[6] *J. G. Proakis, *Digital Communications*, 4th Edn. McGraw-Hill, New York, 2001.
[7] V. A. Kotelnikov, "The theory of optimum noise immunity," Ph.D. Thesis, Molotov Energy Institute, Moscow, January 1947; available under the same name from Dover Books, New York, 1968 (R.A. Silverman, translator).
[8] J. M. Wozencraft and I. M. Jacobs, *Principles of Communication Engineering*, Wiley, New York, 1965.
[9] C. E. Shannon, "Communication in the presence of noise," *Proc. IRE*, **37**, 10–21, 1949; reprinted in Claude Elwood Shannon: Collected Papers, N. J. A. Sloane and A. D. Wyner, eds., IEEE Press, New York, 1993.
[10] H. L. van Trees, *Detection, Estimation, and Modulation Theory*, Part I, Wiley, 1968.
[11] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, 2nd Edn. McGraw-Hill, New York, 1984.
[12] W.B. Davenport, Jr. and W.L. Root, *Random Signals and Noise*. McGraw-Hill, New York, 1958.
[13] *M. Schwartz, *Information, Transmission, Modulation and Noise*, 4th Edn. McGraw-Hill, New York, 1990.
[14] J. B. Anderson, T. Aulin and C.-E. Sundberg, *Digital Phase Modulation*. Plenum, New York, 1986.
[15] M. K. Simon, S. M. Hinedi and W. C. Lindsey, *Digital Communication Techniques*. Prentice-Hall, Englewood Cliffs, NJ, 1995.
[16] W. Webb and L. Hanso, *Quadrature Amplitude Modulation*. Pentech Press, London/IEEE Press, Piscataway, NJ, 1994.
[17] F. Gardner, *Phase-Lock Techniques*, 2nd Edn. Wiley, New York, 1979.
[18] T. J. Baker, "Asymptotic behavior of digital FM spectra," *IEEE Trans. Commun.*, **COM-22**, 1585–1594, 1974.
[19] R. E. Ziemer and R. L. Peterson, *Digital Communications and Spread Spectrum Systems*. Macmillan, New York, 1985.

3

Coding and Information Theory

3.1. Introduction

Just as Chapter 2 summarized modulation theory, we now review coding and information theory with an emphasis on what underlies the rest of the book. We will refine somewhat the ideas of data, channels and coding that were introduced in Chapter 1. A data source is a sequence of independent symbols in this chapter, and a channel converts these to another, possibly different, set of symbols. We will see in more concrete form how a channel code is a set of patterned sequences, each corresponding to a data sequence, and how the patterning allows errors to be recognized and corrected. Channel encoding and decoding are the study of schemes for creating these patterned sequences and for deciding the data therein after the channel has taken its toll. Coding involves signal distances and probabilities of error, as well as coding algorithms, their steps and complexity.

In a book about coded modulation it is worth repeating that with a binary channel the only way to impose a coded patterning on binary data is to add extra symbols to the transmission, symbols that we colloquially call parity checks. Section 3.2 introduces codes for this simple situation. Section 3.3 introduces an important way to view codes, the code trellis. The trellis idea also applies to most presently existing coded modulations, codes which are based only partly or not at all on parity checks. Since most of the codes in this book are most easily viewed as trellis codes, Section 3.4 introduces some basic trellis decoding algorithms. Other ideas in Section 3.4 are code concatenation, the combining of small codes into large ones, and iterative decoding. These concepts lie at the heart of many recent developments in coding.

The rest of Chapter 3 introduces some relevant parts of Shannon information theory, which is a subject quite different from coding. Shannon theory thinks of a data source not so much as symbols but as a probability distribution on the symbols; a channel likewise is a conditional distribution of the channel outputs given its inputs. Information is measured by a functional called entropy. The theory proves limit theorems about this measure before and after the channel and about what measure can flow through the channel. It is often difficult to infer

hard conclusions from information theory about concrete encoders and decoders, but the theory's suggestions are provocative and roughly accurate, and this is the theory's appeal.

Information theory has three main divisions, a theory of channel transmission, of information in data sources and a theory of rate distortion, which is about the transmission of information when imperfect reproduction is allowed. Only the first division is of major interest in this book. We will focus expecially on the Gaussian channel models that underlie coded modulation.

3.2. Parity-check Codes

The chapter begins with codes that work by adding parity-check symbols to the data symbols. We will introduce only a few simple codes, but these nonetheless find direct use later in the book. The structure of more advanced parity-check codes is based on the algebraic theory of finite fields, a major topic that we cannot cover here. Fortunately, these details can usually be divorced from the rest of a coded modulation system. When not, we can recommend [2–4]. The section ends with the notion of a soft channel.

3.2.1. Parity-check Basics

We consider binary channel codes that encode a sequence $\boldsymbol{u}^{(K)}$ of K data bits into a codeword $\boldsymbol{x}^{(N)}$ of length N bits.[1] We will assume for now that the K data bits are gathered together and form the first K bits of the word; the last $N-K$ bits are the added parity-check bits. A code such as this, in which the data bits explicitly appear, is called a *systematic* code. The *rate* of this code, in data bits per channel use, is K/N.

When a codeword passes through a *binary symmetric channel* (BSC), the channel inverts some of the bits of $\boldsymbol{x}$ at random to form the length-N received vector $\boldsymbol{y}$. In a standard BSC, these errors occur independently with probability p. We can write $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{e}$, in which the addition is bit-wise mod-2 and $\boldsymbol{e}$ is another length-N sequence called the error sequence. An error in the jth position means that $e[j] = 1$; otherwise, $e[j] = 0$.

In analogy to the maximum likelihood (ML) receiver in Section 2.3, it is possible to derive an ML receiver for the BSC. As always, the receiver should find the maximum over i of the probability $P[\boldsymbol{y} \mid \boldsymbol{x}_i]$, where $\boldsymbol{x}_i$ denotes the ith of a set of words under consideration. Starting from Eq. (2.3-2) and adapting the steps thereafter, we get the analog to Eq. (2.3-4), that

$$P[\boldsymbol{y} \mid \boldsymbol{x}_i] = (1-p)^{N-d_i} p^{d_i}, \quad p < 1/2, \tag{3.2-1}$$

[1] $\boldsymbol{x}^{(N)}$ denotes a row vector of length N; $\boldsymbol{x}'$ denotes a column vector; $\boldsymbol{x}_i$ denotes the ith of several such vectors. $x[j]$ denotes the jth component of a sequence or vector $\boldsymbol{x}$.

in which d_i is the number of bits in $\boldsymbol{y}$ and $\boldsymbol{x}_i$ that differ. This is simply the number of 1s in the sequence $\boldsymbol{y} + \boldsymbol{x}_i$, the *Hamming distance*, denoted $h_{\mathrm{D}}(\boldsymbol{y}, \boldsymbol{x}_i)$. It is equivalent to maximize the log of Eq. (3.2-1) over i, which yields

$$\text{Find } i \text{ that achieves: } \max_i (N - d_i)\log(1-p) + d_i \log p.$$

By eliminating $N\log(1-p)$ and reversing the sign, this becomes

$$\text{Find } i \text{ that achieves: } \min_i d_i \log[(1-p)/p]. \tag{3.2-2}$$

For $p < 1/2$, this boils down to minimizing d_i, the Hamming distance $h_{\mathrm{D}}(\boldsymbol{y}, \boldsymbol{x}_i)$ to $\boldsymbol{y}$.

When the source probabilities for each x are available, it becomes possible to specify an MAP receiver. In analogy to Eq. (2.3-3), the receiver executes

$$\text{Find } i \text{ that achieves: } \max_i (1-p)^{N-d_i} p^{d_i} P[x[i]],$$

which by the steps above becomes

$$\text{Find } i \text{ that achieves: } \min_i d_i \log[(1-p)/p] - \log P[x[i]]. \tag{3.2-3}$$

As $p \to 0$, the choice in Eq. (3.2-3) becomes the ML choice in Eq. (3.2-2).

In any real transmission system, a modulator somehow underlies the BSC, which is just saying that symbols have material existence when they are transmitted. A simple situation would be that antipodal signaling – or for carrier modulation, either BPSK or half of a QPSK – underlies the BSC. In these cases, p in a BSC is $Q(\sqrt{2E_{\mathrm{b}}/N_0})$. It is common to plot error rate for a BSC decoder against the E_{b}/N_0 obtained from p through this equality. On the other hand, little is known sometimes about the underlying channel, and there may not even be a definable channel error probability. In such cases, one can still employ a minimum Hamming distance decoder to correct errors without, of course, defining a decoder error probability. Probabilities of any kind are not actually required in what follows.

The simplest parity-check code is the *repetition* code, that transmits one data bit per codeword by simply repeating the bit N times. Clearly, a simple majority decision will correct up to $\lfloor N/2 \rfloor$ transmission errors. The code rate is $1/N$. The opposite of a repetition code is a *single parity-check* code, which carries $N-1$ data bits in each word, with the Nth a parity bit chosen so that the bits sum mod-2 to zero. This code cannot correct errors, but it can detect the presence of any odd number; in particular, it can detect a single error out of N, which with a short block length and a reasonable channel, can greatly reduce the undetected error rate. The rate is $(N-1)/N$.

Effective codes most of the time need to avoid the extremes of these two codes. A parity-check code does this by defining the set of codewords to be the null space of a matrix, that is, the row vector $\boldsymbol{x}$ is a codeword if

$$\boldsymbol{x}\boldsymbol{H} = \boldsymbol{0}. \tag{3.2-4}$$

Here $\boldsymbol{H}$ is an $N \times N-K$ binary matrix, $\mathbf{0}$ is the length $N-K$ all-zero row vector, and arithmetic is over the field of integers mod-2. In a single parity-check code, $\boldsymbol{H} = (11\ldots1)'$. For a systematic code with the first K bits in $\boldsymbol{x}$ being the data bits, $\boldsymbol{H}$ will have the form

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{P} \\ \boldsymbol{I}_{N-K} \end{bmatrix}, \tag{3.2-5}$$

where $\boldsymbol{I}_{N-K}$ is the identity matrix of size $N-K$ and the $K \times N-K$ matrix $\boldsymbol{P}$ is what we are free to choose. Equation (3.2-4) expresses in matrix form a set of $N-K$ parity-check equations, all of which must be satisfied by $\boldsymbol{x}$ if $\boldsymbol{x}$ is a codeword.

The solutions to Eq. (3.2-4) form an algebraic group, since the bitwise mod-2 sum of any two solutions to Eq. (3.2-4) is itself a solution.[2] For this reason parity-check codes are called *group* codes, or more often, *linear* codes. The properties of the group so formed are the key to the decoder design for the different types of linear codes. The minimum distance of a linear code is in analogy to Section 2.3 the least Hamming distance between any pair of words. For word $\mathbf{0}$, the closest word is the one with the least weight (i.e. the fewest 1s). Call this one $\boldsymbol{w}$ and its weight d_w; since $\boldsymbol{x}+\boldsymbol{w}$ is a word lying this distance from word $\boldsymbol{x}$, it is clear that any word has a neighbor at the same distance and no closer. d_w is, in fact, $d_{\min}$, the minimum distance between words that occurs in the whole set of codewords. Further arguments like this show that all words in a linear code have the identical constellation of neighbors.

From the triangle inequality, a linear code with a minimum distance decoder can correct all weight-t bit error patterns only if $t < d_{\min}/2$. When channel error sequence $\boldsymbol{e}$ occurs, the received $\boldsymbol{y}$ is $\boldsymbol{x}+\boldsymbol{e}$, and Eq. (3.2-4) becomes

$$\boldsymbol{y}\boldsymbol{H} = (\boldsymbol{x}+\boldsymbol{e})\boldsymbol{H} = \mathbf{0} + \boldsymbol{e}\boldsymbol{H}. \tag{3.2-6}$$

$\boldsymbol{e}\boldsymbol{H}$ is called the *syndrome* of $\boldsymbol{x}$. If $\boldsymbol{e}$ consists of a single 1 in the jth place, the syndrome is the jth row of $\boldsymbol{H}$; if $\boldsymbol{e}$ indicates errors in several places, $\boldsymbol{e}\boldsymbol{H}$ is the sum of the respective rows of $\boldsymbol{H}$. If all the rows of $\boldsymbol{H}$ are distinct, it is clear that all single errors in $\boldsymbol{x}$ can be distinguished by their syndromes and corrected. More generally, if every combination of t rows is distinct, then any combination of t errors can be corrected. A decoder that works by finding the syndrome $\boldsymbol{y}\boldsymbol{H}$, mapping to a set of error positions, and correcting these positions, is called a *syndrome* decoder. Such a simple table look-up procedure is practical for short codes. Much research has gone into finding less bulky procedures that are based on properties of the code group.

When $\boldsymbol{y}\boldsymbol{H} = \mathbf{0}$, $\boldsymbol{y}$ is already a codeword, and is usually the one sent. It will not be so if $\boldsymbol{e}$ itself is a codeword. When $\boldsymbol{e}\boldsymbol{H}$ is not $\mathbf{0}$, an ML decoder, in accordance with Eq. (3.2-2), will decide in favor of the codeword closest to $\boldsymbol{y}$. In a syndrome decoder, it is this error pattern to which the syndrome is mapped. The syndrome decoder cannot guarantee correction of more than $\lfloor d_{\min}/2 \rfloor$ errors, because no decoder based on minimum distance can do so, but it remains to be shown that it really corrects up to t. This follows from the fact that $\boldsymbol{x}_{\min}\boldsymbol{H} = \mathbf{0}$ for the minimum weight codeword $\boldsymbol{x}_{\min}$. Consequently, $d_{\min}$ rows of $\boldsymbol{H}$ sum to $\mathbf{0}$, but no fewer rows do, since then the weight of $\boldsymbol{x}_{\min}$ would then be less. Thus, every sum of t or fewer rows must be unique, which implies that every $\boldsymbol{e}$ with t or fewer errors leads to its own syndrome.

[2] One needs to show as well that the codeword $\mathbf{0}$ is the identity element, that each element has an inverse (namely, itself) and that the addition is commutative.

The Hamming codes are a simple but effective class of codes that correct all single errors in the codeword. By definition, the rows of $\boldsymbol{H}$ for a Hamming code consist of all the length $N-K$ words except $\mathbf{0}$. There are $2^{N-K}-1$ of these and so the length of a codeword is $N = 2^{N-K}-1$. Each row of $\boldsymbol{H}$ is the syndrome for one of the N single-bit errors. Hamming codes have rates of the form $R = (2^\ell - 1 - \ell)/(2^\ell - 1)$, for $\ell = 2, 3, \ldots$

Example 3.2-1 *(The Length-7 Hamming Code).* The first non-trivial code is the one with $N-K = 3$, which creates the code and $\boldsymbol{H}$ shown in Fig. 3.1. The 16 words that satisfy $\boldsymbol{x}\boldsymbol{H} = \mathbf{0}$ are listed. The code is a systematic version of this Hamming code, with the first four bits equal to the data bits; $\boldsymbol{H}$ with a different arrangement of the rows will lead to a different version of the same code. The syndrome $(\boldsymbol{x}+\boldsymbol{e})\boldsymbol{H} = 110$ occurs when there is an error in the first place; although 110 also occurs when there are errors in places 2 and 4, the pattern $\boldsymbol{e} = 1000000$ is the most likely one. All Hamming codes have minimum distance 3, which is clear in the figure for length 7.

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

0 0 0 0	0 0 0	0 1 0 1	1 1 0
1 0 0 0	1 1 0	0 0 1 1	0 1 0
0 1 0 0	0 1 1	1 1 1 0	0 1 0
0 0 1 0	1 1 1	1 1 0 1	0 0 0
0 0 0 1	1 0 1	1 0 1 1	1 0 0
1 1 0 0	1 0 1	0 1 1 1	0 0 1
1 0 1 0	0 0 1	1 1 1 1	1 1 1
0 1 1 0	1 0 0		
1 0 0 1	0 1 1		

Figure 3.1 Parity-check matrix $\boldsymbol{H}$ and the codeword set for the systematic (7, 4) Hamming code. The first 4 bits in each word are the data.

A standard notation for block parity-check codes is the form (N, K), which means codeword length N and K data bits. The above example is a $(7, 4)$ code.

An alternate way to define a parity-check code is by its generator matrix, the $K \times N$ matrix that maps the data bit word $\boldsymbol{u}$ to a codeword $\boldsymbol{x}$ according to

$$\boldsymbol{x} = \boldsymbol{u}\boldsymbol{G}. \tag{3.2-7}$$

For a systematic code, $\boldsymbol{G}$ has the form

$$\boldsymbol{G} = [\boldsymbol{I}_K \mid \boldsymbol{P}], \tag{3.2-8}$$

where $\boldsymbol{P}$ is the $K \times (N - K)$ matrix in Eq. (3.2-5). From the form of Eq. (3.2-7), it must be that each row of $\boldsymbol{G}$ is a codeword, and the code must consist of $\boldsymbol{0}$ plus all $2^K - 1$ sums of the rows of $\boldsymbol{G}$. Since $\boldsymbol{x}\boldsymbol{H} = \boldsymbol{0}$, it must be that $\boldsymbol{u}\boldsymbol{G}\boldsymbol{H} = \boldsymbol{0}$ for all $\boldsymbol{u}$; consequently, it must be that $\boldsymbol{G}\boldsymbol{H} = \boldsymbol{0}$, with $\boldsymbol{0}$ here the $K \times (N - K)$ all-zero matrix. We could equally well write this as $(\boldsymbol{G}\boldsymbol{H})^{\text{tr}} = \boldsymbol{H}^{\text{tr}}\boldsymbol{G}^{\text{tr}} = \boldsymbol{0}$, which is an expression of the fact that $\boldsymbol{H}^{\text{tr}}$ can be a generator matrix and $\boldsymbol{G}^{\text{tr}}$ can be the parity-check matrix, for a length-N code with the new rate $(N - K)/N$. This code, which encodes $N - K$ data bits rather than K, is said to be the *dual code* to the one generated by $\boldsymbol{G}$.

3.2.2. BCH and Reed–Solomon Codes

These important codes are members of a large subclass of the linear codes called *cyclic* codes. These are codes whose words are all cyclic shifts of each other. The $(7, 4)$ Hamming code in Fig. 3.1 is one such code. Words number 2, 3, 7, 13, 8, 11 and 5 form a set of right shifts; there is one other seven-member set of shifts, and the remaining words 1111111 and 0000000 are shifts of themselves. It can be shown that for a cyclic code there exists a generator matrix whose rows are cyclic shifts as well. This new matrix may not set up the same correspondence between data words and codewords, but the overall set of codewords will be the same, and the code is therefore taken as equivalent.

The cyclic property, as well as many of the mechanics of cyclic codes, are easier to express with the delay polynomial notation. Consider the word $\boldsymbol{x} = x[N-1], x[N-2], \ldots, x[0]$, with bit number 0 taken as the rightmost one; its polynomial notation is given by

$$x(D) = \sum_{i=0}^{N-1} x[i]D^i, \tag{3.2-9}$$

in which the polynomial variable D can be thought of also as signifying a delay of one symbol position. Yet another way to generate the words in a code if it is cyclic

is by the polynomial multiplication $u(D)g(D)$, in which $u(D)$ represents the data word $\boldsymbol{u}$, $g(D)$ is the first of the rows in the generator matrix, and operations on coefficients are mod-2. Some investigation shows that because the rows of $\boldsymbol{G}$ can be taken as cyclic shifts, the matrix operation $\boldsymbol{x} = \boldsymbol{u}\boldsymbol{G}$ in Eq. (3.2-7) can indeed be replaced by $x(D) = u(D)g(D)$. $g(D)$ in the $(7, 4)$ code, for example, is $D^3 + D^2 + 1$, which is the row vector 0001101; for the data word $\boldsymbol{u} = 0010$, represented by the polynomial D, the polynomial product $u(D)g(D)$ is $D^4 + D^3 + D$, which is the codeword 0011010. This codeword corresponds to data word 0011 in the Fig. 3.1 list, which is not the $\boldsymbol{u}$ we have just taken, but the $u(D)g(D)$ process will eventually generate the same list if continued through all possible $u(D)$. The operation $u(D)g(D)$ has terms beyond the power $N - 1$, and we emulate the cyclic shift through the requirement that words of order greater than $N - 1$ be taken as

$$x(D) = u(D)g(D) \bmod (D^N - 1), \tag{3.2-10}$$

that is, $x(D)$ is the remainder when $u(D)g(D)$ is divided by $D^N - 1$. It can be shown that $g(D)$ generates a cyclic code if and only if $g(D)$ is a factor of $D^N - 1$, and further that $g(D)$ is the unique nonzero codeword polynomial of least degree in the code. A rich coding theory grows out of these facts and leads to many clever decoders [2–4].

The BCH and Reed–Solomon codes are cyclic codes for which $g(D)$ breaks into factors taken from a certain kind of polynomials that stem from the theory of finite groups. These and the final $g(D)$ are listed in algebraic coding theory texts. In BCH codes the polynomials and the $g(D)$ take on binary coefficients, while RS code polynomial coefficients are non-binary. For any $m > 2$, there exists an (N, K) BCH code that corrects t errors, with $N = 2^m - 1$ and $K \geq 2^m - 1 - mt$. BCH codes are quasi-perfect, which means that for a given number 2^{N-K} of syndromes, they correct all patterns of t or fewer errors, plus a set of patterns of $t + 1$, and no others. For the BSC with $p < 1/2$, no code of length N and rate K/N has lower word error probability than such a code. The Hamming $(7, 4)$ code is a BCH code with $m = 3$ and $t = 1$. A double error correcting $(15, 7)$ BCH code (with $m = 4$ and $t = 2$) is generated by $g(D) = D^8 + D^7 + D^6 + D^4 + 1$; its 256 syndromes can be used to correct all 121 patterns of two or fewer errors, as well as 135 three-error patterns.

In RS codes, the coefficients of $x(D)$, $g(D)$ and the data $u(D)$ are taken as non-binary, and in fact are usually from alphabets of size 2^ℓ. In a way parallel to BCH codes, RS codes correct t or fewer 2^ℓ-ary symbol errors. As an example, there exists a $(15, 9)$ RS code based on hexadecimal symbols, that corrects three symbol errors. The generator is

$$g(D) = D^6 + 7D^5 + 9D^4 + 3D^3 + cD^2 + aD + a,$$

where the hexadecimal coefficients are denoted in the usual way $0, 1, 2, \ldots, a, b, \ldots, f$. Arithmetic rules between these symbols need also to be specified

during the code construction. The code encodes nine hexadecimals, or 36 bits, into words of length 15 symbols (60 bits).

3.2.3. Decoding Performance and Coding Gain

Usually, but not always, decoder performance means probability of error. When probabilities cannot be defined, then the number of errors corrected is the performance measure. Several bounds on N, K and d can be constructed from the combinatorics and the algebra that govern parity-check codes. The most important of these is the Gilbert–Varsharmov bound, which is based on the fact that no combination of $d_{\min}-1$ or fewer rows of $\boldsymbol{H}$ may sum to zero. It can be shown [2–5] that this implies

$$\sum_{i=0}^{d_{\min}-2} \binom{N-1}{i} < 2^{N-K}, \tag{3.2-11}$$

which is the precise form of the bound. It gives a limit on $d_{\min}$ in terms of N and K, or alternately, N and the rate R.

An interesting asymptotic form results when N and K become large at a fixed rate K/N. A result of combinatorics states that for $d_{\min} < N/2$,

$$(1/N)\log\left[\sum_{i=0}^{d_{\min}-2}\binom{N-1}{i}\right] \rightarrow h_{\mathrm{B}}(d_{\min}/N) \tag{3.2-12}$$

as $N \rightarrow \infty$, where $h_{\mathrm{B}}(\,)$ is the binary entropy function (for a plot of this function, see Example 3.6-1). Thus, Eq. (3.2-12) becomes in the limit $h_{\mathrm{B}}(d_{\min}/N) < 1 - K/N = 1 - R$. Since $h_{\mathrm{B}}(\,)$ is monotone in the range of interest, we have

$$d_{\min}/N < h_{\mathrm{B}}^{-1}(1-R). \tag{3.2-13}$$

This is a bound on the achievable minimum distance as a fraction of block length.

When the channel is a BSC with a definable crossover p, the p and the capacity satisfy $p = h_{\mathrm{B}}^{-1}(1-C)$, as we will discuss in Section 3.6. Consider coding at a rate R close to C in this channel. On the average there will be about pN channel errors in each length-N codeword, and yet Eq. (3.2-13) states that $d_{\min}$ is less than $Nh_{\mathrm{B}}^{-1}(1-R)$, which is about Np. Thus no parity-check code exists that *always* corrects even half the number of errors that we expect to occur! Fortunately, parity-check codes correct many error patterns at weights above $d_{\min}/2$, even if they do not correct all of them. Both in theory and in practice, their error probability is set by how far beyond $d_{\min}/2$ they continue to correct most errors.

When a full BSC with crossover p can be defined, one can compute an overbound to codeword error probability for a t-error-correcting code as

$$P_{\mathrm{w}} \le \sum_{n=t+1}^{N}\binom{N}{n} p^n (1-p)^{N-n} \tag{3.2-14}$$

This estimate assumes that $t+1$ or more channel errors always lead to a decoder error, which will be true for at least one such error pattern, but may not be true for too many others, particularly if the patterns have just $t+1$ errors. BCH and Hamming codes correct all the t-error patterns, possibly some at $t+1$, and no others; however, the convolutional codes in the next section correct many longer patterns. The only way to refine Eq. (3.2-14) perfectly is to enumerate the correctable patterns.

For a fixed N and t, as $p \rightarrow 0$ the log of the right side of Eq. (3.2-14) tends to $(t+1)\log p$. The asymptotic form of P_{w} is thus[3]

$$p_{\mathrm{w}} \sim p^{t+1}. \tag{3.2-15}$$

When the decoder decides an incorrect word, data bit errors must necessarily occur, but there is no set rule about how the data bit error probability P_{b} compares to P_{w}. Obviously, $P_{\mathrm{b}} \le P_{\mathrm{w}}$. Once again, an enumeration of the cases is needed and one such technique is the transfer function method (see [5,11,19]).

Since the output of a BSC is one of two symbols (namely, the two that can be sent), it is called a *hard decision channel*. A standard way to view the gain from a code over the BSC is to assume that an AWGN channel and a binary antipodal modulator underlie the BSC. Therefore, $p = Q(\sqrt{2E_{\mathrm{s}}/N_0})$, in which E_{s} is the modulator's symbol energy. Since the code carries R data bits/channel use, our usual energy per data bit must be $E_{\mathrm{b}}^{\mathrm{c}} = E_{\mathrm{s}}/R$, where the superscript c is a reminder that the transmission is coded. Substitution for E_{s} then gives $p = Q(\sqrt{2RE_{\mathrm{b}}^{\mathrm{c}}/N_0})$, and the tight approximation to $Q(\,)$ in Eq. (2.3-18) converts this asymptotically as $p \rightarrow 0$ to

$$p \sim (1/2)\exp(-RE_{\mathrm{b}}^{c}/N_0). \tag{3.2-16}$$

If we combine Eqs (3.2-15) and (3.2-16), and take $P_{\mathrm{b}} = P_{\mathrm{w}}$, then as $p \rightarrow 0$ the log of the data bit error probability P_{b} tends to $(t+1)(-RE_{\mathrm{b}}^{\mathrm{c}}/N_0)\log e$.

Without coding, P_{b} is simply $Q(\sqrt{2E_{\mathrm{b}}^{\mathrm{uc}}/N_0})$, whose log tends to $-(E_{\mathrm{b}}^{\mathrm{uc}}/N_0)\log e$, where uc means uncoded. If we equate these two log probabilities, we obtain the asymptotic ratio $E_{\mathrm{b}}^{\mathrm{c}}/E_{\mathrm{b}}^{\mathrm{uc}}$ that is required to maintain the same log probability; in dB it is

$$G_{\mathrm{h}} = 10\log_{10} R(t+1)\,\mathrm{dB}. \tag{3.2-17}$$

[3] See Section 2.3.2 for the technical definition of $\sim$.

This is the asymptotic AWGN *hard decision coding gain*. It shows how much the uncoded system energy must be increased in a good channel in order to maintain the same error probability.

A *soft decision channel* takes in transmission symbols (binary here) but puts out symbols in a larger alphabet, for instance, over the real numbers. A standard way to view coding gain in this case is to once again assume binary antipodal signaling in AWGN, but now with real inputs to the decoder. One unit of Hamming distance between binary codewords translates to a Euclidean square distance $4E_s$; the minimum distance between codewords becomes $d_{\min}4E_s$. When normalized to the data bit energy, the same distance becomes

$$4E_s d_{\min}/(2E_s/R) = 2Rd_{\min}. \tag{3.2-18}$$

The probability of deciding a word as its nearest neighbor is then from Eq. (2.3-22) the 2-signal probability $Q(\sqrt{2Rd_{\min}E_b^c/N_0})$. Without coding, the antipodal signaling error probability is $Q(\sqrt{2E_b^{uc}/N_0})$ as usual. Following the same steps as for the hard channel case, we obtain the ratio

$$G_s = 10\log_{10} Rd_{\min} \approx 10\log_{10} 2R(t+1/2)\,\text{dB}. \tag{3.2-19}$$

This is the asymptotic *soft decision coding* gain for the AWGN channel. Comparing Eq. (3.2-17) with Eq. (3.2-19) shows that *soft decision leads in theory to a 3 dB coding gain.*

In Section 3.4, we will introduce the idea of a soft decision *decoder*, and it is easy to confuse this notion with a soft-output channel. The soft decoder puts out a probability that its data bit decisions are correct, together with the bits. To reiterate, the soft-output channel puts out a higher alphabet variable, which a decoder, itself either hard or soft, can use to make better decisions.

3.3. Trellis Codes

We now turn to codes whose words have a regularly repeating structure called a trellis. The trellis of a code closely relates to the Viterbi decoder for the code, but the notions of paths, states and branches in a code trellis also provide a language to describe other kinds of decoders. We will begin with convolutional codes, a class of parity-check codes that are also trellis codes. However, continuous-phase modulation (CPM) and partial response signaling (PRS) codes are not in any way parity check. It should be mentioned that all practical codes have some sort of trellis structure, and the study of these structures is an interesting subject, but in this book "trellis" codes will mean those with a certain highly regular structure.

3.3.1. Convolutional Codes

Viewed as a parity-check code, a convolutional code consists of words $\boldsymbol{x}$ defined through a code generator matrix by $\boldsymbol{x} = \boldsymbol{u}\boldsymbol{G}$. Here the arithmetic operations are mod-2, $\boldsymbol{u}$ is the data bit word and $\boldsymbol{G}$ is the generator matrix as usual but with the special form:

$$\boldsymbol{G} = \begin{bmatrix} G_0 & G_1 & G_2 & \cdots & G_m & & \\ & G_0 & G_1 & G_2 & \cdots & G_m & \\ & & G_0 & G_1 & G_2 & \cdots & G_m \\ & & & \cdots & & & \\ & & & & \cdots & & \end{bmatrix} \tag{3.3-1}$$

in which the successive rows consist of $G_0G_1\cdots G_m$ shifted right by a fixed step for each row and the pattern is truncated on the right. The elements not shown are zeros. Take a code of rate $R = (\log_2 \beta)/c$, where c and β are integers; in practical codes β is a power of 2 and so it will be convenient to denote $\log_2 \beta$ as b and take the rate as b/c. The building blocks G_j in Eq. (3.3-1) are $b \times c$ submatrices. It will be easier to work first with the case $b = 1$, which yields codes with rates of the form $1/2, 1/3, 1/4, \ldots$. Then each of the building blocks G_j consists of the c bits in the row vector $\{G_j[1], G_j[2], \ldots, G_j[c]\}$, and there are $m+1$ such building blocks. m is called the *memory* of the convolutional code, since from the form of $\boldsymbol{x} = \boldsymbol{u}\boldsymbol{G}$, a span of $m+1$ data bits affects a given code symbol.[4] m sets the code's complexity and strength.

The concepts are much easier to visualize through an example such as the following, which is a standard example of a short systematic code of rate 1/3.

Example 3.3-1 *(Memory 2 Rate 1/3 Convolutional Code).* The submatrices will be $G_0 = 111$, $G_1 = 011$, $G_2 = 001$, each with length $m+1 = 3$. From Eq. (3.3-1), the generator matrix is the top of Fig. 3.2. A length-N codeword is given by $\boldsymbol{x}^{(N)} = \boldsymbol{u}^{(K)}\boldsymbol{G}$. N and K may take any convenient values, so long as K/N equals the rate. For example, with $N = 15$ and $K = 5$, $\boldsymbol{G}$ is truncated to the 5×15 matrix

$$\begin{bmatrix} 111 & 011 & 001 & & & \\ & 111 & 011 & 001 & & \\ & & 111 & 011 & 001 & \\ & & & 111 & 011 & \\ & & & & 111 & \end{bmatrix}. \tag{3.3-2}$$

The data 10011 will produce the codeword 111 011 001 111 100. (The spaces here and in Eq. (3.3-2) are inserted only for legibility.) The underlined bits, it can be seen, are the data bits; in fact, the corresponding columns in Eq. (3.3-2) taken by themselves form an identity matrix, and so the code is evidently systematic.

[4] An older term for memory is *constraint length*, by which is meant the quantity $m+1$.

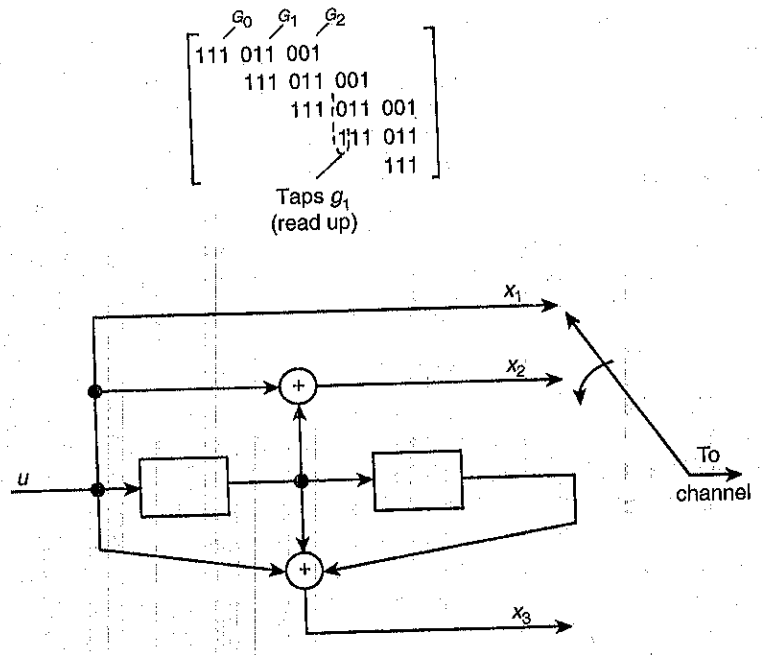

Figure 3.2 Rate 1/3 generator matrix G of width 15, for Example 3.3-1 (*top*); shift register encoder implementation (*bottom*).

The essence of the code generation in Eq. (3.3-2) is that codeword bits are produced in groups of c, groups of three in the example. The bottom of Fig. 3.2 is a circuit that produces the example code bits in a way that is considerably easier to follow than the generator method. The data bits enter a shift register on the left, and the code bits are generated by tap sets and leave on the right. The tap sets, or generators, as they are called, are themselves defined in vector notation as follows:

$$\begin{aligned} \boldsymbol{g}_1 &= \{G_0[1], G_1[1], \ldots, G_m[1]\} \\ \boldsymbol{g}_2 &= \{G_0[2], G_1[2], \ldots, G_m[2]\} \\ &\cdots \\ \boldsymbol{g}_c &= \{G_0[c], G_1[c], \ldots, G_m[c]\}. \end{aligned} \tag{3.3-3}$$

Here, $G_j[n]$ denotes the nth component of a $1 \times c$ submatrix G_j. The second generator, for example, consists of the second bits of each building block. The second tap set will consist of a tap at each '1' in this generator. In Example 3.3-1, the generators are $\boldsymbol{g}_1 = 100$, $\boldsymbol{g}_2 = 110$ and $\boldsymbol{g}_3 = 111$. The top tap set implements $\boldsymbol{g}_1$ and is just a single tap back to the data stream. It produces the first of each group of three codeword bits. Tap sets at the middle and bottom implement $\boldsymbol{g}_2$ and $\boldsymbol{g}_3$.

It will be convenient to denote the first tap set output sequence as $\boldsymbol{x}_1$, the second as $\boldsymbol{x}_2$, etc. up to $\boldsymbol{x}_c$. The c sequences interleave to form the complete codeword.

Yet another way to generate convolutional codewords is by the simple formula

$$x_k[n] = \sum_{j=0}^{m} g_k[j]u[n-j], \qquad k = 1, \ldots, c. \tag{3.3-4}$$

In the notation here, $u[1]$ denotes the rightmost bit in the shift register, the oldest one, just as in the delay notation of Section 3.2.2. Each $\boldsymbol{x}_k$ equals the convolution $\boldsymbol{x}_k = \boldsymbol{g}_k * \boldsymbol{u}$, a convolution of the kth generator with the data bits. In the succeeding chapters, we will exclusively use the convolutional form of Eq. (3.3-4). Generator vectors will be given in left-justified octal notation. For example, the rate 1/2 code with $\boldsymbol{g}_1 = 1101$ and $\boldsymbol{g}_2 = 1111$ has generators 64 and 74, and is called the (64,74) rate 1/2 code. The code in Example 3.3-1 is the (4, 6, 7) rate 1/3 code.

It remains to take the case of rates b/c, when $b > 2$. Now it is easiest to think of the data as arriving in blocks of b, which are de-interleaved into b streams, $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_b$, as for instance in Fig. 3.3. Each stream feeds its own shift register, and taps on these registers feed a group of c summing junctions that produce the c output streams; any register can connect to any junction. Now there are as many as bc length $m+1$ generator sequences, which we can denote as $\boldsymbol{g}_{k,\ell}$, for each $k = 1, \ldots, c$ and each $\ell = 1, \ldots, b$. The convolution form of Eq. (3.3-4) becomes

$$x_k[n] = \sum_{\ell=1}^{b} \sum_{j=0}^{m} g_{k,\ell}[j]u_\ell[n-j], \quad k = 1, \ldots, c \tag{3.3-5}$$

with the $\boldsymbol{x}_k$ interleaved in the usual way to form the complete $\boldsymbol{x}$. The G-matrix building blocks are now obtained from the generators by

$$G_j = \begin{bmatrix} g_{1,1}[j] & g_{2,1}[j] & \cdots & g_{c,1}[j] \\ g_{1,2}[j] & g_{2,2}[j] & \cdots & g_{c,2}[j] \\ & & \cdots & \\ & & \cdots & \\ g_{1,b}[j] & g_{2,b}[j] & \cdots & g_{c,b}[j] \end{bmatrix} \qquad j = 0, \ldots, m. \tag{3.3-6}$$

The notations here are easiest to learn by another standard example.

Example 3.3-2 *(Memory 2 Rate 2/3 Convolutional Code).* Take as generators the six sequences

$$\begin{aligned} \boldsymbol{g}_{1,1} &= 100, & \boldsymbol{g}_{2,1} &= 010, & \boldsymbol{g}_{3,1} &= 110, \\ \boldsymbol{g}_{1,2} &= 011, & \boldsymbol{g}_{2,2} &= 000, & \boldsymbol{g}_{3,2} &= 100. \end{aligned}$$

The shift register circuit that produces the codewords appears at the bottom of Fig. 3.3. Commutators at the input and output de-interleave the input and interleave the output. Two bits enter for each three that leave. The second of each input

$$\begin{bmatrix}
101 & 011 & 000 & & & & & & & & & \\
001 & 100 & 100 & & & & & & & & & \\
 & & 101 & 011 & 000 & & & & & & & \\
 & & 001 & 100 & 100 & & & & & & & \\
 & & & & 101 & 011 & & & & & & \\
 & & & & 001 & 100 & & & & & & \\
 & & & & & & 101 & & & & & \\
 & & & & & & 001 & & & & &
\end{bmatrix}$$

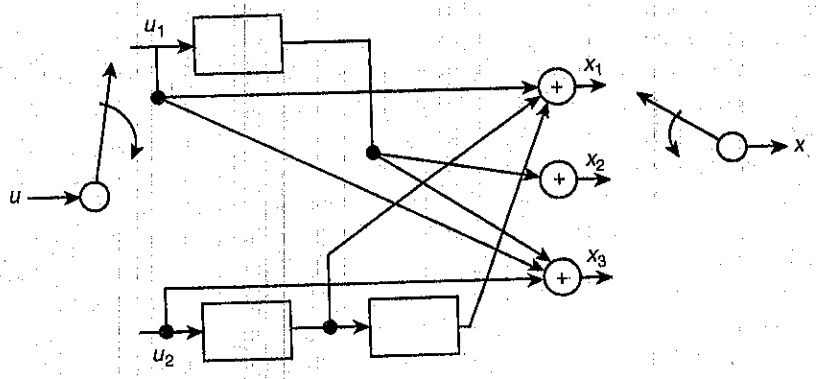

Figure 3.3 Rate 2/3 generator matrix G of width 12, for Example 3.3-2 (*top*); the shift register encoder implementation (*bottom*).

pair has no effect on the second of each output triple; this is signified by $g_{2,2} = 000$. Using Eq. (3.3-6), we can obtain the G-matrix shown at the top of the figure, when the block length is set to 12. Take the data input $\boldsymbol{u} = 10\,01\,11\,00$. De-interleaved, this is $\boldsymbol{u}_1 = 1010$ and $\boldsymbol{u}_2 = 0110$. The output streams are $\boldsymbol{x}_1 = 1000$, $\boldsymbol{x}_2 = 0101$, $\boldsymbol{x}_3 = 1001$; interleaving these gives $\boldsymbol{x} = 101\,010\,000\,011$, which is the result of $\boldsymbol{uG}$. (Here the first bit to enter or leave the encoder circuit is on the left.)

Immense literature exists about the properties of convolutional codes, a review of which can be found in [2–6], and [11] is a full length treatment. Our interest is mostly the concepts of minimum distance and error event, but these are easier to discuss in terms of a code trellis in Section 3.3.2. The references list codes with best minimum distance at each rate and memory m.

A systematic convolutional encoder produces the data bits explicitly as the first b bits of each output group of c. In the convolution formula (3.3-4), it is clear that $\boldsymbol{g}_1$ must be $100\cdots0$ in a rate $1/c$ systematic encoder. A generalization of this applies to Eq. (3.3-5). All of the convolutional encoders so far here have contained feedforward shift registers. It is also possible to generate feedback, or "recursive,"

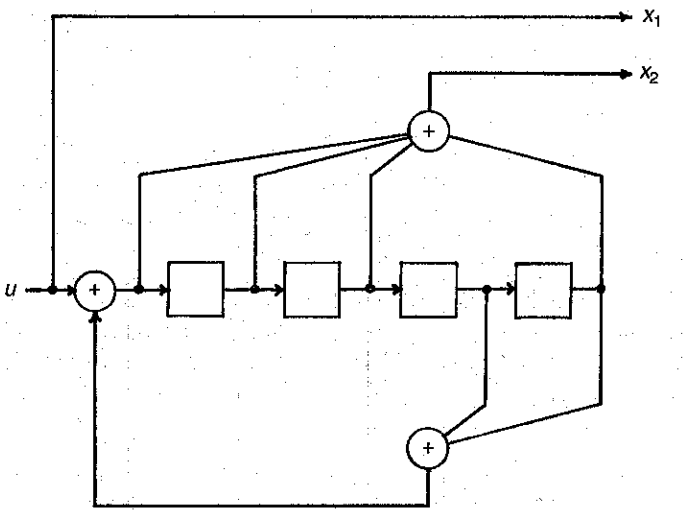

Figure 3.4 Shift register implementation of a rate 1/2 feedback systematic encoder $(1\ g_2/g_1)$, with $g_1 = 46$ and $g_2 = 72$.

convolutional codes by means of a feedback shift register like Fig. 3.4. The figure actually demonstrates a systematic recursive code with rate 1/2: The first bit in each output pair is directly the data bit and the shift register in the figure recursively generates the parity-check symbol.

The feedback register needs a little more explanation, and we will use as an aid the D-transform notation of Section 3.2.2. The register in Fig. 3.4 carries out the polynomial multiplication by $g_2(D)/g_1(D)$, in which the division is the polynomial kind, but the coefficients add and multiply by the mod-2 rule. The highest order term in $g_1(D)$ or $g_2(D)$ corresponds to the rightmost register tap. The division may carry on indefinitely, which is to say that the circuit has infinite unit response. The register has two tap sets, a feedforward one whose taps correspond to the 1s in $\boldsymbol{g}_2$, and a feedback one whose taps are the 1s in $\boldsymbol{g}_1$ ($g_1[0]$ must always be 1). With the corresponding tap sets, the circuit can carry out any $u(D)g_2(D)/g_1(D)$. A general rate $1/c$ feedback convolutional code would require c registers of the kind in Fig. 3.4.

A theory has developed that compares the different types of convolutional codes. We can summarize the important results briefly. For brevity, restrict the rate to $1/c$.

1. For a given rate R and memory m, nonsystematic feedback codes have in theory the largest distance. However, at short and medium m, few, if any, feedback codes are better than the best feedforward code at the same R and m. Thus, nonsystematic feedback codes are seldom used.

2. For any feedforward code, there exists a systematic feedforward encoder that generates the same set of words; however, its memory may be as large as N/c, the number of data symbols. If its memory is limited to that of the original code, the distance will be much worse.
3. For any feedforward code, there exists a systematic feedback code with *the same* m, that has the same codewords and therefore the same distance properties. The words in the two codes may correspond to different data sequences.

Since the feedforward code class includes essentially the best-distance codes, it follows that feedback is an important technique because it allows these best codes to be at the same time systematic. It can be shown that the feedback systematic code that corresponds to the rate $1/c$ feedforward code with generators $g_1(D), \ldots, g_c(D)$ has the generators

$$[1, g_2(D)/g_1(D), g_3(D)/g_1(D), \ldots, g_c(D)/g_1(D)] \qquad (3.3\text{-}7)$$

For example, the rate 1/2 code (46, 72), which has $g_1(D) = 1 + D^3 + D^4$ and $g_2(D) = 1 + D + D^2 + D^4$, has the equivalent systematic feedback code with generators 1 and $g_2(D)/g_1(D)$; the circuit for this is Fig. 3.4.

3.3.2. Code Trellises

Any realizable encoder is a finite-state machine (FSM) and a trellis is a graph of all possible state transversals against time. The trellis idea was suggested by G. D. Forney around 1970. The concept originally grew up as a way to visualize convolutional codes and we will start the discussion of trellises with these. The trellis concept leads directly to a basic decoder, the Viterbi algorithm (VA), which we take up in Section 3.4.

We begin with a FSM description of a convolutional encoder. The state of an encoder such as Fig. 3.2 can be defined at time n by the b-ary data symbols $u[n-1], u[n-2], \ldots, u[n-m]$. The output of the decoder (which is a c-tuple) depends on these symbols plus the present one, $u[n]$; that is, on the last $m+1$ data symbols. The trellis is created from the state transitions by plotting states vertically against time evolving to the right. Figure 3.5 shows the trellis of the machine in Fig. 3.2, assuming that the FSM begins in state 00. Nodes in the trellis represent states in the FSM. The trellis breaks down into *stages*. At each stage, two branches leave each state, one for each present data bit, and after two stages, two branches enter each state. After stage 2, the trellis is fully developed and repeats the same branch pattern indefinitely until the data stops. The branches each contain branch labels, which are the codeword c-tuples, and the data bits in parentheses drive the transition. The states are numbered in a binary coded

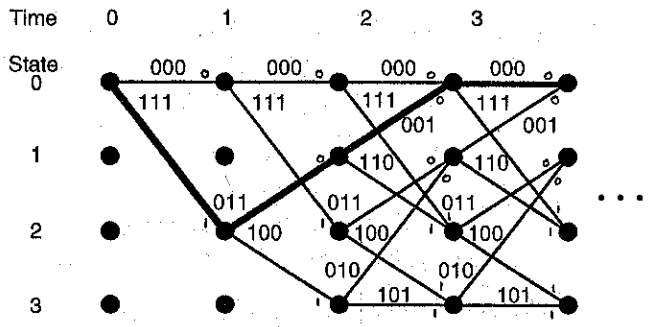

Figure 3.5 The code trellis for the rate 1/3 convolutional code generated in Fig. 3.2. Small figures are the data bits that cause each transition; large figures are branch labels.

decimal representation, with the least significant bit equal to the oldest bit in the shift register, $u[n-m]$.

Any path through the trellis spells out a codeword, and the whole trellis depicts all the possible words. As an example, the data bits $u[0], u[1], \ldots = 100\ldots$ produces the shaded path in Fig. 3.5. (Recall that bit $u[0]$ is the first to enter the encoder shift register.) The path rejoins the all-zero path, caused by $\boldsymbol{u} = 00\ldots0$, after three stages. The set of future paths is identical in front of these two three-branch codeword sections {000, 000, 000} and {111, 011, 001}; the two path sections are said to *merge*, in this case, to merge at state zero after stage 2.

The trellis makes it easy to visualize minimum distance. If convolutional codewords are of length N, which implies N/c trellis stages, the code minimum distance is the least nonzero weight of a path in the trellis. (We use here that the distance structure of the code is the same for all words, and assume that the all-zero codeword is sent.) The general problem of finding the least weight path through a trellis is solved by dynamic programming, which for a trellis is the VA. In finding the least weight path, we will consider only those that leave the all-zero path and later merge to it again; we will take care of the low weight paths at the far right in the trellis later. Some study of the example in Fig. 3.5 shows that among the paths that split and later merge, the least weight path has weight 6, no matter how long the trellis runs. The shaded path is one such path. A minimum distance has occurred rather soon in the trellis and no elongation will give a larger distance. This phenomenon occurs in general and the limit to distance is called the *free distance*, d_f. Not all trellis codes have the same distance structure around all their codewords, and so we give a more general definition as follows:

Definition 3.3-1. The free distance is the least distance that occurs between trellis codewords of unbounded length.

In a convolutional code trellis, the path pair that yields d_f lie separate at least $m+1$ stages, but the separation is often somewhat longer. In general, the

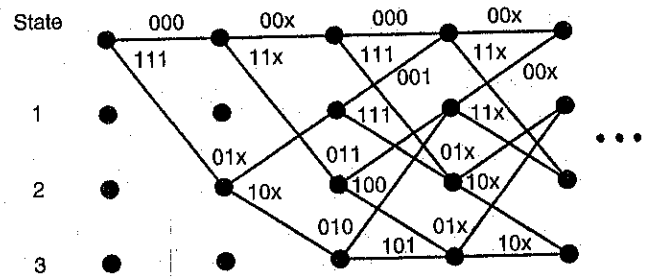

Figure 3.6 A punctured trellis code, using Fig. 3.5 as the mother code and a 2-stage puncture pattern that deletes every sixth bit. Bits denoted "X" are not transmitted.

dynamic program search for d_f must extend far beyond $m + 1$ in order to be sure of finding the minimum merged path pair.

With a convolutional feedforward encoder, the trellis state transitions are easily derived from the shift register contents and the present data symbol: if the present symbol is, for example binary 1, and the present state is 0110, then the next state is 1011, with the old present bit on the left. With recursive codes, the left bit is not necessarily the old present bit. The 2^m-state trellis can nonetheless be filled out by enumerating all the transitions, and the decoding, distance finding, etc., are then the same as always.

The trellis structure is convenient also for visualizing *punctured* codes. These are codes in which parity bits at certain times are simply deleted, or "punctured." Consider again the binary code in Fig. 3.5. If we simply ignore every sixth codeword bit, we have an encoder that puts out five bits for every two data bits, which is a rate 2/5 encoder. A section of this encoder trellis is shown in Fig. 3.6; the free distance is now reduced to five. The original code in Fig. 3.5 is called the mother, or parent, code and the every-sixth bit deletion scheme is called the puncture pattern. The pattern here extends over two trellis stages; other, two-stage patterns could puncture varying combinations of bits number 2, 3, 5 and 6, which would yield rates of 2/5, 2/4, 2/3, depending on how many were deleted. A set of codes like this that all stem from the same parent are called rate-compatible punctured codes. The set gives an easy way to shift the code rate in response to a change in channel quality, since the encoder and decoder for each code is the same except for which bits are to be ignored. Also, puncturing can provide high-rate convolutional codes, which are otherwise difficult to find.

Other Trellis Codes

The trellis structure notion is by no means limited to parity-check codes or to codes based on mod-2 arithmetic. We will close with examples of other kinds of trellis codes that appear in chapters to come.

Rather than mod-2 convolution, it is possible to construct trellis codewords by the real-number convolution

$$x[n] = \sum_{j=0}^{m} g[j]u[n-j]. \tag{3.3-8}$$

This models intersymbol interference, ordinary linear filtering, and the PRS class of coded modulations, which are all discussed in Chapter 6. As a simple example, a trellis based on $g(D) = 1 + aD^2$ is shown[5] in Fig. 3.7; symbols in $\boldsymbol{u}$ take the values ± 1, and the branch labels are real numbers, and the encoder state $\{u[n-1]\,u[n-2]\}$ is one of the four two-tuples $\{-1-1, -1+1, +1-1, +1+1\}$, which are labeled respectively $\{0, 1, 2, 3\}$. The essential difference between this trellis and Fig. 3.5 is the arithmetic rule, and no more.

Another kind of trellis code is shown in Fig. 3.8. This is a CPM trellis, one that shows the codewords generated by Eq. (2.5-27) when the CPM coding is a kind called 3RC with modulation index $h = 1/2$. Now the data symbols in $\boldsymbol{u}$ are again ± 1, and the branch labels are excess phase transitions $\phi(t)$, where the entire signal is $s(t) = \sqrt{2E_s/T}\cos(\omega_0 t + \phi(t))$. A selection of these phase transitions is shown next to the trellis; respective transitions occur between the states that are shown. The states in this CPM trellis are enumerated by a phase that is a multiple of $2\pi/3$ radians and by the two previous data symbols. Many such CPM trellises are employed in Chapter 5.

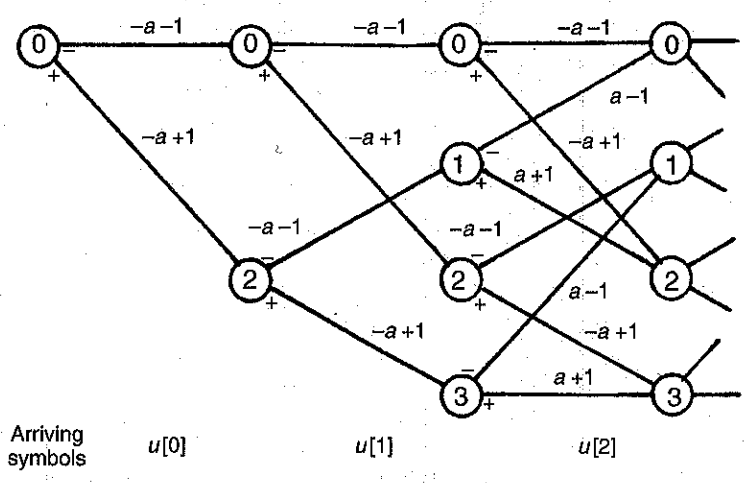

Figure 3.7 PRS code trellis generated by $1 + aD^2$. Branch labels are single real numbers. $\pm$ indicate data symbols that cause transitions. Symbols before $u[0]$ are -1.

[5] In Chapter 6, the delay notation will be exchanged for the z-transform notation $1 + az^{-2}$.

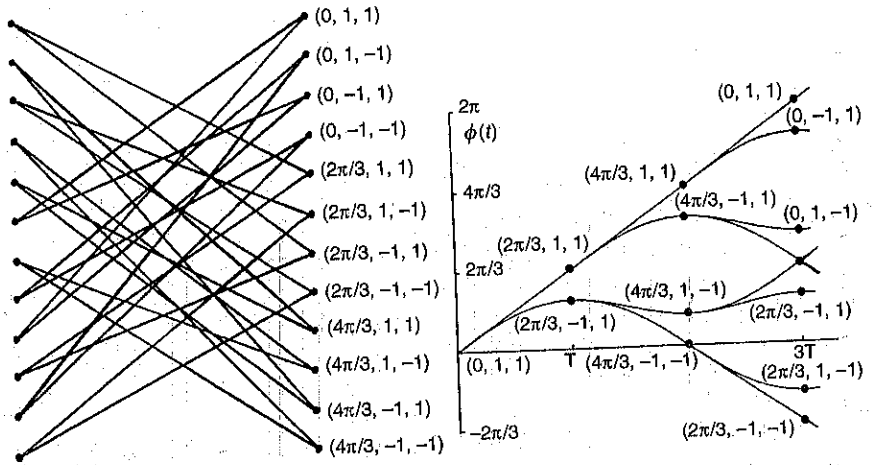

Figure 3.8 CPM trellis for the binary 3RC scheme with index $h = 2/3$ (memory 3); at right is a partial library of branch labels, which are T-second phase transitions. State descriptor consists of a phase and two data symbols. (Adapted from [8].)

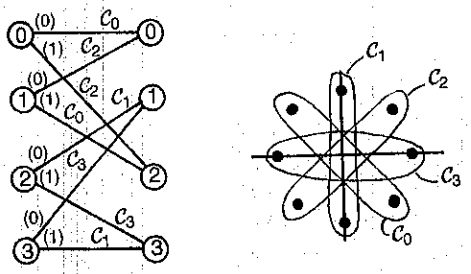

Figure 3.9 One stage of a TCM code trellis based on subsets of an 8PSK constellation. Branch labels are the four 2-point subsets C_0, C_1, C_2, C_3. Small figures are data bits.

Yet another trellis is the TCM trellis of Fig. 3.9. This one is binary and similar in appearance to Fig. 3.5, except that the branch labels are *sets*, in this case, subsets of points in an 8PSK constellation shown at the side. There are four two-point sets C_0, C_1, C_2, C_3, which are subsets of 8PSK as shown. In this code, one data bit drives a rate 1/2 encoder with generators $(g_1(D), g_2(D))$ equal to $(1 + D + D^2, 1 + D^2)$; this generates bit pairs $(x_1[n], x_2[n])$, which take values 00, 10, 11 and 01. This pair selects the subset C_0, C_1, C_2, C_3, respectively. The code carries a second data bit by selecting which of the two points in the subset is sent. The overall data rate is thus two data bits per 8PSK symbol interval. This kind of coding based on sets is introduced in Chapter 4.

3.4. Decoding

The maximum likelihood principle is an obvious design for a decoder: find the most likely transmitted message given the observed channel output. We developed an ML decoder for Gaussian noise in Chapter 2 and for the binary channel in Section 3.2. In both cases, as it turns out, the decoder seeks the closest codeword to the channel sequence, in a Euclidean distance sense or in a Hamming sense. We first present an efficient ML receiver for a trellis code, which is the VA.

There are, however, other design principles than the ML one. We might wish to use a non-ML receiver that has virtually ML error performance at, say, low error probabilities. Another design principle is bounded distance detection. Here the decoder is designed to correct all error patterns with a certain weight, or in the Euclidean distance case, all noises of a certain size. This principle is particularly useful when no definable channel error probabilities exist.

The section will focus on decoders for codes with a regular trellis structure. Most of this book is devoted to such codes. A large body of decoders exists for block parity-check codes, aside from the basic syndrome decoder in Section 3.2. For these we must defer to the references, especially [2–5].

3.4.1. Trellis Decoders and the Viterbi Algorithm

The aim of a trellis decoder is to find the closest path in the trellis to the received sequence. If it succeeds, it will have found the maximum likelihood codeword. One option is to search the entire trellis; this is the VA, to which we return momentarily. Another option is to search only a small region, where the nearest codeword is likely to be. An older term for these schemes is sequential decoding; a more modern one is *reduced-search* decoding. With some reduced-search decoders, it cannot be guaranteed that the ML path is found, although with most the error performance at moderate to high signal-to-noise ratio (SNR) is essentially that of an ML decoder.

There are many reduced-search schemes, but in their gross behavior they fall into two categories, *breadth-first* decoders, that search forward through the trellis in only one direction, and *backtracking* decoders, that can backtrack. Examples of the latter are the Fano and stack algorithms, and since these do not much figure in the rest of the book, we will not discuss them further. The best known breadth-first decoder, aside from the VA, is the M-algorithm.[6] This scheme works forward through the trellis, retaining just M paths; at each stage it makes the $2^b M$ extensions of these and keeps only the M paths with the least cumulative distance. Breadth-first decoders are inherently less efficient than backtracking ones, but

[6] A less common name for this scheme is list algorithm. However, the algorithm does not maintain a "list" in the computer science sense.

the complexity and variability of the latter have discouraged their use. Complete treatments of all these schemes appear in [5,11].

Whatever algorithm a decoder may follow, two basic measures of its complexity are the number of trellis paths it needs to view and store, and the length of these paths. The VA stores one path to each trellis state, or 2^m, altogether; the M-algorithm stores M. While it is true that the decoder must go through a number of steps to extend, view and sort each path, it is known that in theory these steps are in linear proportion to the number of paths. It is usually possible (see [5]) to relate the path number to the performance achieved, either a size of noise corrected or a probability of error. A reduced-search decoder needs to store much fewer paths than a VA, particularly in the case of the partial response codes in Chapter 6.

The length of the trellis paths stored by a decoder is its decoder decision depth, or alternately, its observation window, denoted N_{win}. A trellis decoder in the modern view moves its way down a trellis of indefinite depth, with new parts of the trellis entering the window at the right at stage n, and a final bit decision exiting on the left at stage $n - N_{\text{win}}$.[7] Hopefully, the remaining trellis paths in the decoder storage all share a single ancestor branch on the left. If not, one is chosen as the decoder output and all paths that stem from another branch are dropped from storage. The decoding delay is N_{win}. We will illustrate these concepts shortly with the VA.

The decision depth of a decoder should ideally be long enough so that the decoder error performance is the same as that of a decoder with unlimited depth. It is easiest to estimate a proper decision depth under the bounded distance design principle. The resulting depth is in fact a parameter of the code, not the decoder, and is called the code decision depth L_{dec}. Suppose it is desired to correct up to $d/2$ errors. Then when all paths are extended L_{dec} forward from some node, none should lie closer to the correct path than d; otherwise, the minimum distance as seen by the decoder will be less than d. Dynamic program algorithms exist that find the worst-case trellis path and starting state, and thus evaluate the $L_{\text{dec}}(d)$ to achieve a given d. Such decision depths are tabulated for many trellis codes (see [7] for convolutional codes and Section 5.2 for CPM codes). For good rate b/c convolutional codes and $d \leq d_{\text{f}}$, a simple law holds as d grows: The needed L_{dec} at d satisfies

$$L_{\text{dec}}(d) \approx \frac{d}{ch_{\text{B}}^{-1}(1-R)} \text{ stages.} \qquad (3.4\text{-}1)$$

Here $h_{\text{B}}^{-1}()$ is the inverse of the binary entropy function. At $R = 1/2$, the law is $L_{\text{dec}}(d) \sim 4.5d$; at $R = 1/3$, it is $L_{\text{dec}}(d) \sim 1.9d$. It is important to note that when d is the full free distance, decision depth in Eq. (3.4-1) greatly exceeds encoder memory, and this is generally true for all coded modulation.

[7] A decoder with such a moving window is sometimes called a sliding block decoder. Originally, trellis decoders were considered to run to the end of the trellis and then decide the entire trellis path.

When the design principle is error probability, law (3.4-1) still roughly holds, but the probabilities of different events are such that in practice the decision depth can be a half or so of Eq. (3.4-1).

The Viterbi Algorithm

The VA searches the entire trellis for the least-distant path, in the most efficient manner possible. It finds wide use despite its exhaustive character for a number of reasons: it repeats a similar step at each node, its moves are synchronous and unidirectional, it adopts easily to soft channel outputs; finally, small trellises, where the exhaustiveness does not matter, often have attractive coding gains. The central idea in the algorithm, the optimality principle, was published by Viterbi in 1967 [9]. The full algorithm is an application of dynamic programming, an earlier procedure that finds the shortest route in a graph.[8]

It is easiest to describe the VA with respect to a fixed code, and we will use Fig. 3.5. The stage by stage progress of the algorithm is in Fig. 3.10. The encoder begins from state 0. In Fig. 3.10(a), the first bit triple received is 001 and the first two branches out of state 0 are drawn. The Hamming distances from these stage-0 branches (000 and 111 in Fig. 3.5) to the channel output are 1 and 2; the cumulative distances along the two paths in are shown in parentheses, and are just these same values. In Fig. 3.10(b), the channel output 110 arrives and two path extensions

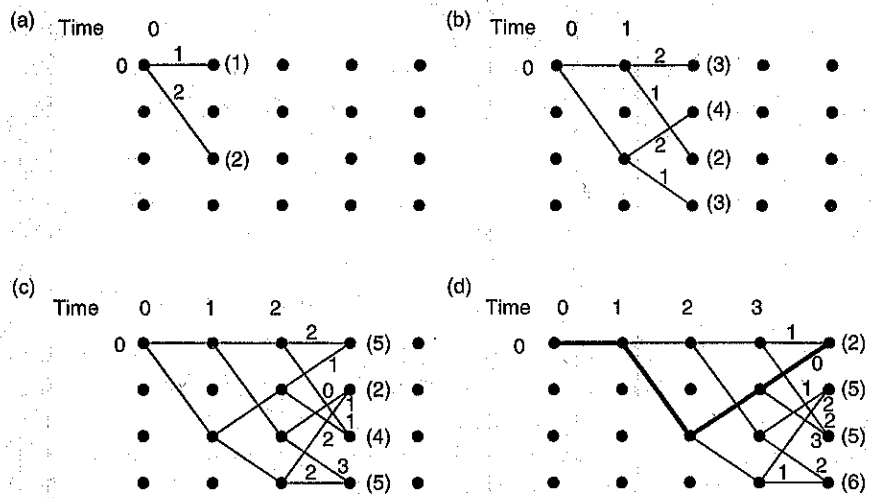

Figure 3.10 Viterbi decoding when the trellis is that of Fig. 3.5 and (a) 001, (b) 110, (c) 011, (d) 001 is received. Numbers on branches are distance increments; those in parentheses are distance accumulations.

[8] The dynamic programming observation was made by J. Omura around 1969.

from stage 1 have been made. The new branches create four paths in total, whose distance increments are shown on the new branches and whose cumulative distance again appears in parentheses. After the four stage-2 node extensions are made in Fig. 3.10(c), two paths merge at each node.

Now comes the critical idea in the VA. From the two paths entering each node at time 3, the one of those with the least total distance, called the *survivor*, is kept and the other path is deleted all the way back to the earlier node; if both have the same distance a survivor is selected at random. After all the path deletions, the distances of the best paths into each time-3 node are (5), (2), (4) and (5), and the surviving trellis looks like the first three stages of Fig. 3.10(d). The stage-4 path extension is shown in Fig. 3.10(d), along with the set of survivor distances (2), (5), (5) and (6). It appears that after four trellis stages, the least-distant path (heavier) starts from state 0 and runs to state 0 at time 4, and furthermore, it has total distance 2. This surviving path has path map $u[0]u[1]u[2]u[3] = 0100$, and corresponds to the partial codeword 000 111 011 001. There are two other words in the full trellis that enter and leave the same nodes, namely those with path maps 0000 and 1000. Their distances are both 6. One can trace the survivor paths to the other nodes at time 4 and find that no path has cumulative distance less than 5.

The VA procedure in Fig. 3.10 can be extended indefinitely. The reason that it finds the least-distant path to each state at each level is the so-called *optimality principle* of dynamic programming: when paths through a graph merge at a node, only a least-cost alternative need be retained, and a least-cost path through the entire graph exists that does not contain the dropped path sections.

In the decoding example here, all survivors after time 2 stem from the same initial branch, so that this branch, corresponding to data bit 0, could be released as output. In a software implementation the path maps into each node at time n and the cumulative distance of each node are stored in two arrays. In the extension to time $n + 1$, two new arrays are created as the path extensions are made, and survivors overwrite deletions. After the creation of stage $n + 1$, the new arrays contain the survivors and become the old arrays for the next stage. In a hardware implementation, an array of similar processor units corresponds to the state nodes at time n; these write to each other through a connection butterfly that mimics the trellis stage branches.

We have shown the VA search as beginning from a start state at time 0. Very often, a transmission has a defined end state as well, in which case the trellis collapses down to this single state. If so, the trellis codeword is said to be *terminated*. We can take the example of binary feedforward convolutional codes of rate $1/c$. In general, m extra stages are needed to terminate a codeword[9] and since this means adding m terminating 0s to the message, the overall rate is degraded. With length-K data frames, a rate b/c code degrades to $bK/c(K + m)$.

[9] In theory, L_{dcc} terminating bits must be used, but a length equal to the shift register is the usual choice.

Another technique of termination is tailbiting, where the start state is defined to be the state at the end of the data symbols. Yet another strategy is to ignore the termination problem altogether. In this case the entire best path in the decoder storage is released when the front of the search reaches the last encoder depth. The path map bits near the front of the map will be less reliable in this case. Viewed as a block code, a trellis code without termination has a poor minimum distance, but the distance only affects the last few data symbols.

Although the discussion here has been carried out with a convolutional code example, the ideas of decision depth, paths, trellis searching, and termination apply to any code with a trellis, and in particular to those in Chapters 4–6.

Error Events in Trellis Decoding

We define an *error event* to begin when the decoder output trellis path splits from the correct trellis path and to end when it merges again. Such events are the basic mechanism of error in trellis coding, and other occurrences, such as data bit errors, derive from them. A bit error necessarily occurs when an event begins but may not occur when it ends.

Figure 3.11 portrays some different types of error events. When the decoder chooses a path at or near the minimum distance from the correct one, a "short" event occurs, whose length is close to the memory m of the code. These are the typical events in a good channel. In convolutional and PRS decoding, short events imply data symbol errors in the first few branches after the split, but none thereafter. The data symbol error rate tends in this case to be 1–2 times the event error rate. In CPM and TCM, bit errors occur at the beginning and end of a short event. In poorer channels, "long" error events occur, and typically these imply a 50% data error rate during the event. The overall BER is thus a multiple of the

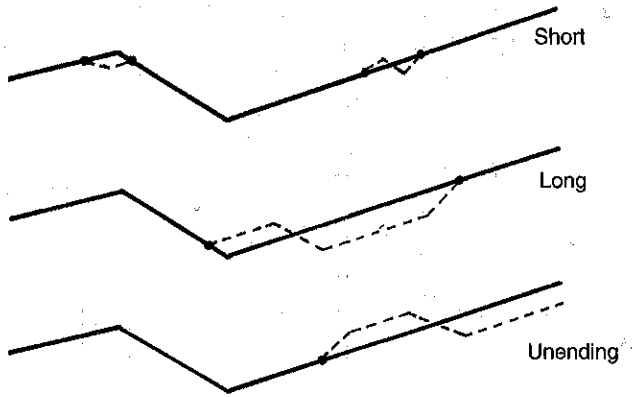

Figure 3.11 Types of error events: two short events (*top*); a long event (*middle*); an event that fails to terminate (*bottom*). Solid path is sent, dashed path is decoder output.

event rate that depends on the balance of short and long events. In reduced-search decoders with certain codes, indefinitely long events can happen. These occur because the decoder path storage has completely lost the correct path. In a properly designed decoder, these would occur only when a VA decoder would have failed as well, at least briefly.

We now turn to the calculation of probabilities for these decoder error events. As in the previous discussions, we will assume that the channel is AWGN or the BSC and that the decoder seeks a minimum distance path: that is to say, the decoder is ML or in the case of a reduced search, it at least attempts to be. Over the BSC, the signals are binary symbols and the distance can be Hamming; with AWGN, the signals are real functions of time and the distance is Euclidean. Since this is a coded modulation book, we will focus on the more general AWGN case.

Suppose that decoding is correct up to time n and that an alternative trellis path representing signal $s(t)$ exists and splits from the correct path $s_0(t)$ at this time. $s(t)$ lies distance d_i away from $s_0(t)$. Signal space theory in Section 2.3.2 then shows that in AWGN the probability is $Q(\sqrt{d_i^2 E_b/N_0})$ that the received $r(t)$ lies closer to $s(t)$ than to $s_0(t)$. Let P_{ev} be the probability that some event occurs at time n. If there are several alternatives, P_{ev} at this juncture is overbounded by $\sum A_i Q(\sqrt{d_i^2 E_b/N_0})$, where A_i is the number of alternatives at distance d_i. The sum is dominated as usual by the Q term for the closest alternative, and in the full unbounded trellis, its distance must be the free distance d_f. If there are many alternatives we can argue as in Section 2.3.2 that P_{ev} has the form $P_{ev} \sim A_0 Q(\sqrt{d_f^2 E_b/N_0})$, asymptotically in E_b/N_0.

The sort of argument here gives good estimates of P_{ev} for CPM and partial response codes at all useful E_b/N_0, if one notes carefully what alternatives cannot occur at nodes along the correct path. On the average over a long transmission it can be said that a log–log plot of P_{ev} vs E_b/N_0, a so-called "water fall curve", like those in Fig. 2.17, will tend at high E_b/N_0 toward the curve $AQ(\sqrt{d_f^2 E_b/N_0})$, with A the number of neighbors at d_f. The bit error rate (BER) plot will lie somewhat above, but will eventually converge to the same curve.

With convolutional codes, it is less easy to get tight estimates to P_{ev}. An approach called the transfer function bound has been devised which is partially successful. By the Mason gain technique, a transfer function

$$T(\delta) = \sum A_i \delta^i \tag{3.4-2}$$

is obtained from the code generator state diagram, where A_i is the number of trellis paths of weight i. It can be shown that over a BSC with crossover p, P_{ev} has the estimate

$$P_{ev} \sim T(\delta)\big|_{\delta=\sqrt{4p(1-p)}} \tag{3.4-3}$$

An analogous bound can be derived for BERs. Details may be found in [2]–[6].

3.4.2. Iterative Decoding and the BCJR Algorithm

The central theme in the preceding section was the decoding of a trellis path. The VA/dynamic program search of the code trellis yielded the maximum likelihood path. As for the data bits, these were taken to be the bits that drove the ML path. In some applications, it is necessary to know not the bits, but the *probability* of the bits. This section introduces some of these situations and the parallel to the VA called the BCJR algorithm, which produces these probabilities. Outputs such as these are an example of a *soft output*, an output that is not simply a data transmission symbol, but may even be a real number such as a probability. We have already seen soft-output channels; here we encounter soft-output decoders.

As an example of a transmission system that profits from soft-output decoding, we introduce concatenated coding systems. Two basic alternatives are shown in Fig. 3.12. The top system is *serial concatenation*, in which an outer encoder feeds an inner decoder, whose outputs pass through the physical channel. An inner decoder detects the inner encoding. This decoder could put out a hard estimate y_{out} of x_{out}, the inner encoder's input. Some thought, however, shows that the outer decoder might profit from some soft information about the symbols of y_{out}, such as the probability of each symbol. A way to visualize the situation is to imagine the inner encoder/channel/inner decoder as an "inner channel," the medium in fact seen by the outer encoder and decoder. The outer decoder will surely profit from outer channel soft outputs.

It is also conceivable that the inner decoder can profit from the deliberations of the outer decoder. The inner decoder output y_{out} may, for example,

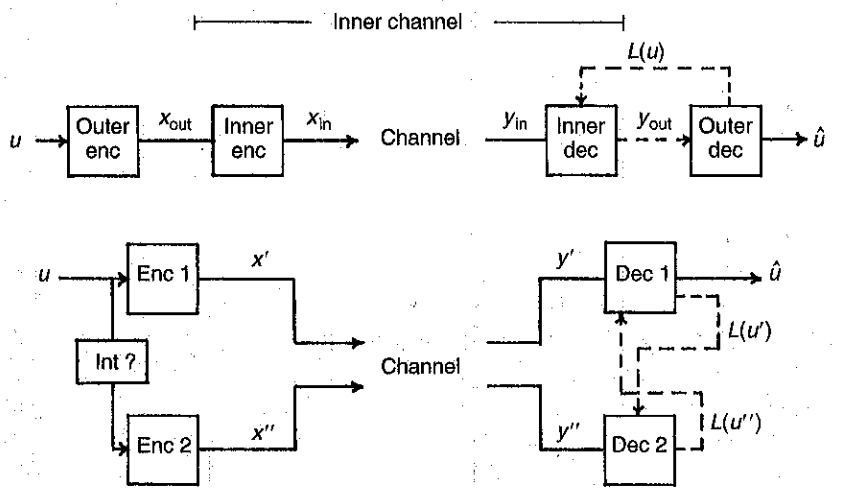

Figure 3.12 Comparison of basic serial (*top*) and parallel (*bottom*) code concatenation. Some possible soft information flows are shown.

somehow not be a legal outer codeword x_{out}. Information of this sort is shown as the feedback $L(u)$ in Fig. 3.12.

A second kind of concatenation is *parallel concatenation*. Here the data symbols are somehow encoded twice and sent at different times through the physical channel or through two separate channels. The two channel sequences are decoded, but it is clear that the two decoders can profit from an interchange, because both are guessing at the same data symbols. These flows are shown as $L(u')$ and $L(u'')$.

A common way to organize the feedback in these two systems is through *iterative decoding*: one decoder processes its inputs, then feeds information to the other, which does likewise, and so on back to the first, until both estimates stabilize. There are two important subtleties in this method. First, the decoding of a large code structure is broken down into small steps with small codes. Investigation has shown that large-code performance is obtained this way with relatively small decoding effort. Second, for the iterating to be effective, soft information must interchange.

The BCJR Algorithm

This scheme computes the data symbol probability at each trellis depth, or equivalently, the probability that each trellis branch was sent. The scheme accepts channel outputs as inputs but it can also accept a set of *a priori* probabilities about the symbols that originally drove the encoder, if these are available. By taking as output the most likely data symbol, we can obtain a hard symbol output from the BCJR algorithm, but if we take a sequence of most likely trellis branches as output, they will not necessarily form a valid trellis path. The algorithm was developed in the early 1970s by researchers who studied early iterative decoders, and it first appeared formally in 1974 in Bahl *et al.* [10], from which comes the algorithm's name. Since it puts out symbol by symbol probabilities, the algorithm is called a symbol by symbol MAP decoder. Provided that the start and stop states are known, the BCJR computes the true *a posteriori* probabilities of each symbol taken in isolation.[10]

The BCJR consists of two recursions on the trellis structure, instead of the one in the VA. These are pictured in Fig. 3.13. They calculate two working vectors called $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. (The quantity $\boldsymbol{\lambda}$ will be defined later.) The components $\alpha[0], \alpha[1], \ldots, \alpha[\mu]$ and $\beta[0], \beta[1], \ldots, \beta[\mu]$ are special probabilities that apply to states $0, \ldots, \mu$ at some stage in a $\mu + 1$ state trellis. The row vector $\boldsymbol{\alpha}$ is defined by

$$\alpha_k[j] \triangleq P[\text{Observe } y(1:k),\ \text{Encoder in state } j \text{ at time } k]. \tag{3.4-4}$$

[10] Otherwise, BCJR gives a slightly biased calculation (see [11,22]). It is also possible to design a sequence MAP algorithm, whose output is the *a posteriori* most likely codeword and its probability.

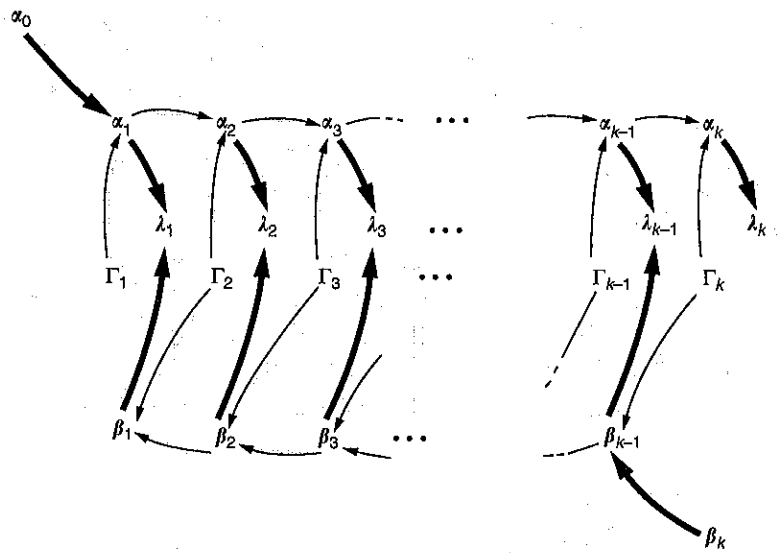

Figure 3.13 A picture of the BCJR algorithm. $\boldsymbol{\Gamma}_1, \ldots, \boldsymbol{\Gamma}_K$ are first generated from the observed outputs $y[1], \ldots, y[K]$. Forward α recursion runs across the top; backwards β recursion across the bottom. The scheme is initialized by $\boldsymbol{\alpha}_0$ and $\boldsymbol{\beta}_k$.

Here $\boldsymbol{\alpha}_k$ is the α-vector at time k and the special notation $y(1:k)$ means the sequence of stage by stage channel observations $y[1], \ldots, y[k]$. Each $y[k]$ is the entire, possibly composite, observation for one stage; in the case of a rate $1/c$ convolutional code, this would comprise c channel outputs. The column vector at stage k is defined by

$$\beta_k[i] \triangleq P[\text{Observe } y(k+1:K) \mid \text{Encoder in state } i \text{ at time } k]. \tag{3.4-5}$$

We also need a matrix $\boldsymbol{\Gamma}$ at stage k, whose i, j element is

$$\boldsymbol{\Gamma}_k[i, j] \triangleq P[\text{Observe } y(k), \text{Encoder in state } j \text{ at time } k \mid \text{Encoder in state } i \text{ at } k-1]. \tag{3.4-6}$$

Note that $\alpha_k[j]$ is the probability of the channel observations at times $1, \ldots, k$ and at the same time state j at k; $\beta_k[j]$ is the probability of the observations at $k+1, \ldots, K$ given state j at k; $\boldsymbol{\Gamma}_k$ is a modification of the usual state transition matrix to include the kth observation. Technically, the algorithm seeks at every step the probabilities of states *given* the entire observations $\boldsymbol{y}$, but since it is true for any event $\mathcal{E}$ that $P[\mathcal{E} \mid \boldsymbol{y}] = P[\mathcal{E}, \boldsymbol{y}]/P[\boldsymbol{y}]$, and since $P[\boldsymbol{y}]$ is fixed throughout, the algorithm needs only to find the probabilities of the states *and* $\boldsymbol{y}$. In some applications one needs to divide by $P[\boldsymbol{y}]$ before using the BCJR result.

Now we can define the two recursions. We assume that the encoding starts and ends in state 0. The *forward recursion* of the BCJR algorithm is

$$\boldsymbol{\alpha}_k = \boldsymbol{\alpha}_{k-1}\boldsymbol{\Gamma}_k, \quad k = 1, \ldots, K \tag{3.4-7}$$

with $\boldsymbol{\alpha}_0$ set to $(1, 0, \ldots, 0)$. The *backward recursion* is

$$\boldsymbol{\beta}_k = \boldsymbol{\Gamma}_{k+1}\boldsymbol{\beta}_{k+1}, \quad k = K-1, \ldots, 1 \tag{3.4-8}$$

with $\boldsymbol{\beta}_K$ set to $(1, 0, \ldots, 0)'$. The algorithm can account for different initial and final encoder conditions by a different $\boldsymbol{\alpha}_0$ and $\boldsymbol{\beta}_K$. The object of the BCJR is to find the probability of the transition from state i to state j ending at time k, while observing the entire channel output $\boldsymbol{y}$. We abbreviate the last as

$$\sigma_k[i \to j] = P\left[\text{Observe } \boldsymbol{y},\ \text{Encoder in state } j \text{ at } k,\ \text{Encoder at state } i \text{ at } k-1\right]. \tag{3.4-9}$$

It will be easiest to organize the calculation of $\sigma_k(i \to j)$ as a theorem and we will justify there also the recursions (3.4-7) and (3.4-8).

THEOREM 3.4-1. *Suppose codeword symbols are generated via a Markov chain and sent over a memoryless channel. Then recursions (3.4-7) and (3.4-8) hold. Furthermore,*

$$\sigma_k[i \to j] = \alpha_{k-1}[i]\Gamma_k[i, j]\beta_k[j]. \tag{3.4-10}$$

Proof. We first show Eq. (3.4-10). For brevity, replace the statement "Encoder in state j at time k" with the event $\{S_k = j\}$. The proof works by breaking the event (3.4-9) into five simultaneous events as follows:

$$P[S_{k-1} = i, S_k = j, \boldsymbol{y}] = P[S_{k-1} = i, S_k = j,\ y(1:k-1),\ y(k),\ y(k+1:K)].$$

It may be helpful to denote these five events in the abstract as, respectively, $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}$. For any five events it is true that $P[\mathcal{ABCDE}] = P[\mathcal{AC}]P[\mathcal{BD} \mid \mathcal{AC}]P[\mathcal{E} \mid \mathcal{ABCD}]$. For our specific events, we may drop event $\mathcal{C}$ in the conditioning $\mathcal{AC}$ in the probability $P[\mathcal{BD} \mid \mathcal{AC}]$. The reason is fundamental and rather subtle: with a Markov chain, future outcomes, conditioned on the present outcome, are independent of the past. The same applies when another sequence of independent variables, the channel noise, are added to the Markov outcomes. In this probability, the future event $\mathcal{BD} = \{S_k = j, y(k)\}$, when conditioned on $\mathcal{A} = \{S_{k-1} = i\}$, is independent of the past event $\mathcal{C} = \{y(1:k-1)\}$. By the same logic, the probability $P[\mathcal{E} \mid \mathcal{ABCD}]$ is equal to $P[\mathcal{E} \mid \mathcal{B}]$. What remains is

$$P[\mathcal{AC}]P[\mathcal{BD} \mid \mathcal{A}]P[\mathcal{E} \mid \mathcal{B}]$$
$$= P[S_{k-1} = i, y(1:k-1)]P[S_k = j, y(k) \mid S_{k-1} = i]\, P[y(k+1:K) \mid S_k = j],$$

which is precisely Eq. (3.4-10).

To demonstrate the recursion (3.4-7), we start by writing

$$\alpha_k[j] = P[S_k = j, y(1:k)] = \sum_i P[S_{k-1} = i, S_k = j,\ y(1:k)].$$

We can break the event $\{y(1:k)\}$ into $\{y(1:k-1),\ y(k)\}$ and write the right-hand sum as

$$\sum_i P[S_{k-1} = i, S_k = j,\ y(1:k-1),\ y(k)].$$

As before we can denote these four events respectively as $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ and note that in general $P[\mathcal{ABCD}] = P[\mathcal{AC}]P[\mathcal{BD} \mid \mathcal{AC}]$. By the Markov argument, $\mathcal{C}$ may be dropped in the second probability. What remains is

$$\sum_i P[S_{k-1} = i,\ y(1:k-1)]\, P[S_k = j,\ y(k) \mid S_{k-1} = i],$$

which is $\sum_i \alpha_{k-1}[i]\Gamma_k[i, j]$. This sum is just the jth component of the row vector $\boldsymbol{\alpha}_{k-1}\boldsymbol{\Gamma}_k$, from which Eq. (3.4-7) follows. A similar reasoning shows Eq. (3.4-8). □

Since the basic BCJR finds just the trellis transition (i.e. branch) probabilities, some further calculation may be required in a particular application. The probability that $\boldsymbol{y}$ occurs and the encoder was in state j at time k (i.e. of a *node*) is, using Eq. (3.4-7),

$$\sum_i \sigma_k[i \to j] = \sum_i \alpha_{k-1}[i]\Gamma_k[i, j]\beta_k[j] = \alpha_k[j]\beta_k[j], \tag{3.4-11}$$

that is, the product of the jth components of vectors $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$. The componentwise product of $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}'_k$ is given the special name $\boldsymbol{\lambda}_k$; this vector is a list of all the node probabilities. In order to find the probability that a given data symbol $u(k)$ was sent, we would sum the probabilities of all transitions that imply this symbol:

$$P[u(k) = \ell] = \sum_{i,j \text{ in } \mathcal{L}} \sum \sigma_k[i \to j]. \tag{3.4-12}$$

Here $\mathcal{L}$ is the set of trellis transitions that correspond to data value ℓ.

Finally, the BCJR can be easily extended to the case where *a priori* probabilities are available for the data symbols. In an iterative decoder, for example, these could be outputs from a previous BCJR algorithm. A close look at the foregoing will show that only the transition probabilities $\Gamma_k[i, j]$ are affected, and so the construction of this series of matrices needs to be modified. This kind of BCJR algorithm is variously called the Apri-BCJR, the Apri-MAP or the soft-in soft-out BCJR algorithm.

There is admittedly more calculation and storage in the BCJR algorithm than in the VA and this is its chief drawback. At each trellis depth, a matrix $\mathbf{\Gamma}_k$ needs to be constructed and stored; then the forward and backward recursions must be run. Fortunately, some applications use short codes and then there is no reason not to simply run the algorithm. The development of simpler near-optimal substitutes for BCJR is an active research area.

3.5. The Shannon Theory of Channels

As remarked at the beginning of the chapter, only one part of information theory is of interest to us in this book, namely the probabilistic theory of channels. This theory, like the rest of Shannon theory, is about limit theorems for random variables. It needs to be distinguished carefully from the coding theory in Sections 3.2–3.4, which concerns itself with codeword symbols, noise, distance and error correction. Shannon theory makes many provocative *suggestions* about encoding and decoding, but it is a free-standing theory based on its own premises. A full treatment of information theory is available in the classic text of Gallager [12]; a newer text with a strongly information-theoretic point of view is Cover and Thomas [13].

In information theory, an *information source* X is a random variable defined by a probability distribution $P[x]$. The source can consist of a sequence of variables $X[1], \ldots, X[K]$, which we can take as the one object $X^{(K)}$ with distribution $P[x^{(K)}]$. The $X[k]$ may also be correlated, although in this book we will assume the source outcomes are independent so that $P[X^{(K)}] = \prod_k P[x[k]]$; then the distribution of just the kth variable defines the measure $P[\ ]$ for the entire sequence.

The measure of information in a source X is its *entropy*, defined for discrete variables as[11]

$$H(X) = -\sum_i P[x_i] \log_2 P[x_i] \text{ (bits/outcome)}, \tag{3.5-1}$$

where the subscript i now enumerates the different outcomes of x. If X is a continuous variable, an integration replaces the sum sign in Eq. (3.5-1). If the variable is $X^{(K)}$ in the IID sequence $X[1], \ldots, X[K]$, then it is easy to show that $H(X^{(K)}) = KH(X)$, where X is one variable in the sequence. For discrete X with I outcomes, $H(X) \leq \log_2 I$, with equality if and only if the outcomes all have probability $1/I$. A variable with outcomes in $\{-3, -1, +1, +3\}$, for example, has entropy 2 bits/outcome or less. It needs to be stressed that "bits" here is a real-number measure of information, just as meters are a measure of length; this is not to be confused with the earlier meaning of bits as symbols.

[11] Here and throughout, capital X denotes a random variable and lower case x denotes an actual outcome.

When there are two information sources X and Y, they may be correlated. For example, when X passes through a distorting medium to become Y, we hope that X and Y are quite closely correlated. For a particular outcome y, the *conditional entropy* of X is defined to be

$$H(X \mid y) \triangleq -\sum_i P[x_i \mid y] \log_2 P[x_i \mid y] \text{ (bits/outcome)}, \tag{3.5-2}$$

in which $P[x_i \mid y]$ is the probability of outcome x_i given y. The expectation of $H(X \mid y)$ over y,

$$H(X \mid Y) = \sum_i H(X \mid y_j) P[y_j] \text{ (bits/outcome)}, \tag{3.5-3}$$

is called the *equivocation* between X and Y. If X and Y are independent, the equivocation $H(X \mid Y)$ is simply $H(X)$.

A transmission or storage medium in Shannon theory is called a channel. The transmission of a source outcome comprises one use of the channel. When successive uses of a channel are independent, the channel is said to be memoryless. As with an information source, Shannon theory defines a channel in terms of probabilities. This time, the definition is by means of the conditional probability distribution $P[y \mid x]$, the probability the channel output is y given that its input is x. As with any conditional distribution, $\sum_j P[y_j \mid x] = 1$, for each input x. A memoryless channel for which both X and Y are discrete is called a discrete memoryless channel (DMC). The channel output is another information source, Y, whose defining distribution is $P[y_j] = \sum_i P[y_j \mid x_i] P[x_i]$. If the channel and the input source are both memoryless, then so is source Y. When Y is a continuous variable, the channel is input-discrete only, and both the foregoing summations become integrals.

In this book we make use of only a few classic channels. The simplest is the BSC, already introduced in Section 3.2.1, for which inputs and outputs are taken from $\{0, 1\}$ and for which $P[1 \mid 1] = P[0 \mid 0] = 1 - p$ and $P[1 \mid 0] = P[0 \mid 1] = p$. A convenient way to diagram the BSC probabilities is shown in Fig. 3.14. In other channels, symbols are either received perfectly or erased, that is, there is no hint of the input at the output. In information theory this is the binary erasure channel (BEC). Its model is shown in Fig. 3.14; with probability p the inputs are converted to a special erasure output, denoted e.[12] A combination of the BEC and the BSC is the binary symmetric erasure channel (BSEC), in Fig. 3.14. As the figure shows, the binary input can be received as either 1 or 0 or e, with the last signifying in effect that the reception was unreliable. The BSEC is a one-step-softened BSC: the more outcomes that Y can take, the "softer" is the channel output.

[12] It is a powerful advantage to know that any 1 or 0 received is necessarily correct. Decoders that fill in erasures are called erasure decoders. In general, a decoder can correct twice as many erasures as it can errors.