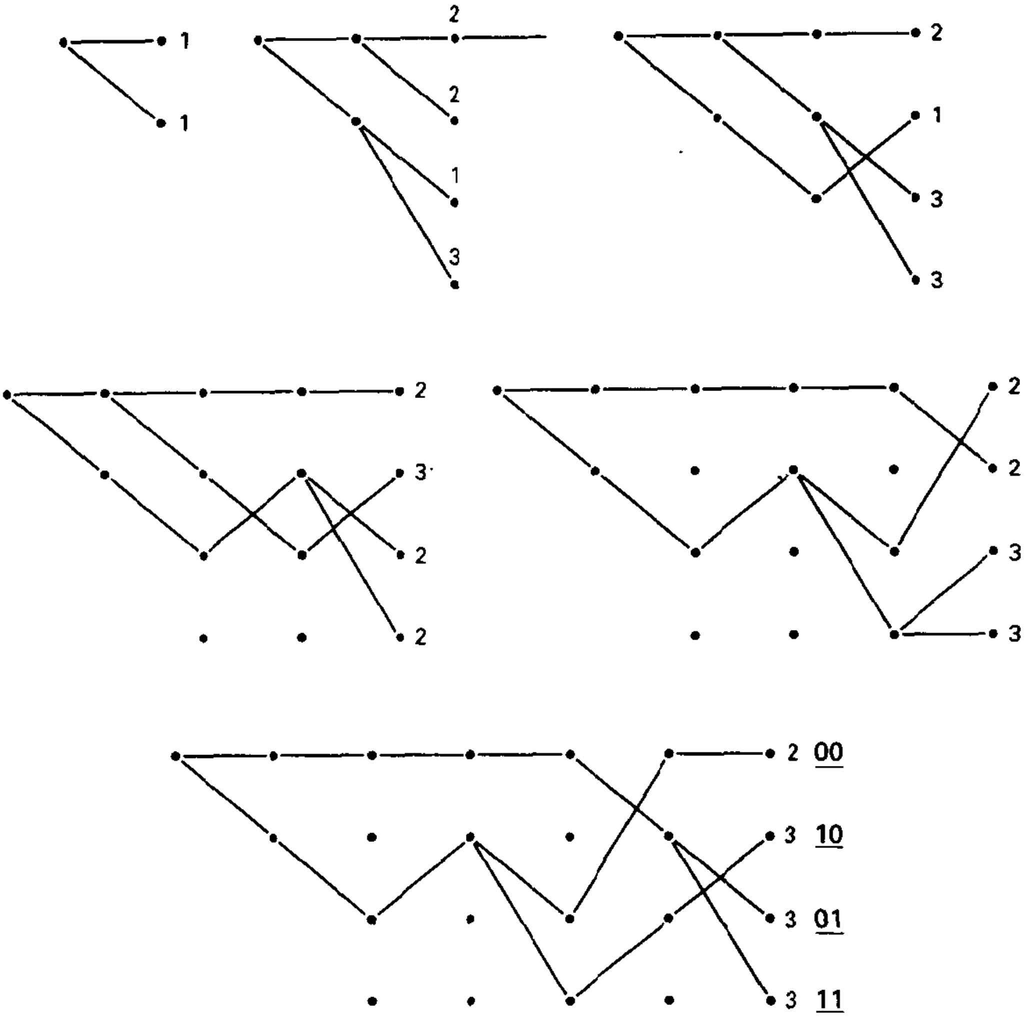


Figure 6.3.4a Evolution of survivors, hard-decison decoding.

succeeds when hard-decision decoding errs. As discussed in our earlier study of decision theory, it is possible to construct noise sequences for which binary decoding produces the correct sequence while unquantized decoding produces an error, but on balance the second option is superior, as we shall see.

Before moving to a discussion of implementation details for the VA, it is interesting to note that the algorithm does not directly minimize the probability of symbol error at any given position of the message, but instead chooses the entire message that is most likely. To actually minimize symbol error probability, extra trellis computation must be done, including a forward/backward recursion, for truly minimizing error probability [30, 31] and it is simply not worth the extra effort. The same actually pertains in block coding: an ML decoder chooses the codeword with highest likelihood, not necessarily the policy that would minimize error probability in various positions of the decoded message.

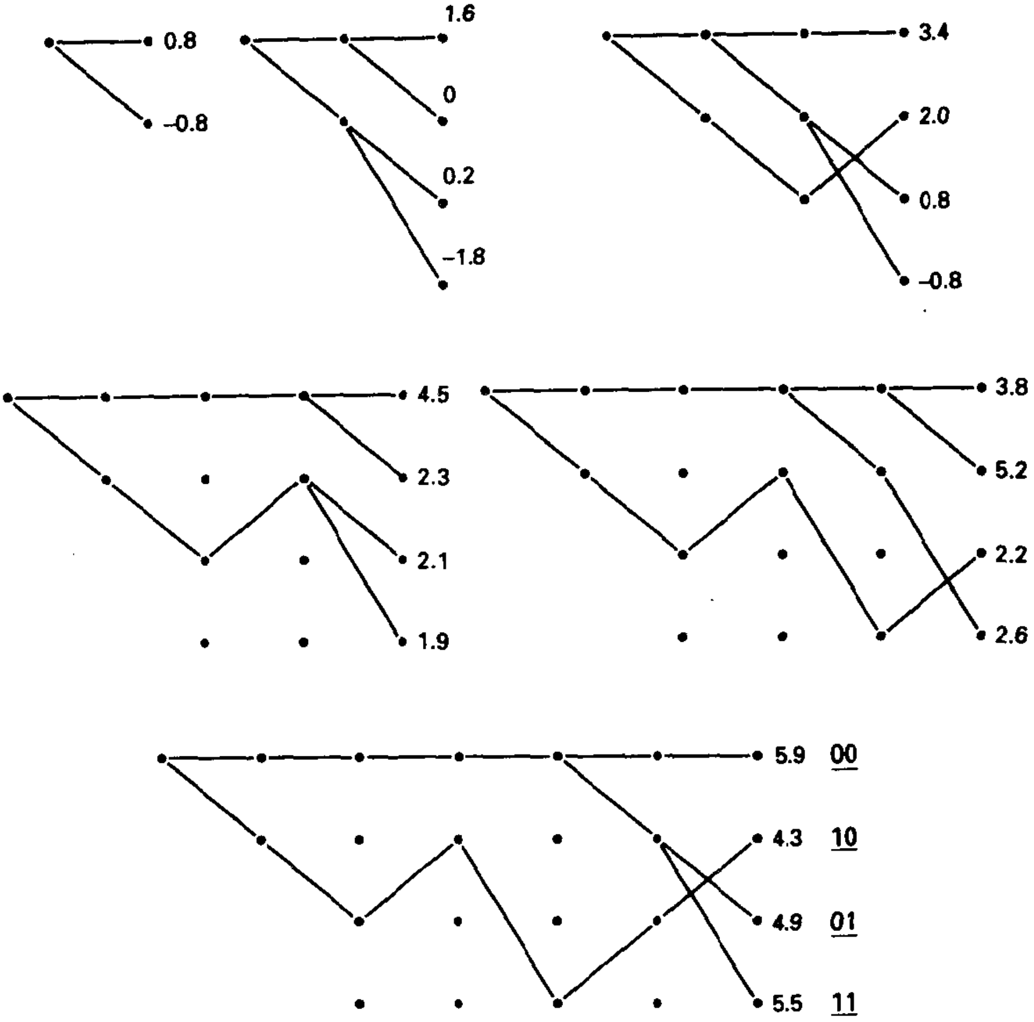


Figure 6.3.4b Evolution of survivors, soft-decison decoding.

6.3.2 Implementation Issues[10]

The fundamental sequence of operations is identical at each state S_i and is known as ***add–compare–select** (ACS)*. This suggests a building-block approach for decoder implementation, since each state is updated in similar fashion. A software implementation of the decoder benefits from the highly repetitive nature of the basic operations. Furthermore, the decoder algorithm is a perfect candidate for parallel implementation in VLSI architecture due to decoupling of the computation. Specifically, we can always identify sets of originating states and sets of next states that have complete intraconnectivity, but no connectivity with other state sets, allowing parallelism in computation. In the case of $R = \frac{1}{2}$ codes, we can always identify *pairs* of originating states that communicate with pairs of next states, as shown in Figure 6.3.5; such structure is reminiscent of the

[10] Clark and Cain [50] includes a fine discussion of practical issues surrounding the VA.

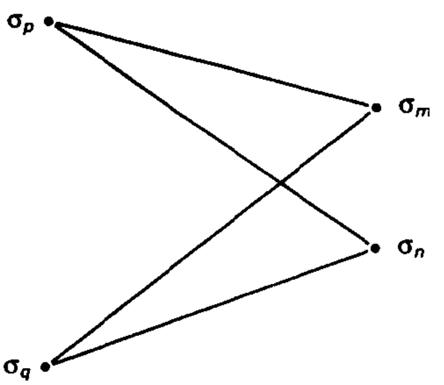


Figure 6.3.5 Butterfly section of a trellis. A set of states has the same set of antecedent states.

"butterfly" graph of the fast Fourier transform algorithm. More generally, we will encounter butterfly subgraphs in the trellis with q^k previous states connecting to a set of q^k next-states. For the trellis of Example 6.6, we have a single four-input, four-output butterfly. Such trellises with a large multiplicity of branches per state, for example, corresponding to an $R = \frac{7}{8}$ code, are relatively awkward to decode at high speeds. This has led to the concept of punctured convolutional codes [19], discussed in the previous section, which allow an $R = \frac{1}{2}$ encoder/decoder to be invoked to implement a high-rate convolutional code. The decoder merely inserts a neutral metric, or skips the metric calculation, when a punctured symbol position is encountered.

Measured in either execution time per trellis stage for a sequential processor or chip size in a parallel VLSI implementation, the decoder complexity is proportional to q^{ν}. In particular, the number of ACS operations, as well as the number of survivor paths, is q^{ν}, and the number of cumulative metric calculations or branches evaluated is $q^{\nu+1}$. Thus, the algorithm is limited in practice to reasonably small values of ν, although feasibility depends on the technology of implementation and the required speed of transmission. A widely utilized $R = \frac{1}{2}$ code has $\nu = 6$, with $g_0 = 133_8, g_1 = 171_8$, implying a 64-state decoder with 32 two-point butterflies in the trellis. This code has become something of a de facto standard in the telecommunications world, primarily because it provides attractive coding gain for still manageable complexity. VLSI implementations of the decoder that operate at bit rates in excess of 20 Mbps have been developed for the commercial market [32, 33].

Several issues are involved in actual implementation of the algorithm. Foremost is the path memory management. For long messages, say with 100,000 bits or even unterminated communication, we would hope to avoid maintenance of survivor paths of that length, especially since only the recent past information symbols are statistically linked by the encoder memory to the current received data (recall the banded structure of the generator matrix for convolutional codes). In fact, a finite-memory (or fixed-lag) decoder with path memory equal to the decision depth, N_D, is essentially maximum likelihood [34]. This amount of memory ensures that all unmerged paths branching from the correct path at the symbol release time have Hamming distance greater than the free distance, and therefore at high signal-to-noise ratio, the free distance event(s) will dominate the error probability. We can no longer guarantee selection of the global ML path when the decoder truncates its memory, but we can make premature truncation of an ultimately maximum likelihood path have negligibly small probability, relative to the probability of error for an infinite memory decoder. Typically, the decoder delay N_d is chosen larger than the decision depth, N_D, defined in Section 6.2, for margin in this regard, and a typical rule of thumb is to use a decoder delay of four to five times the memory order m of the code, although this rule seems to have evolved from studies of binary rate $\frac{1}{2}$ codes. High-rate punctured codes need even longer decoder memory.

The manner in which survivor path histories are actually stored is usually the following, described for the case when $k = 1$. For each state, we store a 1-bit pointer signifying whether the upper or lower of two previous states produced the survivor. After penetration into the trellis for N_d levels, we "traceback" these binary strings, using these strings to follow the state sequence backward in time, ultimately releasing a single bit. Actually, because this traceback requires fetching survivors, extracting a pointer bit, fetching a new survivor, and so on, it is advantageous to traceback every 8 levels, say, releasing the oldest 8 bits of the best survivor. This amortizes traceback effort over 8 decoded bits. In a software implementation, registers can store the path survivors, and these can be updated by shifting, appending appropriate winning extensions, and copying into the desired location in memory.

Finite memory decoding presents another issue related to symbol release. With high probability, given adequate path decision delay N_d, the path survivors at stage j will have a common ancestry (and path history) at stage $j-N_d$, and, if so (as in Figure 6.3.6a), there is no ambiguity as to which symbol should be released. Occasionally, these histories will not agree, however, as illustrated in Figure 6.3.6b. We might then adopt the rule of releasing the oldest symbol of the path with greatest metric at level j; however, this requires

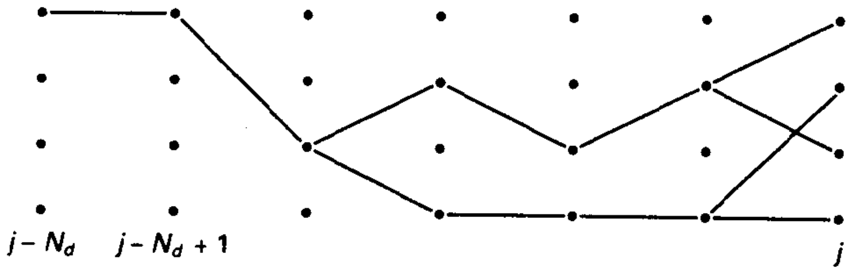


Figure 6.3.6a Survivor path histories that agree at release depth $j - N_d$.

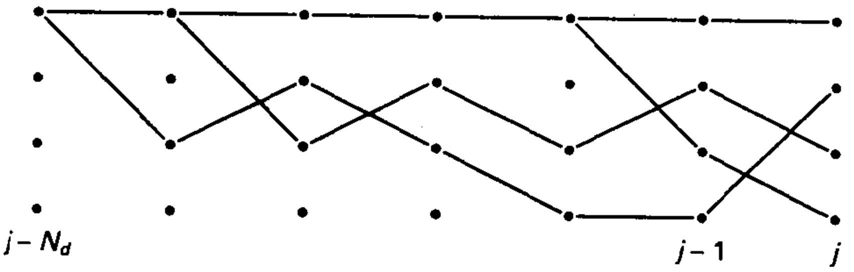


Figure 6.3.6b Survivor path histories that conflict at release depth.

additional metric sorting. Another reasonable rule would be to perform majority voting on the symbol to be released. Typically, little performance is lost if the decoder arbitrarily releases the oldest symbol on the surviving path to an arbitrary state, again presuming that N_d is adequately large. In any case, decision errors due to truncation do not propagate. Further memory management issues in trellis decoding are discussed by Rader in [35].

Another implementation issue involves metric calculations. Instead of computing log-likelihood branch metrics as prescribed in (6.3.1), for example, we would prefer a fast table lookup of the branch metric. This can be accomplished if the decoder input alphabet is discrete, that is, quantized to some degree. Of course, if binary transmission is employed, binary decisions could be made by the demodulator, in which case the metric computation is a trivial exclusive-OR operation of the received symbol with the hypothesized code symbol. We have seen already for block codes that hard-decision demodulation will cost about 2 dB in performance on the antipodal, AWGN channel, vis-a-vis *unquantized demodulation and decoding*. The practical question is, "What level of quantization is acceptable?" Simulation results for binary antipodal modulation and rate $\frac{1}{2}$ codes show that properly scaled 3-bit (8-level) quantization performs within about 0.25 dB of the unquantized decoder, while 2-bit (4-level) quantization costs about 1 dB. These numbers also emerge from a study of R_0 for quantized channels, as discussed in Chapter 4.

If the receiver output data are quantized to Q levels and the number of possible signals whose metric must be found is q, then metrics may be precomputed and stored in a table of size q by Q. The entries in this table should be the log likelihood for the pair; that is, table entry (i, j) is the logarithm of the probability of receiving quantizer level j, given that signal i is sent. These will be noninteger valued, but they may be scaled and rounded to obtain integer values if desired.

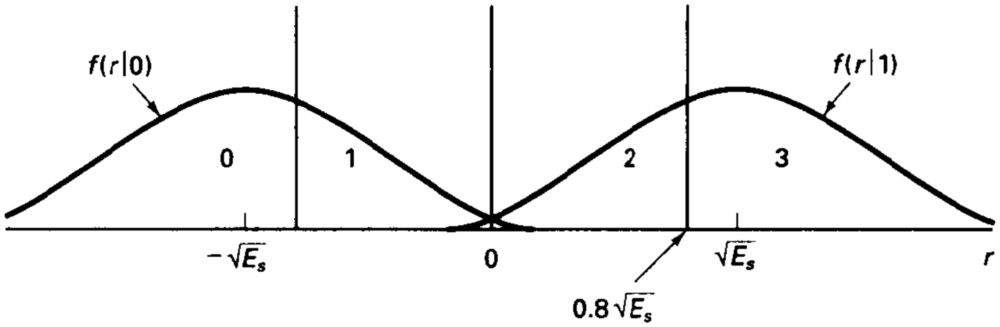
Example 6.13 Scaling of Metrics for 4-Level Quantization

Suppose we employ 4-level quantization for an $R = \frac{1}{2}$ binary convolutional code, with $E_s/N_0 = 2$ dB on a Gaussian channel. (The equivalent E_b/N_0 figure is 5 dB.) In Figure 6.3.7, we illustrate the demodulator output p.d.f. for a single received code symbol and place quantizer thresholds at 0 and $\pm 0.8E_s^{1/2}$. (This is essentially optimum for a four-level quantizer.) In so doing, we induce a DMC with the symmetric transition probability diagram shown.

By taking the natural logarithm of these numbers and then adding 0.2 and rounding to the nearest integer, we obtain the integer-valued metric table shown in Figure 6.3.7. Rounding to integers is not necessary, but merely serves to illustrate the kinds of liberties we may take with metric precision.

A final practical issue is that of metric accumulation and potential overflow. One way to handle the metric growth is to check whether the cumulative metric to state $\sigma_j = S_0$ at time j is nearing a critical value and then subtract a constant value from *all* cumulative metrics to place the metrics back in the range of safety. Survivor metrics tend to stay reasonably close in value; otherwise, they would not survive. So checking any single metric for overflow is suitable. A simple procedure recently described by Hekstra in [36] is to utilize two's-complement arithmetic for metric addition, in which case the overflow problem is handled naturally.

Numerous investigations of VLSI architecture issues related to the VA have been made, and the references [37–39] can be consulted for readers interested in this aspect.



Input \ Output	0	1	2	3
0	0.64	0.32	0.038	0.0007
1	0.0007	0.038	0.32	0.64

Transition Probabilities

	0	1	2	3
0	–0.45	–1.14	–3.27	–7.26
1	–7.26	–3.27	–1.14	–0.45

Log Likelihoods

0	–1	–3	–7
–7	–3	–1	0

Translated, Rounded Metrics

Add 0.2 and Round

Figure 6.3.7 Four-level quantization for antipodal transmission, $Q = 4$, $E_s/N_0 = 2$ dB.

6.4 ERROR PROBABILITY WITH ML DECODING OF CONVOLUTIONAL CODES

We are now ready to evaluate the error probability for the ML decoder, given a particular convolutional encoder, a modulation/demodulation scheme, and a channel model. We must first define the decoding error event appropriately, however, for we should recognize that, on any nontrivial channel, as the message length increases, the probability of *at least one* decoder detour from the transmitted route approaches 1. (In more casual terms, an error-producing sequence of trials will eventually happen, if it is possible to happen, in repeated trials of a probabilistic experiment!) Thus, in distinction with block codes, it is normally not the message error probability that is evaluated for convolutional codes, but instead a measure of the *frequency* of output information errors. Even though the message error probability may approach 1 with increasing message length, the decoded symbol error probability can remain quite low. (This issue also pertains to block coding: if a message is transmitted as a sequence of N codewords, the probability of *message* error tends to 1 as N increases for fixed codeword length n. However, the probability that any specific message symbol is in error may still be small.)

Convolutional codes are linear codes, making the distance structure invariant to choice of reference sequence, and if the channel is uniform from the input, the all-zeros information sequence can be adopted as the transmitted sequence for error analysis purposes. We desire the decoder to select this same trajectory in the trellis after a small delay associated with the path comparison process.

Figure 6.4.1 illustrates a typical decoding produced by a trellis decoder, showing that two detours, or departures, from the all-zeros path were ultimately produced. We say ***node errors*** occur, or ***decoding error events*** commence, at trellis levels j_1 and j_2, although the decoder does not actually select these paths until some later time. (Some define the time of a node error as the time of remerging—the difference is immaterial in the end.) These decoder detours occurred precisely because the metric increment of the incorrect path was greater than that of the correct path *over the unmerged segments*. It may be that two other paths shown in dashed lines in Figure 6.4.1 also have greater metric increment over their unmerged spans than the all-zeros path; however, these were not selected (and did not induce a node error at their starting point) because the path shown as the selected path had still better metric as measured by the Viterbi algorithm.

We are ultimately interested in the probability of two events. The first is the event that at some time, say j, the ultimately selected path will diverge from the all-zeros path. We say this constitutes a ***node error at stage j*** and denote the probability of this event by $P_e(j)$. As might be expected by now, we will settle for tight upper and lower bounds to this probability. Of greater interest is the postdecoding probability of symbol error, denoted P_s. We will get to this by first bounding the probability of node error.

To evaluate $P_e(j)$, let's define the set of all error events diverging from the all-zeros path at time j as I, for "incorrect." Given the preceding discussion, we may bound the probability of this event by

$$P_e(j) \leq P(\text{some } \tilde{\mathbf{x}}^{(i)} \in I \text{ has higher likelihood than the all-zeros path}) \qquad (6.4.1)$$

The inequality follows because it is necessary, but not sufficient, that a path in I have higher likelihood for it to induce a node error at time j, as discussed with reference to Figure 6.4.1. In general, $P_e(j)$ depends on j for a finite-length trellis, since the size of an incorrect subset depends on j, but by assuming a long message so that end effects may be neglected, we may pick any stage, say $j = 0$, for the calculation, and we will call this simply P_e. (The node error probability is, in fact, smaller near the termination of decoding due to extra side information held by the decoder, and our assumption retains a valid upper bound.)

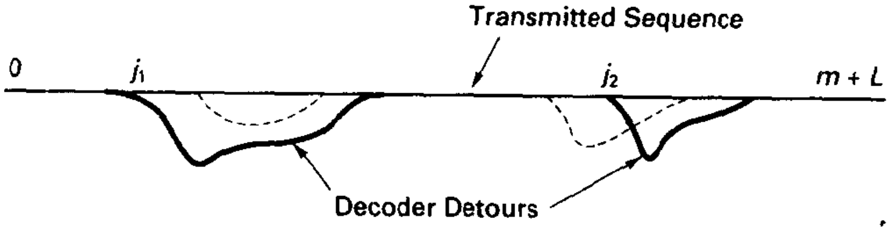


Figure 6.4.1 Error events beginning at j_1 and j_2.

The node error event is a union of error events, each involving the choice of a specific incorrect path instead of the all-zeros path. These constituent error events are all defined in terms of common random variables, however, and exact calculation for P_e is formidable. Thus, we resort to a standard union bound:

$$\begin{aligned} P_e &\le \sum_{\tilde{\mathbf{x}}^{(i)} \in I} P(\tilde{\mathbf{x}}^{(i)} \text{ has higher likelihood than the all-zeros path}) \\ &= \sum_{\tilde{\mathbf{x}}^{(i)} \in I} P[\Lambda(\tilde{\mathbf{x}}^{(i)}, \tilde{\mathbf{r}}) \ge \Lambda(\tilde{\mathbf{x}}^{(0)}, \tilde{\mathbf{r}})]. \end{aligned} \tag{6.4.2a}$$

A simple lower bound is

$$P_e \ge P[\Lambda(\tilde{\mathbf{x}}^{(fd)}, \tilde{\mathbf{r}}) \ge \Lambda(\tilde{\mathbf{x}}^{(0)}, \tilde{\mathbf{r}})], \tag{6.4.2b}$$

where $\tilde{\mathbf{x}}^{(fd)}$ denotes any single error event with distance d_f relative to the all-zeros path.

Each of the probabilities in (6.4.2) is a two-codeword probability of error, which depends on the modulator/channel/demodulator configuration, but in typical cases ultimately on the Hamming distance, d, between the all-zeros sequence and the specific incorrect sequence. (Recall, for example, that the two-codeword probability of error for antipodal signaling on a Gaussian channel has a Q-function dependence on the Hamming distance.) More generally, for memoryless channels, we can invoke the Bhattacharyya bound discussed in Section 4.3:

$$P_2(w) \le B^w, \tag{6.4.3}$$

where B is the Bhattacharyya parameter of the channel.

Because our reference path is the all-zeros path, the Hamming distance to another path is just the Hamming weight of the latter. (This is really the only reason for assuming the all-zeros path.) The node error probability can therefore be bounded as

$$P_e \le \sum_{w=d_f}^{\infty} A(w) P_2(w), \tag{6.4.4}$$

where $A(w)$ denotes the number of error events with Hamming weight w in the incorrect subset, analogous to the weight spectrum of a block code, and d_f is the free distance of the convolutional code defined in the previous section. Evaluation of the upper bound thus requires enumeration of the weight spectrum for the incorrect subset. Evidently, we must consider error sequences with arbitrarily large length and weight, since the incorrect subset is defined for an arbitrarily long trellis. We should anticipate, however, that only the small weight error events are practically significant, and if the encoder is well conceived (noncatastrophic), the weight of progressively longer error events in I keeps growing, and the sum in (6.4.4) will converge quickly.

The required weight enumeration is provided by a clever graph-theoretic approach, due to Viterbi [40], which views the state transition diagram as a signal flow graph. Realizing that our goal is describing sequences that begin in the state $\sigma_j = S_0$, depart from it, and later return, we begin by splitting the all-zero state into an originating state and a terminating state. Any error event, or decoder detour, corresponds to a route from input to output states in this ***split-state transition diagram***. Figure 6.4.2 provides this split-state transition diagram for the encoder of Example 6.1. In general, such diagrams have $S + 1$ nodes or pseudostates. The diagram may also have bypass routes from the source state to the sink state, when some information symbols are not part of the state vector, as in the encoder of Figure 6.1.1e.

Next, let us label each arc of the split-state transition graph with a *path gain* of the form $W^x I^y$, where W and I are dummy indeterminates, signifying "weight" and "information." The integer-valued exponent x denotes the Hamming weight associated with a given state transition, and y is an integer counting the number of *information symbol* errors that would be made should the decoder eventually select the given branch. In Figure 6.4.2, the split-state transition diagram for the code of Example 6.1 has been labeled accordingly. The rationale for this labeling is that it supplies an easy way to measure the distance between the all-zeros sequence and any other: simply multiply gains along the candidate state trajectory (which automatically adds weight exponents and thereby correctly measures path distance). At the same time, we are able to count information symbol discrepancies along any path, again due to exponent additivity. If we follow in Figure 6.4.2 the route produced by the input (1100000...), we find a total path gain of $W^6 I^2$, signifying that the total Hamming weight is 6 and that the path differs from the all-zeros sequence in two information positions. This sequence remerges with the zero state after four time steps.

To find the ***transfer function***, also called the generating function, of the graph, which provides the enumeration of all incorrect paths, we imagine injecting a constant unit input from the source node of the graph and calculate the output. (This technique is commonly employed in the analysis of linear feedback control systems, using Laplace domain gains to label paths.) Before illustrating the algebraic solution, we observe that the resulting function will be a polynomial in W and I of the general form

$$T(W, I) = \sum_{w=d_f}^{\infty} \sum_{i=1}^{\infty} A(w, i) W^w I^i, \tag{6.4.5}$$

where the coefficient $A(w, i)$ gives the number of paths having weight w and i information errors. By setting $I = 1$ in (6.4.5) and summing over i, we obtain the ***distance generating function*** or ***weight enumerating polynomial*** for the convolutional code:

$$T(W) = \sum_{w=d_f}^{\infty} A(w) W^w, \tag{6.4.6a}$$

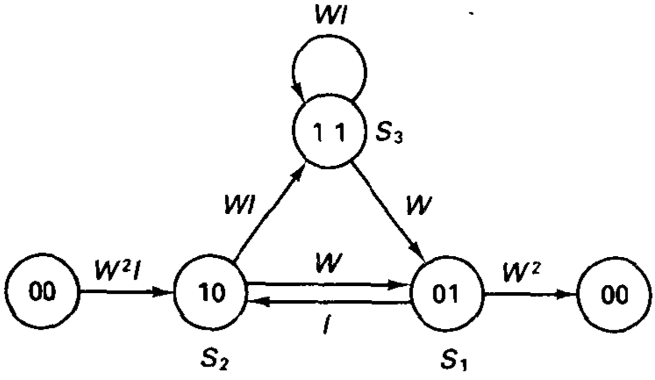


Figure 6.4.2 Split-state signal flow graph for $R = \frac{1}{2}$, $m = 2$ code.

where

$$A(w) = \sum_{i=0}^{\infty} A(w, i). \tag{6.4.6b}$$

Here, $A(w)$ is the number of error events with weight w, without regard to the number of information errors incurred. The exponent of smallest degree in (6.4.6a) is the free distance of the code, defined earlier.

Solution for the gain of such flow graphs can be done by writing node equations at each of the internal nodes and then solving a linear system of equations or by applying graph reduction procedures such as Mason's gain rules [41]. Software packages allowing symbolic manipulation, typified by Mathematica or Macsyma, allow algebraic solution for the transfer function. All these are feasible, however, only for simple codes, and a more useful numerical procedure is used for larger codes, described in Appendix 6A.1. To gain the essential ideas of transfer function calculation, we will again illustrate with the $\nu = 2$ code of Figure 6.1.1a.

Example 6.14 Transfer Function for 4-State Code of Example 6.1

Referring to Figure 6.4.2 and labeling values of states S_1, S_2, and S_3 by V_1, V_2, and V_3 and the output node as V_0, we can write the node constraints as

$$\begin{aligned} V_2 &= 1 \cdot W^2 I + I V_1 \\ V_3 &= W I V_2 + W I V_3 \\ V_1 &= W V_2 + W V_3 \\ V_0 &= W^2 V_1 \end{aligned} \tag{6.4.7}$$

Solution of these equations for the output value V_0 (by elimination or by method of determinants) and application of the definition of transfer function gives

$$\begin{aligned} T(W, I) \triangleq \frac{V_0}{1} &= \frac{W^5 I^1}{1 - 2W^1 I^1} = W^5 I^1 + 2W^6 I^2 + 4W^7 I^3 + \cdots \\ &= \sum_{w=5}^{\infty} 2^{w-5} W^w I^{w-4}, \end{aligned} \tag{6.4.8}$$

where the second step follows from long division. [The fact that the final form collapses so neatly is special to this code, and in general we must leave the result in the form (6.4.5).] This polynomial reveals the following:

1. There exists a single error event of weight 5 (the free distance event) with a single information error,
2. There are two events of weight 6, both with two information errors, four events with weight 7, carrying three information errors, and so on.

We can readily locate these by traversing the graph of Figure 6.4.2. Notice that the path enumeration does not classify events according to length, however.

Furthermore, setting $I = 1$ yields the weight enumerating function

$$\begin{aligned} T(W) &= \sum_{w=5}^{\infty} 2^{w-5} W^w \\ &= W^5 + 2W^6 + 4W^7 + \cdots. \end{aligned} \tag{6.4.9}$$

We will see shortly that P_e is bounded by mere substitution of channel-related values in $T(W)$.

Example 6.15 Transfer Function for Nonbinary Memory-1 Codes

Convolutional codes with memory $\nu = 1$ are commonly known as dual-k codes. The encoders possess q states, where $q = 2^k$, fully connected to each other in the trellis. Odenwalder [42] has shown that the transfer function for $R = \frac{1}{2}$ dual-k codes is

$$T(W, I) = \frac{(q-1)W^4 I}{1 - I[2W + (q-3)W^2]}, \qquad q = 4, 8, 16, \ldots \tag{6.4.10}$$

Upon long division, we find the minimum-weight exponent is 4, which therefore is the free distance of memory-1 codes, and the corresponding coefficient is $q - 1$. This means simply that all $q - 1$ input sequences with a single nonzero input yield a weight-4 output sequence. This is not surprising after inspection of the structure of Figure 6.1.5.

Once the weight enumerating polynomial $T(W)$ is obtained, we have the coefficients $A(w)$. Given a channel model, the relevant two-codeword error probability $P_2(w)$ may be formulated, and if $P_2(w)$ can be expressed as an exponential form, B^w, then the node error probability P_e is bounded by

$$P_e < \sum_w A(w)P_w(w) \le \sum_w A(w)B^w = T(W)_{|W=B} \tag{6.4.11}$$

Examples will follow for specific channels.

The quantity in (6.4.11) should be properly interpreted: it is an upper bound to the probability, P_e, that at any time j the decoder will have eventually selected a path splitting from the correct path at time j. The sum simply bounds the marginal probability of a node error, and it is not correct to say that the probability of having node errors at times j and $j + m + 1$, when the closest two node errors may occur, is P_e^2; that is, the node error process exhibits dependencies. In the operation of a real decoder, (6.4.11) supplies a bound on the average frequency of error events measured over time.

The *symbol error probability*, P_s, at the decoder output is usually of more interest than the node error probability. We define P_s as the expected number of symbol errors produced per trellis level, divided by the number of *information* symbols processed per level:

$$P_s \triangleq \frac{E[\text{number of symbol errors}]}{k} \le \frac{1}{k}\sum_w \sum_i i A(w, i) P_2(w). \tag{6.4.12a}$$

This is just a weighting of each term in (6.4.4) by the number of information symbol errors that would occur if the specific path is selected and then normalizing by the number of symbols processed per level of the trellis. Although of lesser interest, a lower bound on P_s is obtained by examining free distance error events, multiplying its probability of selection by the *maximum* number of information errors attached to one of these, $i_{\max}$, and then dividing by k:

$$P_s > \frac{i_{\max}}{k} P_2(d_f). \tag{6.4.12b}$$

A simple lower bound is obtained by setting $i_{\max} = 1$.

Now notice that if we formally differentiate the series $T(W, I)$ with respect to I and then set $I = 1$ we obtain the polynomial

$$\frac{\partial T(W, I)}{\partial I}\bigg|_{I=1} = \sum_{w=d_f} \sum_{i=0} iA(w, i)W^w, \tag{6.4.13}$$

whose coefficients $iA(w, i)$ are the required terms in (6.4.12a). This reveals that $T(W, I)$ provides the complete path enumeration required to upper bound both P_e and P_s.

Example 6.16 Symbol Error Probability for the $R = \frac{1}{2}$, $\nu = 2$ Code

For the code of Example 6.1 we determined in (6.4.7) that

$$T(W, I) = \frac{W^5 I^1}{1 - 2W^1 I^1} = \sum_{w=5}^{\infty} 2^{w-5} W^w I^{w-4}. \tag{6.4.14a}$$

By differentiating with respect to I, and then setting $I = 1$, we find that

$$\frac{\partial T(W, I)}{\partial I}\bigg|_{I=1} = \sum_{w=5}^{\infty} (w - 4)2^{w-5} W^w. \tag{6.4.14b}$$

Study of the polynomial coefficients reveals a *total* information weight of 1 on weight-5 error events, information weight of 4 on the weight-6 events, and so on. This is in agreement with the data of Table 6.10.

The upper and lower bounds then become

$$\begin{aligned} 1 \cdot P_2(5) < P_s &\le \sum_{w=5}^{\infty} (w - 4)2^{w-5} P_2(w) \\ &= P_2(5) + 2P_2(6) + 3P_2(7) + \cdots. \end{aligned} \tag{6.4.14c}$$

Generally, we will find that the upper and lower bounds pinch together as channel quality improves, meaning that performance is dominated by free-distance error events.

In summary, the transfer function $T(W, I)$ has been linked with the computation of node error probability and its partial derivative with the calculation of symbol error probability. We now proceed to apply this bounding procedure for several channel models of interest.

6.4.1 Performance of Binary Convolutional Codes on Nonfading Channels

The preceding formulation is known generally as the transfer function bounding approach and is general to memoryless channels through invocation of the Bhattacharyya bound on $P_2(w)$. It is most appropriate for uniform-from-the-input (UFI) channels, those for which every input has a statistically uniform connection to the output. UFI channels include binary antipodal, M-ary orthogonal, M-PSK, and others, on the AWGN channel and most other practical situations. In any case, the formulation at least provides an upper bound on more general memoryless channels. All that is required to proceed further is an expression for the two-codeword probability of error. In this section we compare the performance of various coding options on several binary channels of practical interest, illustrating the application of the upper bounding procedure.

Antipodal Signaling on the Gaussian Channel

Let's assume that the binary code symbols are transmitted using a binary antipodal signaling strategy. We recall that the probability of confusing two codewords having intercodeword Hamming distance d, when antipodal signaling is employed in additive white Gaussian noise, is

$$P_2(d) = Q\left[\left(\frac{2E_b}{N_0}dR\right)^{1/2}\right]. \tag{6.4.15}$$

This again follows from the fact that each unit of Hamming distance under antipodal signaling produces Euclidean signal space distance of $4E_s$, and E_s is in turn represented by RE_b.

By substituting (6.4.15) into the transfer function bound obtained in the previous section, we have that the node error probability is upper bounded by

$$P_e < \sum_{w=d_f}^{\infty} A(w)Q\left[\left(\frac{2E_b}{N_0}wR\right)^{1/2}\right]. \tag{6.4.16}$$

By employing an upper bound to the Q-function, $Q(x) < e^{-x^2/2}/2$ (see Section 2.2), we can (more loosely) upper bound the node error probability by

$$P_e < \frac{1}{2}\sum_{w=d_f}^{\infty} A(w)e^{-(E_b/N_0)wR}. \tag{6.4.17}$$

Comparison of this expression with (6.4.5) and (6.4.6) shows that

$$P_e < \frac{1}{2}T(W, I)_{|I=1, W=e^{-RE_b/N_0}}, \tag{6.4.18}$$

revealing that an upper bound on node error probability requires only substitution of appropriate numerical values into $T(W, I)$. This feature is really the ultimate benefit of the transfer function methodology.

The upper bound of (6.4.18) can be improved by using a tighter bound for each term in (6.4.16). Specifically, we use the inequality

$$Q(\sqrt{x+y}) \leq Q(\sqrt{x})e^{-y/2}, \qquad x, y \geq 0. \tag{6.4.19}$$

(This result is left as an exercise.) By redefining the sum index in (6.4.16), we can rewrite the sum as

$$P_e < \sum_{i=0}^{\infty} A(i+d_f)Q\left[\left(\frac{2E_b(i+d_f)R}{N_0}\right)^{1/2}\right]. \tag{6.4.20}$$

Applying (6.4.19) to this expression gives

$$
\begin{aligned}
P_e &< Q\left[\left(\frac{2E_b d_f R}{N_0}\right)^{1/2}\right]\sum_{i=0}^{\infty} A(i+d_f)e^{-iRE_b/N_0} \\
&= Q\left[\left(\frac{2E_b d_f R}{N_0}\right)^{1/2}\right] e^{d_f RE_b/N_0}\sum_{w=d_f}^{\infty} A(w)e^{-wRE_b/N_0}.
\end{aligned}
\tag{6.4.21}
$$

The last sum can again be recognized as $T(W)$ with appropriate substitution of values for W and I. Thus, the tighter, but more cumbersome, upper bound is

$$
P_e < Q\left[\left(\frac{2E_b d_f R}{N_0}\right)^{1/2}\right] e^{d_f RE_b/N_0} T(W)_{|W=e^{-RE_b/N_0}}. \tag{6.4.22}
$$

The only difference between this upper bound and that of (6.4.18) is a tightening of the multiplier coefficient; at typical code rates and SNRs this represents an important difference in apparent energy efficiency. Both upper bounds give the same asymptotic (high SNR) dependence, however.

Continuing, use of the inequality $Q(x) \le e^{-x^2/2}/(2\pi)^{1/2}x$, also developed in Section 2.2, yields the slightly more compact result

$$
P_e < \frac{1}{(2\pi d_f RE_b/N_0)^{1/2}} T(W)_{|W=e^{-RE_b/N_0}}. \tag{6.4.23}
$$

This expression is accurate when E_b/N_0 is relatively large, but obviously becomes a weak upper bound for small SNR. There, (6.4.22) is preferred.

In exactly similar manner, an upper bound on symbol error probability is obtainable for this channel. The weaker bound, analogous to that of (6.4.18) for node error probability, is

$$
P_s < \frac{1}{2k}\frac{\partial T(W,I)}{\partial I}_{|I=1,W=e^{-RE_b/N_0}}. \tag{6.4.24}
$$

A tighter upper bound (for large SNR) is obtained by the more careful bounding of the Q-function as performed previously:

$$
\begin{aligned}
P_s &< \frac{1}{k(2\pi d_f RE_b/N_0)^{1/2}}\frac{\partial T(W,I)}{\partial I}_{|I=1,W=e^{-RE_b/N_0}} \\
&= \frac{1}{k(2\pi d_f RE_b/N_0)^{1/2}}\sum_{w=d_f} N(w)W^{w}_{|W=e^{-RE_b/N_0}}.
\end{aligned}
\tag{6.4.25}
$$

Here $N(w)$ is known as the information weight of the distance-w error events, that is, the sum of the number of message 1's on all distance-w detours.

In Figure 6.4.3, upper bounds on P_s are plotted versus E_b/N_0 for $R = \frac{1}{2}$ convolutional codes with $\nu = 2$, 4, and 6, using the bound of (6.4.25). (We have not shown lower bounds, although it is known that the upper bounds are asymptotically tight at higher SNR; Exercise 6.4.6 invites a calculation of these for comparison.) For the $v = 2$ case, the bound of (6.4.24) is also depicted. Two comments are noteworthy. First, the (6.4.25) is significantly more accurate than the first in the small error probability region of normal interest. Second, the energy efficiency difference between the $\nu = 2$ and $\nu = 6$ codes is about 1.7 dB. If we had plotted results for intermediate memory orders, we would find a gain of about 0.4 dB in energy efficiency at $P_s = 10^{-5}$ for every increment of ν, or every doubling of the trellis size, for ν ranging from 2 to 6 (see also Heller and Jacobs [43]). We should not extrapolate this rule of thumb too far, for if applied to larger memory codes, we soon project performance violating the Shannon capacity bound!

If we compare the leading term of (6.4.25) with the performance of uncoded antipodal signaling, $P_s = Q[(2E_b/N_0)^{1/2}]$ and ignore the nonexponential dependence in the expression for coded symbol error probability, we see an effective gain in the energy-to-noise density ratio of $10 \log(Rd_f)$ dB. This quantity is often dubbed ***asymptotic coding gain (ACG)*** because it represents the savings afforded by coding at high SNR where the leading term of (6.4.25) is truly dominant, and even the error multiplier is insignificant in terms of energy efficiency. In this sense the interpretation of ACG is exactly as it was for block codes. This coding gain translates directly into increased link distance, smaller antennas, or smaller transmitter powers for equivalent bit rates. Or, for these parameters held fixed, coding gain represents a potential increase in bit rate. Thus, for the 4-state $R = \frac{1}{2}$ code with $d_f = 5$, we project an asymptotic coding gain of $10 \log(5/2) = 4$ dB. The actual gain at $P_b = 10^{-5}$ is, according to Figure 6.4.3, nearer 3.5 dB. The discrepancy is due to two effects: (1) at the prescribed error probability level, the error multiplier effect is not truly negligible; and (2) error events with distance greater than d_f are somewhat important. Generally, we will find that simple codes approach their asymptotic gains relatively quickly, while larger memory codes are slower to achieve the asymptotic gain. For example, the 64-state, $R = \frac{1}{2}$ code has a projected gain of $10 \log(10/2) = 7$ dB,

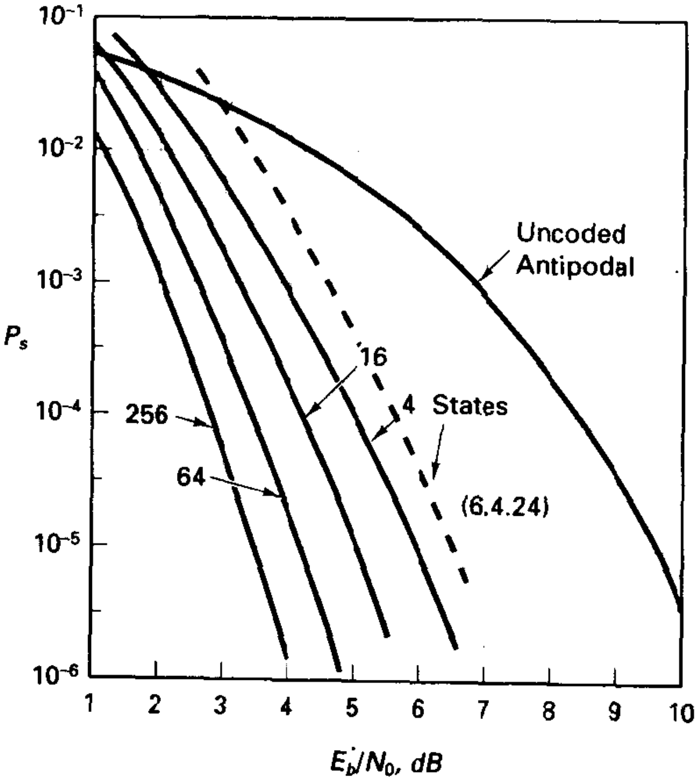


Figure 6.4.3 Probability of decoded symbol error for $R = \frac{1}{2}$ convolutional codes, state complexity varying.

but at $P_b = 10^{-5}$ the actual gain (according to the upper bound) is a more modest 5 dB. (See again Figure 6.4.3.)

A corollary of this discussion on high SNR behavior is that for increasingly large SNR the dominant error types in Viterbi decoders are free-distance error events (there may be more than one type of course). Typically, these are short detours having length m (or slightly larger) trellis levels and with a relatively few number of symbol errors. On the other hand, as SNR decreases the longer error events become more prevalent. Table 6.11

TABLE 6.11A BIT ERROR LOCATIONS AND DISTANCE OF ERROR EVENT, $E_b/N_0 = 6$ dB, $R = \frac{1}{2}$, 4-STATE CODE, $P_b = 10^{-5}$

Error Locations	Distance
77241	5
222692, 222694	6
292265	5
378008	5
383692	5
398652	5
496940	5
585536	5
648275	5

TABLE 6.11B BIT ERROR LOCATIONS AND DISTANCE OF ERROR EVENT, $E_b/N_0 = 3$ dB, $R = \frac{1}{2}$, 4-STATE CODE, $P_b = 36 \cdot 10^{-3}$

Error Locations	Distance
455,456,457,458	8
714	5
937,938	6
2888,2889	6
2924,2926,2928	7
3049	5
3605,3606,3607,3609	8
4198,4200,4201,4203	8
4271,4272	6
4287	5
5894	5
6223	5
7094	5
7737,7739	6
7887,7888	6
7991,7992,7994,7996,7998,8000,8001,8003	12
10109,10111	6
10771	5
11088,11090	6
11774,11776,11777,11779,11780,11782	10

lists bit error locations from a Monte Carlo simulation of the code of Example 6.1 at two different SNRs; we can see the predominance of single-bit (d_f) error events at higher SNR, while longer error events begin to emerge at the lower SNR.

On the coherent Gaussian channel, use of low-rate, wider bandwidth codes is advantageous, assuming that there is no bandwidth constraint. This is predictable from R_0 analysis as we have seen, and actual good codes reflect this gain. In Figure 6.4.4, transfer function upper bounds are shown for modest complexity 16-state codes with rates $\frac{1}{3}$, $\frac{1}{2}$, and $\frac{2}{3}$. Except for the high-error-rate region, where the bounds are dubious anyway, the low-rate code performs best, although not by large amounts. Asymptotic coding gain calculations also reflect this improvement, obtained by an increased product of rate and free distance; the ACGs of the rate $\frac{1}{3}$, $\frac{1}{2}$, and $\frac{2}{3}$ 16-state codes are 6, 5.4, and 5.2 dB, respectively, as reference to tables of OFD codes in Section 6.2 will show.

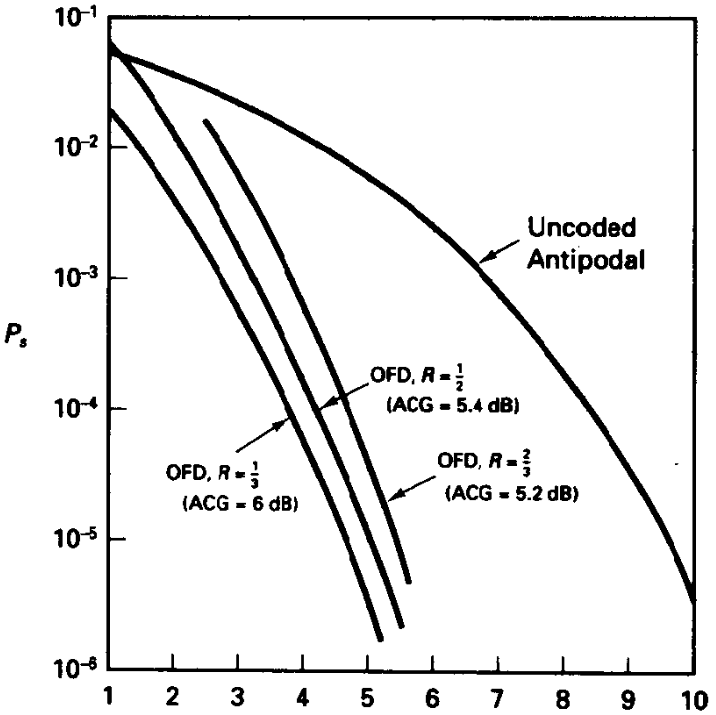


Figure 6.4.4 Probability of decoded symbol error for 16-state convolutional codes, code rate varying.

Binary Symmetric Channel

To apply the upper bound analysis for convolutional coding on the BSC, we merely need the two-codeword error probability, which for the BSC can be bounded by the Bhattacharyya bound of (4.3.15). (Somewhat tighter upper bounds can be obtained by expressing the two-codeword error probability more accurately in terms of d, as shown by Van de Meeberg [44]; however, these approaches do not lead to the compact results found in the following.) The Bhattacharyya bound on the two-codeword probability of error between the all-zeros sequence and one with Hamming weight w is

$$P_2(w) = \left[(4\epsilon(1-\epsilon))^{1/2}\right]^{w}, \tag{6.4.26}$$

where ϵ is the channel crossover probability for each code symbol. We may apply this term by term in (6.4.9) and again compare this relation to the power series for $T(W, I)$, (6.4.4), to obtain

$$P_e < T(W)_{|W=(4\epsilon(1-\epsilon))^{1/2}}. \tag{6.4.27}$$

Again, the utility of $T(W)$ appears: we merely need to substitute appropriate values for W to obtain upper bounds on desired error probabilities.

Similarly, to upper-bound symbol error probability, we have

$$P_s < \frac{1}{k}\frac{\partial T(W, I)}{\partial I}_{|I=1, W=[4\epsilon(1-\epsilon)]^{1/2}}. \tag{6.4.28}$$

This expression can be evaluated as a function of the crossover probability ϵ for any binary modulation technique, but an interesting comparison is obtained when we view the BSC as a sign-quantized (hard-decision) antipodal signaling channel. In that case

$$\epsilon = Q\left[\left(\frac{2E_b}{N_0}R\right)^{1/2}\right]. \tag{6.4.29}$$

Evaluation of the upper bound for the 4-state, 16-state, and 64-state $R = \frac{1}{2}$ codes is shown in Figure 6.4.5, and comparison with the unquantized counterparts in Figure 6.4.3 shows that a roughly 2-dB loss ensues when hard-decision decoding is performed. This also closely reflects our judgment based on the R_0 parameter, found in Section 4.5,

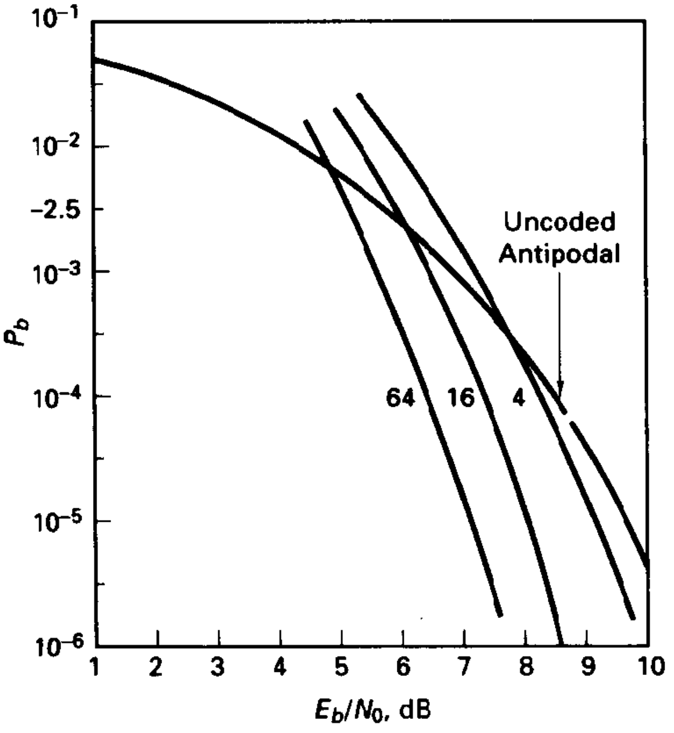


Figure 6.4.5 Probability of decoded symbol error for $R = \frac{1}{2}$ convolutional codes, state complexity varying, hard-decision coding.

and parallels similar finding for block codes on this channel. Asymptotic coding gains are even more pessimistic, projecting a 3-dB penalty for hard-decision decoding (see Exercise 6.4.7). Lowering the code rate below $\frac{1}{2}$ buys minor gains in energy efficiency for coherent detection of symbols prior to hard decisions, reflecting the predictions of Section 4.5.2.

One of the most important reasons for the preeminence of convolutional codes in the last decade on Gaussian channels is that it is only a little more difficult (by changing the metric calculation) to perform maximum likelihood (correlation) decoding instead of hard-decision decoding, thereby gaining a very inexpensive 2- to 3-dB gain in performance. Such has not been the case with block codes, although there has been important progress in soft-decision decoding of block codes.

6.4.2 Generalization to Bhattacharyya Expression

For an arbitrary memoryless channel, we can often obtain a bound on the two-codeword error probability of the form (6.4.3), and the possible cases are surprisingly varied, including noncoherent detection, fading channels, jamming channels, and various assumptions about decoder side information, and can accommodate suboptimal, non-ML decoder metrics. In any case, we can obtain the bounds

$$P_e < T(W)_{|W=B} \tag{6.4.30a}$$

and

$$P_s < \frac{1}{k}\frac{\partial T(W,I)}{\partial I}_{|I=1,W=B}, \tag{6.4.30b}$$

where B is the Bhattacharyya or Chernoff bound parameter appropriate for a given setting. Simon et al. [45] discuss this application of transfer function bounding in detail.

6.4.3 Nonbinary Convolutional Codes and Noncoherent Detection

On noncoherent channels, at least where bandwidth is not a premium, convolutional coding combined with orthogonal signaling and noncoherent detection provides high energy efficiency, as calculations of Chapter 4 indicated. The techniques can include coding a q-ary sequence with a nonbinary convolutional code, as with the dual-k codes, the codes of Ryan and Wilson [18]. Alternatively, we may simply map a binary bit stream onto a q-ary orthogonal alphabet using the binary-to-q-ary codes of Trumpis [8]. Viterbi [46] has also described orthogonal convolutional codes, a simple encoding scheme that maps a binary sequence onto a q-ary sequence using a shift register with $\log_2 q$ stages, as shown in Figure 6.4.6 for the case of $q = 32$. It is not difficult to conclude that the free distance of such an encoding scheme is $\log_2 q$ Hamming units (in symbols). The trellis also has $q/2$ states, so this scheme links the trellis size to the modulator set. Each trellis state communicates with two others.

In Figure 6.4.7, we present some results of Ryan and Wilson [18] for rate $\frac{1}{2}$ 4-ary convolutional coding with memory order 3. From Table 6.6, we find that the best code

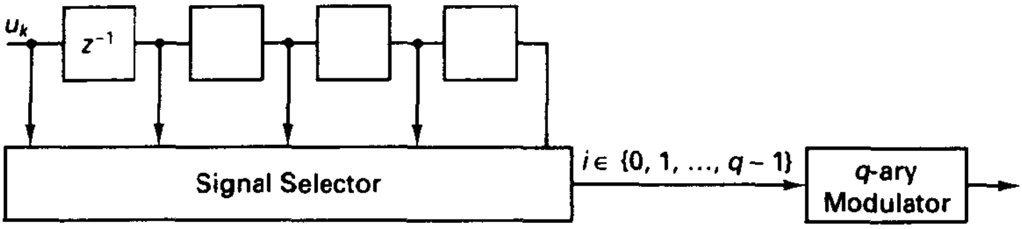


Figure 6.4.6 Convolutional orthogonal encoder.

with these parameters has a free distance 8 and that the information symbol weights at distances 8 and 9 are 39 and 42, respectively. Using these two terms in a transfer function bound (this is no longer an upper bound when terms in the sum are dropped), we approximate the symbol error probability at the decoder output by

$$P_s < 39P_2(8) + 42P_2(9),$$

where $P_2(w)$ here is the probability of confusing two sequences built with 4-ary orthogonal signals and when noncoherent detection is employed. An approximation to the optimal metric for decoding is the square-law metric; that is, the stage metric on a given path is the square of the correlator channel corresponding to the hypothesized code symbols.

Bucher [46] has also studied coding for q-ary orthogonal sets.

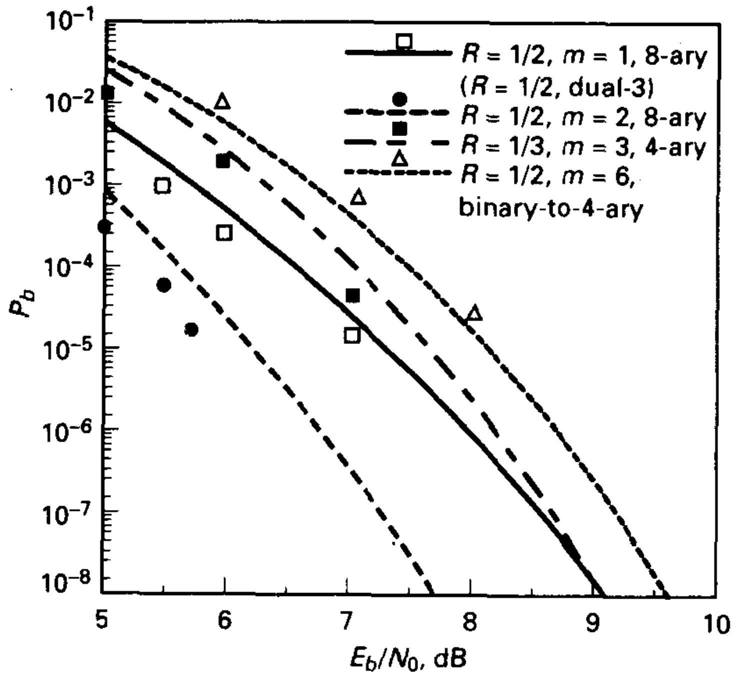


Figure 6.4.7 Bit error probability upper bounds and simulation results for nonbinary convolutional codes, q-ary orthogonal signals, noncoherent detection, AWGN channel.

6.4.4 Fading Channel Performance

On Rayleigh fading channels, it is normally hoped that sufficient interleaving of the convolutional encoder output and deinterleaving at the receiver output can establish a memoryless channel as seen by the encoder/decoder tandem. Whether this is in fact achievable depends on channel decorrelation time and the allowed transmission delay. If such is the case, then channel coding can accomplish an implicit time-diversity effect, as we saw with block coding. An advantage here is that soft-decision decoding is relatively easy to implement, and the potential diversity order is as large as the free Hamming distance of the code, rather than roughly half this value.

Modestino and Mui [48] have studied convolutional coding on Rician fading channels, in conjunction with antipodal signaling, coherent detection, and soft-decision decoding. Figure 6.4.8 shows results for the Rayleigh fading case with a rate $\frac{1}{3}$ code with and without interleaving.

Example 6.17 Fading Channel Application Using a $R = \frac{1}{2}$ Convolutional Code

Consider a tropospheric scatter application, where over-the-horizon communication is sought using transmission in the 50-MHz region, exploiting the refraction of electromagnetic energy in the troposphere. This channel is known for its strong fading effects, which are also quite slow. Suppose the desired data rate is 19,200 bps and that we opt for a memory 6, rate $\frac{1}{2}$ convolutional code. Code symbols can be interleaved using a 1024 by 64 interleaving array to combat slow fading, as shown in Figure 6.4.9. Thus, the interleaving depth is $B = 4096$ code symbols, which at the 38,400 symbol per second rate is about $\frac{1}{10}$ second, perhaps long enough so that the channel fading on symbols as seen at the output of the deinterleaver is relatively independent. The interleaver width of 64 code symbols is consistent with a decision depth in the decoder of 32 stages.

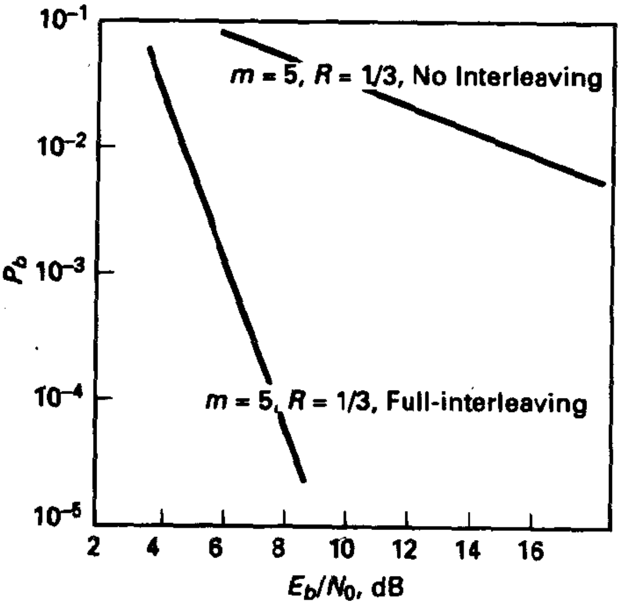


Figure 6.4.8 Transfer function upper bounds for coded PSK on Rayleigh channel. Note zero diversity without interleaving. Taken from Modestino, and Mui [48].

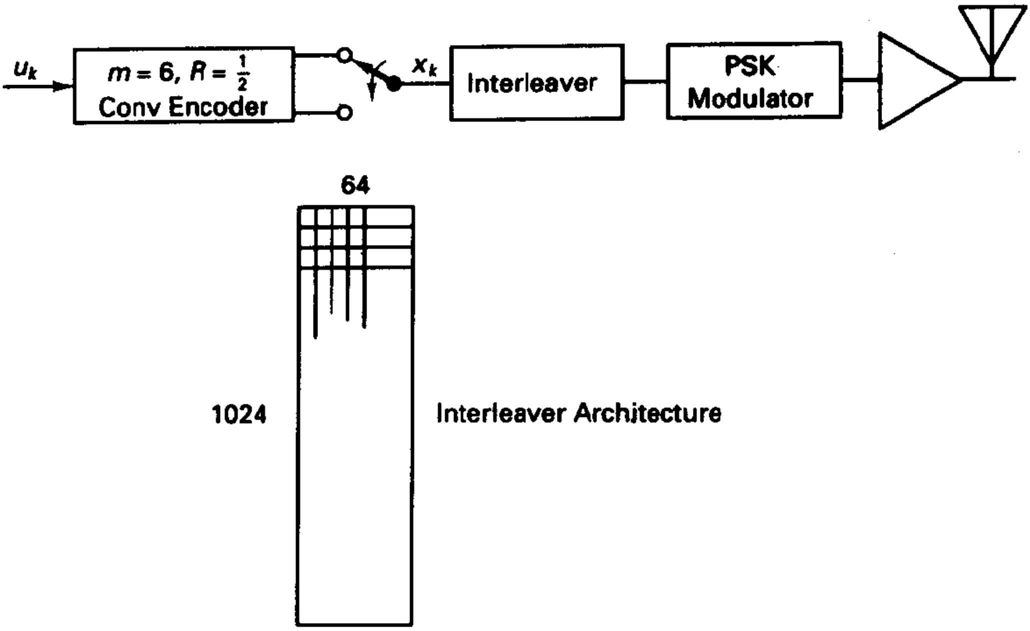


Figure 6.4.9 Transmitter for Example 6.17.

6.5 OTHER DECODING PROCEDURES: SEQUENTIAL DECODING AND FEEDBACK DECODING

Maximum likelihood decoding of trellis codes is the preferred decoding method, provided that the receiver complexity, primarily measured by the number of decoder states, is acceptably small. Ultimately, this judgment must be made in the context of requirements on decoder speed and available technology. In this section, we focus on alternative trellis decoding possibilities that forego the requirement for maximum likelihood decoding in exchange for simplified decoding; these are known as ***sequential decoding*** and ***feedback decoding***.

6.5.1 Sequential Decoding

Sequential decoding algorithms actually predate the maximum likelihood decoding algorithm of Viterbi and do not utilize a trellis perspective on the code. Instead, we simply imagine the set of all message sequences as populating a tree, rooted at the known starting state of the encoder. A sequential decoder attempts to find a high, if not highest, likelihood path from the starting state to the known ending state. It does so by a sparse search of the trellis (or tree), using one of two basic algorithms. The algorithms are as follows:

1. The Fano algorithm [51], which proceeds forward and backward in the trellis, continually trying to forge its way forward along a high likelihood route.
2. The stack algorithm, due to both Zigangirov [52] and Jelinek [53]. This algorithm is more memory intensive, maintaining candidate paths on a stack, resorting the

stack according to path likelihood, and proceeding along the currently best path. It does not, however, backtrack in the sense of the Fano algorithm.

The decoding is usually applied to packets, or frames, having length $L+m$. With either algorithm, as soon as a path with the requisite length $L+m$ is found, it is released as the decoded path.

Computational effort in sequential decoding is largely independent of the memory order of the encoder, in sharp contrast to the Viterbi algorithm. Thus, we usually find sequential decoding applied to codes with large memory order, say 20 symbols, for which the free distance is large. Furthermore, the encoders are often systematic, feed forward in form for simplicity; the loss in free distance can be made up with increased memory order without unduly affecting decoding effort.

Decoding can fail in two ways. First, the decoder may choose an incorrect path of the required length; this is normally very unlikely, however, if the free distance is large, hence the desire for large memory order. The second failure mode, and the more typical one, is due to variable computation in decoding: we cannot be sure at the outset how much computation in exploring the code tree will be required, although we can assess its statistical distribution. Essentially, the decoder can fail to decode correctly by simply running out of time; either the time to decode a fixed-length packet of data is too large, or in an indefinitely long message the decoder's penetration depth falls too far behind the incoming data, producing a buffer overflow and need to restart. These events are more probable as the channel quality degrades.

When such a decoding failure occurs, it is at least a detected error event, so a frame erasure is normally declared, and a request for retransmission can be initiated. Alternatively, the raw data can be released to the user, with a detected error stamp, if retransmission is not possible. Sequential decoding can be used effectively in conjunction with automatic-repeat-request (ARQ) protocols.

Stack Algorithm

We will discuss only the stack algorithm, although there seem to be advantages to the Fano algorithm in very high speed applications. For a discussion of the Fano sequential decoder, the reader is referred to Lin and Costello [6] and the earlier references [51]. (Incidentally, the name stack algorithm is somewhat misleading; there really is no stack in the conventional data structure sense of the term, but only a list capable of being sorted.)

Before discussing the details of the algorithm, we should note that with sequential decoders we need a fair means of scoring the relative merits of decoder hypotheses that have different lengths and thereby deciding where to proceed next in the tree search. Fano proposed that the proper branch metric should be, given a discrete-valued demodulator output $\mathbf{r}_j$,

$$\lambda\left(\mathbf{r}_j, \mathbf{x}_j^{(i)}\right) = \log_2\left[\frac{P(\mathbf{r}_j \mid \mathbf{x}_j^{(i)})}{P(\mathbf{r}_j)}\right] - nR, \tag{6.5.1}$$

where R is the code rate in bits per code symbol and n is the number of code symbols per branch in the tree or trellis. Since each branch is the result of n independent channel actions, the branch metrics can be evaluated as sums of symbol metrics if desired. In this case, the metric for scoring received symbol r against transmitted symbol x is

$$\lambda(r, x) = \log_2 \left[\frac{P(r|x)}{P(r)} \right] - R. \tag{6.5.2}$$

If the demodulator supplies continuous random variables, we use p.d.f.'s in place of probabilities in (6.5.2).

Fano proposed this metric for sequential decoding of tree codes on rather heuristic grounds, and the metric has come to be known as the Fano metric. Massey [54] justifies its adoption as follows. Suppose we have B codewords $\tilde{\mathbf{x}}^{(i)}, i = 0, 1, \ldots, B-1$, having variable lengths $n_0, n_1, \ldots, n_{B-1}$ in a hypothesis tree, and to each codeword $\tilde{\mathbf{x}}^{(i)}$ we append a *random* tail $\tilde{\mathbf{t}}^{(i)}$ selected from the code alphabet to make the net length of each codeword identical, that is, $n(L+m)$. We designate each such complete string $\tilde{\mathbf{z}}^{(i)}$. The connection with sequential decoding is that at some stage of the tree search our decoder will be evaluating paths of differing lengths, and in particular needs to ascertain which path to extend next. The most probable codeword sequence, given the complete received sequence $\tilde{\mathbf{r}}$ to date, is the natural choice, and adoption of the preceding metric properly measures a posteriori probability, given the codeword's length, and assuming equiprobable selection of all complete strings $\tilde{\mathbf{z}}^{(i)}$. There seems to be no claim that this metric will minimize average computation or that it maximizes the probability of eventually locating the ML path. However, it is certainly well motivated, and simulation has shown other metrics involving other "biases" to be inferior.

A flow chart of the stack algorithm is found in Figure 6.5.1. We begin the decoding cycle by initializing the stack with the root node and define the initial metric as 0. All q^k extensions of the root node to level 1 are evaluated by adding the branch metrics as in (6.5.1) to the initial metric. These list entries are then sorted on the basis of cumulative metric (actually, only the best must be found at this point in our presentation). The new top-of-stack entry is then extended to all its descendant trellis nodes and then deleted from the list. At this point there are $q^{2k} - 1$ list entries, having differing path lengths. Sorting finds the single best among this list, and so on. At the end of each extension cycle, we check whether the top-of-list path has length corresponding to the tree length $L + m$, and, if so, terminate the procedure, releasing the top-of-stack path as our decoded sequence. If the best path has insufficient length, we repeat another extension cycle. Notice that buffering of the received data is required while the search is proceeding.

The list size grows with each extension cycle; if $q = 2$ and $k = 1$, the list size increases by one cell for each cycle. Memory overflow due to finite memory size limitations is an obvious concern, and we address this later. If the channel is noiseless, then the top-of-stack path will have one extension that always remains the best of the list, and both stack growth and computational effort are minimal. Channel noise causes the decoder to have ambiguity about which path to extend, causing an increase in both computation and stack size.

The algorithm can include a check to see whether a newly extended path reaches a state reached by another path with the same length in the list. If this were to happen, we would say the paths were merged and thus would have identical descendant subtrees. The weaker of the two paths should be removed, as in the VA. Actually, there is no critical

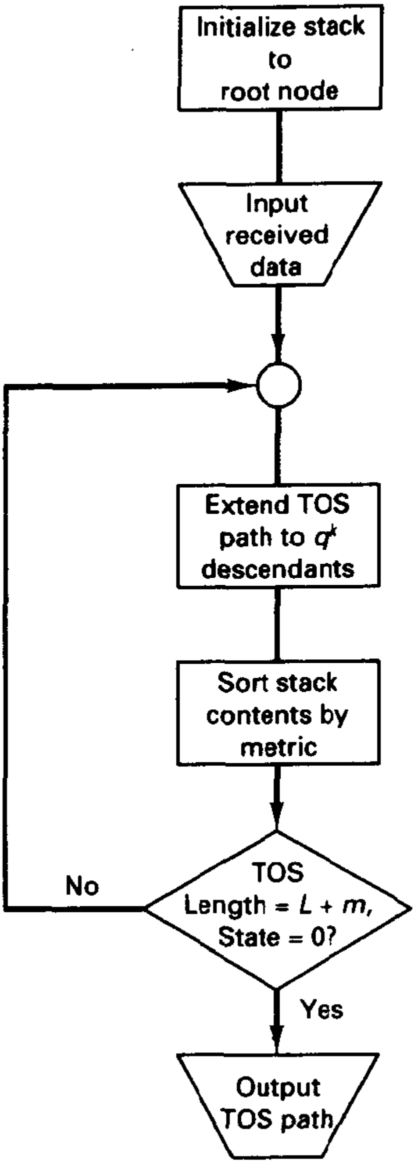

Initialize stack to root node
Input received data
Extend TOS path to q^k descendants
Sort stack contents by metric
TOS Length = $L + m$, State = 0?
No
Yes
Output TOS path

Figure 6.5.1 Flow diagram of stack decoding algorithm.

problem with maintaining both paths, except that both might be extended unnecessarily, consuming time and list space. However, this merging possibility is normally a very rare event, since with large memory codes typically used it is very unlikely that two paths that split and remerge m or more stages later will both survive on the stack. It is thus possible in practice to avoid the state merge checking.

Stack Management. For any finite list size (smaller than q^L), stack growth may eventually exceed the memory size. To avoid severe penalties in performance, we simply locate the worst as well as best paths following an extension and purge the poor contenders when the memory limit is reached. A purged path that is presently poor might eventually become the winner if the memory size were unlimited and the path was kept in contention. However, with adequate sizing of the memory this is a rare event.

The stack sorting procedure is also of concern for large lists. To find the best and $q^k - 1$ worst entries in a list of N objects requires $q^k N$ comparisons. Jelinek [53] proposed a simple scheme for avoiding the sorting complexity. We merely classify, or quantize, extended paths according to metric bins, or "buckets," and do not further sort within buckets. The extension cycle extends *any* path in the best-metric bucket; if this is empty, we proceed to the next bucket, and so on. We must keep the metric quantization fine enough so that we do not extend slightly inferior paths too often. An alternative is to sort for the best within the best nonempty bucket prior to each cycle, for this is typically a much smaller number.

Computation Variability. Computation to decode a given message using the stack algorithm (or Fano's algorithm) is a random variable induced by the channel's noise mechanism. We typically measure the computational effort of a stack decoder by the number of top-of-stack path extensions. It is known, for example [6], that for a *large code tree*, the distribution for this random variable is

$$P(C \geq c) \approx Ac^{-\rho}, \qquad c \text{ an integer,} \tag{6.5.3}$$

for c large, that is, in the tail of the distribution, with A a proportionality constant dependent on packet length. This distribution is known as the ***Pareto distribution*** and is notably heavy tailed. The exponent ρ in (6.5.3) is given by the solution to

$$E_0(\rho) = \rho R, \tag{6.5.4}$$

where $E_0(\rho)$ is the Gallager function introduced in Section 4.4.

Given the probability mass function in the tail of (6.5.3), we find that the *expected* number of computations is unbounded if $\rho < 1$.[11] In such situations, we must anticipate serious delay due to large computation and memory overflow problems for continuously arriving data. Thus, the critical rate R for finite expected computation is

$$R < E_0(1) = R_0, \tag{6.5.5}$$

where R_0 is the channel quality parameter introduced in Chapter 4. Because of this computational significance, this parameter was originally designated, and still is by many, the *computational cutoff rate*, R_{comp}, since coding with rate greater than this value suggests decoding problems, on average, for a sequential decoder. Notice that we might still successfully decode when $R > R_0$, particularly with short packets, but the expected decoder effort rapidly increases in this region. This is another justification for trying to engineer a modulator/channel/demodulator that provides largest R_0.

Example 6.18 Sequential Decoding on a BSC

Suppose that we have a BSC available for coded transmission, with $\epsilon = 0.03$. Use of the R_0 expression for the BSC developed in Example 4.7 gives

$$R_0 = 1 - \log_2\left[1 + (4\epsilon(1-\epsilon))^{1/2}\right] = 0.576 \text{ bit/channel symbol.} \tag{6.5.6}$$

This suggests that $R = \frac{1}{2}$ codes are feasible from a decoder computation standpoint. (A rule of thumb seems to be that operating sequential decoders with $R > 0.9R_0$ is to be avoided.)

Here the channel inputs and outputs are binary, and the per-symbol metrics calculated according to (6.5.2) are 0.452 and -4.52 for the $x = r$ and $x \neq r$ conditions, respectively.

[11] For any finite frame size, the computation is finite.

It is possible to scale and round these real metrics to integer values, and in this case a good choice is +1 and −10 for the two cases.

Let's reprocess the received data of Example 6.12, corresponding to a rate $\frac{1}{2}$ code with memory 2 and a message length of $L = 4$ bits, terminated with two zeros. The binary-quantized version of the data was (01, 10, 00, 00, 11, 00). We will use the per-symbol metrics listed previously, appropriate for $\epsilon = 0.03$, although the reception of four channel errors in 12 bits is not very consistent with this model.

In Figure 6.5.2, we show the stack contents after sorting, listing the information sequences (oldest bit on left) and the associated cumulative metrics. Notice that the decoded path is eventually 101000, agreeing with the VA solution in this case, and that a total of 7 top-of-stack node expansions was made, although the decoder's next choice would have been to retreat to an earlier-explored path of length 4. This compares with a minimum of six extensions for a no-channel-error condition. Notice that the resolution of metric ties influences the actual amount of computation here.

We might conclude that the stack decoder achieved the same result with far less computation than the Viterbi algorithm, but the comparison is not easily made. First, this is a short example. Second, stack sorting is not a trivial exercise. Finally, there is no guarantee that the sequential decoder will produce the ML path.

0; -9	1; -9	10; -7	101; -5
1; -9	00; -18	00; -18	00; -16
	01; -18	01; -18	01; -16
		11; -29	100; -27
			11; -29

1010; -14	10100; -12	101000; -10	(Done)
1011; -14	1011; -14	1011; -14	
00; -16	00; -16	00; -16	
01; -16	01; -16	01; -16	
100; -27	100; -27	100; -22	
11; -29	11; -29	11; -24	
	10101; -34	101001; -32	
		10101; -34	

Figure 6.5.2 Stack evolution, Example 6.18, hard-decision metric: $\lambda(r, x) = 1$, $r = x$; $\lambda(r, x) = -10$, $r \neq x$.

Further discussion of sequential decoding, particularly pertaining to high-rate convolutional codes, is found in [55]. Figure 6.5.3 shows distributions of computation in processing fixed-length frames for varying channel quality. Note that when the R_0 parameter drops to near R computational effort increases dramatically.

6.5.2 Feedback Decoding

Feedback decoding is a procedure designed for hard-decision applications, say for a BSC, and one which has fixed computation per decoded symbol. The essential idea is that channel errors will produce a syndrome sequence when the parity check equations

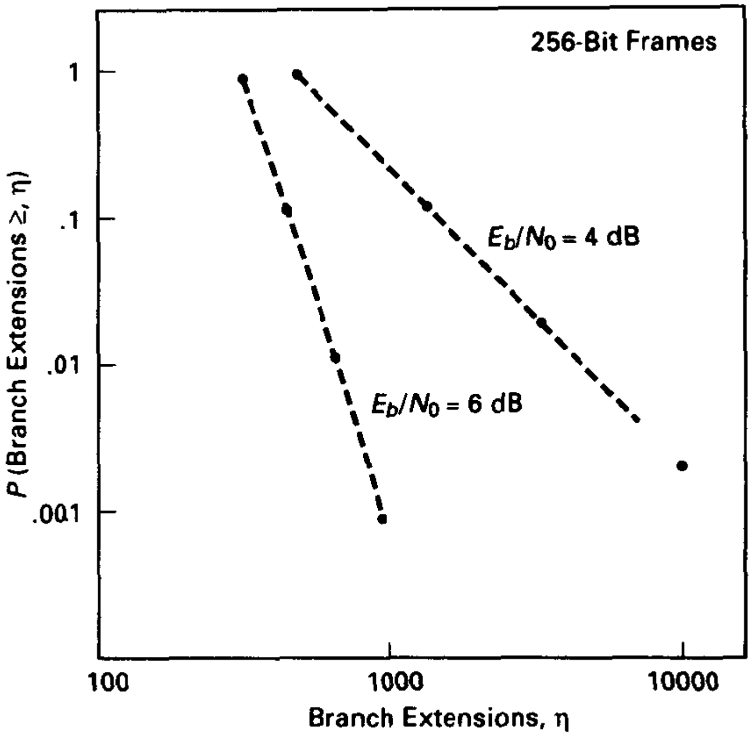


Figure 6.5.3 Distribution of computational effort. Sequential decoding with hard-decision decoding, E_b/N_0 varying.

of the code are tested, and as with linear block codes, a processing of this syndrome can estimate whether a symbol error exists in a given position and repair it.

To begin, suppose the encoder is a rate $\frac{1}{2}$ systematic feed-forward encoder with memory order m. Let the channel error sequences be $e_j^{(0)}$ and $e_j^{(1)}$ on the information and parity bit positions, respectively. Computation of the syndrome sequence in the systematic case amounts to reencoding the received information stream (perhaps containing errors) and adding this to the received parity stream, as shown in Figure 6.5.4. The

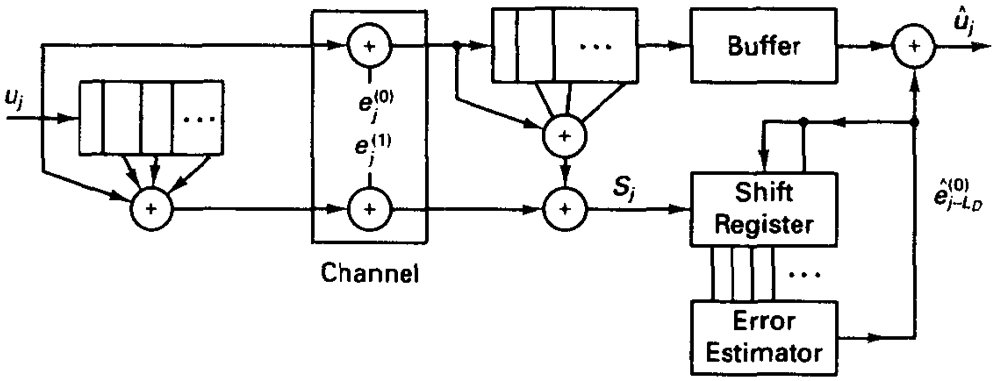


Figure 6.5.4 Feedback decoder for Example 6.19.

syndrome bit at a given time will be a modulo-2 sum of the error sequence in certain positions. By collecting syndromes over a window of length $L_D \geq m$, we can make inference about the error sequence. Specifically, we can precompute a table addressed by the 2^{L_D} possible syndromes that outputs the *single* bit corresponding to the error estimate in the information position at time $j - L_D$. (Notice the similarity with Meggitt decoding of cyclic codes in this regard.) If we infer an error occurred in the given position, we must patch the syndrome sequence to remove the influence of this suspected error.

The complexity of feedback decoding evidently resides in the table-lookup function, and this procedure is feasible only when the table size is reasonable. Modern gate array technology is one way to implement the required Boolean function.

The performance of feedback decoding is governed by the column distance function at the depth corresponding to L_D. Specifically, if the column distance function equals d at depth L_D, then a feedback decoder is capable of correcting

$$t_{\text{feedback}} = \left\lfloor \frac{d-1}{2} \right\rfloor \qquad (6.5.7)$$

or fewer errors in the span of nL_D bits influencing the syndrome sequence. Thus, for these decoders it is clearly desirable to maximize $d_c(L_D)$ or, perhaps more to the point, to minimize, over choice of encoders, the L_D needed to achieve a given distance.

The procedure extends readily to systematic rate $\frac{k}{n}$ codes and even to nonsystematic codes. However, at least for delay $L_D = m$, we can find a systematic code capable of achieving maximal column distance $d_c(L_D)$, so there is no apparent benefit to use of nonsystematic codes. For larger decoding delay, nonsystematic feed-forward codes can be superior, as we have claimed for free distance, but the decoder complexity in this regime is such that Viterbi decoding should probably be adopted anyway.

Error propagation is a possible phenomenon with feedback decoding, for if a correct bit is mistakenly altered, the syndrome sequence is further corrupted. The error propagation is, however, not indefinite provided that the decoder delay is at least m. Further discussion on feedback decoding is given by Heller [56] (see also [50]).

Example 6.19 Feedback Decoding for Encoder of Example 6.5

In Figure 6.1.1b we showed a memory-5, rate $\frac{1}{2}$ systematic encoder and described parity check computation in Example 6.5. This encoder happens to have the largest attainable column distance at depth 6, that is, 5. (This is also called the minimum distance.) This distance information implies that the decoder is capable of correcting up to two errors in the 12 bits residing in a decoder span of 12 channel bits.

We illustrated the syndrome former in Figure 6.1.7 and from this diagram can conclude that the syndrome sequence is given by

$$s_j = e_j^{(0)} + e_j^{(1)} + e_{j-1}^{(0)} + e_{j-2}^{(0)} + e_{j-4}^{(0)} + e_{j-5}^{(0)}. \qquad (6.5.8)$$

Similarly, we can express each of the syndrome bits at times $j-1, \ldots, j-5$, giving a linear relation between these 6 syndrome bits and the 12 possible channel error positions.

To determine the decoding table, we assume that all errors prior to time $j-5$ have been repaired by previous decoding cycles, and thus these bit positions can be eliminated from the syndrome equations. In this case, the resulting syndrome equations become

$$\begin{aligned}
S_j &= e_j^{(0)} + e_j^{(1)} + e_{j-1}^{(0)} + e_{j-2}^{(0)} + e_{j-4}^{(0)} + e_{j-5}^{(0)},\\
S_{j-1} &= e_{j-1}^{(0)} + e_{j-1}^{(1)} + e_{j-2}^{(0)} + e_{j-3}^{(0)} + e_{j-5}^{(0)},\\
S_{j-2} &= e_{j-2}^{(0)} + e_{j-2}^{(1)} + e_{j-3}^{(0)} + e_{j-4}^{(0)},\\
S_{j-3} &= e_{j-3}^{(0)} + e_{j-3}^{(1)} + e_{j-4}^{(0)} + e_{j-5}^{(0)},\\
S_{j-4} &= e_{j-4}^{(0)} + e_{j-4}^{(1)} + e_{j-5}^{(0)},\\
S_{j-5} &= e_{j-5}^{(0)} + e_{j-5}^{(1)}.
\end{aligned} \tag{6.5.9}$$

Notice that a single error in $e_{j-5}^{(0)}$, corresponding to the information position about to emerge from the buffer, causes a syndrome pattern of (110111). We can show that the 64 syndrome patterns divide into two distinct classes, a class for which we decide to change the bit $r_{j-5}^{(0)}$ and a class for which we do not. All single errors and double errors involving $e_{j-5}^{(0)}$ are attached to distinct syndromes in the first class, while single and double errors not involving this bit are attached to syndromes in the other class. (These errors will be repaired at a later time.) The correction circuitry can be implemented as a 6-input, 1-output Boolean network. It should be clear that this form of decoder is capable of operation at very high speeds. Furthermore, by interleaving to appropriate depth, burst-error correction at high speeds is quite feasible.

For small ϵ, the output error probability can be approximated by

$$P_b \approx C\epsilon^3 \tag{6.5.10}$$

where C is a multiplier factor, since some three-error events can cause a bit decoding error.

6.6 TRELLIS CODING WITH EXPANDED SIGNAL SETS FOR BAND-LIMITED CHANNELS

From a classical perspective, coding for error control has implied an increase in spectrum bandwidth in return for some decrease in required signal-to-noise ratio, the latter called *coding gain*. The reason for bandwidth expansion is that the rate of the encoder output, in symbols per second, is greater than the rate at the encoder input, assuming identical alphabets, by a factor equaling the inverse of the code rate, R. Thus, rate $\frac{1}{2}$ encoding increases the transmitted symbol rate by a factor of 2 relative to the uncoded symbol rate. Furthermore, although any encoding (block or trellis structured) introduces statistical dependencies into the output symbol stream, good codes are rather well approximated as having output sequences that are independent, but at the increased rate. Thus, assuming that both coded and uncoded alternatives employ the same modulator signal set, we can infer that the spectrum of the coded signal is merely rescaled in frequency by the inverse code rate. Usually, this is in fact exactly the case. This has led to a misunderstanding that error-control coding could only provide significant gains in energy efficiency when the possibility of bandwidth expansion is present.

An alternative viewpoint on coding was introduced by Ungerboeck [4], which rather revolutionized thinking about coding and the effect on bandwidth. The general topic is now referred to as ***trellis coded modulation***, or TCM. Ungerboeck reasoned that the modulator symbol set could be enlarged when coding is adopted, relative to that needed by uncoded signaling, and redundant modulator sequences selected with memory chosen from this largèr constellation. If the signal set *dimensionality per information bit* remains unchanged, then, to at least first order, the power spectrum remains identical to that of the uncoded signal. Surprisingly, perhaps, actual coding gains on the AWGN channel can nonetheless be quite impressive, 3 to 6 dB. In hindsight, this notion was waiting for all to exploit, foretold in the form of channel capacity and/or the R_0 results we have seen in Chapter 4! The key realization is that coding and modulation should be performed with full attention to Euclidean distance between coded modulator output sequences, without any particular attention to Hamming distance at the encoder output. A comprehensive survey dedicated to this topic is [57].

6.6.1 Set Partitioning

The essence of trellis coding onto expanded signal sets involves the notion of ***set partitioning***. Consider a modulator constellation having M points in an N-dimensional space. (Initial descriptions of Ungerboeck were restricted to one- and two-dimensional spaces.) The constellations may be M-ary AM in one dimension, M-ary QAM or PSK in two dimensions, or more generally any regular arrangement of points in N dimensions, such as subsets of N-dimensional lattices. In the latter context, we refer to the process as lattice partitioning.

We first partition the original set into p_1 equal-sized disjoint subsets, or ***cosets***, each of size M/p_1 points and denote these subsets $A_1, A_2, \ldots, A_{p_1}$. This partitioning is done so that within each subset the minimum Euclidean distance between signal points is maximal for the adopted subset size and uniform across subsets. (This will always be possible with initial constellations having standard symmetry, and if the constellation is a lattice, the lattice ĉoset decomposition provides the recipe.) Next, each of these subsets are further split into p_2 subsets, denoted $B_0, B_1, \ldots$. Proceeding recursively in this manner, the original constellation may be eventually decomposed into single-point subsets. Typically, the splitting factors p_i are equal, and often two, but this is not necessary.

Three examples should serve to clarify set partitioning.

Example 6.20 Partitioning of 8-PSK

The 8-PSK constellation and its natural partitioning sequence are shown in Figure 6.6.1. In each step, we employ splitting by two, so the subset sizes shrink by two, and we may notice that the first level of partitioning produces two rotated QPSK sets, the next level produces four binary PSK sets, and the final level has single-point subsets. The sequence of right/left branches in the partitioning tree produces binary 3-bit labels to each of the original 8-PSK points; clearly, there are many equivalent labelings, and we have selected one that amounts to a natural binary progression as we move around the constellation circle. In effect, this labeling decision produces a mapping from binary 3-tuples to signal points, and Ungerboeck dubbed this "mapping by set partitioning." The actual labeling of signal points is not crucial, however, until we proceed to build encoders and decoders.

More importantly, note as the partitioning proceeds the intrasubset minimum distance increases from the 8-PSK distance to the QPSK distance, to the distance of antipodal signals. The sequence of squared minimum distances is thus $0.585E_s, 2E_s$, and $4E_s$, where E_s is

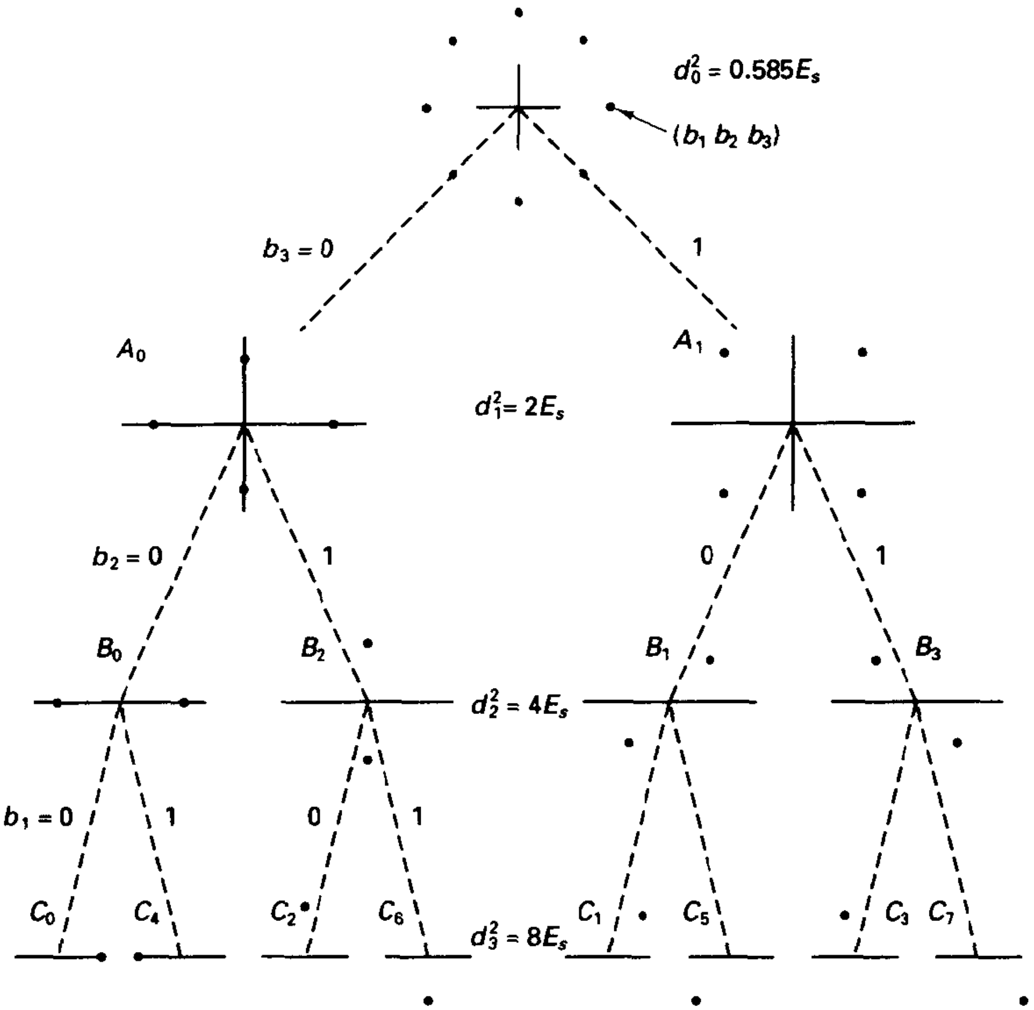


Figure 6.6.1 Set partitioning for 8-PSK with associated bit labeling.

the energy attached to each symbol. We can attach infinite intraset distance to singleton sets.

Example 6.21 Partitioning of the Two-dimensional Lattice Z^2

Another important example is that of partitioning the two-dimensional integer lattice Z^2, which we shall illustrate with 16-QAM. Figure 6.6.2a depicts the partition sequence. Notice here that the progression of intrasubset squared distances increases by 2 at each partition level, according to $4a^2, 8a^2, 16a^2, \ldots$, where $2a$ is the constellation spacing along each signal-space axis. In Figure 6.6.2b, a similar partitioning is depicted for the 32-cross constellation. Notice that the same progression of minimum distance occurs, and the sets have the same size at all levels, but the subsets are not congruent. This is no real issue, however.

Example 6.22 Partitioning of the Four-dimensional Lattice D_4

We recall from Chapter 3 that the best lattice packing in four dimensions is produced by the lattice arrangement commonly designated $\mathbf{D}_4$. The easiest recipe for this lattice is to

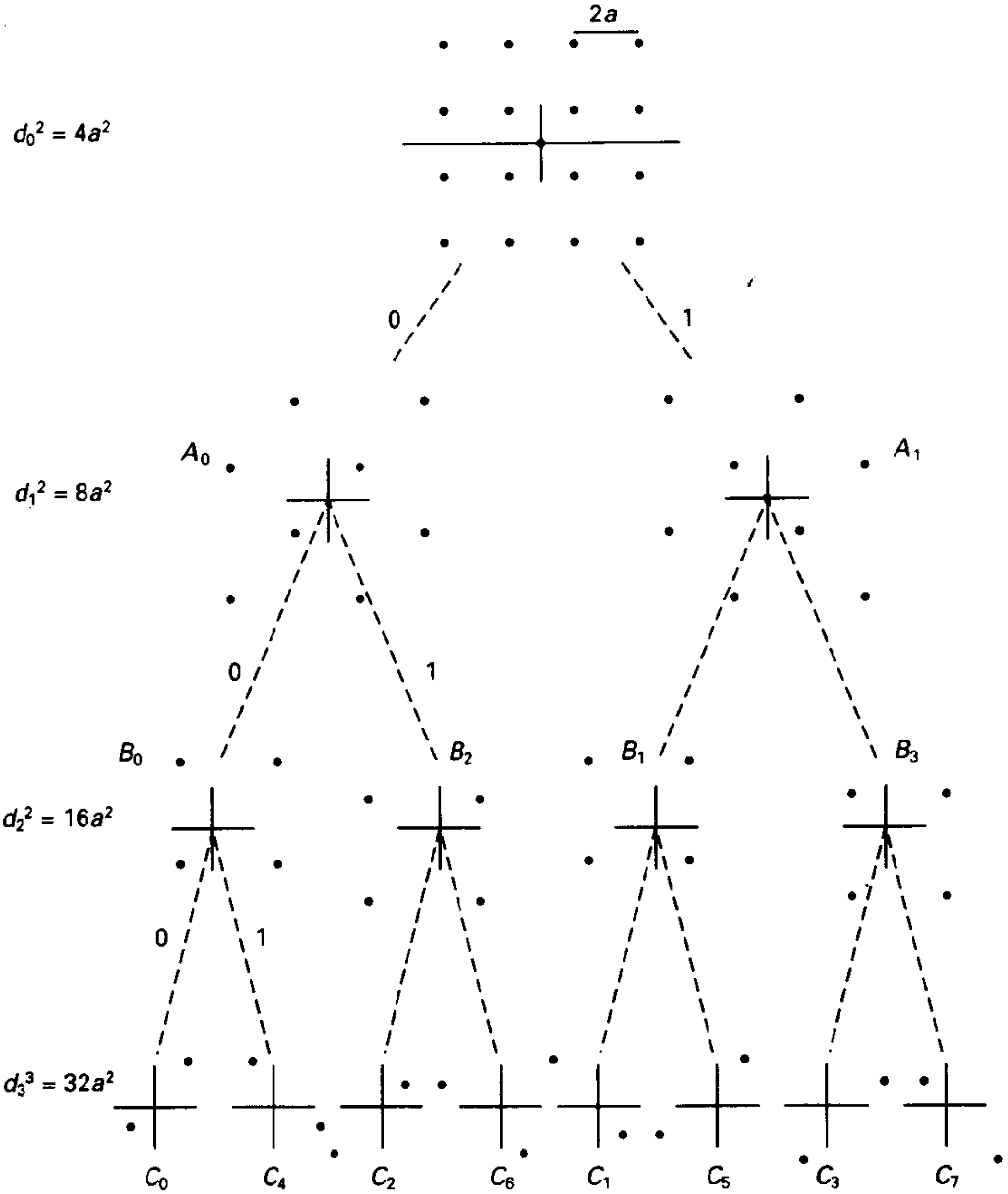


Figure 6.6.2a Set partitioning for 16-QAM.

consider the lattice points, or signals, to be all combinations of four-tuples of the form $(x_1, x_2, x_3, x_4 \mid \Sigma x_i = 0, \mathrm{mod}\, 2)$. To center this lattice at the origin, it is common to translate the lattice by adding the vector $\left(\frac{1}{2}, 0, \frac{1}{2}, 0\right)$ to each point.

In Chapter 3, we constructed a 256-point constellation by selecting low-energy vectors from this translated lattice. In an uncoded system, 8 bits could be communicated by transmission of one of these four-dimensional signal points. If we wish to apply trellis coding to this modulation option, however, we can proceed to partition the 256-point set. It is known that $\mathbf{D}_4$ partitions into four scaled and rotated copies of itself; that is, upon four-way division, each subset has the same geometry as the parent set. Thus, the 256-point

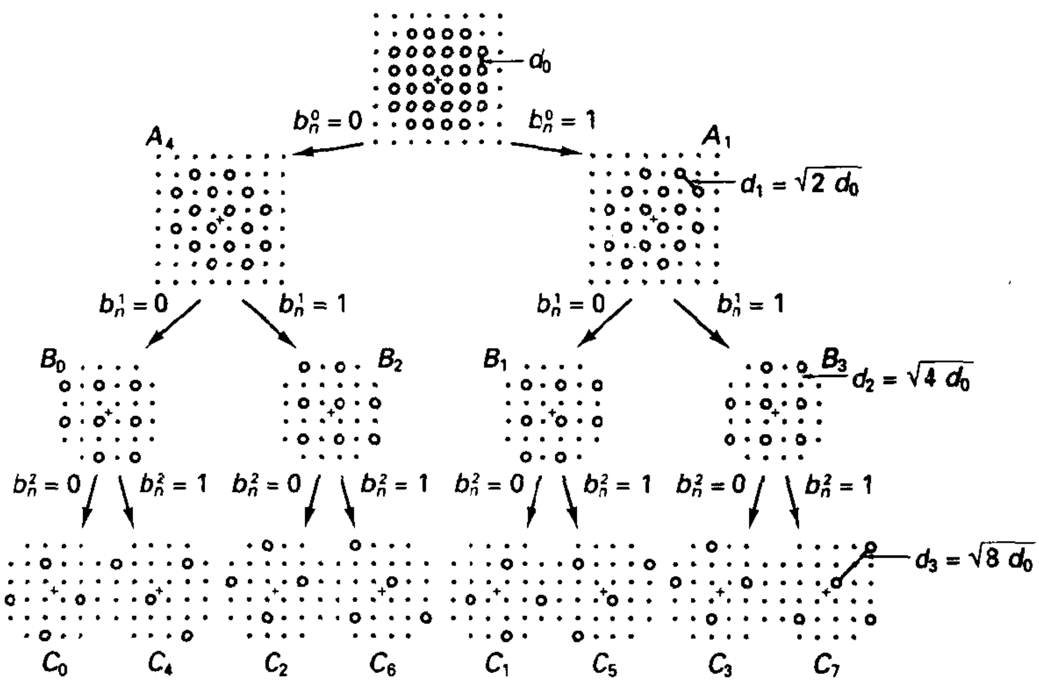
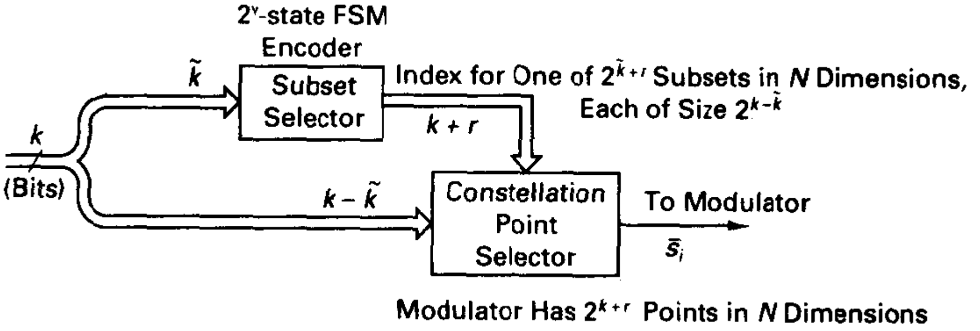


Figure 6.6.2b Set partitioning of the 32-CROSS signal set.

constellation divides into four 64-point D_4 subsets, and these generate sixteen 16-point subsets, and so on. Each time this division occurs, the minimum intrasubset squared distance increases by 2.

Here is the essence of trellis coding for band-limited channels, schematically depicted in Figure 6.6.3. We begin with a desire to communicate k bits per modulator symbol and adopt a modulator constellation that is slightly bigger than 2^k. (Often, the size is exactly 2^{k+1}, and we speak of expanding the constellation by a factor of 2 over that needed for uncoded transmission.) Next, we adopt some $\tilde{k} \leq k$ to be the number of input bits that enter the encoder and influence part of the state. This leaves $k - \tilde{k}$ bits as uncoded bits. The encoder is typically a binary convolutional encoder, which adds $r = 1$ bit of redundancy, producing $\tilde{k} + 1$ output bits.



2^ν-state FSM Encoder
$\tilde{k}$
Subset Selector
Index for One of $2^{\tilde{k}+r}$ Subsets in N Dimensions,
Each of Size $2^{k-\tilde{k}}$
$\tilde{k}+r$
k
(Bits)
$k-\tilde{k}$
Constellation Point Selector
To Modulator
$\bar{s}_i$
Modulator Has 2^{k+r} Points in N Dimensions

Figure 6.6.3 Generic trellis coded modulation structure.

To marry this encoding with the modulator, we partition the original modulator constellation to obtain $2^{\tilde{k}+1}$ subsets, each with $2^{k-\tilde{k}}$ points. (Note that the total number of points in the constellation is thus 2^{k+1}.) Furthermore, the subsets are labeled by a vector of $\tilde{k}+1$ bits produced by the encoder, and we may say the encoder has chosen a subset for modulation. The label is exactly the $\tilde{k}+1$ highest bits on the set partitioning tree; that is, the encoder specifies the modulator input bits $b_1, b_2, \ldots, b_{\tilde{k}+1}$ at time j. The remaining $k - \tilde{k}$ uncoded bits then merely select a member of this chosen subset for transmission. To put this another way, the coset sequence selection has memory and redundancy, but the selection of a point within a coset does not. These signals are relatively easy to distinguish in noise, however.

Obviously, there are several design factors, including the division of input bits between the two paths, the encoder memory, and the modulator constellation type. These distinguish different TCM schemes.

6.6.2 Hand Design of Codes

Simple trellis codes may be designed by hand and already offer impressive coding gains. This design recipe is not suggested for exhaustive code studies, but is an important illustrative tool. The procedure begins with selection of a trellis size measured in number of encoder states. We denote the trellis size as $S = 2^{\nu}$, so ν again corresponds to the number of binary memory elements in the trellis encoder. We also specify a throughput k in units of bits per trellis level. Thus, 2^k is the number of trellis branches leaving and merging with each trellis state. These total number of arcs can be allocated in several ways in general. For example, if $k = 3$, the eight required graph transitions could be divided as eight single branches, four groups of two, or two groups of four. These choices correspond to $\tilde{k} = 3, 2$, and 1, respectively. Knowing which is best is not immediately obvious, but given a trial choice, we merely assign constellation subsets of proper size to these various state transitions. Ungerboeck proposed some heuristic design rules that have basically withstood more general attacks allowed by computer search. These rules are as follows:

1. Employ all subsets equally often in labeling the trellis.
2. For subset assignments that share a common splitting state or merging state, choose subsets between which the minimum *interset* distance is largest.

(For small trellises, where subsets are assigned only once each, it can be seen that rule 2 cannot generally be satisfied.) These rules still allow considerable latitude in labeling of a large trellis, pointing to the need to examine many codes.

Example 6.23 4-State Code for $k = 2$, 8-PSK

This is the archetypal example of trellis coded modulation, supplying the same spectral efficiency as uncoded QPSK, but with roughly 3 dB coding gain. The throughput is $k = 2$ bits per modulator symbol, and a convenient expanded constellation is 8-PSK. The requisite number of branches per trellis state is four, and we propose to assign branching as two sets of two, as shown in Figure 6.6.4. With reference to Figure 6.6.1, this means that antipodal

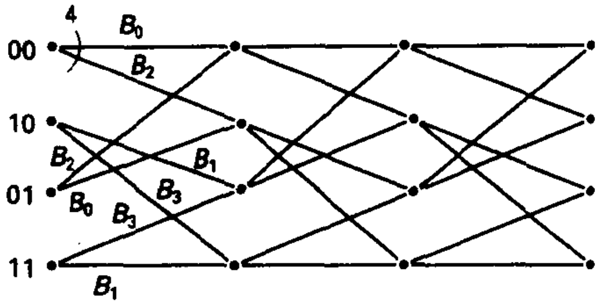


Figure 6.6.4 Trellis labeling for 4-state, 8-PSK code. Subsets have size 2.

signal sets B_0, B_1, B_2, and B_3 will be assigned to various state transitions. Also, this choice has adopted $k = \tilde{k} = 1$, so one of the input bits is left uncoded.

Because there are four antipodal subsets in the constellation and the graph requires eight subset assignments, each subset will be used exactly twice. Following rule 2, we assign as subsets to trellis branches leaving (or merging to) common states those that differ by a 90° rotation. This ensures that the minimum distance between subset members is maximized for the initial and final stages of multistage error events.

Perhaps the most intriguing aspect of this trellis code design is the presence of ***parallel transitions***; that is, any given state can transition to its next-state mates by two paths in this case. An immediate corollary is that single-step error events are possible in ML decoding. We did not find this in the case of standard binary convolutional codes, which were designed for maximum Hamming distance, principally because the free Hamming distance could otherwise never exceed n, the number of output symbols per shift time. Under the criterion of Euclidean distance maximization, however, parallel transition designs are common, at least for relatively small trellises, and for large k.

It should be clear that this example code, and generalizations following Figure 6.6.3, are finite-state codes, amenable to decoding by the Viterbi algorithm. The only novel aspect to be incorporated in decoding is the possibility of parallel transitions, but this is simply incorporated. The principle of optimality is satisfied if we do decoding in two steps:

1. In each subset, find the most likely constellation point to have produced the given observation and call these *subset winners*.
2. Perform standard trellis decoding using only these subset winners and their maximum likelihood metrics.

In other words, decoding can be implemented with a standard trellis decoder having $2^{\tilde{k}}$ arcs entering and exiting trellis states, preceded by a subset preprocessor that supplies subset winner information.

For such a decoder, many types of decoding error events are possible, and we again need to enumerate the various Euclidean distances, the number of sequences having these distances, their information weight, and so on. For high signal-to-noise ratio, the ***free distance*** of the code, defined again as the minimum Euclidean distance between any two distinct coded sequences, is the principal figure of merit. To determine the free distance of the code in Example 6.23, we observe that the Euclidean distance between single-step detour events is $d_1^2 = 4E_s$. (This holds regardless of the transition we study, by symmetry of the subsets.) Two-step error events are not allowed by inspection of the trellis, but three-stage and longer error events are possible. The minimum distance among all three-stage error events is

$$d_3^2 = 2E_s + 0.585E_s + 2E_s = 4.585E_s. \tag{6.6.1}$$

It may be shown by exhaustion that all longer error events have still larger distance. Thus, we find that the free distance of the code is dominated by one-step error events, and

$$d_f^2 = 4E_s \qquad (\text{four} - \text{state code}, 8 - \text{PSK}). \tag{6.6.2}$$

Since each code symbol carries the energy of *two* information bits, $E_s = 2E_b$, and thus $d_f^2 = 8E_b$. The two-codeword error probability for this free distance error event is $Q[(d_f^2/2N_0)^{1/2}]$, which is exactly 3 dB more efficient in energy utilization than uncoded PSK/QPSK.

As with conventional binary convolutional codes, we can adopt a union bound approach to computing performance, say node error probability. We can count that for any transmitted path in the trellis,[12] there is precisely one single-step error event, four three-step events with squared distance $4.585E_s$, and so on. The multiplicity of four follows from two neighbors in each subset at squared distance $2E_s$ on the split and merge segments, but only one neighbor with squared distance $0.585E_s$ on the middle segment of the three-step event. Thus, the first two terms in a union bound for node error probability are

$$\begin{aligned} P_e &< Q\left[\left(\frac{d_1^2}{2N_0}\right)^{1/2}\right] + 4Q\left[\left(\frac{d_3^2}{2N_0}\right)^{1/2}\right] + \cdots \\ &= Q\left[\left(\frac{4E_b}{N_0}\right)^{1/2}\right] + 4Q\left[\left(\frac{4.585E_b}{N_0}\right)^{1/2}\right] + \cdots. \end{aligned} \tag{6.6.3}$$

This points to an asymptotic coding gain, relative to uncoded QPSK that has the *same bandwidth* equaling 3 dB. Note that nonfree-distance events may have only slightly larger distance and thus contribute significantly to error probability at moderate SNR.

Furthermore, because there is only a single (uncoded) bit error attached to the dominant single-step error event, the bit error probability is given asymptotically by

$$P_b \approx Q\left[\left(\frac{4E_b}{N_0}\right)^{1/2}\right] \tag{6.6.4}$$

Again, this is precisely 3 dB superior to uncoded QPSK.

We recall that our original trellis architecture was chosen to have two-sets-of-two branching. An alternative possibility has four-sets-of-one branching, for which single-point subsets would be assigned to each state transition. Now, the one-stage error events disappear, but it can be found by trial and error that any assignment of points to the trellis leads to a two-stage error event with distance $d_2^2 = 2.585E_s$, and thus this trellis structure is suboptimal. This provides a convenient point to mention that the best binary

[12] This symmetry condition will hold whenever we are dealing with a design involving a uniformly partitioned constellation and a convolutional encoder.

Hamming distance code with $R = \frac{2}{3}$ has trellis structure identical to the latter choice. Even with the best assignment of 8-PSK points to output 3-tuples, we produce an inferior code for the AWGN channel by taking an optimal binary Hamming distance code and mapping the code vectors onto the modulator set.

If we consider the design of 8-state codes, it should be obvious that, if we hope to have increased free distance and thus larger asymptotic coding gain, something must be done to rid the trellis of the dominant one-stage error events. Now, the way to do so is to switch to four-sets-of-one branching. Correspondingly, $\tilde{k} = k = 2$ for this design, since both input bits influence the state vector. A little experimentation with possible subset labelings will show that the free distance can climb to $d_f^2 = 4.585E_s$, producing another 0.6 dB of asymptotic coding gain.

A final remark that is apparent from this design is that only the pattern of constellation points assigned to the trellis is important in establishing performance, and not the actual bit labels attached to constellation points. This labeling enters the picture only when we design encoders. If the constellation points in 8-PSK are labeled with a natural binary progression around the circle from the zero-phase position, it is readily shown that the encoder of Figure 6.6.5 implements the desired encoding operation. Notice that one input bit forms a subset selection by specifying the two least-significant bits of the label, while the uncoded bit picks one of the two antipodal signals in a subset. If the labeling is changed to Gray-coded labeling or some other labeling, the encoder must be changed accordingly.

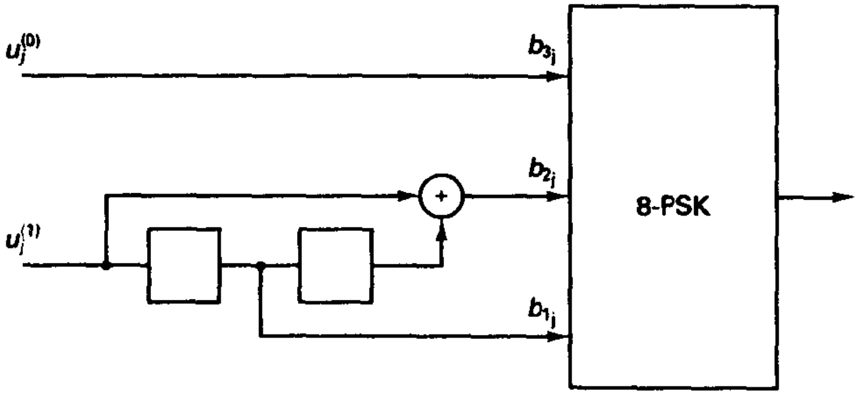


Figure 6.6.5 Feedforward encoder realization of best 4-state code for 8-PSK.

Example 6.24 Trellis-coded 16-QAM with $k = 3$ Bits/Symbol

Figure 6.6.2 illustrated set partitioning for the 16-QAM constellation. The best four-state code for sending $k = 3$ bits per symbol using 16-QAM uses $\tilde{k} = 1$ coded bit to produce a 2-bit subset label for subsets of size four [4]. The remaining two input bits then select a point from the selected four-point subset. Figure 6.6.6 shows the appropriate trellis diagram; there is obvious similarity with that of the 4-state code for 8-PSK, except that the subsets have size 8 here.

Again, we will find that the single-step error events are dominant in the free-distance study. The minimum squared distance between members of these four-point subsets is $16a^2$, assuming that $2a$ is the along-axis spacing in the constellation. The three-step error events

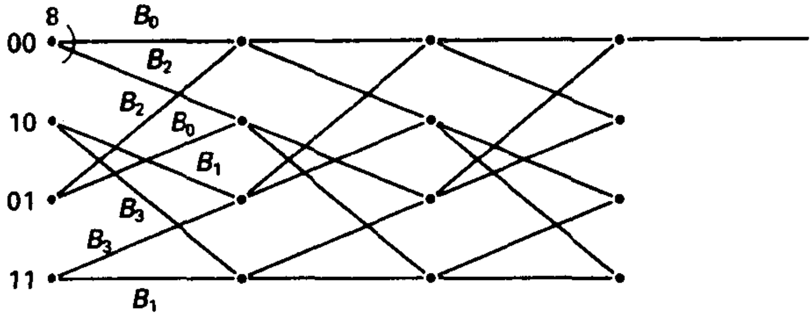


Figure 6.6.6 Trellis labeling for 4-state, 16-QAM code. Note similarity with 4-state, 8-PSK design. Subsets have size 4.

can be shown to have squared distance at least $20a^2$. Recalling that the average energy per symbol in 16-QAM is $10a^2$, as calculated in Chapter 3, and that $E_s = 3E_b$ here, we find $d_f^2 = 4.8E_b$. The leading term in the union bound for error probability is then

$$P_e \approx 2Q\left[\left(\frac{2.4E_b}{N_0}\right)^{1/2}\right], \tag{6.6.5}$$

which is 4.4 dB superior to the performance of uncoded 8-PSK, a system that has the same spectral efficiency. (The multiplier 2 accounts for the fact that each transmitted sequence has two nearest-neighbor one-step error events.)

As mentioned, the trellis structure for this code matches that of the previous example, meaning that the convolutional encoder portion of the system is *identical.* We merely have one additional uncoded bit. To generalize, if we wished to send 7 bits per interval using a 256-QAM constellation, we would use $\tilde{k} = 1$, with 6 uncoded bits. Subsets are size 64 here. All this pertains to 4-state codes.

Referring to the generic system in Figure 6.6.3, the design questions are, for a given k and S, as follows:

1. What constellation should be used?
2. What is the proper $\tilde{k}$ (or what form of trellis branching should be used)?
3. What is the best subset labeling pattern for the trellis or, equivalently, the best encoder?

Regarding the constellation issue, certain engineering constraints may dictate part of the answer. For example, codes with constant energy per transmitted symbol may be important. This might augur well for M-ary PSK modulation. Of more interest is the necessary constellation size. Ungerboeck [4], using information-theoretic arguments, argued that expansion by a factor of 2 is adequate for two-dimensional constellations; that is, if uncoded modulation requires an M-point QAM constellation, the coded system will operate effectively with a $2M$-point QAM constellation. In so doing, the dimensionality per information bit remains unchanged, and, to first order, so does the power spectrum. In Chapter 4, we encountered this basic idea: a look at R_0 curves for two-dimensional constellations will show that 8-point constellations are adequate, in the information-theoretic sense, to support reliable communication with $R = 2$ bits per symbol. Likewise, good 16-point constellations are adequate to fashion good codes with 3 bits/interval. This constellation doubling theme carries over to higher-dimensional cases as well and allows the constellation expansion, *per dimension*, to be smaller than 2, a helpful fact for modem designers [58].

Larger-complexity codes have been found by computer search, combining binary convolutional codes with "mapping by set partitioning" to select modulator points. That is, we can adopt the labeling implied by the set partitioning process, and exemplified in Figure 6.6.1, and then search over the class of binary convolutional encoders with S states, determining the free distance for each such code. As earlier claimed, any feed-forward nonsystematic convolutional code can be realized as a linear, systematic encoder with output feedback, in the sense that the two systems produce the same space of code sequences. Since systematic encoders are inherently noncatastrophic, it is convenient to search the class of encoders having feedback. (This also slightly reduces the search class.)

Systematic encoders with feedback are specified by listing parity check polynomials $h^i(D) = h_0^i + h_1^i D + \cdots + h_\nu^i D^\nu$, for $i = 0, 1, \ldots, k$. For encoders with only $\tilde{k}$ coded bits, we will have that $h^i(D) = 0, \tilde{k} < i \leq k$ [4], which says that the uncoded bit sequences need not obey any parity constraints. Figure 6.6.7 illustrates the generic encoder in this form, in particular showing that only $k - \tilde{k}$ inputs influence the state vector.

To illustrate the specification of encoders in this form, the encoder that is optimal for a set-partitioned 8-PSK constellation and four-state trellis Example 6.23 is specified by

$$h^0(D) = 1 + D^2, \qquad h^1(D) = D, \qquad h^2(D) = 0. \tag{6.6.6}$$

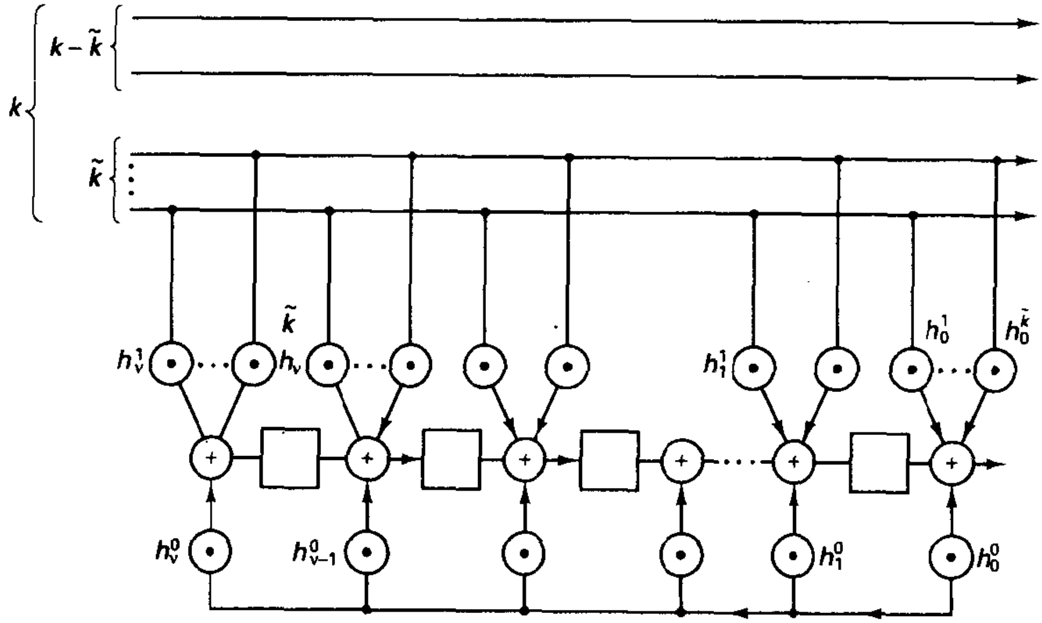


Figure 6.6.7 Generic 2^ν-state encoder in systematic form with feedback.