AGENDA
1. Logistics
2. Course Pitch
3. Basic problem in coding theory
4. Formal definitions
5. Rate vs. Distance: Hamming bound

1. LOGISTICS
   • This course meets M/W 4:30-5:50pm
   • The COURSE WEBPAGE is at web.stanford.edu/~marykw/classes/CS250_W19
     • That is where you will find:
       - logistics info & tentative schedule
       - Lecture notes and Homework assignments
       - HIGHLIGHTS:
         • 4 HW Assignments. First one will be released WEDNESDAY.
         • Final project. This will be in teams of size 1-3 people.
         • No required textbook. I will post notes and reading

2. COURSE PITCH
   “ALGEBRAIC ERROR CORRECTING CODES”
   I. Error correcting codes are a fundamental tool for
      • communication
      • storage
      • complexity theory
      • algorithm design
      • cryptography
      • pseudorandomness
      • etc...
   II. Algebraic techniques are a fundamental tool for designing ECCs.
   
   Basically, this course is about the following fact:
   LOW-DEGREE POLYNOMIALS DON'T HAVE TOO MANY ROOTS.

As we will see, this fact is stupidly useful throughout CS and EE.
In this class we will discuss:

- Basics of Error Correcting Codes: combinatorial bounds + existential results
- Some basic abstract algebra: [finite fields - nothing fancy]
- The classic polynomial codes: Reed-Solomon and Reed-Muller
- [If time we will mention] fancier polynomial codes: Multiplicity Codes, Folded RS codes, Polar codes
- Algorithms for manipulating these codes in various settings: Unique decoding, list decoding, local decoding
- Applications?

In this class we will NOT discuss:

- Nitty gritty details of any one application (this is a THEORY course)
- LDPC codes, Turbo codes, Repotor Codes, Fountain Codes...
  [See Montanari's course EE388 for all that good stuff]

At the end of this course:

YOU SHOULD HAVE THE TOOLS TO USE
ERROR-CORRECTING CODES (and the algebraic tools behind them)
IN YOUR OWN RESEARCH/LIFE.

That means:

- Enough familiarity with terminology, constructions, algorithms, and notions of decoding to pick up a research paper and understand it.

- Exposure to lots of examples of how ECCs can be useful in a wide variety of settings.
The BASIC PROBLEM in CODING THEORY

The picture looks like this:

- MESSAGE \( x \) of length \( k \)
- CODEWORD \( c \) of length \( n \)
- ENCODE
- SOMETHING BAD HAPPENS
- CORRUPTED CODEWORD \( \tilde{c} \)

maybe some entries got deleted, changed, etc.

The goal:

**GIVEN \( \tilde{c} \), FIND (SOMETHING ABOUT) \( x \).**

**EXAMPLE: COMMUNICATION**

1. Alice has some message \( x \) she would like to send to Bob.
2. Alice encodes her message as \( c \).
3. Alice sends \( c \) to Bob over a noisy channel; Bob hears \( \tilde{c} \).
4. What did Alice mean to say???
EXAMPLE: STORAGE

1. Suppose $x$ is a file.
2. $x$ is stored; say on a CD or in a RAID array.
3. Something bad happens...
4. I still want $x$!

THINGS WE CARE ABOUT:

1. We should be able to handle the **SOMETHING BAD**, whatever that means.
2. We should be able to recover **WHAT WE WANT TO KNOW** about $x$.
3. We want to **MINIMIZE OVERHEAD**: $k/n$ should be as small as possible.
4. We want to do all this **EFFICIENTLY**.

**QUESTION** What are the trade-offs between 1-4?

It depends on how we model things:

- What is the **SOMETHING BAD**?
- What exactly do we **WANT TO KNOW**?
- What counts as **EFFICIENT**? What kind of access do we have to $C$?

Today we'll look at one way of answering these questions.

There are many legit ways, and we will see more throughout the quarter.
4) **Formal Definitions**

Let \( \Sigma \) be any finite set and let \( n > 0 \) be an integer.

**DEF.** A **code** \( C \) of **block length** \( n \) over an **alphabet** \( \Sigma \) is a subset \( C \subseteq \Sigma^n \).

So far, this is not a very interesting definition.

**EXAMPLE 1.** \( C = \{ \text{HELLOWORLD, BRUNCHTIME, ALLTHETIME} \} \)

is a code of block length 10 over \( \Sigma = \{A, B, \ldots, X, Y, Z\} \).

**EXAMPLE 2.** \( C = \{ (0, 0, 0, 0), (0, 0, 1, 1), (0, 1, 0, 1), (0, 1, 1, 0), (1, 0, 0, 1), (1, 0, 1, 0), (1, 1, 0, 0), (1, 1, 1, 1) \} \)

is a code of length 4 over \( \Sigma = \{0, 1\} \).

If \( \Sigma = \{0, 1\} \), we say \( C \) is a **binary code**.

What does this have to do with the picture from before? [this one]

Consider the map \( \text{ENC}: \{0, 1\}^3 \to \{0, 1\}^4 \) given by:

\[
\text{ENC}: (x_1, x_2, x_3) \mapsto (x_1, x_2, x_3, x_1 + x_2 + x_3 \mod 2)
\]

For example, \( \text{ENC}(0, 1, 1) = (0, 1, 1, 0) \).

Then \( C = \text{Im}(\text{ENC}) \). That is, \( C \) is the set of all codewords that could be obtained using this encoding map.
The second example can actually be used to fix bad stuff.

Suppose you see:

\[ 0 \quad \times \quad 0 \quad 1 \]

\[ \text{The SOMETHING BAD that happened obscured this entry.} \]

What is the missing bit?
It must be a 1, since \( 0 + \times + 0 = 1 \mod 2 \).

Suppose instead you see:

\[ 0 \quad 0 \quad 0 \quad 1 \]

Then we know SOMETHING went wrong (at least one bit was flipped) but we are not sure what it was.

We say that the code in EXAMPLE 2 can DETECT one error. But it cannot CORRECT one error. Let's see a code that can.

**EXAMPLE 3.** Consider the encoding map \( \text{ENC}: \{0,1\}^4 \rightarrow \{0,1\}^7 \)

\[ \text{ENC}: (x_1, x_2, x_3, x_4) \mapsto (x_1, x_2, x_3, x_4, x_2 + x_3 + x_4, x_1 + x_3 + x_4, x_1 + x_2 + x_4) \]

Let \( C = \text{Im}(\text{ENC}) \). So \( C \subseteq \{0,1\}^7 \), aka \( C \) is a BINARY CODE of LENGTH 7.

**ANOTHER WAY to VISUALIZE this CODE:**

\[ c_5 = x_2 x_3 x_4 \]
\[ c_6 = x_1 x_3 x_4 \]
\[ c_7 = x_1 x_2 x_3 \]

Put the message \( x_1, x_2, x_3, x_4 \) in the middle, and then the circles tell you how to fill in the rest.
**PUZZLE:** I took some $c \in C$ and flipped at most one bit, to obtain:

$$\hat{c} = (0, 1, 1, 1, 0, 1, 0)$$

What is $c$?

**ALTERNATIVE LOOK**

at the PUZZLE:

**SOLUTION:**

In a legit codeword, all three circles should sum to 0. Here, both the green circle and the red circle are messed up.

The thing those circles have in common is $\hat{c}_3$. So if I flip $\hat{c}_3$ I should get a legit codeword again:

$$\hat{c} = (0, 1, 0, 1, 0, 1, 0) \text{ or } (0, 1, 0, 1, 0, 1, 0)$$

And, $\hat{c}$ is the ONLY solution because flipping any other bit would mess up other circles.
Hooray! That works. But it seems pretty ad hoc. For the rest of this lecture and some of next one, we'll try to introduce some formalism to make this solution seem less ad hoc. At the same time we will flesh out what we care about for ECCs.

First some definitions:

**DEF.** The **Hamming Distance** between $x, y \in \Sigma^n$ is

$$\Delta(x, y) := \sum_{i=1}^{n} \mathbb{1}\{x_i \neq y_i\}$$

The **Relative Hamming Distance** between $x, y \in \Sigma^n$ is

$$s(x, y) := \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\{x_i \neq y_i\} = \frac{\Delta(x, y)}{n}.$$  

**DEF.** The **Minimum Distance** of a code $C \subseteq \Sigma^n$ is

$$\min_{c \neq c', c \in C} \Delta(c, c').$$

**CLAIM** The code in EXAMPLE 3 has minimum distance 3.
If the CLAIM is true, it explains why that code can correct one error:

Indeed, if $\Delta(c, c') \geq 3 \forall c \neq c' \in C$, then

$$\Delta(\tilde{c}, c) = 1 \Rightarrow \Delta(\tilde{c}, c') \geq 2 \forall c' \in C \text{ other than } c.$$ 

by the triangle inequality. Thus, the "correct" codeword $c \in C$ is uniquely defined by "the one that is closest to $\tilde{c}$.”

To prove the CLAIM:

- You can probably convince yourself by staring at the diagram (in the same way we convinced ourselves we could always fix one error).

- But we'll see a much less ad hoc way to establish distance after we build up some machinery for LINEAR CODES in Lecture 2, so let's put it aside for now.

The POINT of this discussion was that:

**MINIMUM DISTANCE** is a reasonable proxy for robustness.

---

**Disclaimer**

I will frequently draw pictures as though Hamming distance is Euclidean distance, and $\mathbb{F}_2^n$ is $\mathbb{R}^n$.

---

**Note:**

- We should be able to handle the SOMETHING BAD, whatever that means.
- We should be able to recover WHAT WE WANT TO KNOW about $x$. 
That is,

- In EXAMPLE 2, the code had minimum distance 2 (check this!) and could CORRECT 1 ERASURE and DETECT 1 ERROR.
- In EXAMPLE 3, the code had minimum distance 3, and could CORRECT 1 ERROR.

More generally, a code with distance d can:

- correct \( \leq d - 1 \) erasures
- detect \( \leq d - 1 \) errors
- correct \( \leq \left\lfloor \frac{d - 1}{2} \right\rfloor \) errors

For these two, the (inefficient) algorithm is:

"if you see \( \hat{c} \), return \( c \in \mathbb{C} \) that's closest to \( \hat{c} \)."

For this one, the (inefficient) alg. is:

"If \( \hat{c} \in \mathbb{C} \), say that something is wrong."

The picture looks like this:

- If \( c \) is the "correct" codeword and \( \leq \left\lfloor \frac{d - 1}{2} \right\rfloor \) errors are introduced, we may end up with \( \hat{c}_1 \). Since all the balls are disjoint, we can find \( c \) from \( \hat{c}_1 \).
- However, if \( \leq d - 1 \) errors are introduced, we may end up with \( \hat{c}_2 \). Now it's possible that \( \hat{c}_2 \) came from \( c \) or that it came from \( c' \); we can't tell. However, since each ball doesn't contain any codeword other than its center, we can tell that something went wrong.
Returning to this, we can now clarify the first two things:

THINGS WE CARE ABOUT:

1. We should be able to handle the SOMETHING BAD, whatever that means.
2. We should be able to recover WHAT WE WANT TO KNOW about x.
3. We want to MINIMIZE OVERHEAD: k/n should be as small as possible.
4. We want to do all this EFFICIENTLY.

If we want:

1. We should be able to handle \( \lfloor \frac{d-1}{2} \rfloor \) WORST-CASE ERRORS or d-1 WORST-CASE ERASURES
2. We want to recover ALL OF x (aka correct the errors or erasures)

Then we should say \( 1 \) & \( 2 \) We want MINIMUM DISTANCE \( d \).

Next we will move on to \( 3 \).

Aside. A natural question at this point is, “what if I don't want to handle worst-case errors/erasures?” For example, if my code has minimum distance \( d \), and I have two codewords:

\[
C = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) \in \mathbb{F}_{2}^{12}
\]

\[
C' = (1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0) \in \mathbb{F}_{2}^{12}
\]

Then if an adversary chooses to flip the first two bits, we'd be confused. But instead say two bits get flipped at random. The probability we get confused is \( \frac{\binom{d}{2}}{2^d} \), which might be quite small!

The random-error model (also called the “Shannon model” or “Stochastic model”) is natural and important! We will discuss it a little bit in this class. However, most of our focus will be in the worst-case model (also called the “Hamming model” or “adversarial model.”)
Moving on to 3, what do we mean by “overhead”?

**DEF.** The **MESSAGE LENGTH** (sometimes called **DIMENSION**) of a code $C$ over an alphabet $\Sigma$ is defined to be $k = \log_{|\Sigma|} |C|$.  

This definition makes sense with our operational understanding:

\[ (\text{message } x \text{ of length } k \text{ over } \Sigma) \xrightarrow{\text{Enc}} (\text{codeword } c \in C) \]

$|\Sigma|^k$ possibilities  \quad  $|C|$ possibilities

So $|\Sigma|^k = |C|$  \implies  $k = \log_{|\Sigma|} |C|$.  

**DEF.** The **RATE** of a code $C \subseteq \Sigma^n$ with block length $n$ over an alphabet $\Sigma$ is 

\[ R = \frac{\log_{|\Sigma|} |C|}{n} = \frac{\text{message length } k}{\text{block length } n}. \]

So if $R$ is close to 1, that's **GOOD**. Not much overhead.  
And if $R$ is close to 0, that's **BAD**. Lots of overhead.  

**DEF.** A code with distance $d$, message length $k$, block length $n$, and alphabet $\Sigma$ is called a $(n, k, d)_{|\Sigma|}$ code.

**QUESTION.** What is the best trade-off between rate and distance?  

This question is still open for binary codes.  
But there's lots we do know.
What is the best trade-off between rate and distance we can hope for? The HAMMING BOUND gives one bound on this.

Let's return to the picture we had before, with disjoint Hamming balls of radius \( L^{d-1}_r \):

• We have \(|C|\) disjoint Hamming balls of radius \( L^{d-1}_r \).
• There can't be too many of them or they wouldn't all fit in \( \Sigma^n \).

To be a bit more precise:

**DEF.** The HAMMING BALL in \( \Sigma^n \) of radius \( e \) about \( x \in \Sigma^n \) is

\[
B_{\Sigma^n}(x,e) = \{ y \in \Sigma^n : \Delta(x,y) \leq e \}.
\]

The VOLUME of \( B_{\Sigma^n}(x,e) \) is \( \text{Vol}_{\Sigma^n}(e,n) = |B_{\Sigma^n}(x,e)| \)

Notice that \( |B_{\Sigma^n}(x,e)| \) does not depend on \( x \).

Say that \( |\Sigma| = q \). Then

\[
\text{Vol}_q(e,n) = 1 + (q)^1 + (q)^2 + \cdots + (q)^e
\]

*Sometimes I will drop the "\( \Sigma \)" from the \( B_{\Sigma^n}(x,e) \) notation*.

*Sometimes I will write \( B_{\Sigma^n}(x,e/n) \) if it's more convenient to talk about relative distance.*

\[
\text{Vol}_q(e,n) = \sum_{\text{all the elements of } \Sigma^n \text{ of weight } 1} + \sum_{\text{all the elements of } \Sigma^n \text{ of weight } 2} + \cdots + \sum_{\text{all the elements of } \Sigma^n \text{ of weight } e}
\]
So that means that if a code \( C \in \Sigma^n \) has distance \( d \) and message length \( k \), where \( |\Sigma| = q \),

\[
|C| \cdot \text{Vol}_q \left( \left\lfloor \frac{d-1}{2} \right\rfloor, n \right) \leq q^n
\]

so taking logs of both sides,

\[
\log_q(|C|) + \log_q \left( \text{Vol}_q \left( \left\lfloor \frac{d-1}{2} \right\rfloor, n \right) \right) \leq n
\]

\[
\Rightarrow \quad \text{Rate} = \frac{k}{n} \leq 1 - \frac{\log_q \left( \text{Vol}_q \left( \left\lfloor \frac{d-1}{2} \right\rfloor, n \right) \right)}{n}
\]

This is called the HAMMING BOUND.

Back to EXAMPLE 3, which was a \( (7, 4, 3)_2 \) code.

- We have \( \left\lfloor \frac{d-1}{2} \right\rfloor = 1 \)
- \( \text{Vol}_2 \left( 1, 7 \right) = 1 + \binom{7}{1} 1 = 8 \)
- So \( \frac{k}{n} \leq 1 - \frac{\log_2(8)}{7} = 1 - \frac{3}{7} = \frac{4}{7} \).
- And in fact \( \frac{k}{n} = \frac{4}{7} \), so in this case the Hamming bound is tight!

Notes about this example:

- When the Hamming bound is tight, we say the code is PERFECT.
- EXAMPLE 3 (which is perfect) is a special case of something called a HAMMING CODE.
- You will explore this more on your homework.
Now we understand the first 3 of our desiderata:

**Things we care about:**

- **SOMETHING BAD** that we should be able to handle.
- **WHAT WE WANT TO KNOW** that we should be able to recover.
- **MINIMIZE OVERHEAD** that we want to minimize.
- **EFFICIENTLY** that we want to do all this.

That is, (for now), our goal is to design codes $C \subseteq \Sigma^n$ so that:

- The **DISTANCE** of $C$ is as large as possible.
- The **RATE** of $C$ is as close to 1 as possible.

Even without the algorithmic considerations, understanding the trade-off between rate and distance turns out to be a fascinating combinatorial question!

In fact, for binary codes ($|\Sigma|=2$), this question is **STILL OPEN**! (We saw that EXAMPLE 3 was optimal for $n=7$ and $k=4$, but what about in general?)

Next time, we'll give an overview of abstract algebra, and then give some more definitions that will further de-ad-hocify EXAMPLE 3.

That's it for today.

**Questions to ponder:**

1. How would you generalize the code in EXAMPLE 3 to larger $n$?
2. What is the best bound you can come up with on the rate of a code $C \subseteq \Sigma^n$ with distance $d$? Can you beat the Hamming bound?