18.8.5 Early calculations based on Shannon's theorem set the maximum rate of information transfer lower than that which is now achieved. The resolution of this paradox is the improvement of SNR and methods to exploit more of the bandwidth. Calculate the channel capacity of a telephone line that passes from 200 Hz to 3400 Hz and has a signal-to-noise ratio of about 20–25 dB. Calculate the capacity for a digital telephone line that passes from 200 Hz to 3800 Hz and encodes using logarithmic PCM (12-13 bits).

18.8.6 The 'maximum reach' of a DSL modem is defined to be the distance over which it can function when the only source of interference is thermal white noise. The attenuation of a twisted pair of telephone wires for frequencies over 250 KHz can be approximated by

$$A(f) = e^{-s\left(\kappa_1\sqrt{f}+\kappa_2 f\right)L}$$

where L is the cable length in Km. For 24-gauge wire $\kappa_1 = 2.36 \cdot 10^{-3}, \kappa_2 = -0.34 \cdot 10^{-8}$ and for thinner 26-gauge wire $\kappa_1 = 2.98 \cdot 10^{-3}, \kappa_2 = -1.06 \cdot 10^{-8}$. Assume that the transmitter can transmit 13 dBm between 250 KHz and 5 MHz and that the thermal noise power is -140 dBm per Hz. Write a program to determine the optimal transmitter power distribution and the capacity for lengths of 1, 2, 3, 4, and 5 Km.

18.9 Error Correcting Codes

In order to approach the error-free information rate guaranteed by Shannon, modem signals and demodulators have become extremely sophisticated; but we have to face up to the fact that no matter how optimally designed the demodulator, it will still sometimes err. A short burst of noise caused by a passing car, a tone leaking through from another channel, changes in channel frequency characteristics due to rain or wind on a cable, interference from radio transmitters, all of these can cause the demodulator to produce a bit stream that is not identical to that intended. Errors in the reconstructed bit stream can be catastrophic, generating annoying clicks in music, causing transferred programs to malfunction, producing unrecoverable compressed files, and firing missile banks when not intended. In order to reduce the probability of such events, an *error correcting code* (ECC) may be used.

Using the terminology of Section 18.7, an ECC is a method of channel encoding designed to increase reliability. Error correcting codes are independent of the signal processing aspects of the bit transfer (line coding); they are purely mathematical mechanisms that detect whether bits have become

corrupted and how to recover the intended information. How can bit errors be detected? Were we to send 00011011 and 01011010 was received instead, how could this possibly be discovered? The strategy is that after optimizing the source coding to use the minimal number of bits possible, the channel coding *adds* new bits in order to be able to detect errors. A parity bit is a simple case of this; to seven data bits we can add an eighth bit that ensures that the number of ones is even. Any single-bit error will be *detected* because there will be an odd number of ones, but we will not know which bit is in error. A simplistic error *correction* scheme could send each bit three times in succession (e.g., send 000000000111111000111111 rather than directly sending the message 00011011). Were any single bit to be incorrectly received (e.g. 000010000111111000111111), we could immediately detect this and correct it. The same is true for most combinations of two bit errors, but if the two errors happen to be the same bit triplet, we would be able to detect the error but not to correctly correct it.

The error detection and correction method we just suggested is able to correct single-bit errors, but requires tripling the information rate. It turns out that we can do much better than that. There is a well-developed, mathematically sophisticated theory of ECCs that we will not be able to fully cover. This and the next two sections are devoted to presentation of concepts of this theory that we will need.

All ECCs work by allowing only certain bit combinations, known as *codewords*. The parity code only permits codewords with an even number of ones; thus only half the possible bitvectors are codewords. The bit tripling ECC works because only two of the eight possible bit triplets are allowed; thus of the 2^{3k} bitvectors of length $3k$, only one out of every eight are codewords.

The second essential concept is that of distance between bitvectors. The most commonly used distance measure is the *Hamming distance* $d(b_1, b_2)$. (the same Hamming as the window). The Hamming distance is defined as the number of positions in which two bitvectors disagree (e.g., $d(0011, 0010) = 1$). For bitvectors of length N, $0 \leq d(b_1, b_2) \leq N$ and $d(b_1, b_2) = 0$ if and only if $b_1 = b_2$.

If we choose codewords such that the minimum Hamming distance d_{min} between any two is M, then the code will be able to detect up to $M-1$ errors. Only if M errors occur will the error go unnoticed. Similarly, a code with minimum Hamming distance M will be able to correct less than $\frac{1}{2}M$ errors. Only if there are enough errors to move the received bitvector closer to another codeword (i.e., half the minimum Hamming distance) will choosing the closest codeword lead to an incorrect result.

How do we protect a message using an error correcting code? One way is to break the bits of information into blocks of length k. We then change this k-dimensional bitvector into a codeword in n-dimensional space, where $n > k$. Such a code is called an n/k rate *block code* (e.g., parity is a 8/7 block code while the bit tripling code is a 3/1 block code). The codewords are sent over the channel and decoded back into the original k bits at the receiver. The processing is similar to performing FFTs on nonoverlapping blocks. Sometimes we need to operate in real-time and can't afford to wait for a block to fill up before processing. In such cases we use a *convolutional code* that is reminiscent of a set of n FIR filters. Each clock period k new bits are shifted into a FIFO buffer that contains previously seen bits, the k oldest bits are shifted out, and then n bit-convolution computations produce n output bits to be sent over the channel. The buffer is called a *shift register* since the elements into and from which bits are shifted are single-bit registers.

We can now formulate the ECC design task. First, we decide whether a block or convolutional code is to be used. Next, the number of errors that must be detected and the number that must be corrected are specified. Finally, we find a code with minimal rate increase factor n/k that detects and corrects as required.

At first glance finding such codes seems easy. Consider a block code with given k. From the requirements we can derive the minimal Hamming distance between codewords, and we need only find a set of 2^k codewords with that minimal distance. We start with some guess for n and if we can't find 2^k codewords (e.g., by random search) that obey the constraint we increase n and try again. For large block lengths k the search may take a long time, but it need be performed only once. We can now randomly map all the possible k-dimensional bitvectors onto the codewords and can start encoding. Since the encoding process is a simple table lookup it is very fast. The problem is with decoding such a code. Once we have received an n-dimensional bitvector we need to compute the Hamming distances to each of the 2^k codewords and then pick the closest. This is a tremendous amount of work even for small k and completely out of the question for larger block lengths.

Accordingly we will tighten up our definition of the ECC design problem. Our task is to find a code with minimal n/k that can be *efficiently decoded.* Randomly chosen codes will always require brute force comparisons. In order to reduce the computational complexity of the decoding we have to add structure to the code. This is done using algebra.

EXERCISES

18.9.1 Consider the bit-tripling code. Assume the channel is such that the probability of an error is p (and thus the probability of a bit being correctly detected is $1-p$). Show that the average probability of error of the original bit stream is $P_{err} = 3p^2(1-p) + p^3$. Obviously, $P_{err} = p$ for $p = 0$ and $p = 1$. What is P_{err} for $p = \frac{1}{2}$? Graph P_{err} as a function of p. For $\frac{1}{2} < p < 1$ we see that $P_{err} > p$, that is, our *error correction* method increases the probability of error. Explain.

18.9.2 The bit-tripling code can correct all single-bit errors, and most two-bit errors. Starting with eight information bits, what percentage of the two-bit errors can be corrected? What percentage of three-bit errors can be detected?

18.9.3 A common error detection method is the *checksum*. A checksum-byte is generated by adding up all the bytes of the message modulo 256. This sum is then appended to the message and checked upon reception. How many incorrectly received bytes can a 'checkbyte' detect? How can this be improved?

18.10 Block Codes

About a year after Shannon's publication of the importance of channel codes, Hamming actually came up with an efficiently decodable ECC. To demonstrate the principle, let's divide the information to be encoded into four-bit blocks $d_3d_2d_1d_0$. Hamming suggested adding three additional bits in order to form a 7/4 code. A code like this that contains the original k information bits unchanged and simply adds $n-k$ *checkbits* is called a *systematic code*. In the communications profession it is canonical to send the data first, from least to most significant bits and the checkbits afterward, thus the seven-dimensional codewords to be sent over the channel are the vectors $(a_0a_1a_2a_3a_4a_5a_6) = (d_0d_1d_2d_3c_0c_1c_2)$. The checkbits are computed as linear combinations of the information bits

$$\begin{aligned} c_0 &= d_0 + d_1 + d_3 \\ c_1 &= d_1 + d_2 + d_3 \\ c_2 &= d_0 + d_2 + d_3 \end{aligned} \tag{18.21}$$

where the addition is performed modulo 2 (i.e., using xor). If information bit d_0 is received incorrectly then checkbits c_0 and c_1 will not check out. Similarly, an incorrectly received d_1 causes problems with c_0 and c_2, a flipped

d_2 means c_1 and c_2 will not sum correctly, and finally a mistaken d_3 causes all three checkbits to come out wrong. What if a checkbit is incorrectly received? Then, and only then, a single c_i will not check out. If no checkbits are incorrect the bitvector has been correctly received (unless a few errors happened at once).

The Hamming code is a *linear code*; it doesn't matter if you sum (xor) two messages and then encode them or encode the two messages and then sum them. It is thus not surprising that the relationship between the k-dimensional information bitvector $\underline{d}$ and the n-dimensional codeword $\underline{a}$ can be expressed in a more compact fashion using matrices, $\underline{a} = \underline{\underline{G}}\,\underline{d}$

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

where all the operations are to be understood modulo 2. The n-by-k matrix $\underline{\underline{G}}$ is called the *generator matrix* since it generates the codeword from the information bits. All linear ECCs can be generated using a generator matrix; all systematic codes have the k-by-k identity matrix as the top k rows of $\underline{\underline{G}}$.

The Hamming 7/4 code can correct all single-bit errors, and it is optimal since there are no 6/4 codes with this characteristic. Although it does make the job of locating the bit in error simpler than checking all codewords, Hamming found a trick that makes the job easier still. He suggested sending the bits in a different order, $d_3d_2d_1c_1d_0c_2c_0$ and calling them $h_7h_6h_5h_4h_3h_2h_1$. Now h_1, h_2 and h_4 are both received and computed, and the correction process is reduced to simply adding the indices of the incorrect checkbits. For example, if h_1 and h_2 don't check out then $h_{1+2} = h_3$ should be corrected.

Hamming's code avoids searching all the codewords by adding algebraic structure to the code. To see this more clearly let's look at all the codewords (in the original order)

$$\begin{array}{llll} 0000000 & 1000101 & 0010110 & 1010011 \\ 0001011 & 1001110 & 0011101 & 1011000 \\ 0100111 & 1100010 & 0110001 & 1110100 \\ 0101100 & 1101001 & 0111010 & 1111111 \end{array} \tag{18.22}$$

and note the following facts. Zero is a codeword and the sum (modulo 2) of every two codewords is a codeword. Since every bitvector is its own additive

inverse under modulo 2 arithmetic we needn't state that additive inverses exist for every codeword. Thus the codewords form a four-dimensional subspace of the seven-dimensional space of all bitvectors. Also note that the circular rotation of a codeword is a codeword as well, such a code being called a cyclic code.

The *weight* of a bitvector is defined as the number of ones it contains. Only the zero vector can have weight 0. For the Hamming 7/4 code the minimal weight of a nonzero codeword is 3 (there are seven such codewords); then there are seven codewords of weight 4, and one codeword of weight 7. If the zero codeword is received with a single-bit error the resulting bitvector has weight 1, while two errors create bitvectors of weight 2. One can systematically place all possible bitvectors into a square array based on the code and weight. The first row contains the codewords, starting with the zero codeword at the left. The first column of the second row contains a bitvector of weight 1, (a bitvector that could have been received instead of the zero codeword were a single-bit error to have taken place). The rest of the row is generated by adding this bitvector to the codeword at the top of the column. The next row is generated the same way, starting with a different bitvector of weight 1. After all weight 1 vectors have been exhausted we continue with vectors of weight 2. For the 7/4 Hamming code this array has eight rows of 16 columns each:

0000000	1000101	0100110	...	1011001	0111010	1111111
1000000	0000101	1100110	...	0011001	1111010	0111111
0100000	1100101	0000110	...	1111001	0011010	1011111
0010000	1010101	0110110	...	1001001	0101010	1101111
0001000	1001101	0101110	...	1010001	0110010	1110111
0000100	1000001	0100010	...	1011101	0111110	1111011
0000010	1000111	0100100	...	1011011	0111000	1111101
0000001	1000100	0100111	...	1011000	0111011	1111110

In error correcting code terminology the rows are called *cosets*, and the leftmost element of each row the *coset leader*. Each coset consists of all the bitvectors that could arise from a particular error (coset leader). You can think of this array as a sort of addition table; an arbitrary element $\underline{v}$ is obtained by adding (modulo 2) the codeword at the top of its column $\underline{a}$ to the coset leader at the left of its row $\underline{e}$ (i.e., $\underline{v} = \underline{a} + \underline{e}$).

The brute force method of decoding is now easy to formulate. When a particular bitvector $\underline{v}$ is received, one searches for it in the array. If it is in the first row, then we conclude that there were no errors. If it is not, then the codeword at the top of its column is the most probable codeword and

the coset leader is the error. This decoding strategy is too computationally complex to actually carry out for large codes, so we add a mechanism for algebraically locating the coset leader. Once the coset leader has been found, subtracting it (which for binary arithmetic is the same as adding) from the received bitvector recovers the most probable original codeword.

In order to efficiently find the coset leader we need to introduce two more algebraic concepts, the *parity check matrix* and the *syndrome*. The codewords form a k-dimensional subspace of n space; from standard linear algebra we know that there must be an $(n-k)$-dimensional subspace of vectors all of which are orthogonal to all the codewords. Therefore there is an $(n-k)$-by-n matrix $\underline{\underline{H}}$ called the *parity check matrix*, such that $\underline{\underline{H}}\,\underline{a} = 0$ for every codeword $\underline{a}$. It's actually easy to find $\underline{\underline{H}}$ from the generator matrix $\underline{\underline{G}}$ since we require $\underline{\underline{H}}\,\underline{\underline{G}}\,\underline{d} = 0$ for all possible information vectors $\underline{d}$, which means the $(n-k)$-by-k matrix $\underline{\underline{H}}\,\underline{\underline{G}}$ must be all zeros. Hence the parity check matrix for the 7/4 Hamming code is

$$\underline{\underline{H}} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

as can be easily verified. The parity check matrix of a systematic n/k code has the $(n-k)$-by-$(n-k)$ identity matrix as its rightmost $n-k$ columns, and the rest is the transpose of the nonidentity part of the generator matrix.

What happens when the parity check matrix operates on an arbitrary n-dimensional bitvector $\underline{v} = \underline{a} + \underline{e}$? By definition the codeword does not contribute, hence $\underline{\underline{H}}\,\underline{v} = \underline{\underline{H}}\,\underline{e}$ the right-hand side being a $(n-k)$-dimensional vector called the *syndrome*. The syndrome is thus zero for every codeword, and is a unique indicator of the coset leader. Subtracting (adding) the coset from the bitvector gives the codeword. So our efficient method of decoding a linear code is simply to multiply the incoming bitvector by the parity check matrix to obtain the syndrome, and then adding the coset leader with that syndrome to the incoming bitvector to get the codeword.

By mapping the encoding and decoding of linear ECCs onto operations of linear algebra we have significantly reduced their computational load. But there is an even more sophisticated algebraic approach to ECCs, one that not only helps in encoding and decoding, but in finding, analyzing, and describing codes as well. This time rather than looking at bitstreams as vectors we prefer to think of them as polynomials! If that seems rather abstract, just remember that the general digital signal can be expanded as a sum over shifted unit impulses, which is the same as a polynomial in the

time delay operator z^{-1}. The idea here is the same, only we call the dummy variable x rather than z^{-1}, and the powers of x act as place keepers. The k bits of a message $(d_0 d_1 d_2 d_3 \ldots d_{k-1})$ are represented by the polynomial $d(x) = d_0 + d_1 x + d_2 x^2 + d_3 x^3 + \ldots + d_{k-1} x^{k-1}$; were we to take $x = 2$ this would simply be the bits understood as a binary number.

The polynomial representation has several advantages. Bit-by-bit addition (xor) of two messages naturally corresponds to the addition (modulo 2) of the two polynomials (*not* to the addition of the two binary numbers). Shifting the components of a bitvector to the left by r bits corresponds to multiplying the polynomial by x^r. Hence a systematic n/k code that encodes a k-bit message d into an n-bit codeword a by shifting it $(n-k)$ bits to the left and adding $(n-k)$ checkbits c, can be thought of as a transforming of a $(k-1)$-degree polynomial $d(x)$ into a code polynomial $a(x)$ of degree $(n-1)$ by multiplying it by the appropriate power of x and then adding the degree $(n-k-1)$ polynomial $c(x)$.

$$a(x) = x^{n-k} d(x) + c(x) \tag{18.23}$$

Of course multiplication and division are defined over the polynomials, and these will turn out to be useful operations—operations not defined in the vector representation. In particular, we can define a code as the set of all the polynomials that are multiples of a particular generator polynomial $g(x)$. The encoding operation then consists of finding a $c(x)$ such that $a(x)$ in equation (18.23) is a multiple of $g(x)$.

Becoming proficient in handling polynomials over the binary field takes some practice. For example, twice anything is zero, since anything xored with itself gives zero, and thus everything equals its own negative. In particular, $x + x = 0$ and thus $x^2 + 1 = x^2 + (x + x) + 1 = (x + 1)^2$; alternatively, we could prove this by $x^2 + 1 = (x + 1)(x - 1)$ which is the same since $-1 = 1$. How can we factor $x^4 + 1$? $x^4 + 1 = x^4 - 1 = (x^2 + 1)(x^2 - 1) = (x^2 + 1)^2 = (x + 1)^4$. As a last multiplication example, it's easy to show that $(x^3 + x^2 + 1)(x^2 + x + 1) = x^5 + x + 1$. Division is similar to the usual long division of polynomials, but easier. For example, dividing $x^5 + x + 1$ by $x^2 + x + 1$ is performed as follows. First x^5 divided by x^2 gives x^3, so we multiply $x^3(x^2 + x + 1) = x^5 + x^4 + x^3$. Adding this (which is the same as subtracting) leaves us with $x^4 + x^3 + x + 1$ into which x^2 goes x^2 times. This time we add $x^4 + x^3 + x^2$ and are left with $x^2 + x + 1$, and therefore the answer is $x^3 + x^2 + 1$ as expected.

With this understanding of binary polynomial division we can now describe how $c(x)$ is found, given the generator polynomial $g(x)$ and the message polynomial $d(x)$. We multiply $d(x)$ by x^{n-k} and then divide by $g(x)$, the

remainder being taken to be $c(x)$. This works since dividing $x^{n-k}d(x)+c(x)$ by $g(x)$ will now leave a remainder $2c(x) = 0$. For example, the 23/12 Golay code has the generator polynomial $g(x) = 1+x+x^5+x^6+x^7+x^9+x^{11}$; in order to use it we must take in 12 bits at a time, building a polynomial of degree 22 with the message bits as coefficients of the 12 highest powers, and zero coefficients for the 11 lowest powers. We then divide this polynomial by $g(x)$ and obtain a remainder of degree 10, which we then place in the positions previously occupied by zeros.

The polynomial representation is especially interesting for cyclic codes due to an algebraic relationship between the polynomials corresponding to codewords related by circular shifts. A circular shift by m bits of the bitvector $(a_0a_1 \ldots a_{n-1})$ corresponds to the modulo n addition of m to all the powers of x in the corresponding polynomial.

$$a_0x^{0+m \bmod n} + a_1x^{1+m \bmod n} + \ldots + a_{n-1}x^{(n-1)+m \bmod n}$$

This in turn is equivalent to multiplication of the polynomial by x^m modulo $x^n + 1$. To see this consider multiplying a polynomial $a(x)$ by x to form $xa(x) = a_0x+a_1x^2+\ldots+a_{n-1}x^n$. In general, this polynomial is of degree n and thus has too many bits to be a codeword, but by direct division we see that $x^n + 1$ goes into it a_{n-1} times with a remainder $\tilde{a}(x) = a_{n-1} + a_0x + a_1x^2 + \ldots + a_{n-2}x^{n-1}$. Looking carefully at $\tilde{a}(x)$ we see that it corresponds to the circular shift of the bits of $a(x)$. We can write

$$x\,a(x) \;=\; a_{n-1}(x^n + 1) + \tilde{a}(x)$$

and thus say that $xa(x)$ and $\tilde{a}(x)$ are the same modulo $x^n + 1$. Similarly, $(x^2a(x)) \bmod (x^n + 1)$ corresponds to a circular shift of two bits, and $(x^ma(x)) \bmod (x^n + 1)$ to a circular shift of m bits. Thus cyclic codes have the property that if $a(x)$ corresponds to a codeword, then so does $(x^ma(x)) \bmod (x^n + 1)$.

In 1960, two MIT researchers, Irving Reed and Gustave Solomon, realized that encoding bit by bit is not always the best approach to error detection and correction. Errors often come in bursts, and a burst of eight consecutive bit errors would necessitate an extremely strong ECC that could correct any eight-bit errors; but eight consecutive bit errors are contained in at most two bytes, thus if we could work at the byte level, we would only need a two-byte correcting ECC. The simplest byte-oriented code adds a single checkbyte that equals the bit-by-bit xor of all the data bytes to a block of byte-oriented data. This is equivalent to eight interleaved parity checks and can detect any single byte error and many multibyte ones, but cannot correct

any errors. What Reed and Solomon discovered is that by using r checkbytes one can detect any r errors and correct half as many errors. Discussing the theory of Reed-Solomon codes would take us too far astray, but the basic idea is to think of the bytes as polynomials with bits as coefficients

$$B(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

where the b_n are bits. Dividing the bit stream into n bytes and adding all these bytes together as polynomials

$$c_0 = B_0(x) + B_1(x) + B_2(x) + \ldots + B_{n-1}$$

results in the checkbyte mentioned above. Additional checkbytes can be generated that allow detection of the position of the error.

EXERCISES

18.10.1 A systematic code adds $r = n - k$ checkbits and thus allows for $2^r - 1$ different errors to be corrected. So for all single-bit errors (including errors of the checkbits themselves) to be corrected we require $2^r - 1 \geq n = k + r$. For $k = 4$ we require at least $r = 3$, i.e., a 7/4 code. What does this bound predict for the minimum sizes of systematic codes for $k = 3, 5, 8, 11, 12, 16$?

18.10.2 Find the codewords of the 7/4 Hamming code in Hamming's order. Show that the inverse of every Hamming 7/4 codeword is a code word. Show that the sum (modulo 2) of every two codewords is a codeword. What is the minimum Hamming distance d_{min}? Show that the code is not cyclic but that the code in the original order (as given in the text) is.

18.10.3 In the 7/4 Hamming code the inverse of every codeword (i.e., with ones and zeros interchanged) is a codeword as well. Why?

18.10.4 Why are systematic codes often preferred when the error rate is high?

18.10.5 Find equations for the checkbits of the 9/5 Hamming code. How can the bits be arranged for the sum of checkbit indices to point to the error? How should the bits be arranged for the code to be cyclic?

18.10.6 Show that over the binary field $x^n + 1 = (x + 1)(x^{n-1} + x^{n-2} + \ldots + 1)$.

18.10.7 The 16-bit cyclic redundancy check (CRC) error detection code uses the polynomial $1 + x^5 + x^12 + x^{16}$. Write a routine that appends a CRC word to a block of data, and one that tests a block with appended CRC. How can the encoding and decoding be made computationally efficient? Test these routines by adding errors and verifying that the CRC is incorrect.

18.10.8 To learn more about block codes write four programs. The first `bencode` inputs a file and encodes it using a Hamming code; the second `channel` inputs the output of the first and flips bits with probability p (a parameter); the third `bdecode` decodes the file with errors; the last `compare` compares the original and decoded files and reports on the average error P_{err}. Experiment with binary files and plot the empirical P_{err} as a function of p. Experiment with text files and discover how high p can be for the output to be decipherable.

18.11 Convolutional Codes

The codes we discussed in the previous section are typically used when the bits come in natural blocks, but become somewhat constraining when bits are part of real-time 'bit signals'. The reader who prefers filtering by convolution rather than via the FFT (Section 15.2) will be happy to learn that convolutional codes can be used instead of block codes. Convolutional encoders are actually analogous to several simultaneous convolutions; for each time step we shift a bit (or more generally k bits) into a static bit buffer, and then output n bits that depend on the K bits in this buffer. We have already mentioned that in ECC terminology the static buffer is called a shift register, since each time a new bit arrives the oldest bit in the shift register is discarded, while all the other bits in the registers are shifted over, making room for the new bit.

The precise operation of a convolutional encoder is as follows. First the new bit is pushed into the shift register, then all n convolutions are computed (using modulo two arithmetic), and finally these bits are interleaved into a new bit stream. If n convolutions are computed for each input bit the code's rate is $n/1$. Since this isn't flexible enough, we allow k bits to be shifted into the register before the n outputs bits are computed, and obtain an n/k rate code.

The simplest possible convolutional code consists of a two-bit shift register and no nontrivial arithmetic operations. Each time a new bit is shifted into the shift register the bit left in the register and the new bit are output; In other words, denoting the input bits x_n, the outputs bits at time n are x_n and x_{n-2}. If the input bit signal is 1110101000 the output y_n will be 11111100110011000000. A shift register diagram of the type common in the ECC world and the equivalent DSP flow diagram are depicted in Figure 18.12.

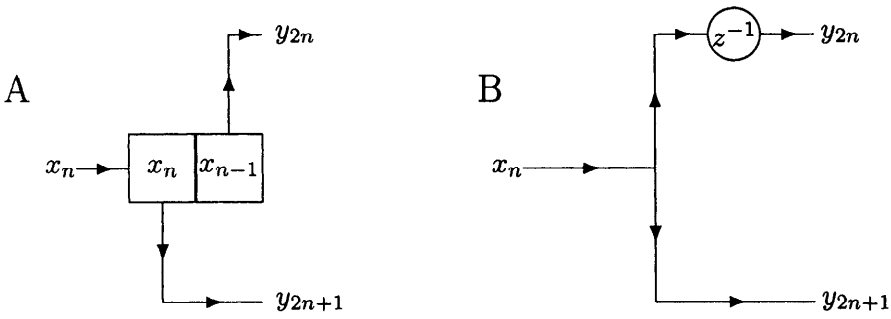

Figure 18.12: A trivial convolutional code. In (A) we see the type of diagram used in error correcting code texts, while in (B) we have the conventional DSP flow diagram equivalent. The output bit signal is formed by interleaving the two outputs into a single bit stream, although this parallel to serial conversion is not explicitly shown.

Why does this code work? As for block codes the idea is that not every combination of bits is a valid output sequence. We cannot say that given a bit the next bit must be the same, since we do not know whether the present bit is already the replica of the previous one. However, an isolated zero or one, as in the bit triplets 010 and 101, can never appear. This fact can be used to detect a single-bit error and even some double-bit errors (e.g., the two middle bits of 00001111). Actually all single-bit errors can be corrected, since with few errors we can easily locate the transitions from 1 to 0 and back and hence deduce the proper phase.

In Figure 18.13 we see a somewhat more complex convolutional code. This code is universally the first presented in textbooks to demonstrate the

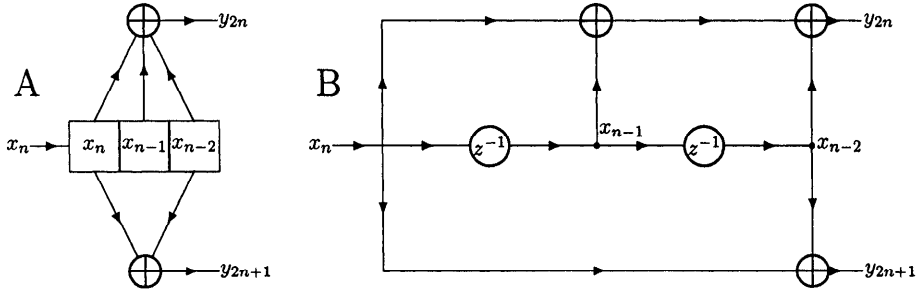

Figure 18.13: A simple convolutional code (the one universally presented in ECC texts). In (A) we see the ECC style diagram, while in (B) we have the conventional DSP flow diagram equivalent. Note that the addition in the latter diagram is modulo 2 (xor).

state	input	output	new state
0(00)	0	0(00)	0(00)
0(00)	1	3(11)	2(10)
1(01)	0	3(11)	0(00)
1(01)	1	0(00)	2(10)
2(10)	0	1(01)	1(01)
2(10)	1	2(10)	3(11)
3(11)	0	2(10)	1(01)
3(11)	1	1(01)	3(11)

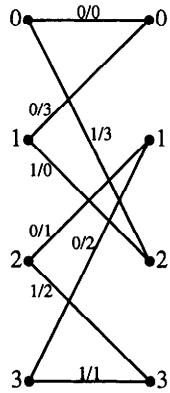

Figure 18.14: The function table and diagram for the simple convolutional code. There are four states and both the output and the new state are dependent on the state and the input bit. The new state is obtained by shifting the present state to the right and placing the input bit into the two's place. In the diagram each arc is labeled by two binary numbers, input/output.

principles of convolutional codes, and we present it in order not to balk tradition. Since there are two delay elements and at each time two bits are output, any input bit can only influence six output bits. We would like to call this number the 'influence time', but ECC terminology constrains us to call it the 'constraint length'. Since the constraint length is six, not all combinations of seven or more consecutive bits are possible. For example, the sequence 00000010 is not a possible output, since six consecutive 0s imply that the shift register now contains 0s, and inputting a 1 now causes two 1s to be output.

We have specified the output of a convolutional encoder in terms of the present and past input bits, just as we specify the output of an FIR filter in terms of the present and past input values. The conventional methodology in the ECC literature prefers the state-space description (see Section 6.3) where the present output bits are computed based on the present input and the internal state of the encoder. The natural way to define the internal state of the encoder is via the two bits x_{n-1} and x_{n-2}, or by combining these two bits into $s = 2x_{n-1} + x_{n-2}$ which can take on values 0, 1, 2, and 3. The encoder output can now be described as a function of the present input and the value of the internal state s. We tabulate this function and depict it graphically in Figure 18.14. In the figure each arc is labeled by a pair of numbers in binary notation. The first number is the input required to cause the transition from the pre-arc state to the post-arc one; the second number is the output emitted during such a transition.

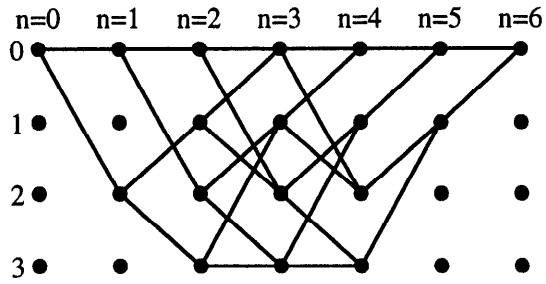

Figure 18.15: Trellis diagram for the simple convolutional code. We draw all the paths from state zero at $n = 0$ that return to state zero at time $n = 6$.

Accepting the state-space philosophy, our main focus of interest shifts from following the output bits to following the behavior of the encoder's internal state as a function of time. The main purpose of the encoder is to move about in state-space under the influence of the input bits; the output bits are only incidentally generated by the transitions from one state to another. We can capture this behavior best using a *trellis*, an example of which is given in Figure 18.15. A trellis is a graph with time advancing from left to right and the encoder states from top to bottom. Each node represents a state at a given time. In the figure we have drawn lines to represent all the possible paths of the encoder through the trellis that start from state 0 and end up back in state 0 after six time steps. Each transition should be labeled with input and output bits as in Figure 18.14, but these labels have been hidden for the sake of clarity.

How are convolutional codes decoded? The most straightforward way is by exhaustive search, that is, by trying all possible inputs to the encoder, generating the resulting outputs, and comparing these outputs with the received bit signal. The decoder selects the output bit sequence that is closest (in Hamming distance) to the bit signal, and outputs the corresponding input. In Table 18.2 we tabulate the output of the traditional convolutional code of Figure 18.13 for all possible six-bit inputs. Upon receiving twelve bits that are claimed to be the output of this coder from the all-zero initial state, we compare them to the right-hand column, select the row closest in the Hamming sense, and output the corresponding left-hand entry. This exhaustive decoding strategy rapidly gets out of hand since the number of input signals that must be tried increases exponentially with the number of bits received. We need to find a more manageable algorithm.

The most commonly employed decoding method is the *Viterbi algorithm*, which is a dynamic programming algorithm, like the DTW algorithm of

input	output	input	output	input	output	input	output
000000	000000000000	000010	000000001110	000001	000000000011	000011	000000001101
100000	111011000000	100010	111011001110	100001	111011000011	100011	111011001101
010000	001110110000	010010	001110111110	010001	001110110011	010011	001110111101
110000	110101110000	110010	110101111110	110001	110101110011	110011	110101111101
001000	000011101100	001010	000011100010	001001	000011101111	001011	000011100001
101000	111000101100	101010	111000100010	101001	111000101111	101011	111000100001
011000	001101011100	011010	001101010010	011001	001101011111	011011	001101010001
111000	110110011100	111010	110110010010	111001	110110011111	111011	110110010001
000100	000000111011	000110	000000110101	000101	000000111000	000111	000000110110
100100	111011111011	100110	111011110101	100101	111011111000	100111	111011110110
010100	001110001011	010110	001110000101	010101	001110001000	010111	001110000110
110100	110101001011	110110	110101000101	110101	110101001000	110111	110101000110
001100	000011010111	001110	000011011001	001101	000011010100	001111	000011011010
101100	111000010111	101110	111000011001	101101	111000010100	101111	111000011010
011100	001101100111	011110	001101101001	011101	001101100100	011111	001101101010
111100	110110100111	111110	110110101001	111101	110110100100	111111	110110101010

Table 18.2: Exhaustive enumeration of the simple convolutional code (the one universally presented in ECC texts). We input all possible input sequences of six bits and generate the outputs (assuming the shift register is reset to all zeros each time). These outputs can be compared with received bit signal.

Section 8.7. To understand this algorithm consider first the following related problem. You are given written directions to go from your house to the house of a fellow DSP enthusiast in the next town. A typical portion of the directions reads something like 'take the third right turn and proceed three blocks to the stop sign; make a left and after two kilometers you see a bank on you right'. Unfortunately, the directions are handwritten on dirty paper and you are not sure whether you can read them accurately. Was that the third right or the first? Does it say three blocks or two blocks? One method to proceed is to follow your best bet at each step, but to keep careful track of all supplied information. If you make an error then at some point afterward the directions no longer make any sense. There was supposed to be a bank on the right but there isn't any, or there should have been a stop sign after 3 blocks but you travel 5 and don't see one. The logical thing to do is to backtrack to the last instruction in doubt and to try something else. This may help, but you might find that the error occurred even earlier and there just happened to be a stop sign after three blocks in the incorrect scenario.

This kind of problem is well known in computer science, where it goes under the name of the 'directed search' problem. Search problems can be represented as trees. The root of the tree is the starting point (your home) and each point of decision is a node. Solving a search problem consists of going from node to node until the goal is reached (you get to the DSP enthusiast's house). A search problem is 'directed' when each node can be assigned a cost that quantifies its consistency with the problem specification

(how well it matches the directions so far). Directed search can be solved more systematically than arbitrary search problems since choosing nodes with lower cost brings us closer to the goal.

There are two main approaches to solving search problems. 'Depth-first' solution, such as that we suggested above, requires continuing along a path until it is obviously wrong, and then backtracking to the point of the last decision. One then continues along another path until it becomes impossible. The other approach is 'breadth-first'. Breadth-first solution visits all decision nodes of the same depth (the same distance from the starting node) before proceeding to deeper nodes. The breadth-first solution to the navigation problem tries every possible reading of the directions, going only one step. At each such step you assign a cost, but resist the temptation to make any decision even if one node has a low cost (i.e., matches the description well) since some other choice may later turn out to be better.

The decoding of a convolutional code is similar to following directions. The algebraic connection between the bits constitute a consistency check very much like the stop sign after three blocks and the bank being on the right. The Viterbi algorithm is a breadth-first solution that exploits the state-space description of the encoder. Assume that we know that the encoder is in state 0 at time $n = 0$ and start receiving the encoded (output) bits. Referring back to Figure 18.15, just as the decoder generated particular transitions based on the input bits (the first number of the input/output pair in Figure 18.14) the Viterbi decoder tries to guess which transition took place based on the output bits (the second number of the pair). Were the received bit signal error free, the task would be simple and uniquely defined. In the presence of bit errors sometimes we will make an improper transition, and sometimes we cannot figure out what to do at all. The breadth-first dynamic programming approach is to make *all* legal transitions, but to calculate the Hamming distance between the received bits and the encoder output bits that would actually have caused this transition. We store in each node of the trellis the minimal accumulated Hamming distance for paths that reach that node. We have thus calculated the minimal number of errors that need to have occurred for each possible internal state. We may guess that the path with minimal number of errors is the correct one, but this would be too hasty a decision. The proper way to identify the proper path in state-space is to have patience and watch it emerge.

To see how this happens let's work out an example, illustrated in Figure 18.16. We'll use as usual the simple convolutional code of Figure 18.13 and assume that the true input to the encoder was all zeros. The encoder output was thus all zeros as well, but the received bit signal has an erroneous

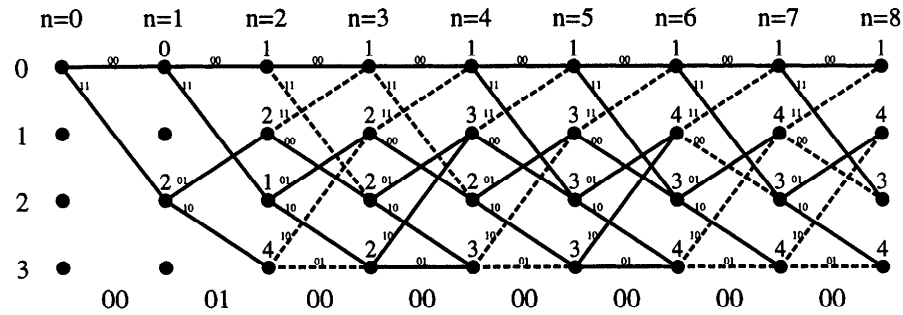

Figure 18.16: Viterbi decoding for the simple convolutional code. The bottom line contains the received bit signal, which contains a single bit error. The trellis nodes are labeled with the accumulated number of bit errors and the arcs with the output bits corresponding to the transition. Dash arcs are ones that do not survive because there is a path to the post-arc node that has fewer bit errors. The single bit error is corrected after 6 time steps.

one due to noise. We assign costs to the nodes in the trellis starting from state zero at time $n = 0$. In the first time step we can reach state 0 if 00 were transmitted and state 2 were 11 transmitted. Since 00 is received, state 0 is labeled as being reachable without bit errors, while state 2 is labeled with a 2 representing two bit errors. In the next time step the bit error occurs. From the 0 state at $n = 1$ the code can transition either to itself or to state 2, implying single-bit errors. From state 2 at $n = 1$ we would arrive at state 1 assuming the 01 that was received was indeed transmitted, thus accruing no additional error; the state 1 node at $n = 2$ is thus labeled with total error 2 since the entire path from state 0 at $n = 0$ was 1101 as compared with the 0001 that was received. From state 2 at $n = 1$ we could also arrive at state 3 at $n = 2$ were 10 to have been sent, resulting in the maximal total number of bit errors (1110 as compared to 0001).

In this way we may continue to compute the total number of bit errors to reach a given state at a given time. In the next step we can reach the 0 state via two different paths. We can transition from state 0 at $n = 2$, implying that the total bit sequence transmitted was 000000, or we can get there from state 2 at $n = 2$ were the sequence transmitted to have been 110111. Which cost do we assign to the node? Viterbi's insight (and the basis of all 'dynamic programming' algorithms) was that we need only assign the lower of the two costs. This reasoning is not hard to follow, since we are interested in finding the lowest-cost path (i.e., the path that assumes the fewest bit errors in the received bit signal). Suppose that later on we determine that the lowest-cost path went through this node, and we are interested in determining how the states evolved up to this point. It is obvious that the lowest-error path that

reaches this node must have taken the route from the 0 state at $n = 2$. So the global optimum is found by making the local decision to accept the cost that this transition implies. In the figure we draw the transition that was ruled out as a dashed line, signifying that the best path does not go through here. So at each time only four surviving paths must be considered.

In general, at time step n we follow all legal transitions from nodes 0, 1, 2, and 3 at time $n - 1$, add the new Hamming distance to the cost already accrued in the pre-arc node, and choose the lower of the costs entering the post-arc node. In the figure the transitions that give minimal cost are depicted by solid lines. If two transitions give the same accumulated cost we show both as solid lines, although in practical implementations usually one is arbitrarily chosen. The reader should carefully follow the development of the trellis in the figure from left to right.

Now that we have filled in the trellis diagram, how does this help us decode the bit signal? You might assume that at some point we must break down and choose the node with minimal cost. For example, by time $n = 5$ the path containing only state 0 is clearly better than the other four paths, having only one error as compared with at least three errors for all other paths. However, there is no need to make such risky quantitative decisions. Continuing on until time $n = 8$ the truth regarding the error in the fourth bit finally comes to light. Backtracking through the chosen transitions (solid lines in the figure) shows that all the surviving paths converge on state 0 at time $n = 6$. So without quantitatively deciding between the different states at time $n = 8$ we still reach the conclusion that the most likely transmitted bit signal started 00000000, correcting the mistakenly received bit. It is easy to retrace the arcs of the selected path, but this time using the encoder input bits (the first number from Figure 18.13) to state that the uncoded information started 000.

In a more general setting, the Viterbi decoder for a convolutional code which employs m delays (i.e., has $s = 2^m$ possible internal states), fills in a trellis of s vertical states. At each time step there will be s surviving paths, and each is assigned a cost that equals the minimal number of bit errors that must have occurred for the code to have reached this state. Backtracking, we reach a time where all the surviving paths converge, and we accept the bits up to this point as being correct and output the original information corresponding to the transitions made. In order to save computational time most practical implementations do not actually check for convergence, but rather assume that all paths have converged some fixed time L in the past. The Viterbi decoder thus outputs predictions at each time step, these information bits being delayed by L.

EXERCISES

18.11.1 In exercise 8.7.1 we introduced the games of doublets. What relationships exist between this game and the decoding of convolutional codes?

18.11.2 Each of the convolutions that make up a convolutional code can be identified by its impulse response, called the generator sequence in coding theory. What is the duration of the impulse response if the shift register is of length K? The constraint length is defined to be the number of output bits that are influenced by an input bit. What is the constraint length if the shift register length is K, and each time instant k bits are shifted in and n are output?

18.11.3 Which seven-tuples of bits never appear as outputs of the simple convolutional code given in the text?

18.11.4 Repeat the last exercise of the previous section for convolutional codes. You need to replace the first program with `cencode` and the third with `cdecode`. Use the Viterbi algorithm.

18.11.5 The convolutional code $y_{2n} = x_n + x_{n-1}, y_{2n+1} = x_n + x_{n-2}$ is even simpler than the simple code we discussed in the text, but is not to be used. To find out why, draw the trellis diagram for this code. Show that this code suffers from *catastrophic error propagation*, that is, misinterpreted bits can lead to the decoder making an unlimited number of errors. (Hint: Assume that all zeros are transmitted and that the decoder enters state 3.)

18.12 PAM and FSK

Shannon's information capacity theorem is not constructive, that is, it tells us what information rate *can* be achieved, but does not actually supply us with a method for achieving this rate. In this section we will begin our quest for efficient transmission techniques, methods that will approach Shannon's limit on real channels.

Let's try to design a modem. We assume that the input is a stream of bits and the output a single analog signal that must pass through a noisy band-limited channel. Our first attempt will be very simplistic. We will send a signal value of $+1$ for every one in the input data stream, and a 0 for every zero bit. We previously called this signal NRZ, and we will see that it is the simplest type of **P**ulse **A**mplitude **M**odulation (PAM). As we know, the bandwidth limitation will limit the rate that our signal can change, and so we will have a finite (in fact rather low) information transmission rate.