MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 18, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 5
Coping with Bit Errors using Error Correction Codes

Recall our main goal in designing digital communication networks: to send information both reliably and efficiently between nodes. Meeting that goal requires the use of techniques to combat bit errors, which are inevitable in both commmunication channels and storage media (storage may be viewed as "communication across time"; you store something now and usually want to be able to retrieve it later).

The key idea we will apply to achieve reliable communication is the addition of *redundancy* to the transmitted data, to improve the probability that the original message can be reconstructed from the possibly corrupted data that is received. The sender has an *encoder* whose job is to take the message and process it to produce the *coded bits* that are then sent over the channel. The receiver has a *decoder* whose job is to take the received (coded) bits and to produce its best estimate of the message. The encoder-decoder procedures together constitute **channel coding**; good channel codes provide **error correction** capabilities that reduce the bit error rate (i.e., the probability of a bit error).

With proper design, full error correction may be possible, provided only a small number of errors has occurred. Even when too many errors have occurred to permit correction, it may be possible to perform *error detection*. Error detection provides a way for the receiver to tell (with high probability) if the message was decoded correctly or not. Error detection usually works by the sender and receiver using a different code from the one used to correct errors; common examples include the *cyclic redundancy check* (CRC) or *hash functions*. These codes take n-bit messages and produce a compact "signature" of that message that is much smaller than the message (e.g., the popular CRC-32 scheme produces a 32-bit signature of an arbitrarily long message). The sender computes and transmits the signature along with the message bits, usually appending it to the end of the message. The receiver, after running the decoder to correct errors, then computes the signature over its estimate of the message bits and compares that signature to its estimate of the signature bits in the received data. If the computed and estimated signatures are not equal, then the receiver considers the message to have one or more bit errors; otherwise, it assumes that the message has been received correctly. This latter assumption is probabilistic: there is some non-zero (though very small, for good signatures) probability that the estimated and computed signatures match, but the receiver's decoded message is different from the sender's. If the signatures don't match, the receiver and sender may use some higher-layer protocol to arrange for the message to be retransmitted; we will study such schemes later. We will not study error detection codes like CRC or hash functions in this course.

Our plan for this chapter is as follows. To start, we will assume a **binary symmetric channel** (BSC), which we defined and explained in the previous chapter; here the probability of a bit "flipping" is ε. Then, we will discuss and analyze a simple redundancy scheme called a *replication code*, which will simply make n copies of any given bit. The replication code has a *code rate* of $1/n$—that is, for every useful *message* bit, we end up transmitting n total bits. The overhead of the replication code of rate c is $1 - {}^1\!/_n$, which is rather high for the error correcting power of the code. We will then turn to the key ideas that allow us to build powerful codes capable of correcting errors without such a high overhead (or equivalently, capable of correcting far more errors at a given code rate compared to the replication code).

There are two big, inter-related ideas used in essentially all error correction codes. The first is the notion of **embedding**, where the messages one wishes to send are placed in a geometrically pleasing way in a larger space so that the distance between any two valid points in the embedding is large enough to enable the correction and detection of errors. The second big idea is to use **parity calculations**, which are linear functions over the bits we wish to send, to generate the redundancy in the bits that are actually sent. We will study examples of embeddings and parity calculations in the context of two classes of codes: **linear block codes**, which are an instance of the broad class of **algebraic codes**, and **convolutional codes**, which are perhaps the simplest instance of the broad class of **graphical codes**.

We start with a brief discussion of bit errors.

■ 5.1 Bit Errors and BSC

A BSC is characterized by one parameter, ε, which we can assume to be $< 1/2$, the probability of a bit error. It is a natural discretization of AWGN, which is also a single-parameter model fully characterized by the variance, σ^2. We can determine ε empirically by noting that if we send N bits over the channel, the expected number of erroneously received bits is $N \cdot \varepsilon$. By sending a long known bit pattern and counting the fraction or erroneously received bits, we can estimate ε, thanks to the *law of large numbers*. In practice, even when the BSC is a reasonable error model, the range of ε could be rather large, between 10^{-2} (or even a bit higher) all the way to 10^{-10} or even 10^{-12}. A value of ε of about 10^{-2} means that messages longer than a 100 bits will see at least one error on average; given that the typical unit of communication over a channel (a "packet") is generally between 500 bits and 12000 bits (or more, in some networks), such an error rate is too high.

But is ε of 10^{-12} small enough that we don't need to bother about doing any error correction? The answer often depends on the data rate of the channel. If the channel has a rate of 10 Gigabits/s (available today even on commodity server-class computers), then the "low" ε of 10^{-12} means that the receiver will see one error every 10 seconds on average if the channel is continuously loaded. Unless we include some mechanisms to mitigate the situation, the applications using the channel may find errors occurring too frequently.

On the other hand, a ε of 10^{-12} may be fine over a communication channel running at 10 Megabits/s, as long as there is some way to detect errors when they occur.

In the BSC model,a transmitted bit b (0 or 1) is interpreted by the receiver as $1 - b$ with probability ε and as b with probability ε. In this model, each bit is corrupted independently and with equal probability (which makes this an "iid" random process, for "*independent and identically distributed*"). We call ε the *bit-flip probability* or the "error probability", and sometimes abuse notation and call it the "bit error rate" (it isn't really a "rate", but the term is still used in the literature). Given a packet of size S bits, it is straightforward to calculate the probability of the entire packet being received correctly when sent over a BSC with bit-flip probability ε:

$$\mathbb{P}(\text{packet received correctly}) = (1 - \varepsilon)^S.$$

The packet error probability, i.e., the probability of the packet being incorrect, is 1 minus this quantity, because a packet is correct if and only if all its bits are correct.

Hence,

$$\mathbb{P}(\text{packet error}) = 1 - (1 - \varepsilon)^S. \tag{5.1}$$

When $\varepsilon << 1$, a simple first-order approximation of the PER is possible because $(1 + x)^N \approx 1 + Nx$ when $|x| << 1$. That approximation gives the pleasing result that, when $\varepsilon << 1$,

$$\mathbb{P}(\text{packet error}) \approx 1 - (1 - S\varepsilon) = S\varepsilon. \tag{5.2}$$

The BSC is perhaps the simplest discrete error model that is realistic, but real-world channels exhibit more complex behaviors. For example, over many wireless and wired channels as well as on storage media (like CDs, DVDs, and disks), errors can occur in *bursts*. That is, the probability of any given bit being received wrongly depends on recent history: the probability is higher if the bits in the recent past were received incorrectly. Our goal is to develop techniques to mitigate the effects of both the BSC and burst errors. We'll start with techniques that work well over a BSC and then discuss how to deal with bursts.

■ 5.2 The Simplest Code: Replication

In general, a channel code provides a way to map message words to codewords (analogous to a source code, except here the purpose is not compression but rather the addition of redundancy for error correction or detection). In a replication code, each bit b is encoded as n copies of b, and the result is delivered. If we consider bit b to be the *message word*, then the corresponding *codeword* is b^n (i.e., $bb...b$, n times). In this example, there are only two possible message words (0 and 1) and two corresponding codewords. The replication code is absurdly simple, yet it's instructive and sometimes even useful in practice!

But how well does it correct errors? To answer this question, we will write out the probability of overcoming channel errors for the BSC error model with the replication code. That is, if the channel corrupts each bit with probability ε, what is the probability that the receiver decodes the received codeword correctly to produce the message word that was sent?

The answer depends on the decoding method used. A reasonable decoding method is *maximum likelihood decoding*: given a received codeword, r, which is some n-bit combina-

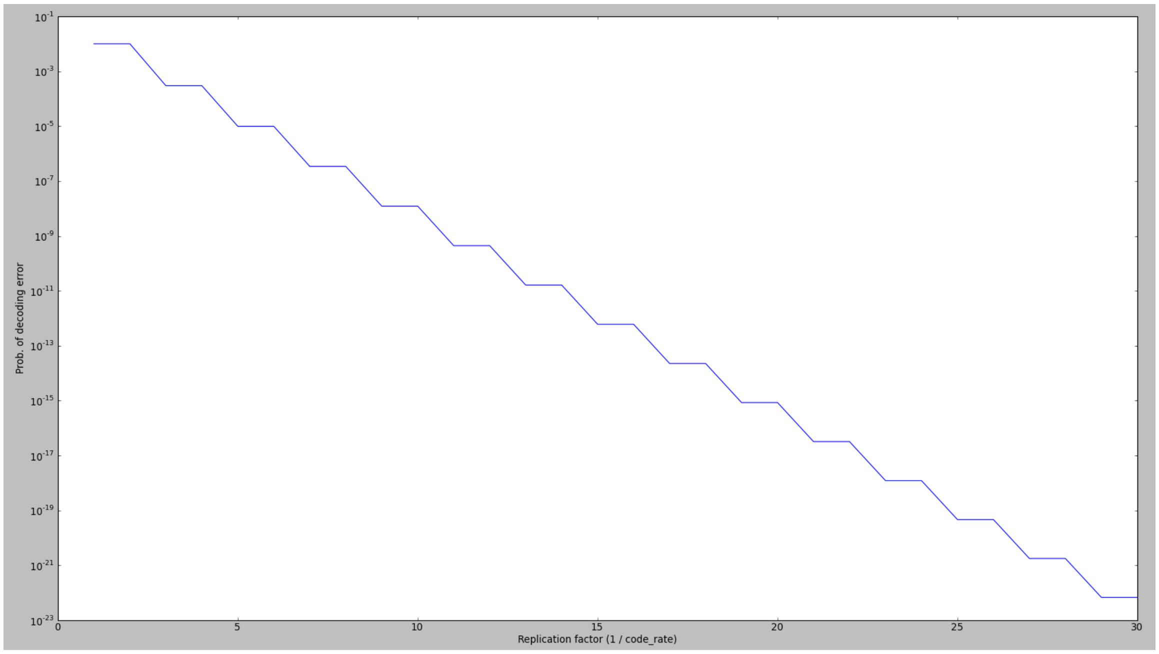


Figure 5-1: Probability of a decoding error with the replication code that replaces each bit *b* with *n* copies of *b*. The code rate is $1/n$.

tion of 0′s and 1′s, the decoder should produce the most likely message that could have caused r to be received. Since the BSC error probability, ε, is smaller than $1/2$, the most likely option is the codeword that has the most number of bits in common with r. This decoding rule results in the minimum probability of error when all messages are equally likely.

Hence, the decoding process is as follows. First, count the number of 1′s in r. If there are more than $c/2$ 1′s, then decode the message as 1. If there are more than $c/2$ 0′s, then decode the message as 0. When c is odd, each codeword will be decoded unambiguously. When c is even, and has an equal number of 0′s and 1′s, the decoder can't really tell whether the message was a 0 or 1, and the best it can do is to make an arbitrary decision. (We have tacitly assumed that the *a priori* probability of sending a message 0 is the same as that of sending a 1.)

We can write the probability of decoding error for the replication code as follows, being a bit careful with the limits of the summation:

$$P(\text{decoding error}) = \begin{cases} \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \binom{n}{i} \varepsilon^i (1-\varepsilon)^{n-i} & \text{if } n \text{ odd} \\ \sum_{i=\frac{n}{2}+1}^{n} \binom{n}{i} \varepsilon^i (1-\varepsilon)^{n-i} + \frac{1}{2}\binom{n}{n/2} \varepsilon^{n/2} (1-\varepsilon)^{n/2} & \text{if } n \text{ even} \end{cases} \tag{5.3}$$

The notation $\binom{n}{i}$ denotes the number of ways of selecting i objects (in this case, bit positions) from n objects.

When n is even, we add a term at the end to account for the fact that the decoder has a fifty-fifty chance of guessing correctly when it receives a codeword with an equal number of 0′s and 1′s.

Figure 5-1 shows the probability of decoding error as a function of the replication factor,

n, for the replication code, computed using Equation (5.3). The y-axis is on a log scale, and the probability of error is more or less a straight line with negative slope (if you ignore the flat pieces), which means that the decoding error probability decreases exponentially with the code rate. It is also worth noting that the error probability is the same when $n = 2\ell$ as when $n = 2\ell - 1$. The reason, of course, is that the decoder obtains no additional information that it already didn't know from any $2\ell - 1$ of the received bits.

Despite the exponential reduction in the probability of decoding error as n increases, the replication code is extremely inefficient in terms of the overhead it incurs, for a given rate, $1/n$. As such, it is used only in situations when bandwidth is plentiful and there isn't much computation time to implement a more complex decoder.

We now turn to developing more sophisticated codes. There are two big related ideas: *embedding messages into spaces in a way that achieves structural separation* and *parity (linear) computations over the message bits*.

■ 5.3 Embeddings and Hamming Distance

Let's start our investigation into error correction by examining the situations in which error detection and correction are possible. For simplicity, we will focus on single-error correction (SEC) here. By that we mean codes that are guaranteed to produce the correct message word, given a received codeword with zero or one bit errors in it. If the received codeword has more than one bit error, then we can make no guarantees (the method might return the correct message word, but there is at least one instance where it will return the wrong answer).

There are 2^n possible n-bit strings. Define the *Hamming distance* (HD) between two n-bit words, w_1 and w_2, as the number of bit positions in which the messages differ. Thus $0 \leq \text{HD}(w_1, w_2) \leq n$.

Suppose that $\text{HD}(w_1, w_2) = 1$. Consider what happens if we transmit w_1 and there's a single bit error that inconveniently occurs at the one bit position in which w_1 and w_2 differ. From the receiver's point of view it just received w_2—the receiver can't detect the difference between receiving w_1 with a unfortunately placed bit error and receiving w_2. In this case, we cannot guarantee that all single bit errors will be corrected if we choose a code where w_1 and w_2 are both valid codewords.

What happens if we increase the Hamming distance between any two valid codewords to 2? More formally, let's restrict ourselves to only sending some subset $\mathcal{S} = \{w_1, w_2, ..., w_s\}$ of the 2^n possible words such that

$$\text{HD}(w_i, w_j) \geq 2 \text{ for all } w_i, w_j \in \mathcal{S} \text{ where } i \neq j \tag{5.4}$$

Thus if the transmission of w_i is corrupted by a single error, the result is *not* an element of $\mathcal{S}$ and hence can be detected as an erroneous reception by the receiver, which knows which messages are elements of $\mathcal{S}$. A simple example is shown in Figure 5-2: 00 and 11 are valid codewords, and the receptions 01 and 10 are surely erroneous.

We define the *minimum Hamming distance of a code* as the minimum Hamming distance between any two codewords in the code. From the discussion above, it should be easy to see what happens if we use a code whose minimum Hamming distance is D. We state the property formally:

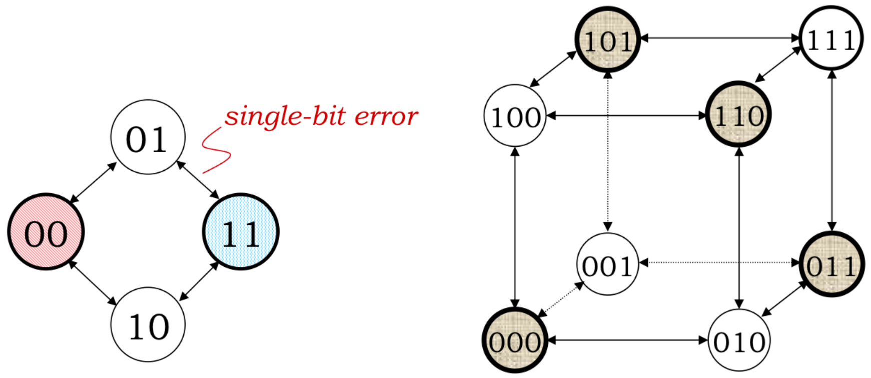


Figure 5-2: Codewords separated by a Hamming distance of 2 can be used to detect single bit errors. The codewords are shaded in each picture. The picture on the left is a (2,1) repetition code, which maps 1-bit messages to 2-bit codewords. The code on the right is a (3,2) code, which maps 2-bit messages to 3-bit codewords.

Theorem 5.1 *A code with a minimum Hamming distance of D can detect* any *error pattern of $D-1$ or fewer errors. Moreover, there is at least one error pattern with D errors that cannot be detected reliably.*

Hence, if our goal is to detect errors, we can use an embedding of the set of messages we wish to transmit into a bigger space, so that the minimum Hamming distance between any two codewords in the bigger space is at least one more than the number of errors we wish to detect. (We will discuss how to produce such embeddings in the subsequent sections.)

But what about the problem of *correcting* errors? Let's go back to Figure 5-2, with $\mathcal{S} = \{00, 11\}$. Suppose the received sequence is 01. The receiver can tell that a single error has occurred, but it can't tell whether the correct data sent was 00 or 11—both those possible patterns are equally likely under the BSC error model.

Ah, but we can extend our approach by producing an embedding with more space between valid codewords! Suppose we limit our selection of messages in $\mathcal{S}$ even further, as follows:

$$\text{HD}(w_i, w_j) \geq 3 \text{ for all } w_i, w_j \in \mathcal{S} \text{ where } i \neq j \tag{5.5}$$

How does it help to increase the minimum Hamming distance to 3? Let's define one more piece of notation: let $\mathcal{E}_{w_i}$ be the set of messages resulting from corrupting w_i with a single error. For example, $\mathcal{E}_{000} = \{001, 010, 100\}$. Note that $\text{HD}(w_i, \text{an element of } \mathcal{E}_{w_i}) = 1$.

With a minimum Hamming distance of 3 between the valid codewords, observe that there is no intersection between $\mathcal{E}_{w_i}$ and $\mathcal{E}_{w_j}$ when $i \neq j$. Why is that? Suppose there was a message w_k that was in both $\mathcal{E}_{w_i}$ and $\mathcal{E}_{w_j}$. We know that $\text{HD}(w_i, w_k) = 1$ and $\text{HD}(w_j, w_k) = 1$, which implies that w_i and w_j differ in at most two bits and consequently $\text{HD}(w_i, w_j) \leq 2$. (This result is an application of Theorem 5.2 below, which states that the Hamming distance satisfies the triangle inequality.) That contradicts our specification that their minimum Hamming distance be 3. So the $\mathcal{E}_{w_i}$ don't intersect.

So now we can *correct* single bit errors as well: the received message is either a member of $\mathcal{S}$ (no errors), or is a member of some particular $\mathcal{E}_{w_i}$ (one error), in which case the receiver can deduce the original message was w_i. Here's another simple example: let $\mathcal{S} = \{000, 111\}$.

So $\mathcal{E}_{000} = \{001, 010, 100\}$ and $\mathcal{E}_{111} = \{110, 101, 011\}$ (note that $\mathcal{E}_{000}$ doesn't intersect $\mathcal{E}_{111}$). Suppose the received sequence is 101. The receiver can tell there has been a single error because $101 \notin \mathcal{S}$. Moreover it can deduce that the original message was most likely 111 because $101 \in \mathcal{E}_{111}$.

We can formally state some properties from the above discussion, and specify the error-correcting power of a code whose minimum Hamming distance is D.

Theorem 5.2 *The Hamming distance between n-bit words satisfies the triangle inequality. That is,* $HD(x, y) + HD(y, z) \geq HD(x, z)$.

Theorem 5.3 *For a BSC error model with bit error probability* $< 1/2$, *the maximum likelihood decoding strategy is to map any received word to the valid codeword with smallest Hamming distance from the received one (ties may be broken arbitrarily).*

Theorem 5.4 *A code with a minimum Hamming distance of D can correct* any *error pattern of* $\lfloor \frac{D-1}{2} \rfloor$ *or fewer errors. Moreover, there is at least one error pattern with* $\lfloor \frac{D-1}{2} \rfloor + 1$ *errors that cannot be corrected reliably.*

Equation (5.5) gives us a way of determining if single-bit error correction can always be performed on a proposed set $\mathcal{S}$ of transmission messages—we could write a program to compute the Hamming distance between all pairs of messages in $\mathcal{S}$ and verify that the minimum Hamming distance was at least 3. We can also easily generalize this idea to check if a code can always correct more errors. And we can use the observations made above to decode any received word: just find the closest valid codeword to the received one, and then use the known mapping between each distinct message and the codeword to produce the message. The message will be the correct one if the actual number of errors is no larger than the number for which error correction is guaranteed. The check for the nearest codeword may be exponential in the number of message bits we would like to send, making it a reasonable approach only if the number of bits is small.

But how do we go about finding a good embedding (i.e., good code words)? This task isn't straightforward, as the following example shows. Suppose we want to reliably send 4-bit messages so that the receiver can correct all single-bit errors in the received words. Clearly, we need to find a set of messages $\mathcal{S}$ with 2^4 elements. What should the members of $\mathcal{S}$ be?

The answer isn't obvious. Once again, we could write a program to search through possible sets of n-bit messages until it finds a set of size 16 with a minimum Hamming distance of 3. An exhaustive search shows that the minimum n is 7, and one example of $\mathcal{S}$ is:

$$\begin{array}{cccc}
0000000 & 1100001 & 1100110 & 0000111 \\
0101010 & 1001011 & 1001100 & 0101101 \\
1010010 & 0110011 & 0110100 & 1010101 \\
1111000 & 0011001 & 0011110 & 1111111
\end{array}$$

But such exhaustive searches are impractical when we want to send even modestly longer messages. So we'd like some constructive technique for building $\mathcal{S}$. Much of the theory and practice of coding is devoted to finding such constructions and developing efficient encoding and decoding strategies.

Broadly speaking, there are two classes of code constructions, each with an enormous number of example instances. The first is the class of **algebraic block codes**. The second is the class of **graphical codes**. We will study two simple examples of **linear block codes**, which themselves are a sub-class of algebraic block codes: rectangular parity codes and Hamming codes. We also note that the replication code discussed in Section 5.2 is an example of a linear block code.

In the next two chapters, we will study **convolutional codes**, a sub-class of graphical codes.

■ 5.4 Linear Block Codes and Parity Calculations

Linear block codes are examples of algebraic block codes, which take the set of k-bit messages we wish to send (there are 2^k of them) and produce a set of 2^k codewords, each n bits long ($n \geq k$) using *algebraic operations* over the block. The word "block" refers to the fact that any long bit stream can be broken up into k-bit blocks, which are each then expanded to produce n-bit codewords that are sent.

Such codes are also called (n, k) codes, where k message bits are combined to produce n code bits (so each codeword has $n - k$ "redundancy" bits). Often, we use the notation (n, k, d), where d refers to the minimum Hamming distance of the block code. The *rate* of a block code is defined as k/n; the larger the rate, the less the redundancy overhead incurred by the code.

A *linear* code (whether a block code or not) produces codewords from message bits by restricting the algebraic operations to *linear functions* over the message bits. By linear, we mean that any given bit in a valid codeword is computed as the weighted sum of one or more original message bits.

Linear codes, as we will see, are both powerful and efficient to implement. They are widely used in practice. In fact, *all* the codes we will study—including convolutional codes—are linear, as are most of the codes widely used in practice. We already looked at the properties of a simple linear block code: the replication code we discussed in Section 5.2 is a linear block code with parameters $(n, 1, n)$.

An important and popular class of linear codes are *binary linear codes*. The computations in the case of a binary code use arithmetic modulo 2, which has a special name: algebra in a *Galois Field* of order 2, also denoted $\mathbb{F}_2$. A field must define rules for addition and multiplication, and their inverses. Addition in $\mathbb{F}_2$ is according to the following rules: $0 + 0 = 1 + 1 = 0; 1 + 0 = 0 + 1 = 1$. Multiplication is as usual: $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0; 1 \cdot 1 = 1$. We leave you to figure out the additive and multiplicative inverses of 0 and 1. Our focus in this book will be on linear codes over $\mathbb{F}_2$, but there are natural generalizations to fields of higher order (in particular, Reed Solomon codes, which are over Galois Fields of order 2^q).

A linear code is characterized by the following theorem, which is both a necessary and a sufficient condition for a code to be linear:

Theorem 5.5 *A code is linear if, and only if, the sum of any two codewords is another codeword.*

For example, the block code defined by codewords $000, 101, 011$ is *not* a linear code, because $101 + 011 = 110$ is not a codeword. But if we add 110 to the set, we get a linear code because the sum of any two codewords is now another codeword. The code $000, 101, 011, 110$ has a minimum Hamming distance of 2 (that is, the smallest Hamming distance between any two codewords in 2), and can be used to detect all single-bit errors that occur during the transmission of a code word. You can also verify that the minimum Hamming distance of this code is equal to the smallest number of 1's in a non-zero codeword. In fact, that's a general property of all linear block codes, which we state formally below:

Theorem 5.6 *Define the weight of a codeword as the number of 1's in the word. Then, the minimum Hamming distance of a linear block code is equal to the weight of the non-zero codeword with the smallest weight.*

To see why, use the property that the sum of any two codewords must also be a codeword, and that the Hamming distance between any two codewords is equal to the weight of their sum (i.e., $\text{weight}(u + v) = \text{HD}(u, v)$). (In fact, the Hamming distance between any two bit-strings of equal length is equal to the weight of their sum.) We leave the complete proof of this theorem as a useful and instructive exercise for the reader.

The rest of this section shows how to construct linear block codes over $\mathbb{F}_2$. For simplicity, and without much loss of generality, we will focus on correcting single-bit errors. i.e., on *single-error correction* (SEC) codes.. We will show two ways of building the set $\mathcal{S}$ of transmission messages to have single-error correction capability, and will describe how the receiver can perform error correction on the (possibly corrupted) received messages.

We will start with the *rectangular parity* code in Section 5.5, and then discuss the cleverer and more efficient *Hamming code* in Section 5.7.

■ 5.5 Rectangular Parity SEC Code

We define the *parity* of bits $x_1, x_2, \ldots, x_n$ as $(x_1 + x_2 + \ldots + x_n)$, where the addition is performed modulo 2 (it's the same as taking the exclusive OR of the n bits). The parity is even when the sum is 0 (i.e., the number of ones is even), and odd otherwise.

Let parity(s) denote the parity of all the bits in the bit-string s. We'll use a dot, $\cdot$, to indicate the concatenation (sequential joining) of two messages or a message and a bit. For any message M (a sequence of one or more bits), let $w = M \cdot \text{parity}(M)$. You should be able to confirm that $\text{parity}(w) = 0$. This code, which adds a parity bit to each message, is also called the *even parity* code, because the number of ones in each codeword is even. Even parity lets us detect single errors because the set of codewords, $\{w\}$, each defined as $M \cdot \text{parity}(M)$, has a Hamming distance of 2.

If we transmit w when we want to send some message M, then the receiver can take the received word, r, and compute parity(r) to determine if a single error has occurred. The receiver's parity calculation returns 1 if an odd number of the bits in the received message has been corrupted. When the receiver's parity calculation returns a 1, we say there has been a *parity error*.

This section describes a simple approach to building an SEC code by constructing multiple parity bits, each over various subsets of the message bits, and then using the resulting parity errors (or non-errors) to help pinpoint which bit was corrupted.

d_{11}	d_{12}	d_{13}	d_{14}	p_row(1)
d_{21}	d_{22}	d_{23}	d_{24}	p_row(2)
p_col(1)	p_col(2)	p_col(3)	p_col(4)	

Figure 5-3: A 2×4 arrangement for an 8-bit message with row and column parity.

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

(a)

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

(b)

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

(c)

Figure 5-4: Example received 8-bit messages. Which, if any, have one error? Which, if any, have two?

Rectangular code construction: Suppose we want to send a k-bit message M. Shape the k bits into a rectangular array with r rows and c columns, i.e., $k = rc$. For example, if $k = 8$, the array could be 2×4 or 4×2 (or even 8×1 or 1×8, though those are a little less interesting). Label each data bit with a subscript giving its row and column: the first bit would be d_{11}, the last bit d_{rc}. See Figure 5-3.

Define p_row(i) to be the parity of all the bits in row i of the array and let R be all the row parity bits collected into a sequence:

$$R = [\text{p\_row}(1), \text{p\_row}(2), \ldots, \text{p\_row}(r)]$$

Similarly, define p_col(j) to be the parity of all the bits in column j of the array and let C be all the column parity bits collected into a sequence:

$$C = [\text{p\_col}(1), \text{p\_col}(2), \ldots, \text{p\_col}(c)]$$

Figure 5-3 shows what we have in mind when $k = 8$.

Let $w = M \cdot R \cdot C$, i.e., the transmitted codeword consists of the original message M, followed by the row parity bits R in row order, followed by the column parity bits C in column order. The length of w is $n = rc + r + c$. This code is linear because all the parity bits are linear functions of the message bits. The rate of the code is $rc/(rc + r + c)$.

We now prove that the rectangular parity code can correct all single-bit errors.

Proof of single-error correction property: This rectangular code is an SEC code for all values of r and c. We will show that it can correct all single bit errors by showing that its minimum Hamming distance is 3 (i.e., the Hamming distance between any two codewords is at least 3). Consider two different uncoded messages, M_i and M_j. There are three cases to discuss:

- If M_i and M_j differ by a single bit, then the row and column parity calculations involving that bit will result in different values. Thus, the corresponding codewords, w_i and w_j, will differ by three bits: the different data bit, the different row parity bit, and the different column parity bit. So in this case $\text{HD}(w_i, w_j) = 3$.
- If M_i and M_j differ by two bits, then either (1) the differing bits are in the same row, in which case the row parity calculation is unchanged but two column parity

calculations will differ, (2) the differing bits are in the same column, in which case the column parity calculation is unchanged but two row parity calculations will differ, or (3) the differing bits are in different rows and columns, in which case there will be two row and two column parity calculations that differ. So in this case $\text{HD}(w_i, w_j) \geq 4$.

- If M_i and M_j differ by three or more bits, then $\text{HD}(w_i, w_j) \geq 3$ because w_i and w_j contain M_i and M_j respectively.

Hence we can conclude that $\text{HD}(w_i, w_j) \geq 3$ and our simple "rectangular" code will be able to correct all single-bit errors.

Decoding the rectangular code: How can the receiver's decoder correctly deduce M from the received w, which may or may not have a single bit error? (If w has more than one error, then the decoder does not have to produce a correct answer.)

Upon receiving a possibly corrupted w, the receiver checks the parity for the rows and columns by computing the sum of the appropriate data bits *and* the corresponding parity bit (all arithmetic in $\mathbb{F}_2$). This sum will be 1 if there is a parity error. Then:

- If there are no parity errors, then there has not been a single error, so the receiver can use the data bits as-is for M. This situation is shown in Figure 5-4(a).

- If there is single row or column parity error, then the corresponding parity bit is in error. But the data bits are okay and can be used as-is for M. This situation is shown in Figure 5-4(c), which has a parity error only in the fourth column.

- If there is one row and one column parity error, then the data bit in that row and column has an error. The decoder repairs the error by flipping that data bit and then uses the repaired data bits for M. This situation is shown in Figure 5-4(b), where there are parity errors in the first row and fourth column indicating that d_{14} should be flipped to be a 0.

- Other combinations of row and column parity errors indicate that multiple errors have occurred. There's no "right" action the receiver can undertake because it doesn't have sufficient information to determine which bits are in error. A common approach is to use the data bits as-is for M. If they happen to be in error, that will be detected by the error detection code (mentioned near the beginning of this chapter).

This recipe will produce the most likely message, M, from the received codeword if there has been at most a single transmission error.

In the rectangular code the number of parity bits grows at least as fast as $\sqrt{k}$ (it should be easy to verify that the smallest number of parity bits occurs when the number of rows, r, and the number of columns, c, are equal). Given a fixed amount of communication "bandwidth" or resource, we're interested in devoting as much of it as possible to sending message bits, not parity bits. Are there other SEC codes that have better code rates than our simple rectangular code? A natural question to ask is: *how little redundancy can we get away with and still manage to correct errors?*

The Hamming code uses a clever construction that uses the intuition developed while answering the question mentioned above. We answer this question next.

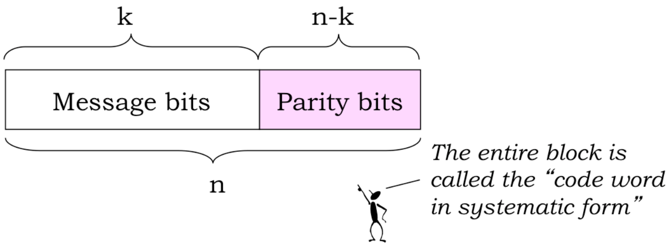

k
n-k
Message bits
Parity bits
n
The entire block is called the "code word in systematic form"

Figure 5-5: A codeword in systematic form for a block code. Any linear code can be transformed into an equivalent systematic code.

■ 5.6 How many parity bits are needed in an SEC code?

Let's think about what we're trying to accomplish with an SEC code: the correction of transmissions that have a single error. For a transmitted message of length n there are $n+1$ situations the receiver has to distinguish between: no errors and a single error in a specified position along the string of n received bits. Then, depending on the detected situation, the receiver can make, if necessary, the appropriate correction.

Our first observation, which we will state here without proof, is that any linear code can be transformed into an equivalent **systematic** code. A systematic code is one where every n-bit codeword can be represented as the original k-bit message followed by the $n-k$ parity bits (it actually doesn't matter how the original message bits and parity bits are interspersed). Figure 5-5 shows a codeword in systematic form.

So, given a systematic code, how many parity bits do we absolutely need? We need to choose n so that single error correction is possible. Since there are $n-k$ parity bits, each combination of these bits must represent *some* error condition that we must be able to correct (or infer that there were no errors). There are 2^{n-k} possible distinct parity bit combinations, which means that we can distinguish at most that many error conditions. We therefore arrive at the constraint

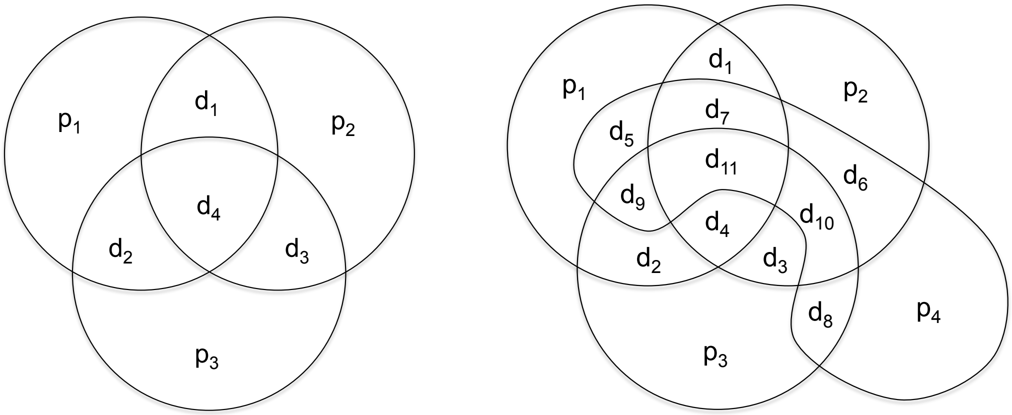
$$n+1 \leq 2^{n-k} \tag{5.6}$$

i.e., there have to be enough parity bits to distinguish all corrective actions that might need to be taken (including no action). Given k, we can determine $n-k$, the number of parity bits needed to satisfy this constraint. Taking the log (to base 2) of both sides, we can see that the number of parity bits **must** grow at least *logarithmically* with the number of message bits. Not all codes achieve this minimum (e.g., the rectangular code doesn't), but the Hamming code, which we describe next, does.

We also note that the reasoning here for an SEC code can be extended to determine a lower bound on the number of parity bits needed to correct $t > 1$ errors.

■ 5.7 Hamming Codes

Intuitively, it makes sense that for a code to be efficient, each parity bit should protect as many data bits as possible. By symmetry, we'd expect each parity bit to do the same amount of "work" in the sense that each parity bit would protect the same number of data



(a) (7,4) code

(b) (15,11) code

Figure 5-6: Venn diagrams of Hamming codes showing which data bits are protected by each parity bit.

bits. If some parity bit is shirking its duties, it's likely we'll need a larger number of parity bits in order to ensure that each possible single error will produce a unique combination of parity errors (it's the unique combinations that the receiver uses to deduce which bit, if any, had an error).

The class of Hamming single error correcting codes is noteworthy because they are particularly efficient in the use of parity bits: the number of parity bits used by Hamming codes grows logarithmically with the size of the codeword. Figure 5-6 shows two examples of the class: the (7,4) and (15,11) Hamming codes. The (7,4) Hamming code uses 3 parity bits to protect 4 data bits; 3 of the 4 data bits are involved in each parity computation. The (15,11) Hamming code uses 4 parity bits to protect 11 data bits, and 7 of the 11 data bits are used in each parity computation (these properties will become apparent when we discuss the logic behind the construction of the Hamming code in Section 5.7.1).

Looking at the diagrams, which show the data bits involved in each parity computation, you should convince yourself that each possible single error (don't forget errors in one of the parity bits!) results in a unique combination of parity errors. Let's work through the argument for the (7,4) Hamming code. Here are the parity-check computations performed by the receiver:

$$\begin{aligned}
E_1 &= (d_1 + d_2 + d_4 + p_1) \mod 2 \\
E_2 &= (d_1 + d_3 + d_4 + p_2) \mod 2 \\
E_3 &= (d_2 + d_3 + d_4 + p_3) \mod 2
\end{aligned}$$

where each E_i is called a *syndrome* bit because it helps the receiver diagnose the "illness" (errors) in the received data. For each combination of syndrome bits, we can look for the bits in each codeword that appear in *all* the E_i computations that produced 1; these bits are potential candidates for having an error since any of them could have caused the observed parity errors. Now eliminate from the candidates those bits that appear in *any* E_i computations that produced 0 since those calculations prove those bits didn't have errors.

We'll be left with either no bits (no errors occurred) or one bit (the bit with the single error).

For example, if $E_1 = 1$, $E_2 = 0$ and $E_3 = 1$, we notice that bits d_2 and d_4 both appear in the computations for E_1 and E_3. However, d_4 appears in the computation for E_2 and should be eliminated, leaving d_2 as the sole candidate as the bit with the error.

Another example: suppose $E_1 = 1$, $E_2 = 0$ and $E_3 = 0$. Any of the bits appearing in the computation for E_1 could have caused the observed parity error. Eliminating those that appear in the computations for E_2 and E_3, we're left with p_1, which must be the bit with the error.

Applying this reasoning to each possible combination of parity errors, we can make a table that shows the appropriate corrective action for each combination of the syndrome bits:

$E_3E_2E_1$	Corrective Action
000	no errors
001	p_1 has an error, flip to correct
010	p_2 has an error, flip to correct
011	d_1 has an error, flip to correct
100	p_3 has an error, flip to correct
101	d_2 has an error, flip to correct
110	d_3 has an error, flip to correct
111	d_4 has an error, flip to correct

■ 5.7.1 Is There a Logic to the Hamming Code Construction?

So far so good, but the allocation of data bits to parity-bit computations may seem rather arbitrary and it's not clear how to build the corrective action table except by inspection.

The cleverness of Hamming codes is revealed if we order the data and parity bits in a certain way and assign each bit an index, starting with 1:

index	1	2	3	4	5	6	7
binary index	001	010	011	100	101	110	111
(7,4) code	p_1	p_2	d_1	p_3	d_2	d_3	d_4

This table was constructed by first allocating the parity bits to indices that are powers of two (e.g., 1, 2, 4, . . .). Then the data bits are allocated to the so-far unassigned indicies, starting with the smallest index. It's easy to see how to extend this construction to any number of data bits, remembering to add additional parity bits at indices that are a power of two.

Allocating the data bits to parity computations is accomplished by looking at their respective indices in the table above. Note that we're talking about the *index* in the table, not the subscript of the bit. Specifically, d_i is included in the computation of p_j if (and only if) the logical AND of binary index(d_i) and binary index(p_j) is non-zero. Put another way, d_i is included in the computation of p_j if, and only if, index(p_j) contributes to index(d_i) when writing the latter as sums of powers of 2.

So the computation of p_1 (with an index of 1) includes all data bits with odd indices: d_1, d_2 and d_4. And the computation of p_2 (with an index of 2) includes d_1, d_3 and d_4. Finally, the computation of p_3 (with an index of 4) includes d_2, d_3 and d_4. You should verify that these calculations match the E_i equations given above.

If the parity/syndrome computations are constructed this way, it turns out that $E_3E_2E_1$, treated as a binary number, gives the index of the bit that should be corrected. For example, if $E_3E_2E_1 = 101$, then we should correct the message bit with index 5, i.e., d_2. This corrective action is exactly the one described in the earlier table we built by inspection.

The Hamming code's syndrome calculation and subsequent corrective action can be efficiently implemented using digital logic and so these codes are widely used in contexts where single error correction needs to be fast, e.g., correction of memory errors when fetching data from DRAM.

Acknowledgments

Many thanks to Katrina LaCurts for carefully reading these notes and making several useful comments.

Problems and Questions

1. Prove that the Hamming distance satisfies the triangle inequality. That is, show that $\text{HD}(x, y) + \text{HD}(y, z) \geq \text{HD}(x, z)$ for any three n-bit binary words.

2. Consider the following rectangular linear block code:

```
D0  D1  D2  D3  D4   | P0
D5  D6  D7  D8  D9   | P1
D10 D11 D12 D13 D14  | P2
-------------------------
P3  P4  P5  P6  P7   |
```

   Here, `D0`–`D14` are data bits, `P0`–`P2` are row parity bits and `P3`–`P7` are column parity bits. What are n, k, and d for this linear code?

3. Consider a rectangular parity code as described in Section 5.5. Ben Bitdiddle would like use this code at a variety of different code rates and experiment with them on some channel.

   (a) Is it possible to obtain a rate lower than 1/3 with this code? Explain your answer.

   (b) Suppose he is interested in code rates like 1/2, 2/3, 3/4, etc.; i.e., in general a rate of $\frac{\ell-1}{l}$, for some integer $\ell > 1$. Is it always possible to pick the parameters of the code (i.e, the block size and the number of rows and columns over which to construct the parity bits) so that any such code rate of the form $\frac{\ell-1}{l}$ is achievable? Explain your answer.

4. Two-Bit Communications (TBC), a slightly suspect network provider, uses the following linear block code over its channels. All arithmetic is in $\mathbb{F}_2$.

$$P_0 = D_0, P_1 = (D_0 + D_1), P_2 = D_1.$$

   (a) What are n and k for this code?

   (b) Suppose we want to perform syndrome decoding over the received bits. Write out the three syndrome equations for E_0, E_1, E_2.

   (c) For the eight possible syndrome values, determine what error can be detected (none, error in a particular data or parity bit, or multiple errors). Make your choice using maximum likelihood decoding, assuming a small bit error probability (i.e., the smallest number of errors that's consistent with the given syndrome).

   (d) Suppose that the the 5-bit blocks arrive at the receiver in the following order: D_0, D_1, P_0, P_1, P_2. If 11011 arrives, what will the TBC receiver report as the received data after error correction has been performed? Explain your answer.

   (e) TBC would like to improve the code rate while still maintaining single-bit error correction. Their engineer would like to reduce the number of parity bits by 1. Give the formulas for P_0 and P_1 that will accomplish this goal, or briefly explain why no such code is possible.

5. Pairwise Communications has developed a linear block code over $\mathbb{F}_2$ with three data and three parity bits, which it calls the *pairwise code*:

$$\begin{aligned} P_1 &= D_1 + D_2 \qquad \text{(Each } D_i \text{ is a data bit; each } P_i \text{ is a parity bit.)} \\ P_2 &= D_2 + D_3 \\ P_3 &= D_3 + D_1 \end{aligned}$$

   (a) Fill in the values of the following three attributes of this code:

   (i) Code rate = ____________

   (ii) Number of 1s in a minimum-weight non-zero codeword = ____

   (iii) Minimum Hamming distance of the code = ________

6. Consider the same "pairwise code" as in the previous problem. The receiver computes three syndrome bits from the (possibly corrupted) received data and parity bits: $E_1 = D_1 + D_2 + P_1, E_2 = D_2 + D_3 + P_2$, and $E_3 = D_3 + D_1 + P_3$. The receiver performs maximum likelihood decoding using the syndrome bits. For the combinations of syndrome bits in the table below, state what the maximum-likelihood decoder believes has occured: no errors, a single error in a specific bit (state which one), or multiple errors.

$E_3 E_2 E_1$	Error pattern [No errors / Error in bit ... (specify bit) / Multiple errors]
0 0 0	
0 0 1	
0 1 0	
0 1 1	
1 0 0	
1 0 1	
1 1 0	
1 1 1	

7. Alyssa P. Hacker extends the aforementioned pairwise code by adding an *overall parity bit*. That is, she computes $P_4 = \sum_{i=1}^{3}(D_i + P_i)$, and appends P_4 to each original codeword to produce the new set of codewords. What improvement in error correction or detection capabilities, if any, does Alyssa's extended code show over Pairwise's original code? Explain your answer.

8. For each of the sets of codewords below, determine whether the code is a linear block code over $\mathbb{F}_2$ or not. Also give the rate of each code.

   (a) {000,001,010,011}.

   (b) {000, 011, 110, 101}.

   (c) {111, 100, 001, 010}.

   (d) {00000, 01111, 10100, 11011}.

   (e) {00000}.

9. For any linear block code over $\mathbb{F}_2$ with minimum Hamming distance at least $2t + 1$ between codewords, show that:

$$2^{n-k} \geq 1 + \binom{n}{1} + \binom{n}{2} + \ldots \binom{n}{t}.$$

Hint: How many errors can such a code always correct?

10. For each (n, k, d) combination below, state whether a linear block code with those parameters exists or not. Please provide a brief explanation for each case: if such a code exists, give an example; if not, you may rely on a suitable necessary condition.

    (a) $(31, 26, 3)$: **Yes / No**

    (b) $(32, 27, 3)$: **Yes / No**

    (c) $(43, 42, 2)$: **Yes / No**

    (d) $(27, 18, 3)$: **Yes / No**

    (e) $(11, 5, 5)$: **Yes / No**

11. Using the Hamming code construction for the $(7, 4)$ code, construct the parity equations for the $(15, 11)$ code. How many equations does this code have? How many message bits contribute to each parity bit?

12. Prove Theorems 5.2 and 5.3. (Don't worry too much if you can't prove the latter; we will give the proof when we discuss convolutional codes in Lecture 8.)

13. The weight of a codeword in a linear block code over $\mathbb{F}_2$ is the number of 1's in the word. Show that any linear block code must either: (1) have only even weight codewords, or (2) have an equal number of even and odd weight codewords.
Hint: Proof by contradiction.

14. There are N people in a room, each wearing a hat colored red or blue, standing in a line in order of increasing height. Each person can see only the hats of the people in front, and does not know the color of his or her own hat. They play a game as a team, whose rules are simple. Each person gets to say one word: "red" or "blue". If the word they say correctly guesses the color of their hat, the team gets 1 point; if they guess wrong, 0 points. Before the game begins, they can get together to agree on a protocol (i.e., what word they will say under what conditions). Once they determine the protocol, they stop talking, form the line, and are given their hats at random.

    Can you develop a protocol that will maximize their score? What score does your protocol achieve?