

## Lecture 7: Error-Correcting Codes and Latin Squares, Part 1/2

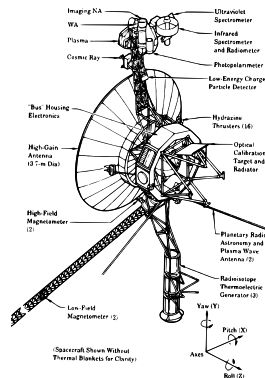
Week 3

Mathcamp 2012

## 1 Error-Correcting Codes

Over the next two lectures, we're going to study the following problem:

**Problem.** Suppose that you are the Voyager 1 probe. You are currently on the outer limits of the solar system, and about to leave the solar system forever! Consequently, you want to call your parents. However, you are currently separated from your parents by the vast interstellar void of SPAAAAAAAAACE.



SPAAAAACE



The vast interstellar void of space has an annoying habit of occasionally containing stray electromagnetic waves that interfere with your communications back home; when you send a signal back home (in binary, naturally), occasionally one of your 1's will be switched into a 0, or vice-versa. Assume that no more than one out of three consecutive bits in any message you send will be scrambled.

How can you call home?

One solution to this problem you might come up with is to simply “build redundancy” into the signals you send home, by sending (say) six 1's every time you want to send a 1, and six 0's every time you want to send a 0. For example, to send the message “0101,” you'd send

000000111111000000111111.

Then, even if some of the bits are flipped, your parents back on earth could still decode your message. In particular, if at most one bit out of any consecutive three is flipped, each of your all-0's blocks will have at most two 1's in them after errors are introduced, while each of your all-1's blocks will have at most two 0's in them after errors. In either case, we would never confuse one of these blocks with the other: if your parents received the signal

010010001111100001101011,

they'd break it up into the four blocks

010010001111100001101011,

and "correct" the errors to

000000111111000000111111,

which is unambiguously the signal 0101.

This code can correct for the presence of one error out of any three consecutive blocks, but no better (i.e. if we could have more than two errors in a block of six, we might have three errors in a string of six: in this case it would be impossible to tell what our string was intended to be. For example, the string 000111 could have resulted from three errors on the signal 000000, or three errors on the signal 111111.) It can accomplish this at the cost of sending  $6k$  bits whenever it wants to transmit  $k$  bits of information.

Can we do better? In specific, can we make a code that is more efficient (i.e. needs less bits to transmit the same information,) or can correct for more errors? With a little thought, it's easy to improve our code above: if we instead simply replace each 0 with just 000 and each 1 with 111, our code can still correct for the presence of at most one error in any three consecutive blocks (for example, 101 is unambiguously the result of one error to 111,) and now needs to send just  $3k$  bits to transmit  $k$  bits of information.

There are more interesting codes than just these repetition codes: consider for example the codeword table

word	signal to transmit
000	000000
100	100011
010	010101
001	001110
011	011011
101	101101
110	110110
111	111000

In this code, we encode messages by breaking them into groups of three, and then replacing each string of three with the corresponding group of six. For example, the message "010 101 111" would become

010101101101111000.

In this code, every word in the table above differs from any other word in at least three spots (check this!) Therefore, if we have at most 1 error in any six consecutive bits, we would never confuse a word here with any other word: changing at most one bit in any block of six would still make it completely unambiguous what word we started with.

Therefore, if we sent the string that we described above, and people on Earth received

010111101111110000,

they would first break it into groups of six

010111101111110000,

and then look through our codeword table for what words these strings of six could possibly be, if at most one error in every six consecutive bits could occur:

010101101101111000.

This then decodes to “010 101 111,” the message we sent.

This code can correct for at most one error in any six consecutive bits (worse than our earlier code,) but does so much more efficiently: it only needs to send  $2k$  bits to transmit a signal with  $k$  bits of information in it.

So: suppose we know ahead of time the maximum number of errors in any consecutive string of symbols. What is the most efficient code we can make to transmit our signals?

At this time, it makes sense to try to formalize these notions of “maximum number of errors” and “efficiency.” Here are a series of definitions, that formalize the words and ideas we’ve been playing with in this talk:

**Definition.** A  $q$ -ary code  $C$  of length  $n$  is a collection  $C$  of words of length  $n$ , written in base  $q$ . In other words,  $C$  is just a subset of  $(\mathbb{Z}/q\mathbb{Z})^n$ .

**Example.** The “repeat three times” code we described earlier is a 2-ary code of length 3, consisting of the two elements  $\{(000), (111)\}$ . We used it to encode a language with two symbols, specifically 0 and 1.

The second code we made is a 2-ary code of length 6, consisting of the 8 elements we wrote down in our table.

**Definition.** Given a  $q$ -ary code  $C$  of length  $n$ , we define its **information rate** as the quantity

$$\frac{\log_q(\# \text{ of elements in } C)}{n}$$

This, roughly speaking, captures the idea of how “efficient” a code is.

**Example.** The “repeat three times” code we described earlier contains two codewords of length 3; therefore, its information rate is

$$\frac{\log_2(2)}{3} = \frac{1}{3}.$$

This captures the idea that this code needed to transmit three bits to send any one bit of information.

Similarly, the second code we made contains 8 codewords of length six, and therefore has information rate

$$\frac{\log_2(8)}{6} = \frac{3}{6} = \frac{1}{2}.$$

Again, this captures the idea that this code needed to transmit two bits in order to send any one bit of information.

**Definition.** The **Hamming distance**  $d_H(\mathbf{x}, \mathbf{y})$  between any two elements  $\mathbf{x}, \mathbf{y}$  of  $(\mathbb{Z}/q\mathbb{Z})^n$  is simply the number of places where these two elements disagree.

Given a code  $C$ , we say that the minimum distance of  $C$ ,  $d(C)$ , is the smallest possible value of  $d_H(\mathbf{x}, \mathbf{y})$  taken over all distinct  $\mathbf{x}, \mathbf{y}$  within the code.

**Example.** The Hamming distance between the two words

12213, 13211

is 2, because they disagree in precisely two places. Similarly, the Hamming distance between the two words

*TOMATO, POTATO*

is 2, because these two words again disagree in precisely two places.

The “repeat three times” code from earlier has minimum distance 3, because the Hamming distance between 000 and 111 is 3.

Similarly, the second code we described from earlier has minimum distance 3, because every two words in our list disagreed in at least 3 places.

The following theorem explains why we care about this concept of distance:

**Theorem.** A code  $C$  can detect up to  $s$  errors in any received codeword as long as  $d(C) \geq s + 1$ . Similarly, a code  $C$  can correct up to  $t$  errors in any received codeword to the correct codeword as long as  $d(C) \geq 2t + 1$ .

**Proof.** If  $d(C) \geq s + 1$ , then making  $s$  changes to any codeword cannot change it into any other codeword, as every pair of codewords differ in at least  $s + 1$  places. Therefore, our code will detect an error as long as at most  $s$  changes occur in any codeword.

Similarly, if  $d(C) \geq 2t + 1$ , then changing  $t$  entries in any codeword still means that it differs from any other codeword in at least  $t + 1$  many places; therefore, the codeword we started from is completely unambiguous, and we can correct these errors.

**Example.** Using this theorem, we can see that both of our codewords can correct at most one error in any codeword, because their Hamming distances were both three.

Now that we’ve made this formal, we can now state our question rigorously:

**Problem.** Suppose that you are given a base  $q$ , a length  $n$  for your codewords, and a minimum distance  $d$  that you want your codewords to be from each other (because you want to be able to correct up to  $\lceil (d - 1)/2 \rceil$  many errors in any codeword, for example.)

What is the maximum size of  $C$  — in other words, what is the maximum information rate you can get a code to have with these parameters?

Amazingly enough, this problem is wide open for tons of values! We really know very little about these maximum values: for example, when  $n = 10, q = 2, d = 3$  this question is still open. (I think we know it’s between 72 and 79?)

## 2 Why Mention This in a Latin Square Class?

Fun fact:

**Theorem 1** *For any value of  $q$ , the largest number of elements in any  $q$ -ary code  $C$  of length 4, distance 3 is  $q^2$ . This is attainable if and only if there are a pair of mutually orthogonal Latin squares of order  $q$ .*

Proofs: tomorrow!