MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: September 27, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

LECTURE 6
Coping with Bit Errors

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 an inevitable property of both commmunication channels and storage media.

The key idea we will apply to achieve reliable communication is *redundancy*: by replicating data to ensure that it can be reconstructed from some (correct) subset of the data received, we can improve the likelihood of fixing errors. This approach, called **error correction**, can work quite well when errors occur according to some random process. Error correction may not be able to correct all errors, however, so we will need a way to tell if any errors remain. That task is called **error detection**, which determines whether the data received after error correction is in fact the data that was sent. It will turn out that all the guarantees we can make are probabilistic, but we will be able to make our guarantees on reliabile data receptions with very high probability. Over a communication channel, the sender and receiver implement the error correction and detection procedures. 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**.

Our plan is as follows. First, we will define a model for the kinds of bit errors we're going to handle and revisit the previous lectures to see why errors occur. Then, we will discuss and analyze a simple redundancy scheme called a *replication code*, which will simply make c copies of any given bit. The replication code has a *code rate* of $1/c$—that is, for every useful bit we receive, we will end up encoding c total bits. The overhead of the replication code of rate c is $1 - 1/c$, which is rather high for the error correcting power of the code. We will then turn to the key ideas in that allow us to build powerful codes capable of correcting errors without such a high overhead (or, capable of correcting far more errors at a given code rate than the trivial replication code).

There are two big ideas that are used in essentially all channel codes: the first is the notion of **embedding**, where the messages one wishes to send are placed in a geometrically pleasing way into an embedding 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** (or more generally, linear functions) over the bits we wish to send to produce 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 an instance of the broad class of **graphical codes**.[1]

6.1 Bit error models

In the previous lectures, we developed a model for how channels behave using the idea of linear time-invariance and saw how noise corrupted the information being received at the other end of a channel. We characterized the output of the channel, Y, as the sum of two components

$$y[n] = y_{\text{nf}}[n] + \textit{noise}, \tag{6.1}$$

where $y_{\text{nf}}[n]$ is the sum of two terms. The first is the noise-free prediction of the channel output, which can be computed as the convolution of the channel's unit sample response with the input X, and the second is a random additive *noise* term. A good noise model for many real-world channels is Gaussian; such a model is has a special name: *additive white Gaussian noise*, or *AWGN*.[2] AWGN has mean 0 and is fully characterized by the variance, σ^2. The larger the variance, the more intense the noise.

One of the properties of AWGN is that there is *always* a non-zero probability that a voltage transmitted to represent a 0 will arrive at the receiver with enough noise that it will be interpreted as a 1, and vice versa. For such a channel, if we know the transmitter's signaling levels and receiver's digitizing threshold, we know (from the earlier lecture on noise) how to calculate the *probability of bit error* (BER). For the most part, we will assume a simple (but useful) model that follows from the properties of AWGN: a transmitted 0 bit may be digitized at the receiver as a 1 (after deconvolution) with some probability p (the BER), and a transmitted 1 may be digitized as a 0 at the receiver (after deconvolution) with the same probability p. Such a model is also called a *binary symmetric channel*, or *BSC*. The "symmetric" refers to the property that a 0 becomes a 1 and vice versa with the same probability, p.

BSC is perhaps the simplest 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 likelihood 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.

A BSC error model is characterized by one number, p. We can determine p empirically by noting that if we send N bits over the channel, the expected number of erroneously

[1] Graphical codes are sometimes also called "probabilistic codes" in the literature, for reasons we can't get into here.

[2] The "white" part of this term refers to the variance being the same over all the frequencies being used to communicate.

received bits is Np. Hence, by sending a known bit pattern and counting the fraction or erroneously received bits, we can estimate p. In practice, even when BSC is a reasonable error model, the range of p 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 p 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 at least 1000 bits long (and often bigger), such an error rate is too high.

But is a p 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" p of 10^{-12} means that the receiver will see one error every 10 seconds on average if the channel is continuously loaded. Unless we take some mechanisms to mitigate the situation, the applications using the channel may find this error rate (in time) too frequent. On the other hand, a p of 10^{-12} may be quite fine over a communication channel running at 10 Megabits/s, as long as there is some way to detect errors when they occur.

It is important to note that the error rate is not a fixed property of the channel: it depends on the strength of the signal relative to the noise level (aka the "signal to noise ratio", or SNR). Moreover, almost every communication channel trades-off between the raw transmission rate ("samples per bit" in this course) and the BER. As you increase the bit rate at which data is sent (say, by reducing the number of samples per bit), the BER will also increase, and then you need mechanisms to bring the BER down to acceptable values once more—and those mechanisms will reduce the achievable bit rate.

■ 6.2 The Simplest Code: Replication

In a replication code, each bit b is encoded as c copies of b, and the result is delivered. If we consider bit b to be the *message word*, then the corresponding *code word* is b^r (i.e., $bb...b$, c times). Clearly, (there are only two possible message words and corresponding code words). The beauty of the replication code is how absurdly simple it is.

But how well might it correct errors? To answer this question, we will write out the probability of decoding error for the BSC error model with the replication code. That is, if the channel corrupts each bit with probability p, what is the probability that the receiver decodes the received code word 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 code word, r, which is some c-bit combination of 0's and 1's, the decoder should produce the most likely message that could have caused r to be received. If the BSC error probability p, is small (in particular, smaller than $1/c$, which is almost always true for practical values of p and c), then the most likely option is the message word that has the most number of bits in common with r.

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 code word 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 the sender sending a message 0 is the same

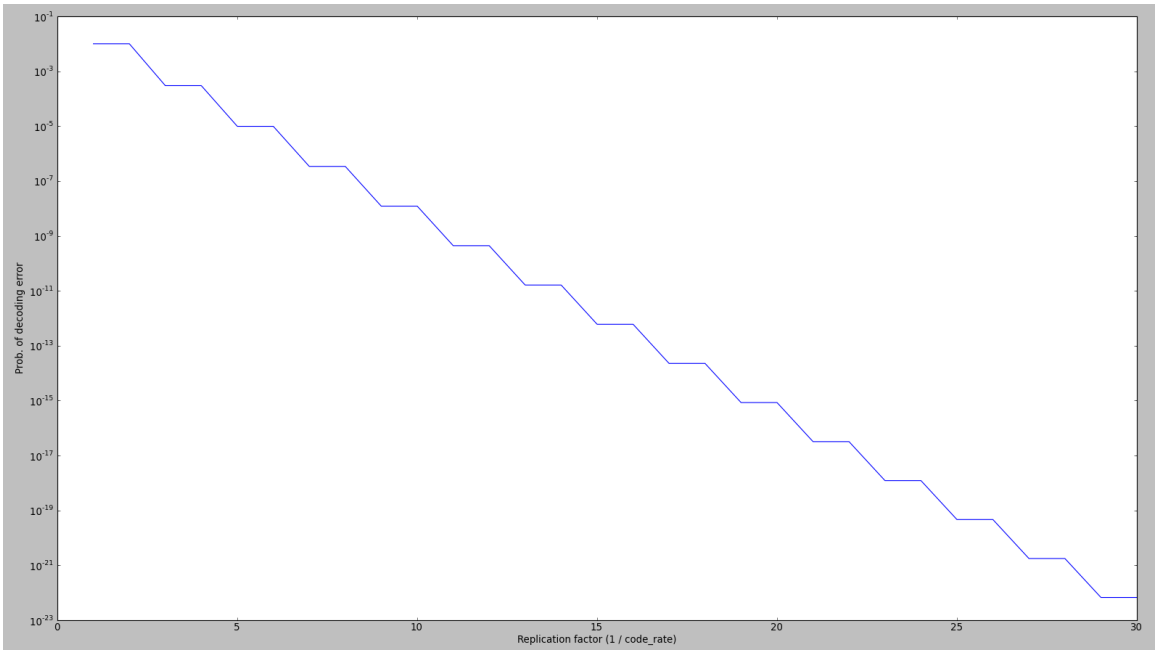


Figure 6-1: Probability of a decoding error with the replication code that replaces each bit b with c copies of b. The code rate is $1/c$.

as a 1.)

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

$$P(\text{decoding error}) = \begin{cases} \sum_{i=\lceil \frac{c}{2} \rceil}^{c} \binom{c}{i} p^i (1-p)^{c-i} & \text{if } c \text{ odd} \\ \sum_{i=\lceil \frac{c}{2} \rceil}^{c} \binom{c}{i} p^i (1-p)^{c-i} + (1/2)\binom{c}{c/2} p^{c/2}(1-p)^{c/2} & \text{if } c \text{ even} \end{cases} \tag{6.2}$$

When c 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 6-1 shows the probability of decoding error from Eq.(6.2) as a function of the code rate for the replication code. 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 $c = 2\ell$ as when $c = 2\ell - 1$. 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.

Given a chunk of data of size s bits, we can now calculate the probability that it will be in error after the error correction code has been applied. Each message word (1 bit in the case of the replication code) will be decoded incorrectly with probability q, where q is given by Eq.(6.2). The probability that the entire chunk of data will be decoded correctly is given by $(1-q)^s$, and the desired error probability is therefore equal to $1-(1-q)^s$. When $q << 1$, that error probability is approximately qs. This result should make intuitive sense.

Despite the exponential reduction in the probability of decoding error as c increases,

the replication code is extremely inefficient in terms of the overhead it incurs. 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 ideas: *embedding messages into spaces in a way that achieves structural separation* and *parity (linear) computations over the message bits*.

■ 6.3 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.

There are 2^n possible n-bit strings. Let's 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 M_1 with a unfortunately placed bit error and receiving M_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 code words.

What happens if we increase the Hamming distance between any two valid code words is at least 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

$$\text{HD}(w_i, w_j) \geq 2 \text{ for all } w_i, w_j \in \mathcal{S} \text{ where } i \neq j \tag{6.3}$$

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 6-2: 00 and 11 are valid code words, and the receptions 01 and 10 are surely erroneous.

It should be easy to see what happens as we use a code whose minimum Hamming distance between any two valid code words is D. We state the property formally:

Theorem 6.1 *A code with a minimum Hamming distance of D can detect* any *error pattern of $D-1$ errors or less. 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 code words 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 **??**, with $\mathcal{S} = \{00, 11\}$. Suppose the receiver receives 01. It 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.

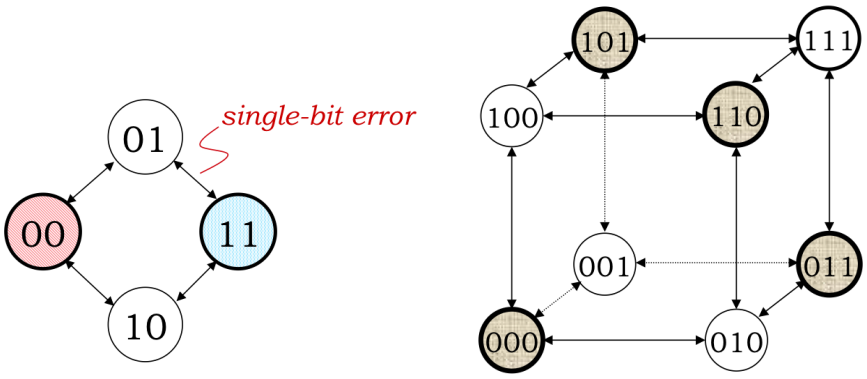


Figure 6-2: Code words separated by a Hamming distance of 2 can be used to detect single bit errors. The code words are shaded in each picture. The picture on the left is a (2,1) repetition code, which maps 1-bit messages to 2-bit code words. The code on the right is a (3,2) code, which maps 2-bit messages to 3-bit code words.

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:

$$\text{HD}(w_i, w_j) \geq 3 \text{ for all } w_i, w_j \in \mathcal{S} \text{ where } i \neq j \tag{6.4}$$

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, $\mathcal{E}_{00} = \{01, 10\}$. Note that HD($w_i$, an element of $\mathcal{E}_{w_i}$) $= 1$.

With a minimum Hamming distance of 3 between the valid code words, 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 HD$(w_i, w_k) = 1$ and HD$(w_j, w_k) = 1$, which implies that w_i and w_j differ in at most two bits and consequently HD$(w_i, w_j) \leq 2$. That contradicts our specification that their minimum Hamming distance be 3. So the $\mathcal{E}_{w_i}$ don't intersect.

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 receiver receives 101. It can tell there's been a single error since $101 \notin \mathcal{S}$. Moreover it can deduce that the original message was 111 since $101 \in \mathcal{E}_{111}$.

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

Theorem 6.2 *The Hamming distance between n-bit words satisfies the triangle inequality. That is, $HD(x, y) + HD(y, z) \geq HD(x, z)$.*

Theorem 6.3 *For a BSC error model with small bit error probability, the maximum likely decoding strategy is to map any received word to the valid code word with smallest Hamming distance from the received one (ties may be broken arbitrarily).*

Theorem 6.4 *A code with a minimum Hamming distance of D can correct* any *error pattern of*

$\lfloor \frac{D-1}{2} \rfloor$ errors or less. Moreover, there is at least one error pattern with $\lfloor \frac{D-1}{2} \rfloor + 1$ errors that cannot be corrected reliably.

Equation (6.4) 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 code word to the received one, and then use the known mapping between each distinct message and the code word to produce the message. That check may be exponential in the number of message bits we would like to send, but would be reasonable 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. Quick, what should the members of $\mathcal{S}$ be?

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:

0000000	1100001	1100110	0000111
0101010	1001011	1001100	0101101
1010010	0110011	0110100	1010101
1111000	0011001	0011110	1111111

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: Hamming codes and rectangular parity codes. We also note that the replication code discussed in Section 6.2 is an example of a linear block code.

In later lectures, we will study **convolutional codes**, a sub-class of graphical codes.

■ 6.4 Linear Block Codes

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 code words, 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 is broken up into k-bit blocks, which are then expanded to produce n-bit code words that are sent.

Such codes are also called (n, k) codes, where k message bits are combined to produce n code bits (so each code word 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 overhead incurred by the code.

A linear block code restricts the algebraic operations to *linear functions* over the data bits. By linear, we mean that any given bit in a valid code word 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, the replication code we discussed in Section 6.2 is an example of a linear block code with parameters $(c, 1, c-1)$.

To develop a little bit of intuition about the linear operations, let's start with a "hat" puzzle, which might at first seem unrelated to coding.

> 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 any 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 think of a protocol that will maximize their score? What score does your protocol achieve?*

A little bit of thought will show that there is a way to use the concept of *parity* to enable $N-1$ of the people to correctly decode the colors of their hats. In general, "parity" of a set of bits $x_1, x_2, \ldots x_n$ is simply equal to $(x_1 + x_2 + \ldots + x_n)$, where the addition is performed modulo 2 (it's the same as taking the exclusive OR of the bits). Even parity occurs when the sum is 0 (i.e., the number of 1's is even), while odd parity is when the sum is 1.

Arithmetic modulo 2 has a special name: it's a *Galois Field* of order 2, also denoted $\mathbb{F}_2$. (Later we will talk about Galois Fields of order 2^q when we discuss Reed Solomon codes.) A field must define rules for addition and multiplication. Addition in $\mathcal{O}_2$ is as stated above: $0+0 = 1+1 = 0, 1+0 = 0+1 = 0$. Multiplication is as usual: $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0, 1 \cdot 1 = 1$.

A linear block code simply produces code words by calculating them from the message bits; when it uses the rules mentioned above, we have a linear code over $\mathbb{F}_2$. A linear code is characterized by the following rule:

Definition 6.1 *A block code is said to be linear if, and only if, the sum of any two code words is another code word.*

For example, the code defined by code words 000, 101, 011 is *not* a linear code, because 101 + 011 = 110 is not a code word. But if we add 110 to the set, the result is a linear code because the sum of any two code words is another code word. The code (000, 101, 011, 110) has Hamming distance 2 and can be used to detect all 1-bit errors. 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 code word. In fact, that's a general property of all linear block codes, which we state formally below.

Theorem 6.5 *Define the weight of a code word 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 code word with*

the smallest weight.

To see why, use the property that the sum of any two code words must also be a code word, and that the Hamming distance between any two code words is equal to the weight of their sum (i.e., weight$(u+v) = \text{HD}(u,v)$).

The rest of this section shows how to construct linear block codes over $\mathbb{F}_2$. We will focus on correcting single-bit errors. We will show two ways of building the set $\mathcal{S}$ of transmission messages such that the size of $\mathcal{S}$ will allow us to send messages of some specific length, and to describe how the receiver can perform error correction on the (possibly corrupted) received messages. These are both examples of *single error correcting* (SEC) codes.

We will start with a simple "rectangular" code, then discuss the cleverer and more efficient Hamming code.

■ 6.4.1 A simple "rectangular" SECC

Let parity(w) equal the sum over $\mathbb{F}_2$ of all the bits in word w. Borrowing from Python's notation, we'll use $+$ to indicate the concatenation (sequential joining) of two messages or a message and a bit. Let $w' = w + \text{parity}(w)$. You should be able to confirm that parity$(w') = 0$.

What's useful about parity is that it lets us detect single errors. If we transmit w' when we want to send some message w, then the receiver can compute parity$(w'_{received})$ 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*.

How does being able to *detect* a single error help us *correct* the error? The trick is to use multiple parity bits and use the resulting parity errors (or non-errors) to help triangulate which bit was corrupted. If we're careful in designing our parity calculations, each possible single error will be reflected in a different combination of parity errors. This will allow the receiver to deduce from the parity errors which bit needs repair.

Here's a simple approach to building a SEC code. 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 1×8, 2×4, 4×2, or 8×1. Label each data bit with subscript giving its row and column: the first bit would be d_{11}, the last bit d_{rc}. See Figure 6-3.

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

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

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

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

Figure 6-3 shows what we have in mind when $k = 8$.

Let $M_T = M + R + C$, i.e., the transmitted message will consist of the original message M, followed by the row parity bits R in row order, followed by the column parity bits C

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

Figure 6-3: 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 6-4: Example received 8-bit messages. Which have an error?

in column order. The length of M_T is $k + r + c$.

Can the receiver correctly deduce M from the received M_T, which may or may not have a single bit error?

Upon receiving a possibly corrupted M_T, the receiver checks the parity for the rows and columns by computing the modulo-2 sum of the appropriate data bits *and* the corresponding parity bit. This sum will be zero if no errors are detected for a particular calculation and 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 is the situation shown in Figure 6-4(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 is the situation shown in Figure 6-4(c) which only has a parity error 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 receiver repair the error by flipping that data bit and then use the repaired data bits for M. This is the situation shown in Figure 6-4(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 multiple errors have occurred. There's no "right" action the receiver can undertake since 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 when validating the packet's check bits and the packet dropped at that point.

This recipe will deduce M from M_T if there has been at most a single transmission error in M_T.

Harking back to the previous discussion on Hamming distance, if this is an SEC code, we should be able to show that the minimum Hamming distance between different M_T is at least 3. Consider two different 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 M_{Ti} and M_{Tj} will differ by

three bits: the different data bit, the different row parity bit, and the different column parity bit. So in this case HD(M_{Ti}, M_{Tj}) = 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 HD(M_{Ti}, M_{Tj}) ≥ 4.

- if M_i and M_j differ by three or more bits, then in this case HD(M_{Ti}, M_{Tj}) ≥ 3 since M_{Ti} and M_{Tj} contain M_i and M_j respectively.

Hence we can conclude that HD(M_{Ti}, M_{Tj}) ≥ 3 and our simple "rectangular" code is shown to be capable of single error correction.

In this code the number of parity bits grows as $\sqrt{k}$ (when the number of rows and columns are equal). Given a fixed amount of communication bandwidth, 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 applies the answer to the above question, which we will now answer.

■ 6.4.2 How many parity bits are needed in a SEC code?

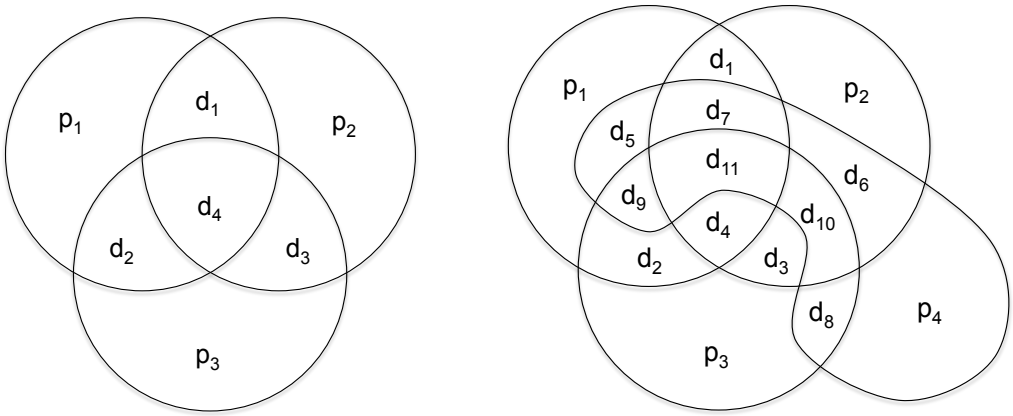
Let's think about what we're trying to accomplish with a SEC code: the goal is to correct transmissions with at most a single error. For a transmitted message of length n there are $n + 1$ situations the receiver has to distinguish between: no errors and a single error in any of the n received bits. 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 a **systematic** code. A systematic code is one where every n-bit code word 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).

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

$$n + 1 \leq 2^{n-k} \tag{6.5}$$

i.e., there have to be enough parity bits to distinguish all corrective actions that might need to be taken. Given k, we can determine the number of parity bits $(n - k)$ needed to satisfy this constraint. Taking the log base 2 of both sides, we can see that the number of parity bits grows roughly *logarithmically* with the number of message bits.



(a) (7,4) code

(b) (15,11) code

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

■ 6.4.3 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 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 a single error).

The class of Hamming single error correcting codes is widely used since they are particularly efficient in the use of parity bits: the number of parity bits used by Hamming codes grows as $\log k$.

Figure 6-5 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; 7 of the 11 data bits are used in each parity computation.

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 the E_i are called the *syndrome* bits since they help receiver diagnose the "illness" (errors) in the received message. For each combination of syndrome bits, we can look for the bits in M_T that appear in *all* 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 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

So far so good, but the allocation of data bits to parity-bit computations probably seems 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. Using this construction we'll end up with $O(\log k)$ parity bits for k data bits.

Allocating the data bits to parity computations is accomplished by looking at their respective indices in the table above. (Note 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 the logical AND of index(d_i) and index(p_j) is non-zero. 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. This matches 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 is exactly the corrective action described in the earlier table we built by inspection.

The Hamming SECC 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.

■ Problems and Questions

These questions are to help you improve your understanding of the concepts discussed in this lecture. The ones marked ***PSet*** are in the online problem set.

1. Show 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 numbers in $\mathbb{F}_2$.

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. 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.

4. For any linear block code over $\mathbb{F}_2$ with minimum Hamming distance at least $2t+1$ between code words, 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?

5. The weight of a code word 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 code words, or (2) have an equal number of even and odd weight code words.