

EE 334
Error Detection and Correction

One of the nice things about transmitting binary information is that extra bits can be added to the data stream for the purpose of detecting errors and, in some cases, correcting errors that were introduced during transmission. In this lecture, we'll cover one method for detecting errors and two methods that enable both detection and correction.

Parity Checking

Perhaps the easiest of the error detection schemes is to force and check the parity of a series of bits. **Parity** refers to the number of ones (bits of value = 1) in a section of a data stream. If the number of ones is odd, the parity is called **odd parity**. If the number of ones is even, then it's **even parity**. Parity can be applied to any number of bits. For instance, in an earlier set of notes we saw that a parity bit was added to a 7-bit ASCII character (the letter "M"). To transmit the "M" actually required sending 8 bits: the 7-bit ASCII code for "M" plus the one parity bit.

1. Encode the following bit sequences for transmission using odd parity. In each case you must add an extra bit to the end of the sequence to either force or maintain an odd number of ones.

1001101 0001000 1110011 0110000

2. Encode the following bit sequences for transmission using even parity.

1001101 0001000 1110011 0110000

3. You received the following bit sequences that were sent with odd parity. Indicate whether or not an error occurred during transmission.

1110110001111000 0000111100110101

4. In the previous problem, do you know where the error is in the sequence that had an error? Why or why not?
5. In problem #3, are you really sure that the sequence that passed the parity check doesn't have an error? IOW, is there a way that several errors could trick you into thinking there wasn't an error?

As with most communication methods, the sender and receiver have to agree on a set of rules so that the message can be encoded and transmitted and then decoded once received.

6. For parity checking, what things need to be agreed upon by the sender and receiver?

EE 334
Error Detection and Correction

7. As the receiver, what do you do if you detect an error in what was sent to you?

Parity checking is extremely easy to use. However, it only allows the detection of errors. Let's look now at two systems that allow you to detect and correct single bit errors.

Longitudinal Redundancy Checking (LRC)

LRC enables the receiver to detect and correct an error in a single bit of transmitted data. LRC does this by applying several parity checks to a sequence of bits. Let's learn by doing.

8. We want to transmit the following data to another computer: 1001000110001110. The two computers are using LRC. They have agreed to arrange the data into a 4x4 block as shown below. **The data block will be encoded adding parity bits to all rows and columns, even the all-parity columns.** Encode the block for transmission using **even parity**.

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

9. The block was correctly encoded and then sent. However, during transmission, an error occurred in one of the bits. Apply the parity checks to the bits received (already in grid format) to find the error.

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

10. The following **odd parity** LRC block was received by your computer. Was there an error in transmission? If so where? How do you correct the error?

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

Hamming Codes

Pioneering computer engineer Richard Hamming invented this code system way back in 1950. Hamming Codes have been very popular over the years because of their simplicity and ability to scale with the number of data bits they guard (see the section on transmission efficiency later).

Note: The Hamming method described in the course supplementary material is a degenerate version. Only use the method you learn here as it is more robust (and complete!).

Encoding Data Using Hamming Codes

To encode data for transmission using a Hamming Code, we must insert extra bits into the data at certain points. The process is fairly simple. We'll learn by example using our friend ASCII "M", 1001101. Work in the space below.

11. First, going from right-to-left, number some spots where we'll place our bits. You'll need 11 in this case.
12. Mark the spots with a value that's a power of 2 with an "h" (for "Hamming"). These spots are where we'll put our extra bits that guard the data bits.
13. Mark the remaining spots with a "d" (for "data").
14. Copy, in order, the data bits of the letter "M" into the spots labeled "d".
15. Now, in a column format, write down the numbers of the positions where the data bits have a value of 1 (one).
16. Next to the numbers, translate those numbers into binary. Use four digits since you need numbers as high as 11. It'll help tremendously to line up each digit of the binary numbers into columns. Note that you also have four Hamming bit positions. This is not a coincidence.
17. Going column by column, XOR the binary digits together. This is called a **bit-wise XOR**. Remember, when XORing multiple bits at the same time, if there are an odd number of 1s, the answer will be 1. Otherwise it will be 0.
18. Copy, in order, the results of the XOR operation into the spots labeled "h".

Decoding a Hamming Encoded Message

Decoding a Hamming encoded message is pretty much the reverse of the encoding process. The main difference is that we don't need to generate the Hamming bits. Instead we just check the message. Let's check the letter "M" that we just encoded.

19. In the workspace above, copy the 11 bits of the encoded letter "M".
20. Going from right-to-left, number the bits.
21. Label the positions as being for Hamming bits or data bits. (This is not really needed, but for learning and the simple cases we'll consider, it is useful.)
22. In a column format, write down the position numbers that have a bit value of 1.
23. Convert those position numbers into 4-bit binary numbers.
24. Do a bit-wise XOR on all of the binary numbers.
25. If the result of the XOR is all zeros, then there is not an error. If the result of the XOR is non-zero, then the result tells you the bit position that has an error – the error can be a data bit or a Hamming bit!

Let's do a couple of examples.

26. Hamming encode this 12-bit sequence: 0010 0011 1101 1000. Remember: every position that's a power of 2 is reserved for Hamming bits.

EE 334
Error Detection and Correction

27. Decode the following Hamming encoded transmission. If there is an error, tell where it is and write down the corrected transmission. Write down the data bits contained in the encoded message (once corrected).

101001001101

28. Are any of the detection and correction methods you've learned about fool-proof? Why or why not?

29. When are the methods you've learned about most useful?

Parity, LRC, and Hamming Codes are not detection and correction methods out there.

A popular error detection (can't correct) method is called **Cyclical Redundancy Checking (CRC)**. You might have downloaded files that were CRC encoded. CRC can be applied to a bit stream of any length. It produces a short code (usually 32 or 64 bits) that based on the data. Both the data and the code are transmitted. If the data or the code is corrupted during transmission, it is highly likely (though not guaranteed) that reapplying the CRC encoding to the data will not produce the same CRC code. If the codes don't match, then either the received data or the code was corrupted.

Reed-Solomon codes can detect and correct multiple errors (up to a point, of course). They are used to protect data on CDs, DVD, and Blu-Ray discs and for DSL and WiMAX transmissions. Ever wonder how your CD player can play music correctly even though the disc is scratched? This is why.

Transmission Efficiency

Adding extra bits to a transmission to enable error checking is a useful thing no doubt. However, adding extra bits adds to the number of bits that need to be transmitted:

$$\text{data bits} + \text{error checking bits} = \text{total bits transmitted}$$

Once the message is received, the error checking bits are eventually discarded (they've served their purpose) and only the data bits are kept. So, while useful, the error checking bits are a necessary but extra cost of doing business. In the case of error checking, we are concerned with the **transmission efficiency**:

$$\text{Efficiency} = \frac{\text{\# data bits}}{\text{\# data bits} + \text{\# error checking bits}} = \frac{\text{\# data bits}}{\text{total bits}}$$

30. What is the efficiency of transmitting a 7-bit ASCII character using even parity encoding?

31. What is the efficiency of transmitting 64 bits of data (an 8x8 block of data) under the LRC method? You need to account for the parity bits that will be added to encode the data.

32. As with all things costly, we'd like to keep the cost as low as possible by being as efficient as possible. How do we make data transmission with error checking as efficient as possible?

33. What's the down side to high efficiency?