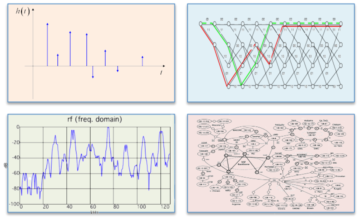


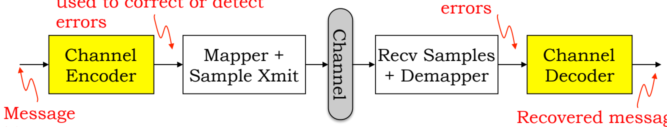
INTRODUCTION TO EECS II
DIGITAL COMMUNICATION SYSTEMS

6.02 Fall 2011
Lecture #5: Error Correction Codes - 1

- Channel coding: applying redundancy to correct errors
- Embeddings and Hamming distance: structural separation
- Parity equations & linear functions
- Linear (n,k) block codes & rectangular parity code

Channel Coding

Our plan to deal with bit errors:



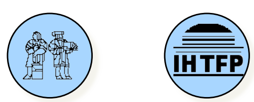
Channel coding is about *error correction* (and *error detection).*

We will design codes to correct commonly occurring errors, e.g., error bursts of bounded length.

We will also design codes to reduce the effective bit error rate, i.e., the probability of a decoding error.

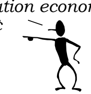
Error Model: Binary Symmetric Channel

Suppose we wanted to reliably transmit the result of a single coin flip:

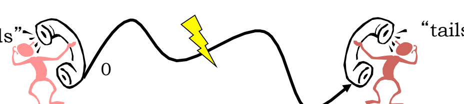


Heads: "0" Tails: "1"

This is a prototype of the "bit" coin for the new information economy. Value = 12.5¢



Suppose that during transmission a "0" is turned into a "1" or a "1" is turned into a "0" with probability p_e. This is a *binary symmetric channel* (BSC).



Key Idea: Redundancy

If bit errors are independent, then probability of multiple bits *all* being wrong reduces rapidly.

P(k bits all wrong) = p^k

If I replicate each bit *twice* can I improve error correction? Can I detect errors if they occur?

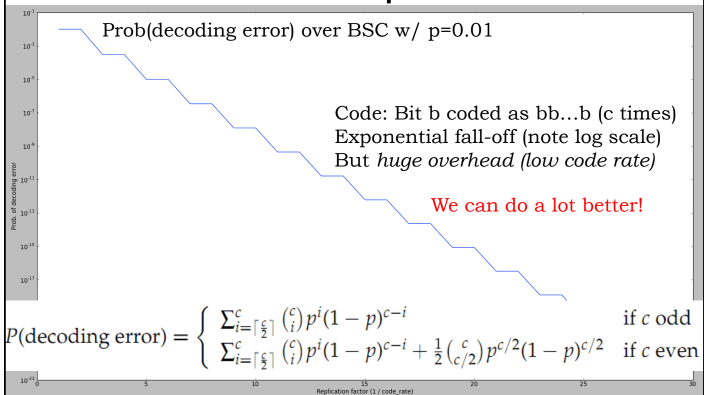
If I send the same bit three times, what is the probability of a bit error in the decoding?

Decoding rule: majority vote!

Generalize to sending c copies of each bit
→ The simplest error correction code, the *replication code*

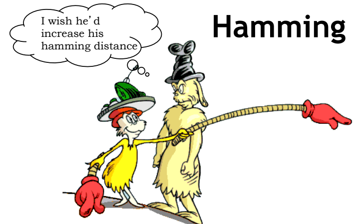
Message bit b → Codeword $bbb...b$ (c times)
Decoder: Count #0's and #1's, pick majority

Performance of Replication Code



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

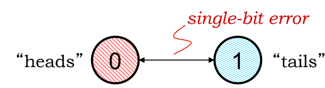
Hamming Distance



The number of bit positions in which the corresponding bits of two encodings of the same length are different

The Hamming Distance (HD) between a valid binary code word and the same code word with *e* errors is *e*.

The problem with no coding is that the two valid code words ("0" and "1") also have a Hamming distance of 1. So a single-bit error changes a valid code word into another valid code word…

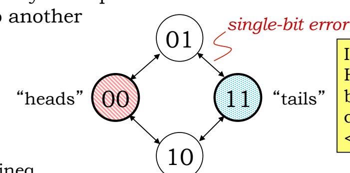


What is the Hamming Distance of the replication code?

Embedding for Structural Separation

Encode so that the codewords are "far enough" from each other

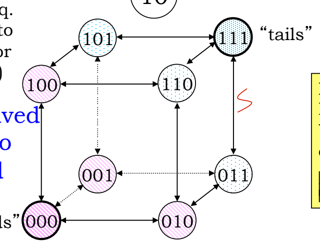
Likely error patterns shouldn't transform one codeword to another



If D is the minimum Hamming distance between codewords, we can detect **all** patterns of <= (D-1) bit errors

Use triangle ineq. property of HD to show single error correction (SEC)

Decode received codeword *r* to nearest valid codeword



If D is the minimum Hamming distance between codewords, we can correct **all** patterns of $\left\lfloor \frac{D-1}{2} \right\rfloor$ or fewer bit errors

Gaining Some Insight: Parity Calculations

We can add single-bit error detection to any length code word by adding a *parity bit* chosen to guarantee the Hamming distance between any two valid code words is at least 2.

Parity: addition in GF(2): 0+0=0, 1+0=0+1=1, 1+1=0
multiplication: 0*0=0*1=1*0 =0, 1*1=1

GF(2) arithmetic: Can count by summing the bits in the word modulo 2 (equivalent to XOR'ing the bits together).

A Simple Code: Parity Check

- Add a parity bit to message of length k to make the total number of "1" bits even (aka "even parity").
- If the number of "1"s in the received word is *odd*, there there has been an error.

0 1 1 0 0 1 0 1 0 0 1 1 → original word with parity bit
0 1 1 0 0 0 0 1 0 0 1 1 → single-bit error (detected) bit
0 1 1 0 0 0 1 1 0 0 1 1 → 2-bit error (not detected) bit

- Hamming distance of parity check code is 2
  – Can detect all single-bit errors
  – In fact, can detect all odd number of errors
  – But cannot detect even number of errors
  – And cannot correct any errors

6.02 Fall 2011 Lecture 5, Slide #10

Linear Block Codes

Can we extend the parity check idea and add more parity bits by combining different message bits?

Block code: *k* message bits encoded to n code bits
I.e., each of 2^k messages encoded into a unique n-bit combination via a *linear transformation.*
Set of parity equations (in GF(2)) represents code.

Key property: Sum of any two codewords is *also* a codeword → necessary and sufficient for code to be linear.

(n,k) code has rate k/n.
Sometime written as (n,k,d), where d is the Hamming Distance of the code.

6.02 Fall 2011 Lecture 5, Slide #11

Examples: What are n, k, d here?

{111, 000} (3,1,3). Rate= 1/3.

{0000, 1100, 0011, 1111} (4,2,2). Rate = ½.

{00000} {5,0,_). Rate = 0!

{1111, 0000, 0001} → Not linear codes!

{1111, 0000, 0010, 1100} ↗

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

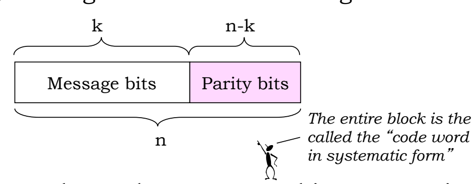
(7,4,3) code. Rate = 4/7.

The HD of a linear code is the number of "1"s in the non-zero codeword with the smallest # of "1"s

6.02 Fall 2011 Lecture 5, Slide #12

(n,k) Systematic Linear Block Codes

- Split data into *k*-bit blocks
- Add (*n-k*) parity bits to each block using (*n-k*) linear equations, making each block *n* bits long

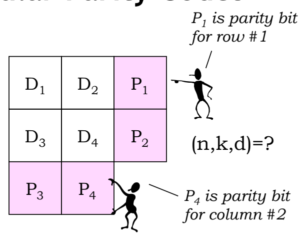


- Every linear code can be represented in systematic form

6.02 Fall 2011 Lecture 5, Slide #13

Example: Rectangular Parity Codes

Idea: start with rectangular array of data bits, add parity checks for each row and column. Single-bit error in data will show up as parity errors in a particular row and column, pinpointing the bit that has the error.



0 1 1
1 1 0
1 0

Parity for each row and column is correct ⇒ no errors

0 1 1
1 0 0
1 0

Parity check fails for row #2 and column #2 ⇒ bit D_4 is incorrect

0 1 1
1 1 1
1 0

Parity check only fails for row #2 ⇒ bit P_2 is incorrect

Rectangular Code Corrects Single Errors

Claim: The HD of the rectangular code with *r* rows and *c* columns is 3. Hence, it is a single error correction (SEC) code.

Code rate = *rc* / (*rc* + *r* + *c*).

If we add an overall parity bit P, we get a (rc+r+c+1, rc, 4) code

Improves error detection but not correction capability

D_1	D_2	D_3	D_4	P_1
D_5	D_6	D_7	D_8	P_2
D_9	D_{10}	D_{11}	D_{12}	P_3
P_4	P_5	P_6	P_7	P

Proof: Three cases.
(1) Msgs with HD 1 → differ in 1 row and 1 col parity
(2) Msgs with HD 2 → differ in either row OR col or both → HD >= 4 here.
(3) Msgs with HD 3 or more → systematic code so differ in that many bits

Decoding Rectangular Parity Codes

Receiver gets possibly corrupted word, *w*.

Calculates all the parity bits from the data bits.

If no parity errors, return *rc* bits of data.

Single row or column parity bit error → *rc* data bits are fine, return them

If parity of row *x* and parity of column *y* are in error, then the data bit in the (*x*,*y*) position is wrong; flip it and return the *rc* data bits

All other parity errors are *uncorrectable*. Return the data as-is, flag an "uncorrectable error"

What Next?

Linear block codes are widely used and are powerful → we've just seen the tip of the iceberg

Rectangular code is a good example, but
- #parity bits grows at least as sqrt(*k*) where *k* is #message bits
- Can we do better? What's the best we can do?

And can we decode linear block codes more systematically?

Next lecture: Bounds on the best possible code for a given error correction goal, Hamming codes, syndrome decoding of linear block codes, and interleaving for burst errors