CS5275 Lecture 9: Error-Correcting Codes

Jonathan Scarlett

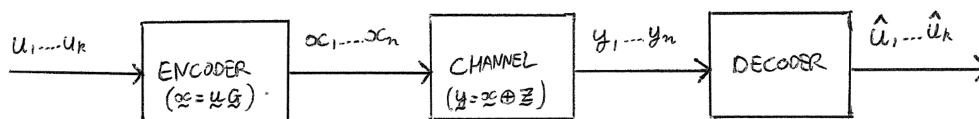
December 6, 2024

Useful references:

- Blog post on Hamming codes[1]
- TCS Toolkit lectures: `https://www.youtube.com/watch?v=H7XFslRXJys` and the 4 videos after
- Cover/Thomas "Elements of Information Theory", Section 7.11
- MacKay "Information Theory, Inference, and Learning Algorithms", Sections 11.4–11.5, Chapters 13–14
- (Beyond the scope of this course) Survey article on coding,[2] and/or the advanced textbook "Modern Coding Theory" (Richardson and Urbanke)

1 Background: Reliable Communication

- The communication problem was already mentioned in the information theory lecture, and is slightly modified here to highlight that what we are sending is *bits*.
- The setup is summarized as follows (we will explain the equation $\mathbf{x} = \mathbf{u}\mathbf{G}$ later):



  - $\mathbf{u} = (u_1, \dots, u_k)$ is a sequence of k bits that we would like to send.
  - That sequence gets encoded into a binary codeword $\mathbf{x} = (x_1, \dots, x_n)$ (later we will also consider non-binary codewords), where $n \geq k$ because the idea of encoding is to *add redundancy that allows resilience to bits getting flipped.*
  - The codeword $\mathbf{x} = (x_1, \dots, x_n)$ is sent through a communication channel to produce $\mathbf{y} = (y_1, \dots, y_n)$, which is also binary. The channel can be described as $\mathbf{y} = \mathbf{x} \oplus \mathbf{z}$, where $\mathbf{z} \in \{0, 1\}^n$ indicates which bits are flipped, and the operator $\oplus$ is applied bit-by-bit.
  - These output bits are used to construct an estimate $\hat{\mathbf{u}} = (\hat{u}_1, \dots, \hat{u}_k)$ of the original k bits.

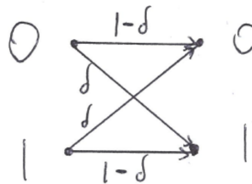
[1] `https://jeremykun.com/2015/03/02/hammings-code/`
[2] `https://arxiv.org/pdf/1908.09903.pdf`

- Our goal is to achieve *speed* and *reliability* in communication:
  - *Reliability* means ensuring that $\hat{\mathbf{u}} = \mathbf{u}$ (or, if the channel is modeled as being random, having a *high probability* of this occurring).
  - *Speed* is captured by the notion of *rate*:

$$R = \frac{k}{n}.$$

    For example, if the encoder maps k bits to $n = 3k$ bits, then the rate is $\frac{1}{3}$.

- We will consider two types of modeling for the channel:
  - Under a probabilistic model, each y_i is assumed to be produced from the corresponding x_i by passing it through a *binary symmetric channel* (BSC):



    That is, the output is correct with probability $1 - \delta$, but flipped with probability δ.
  - An alternative approach is to have no probabilistic model, but simply *assume that at most δn bits are flipped* for some $\delta \in (0, 1)$.

- In the probabilistic case, much more general channel models are possible, with transition law $P_{Y|X}$ and possibly non-binary alphabets, but we will focus on the BSC.

- Another model of particular interest is the *erasure model*, where (in the probabilistic case) $Y = X$ with probability $1 - \epsilon$, but Y equals an "erasure symbol" with probability ϵ, with that symbol indicating "x_i is unknown". Similarly, in the non-probabilistic case, we could assume that at most ϵn symbols get erased for some $\epsilon \in (0, 1)$.

2 Parity Checks and the Hamming Code

Parity check.

- A *parity check* of a sequence of bits $b_1, \ldots, b_m$ is an additional bit equaling 1 if the number of 1's in $b_1 \ldots, b_m$ is odd, and 0 if the number of 1's is even.
  - Hence, in either case, there is an even number of 1's in the sequence $b_1, \ldots, b_m c$, where c is the parity check bit

- We can express this via modulo-2 arithmetic: For $a, b \in \{0, 1\}$, let the $\oplus$ operator be defined as

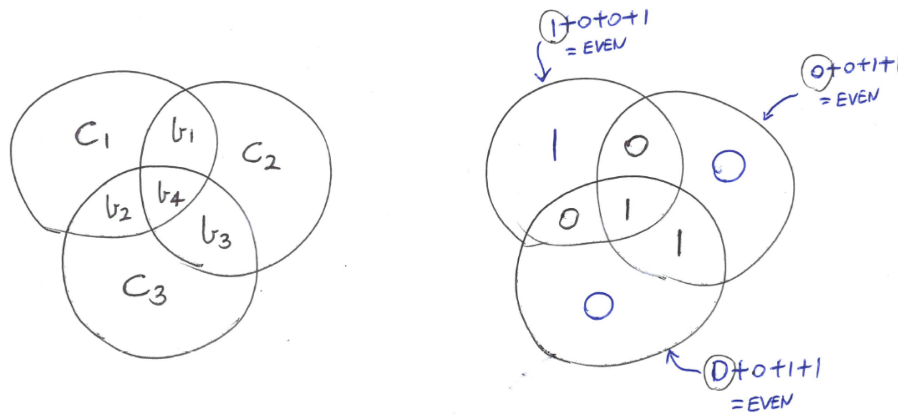
$$0 \oplus 0 = 1 \oplus 1 = 0$$
$$0 \oplus 1 = 1 \oplus 0 = 1.$$

  Then the parity check of $b_1, \ldots, b_m$ is $c = b_1 \oplus \ldots \oplus b_m$.

- If we transmit the length-$(m+1)$ sequence $b_1 \dots b_m c$ across a channel, and one of the bits is flipped, we will notice that the total number of 1's is no longer even. Hence, we can *detect* a single bit flip. However, a little thought reveals that we cannot *correct* it.
- The idea that permits error correction: *Send multiple parity checks applied to different groups of bits*

Simple examples.

- Perhaps the simplest type of code is the *repetition code*, which (for example) encodes 1010 into 111000111000. By a majority vote rule, this permits the *correction* of one bit flip in each of the groups of 3 bits. We can also repeat more than 3 times for improved reliability, but this quickly becomes extremely inefficient in terms of the communication rate.
- A famous example of a less trivial code is the *Hamming code*, which (for the simplest version of it) maps 4 bits to 7 bits and can correct a single bit flip:



Here the original bits are $b_1b_2b_3b_4$, and the check bits $c_1c_2c_3$ are chosen to make the number of 1's per circle even. In the above terminology, each c_i is a *parity check* of the b_i's in the corresponding circle.

  - Note: When we talk about being able to correct a certain number of bit flips, this may include flips in both the uncoded bits b_i and the check bits c_i.
  - How to correct one bit flip: Observe that each possible single bit flip changes a unique combination of circles from "even number of 1s" to "odd number of 1s" (e.g., the middle bit changes all 3 circles; the top-left bit only changes the top-left circle). Therefore, the decoder can check which circles have an odd number of 1s, and un-flip the corresponding bit. If the decoder sees all 3 circles already having an even number of 1s, then no changes are made.

3 Linear Codes

Recall from the first section that k is the number of message bits, n is the number of codeword bits, and $R = \frac{k}{n}$ is the communication rate.

Definition and generator matrix.

- For reasons to be made clear shortly, we say that any code comprised of parity checks is a *linear code*.

- We distinguish between the following two cases:
  - A **systematic parity-check code** is one in which the first k (out of n) bits of $\mathbf{x}$ are always precisely the original k bits, and the remaining $n-k$ bits are parity checks:

$$x_i = u_i, \quad i = 1, \dots, k,$$
$$x_i = \bigoplus_{j=1}^{k} u_j g_{j,i}, \quad i = k+1, \dots, n$$

    where $g_{j,i} = 1$ if the parity check in location i includes u_j, and $g_{j,i} = 0$ otherwise. For instance, the Hamming code described above is a systematic parity-check code.
  - A **general parity-check code** is one in which all n codeword bits may be arbitrary parity checks:

$$x_i = \bigoplus_{j=1}^{k} u_j g_{j,i}, \quad i = 1, \dots, n.$$

    Clearly a systematic code is a special case of this, since it corresponds to setting $g_{j,i} = \mathbf{1}\{j = i\}$ for $i = 1, \dots, k$.
- The above (general) formula for generating each x_i from $u_1, \dots, u_k$ can be succinctly summarized as a (modulo-2) vector-matrix multiplication:

$$\mathbf{x} = \mathbf{u}\mathbf{G}, \quad \text{(in modulo-2 arithmetic)}$$

  where $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{u} = (u_1, \dots, u_k)$ are the suitable row vectors, and

$$\mathbf{G} = \begin{bmatrix} g_{1,1} & g_{1,2} & \cdots & g_{1,n} \\ g_{2,1} & g_{2,2} & \cdots & g_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k,1} & g_{k,2} & \cdots & g_{k,n} \end{bmatrix}$$

  is known as the *generator matrix*.
  - Interpretation: The 1's in each column indicate which bits are included in the parity check
- In the special case of a systematic code, this simplifies to

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & \dots & 0 & g_{1,k+1} & \cdots & g_{1,n} \\ 0 & 1 & \dots & 0 & g_{2,k+1} & \cdots & g_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & g_{k,k+1} & \cdots & g_{k,n} \end{bmatrix}$$

  with the left-most $k \times k$ sub-matrix being the identity matrix.
- With the mapping from $\mathbf{u}$ to $\mathbf{x}$ being described by the matrix multiplication $\mathbf{x} = \mathbf{u}\mathbf{G}$ (in modulo-2 arithmetic), we can now justify the terminology *linear code*: If $\mathbf{u}$ and $\mathbf{u}'$ are two different message

sequences, and their corresponding codewords are $\mathbf{x} = \mathbf{u}\mathbf{G}$ and $\mathbf{x}' = \mathbf{u}\mathbf{G}'$, then

$$\begin{aligned} \mathbf{x} \oplus \mathbf{x}' &= \mathbf{u}\mathbf{G} \oplus \mathbf{u}'\mathbf{G} \\ &= (\mathbf{u} \oplus \mathbf{u}')\mathbf{G}, \end{aligned}$$

which means that $\mathbf{x} \oplus \mathbf{x}'$ is also a codeword (corresponding to message $\mathbf{u} \oplus \mathbf{u}'$). In other words, the (modulo-2) sum of any two valid codewords is another valid codeword.

- Note: We have extended the $\oplus$ notation to vectors/sequences, which is done via a bit-by-bit application of the definition above, e.g., $0001 \oplus 1101 = 1100$ and $101 \oplus 010 = 111$.

- **Examples.** With $k = 4$, the generator matrices for the single-parity-check code and Hamming code (described at the start of the lecture) are given by

$$\mathbf{G}_{\text{parity}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}, \qquad \mathbf{G}_{\text{Hamming}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}. \tag{1}$$

Sometimes you might see the latter with the last 3 columns in a different order, but re-ordering columns just amounts to re-labeling the indices of parity check bits.

  - Pre-multiplying $\mathbf{G}_{\text{Hamming}}$ by all 16 possible $\mathbf{u}$ sequences, we can list all the codewords of the Hamming code:

$$\begin{array}{cccc} 0000000 & 0001111 & 0010011 & 0011100 \\ 0100101 & 0101010 & 0110110 & 0111001 \\ 1000110 & 1001001 & 1010101 & 1011010 \\ 1100011 & 1101100 & 1110000 & 1111111. \end{array}$$

  - For instance, the codeword 1100011 is obtained from $\mathbf{u} = 1100$ by taking the modulo-2 sum of the first two rows of $\mathbf{G}_{\text{Hamming}}$. For $\mathbf{u} = 1111$, we take the modulo-2 sum of all 4 rows.

- As we will see in the tutorial, any general code can be reduced to an "equivalent" systematic code by applying a technique similar to Gaussian elimination on $\mathbf{G}$.

Parity-check matrix.

- A matrix closely related to $\mathbf{G}$, but which will be more directly useful when it comes to decoding, is called the *parity-check matrix* $\mathbf{H}$. It is an $n \times (n-k)$ matrix that satisfies

$$\mathbf{x}\mathbf{H} = \mathbf{0} \iff \mathbf{x} \text{ is a valid codeword.}$$

Notice the distinction:

  - $\mathbf{G}$ is used to *generate* $\mathbf{x}$ from $\mathbf{u}$;
  - $\mathbf{H}$ is used to *check* if $\mathbf{x}$ can be generated from *any* $\mathbf{u}$ (doing this naively using $\mathbf{G}$ by testing all 2^k possible $\mathbf{u}$ would be grossly inefficient).

- While such check matrices exist for all generator matrices, we will focus our attention on the systematic case, as it is much simpler. Recall the two formulas we gave for x_i (in terms of the $\{u_j\}$ and $\{g_{j,i}\}$) in the systematic case; substituting the first into the second gives

$$x_i = \bigoplus_{j=1}^{k} x_j g_{j,i}, \quad i = k+1, \dots, n.$$

Adding x_i (modulo 2) to both sides, the left-hand side gives $x_i \oplus x_i = 0$, and we are left with

$$\left(\bigoplus_{j=1}^{k} x_j g_{j,i} \right) \oplus x_i = 0, \quad i = k+1, \dots, n.$$

Converting to matrix form reveals that the following choice of $\mathbf{H}$ indeed gives $\mathbf{x}\mathbf{H} = \mathbf{0}$:

$$\mathbf{H} = \begin{bmatrix} g_{1,k+1} & g_{1,k+2} & \cdots & g_{1,n} \\ g_{2,k+1} & g_{2,k+2} & \cdots & g_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k,k+1} & g_{k,k+2} & \cdots & g_{k,n} \\ 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}.$$

It is also not hard to establish the opposite, i.e., if $\mathbf{x}\mathbf{H} = \mathbf{0}$ then $\mathbf{x}$ is indeed a valid codeword.

- Stated more succinctly, we have

$$\mathbf{G} = [\mathbf{I}_k \;\; \mathbf{P}] \quad \Longrightarrow \quad \mathbf{H} = \begin{bmatrix} \mathbf{P} \\ \mathbf{I}_{n-k} \end{bmatrix},$$

where $\mathbf{I}_m$ is the $m \times m$ identity matrix, and $\mathbf{P}$ is the remaining $k \times (n-k)$ submatrix of $\mathbf{G}$.

- **Examples.** The check matrices corresponding to (1) are

$$\mathbf{H}_{\text{parity}} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \qquad \mathbf{H}_{\text{Hamming}} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

- To give a flavor of why the check matrix is useful for decoding, notice that if $\mathbf{y} = \mathbf{x} \oplus \mathbf{z}$ with $\mathbf{z}$ indicating

which bits got flipped, then

$$\begin{aligned} \mathbf{yH} &= (\mathbf{x} \oplus \mathbf{z})\mathbf{H} \\ &= (\mathbf{xH}) \oplus (\mathbf{zH}) \\ &= \mathbf{zH}, \end{aligned} \tag{2}$$

where we first used linearity, and then the fact that $\mathbf{xH} = \mathbf{0}$ for any valid codeword $\mathbf{x}$.

– In particular, if $\mathbf{z}$ only contains a single 1 (i.e., only one bit got flipped), then $\mathbf{yH}$ is simply the i-th row of $\mathbf{H}$, where i is the index of the flipped bit.
– But notice that in $\mathbf{H}_{\text{Hamming}}$, all the rows are distinct! This means that by looking at $\mathbf{H}_{\text{Hamming}}$, we can immediately identify which bit got flipped and therefore correct it.

Notes on storage and computation.

- Notice that the code is fully specified by $\mathbf{G}$ (or $\mathbf{H}$), which only requires nk bits of storage – much smaller than the exponential storage requirement for the random coding technique used to prove the channel coding theorem!
- However, efficient decoding (getting from the noisy channel output $\mathbf{y}$ back to $\mathbf{u}$) is still challenging, and designing a good choice of $\mathbf{G}$ is also difficult (but do-able!).
- In fact, at this stage it is unclear whether any good choices of G exist! This is addressed in Chapter 14 of MacKay's book (optional reading), where it is shown that a *random generator matrix* G (i.e., each entry is 1 or 0 with equal probability) achieves arbitrarily small error probability on the BSC at all rates below capacity, when used in conjunction with an optimal decoder. Thus, **very good linear codes exist**, at least if we ignore computational constraints on the decoder.

4 Distance Properties

We will only touch on the basics of distance properties here; if you are interested in knowing more, see Chapter 13 of MacKay's book.

Definition and properties.

- **Definition 1.** The *Hamming distance* between two vectors $\mathbf{x} = (x_1, \ldots, x_n)$ an $\mathbf{x}' = (x'_1, \ldots, x'_n)$ (having the same length n) is the number of positions in which they differ:

$$d_{\text{H}}(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{n} \mathbb{1}\{x_i \neq x'_i\}.$$

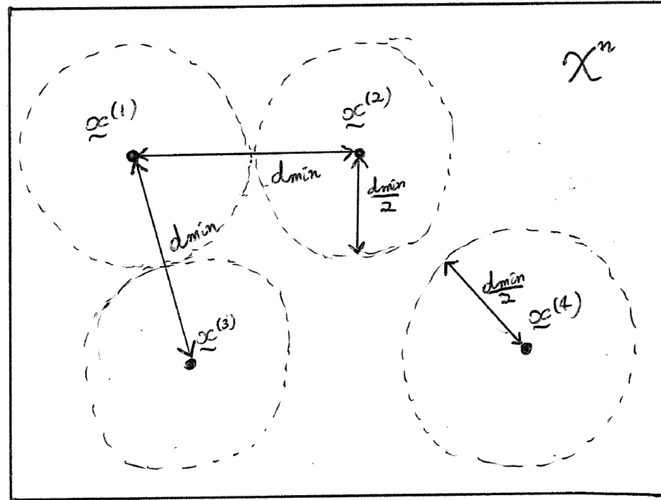
For instance, the Hamming distance between 00110011 and 00010111 is 2.

- **Definition 2.** The *minimum distance* of a codebook $\mathcal{C}$ of length-n codewords is

$$d_{\min} = \min_{\mathbf{x} \in \mathcal{C}, \mathbf{x}' \in \mathcal{C} \,:\, \mathbf{x} \neq \mathbf{x}'} d_{\text{H}}(\mathbf{x}, \mathbf{x}').$$

Intuitively, we should expect higher $d_{\min}$ to mean better robustness to noise in the channel.

- The following illustration shows that if the minimum distance is $d_{\min}$, then (at least given enough computation time) it is possible to *correct up to* $d_{\min} - 1$ *erasures* and *correct up to* $\frac{d_{\min}-1}{2}$ *bit flips*:



  - For correcting erasures: Simply note that if $d_{\min} - 1$ code symbols are replaced by an erasure symbol '?', then there is only one way to fill them in to get a valid codeword (otherwise, the minimum distance would be $d_{\min} - 1$ or less!)
  - For correcting flips: Note that the balls of radius $\frac{d_{\min}-1}{2}$ around each codeword cannot overlap. This means that if we decode each $\mathbf{y}$ to the codeword in its closest ball, we will always be correct if at most $\frac{d_{\min}-1}{2}$ flips occurred.

- **Claim.** If $\mathcal{C}$ is the set of codewords formed by a given linear code with $d_{\min} > 0$,[3] then

$$d_{\min} = \min_{\mathbf{x} \in \mathcal{C} \,:\, \mathbf{x} \neq \mathbf{0}} w(\mathbf{x}),$$

where $w(\mathbf{x}) = \sum_{i=1}^n \mathbb{1}\{x_i = 1\}$ is the *weight* of the codeword $\mathbf{x}$. Hence, for linear codes, the minimum distance equals the minimum weight.

  - Proof: Let $\mathbf{x}', \mathbf{x}''$ be the two codewords at a minimum distance from each other. Then by linearity, $\mathbf{x}' \oplus \mathbf{x}''$ is also a valid codeword, and its weight is precisely $d_{\mathrm{H}}(\mathbf{x}', \mathbf{x}'') = d_{\min}$. In addition, the assumption $d_{\min} > 0$ implies that it is not the all-zero codeword.

    Conversely, since $\mathbf{x} = \mathbf{0}$ is always a valid codeword (corresponding to $\mathbf{u} = \mathbf{0}$), any codeword weight also corresponds to a distance (to the all-zero codeword).

- **Example.** Recall the 16 codewords we listed earlier for the Hamming code. The minimum non-zero weight (and hence minimum distance) is 3, which is consistent with the fact that the Hamming code can correct $\frac{3-1}{2} = 1$ bit flip.

- **Notes on more general codes.**

  - Naturally, there is an inherent trade-off between achieving a high minimum distance (i.e., high $d_{\min}$) and a high rate (i.e., high $R = \frac{k}{n}$). Understanding this trade-off has been a major aspect

[3] A code with $d_{\min} = 0$ is a terrible idea – it means two different vectors of information bits $\mathbf{u}, \mathbf{u}'$ lead to the same codeword!

of coding theory for decades, with two famous examples being the Gilbert-Varshamov bound (achievability / existence) and the Sphere Packing bound (converse / impossibility), which you can look up if you are interested.

  - When designing practical codes, even the following goal is non-trivial: *As $n \to \infty$, achieve a positive rate (i.e., $\frac{k}{n} \to R^* > 0$) and a constant-fraction minimum distance (i.e., $\frac{d_{\min}}{n} \to \delta^* > 0$).*
  - In Section 6 we will cover a particularly well-known class known as Reed-Solomon Codes with very strong distance properties, and in the next lecture we will show that a class called Expander Codes can also achieve the above goal.

(Optional**) Distance vs. capacity.**

- While a high minimum distance seems like a nice property to have, it corresponds to a fundamentally different modeling assumption compared to the channel capacity:
  - The minimum distance goal is aligned with *worst-case errors*, where (for instance) we need to be able to correct *arbitrary patterns* of up to δn bit flips introduced by the channel.
  - The channel capacity goal is aligned with *random errors*, where (for instance) we need to be able to correct bit flips that occur *independently* with probability δ each.

  Neither of these goals should be viewed as "better" than the other in general; either may be preferable depending on the application.

- Of course, it is worth asking whether achieving capacity and attaining good distance properties are actually equivalent goals. However, as argued in Chapter 13 of MacKay's book, this is not the case:
  - The best possible rate with minimum distance δn is at most the channel capacity with *double* the noise level, 2δ;
  - The channel capacity is positive for any $\delta \in \left(0, \frac{1}{2}\right)$, but no positive rate can be achieved for $\delta > \frac{1}{4}$ if one insists on a minimum distance δn;
  - Examples are known where the channel capacity is achieved despite a "bad" minimum distance.
- Although distinct from achieving capacity, the design of codes with good distance properties is a very important problem in its own right. Examples of such codes include BCH codes (a generalization of Hamming codes), and Reed-Solomon codes, the latter of which we cover below.

5 Decoding Techniques

Decoding is the aspect of linear codes that is much harder to make efficient (while maintaining strong guarantees and/or practical performance). We will not explore this aspect in significant detail (see Section 7 for some pointers to topics that cover this).

In this section, we briefly outline some decoding rules that are *optimal* (in some sense to be described) but *computationally inefficient* (unless the number of bit flips is very small).

Maximum-likelihood decoding:

- Suppose that the output $\mathbf{y}$ is produced from $\mathbf{x}$ via a probabilistic model $P_{\mathbf{Y}|\mathbf{X}}$.

- Then, it can be shown (proof omitted) that if the message bits are uniformly random, the following decoding rule is optimal in the sense of minimizing the error probability $P_{\rm e}\mathbb{P}[\hat{\mathbf{u}} \ne \mathbf{u}]$:

$$\hat{\mathbf{u}} = \arg\max_{\mathbf{u}'} P_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}(\mathbf{u}')),$$

  where $\mathbf{x}(\mathbf{u}')$ is the codeword corresponding to $\mathbf{u}'$. This is true even for non-linear codes.

- Unfortunately, the above maximum is over 2^k choices, so checking all of these is prohibitive unless k is very small.

Minimum distance decoding:

- For the binary symmetric channel (BSC), we can write $P_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ as a product $\prod_{i=1}^n P_{Y|X}(y_i|x_i)$, where each $P_{Y|X}(y|x_i)$ is equal to $1-\delta$ if $y = x_i$, and δ otherwise.

- Using this, it can fairly easily be shown that the maximum-likelihood decoding rule becomes equivalent to the following for the BSC when $\delta < \frac{1}{2}$:

$$\hat{\mathbf{u}} = \arg\min_{\mathbf{u}'} d_{\rm H}(\mathbf{x}(\mathbf{u}'), \mathbf{y}).$$

  This is known as *minimum distance decoding.*

- Minimum distance decoding is also highly desirable under the non-probabilistic (distance-based) viewpoint – if there are at most $\frac{d_{\rm min}-1}{2}$ bit flips, then minimum distance decoding is guaranteed to recover the correct codeword.

- However, like with maximum-likelihood decoding, the computational cost may be highly prohibitive.

Syndrome decoding.

- Syndrome decoding is a technique that takes the idea of minimum distance one step further in the special case of linear codes.

- Recall the evaluation of $\mathbf{yH}$ given above:

$$\begin{aligned} \mathbf{yH} &= (\mathbf{x} \oplus \mathbf{z})\mathbf{H} \\ &= \big(\mathbf{xH}\big) \oplus \big(\mathbf{zH}\big) \\ &= \mathbf{zH}, \end{aligned} \tag{3}$$

  with the middle step using the fact that $\mathbf{xH} = \mathbf{0}$ for any valid codeword $\mathbf{x}$.

- The quantity

$$\mathbf{S} = \mathbf{zH} = \mathbf{yH}$$

  is called the *syndrome*, and we have just shown that it can immediately be computed from the check matrix $\mathbf{H}$ given the channel output $\mathbf{y}$.

- It can fairly easily be shown that for a linear code, if the syndrome is $\mathbf{S}$, then the minimum-distance codeword to $\mathbf{y}$ can be obtained by finding

$$\hat{\mathbf{z}} = \arg\min_{\tilde{\mathbf{z}} \,:\, \tilde{\mathbf{z}}\mathbf{H} = \mathbf{S}} w(\tilde{\mathbf{z}})$$

(where $w(\tilde{\mathbf{z}})$ is the number of 1's in $\tilde{\mathbf{z}}$) and then computing

$$\hat{\mathbf{x}} = \mathbf{y} \oplus \hat{\mathbf{z}}.$$

In addition, if the code is systematic, the estimate $\hat{\mathbf{u}}$ of the original information bits can then be formed by taking the first k entries of $\hat{\mathbf{x}}$.

- Using syndrome decoding, if very few bit flips occurred during transmission, then we may avoid searching over an exponentially large number of codewords – there are only n vectors $\tilde{\mathbf{z}} \in \{0,1\}^n$ of weight 1, $\binom{n}{2}$ of weight 2, and so on. Since we are minimizing $w(\cdot)$, we can start from the smallest weights, work upwards, and stop once a match is found.
- On the other hand, if a significant number of bit flips occur during transmission, this decoding method still typically requires far too much computation to be practical. But the general idea still forms the basis of certain very powerful computationally efficient codes.

Interested students may refer to `https://www.comp.nus.edu.sg/~scarlett/CS3236_notes/06-Codes.pdf` for the proofs that were omitted in this section.

6 Reed-Solomon Codes

This section gives an overview Reed-Solomon Codes, which are of one of the most widely-used distance-based codes (i.e., codes designed to have high $d_{\min}$). These are used extensively throughout theoretical computer science, as well as practical applications such as DVD and QR codes.

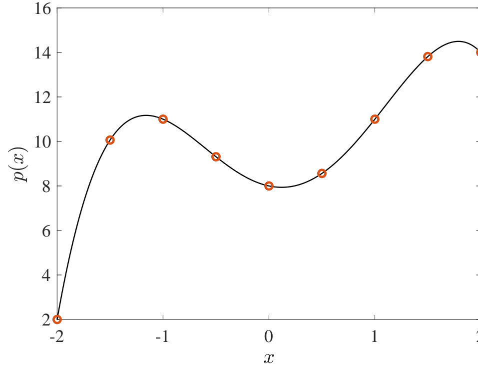
Unlike what we covered in the previous sections, these codes are *non-binary*, i.e., the symbols of the length-n codewords take values in an alphabet with (significantly) more than two symbols. We will later discuss methods for converting a non-binary code to a binary code.

6.1 Intuition

Here we give the rough idea behind Reed-Solomon Codes using concepts that should be more familiar (but can't be used directly).

Imagine that we want to communicate 5 *real-valued* numbers u_0, u_1, u_2, u_3, u_4 from a sender to a receiver, but whenever a number is sent, there is a chance that it gets "erased" and the receiver instead sees a special symbol '?' to indicate that. The following strategy gives a way that we could combat such erasures:

- Interpret u_0, u_1, u_2, u_3, u_4 as *coefficients of a degree-4 polynomial*: $p(x) = u_0 + u_1 x + u_2 x^2 + u_3 x^3 + u_4 x^4$.
- Pick *more than 5* points x to evaluate the polynomial at, say $x \in \{-2, -1.5, \dots, 0, \dots, 1.5, 2\}$, e.g.:



- The encoder sends the corresponding $p(\cdot)$ values, e.g., $p(-2), p(-1.5), \dots, p(2)$ corresponding to the red circles in the above figure.

- The receiver receives a length-9 output, but some of the values are replaced by '?' to mean 'erasure'.

- However, as long as 5 (or more) of them are non-erased, the decoder can recover $p(x)$ – this is because *a degree-d polynomial is uniquely specified by any $d+1$ points.* Since there is a one-to-one correspondence between $p(\cdot)$ and $(u_0, u_1, u_2, u_3, u_4)$, recovering the former means recovering the latter.

Naturally, if some of the $p(\cdot)$ values get *altered* rather than *erased*, then recovery might be more difficult, but mathematically it could still be possible, especially if we increase the number '9' of evaluations above. For example, one could design the decoder based on recovering many polynomials, each corresponding to a different subset of 5 received values. If the number of altered values is small enough, then the polynomial that is recovered most often should be the correct one.

What is perhaps most unrealistic in the above description is the notion of sending and receiving (and performing computation on) infinite-precision real numbers. The idea of Reed-Solomon codes is to use the same idea, but with *finite fields instead of the real number field.* The use of a finite field makes it plausible to implement on a computer.

6.2 Finite Fields

- Mathematically, a field is a set of "numbers" together with addition (+) and multiplication (×) operations that satisfy natural axioms that we mostly won't delve into (e.g., $a \times b = b \times a$). Importantly, there should be a "zero element" (usually denoted by 0) such that $a + 0 = a$, and an "identity element" (usually denoted by 1) such that $a \times 1 = a$. Moreover, every non-zero element a should have some "multiplicative inverse" a^{-1} such that $a \times a^{-1} = 1$, and some "additive inverse" $(-a)$ such that $a + (-a) = 0$.

- The set of real numbers with the usual + and × forms a field. The same goes for the rational numbers. The integers, however, do not form a field, because we can't have $a \times a^{-1} = 1$ for integer-valued a^{-1}.

- In previous sections, we worked with mod-2 arithmetic, in particular using $\oplus$ for addition. Multiplication is trivial in this case: $0 \times a = 0$ and $1 \times a = a$. This is known as the *two-element Galois field* or GF(2). (In this context you can take "Galois" to be synonymous with "finite".)

- More generally, the numbers $\{0, 1, \ldots, p-1\}$ with mod-p arithmetic form a field whenever p is a prime number. For example, if $p = 7$, then we can pick any element (e.g., 3) and find an additive inverse (e.g., 4, since $3 + 4 \equiv 0 \pmod 7$) and a multiplicative inverse (e.g., 5, since $3 \times 5 \equiv 1 \pmod 7$).

- It turns out that the only possible sizes of finite fields are of the form p^m, where p is a prime number and m is a positive integer. Moreover, once such a size is specified, the field is unique (up to relabeling).

- We will proceed using only the above (limited) knowledge, but interested students can refer to Forney's "Introduction to finite fields" lecture notes (Stanford University) for a proper introduction.

6.3 Code Construction

The parameters of a Reed-Solomon code are as follows:

- The message length k and codeword length n;

- The size q of a finite field $\mathbb{F}$; as noted above, we must have $q = p^m$ for some prime p and positive integer m. We further impose the requirement $q \ge n$.

Note that the message of length k is now a sequence of *finite field elements* rather than a sequence of bits, so the total number of messages is q^k rather than 2^k.

Then, similar to what we did with real numbers in Section 6.1, the code construction is as follows:

- Fix a set of n distinct *evaluation points* $a_1, a_2, \ldots, a_n$ in $\mathbb{F}$. Since we assumed $q \ge n$, we can do this in a way such that the a_i are all distinct. There is flexibility in the specific choice of a_i's; a common choice is to let them be of the form $1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$ for suitably-chosen $\alpha \in \mathbb{F}$.

- Given the message symbols $u_0, \ldots, u_{k-1}$ (each in $\mathbb{F}$), define the following polynomial:

$$p(x) = \sum_{i=0}^{k-1} u_i x^i,$$

  where here and subsequently, all arithmetic is done over the finite field $\mathbb{F}$.

- The length-n codeword is then simply $(p(a_1), p(a_2), \ldots, p(a_n))$, i.e., we evaluate the polynomial at the n evaluation points.

This defines a *linear code*, where now the notion of *linear* is a bit more general than the binary case: If $\mathbf{c}$ and $\mathbf{c}'$ are both codewords, then so is $\alpha\mathbf{c} + \beta\mathbf{c}'$ for any $\alpha \in \mathbb{F}$ and $\beta \in \mathbb{F}$. *(Exercise: Try to prove this.)*

Much like in the real-valued case in Section 6.1, dealing with *erasures* is easiest: Since we used a degree-$(k-1)$ polynomial, we can recover the polynomial (and hence the message sequence) as long as k or more symbols remain non-erased. On the other hand, if some of the code symbols are *substituted* by other elements of $\mathbb{F}$ (rather than 'erased'), decoding becomes more difficult.

Having said that, various computationally efficient algorithms are known for decoding Reed-Solomon codes with substitution noise, such as Berlekamp-Welch with time $O(n^3)$, Berlekamp-Massey which is connected to syndrome decoding, and other approaches based on the Fast Fourier Transform. The details of how these decoders work is beyond the scope of this course.

6.4 Properties

In this section, we state and prove some of the most important properties of Reed-Solomon codes. Analogous to the binary case, we define $d_{\min}$ to be the minimum distance (i.e., number of differing symbols) between two codewords.

Claim. The minimum distance of a Reed-Solomon code is $d_{\min} = n - k + 1$.

Proof:

- Like in the binary case, we start with the fact (with a similar proof) that the minimum distance equals the minimum non-zero weight, where "weight" now means "number of non-zero symbols".
- The minimum non-zero weight is equal to $n - \tau$, where τ equals the maximum number of zeros in a non-zero codeword.
- Then, the following fact about polynomials in $\mathbb{R}$ carries over to polynomials on finite fields: *Any non-zero polynomial of degree d has at most d zeros.*
- Since $p(x)$ has degree $k - 1$, this implies $\tau \leq k - 1$ and hence $d_{\min} \geq n - k + 1$. The next claim below is sufficient for deducing that this is not only a lower bound but in fact an equality.

Claim. For any (n, k, q) with $q \geq n$, the minimum distance of a Reed-Solomon code is optimal; no code from $\mathbb{F}^k$ to $\mathbb{F}^n$ can attain a minimum distance higher than $n - k - 1$. *(Note: This is known as the Singleton Bound, or at least a certain form of it)*

Proof:

- Fix an arbitrary code, and suppose that its minimum distance is $d_{\min} = d$.
- If we delete the first $d - 1$ symbols from each codeword, the list of remaining shortened codewords must still all be distinct (due to the minimum distance being d).
- But the number of such codewords is at most q^{n-d+1}, since each symbol takes on one of q values.
- For for a code from $\mathbb{F}^k$ to $\mathbb{F}^n$, the number of codewords is exactly q^k. Comparing to the previous dot point, we get $q^k \leq q^{n-d+1}$, hence $k \leq n - d + 1$, hence $d \leq n - k + 1$.

Discussion.

- So if Reed-Solomon codes have optimal distance properties and can be implemented in a computationally efficient manner, why don't we use them everywhere?
- The answer is that they have a significant disadvantage: *Requiring a large alphabet size ($q \geq n$).*
- The reason this is a disadvantage is that real-world codes typically need to be binary, or need to be some other size (typically much smaller than n) according to the communication channel being used.
- We will next discuss how to "convert" a Reed-Solomon code into a binary code. Unfortunately doing so comes at the cost of losing the property of having the best possible minimum distance, but it can still lead to a very good code.

6.5 From Non-Binary to Binary

A codeword in a Reed-Solomon is a sequence of non-binary characters; for concreteness, let's say that they come from a size-25 set of symbols that we label as $\{A, B, C, \dots, X, Y\}$. But if we want to transmit over a binary channel, or store the codeword on a DVD or QR code (etc.), we need to use binary symbols. How can we use Reed-Solomon codes in such scenarios?

The fact that the message symbols $(u_1, \dots, u_k)$ above are non-binary is not a significant issue; what's more important is making the *codeword symbols* binary. To do so, the basic idea is to *map each non-binary codeword symbol to a binary sequence*, e.g.:

- We could map $A \to 10\dots0$, $B \to 010\dots0$, and so on, up to $Y \to 0\dots01$. This is called *one-hot encoding*. This strategy can be sufficient in certain cases, but it is typically far from ideal, since the code length is blown up by the alphabet size q, leading to a low communication rate.
- If the alphabet size is q, we could map each letter to a unique length-$\lceil \log_2 q \rceil$ binary sequence. This means that the code length only grows according to the *logarithm* of the alphabet size. However, this may not be ideal in terms of error resilience: Within each length-$\lceil \log_2 q \rceil$ block, *any* one of those bits getting flipped amounts to having a symbol substitution in the non-binary code.
- To better balance the goals of error-resilience and not increasing the code length too much, the best approach is usually to *use another (small) error-correcting code to map each non-binary symbol to a binary sequence*. For example, one might use a Hamming-like code that can at least correct a small number of bit flips. This is known as *concatenated coding*, with the Reed-Solomon code being the "outer code" and the binary code being the "inner code".

To name just one specific example, the Justesen code (`https://en.wikipedia.org/wiki/Justesen_code`) is a binary code based on Reed-Solomon coding and concatenation, and it comes with a simple and explicit trade-off between the rate and the minimum distance.

To re-iterate, the conversion of Reed-Solomon codes to binary codes invariably loses the "optimal minimum distance" property stated above, but it often gives something that is still very good.

7 (**Optional**) Other Coding Methods

7.1 Distance-Based Codes

Here we mention just one other famous example beyond Reed-Solomon codes: *Bose-Chaudhuri-Hocquenghem* (BCH) codes are also based on polynomials on finite fields, and they can also be viewed as a generalization of Hamming codes.

Very briefly, the idea is to interpret the message as a polynomial (e.g., in the binary case, interpret 010011 as $x^4 + x + 1$ if the bits are listed from most significant to least significant) and multiply it by some *generator polynomial* $g(x)$ (using finite field arithmetic) to produce another polynomial – its coefficients give the codeword. A suitably-chosen $g(x)$ can ensure good distance separation, and there are also variants that ensure the resulting code is systematic.

BCH codes are well-defined over general finite field sizes (in particular, we have the option of sticking to binary), allow precise control over the minimum distance, and are generally considered easier to decode (using syndrome decoding methods) than Reed-Solomon codes. They have found widespread use in applications such as satellite communication, storage devices, and QR codes.

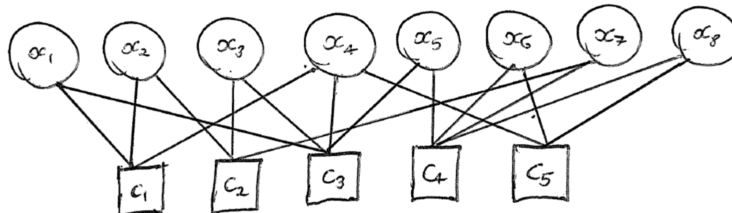
7.2 Capacity-Achieving Codes

History.

- The channel coding theorem was published in 1948. This spawned decades of research on practical coding techniques, but none were seen to come close to achieving capacity for a long time.
- In the early 1990s, *turbo codes* were (empirically) seen to be very close to achieving capacity.
- *Low density parity check* (LDPC) codes were invented in the 1960s, but perhaps due to limited computing power, they remained unused for a long time. After the invention of turbo codes, they were rediscovered and also shown to be nearly capacity-achieving.
- *Polar codes* are a more recent development (2008). They are much nicer to analyze theoretically, in particular to rigorously establish that they are capacity-achieving. Initially they weren't as good in practice, but now they have caught up with (and in same cases even "overtaken") the others and are used extensively.

(Very) Brief overview of LDPC codes.

- As a rough definition, an LDPC code is a (typically non-systematic) linear code such that the check matrix $\mathbf{H}$ contains mostly 0's and relatively few 1's.
- We can picture this via a *factor graph* representation:



Here circles correspond to the n codeword bits ("variable nodes"), and squares correspond to the $n-k$ parity equations ("check nodes"), i.e., if there are no errors, there should be an even number of 1's connected to each square. The *low density* assumption amounts to saying that the above graph is *sparse* (has few edges).

- The reason sparsity / low density is useful is that it permits effective *belief propagation* decoding, in which information is shared between variable nodes and check nodes until accurate beliefs (posterior probability estimates) on the transmitted bits are obtained.
- Strictly speaking, this is only guaranteed to end up with accurate posterior estimates when the underlying graph has no cycles (i.e., one can never follow any path of edges and end up back at the starting node). Although sparse graphs like the ones above do have cycles, they have *very few short cycles*, and for practical purposes this is often nearly as good as having no cycles.
- For a much better introduction to LDPC codes, see Chapter 47 of MacKay's book (see also `https://www.youtube.com/watch?v=RWUxtGh-guY` for a brief introduction in Layman's terms).

(Very) Brief overview of polar codes.

- The rough idea of polar codes is as follows:
  - First consider 2 uses of the channel, $(X_1, X_2) \to (Y_1, Y_2)$.
  - The idea: Apply some pre-processing to (X_1, X_2) to create a related channel $(\widetilde{X}_1, \widetilde{X}_2) \to (Y_1, Y_2)$ such that (i) $I(X_1, X_2; Y_1, Y_2) = I(\widetilde{X}_1, \widetilde{X}_2; Y_1, Y_2)$ (no information is lost); (ii) Comparing the new channel to the original channel, one "channel use" is less noisy, and the other is more noisy.
  - Repeat this procedure several times, moving from 2 to 4 uses of the channel, then 8 uses of the channel, then 16, etc., with many "channel uses" getting less and less noisy, and the rest getting more and more noisy.
  - *For a symmetric binary channel with capacity $C \in (0, 1)$, repeating this procedure eventually leads to a fraction C of "near-perfect" channel uses, and a fraction $1 - C$ of "near-useless" channel uses.*
  - Perfect and useless channels are very easy to handle! If it were truly perfect, we could just send a bit and receive it without noise. If it were truly useless, we would simply avoid using it at all. Polar codes do something similar, sending information only over the "near-perfect" uses.
- Extensions to non-binary and non-symmetric channels were established later.
- For a much better introduction to polar codes, see the following lecture by Emre Telatar: `https://www.youtube.com/watch?v=VhyoZSB9g0w`