha^5\ \alpha$		$1\ \alpha\ \alpha^4\ \alpha^4\ \alpha^6$	
w	$C(w)$	w	$C(w)$	w	$C(w)$
6	7	8	7	10	21
7	56	9	189	11	392

TABLE 6.6D $R = \frac{1}{3}$ CONVOLUTIONAL CODES OVER GF(8)

$m = 2$		$m = 3$		$m = 4$	
$1\ 1\ \alpha^4$		$1\ 1\ 1\ \alpha^5\ 1$		$1\ 1\ 1\ \alpha^4\ \alpha^3\ \alpha$	
$1\ \alpha\ \alpha^4$		$1\ \alpha\ \alpha^5\ \alpha$		$1\ \alpha\ \alpha^4\ \alpha^4\ \alpha^6$	
$1\ \alpha^5\ \alpha^4$		$1\ \alpha^2\ \alpha^5\ \alpha^4$		$1\ \alpha^6\ \alpha\ \alpha^6\ \alpha^3$	
w	$C(w)$	w	$C(w)$	w	$C(w)$
9	7	12	7	15	7
10	0	13	28	16	70

cases, α is a primitive element in $GF(q)$. These codes extend previously known results for dual-k codes. It is noted that q-ary codes routinely, at least for small memory order attain the upper bound of (6.2.7), which in turn reduces typically to $(m + 1)n$.

Finally, in Table 6.7, we show results on free distance for generalizations of Trumpis codes found in [8, 18]. These codes have one binary input per unit time and n q-ary output symbols.

TABLE 6.7A $R = 1$ BINARY-TO-4-ARY CONVOLUTIONAL CODES

$m = 2$		$m = 4$		$m = 6$	
$1\ \alpha\ 1$		$1\ \alpha\ 1\ 1\ \alpha$		$1\ \alpha\ 1\ 1\ \alpha\ 1\ \alpha^2$	
w	$C(w)$	w	$C(w)$	w	$C(w)$
3	1	5	3	7	7
4	4	6	7	8	39

TABLE 6.7B $R = 1$ BINARY-TO-8-ARY CONVOLUTIONAL CODES

$m = 2$		$m = 4$		$m = 6$	
$1\ \alpha\ \alpha^2$		$1\ \alpha\ \alpha^2\ \alpha^3\ 1$		$1\ \alpha\ \alpha^2\ 1\ \alpha^4\ \alpha^3\ \alpha$	
w	$C(w)$	w	$C(w)$	w	$C(w)$
3	1	5	1	7	1
4	2	6	2	8	2

6.2.4 Punctured Convolutional Codes

High-rate, or low-redundancy, convolutional codes are of interest for bandwidth-constrained applications. For example, we may wish $R = \frac{7}{8}$. To construct such an encoder, we could input 7 bits per unit time to a finite-state machine and produce 8 code bits. The corresponding trellis would have 128 branches departing and entering each state, making the maximum likelihood trellis decoder we will study in the next section more difficult to implement. Furthermore, if a change in rate is desired, we would likely require a fundamentally different decoding structure.

This can be overcome by a process of ***puncturing***, first introduced to the convolutional coding realm by Cain, Clark, and Geist [19] precisely for easing the decoder complexity. The idea is very similar to the concept of puncturing of block codes. We simply delete certain code symbols from a lower-rate code, while keeping the parameter k fixed, to obtain a convolutional code with effectively higher rate. This puncturing is performed in periodic manner, creating a time-varying trellis code. For example, if we adopt an $R = \frac{1}{2}$ code and consider frames of P levels in length, the encoder produces $2P$ output symbols over this same interval. Suppose that we read these into an array of size 2 by P, and delete $D \leq P - 1$ of these symbols from the transmission queue. Then the effective code rate is

$$R' = \frac{P}{2P - D}. \tag{6.2.8}$$

It should be clear then that by appropriate choice of P, the period, and D, the number of deleted symbols, we obtain a convolutional code of any desired rate. Specifically, for a period P punctured code derived from a parent code with rate $R = 1/n$, the achievable rates are

$$R' = \frac{P}{P+l}, \qquad l = 1, 2, \ldots, (n-1)P. \tag{6.2.9}$$

The important observation is that we may utilize a decoder for the parent code, aligned with frame boundaries and operating with P trellis levels per frame, to realize the optimal decoder for the high-rate code. All we need to do is puncture, or erase, certain metric calculations that would ordinarily be performed in the low-rate code, these erased positions corresponding to the deleted transmitted symbols. Otherwise, the metric updating and path storage for the Viterbi algorithm is exactly as for the parent code.

Example 6.10 $R = \frac{3}{4}$ Punctured Convolutional Code Obtained from $R = \frac{1}{2}$ Code

Suppose we adopt the familiar $R = \frac{1}{2}$, $m = 2$ convolutional code of Figure 6.1.1a. Let the puncturing period be $P = 3$. By deleting $D = 2$ code bits from every frame of 6 code bits, as shown in Figure 6.2.6, we obtain an $R = \frac{3}{4}$ code, with trellis as shown. The puncturing pattern is defined by the matrix

$$a = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \tag{6.2.10}$$

where 1 connotes that the corresponding code bit is transmitted, and 0 indicates a deleted position. Thus, every third shift time, both code bits produced by the encoder are utilized, while for the other two intervals, the upper and then lower bits are transmitted.

This code can be viewed on a time scale that runs at the frame rate as a standard $R = \frac{3}{4}$ convolutional code. In this view, every state connects to 8 others. Its memory order, m, and state vector dimension ν are unchanged from that of the parent code.

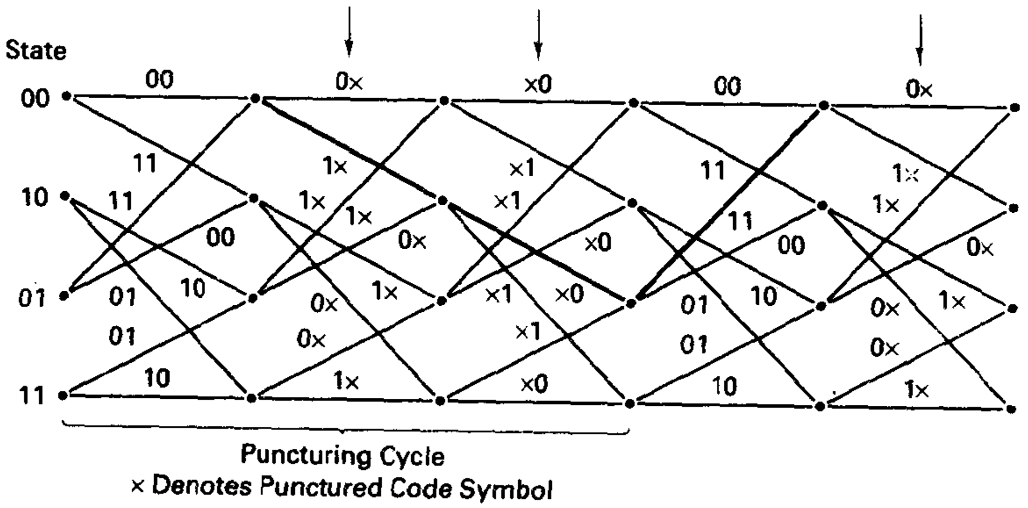


Figure 6.2.6 Trellis for $R = \frac{1}{2}$, $m = 2$ code punctured to $R = \frac{3}{4}$. d_f event highlighted. Puncturing is periodic, deleting positions designated by arrows every three levels.

This example may raise several important questions. First, what puncturing patterns are good? Some are obviously better than others, and some could even produce catastrophic high-rate encoders, even though the parent code is noncatastrophic. Second, is it possible to synthesize good codes using puncturing? More precisely, can a specific optimal $R = \frac{3}{4}$ code, say, be synthesized from a lower-rate encoder? (The answer is yes [20].) More importantly for practice is the observation that a range of code rates can be achieved from a single parent code by merely changing the puncturing map. It turns out that, when we impose this single parent code restriction, slight sacrifice in free distance is sometimes unavoidable; but, generally, the loss is not more than one unit of Hamming distance, and this is somewhat offset by a favorable nearest-neighbor multiplier. Yasuda et al. [21] studied punctured codes obtained from a single parent $R = \frac{1}{2}$ OFD encoder and listed the puncturing patterns shown in Table 6.8 for memory-6 codes (64 states). Other cases are listed in [21] as well.

TABLE 6.8 PUNCTURED CONVOLUTIONAL CODES DERIVED FROM $R = \frac{1}{2}$ CODE WITH $\nu = 6$, $g_0 = 133$, AND $g_1 = 171$

R	Puncturing Table	d_f
$\frac{1}{2}$	1 1	10
$\frac{2}{3}$	11 10	6
$\frac{3}{4}$	110 101	5
$\frac{7}{8}$	1111010 1000101	3
$\frac{13}{14}$	1101000001111 1010111110000	3

Hagenauer [22] has introduced the notion of ***rate-compatible punctured convolutional codes*** (RCPCs) for applications in which either (1) variable error protection is to be assigned to certain bits in a data packet, for example, in digital speech coding by analysis/synthesis methods, or (2) incremental transmission of additional code redundancy in subsequent transmissions is required to enable correct decoding of a packet.

A family of convolutional codes of rates $R = P/(nP - D)$ is constructed from an $R = 1/n$ parent code using a puncturing frame of size $n \times P$ and is a *rate-compatible family* if the deleting map of the low-rate code covers the deleting map of every higher-rate code. In other words, if a 1 appears in a given puncturing table position for a code of rate R, a 1 must appear in that position for all lower-rate codes, and additional transmitted positions can be assigned only where puncturing occurred in the higher-rate codes. If such is the case, we may achieve the effect of a low-rate code incrementally, first transmitting the code symbols of a given high-rate code and incrementally adding, if necessary, only the additional symbols of the lower-rate code, formerly punctured.

Furthermore, the rate-compatible property allows smooth transition from higher to lower rates in the middle of a packet if variable protection is required, since we can operate without reinitializing the encoding register.

We might expect that the rate compatibility constraint imposes additional weaknesses on the codes produced at any specific rate, but this does not seem to be an exorbitant penalty. In Table 6.9 we list information produced by Hagenauer for memory 6 codes, having a parent code that has rate $\frac{1}{3}$. Some flexibility in optimizing the performance profile exists with these codes. For example, the profile is optimal at $R = \frac{1}{2}$ here and falls short of the optimal free distance rate $\frac{1}{3}$ code by one unit. All other free distances are within one unit of (and typically as good as) the best unconstrained code of the same rate and state size and are always as good as the Yasuda codes.

TABLE 6.9
RATE-COMPATIBLE PUNCTURED CONVOLUTIONAL CODES DERIVED FROM $R = \frac{1}{3}$, MEMORY-6 ENCODER, $g_0 = 133$, $g_1 = 171$, AND $g_2 = 621$

R	Puncturing Table	d_f
$\frac{1}{3}$	11111111 11111111 11111111	14
$\frac{1}{2}$	11111111 11111111 00000000	10
$\frac{2}{3}$	11111111 10101010 00000000	6
$\frac{4}{5}$	11111111 10001000 00000000	4
$\frac{8}{9}$	11110111 10001000 00000000	3

Example 6.11 Application of RCPC's to Packet Transmission

Suppose that we have a radio channel packet communication system with either time-varying SNR or in which various terminals exist with different link qualities. At some times we can operate with very low redundancy, or high throughput, while at other moments we need the power of a lower-rate code. Rather than commit to a single compromise code, we could design a RCPC code family with rate $R = \frac{3}{4}$, $\frac{3}{8}$, and $\frac{3}{12}$, used as follows. On the first transmission of a packet, we use the high-rate code and an embedded CRC code at the end of the packet to check for proper decoding. If not, we return a negative acknowledge (NAK) signal, and the transmitter responds in the next available time slot with the incremental redundant bits of the $R = \frac{3}{8}$ code for the same packet. The decoder combines these with the formerly received demodulator outputs to attempt a new decoding. This is tested, and possibly the next lower-rate code is then invoked. This process could continue to arbitrarily low rates in principle, but as a practical matter, it makes sense to abort the decoding after perhaps three trials and request that the cycle start anew.

6.2.5 Optimal Distance Profile Codes

Johannesson and Paaske [23–26] have provided tabulations of ODP convolutional codes of various rates, and $R = \frac{1}{2}$ codes are listed in Table 6.10. Nonsystematic codes are listed, for they achieve larger free distances, although in the ODP sense, systematic codes are just as good. Notice that the memory order of many of these codes is large, rendering maximum likelihood decoding infeasible, but the ODP property lessens the average decoding computation in sequential decoding, for which the computational effort is basically independent of memory order. Johannesson also reports robustly optimal codes, which are simultaneously ODP and OFD.

TABLE 6.10 $R = \frac{1}{2}$ NONSYSTEMATIC OPTIMAL DISTANCE PROFILE CODES

m	g_0	g_1	d_f
2	7	5	5
6	147	135	10
10	3645	2671	14
14	65231	43677	17
18	1352755	1771563	21
22	33455341	24247063	24

6.3 MAXIMUM LIKELIHOOD DECODING OF CONVOLUTIONAL CODES

As was the case with block codes, several decoding approaches are possible with trellis codes. On the trivial side, we may extract the information from any coded sequence produced by a noncatastrophic encoder with a simple algebraic inverse circuit having short delay. Such a decoder does not exploit any performance gain available through coding and, in fact, outputs more errors than were initially present. Of greater interest are procedures that search the trellis for a good (if not best) candidate code sequence in the sense of highest total path likelihood. There are several possible approaches in this class. The earliest procedures proposed are suboptimal for two reasons: they perform sparse search of the trellis paths, for reasons of complexity, and generally do not employ the maximum likelihood path metric. Foremost among these are ***threshold decoding***, which is not a search algorithm per se, but essentially finite-delay syndrome decoding as in

Chapter 5, and ***sequential decoding***, a family of probabilistic search algorithms. Neither class of decoders invokes the later understanding of a trellis structure. We will examine these procedures briefly later in the chapter. The algorithm that selects the maximum likelihood sequence, due to Viterbi, and usually referred to as the ***Viterbi algorithm***, or VA, has emerged as a powerful and practical decoding method for a variety of trellis coded applications, and this section is devoted to its description and implementation details.

6.3.1 Maximum Likelihood Sequence Decoding (Viterbi Algorithm)

This procedure was originally proposed by Viterbi in 1967 [3] as an "asymptotically optimal" decoding algorithm for convolutional codes and was later shown by Omura [27] to correspond to the dynamic programming solution to the problem of finding the maximum likelihood solution. Forney [28] showed the algorithm to be optimal in the sense of choosing the *most probable* (or *minimum probability of error*) *sequence*, provided the proper metric is used in scoring path contenders, and he coined the term *trellis* to conveniently describe the algorithm. The algorithm has been widely applied to various decision problems that involve noisy observation of finite-state Markov systems. These applications include automatic machine recognition of speech and decoding intersymbol interference, presented in Chapter 7. It is worth emphasizing that the algorithm does not rely on a linear system model, but only on a finite-state description, and thus the decoding algorithm extends directly to decoding of more general trellis codes, as we shall see later in the chapter.

We seek the maximum likelihood path of length $L + m$[8] through the trellis, based on receipt of the observation sequence $\tilde{\mathbf{r}} = (\mathbf{r}_0, \mathbf{r}_1, \ldots)$. All trellis paths emanate from the agreed on initial state (typically $\sigma_j = S_0$) at time $j = 0$ to the same final state at stage $j = L + m$. For a memoryless channel, the total path likelihood for any trellis route i, corresponding to a code sequence $\tilde{\mathbf{x}}^{(i)} = (\mathbf{x}_0{}^{(i)}, \mathbf{x}_1{}^{(i)}, \ldots)$, will be a product of likelihoods obtained for the various symbols on the given path. Equivalently, by taking the log likelihood as our objective function for maximization, we find that the total log likelihood is the sum of log likelihoods:

$$\Lambda_i(\tilde{\mathbf{r}}, \tilde{\mathbf{x}}^{(i)}) = \sum_{j=0}^{L+m-1} \lambda(\mathbf{r}_j, \mathbf{x}_j{}^{(i)}), \tag{6.3.1}$$

where j is a stage index, $\mathbf{r}_j$ is the vector of observations made at stage j, and $\mathbf{x}_j{}^{(i)}$ is the hypothesized code vector for the ith codeword (path) at stage j. This log-likelihood metric for each stage is in turn a sum of *symbol* metrics if, as usual, multiple symbols are associated with a given trellis stage.

The optimality of Viterbi's algorithm derives from Bellman's ***principle of optimality*** [29], pertaining to a sequential decision problem whose global objective function is an additive function of costs of transitions between intermediate states. The basic idea is that the globally optimal policy (trellis path) must be an extension of an optimal path to

[8] Again we envision an L-stage message sequence with a suffix of m zero-vector inputs.

some state at an intermediate time j, and this must hold for all time indexes j. The proof is by contradiction: suppose that the globally optimal path passes through state $\sigma_j = m$ at time j, as shown in Figure 6.3.1. If the initial part of the path from starting state to this intermediate state is not an optimal path, we could replace this segment by the dashed segment to this state and thereby improve on our "globally best" choice, a contradiction. (Similarly, paths from any intermediate state to a terminal state must be optimal.) It is crucial to this argument that the objective function increments be dependent only on the specific state transition at a given time and not on future or past state trajectories.

The concept is readily conveyed by considering a hypothetical round-the-world trip, beginning in Zurich, as illustrated in Figure 6.3.2. We agree to travel east, allowing ourselves to visit Moscow or Cairo at the end of the first stage. The next stop

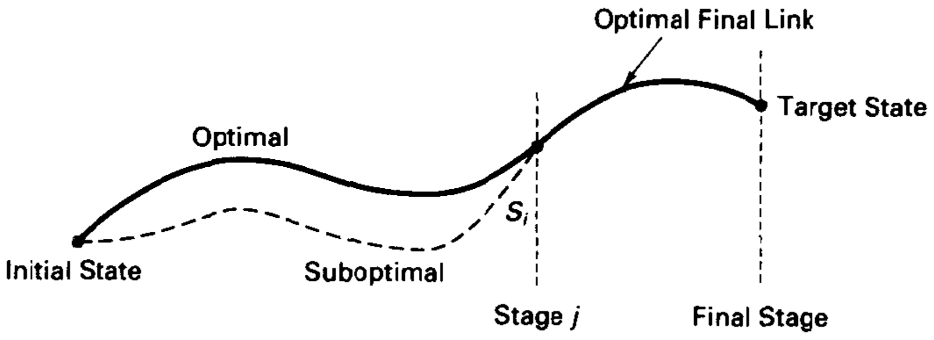


Figure 6.3.1 Principle of optimality: globally-best route is extension of optimal route to intermediate nodes.

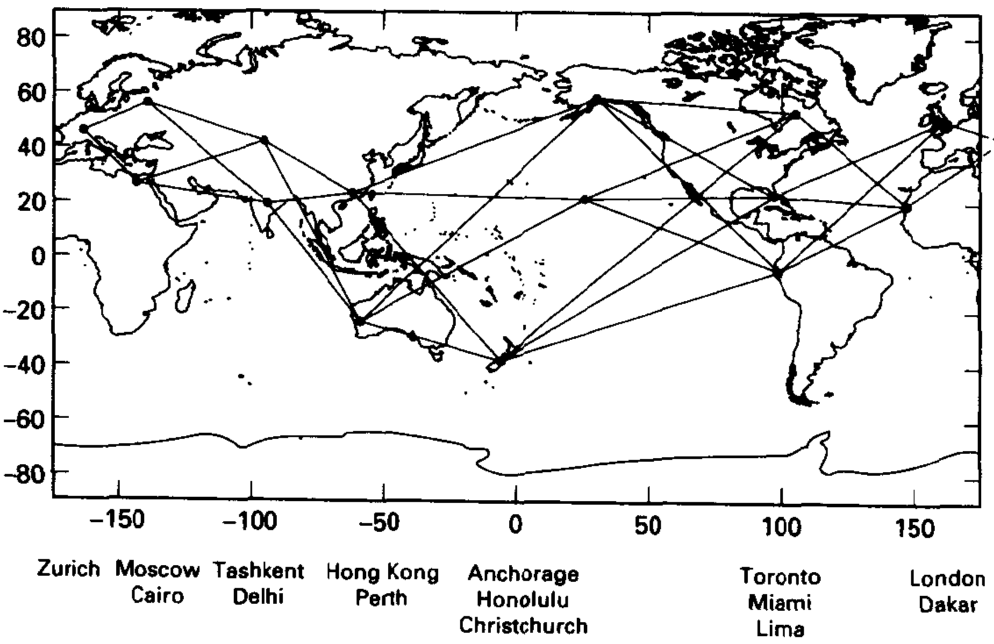


Figure 6.3.2 Trip planning based on principle of optimality.

on the itinerary can be Tashkent or Delhi, and so on as shown. To finish in Zurich, we must pass through either London or Dakar. To each route segment, we assign a metric that depends only on the two cities involved, such as distance in kilometers or purchasing power in a given destination. We assume the routes shown are the only possible routes and that backtracking is not allowed. [This graph is less regular than code trellises, in that the trellises we encounter will have a fixed number of states (cities) at every stage.]

To travel the complete circuit with minimal distance, say, we must first be optimal in reaching Moscow or Cairo. This is not difficult—take the only options available. More profound, though, is that, although we do not know the eventual route as yet, we had better determine the optimal route to each of Tashkent and Delhi. These are extensions of optimal routes to the previous day's cities. Similar facts must be true of routes to Hong Kong and Perth, and so on. What is also crucial to recognize is that *only the best* among the candidates to each city needs to be saved, and we will never risk missing the globally optimum path if we dispense with inferior intermediate paths.

In terms relevant to the decoding of trellis codes, we assert that if the ultimately selected (highest likelihood) trellis route visits some state $\sigma_j = S_i$ at stage j it must have been reached by an optimal route from the beginning to that intermediate node. This must hold true for all states at time j and for all time indexes and constitutes the heart of *dynamic programming* in optimal control theory [29].

To formally describe the algorithm, we first establish some notation. Let $\Sigma = \{S_i, i = 0, 1, \dots, S-1\}$ denote the state space for the encoder, with $S = q^\nu$ denoting the number of states. Let $T = q^k$ denote the number of transitions from any state to next states. For any state S_i, we let $B_i = \{B_{i,p}, p = 1, 2, \dots, T\}$ designate the set of previous states that can transition in one step to state S_i. There are T such previous states for each S_i, and these will be distinct antecedent states unless parallel state transitions exist in the trellis. Finally, attached to any transition of the form $B_{i,p} \to S_i$ are information vectors $\mathbf{u}_{p,i}$ and code symbol vectors $\mathbf{x}_{p,i}$. The latter are, respectively, k-tuples and n-tuples from GF(q). We designate by $\Lambda_j(S_i)$ the ***cumulative metric*** for the optimal path to state S_i at time j. The ***survivor path histories*** are strings denoted by $\{\mathbf{P}_j(S_i)\}$ and are comprised either by sequences of states followed or, more commonly, sequences of input vectors corresponding to the adopted routing. In these terms, the principle of optimality holds that for each state S_i at time j

$$\Lambda_j(S_i) = \max_{B_{i,p}} \left[\Lambda_{j-1}(B_{i,p}) + \lambda(\mathbf{r}_j, \mathbf{x}_j(B_{i,p} \to S_i))\right]$$

and (6.3.2)

$$\{\mathbf{P}_j(S_i)\} = \{\mathbf{P}_{j-1}(B_{i,p}), \mathbf{u}_{p,i}\}.$$

That is, for each state we find the best-metric entering route, called the *survivor*, that is an extension of a survivor to one of the communicating previous states and store only the ***cumulative metric*** of the survivor and the ***survivor path*** by appending the appropriate path suffix. The principal of optimality guarantees that these are sufficient data for storage as the algorithm proceeds through the trellis, and all but one of the q paths entering each state can be pruned away in each iteration.

Variables/Storage

S	Number of states
T	Number of transitions to each state
j	Time index
$\hat{\mathbf{x}}_j(\sigma_i),\ 0 \le \sigma_i \le S-1$	Survivor sequence terminating at state σ_i at time j
$\Lambda_j(\sigma_i),\ 0 \le \sigma_i \le S-1$	Cumulative metric of survivor to σ_i at time j

Initialization

$j = 0$

$\hat{\mathbf{x}}_0(i) = \text{null, all } i$

$\Lambda_0(0) = 0,\ \Lambda_0(\sigma_i) = \infty,\ i \ne 0$

Recursion

```
for $j = 1$ to $m + L$
    for $i = 0$ to $S-1$
        find max $\Lambda_n(\sigma_i) = \Lambda_{n-1}(\sigma_p) + \lambda[r_n, x(\sigma_p \to \sigma_i)]$
            for $p = 1, \ldots, T$
        Store maximum metric in $\Lambda_n(\sigma_i)$
        Update survivor: $\hat{\mathbf{x}}_n(\sigma_i) = (\hat{\mathbf{x}}_{n-1}(\sigma_p), u[\sigma_p = \sigma_i]$
    end $\textcircled{B}$
end
```

Output

Release survivor to σ_0 at time $m + L$ as ML sequence estimate

Figure 6.3.3 Pseudocode for Viterbi algorithm.

Figure 6.3.3 provides a pseudocode for the decoding algorithm. An initialization stage sets all cumulative metrics to a poor value, except that the initial encoder state, typically taken as S_0, is assigned initial metric $\Lambda_0(S_0) = 0$. (This forces eventual selection of a path from the known starting state.) All survivor path histories are set to null sequences. At each increment of time j, the decoder accepts a new input from the demodulator, $\mathbf{r}_j$. The nature and size of this vector depend on the code rate and the modulation/demodulation strategy. In particular, the input data may be real or finite alphabet and may be a scalar or a vector.

The kernel of the algorithm shown takes a "look back" viewpoint. For each state $\sigma_j = S_i$ at time j, we look back to the previous or antecedent states, $B_{i,p}$, $p = 1, 2, \ldots, T$. To the cumulative metrics $\Lambda_{j-1}(B_{i,p})$ of each previous state, we add the branch metric $\lambda(\mathbf{r}_j, \mathbf{x}_{i,p})$, dependent on the new input and the trellis branch being scored. For each such transition to state S_i, we determine if the new metric sum exceeds the current best result for state S_i. If so, we store the new metric sum in $\Lambda_j(S_i)$. If, on the other hand, the new test path has poorer total metric than the current best to the given state, we simply proceed to evaluate the next transition, if any remain. If $B_{i,m}$ is the winning parent node for state S_i, we also update the survivor path histories according to

$\{\mathbf{P}_j(S_i)\} = \{\mathbf{P}_{j-1}(B_{i,m}), \mathbf{u}_{i,m})\}$; that is, we append the information sequence of the newly found best survivor as a suffix to the former survivor path.[9] At point A in Figure 6.3.3, we have completed the assessment for state S_i. The inner loop shown is executed for S states.

At point B in Figure 6.3.3, we have completed a trellis stage and extended the survivor path histories one level deeper, as well as computed new cumulative metrics for these survivors. The complete sequence of operations repeats at level $j+1$ and continues until the end of the message is reached, at which time the survivor sequence to state S_0 is declared the maximum likelihood sequence, assuming that the encoder is brought to this end state. As described thus far, no path decisions are made until the end of the message sequence.

Example 6.12 Decoding Illustration for $R = \frac{1}{2}$, $\nu = 2$ Code

To illustrate an actual decoding process, assume the encoder is that of Figure 6.1.1a. Suppose that the encoder is initialized in the state 00, and the 4-bit all-zeros message (0000) is transmitted, followed by a terminating string of (00). The code sequence is thus the all-zeros sequence, and the modulator produces the signal $s_0(t)$ 12 times, twice per information bit. Suppose that we use antipodal modulation, but the SNR is a rather low $E_s/N_0 = -3$ dB, and that the received demodulator output is

$$\tilde{\mathbf{r}} = (-1.1, 0.3|0.1, -0.9| -0.5, -1.3| -0.5, -0.6|0.2, 0.5| -1.2, -0.9). \tag{6.3.3}$$

When binary (hard) decisions are made on each symbol, four errors exist, somewhat more than we would expect in 12 transmissions, even at the poor SNR assumed. In Figure 6.3.4a, we show the evolution of decoding with hard-decision quantization, with the Hamming metric employed as branch metric. (To maximize path likelihood, we wish to *minimize* Hamming distance.) By assigning initial metrics as described previously, only descendants from the zero state at stage $j = 0$ can survive, and no special algorithm is needed to handle the initial fanout of the trellis. A similar remark applies to the termination stage: we simply decode as usual, but at stage $j = L + m = 6$ simply pick the survivor at state S_0. At the end of the cycle, the information sequence $\tilde{\mathbf{u}} = (101000)$ is released by the decoder, producing two information bit errors.

As an alternative means of decoding, Figure 6.3.4b shows the decoder progression when the unquantized observations are used by the decoder, and the branch metric is

$$\lambda(\mathbf{r}_j, \mathbf{x}_{i,p}) = x_{i,p}^{(0)} r_j^{(0)} + x_{i,p}^{(1)} r_j^{(1)}, \tag{6.3.4}$$

that is, the sum of correlations for the two code symbols at each level. (In performing the correlation we map code symbols 0 and 1 to -1 and 1, respectively, to obtain normalized signal-space coordinates.)

Observe that the Viterbi algorithm chooses different paths in the two cases, and in fact the correct path is chosen in the second case, although the decoder has no way of knowing this. This is no contradiction of optimality, but reflects the fact that the two decoders operate with different input observations and with different path metrics. This example was in fact intentionally constructed to show the ability of soft-decision decoding to outperform hard-decision decoding, but it should not be concluded that soft-decision decoding always

[9]We must be careful to avoid overwriting memory locations corresponding to previous metrics and path histories until all states needing access to this information at the current level have been processed; typically, this can be implemented by double memory buffers and/or a system of temporary pointers.