Error Correcting Codes - A Deep Dive

Isak Oswald
s225375329
Deakin University

May 22, 2025

Contents

1 Introduction

1.1 Background Knowledge required

- Basics of Finite Fields, specifically $\mathbb{F}_2$.
- Basic Linear Algebra such as vectors, matrices, and matrix manipulation.
- Modular Arithmetic.
- Binary (Base two system).
- Boolean logic (AND, XOR, NOT) operations.

1.2 Motivation

In computing, telecommunications, information theory, and coding theory, Error-Correcting-Code (ECC) is used to control and handle a finite amount of errors [1]. ECC is often used over unreliable or noisy communications such as channels between satellites and Earth. ECC not only provides a way to

identify that an error has occurred during transmission but also provides a way to correct that data, avoiding a potentially costly or lengthy resubmission of that data. ECC is not only used for space communication but also is used within more common noisy channels such as Wifi and other long latency connections such as cellular networks and modems. ECC is also used within computer random-access-memory (RAM) to ensure data integrity and smooth operations [7].

2 Finite Fields

Before we begin, we must understand finite fields as this report is mostly looking at $\mathbb{F}_2$, we will look at $\mathbb{F}_n$ in a more advanced section after we understand basic codes using $\mathbb{F}_2$. To put it simply, a Field denoted as $\mathbb{F}_k$, also known as a Galois Field ($GF(k)$) is a mathematical structure that holds a finite number of elements, like a subsection of $\mathbb{N}$ for example. Importantly, finite fields only exist for orders $k = k^n$ where k is prime and $n \geq 1$. For **prime orders** (eg. $\mathbb{F}_2$, $\mathbb{F}_3$, $\mathbb{F}_7$) the field elements are modulo p. On the other hand, for non-prime orders, elements are not integers but are constructed using polynomials over $\mathbb{F}_k$, for example, $\mathbb{F}_4$ can be built by defining polynomials over $\mathbb{F}_2$ modulo an irreducible polynomial such as $x^2 + x + 1$ resulting in elements $\{0, 1, x, x + 1\}$. A finite field has elements exactly as a set does in set theory, the k in $\mathbb{F}_k$ represents the number of elements in that field, for example, in $\mathbb{F}_2$, its elements would be a set $\{0, 1\}$. In this report, we will mostly focus on prime fields such as $\mathbb{F}_2$ where operations abide by binary logic, however, we will look at some polynomials in the future. [9]

A finite field, at some level, behaves like real numbers in terms of algebraic manipulation such as addition, multiplication, associativity, commutativity, etc. In mathematics, finite fields can hold elements, meaning they do not explicitly have to be numbers, however, since we are working binary, for this report we will be working with both 0 and 1 ($\mathbb{F}_2$.). It is a requirement to understand finite fields, specifically $\mathbb{F}_2$ as they make the operations we conduct on elements "make sense" as they provide an algebraic structure for reliable operations. From an application point of view $\mathbb{F}_2$ models digital operations on bits meaning turning them on or off as a result of the operations we do on them. In essence, finite fields are a little more than just "numbers on

which we can do operations" as finite fields can contain objects as a set can. However, in terms of ECCs, we will be operating over files like $\mathbb{F}_2$,$\mathbb{F}_7$, $\mathbb{F}_3$, etc. which hold numerical elements.

The logical operations that we conduct over finite fields can be defined for example, $A \wedge B = (A \oplus B) \oplus (A \wedge B)$. For exactly this reason, $\mathbb{F}_2$ is widely used for operations on bits as every program is a sequence of operations on large vectors in $\mathbb{F}_2$. Considering this, finite fields are essential in terms of finite fields as they allow us to program the encoding and error-correcting processes. In terms of operations we can compute a finite field, since we are working with binary, consider the following truth tables for XOR and computations.

XOR and AND Truth Tables

A	**B**	**A** $\oplus$ **B** (XOR)	**A** $\wedge$ **B** (AND)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

As you may have noticed, in $\mathbb{F}_2$, these perfectly match the logic of multiplication and addition operations which we can use to build our relationship between data bits and redundancy bits. Before we use $\mathbb{F}_2$, we must make sure that its axioms (algebraic operations) hold.

1. **Closure**

   - *Addition*: All results of $a \oplus b$ are in $\mathbb{F}_2$:

     $$0 \oplus 0 = 0, \quad 0 \oplus 1 = 1, \quad 1 \oplus 0 = 1, \quad 1 \oplus 1 = 0.$$

   - *Multiplication*: All results of $a \wedge b$ are in $\mathbb{F}_2$:

     $$0 \wedge 0 = 0, \quad 0 \wedge 1 = 0, \quad 1 \wedge 0 = 0, \quad 1 \wedge 1 = 1.$$

2. **Associativity**:

- *Addition*: For all $a, b, c \in \mathbb{F}_2$,

$$(a \oplus b) \oplus c = a \oplus (b \oplus c).$$

  Example: $(1 \oplus 0) \oplus 1 = 1 \oplus 1 = 0 = 1 \oplus (0 \oplus 1)$.

- *Multiplication*: For all $a, b, c \in \mathbb{F}_2$,

$$(a \wedge b) \wedge c = a \wedge (b \wedge c).$$

  Example: $(1 \wedge 1) \wedge 0 = 1 \wedge 0 = 0 = 1 \wedge (1 \wedge 0)$.

3. **Commutativity**:

   - *Addition*: $a \oplus b = b \oplus a$. Example: $1 \oplus 0 = 0 \oplus 1 = 1$.
   - *Multiplication*: $a \wedge b = b \wedge a$. Example: $1 \wedge 0 = 0 \wedge 1 = 0$.

4. **Distributivity**:

   - *Multiplication distributes over addition*:

$$a \wedge (b \oplus c) = (a \wedge b) \oplus (a \wedge c).$$

     Example: $1 \wedge (0 \oplus 1) = 1 \wedge 1 = 1$, and $(1 \wedge 0) \oplus (1 \wedge 1) = 0 \oplus 1 = 1$.

5. **Identities**:

   - *Additive Identity*: 0 satisfies $a \oplus 0 = a \forall a$. Example: $1 \oplus 0 = 1$.
   - *Multiplicative Identity*: 1 satisfies $a \wedge 1 = a, \forall a \in \mathbb{F}_2$. For example: $1 \wedge 1 = 1$.

6. **Inverses**:

   - *Additive*: Every element has an additive inverse in $\mathbb{F}_2$:

$$0 \oplus 0 = 0 \implies 0^{-1} = 0, \quad 1 \oplus 1 = 0 \implies 1^{-1} = 1.$$

   - *Multiplicative*: Every non-zero element has an mulplicative inverse, in our case, we only have one which is element 1:

$$1 \wedge 1 = 1 \implies 1^{-1} = 1.$$

Since all field axioms hold, $\mathbb{F}_2$ is a field.

$\mathbb{F}_2$ is perfect if we want to operate over only 0's and 1's, however, if we want to operate over entire symbols such as bytes, we can use a field $\mathbb{F}_k$ and polynomials to encode and detect errors at a symbol (or byte in the case) level which we will explore later. Now that we know how finite fields work in terms of error correction, we can look at how we can use them to encode redundancy bits by creating relationships between the bits in our message.

3 Repetition codes

3.1 Introduction

Repetition codes are the simplest form of Error Correction Codes and are the first thing you should understand before diving into more complex topics such as linear-polynomial ECCs. Repetition code essentially involves sending three identical sets of information and when one is damaged due to noise the other 2 will "disagree" with the incorrect copys representation. [3].

3.2 Standard digital images

If we assume that the data we are sending are pixels of an image, every pixel in a monochrome picture is representable in a range from 0-255, with 0 being pure black and 255 being pure white. Since these images are made up of many pixels, or more specifically, lists of numbers, we can model them as binary. For example, if we were communicating some image m we would receive three copies of an image, assuming these are $A_{i,j}$ where i is the image copy, and j is the bit position/s of copy i. For example, $A_{1,3}$ would be copy one, and the bit at index 3 within that copy. Assume that three copies of a random **pixel** in our image were represented as:

$$A_{1,j} = 110|010|000$$
$$A_{2,j} = 110|010|000$$
$$A_{3,j} = 110|110|000$$

During transmission, one of the zeros was corrupted in copy $A_{3,3}$, this means that both $A_{1,3}, A_{2,3}$ "disagree" with $A_{3,3}$'s representation of that bit in the pixel. This means that we can restore the incorrect bit in A_3 to align with

what both A_1 and A_2 have presented. This is the simplest form of ECC and consequently, is the most inefficient due to a large overhead in sending multiple copies.

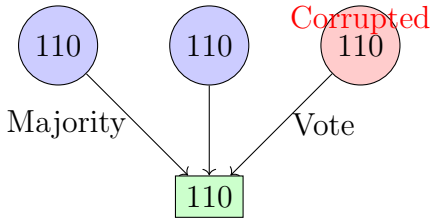


Figure 1: Majority voting correcting a corrupted bit in repetition codes.

3.2.1 Correctness and Limitations

Claim: A 3-copy repetition code can detect and correct any single error per block.

Proof: Let A_1, A_2, and A_3 be the three versions of a block transmitted.

- **If no errors occur:** $A_1 = A_2 = A_3$, there is no contradiction and therefore no action is necessary.
- **If one error occurs:** two blocks will be identical and one will differ. Majority voting will select the correct (non-corrupted) value.
- **If two or more errors occur:** the majority vote may select an incorrect value. Error correction fails.

Hence, the 3-copy repetition code:

- Detects and corrects **up to one error** per block.
- **Fails** to guarantee correctness when **two or more errors** occur in the same block.

3.3 Considerations

3.3.1 Efficiency

Let the original message length be n amount of bits. With a 3-copy repetition, the transmitted length becomes $3n$. Therefore, the transmission efficiency η is:

$$\eta = \frac{n}{3n} = \frac{1}{3} \approx 33.3\%$$

3.3.2 Probabilities

Not only is the efficiency of error-correcting methods important, but also its probability that we will receive an encoded message that we can decode, meaning, "What is the probability that we receive a message that can be corrected?".
To decode a bit correctly using 3-repetition, **at least** two out of three bits must be correct. This can happen in two different cases:

1. All three bits are correct: p^3.
2. Exactly 2 bits are correct: $3p^2(1-p)$, as there are three ways to pick the one bit that is incorrect.

In saying this, that means that:

$$P_{correct} = p^3 + 3p^2(1-p)$$

For example, if $p = 0.9$, then:

$$P_{\text{correct}} = (0.9)^3 + 3 \cdot (0.9)^2 \cdot (0.1) = 0.729 + 3 \cdot 0.81 \cdot 0.1 = 0.729 + 0.243 = 0.972$$

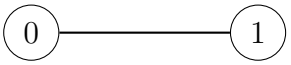
Even though each bit only has a 90% chance of being received correctly, using 3-repetition codes increases the decoding success rate to **97.2%** at the cost of reducing bandwidth efficiency to 33.3%.

Dimensional Spaces

Consider the message m split up into one bit chunks.

$$0|0|1|0|1$$

As binary is base 2, there is really only two options for each of the of the one bit chunks, either zero or one which will be denoted as m_i.

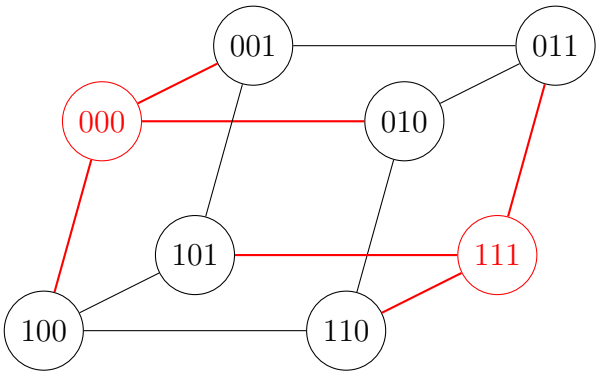


Think of each bit living in a one-dimensional space, where they can either be one or zero or left or right on a plane. Since there are no intermediate values between 1 and 0 (there are no solutions between these points) it means that it is impossible to correct code, as receiving either a zero or a one may be a valid, or corrupted response.

Now, let's consider that m is using the "best two of of three" which means there are three identical bits being sent, or $3m$. If we keep the message split into chunks we will still be left with:

$$000|000|111|000|111 \text{ (which is the same as } 0|0|1|0|1)$$

Now let's consider how each of these bits can be represented.

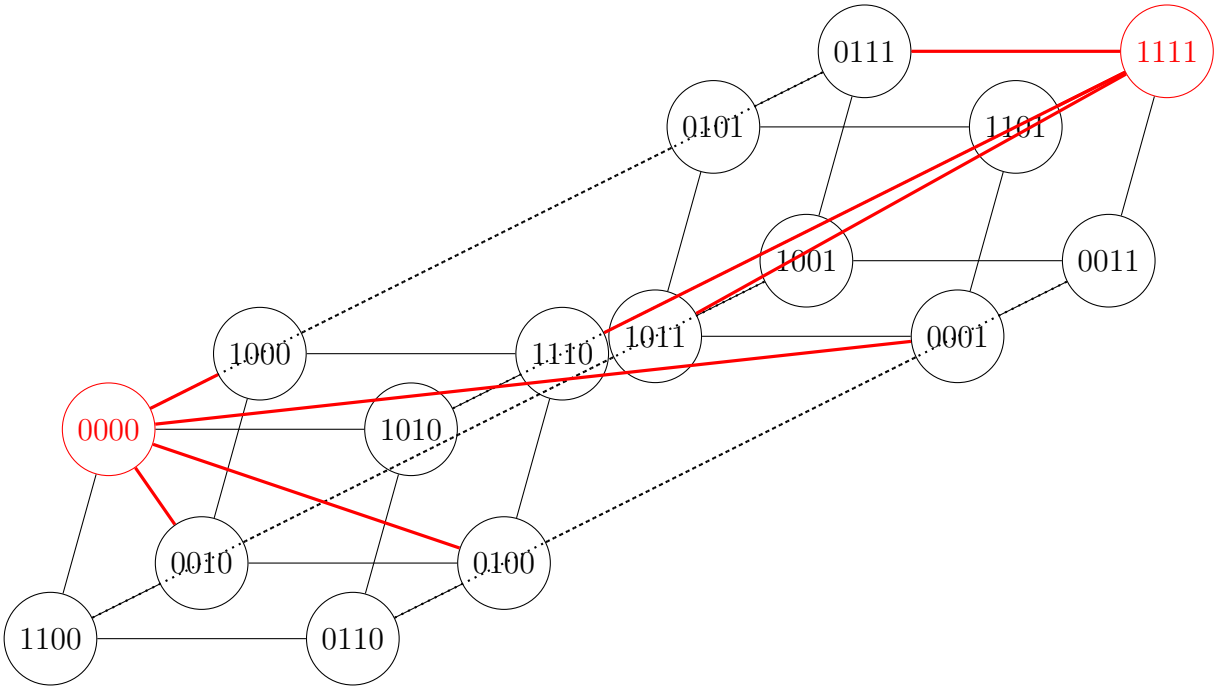


There are now some intermediate values between 0 and 1 (or 000 and 111) which means that we have room to identify errors. The red nodes represent our 0's and 1's (or "valid" bits), whereas the grey nodes consider any bits that have been corrupted during transmission. As we are simply duplicating

m by a factor of 3, a bit representation of 001 for example is not possible, only 000 and 111. We can also see that from any node on the graph, there is always a way to trace it back to either a 0 or a 1 depending on which path is shorter. This turns error correction codes from a more abstract idea to a graph theory problem in a way that we are also dealing with edges (or paths) to "correct" bits.

Let each edge of the cube represent a change in any single bit. So for example, the edge connecting nodes 000 to 100 represents a change in the m_1 bit.

This means that along the x direction, we can see that m_2 is changing from a zero to a one. If we visualize these bits in 3-dimensional-like space, it becomes a lot more interesting in terms of error correction. If we take a message encoded in the "best 2 of 3" method and transmit it, if any of the data is corrupted, all we need to do is find the corrupted bit on this cube and then find the nearest valid word on the cube (or shortest path). This means that if we receive some word chunk $m_i = 0101$ we can locate that on the cube and take the shortest path to the valid bit representation. This method of thinking allows us to create a "map" by finding the closest real value from any intermediate between 0 and 1 in binary.



Now consider the change between nodes from 3-dimensional to 4-dimensional space, there are still only two representations of valid bits in the message, 0000 and 1111. During the transition from 3D to 4D, we see that more nodes can be incorrect, however, there are still only two representations of an un-affected value. These grey or "incorrect" nodes are the key to ECC, the higher the dimension we have, the more incorrect values we can identify and compute as there are more "incorrect" nodes that still have some path to the correct binary representation as all of these are cyclic graphs.

If we use this thinking, this means that the process of encoding a message into an error-correcting code is just the process of moving data into a higher dimensional space. To travel between the spaces (from 1D to 2D for example) we use something called a generator matrix, but before we look at this, let's understand what a matrix (G) is doing under the hood.

4 Introduction to Hamming Codes and Relationships between bits

A hamming code, represented in the format $H(x, y)$ means that we are taking some message length y and transforming it into a codeword of size x. The original message is then transformed from size x to size y, retaining the original message and adding something called **redundancy** bits which are the bits that allow us to detect and correct errors.

The way that the redundancy bits allow us to detect errors is through constructing them based on relations, like a set of systems of equations. By solving these linear equations, we can determine if the system holds (if there is no error present), or in which bit an error occurs.

Say we have three variables, k, p, and n in which k represents the data bits, p represents the parity (or redundancy bits) which will hold the relationships between bits, and n which is our new codeword length. For example in H(3,2):

$$\textbf{Data Bits } (k = 2)\text{: } d_1, d_2$$
$$\textbf{Parity Bits } (p)\text{: } p = d_1 \oplus d_2$$
$$\textbf{Codeword } (n = 3)\text{: } [d_1, d_2, p]$$

As seen, the relationship between the data bits and the parity bits is created based upon a XOR operation. This relationship does not have to be derived by a XOR operation, however, since we are working within $\mathbb{F}_2$, it makes it a simple and approachable choice.

5 Hamming (7,4) code

The previous method takes a 1 bit-message and forms a three bit codeword, such that:

$$0 \to 000$$
$$1 \to 111$$

On the other hand, H(7,4) takes a message of 4 bits, and forms a 7-bit codeword, improving the rate at which we are able to transmit data while still allowing for redundancy:

$$1101 \to 1101001$$
$$0010 \to 0010011$$

[10] Now, let the original message be n bits in size. With the Hamming $(7, 4)$ code, the transmitted length becomes:

$$\frac{7}{4}n$$

Therefore, the transmission efficiency η is:

$$\eta = \frac{n}{\frac{7}{4}n} = \frac{4}{7} \approx 57.14\%$$

This is significantly more efficient than the 3-copy repetition code, while still allowing the detection and correction of single-bit errors.

Now assume we have some arbitrary message m of size n-bits, split into n_t chunks of 7 (our 7 bits of data.) Each chunk has three redundancy bits to align with the Hamming (7,4) code which means 4 will be pure data, and the remaining 3 will be used for redundancy, error correction, and error detection.

Let's assume that our message m was split up into 3 chunks to represent our binary data, we can further abstract this away by representing each bit of n_t data as a, b, c, d, and its redundancy bits as z, y, z. For example:

$$n_1 = abcd\ xyz$$
$$n_2 = abcd\ xyz$$
$$n_3 = abcd\ xyz$$
$$n_4 = abcd\ xyz$$

Since z, y and z are each chunk n_t's bits, we need a way of determining what these bits will be. This can be done as:

$$x = a \oplus b \oplus d$$
$$y = a \oplus c \oplus d$$
$$x = b \oplus c \oplus d$$

As we discussed perviously, we can do operations such as this as we are working with $\mathbb{F}_2$. In our case, the truth table of XOR represents multiplication on $\mathbb{F}_2$ in the same way it does within $\mathbb{N}$. Considering this, we compute the four bit chunk to build our relationship between data bits and x, y, z (our redundancy bits).

$$1101\ xyz \rightarrow 1101\ 100$$

5.1 Error Correction

For example, say there is some encoded message $abcd\ zyz$ such that the message we received was:

$$\underbrace{1001}_{\textbf{Data}}\ \underbrace{010}_{\textbf{Parity Bits}}$$

Where c has been corrupted. (but we don't know this yet)

Since we don't know its c that is corrupted we start by assuming that some bit i has been corrupted (when receiving you don't know if a message has been corrupted and/or where); by examining all of the parity bits that have failed (x, y, z), we are able to rule out either parity bit x, y, or z and determine which one corresponds to the corrupted bit. By calculating the parity checks again, the receiver of the message m is able to determine if either x, y, or z was corrupted based on the pattern of incorrect parity checks. Think of this as solving a system of equations, where instead of solving for unknown variables, we are identifying the location of the error.

Continuing on with our example, since $x = 1 \oplus 0 \oplus 1 = 0$, it seems that all a, b, and d check out in terms of our example, meaning cannot deduce an error at this point. $y = 1 \oplus 0 \oplus 1 \neq 0$, which means there is an error in c. To be thorough, let's also check z. $z = 0 \oplus 0 \oplus 1 \neq 1$, therefore the error must lie in c. Since x validated a, b, and d and y presented an error which must be in c as x checked out, this means that since z also shares that same c, it must contain corruption. If you revert $c \rightarrow 1$, we find that the relationships hold once again. We have successfully detected and corrected our first bit.

Since each parity bit checks a specific set of bits in the codeword for errors, in one of the parity checks (XOR operations), there may be a mismatch indicating that there is an error (in our case it was c). The way that this happens is through the relationship between the redundancy bits and how they have been created, through XOR operations as discussed previously [10].

5.1.1 Can we correct multiple bits?

As hamming codes operate over bits, not symbols such as bytes, we are still unable to correct multiple single-bit errors. $H(7,4)$ does not use polynomials (which allow us to correct on a symbol level) as they still use the "systems of equations" through the XOR operations approach. This means that $H(7,4)$ can detect up to two-bit errors, but only correct one one-bit error.

Since $H(7,4)$ adds **3 parity bits** to 4 bits of data, we are left with **7 bits in total**. Because of the way each bit contains data about the other bits and whether they are correct or incorrect if two or more bits are corrupted, they may go undetected or be completely miscorrected. Going back to our systems of equations, if we had some variable x that relied on z and y, changing y would change the values represented in both equations in the systems of equations, affecting the result of x.

6 Generator Matrices

Now we have an understanding of how we encode messages and that at the heart of Error-Correcting Codes, we are just creating relationships between our message m's bits bits to form our parity bits which we can detect errors through a system of equations. We can formalize this concept through the use of a Generator Matrix. As the name suggests, the Generator Matrix denoted by G gives us a template to create these relationships and therefore parity bits. When using G, we are still creating these same XOR relationships between a, b, c and d. Under the hood, there is no difference in the way we encode messages, just the way we choose to do it.

Continuing from our example, say we have the message we want to encode:

$$\text{Original message } m = [1101]$$

We have our Generator Matrix G which looks like:

$$\mathbf{G} = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right]$$

Denoted by the vertical line, we have the Identity Matrix on the left, and then the recipe to encode our message with parity bits on the right, meaning "How does the bit (denoted by the identity matrix) contribute to x, y, z (the construction of the parity bits). On the left, looking at the columns, we have 1000 which refers to the first bit of our message m. The rows in the Identity Matrix are just the individual bit locations of our message, so each column represents: Bit 1, Bit 2, Bit 3, and Bit 4. When doing this, we are not changing anything about these bits themselves, just building relationships out of them. On the right side of the vertical line, we have the recipe for creating the relationships as discussed previously, if we look at the identity matrix row one (1000) and row 1 of the encoding matrix (1101) it means that to encode bit 1, we have the **exact** same relationship between a, b, c and d as we looked at previously.

For example, if we look at column 1 of the Identity Matrix and the relationship to build we can write something like

$$1000|110$$

With the left side being the location, and the right being the recipe to create the relationship. So bit 1 (which is 1 in our original message) will contribute to the construction of parity bits 1 and 2 (x and y), but not 3 (z). This means:

$$x = 1 \oplus b \oplus d$$
$$y = 1 \oplus c \oplus d$$
$$\textbf{But not } z$$

If we continue the process, for the rest of the rows in the matrix, for example, the second row, we get:

$$x = 1 \oplus 1 \oplus d$$
$$z = 1 \oplus c \oplus d$$
$$\textbf{But not } y$$

As you can see, we are just "continuing" on from our first computation. We knew that x used a (bit 1), b (bit 2), and c(bit 3) and we already placed a there from our first calculation, and then the second row of the party matrix tells us bit b (which is 1) is also used in x, so we just continue on. Let's do one more example, so we can fully solve for x parity bit:

$$x = 1 \oplus 1 \oplus 1$$
$$y = 1 \oplus c \oplus 1$$
$$z = 1 \oplus c \oplus 1$$
$$\textbf{We can see } d \textbf{ is used in all } x, y, z$$

Now that we have completed finding how the parity bit x is constructed from a, b, and d, it should result in 1, as $1 \oplus 1 \oplus 1 = 1$.

During the process, we are just creating the relation for the parity bits, but in practice, we take the dot product of m and G. Even though we are now using the original message, nothing is done to the original message itself (as the Identity matrix is just a representation of the location or bit position).

$$e = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Figure 2: Encoding a message $m = [1101]$ using generator matrix G.

Which results in:

$$
\begin{aligned}
e_1 &= (1 \cdot 1) \oplus (1 \cdot 0) \oplus (0 \cdot 0) \oplus (1 \cdot 0) = 1 \\
e_2 &= (1 \cdot 0) \oplus (1 \cdot 1) \oplus (0 \cdot 0) \oplus (1 \cdot 0) = 1 \\
e_3 &= (1 \cdot 0) \oplus (1 \cdot 0) \oplus (0 \cdot 1) \oplus (1 \cdot 0) = 0 \\
e_4 &= (1 \cdot 0) \oplus (1 \cdot 0) \oplus (0 \cdot 0) \oplus (1 \cdot 1) = 1 \\
e_5 &= (1 \cdot 1) \oplus (1 \cdot 1) \oplus (0 \cdot 0) \oplus (1 \cdot 1) = 1 \\
e_6 &= (1 \cdot 1) \oplus (1 \cdot 0) \oplus (0 \cdot 1) \oplus (1 \cdot 1) = 0 \\
e_7 &= (1 \cdot 0) \oplus (1 \cdot 1) \oplus (0 \cdot 1) \oplus (1 \cdot 1) = 0
\end{aligned}
$$

Notice how e_1 through e_4 are the same as our original message m, this is because the identity matrix is referring to how the additional parity bits will be added to the original message, we are **not** modifying the values, only "pointing" to the bit we will be creating the relationship for. In doing these steps, we have done:

And therefore, our final encoded message with our parity bits is:

$$e = [1101100]$$

As you can see, like we did by hand, our x parity bit is indeed one.

One common question with working with generator matrices is why use them? The reason that we decided to use matrices is that it is used like a map or recipe to encode a message. We have the final product of our bread (e), but we need to know what to add to the flour (m) to get our final product of bread. As stated before, the matrices are just a template and a systematic approach to adding parity bits to a message m.

We could use any way to create these relationships between data bits and parity bits, however, a matrix is not only a template or a "map" for doing this, but it can also be converted into efficient computer codes for algorithms that will do this for us. Considering what we have learned, we can change the right side of G to create our way of encoding messages, we could do this any way we want, however, we would also need a Matrix to "undo" this which is called the Parity Check Matrix, detecting and correcting errors.

6.1 Why does the Generator matrix G produce a new encoded message e with parity bits?

As we discussed previously, we will start with some message vector m, such that $m \in \mathbb{F}_2^k$ where k is the length of the vector m in the finite field $\mathbb{F}_2$. Our generator matrix G is a fixed $k \times n$ matrix over the finite field $\mathbb{F}_2$. n represents the new length of the message and $n \geq k$ as we need additional room for our ECC embedding. Output e is the product of the message m and the matrix G.

The multiplication $m \times G$ results in a new vector e, such that $e \in \mathbb{F}_2^n$ where the elements of e are obtained through the dot product of m with the columns of G. We are multiplying each m_i with the corresponding columns of G. Since m is a $1 \times k$ vector (where k is the length of the message) and our generator matrix G is $k \times n$, this must mean that mG is also a $1 \times n$ vector, thus, $e = e_0, e_1 \ldots, e_{n-1}$ lives in $\mathbb{F}_2^n$.

$$e = \begin{bmatrix} m_1 & m_2 & \ldots & m_k \end{bmatrix} \times \begin{bmatrix} g_{11} & g_{12} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & g_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k1} & g_{k2} & \cdots & g_{kn} \end{bmatrix}$$

Each element e_i of the new vector e if formed on the products of the m_i element and its corresponding G_{kn} element modulus two, as since $m_i \in \mathbb{F}_2$, e_i also needs to be $\in \mathbb{F}_2^k$. This means that:

$$e_i = (m_1 \wedge g_{1i}) \oplus (m_2 \wedge g_{2i}) \oplus \cdots \oplus (m_k \wedge g_{ki})$$

Notice here that we are leaving out the $\pmod 2$ as we are working within $\mathbb{F}_2$. Because G is a $k \times n$ matrix, and our message is a single-level vector ($1 \times k$, the result of e will be in the same one-row vector of m. We are not changing the structure during the process of $m \rightarrow e$, we are just using the matrix G as a template for how we calculate the encoded message e. Matrix G has additional rows, which means that e will have more characters than m, as space has been allocated for error-correcting bits.

By using the computation $e = m \times G$, we are ensuring that we are not changing the structure during the transformation from $m \rightarrow g$, meaning

that each e_i of e is just some product of m_i meaning they can be easily computed back to reveal m (when using a parity check matrix).

If you and the other communicator were to agree on a format for e, it would work but only in simple cases. If you agreed on the first 10 bits of e being the message, the additional 3 are the error bits, and the error bits were calculated through $m_i \oplus m_{i+1}$ for example, it would get confusing if the message was large, or there were multiple errors present. When using matrix G, we provide a way that is computationally efficient to generate these encoded messages e.

7 Parity Check Matrix

Imagine you are receiving some encoded message e (through the use of encoding with G) we need a way actually to identify and correct the corrupted bits. For this, we will use yet another matrix called the Parity Check Matrix H [6].
For every valid c, it has some parity matrix H such that:

$$He^T = 0$$

This essentially means that there is no error in such message e.

Each column of H is the binary representation of the position number of bits 1-7 (assuming hamming (7,4) is being used), written top to bottom. If $He^t \neq 0$ it will give us the location in which an error is present (the bit position).

If we have or Generator Matrix G from the previous section:

$$\mathbf{G} = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right]$$

And now our parity check matrix H to "undo" G:

$$H = \left[\begin{array}{cccc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right]$$

As you can see, G and H look quite similar, on the Identity side of G, we can see the bit positions and on the right, we see which bits contribute to the construction of the parity bits z, y, z, for example, we know that row 1 (bit a) is used in parity bits x, and y. In H we have these relationship equations (101, 111, 011, etc). On the right-hand side of H, we have the transpose of the identity matrix in G which we know corresponds to the bits of data in the message. In saying this, we can say that H is kind of a flip of G, similar to how division undoes multiplication.

7.1 The structure of H

When working with G, we found that matrices are compact ways of storing data as we can read them vertically (inputs) and also horizontally (outputs). Regarding H, we can see that on the right side (the last three columns) is the identity matrix, similar to what we had in G (but on the right side). Each of these columns in H's identity matrix corresponds to a parity bit, either x, y, or z. On the left side, we have the exact contents we had in G (the non-identity side) but flipped horizontally to vertically, this side is the transpose of the parity sub-matrix P^T from G. Each of these columns defines how the data bits (a, b, c, d) contribute to the creation of the parity bits, or how they participate in the parity check equations (the XOR relationships). We saw exactly this in G, but instead of it being an output, it is now an input which makes sense as we are trying to "undo" this in a sense, meaning we are trying to find if the equations hold. If we now look at the rows of H, they provide a parity check equation which should be 0 if no errors have occurred ($He^t = 0$). For example, in the first row, we see

$$a \oplus b \oplus d \oplus x = 0$$

This gives us a way to check if anything has gone wrong, we are checking if this equation holds, this will be one if and only if there is an error in this equation has occurred. Considering this, we have three rows, so we are doing three parity check equations to verify the integrity of the encoded message

we received which is exactly why we have 3-digit syndrome, it is simply the output of three equations.

7.2 How Parity Equations Detect Errors

As we did in the previous sections, let's say that our message is arbitrary values instead of bits like 10112..... Lets now consider the received message were represented as:

$$[a', b', c', d', x', y', z']$$

If the message was transmitted without any errors present, then each received bit will equal the original, for example, $a' = a$, $b' = b$, and so on.

Continuing on from the previous section, lets look at the first row of H which corresponds to the following parity equation:

$$a' \oplus c' \oplus d' \oplus x' = 0$$

Now, recall that each parity bit is created based on XOR relationships and discussed when looking at G, for example, x is constructed as:

$$x = a \oplus c \oplus d$$

If x **was not** corrupted during transmission, then we know $x' = x$, substituting in this we get:

$$a' \oplus c' \oplus d' \oplus (a \oplus c \oplus d) = (a' \oplus a) \oplus (c' \oplus c) \oplus (d' \oplus d)$$

Thanks to the properties of XOR relationship, this expression will equals 0 if the **number** of errors is even. Just say that a and c are corrupted, meaning $a' \neq s$ and $c' \neq c$ we get $1 \oplus 1 \oplus 0$ which equals zero. This means that if they are all correct they are also even. (as $1 \oplus 0$ is 1).

Similarly, if a, c, d are correct, but x (our parity bit) was corrupted (meaning $x' \neq x$), then the parity check becomes:

$$x' \oplus x = 1$$

This logic holds for each row of H, meaning every row detects specific types of parity violations as touched on previously — together, these equations form a **syndrome** which indicates where an error may of occurred.

7.3 Using the Matrix

Now we understand how H and G work and constructed, we can finally use them to encode messages (using G) and correct and detect errors with H using dot products. Up until this stage, it seems that H is created based on G but that is actually not the standard way of doing things in terms of Hamming Codes. In these codes, we construct H first (building the parity check equations) and then create G, meaning that G is actually derived from H and not the other way around (more on this in the upcoming sections). The reason for using matrices is not only to be compact, but also allows for very fast computation within computers and matrices are simply a grid of numbers and our H and G follow some specific rules meaning is very easy to give this information to a computer.

7.3.1 What if we receive the message with no errors?

If the message is received without any errors, the syndrome results in 0 indicating that no error has occurred (the output of our three-row equations). As we discussed previously, we can multiply all rows of H to get our syndrome. Suppose we received the original message we created previously with G as:

$$e = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

We can then compute the syndrome with the steps we learned before:

$$s = He^T = \left[\begin{array}{cccc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} (1 \cdot 1) \oplus (1 \cdot 1) \oplus (0 \cdot 0) \oplus (1 \cdot 1) \\ \oplus (1 \cdot 1) \oplus (0 \cdot 0) \oplus (0 \cdot 0) \\ (1 \cdot 1) \oplus (0 \cdot 1) \oplus (1 \cdot 0) \oplus (1 \cdot 1) \\ \oplus (0 \cdot 1) \oplus (1 \cdot 0) \oplus (0 \cdot 0) \\ (0 \cdot 1) \oplus (1 \cdot 1) \oplus (1 \cdot 0) \oplus (1 \cdot 1) \\ \oplus (0 \cdot 1) \oplus (0 \cdot 0) \oplus (1 \cdot 0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

As you can see, we are multiplying each row of H by our encoded message, doing the equations we discussed before but on all three rows. After

calculation, the syndrome is 0 for all three of our rows($He^T = 0$). This means that there are **no** locations that have an error. In doing this, we have also shown that H is correct—if H were to produce a non-zero syndrome in this case, we would know that our parity check matrix does not reflect G.

7.3.2 What if we received an error instead?

Now, let's assume we receive an error after the transmission. Let's say that our codeword changes from:

$$e = \begin{bmatrix}1 & 1 & 0 & 1 & 1 & 0 & 0\end{bmatrix} \text{ (which is correct)}$$

to:

$$e = \begin{bmatrix}1 & 1 & 0 & 1 & 1 & 0 & 1\end{bmatrix} \text{ (last bit flipped)}$$

Even though we now have corrupted data, we can still use H the same way as we did when we had no errors. Think of each row of H as a parity equation, if one bit is wrong in any equation, it violates **multiple** equations, like in a system of equations. The pattern in which the equations are violated points uniquely to the bad bit. This is exactly like a system of equations, except for solving for unknowns, we are checking which equations hold or which don't.

$$s = He^T = \left[\begin{array}{cccc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix}1\\1\\0\\1\\1\\0\\1\end{bmatrix} = \begin{bmatrix}(1\cdot 1)\oplus(1\cdot 1)\oplus(0\cdot 0)\oplus(1\cdot 1) \\ \oplus(1\cdot 1)\oplus(0\cdot 0)\oplus(0\cdot 1) \\ (1\cdot 1)\oplus(0\cdot 1)\oplus(1\cdot 0)\oplus(1\cdot 1) \\ \oplus(0\cdot 1)\oplus(1\cdot 0)\oplus(0\cdot 1) \\ (0\cdot 1)\oplus(1\cdot 1)\oplus(1\cdot 0)\oplus(1\cdot 1) \\ \oplus(0\cdot 1)\oplus(0\cdot 0)\oplus(1\cdot 1)\end{bmatrix} = \begin{bmatrix}1\\1\\1\end{bmatrix}$$

As we can see, our syndrome after flipping the last bit is:

$$s = \begin{bmatrix}1\\1\\1\end{bmatrix}$$

In the standard (7,4) Hamming code, where the columns of H correspond to the binary representations of bit positions 1 through 7, the syndrome vector **will** exactly indicate the position of the bit in error. In our case here, this is position 7.

In our current setup, this correspondence of calculating the syndrome happens by coincidence rather than by explicit design of H. To reliably use the syndrome for pinpointing the erroneous bit, we would need to modify H to ensure that its columns are set up in a way to not only detect that an error is present, but **where** it is. In doing this, we will have a standard $H(7,4)$ setup where H does not only detect an error but pinpoint its location too.

7.4 Reordering G and H to Match Standard Hamming Syndrome Mapping

While our current generator matrix G and parity-check matrix H are valid and work correctly for detecting errors, they do not make it easy to directly identify which bit is in error from the syndrome. This is because, in H, our columns don't represent bit locations as they did in G. For example, in H the first row is 110, or position 6, not position 1. If you take a closer look at H, the bit positions are all there, they are just all over the place which makes it difficult for us to identify the syndrome as they are not in an intuitive order. In the table below is the bit position that we **should have** as it makes sense, column 1 (bit position 1) is represented as 001, 2 as 010, and so on.

Bit Position	Binary Representation
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Reordering

When exploring H previously we had:

$$H = \left[\begin{array}{cccc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right]$$

The reason we had it mapped this way as it aligned with our explanations of how we could form the parity bits based on a, b, c, d and then detect an error. With our current H, each column of the columns represent the binary participation of each bit in the parity check equations.

Current Column Index	Binary
1	110
2	101
3	011
4	111
5	100
6	010
7	001

To fix this, and have the columns follow ascending bit position order, we can make our new H look like this.:

Reordered H

$$H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

As you can see, the first, second, and third columns are just the columns of G that correspond to the parity bits. If we also look at the rows of our reordered H, the first, second, and third rows correspond to the columns of G. Now that we have rearranged H, of course, we will need to rearrange G as H represents these relationships created by G, in other words, H is derived from G, so if we leave G as is, it's not representing our new H as the parity check equations will be incorrect. Based on this, we are then able to construct our new G.

Original G**:**

$$G = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right]$$

After re-arranging based on H, we are left with our new G:

Reordered G:

$$G = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

This new G still encodes valid codewords, as G can be made based upon your choosing, G is just each bit contribution to the relationship for the parity bits. However, now our new H allows direct syndrome decoding because of our correct bit ordering.

7.5 Using our new matrices

Say we transmitted the following message which was encoded with our new G.

$$e = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Lets say that during transmissing, the fifth bit of our codeword was flipped, lets call this new invalid codeword r:

$$r = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Now following the exact same steps as we did with our previous matrices, we compute the new syndrome:

$$s = Hr^T = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Now lets compute the XOR dot product for each row of H based on r. We will denoted each syndrome bit as s_i:

$$
\begin{aligned}
s_1 &= (0 \cdot 0) \oplus (0 \cdot 1) \oplus (0 \cdot 0) \oplus (1 \cdot 1) \oplus (1 \cdot 1) \oplus (1 \cdot 1) \oplus (1 \cdot 0) \\
&= 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 1
\end{aligned}
$$

$$
\begin{aligned}
s_2 &= (0 \cdot 0) \oplus (1 \cdot 1) \oplus (1 \cdot 0) \oplus (0 \cdot 1) \oplus (0 \cdot 1) \oplus (1 \cdot 1) \oplus (1 \cdot 0) \\
&= 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0
\end{aligned}
$$

$$
\begin{aligned}
s_3 &= (1 \cdot 0) \oplus (0 \cdot 1) \oplus (1 \cdot 0) \oplus (0 \cdot 1) \oplus (1 \cdot 1) \oplus (0 \cdot 1) \oplus (1 \cdot 0) \\
&= 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 = 1
\end{aligned}
$$

And this leaves us with the syndrome:

$$
s = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \Rightarrow \text{Decimal } 5
$$

7.5.1 Why This Works

This works because we aligned the columns of H with the binary representations of the bit positions and then modified G. That means every error produces a syndrome equal to the binary index of the flipped bit. This makes decoding immediate and effective and avoids needing lookup tables or complicated logic like we previously had.

7.6 Generalising Hamming Codes

During our discussions on Hamming, we have looked at the specific Hamming(7,4) code, meaning we have 4 data bits and we are encoding it into a 7-bit codeword by adding three parity bits. We have reconstructed H so that each column is the binary representation of the bit positions in ascending order (instead of all over the place like we previously had). We also explored how from H, we can derive G by transposing the non-identity side of H. However, what if we wanted to work with another hamming code to send more data? like H(15,11) for example?

In general, for a Hamming code of order n, we can construct the parity check matrix H with:

- n rows (one for each parity equation),
- $2^n - 1$ columns (one for each bit position in the codeword),

This means that if H has n rows (H's parity bits) and $2^n - 1$ columns, $2^n - 1 - 1$ bits are encoded. For example, if n is three, we have three parity bits, and 4 encoded bits ($2^3 - 1 - 1 = 4$). A common Hamming codes is $H(256, 247)$ which is where $n = 8$ and therefore $2^8 - 1 - 8 = 247$.

For example, let's create the H(15,11) (which is $H(2^4 - 1, 2^4 - 1 - 4)$) parity-check matrix H. We know that it will have 4 rows (for four equations) as we are creating 4 parity bits. Based on this, we will also have $2^4 - 1$ or 15 columns for bit positions. This will look like:

$$H = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Now that we have H, we know we can derive G to create the encoded messages. This will look like:

$$G = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

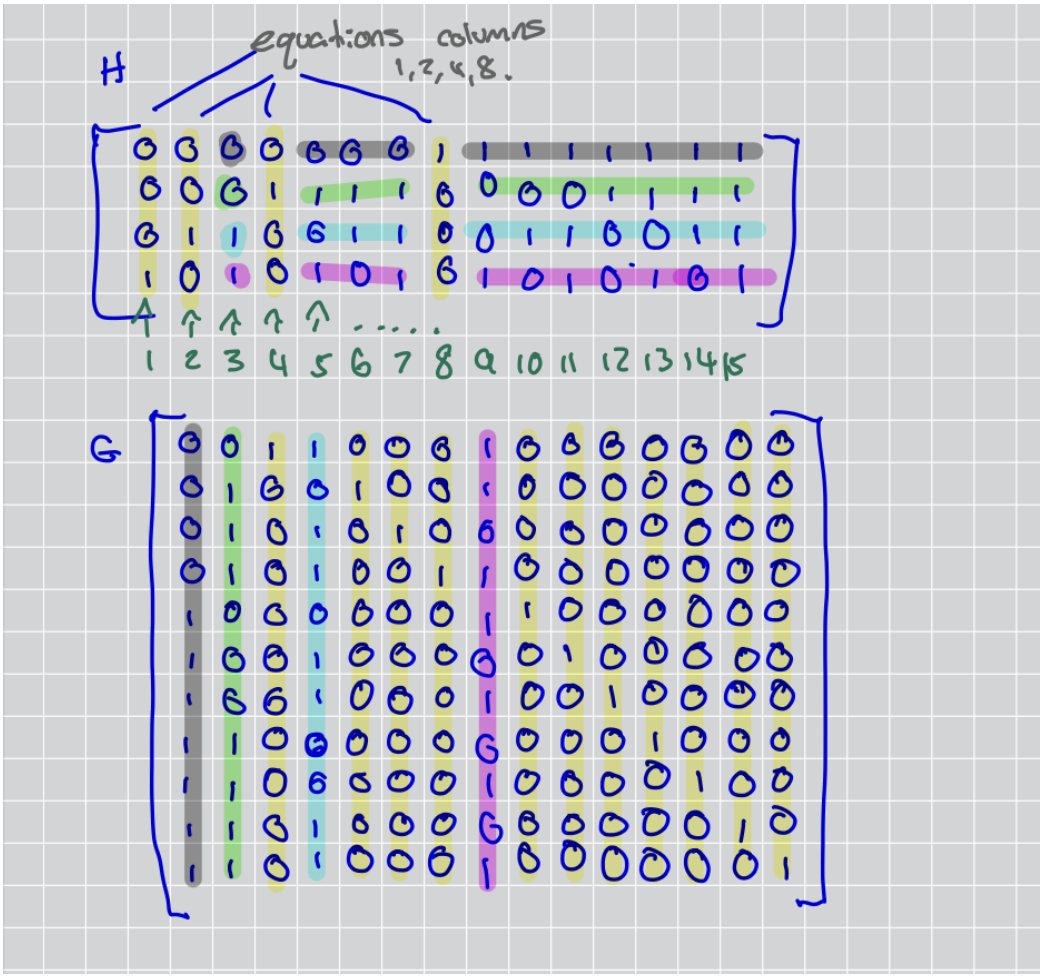


Figure 3: How to derive H from G

Now we have done this, there is one minor issue. There is nothing incorrect in our transformation from G to H however having the least significant bit down the bottom can easily case confusion when reading the syndrome. In our current setup, the column corresponding to 1 is $[1, 0, 0]$ and not $[0, 0, 1]$ which can case confusion when reading the syndrome. For example, if we get the syndrome:

$$s = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

We would have to interpret this as a third bit error (which is right in our case) which gets confusing. So lets re-order H once more with the least significant bit up the top instead, meaning it represent a first bit error:

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

And then following the exact same steps we did previously we get our G once more:

$$G = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Doing this will now avoid all confusion when reading the syndrome. Our previous G and H we technically *not* incorrect but again, this was just for additional clarity.

There is still one issue with this as we have touched on throughout these concepts, we are only able to detect and correct one corrupted bit per encoded message due to how the relationships between each parity bit are created. If we had a system of equations and were trying to see if a, b and c were valid and a and b were corrupted, it would seem as if they are correct, but it's not. Considering this, we are unable to fix **burst errors** meaning multiple corrupted bits in a row. To get around this problem, we use Polynomials to represent the message and then evaluate them at certain points to be constructed. As you may know, you only need a certain amount of points to reconstruct the original polynomial (based on its degree) so we can use this to get around some of our current problems.

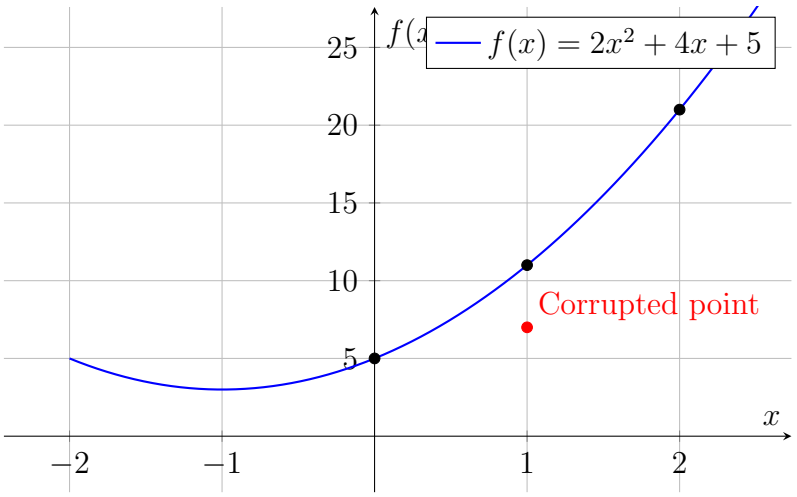
8 Polynomials

In the previous sections, we introduced linear codes that take binary messages and convert them from a 2D message space to a higher-dimensional message space through the use of a generator matrix. Since our valid codewords are "spread apart" with incorrect values in between them, we can correct errors by taking the shortest path from the invalid codeword to the valid codeword assuming bits were corrupted independently (single-bit errors).

Instead of thinking of messages and codewords as vectors, we can think of them as polynomials as we can work with not just one bit, but multiple at any single time [10]. So far, we have been working over $\mathbb{F}_2$, however, it is much more efficient to operate over $\mathbb{F}_k$ as we can represent multiple bits in just one polynomial meaning we can fix burst errors (more than just one error). Before we do this, let's explore the concept over $\mathbb{R}$ first. When using polynomials in error correction which we will look at more in-depth soon, each bit of the message becomes the **coefficient** of the **agreed** degree of polynomial (by receiver and sender). Considering this, if we wanted to send the message $[2, 4, 5]$ 2, 4, and 5 would become the coefficients of our polynomial. Since we want to send three bits of data (which are *symbols* which we will look at later) we need a degree 2 polynomial. It would look like in the form $m(x) = a_0 + a_1x + a_2x^2$:

$$m(x) = 5 + 4x + 2x^2$$

If we plot this function as a smooth curve for intuition we can see (with a red point for the error we have):



When working with $\mathbb{R}$, the curve is smooth and pretty, however, in reality Reed-Solomon codes (which use these polynomials) are done over $\mathbb{F}_k$. This means that the curve will not be smooth as values can only be between 0 and k.

8.0.1 Evaluating and interpolating the polynomials

If we evaluate the polynomial at five different x values, for example, $x = -1, 0, 1, 2, 3$ we obtain 5 y values, meaning we have 5 unique points within our polynomial. As seen above, one of these (for demonstration purposes) is going to be corrupted, Let's say that $f(1)$is received as 3 instead of its correct value 4 due to corruption.

Since we are working with a quadratic polynomial, it's uniquely determined by **3 distinct** points. By teasing every combination of three points out of the five received, we can identify which subset reconstructs the original polynomial. The combinations that do not include the corrupted data ($f(1)$) will result in consistent polynomials, however, any combination of "bad" points will result in an inconsistent polynomial. This comparison between polynomials allows us to pinpoint the incorrect data.

This principle that a polynomial of degree d is uniquely determined by $d+1$ points underlines more advanced error correction such as Reed-Solomon error detections which we will explore shortly. With Reed-Solomon codes, the

only difference to this example is that we are operating over finite fields (like $\mathbb{F}_k$) instead of real numbers.

Despite finite fields being much more difficult to visualize, the underlying concepts behind polynomials remain the same. A step-by-step of this process looks something like this:

1. Encode the messages as polynomials
2. Transmit their values at **selected points**
3. Finally, interpolate the polynomial from a sufficient number of uncorrupted values

8.0.2 Link to Linear Algebra

Under the hood of all of this, it's really just linear algebra that we have covered previously, for example, Let's say the original polynomial in $\mathbb{F}_7$ is:

$$f(x) = a_0 + a_1 x + a_2 x^2$$

You receive the following (possibly corrupted) values:

$$f(0) = y_0$$
$$f(1) = y_1$$
$$f(2) = y_2$$

Which then leads to a systems of equations.

$$\begin{bmatrix} 0^1 & 0^2 & 0^3 \\ 1^0 & 1^1 & 1^2 \\ 2^0 & 2^1 & 2^1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_0 + a_1 \times 0 + a_2 \times 0^2 = y_0 \\ a_0 + a_1 \times 1 + a_2 \times 1^2 = y_1 \\ a_0 + a_1 \times 2 + a_2 \times 2^2 = y_2 \end{bmatrix}$$

Solving this linear system in the field $\mathbb{F}_7$ gives you the coefficients of the original polynomial and therefore, also the original message.

If any of the y_i values are corrupted, the decoder can try all combinations of triplets and check for consistency as discussed in the previous section about polynomials. More advanced decoding techniques, make use of this to efficiently identify and correct errors.

9 Reed-Solomon Codes

Now that we have seen how these polynomials can be used, let's look at a real-world method of ECC called Reed-Solomon codes. This code allows us to use the polynomials to effectively transmit data as symbols rather than just individual bits. Reed-Solomon codes are widely used in CDs, DVDs, and QR codes alongside data transmission systems due to their ability to detect and correct burst errors (consecutive errors).

Let's say that we are working over $\mathbb{F}_7$, even though most codes use $\mathbb{F}_{256}$, $\mathbb{F}_7$ is a good starting point to build understanding.

For example: let's that that we wanted to send the message $m = [3, 1, 6]$. We treat these as **the coefficients of a degree two polynomial** as we did previously:

$$f(x) = 3 + 1x + 6x^2$$

Now, assume that this data has been sent over a noisy network, as we (the receiver) want to decode the message and identify errors (if any). Since we are working with a degree 2 polynomial, we need $k+1$, which in our case is 3 points to define it. To allow for error detection, we will evaluate it as 5 distinct x values: $x = 0, 1, 2, 3, 4$. Remember that this is in $\mathbb{F}_7$ meaning that behind the scenes, we are taking the answer mod 7.

If we compute this, we are left with:

$$\begin{aligned}
f(0) &= 3 + 1 \cdot 0 + 6 \cdot 0^2 = 3 \\
f(1) &= 3 + 1 \cdot 1 + 6 \cdot 1^2 = 3 + 1 + 6 = 10 \equiv 3 \\
f(2) &= 3 + 1 \cdot 2 + 6 \cdot 4 = 3 + 2 + 24 = 29 \equiv 1 \\
f(3) &= 3 + 1 \cdot 3 + 6 \cdot 9 = 3 + 3 + 54 = 60 \equiv 4 \\
f(4) &= 3 + 1 \cdot 4 + 6 \cdot 16 = 3 + 4 + 96 = 103 \equiv 5
\end{aligned}$$

We now have a set of points which is our encoded message $\{(0, 3), (1, 3), (2, 1), (3, 4), (4, 5)\}$. There are the values that we will be transmitting. For example, lets say that one of these points were corrupted:

$$\{(0, 3), (1, 2), (2, 1), (3, 4), (4, 5)\}$$

Now, by reconstructing the polynomial, the combination including the incorrect point will be inconsistent with those that use three correct points. In the next section, we will explore how to reconstruct the polynomial and what happens if one of the points are incorrect.

9.1 Why Polynomials?

The reason polynomials are so efficient is not because they only let us operate on collections of symbols rather than just a singular symbol (or bit) is because of their algebraic structure for redundancy. Polynomials over $\mathbb{F}_q$ means that a polynomial of degree $k-1$ is uniquely defined by k amount of points [10]. The fundamental theorem of arithmetic states that any **non-zero** polynomial of degree k has **at most** k roots. This means that if we have two quadratics that **share** three points, we can take their difference and it will cross the x-axis three times because we have three points. The only "quadratic" that does this is $f(x) = 0$ or $0x^2 + 0x + 0$, this is the same with polynomials of degree k. For example, there exists exactly one quadratic polynomial going through the points (0,1) and (2,1) which is $x^2 - 2x + 1$. If we then evaluate the polynomial at any $n > k$ we are creating redundancy (parity bits). For example, if we had $k = 3$ it means that a quadratic polynomial $m(x)$ requires three points to reconstruct. This is great as if we were to evaluate it at $n = 5$ points, we can lose up to $t = \frac{n-k}{2}$ points and still recover that original polynomial. So in our case $t = \frac{5-3}{2} = 1$ because we still have 4 points, giving us 4 "correct" combinations of 4 points to reconstruct the same polynomial. Think of this as the multi-dimensional representation looked at previously, if we increase the amount of "incorrect" representations of bits, which in this case are polynomials, we still have such a way to trace it back to the original polynomial.

As we also discussed previously, polynomials operate over symbols rather than bits, think of this as instead of error-correcting just bits, we are able to correct chunks of bits (like bytes for example). Another awesome thing about polynomials is that we can directly configure the redundancy by evaluating a certain n. For example, we could evaluate a polynomial at $n = 5$ or $n = 256$.

Polynomials not only offer burst error handling but are also flexible in terms of programmability. We are able to use decoding algorithms such as Berlekamp-Massey to identify and correct corrupted symbols similar to how we were

doing by hand in "best two of three" examples.

9.2 What exactly are Burst-Errors?

We have touched on them previously, but never fully discussed what these are and how they happen. When we have been working with Hamming codes and have been able to detect one-bit errors, one of the main advantages of Reed-Solomon codes is its ability to correct burst errors. What is a burst error? A bust error is a **sequence** of errors that affects a group of consecutive bits (or symbols) in a data stream. Unlike what we have looked at before, burst errors are **not** random errors that impact an isolated part of a message but they do impact several adjacent symbols at a time.

Think of this as bits or symbols being on a conveyor belt (the data stream) and a hot lamp (some interference) that is turned on and off at random periods. When that hot lamp is turned on, it will not affect one bit on the conveyor belt, but will "burn" (corrupt) many consecutive bits at a time. This is especially problematic as without Hamming Codes, we were using relationships to find out where/if an error has been corrupted but if everything is corrupted including *abcd* (data bits) and *zyx* (parity bits), of course, there is no way to possible detect anything, everything is gone. However, in Reed-Solomon codes, we send points on a polynomial with the goal of the receiver reconstructing it, it doesn't matter what is corrupted, as long as they have enough points to reconstruct the polynomial.

10 Reed-Solomon Polynomial Reconstruction in $\mathbb{F}_7$

Given our original polynomial: $f(x) = 3 + x + 6x^2 \in \mathbb{F}_7[x]$, what if we were on the receiving end? We receive the evaluated points (we already agreed on the polynomial) and know we need three valid points to construct the original polynomial.

Case 1: With Corrupted Point $(1, 2)$

Given evaluation points: $\{(0, 3), (1, 2), (2, 1), (3, 4), (4, 5)\}$

Assume $f(x) = a + bx + cx^2$. We set up the system modulo 7, in doing this, we are assuming that these equations hold. However, if one of the points is corrupted (like it is in our case), one of these polynomials will not hold which is where we identify our inconsistency:

1. $a \equiv 3$
2. $3 + b + c \equiv 2 \Rightarrow b + c \equiv 6$
3. $3 + 2b + 4c \equiv 1 \Rightarrow 2b + 4c \equiv 5$
4. $3 + 3b + 2c \equiv 4 \Rightarrow 3b + 2c \equiv 1$
5. $3 + 4b + 2c \equiv 5 \Rightarrow 4b + 2c \equiv 2$

Here we are assuming that Equation 1 is correct as we are **substituting** $a = 3$ **everywhere**. In our case, the first value may be corrupted meaning that $a \neq 3$.
Solution Attempt: From equation 2: $c \equiv 6 - b$. Substituting into equation 3:

$$\begin{aligned} 2b + 4(6 - b) &\equiv 5 \\ -2b + 24 &\equiv 5 \\ -2b &\equiv -19 \\ 2b &\equiv 2 \\ b \equiv 6 &\Rightarrow c \equiv 0 \end{aligned}$$

So from our attempt, we identify that $b = 6$ and $c = 0$. We can verify this by plugging them into other equations and seeing if it holds.
Verification:

$$\begin{aligned} &\text{Equation 4: } 3(6) + 2(0) = 18 \equiv 4 \quad (\neq 1) \quad \text{(Inconsistent)} \\ &\text{Equation 5: } 4(6) + 2(0) = 24 \equiv 3 \quad (\neq 2) \quad \text{(Inconsistent)} \end{aligned}$$

So we know that one point is wrong here, we can identify the incorrect point by substituting b and c into all of the equations, you will find that none of them hold, meaning that the point $(1, 3)$ is corrupted. When we do this, we will find 4 out of 5 equations to be inconsistent with $b = 6$ and $c = 0$ telling us that **at-least** one point is corrupted. Based on the original polynomial being evaluated at 5 points, we already know the expected output of the polynomial, so we go back to trying our (1,3) point. This is why we don't try random things like (2,5), we already know the expected output and what x values are being evaluated as they are agreed on beforehand.

Case 2: With Correct Point $(1, 3)$

Given evaluation points: $\{(0, 3), (1, 3), (2, 1), (3, 4), (4, 5)\}$
We have our equations (since its $\mathbb{F}_7$ everything is mod 7)

1. $a \equiv 3$
2. $3 + b + c \equiv 3 \Rightarrow b + c \equiv 0$
3. $3 + 2b + 4c \equiv 1 \Rightarrow 2b + 4c \equiv 5$
4. $3 + 3b + 2c \equiv 4 \Rightarrow 3b + 2c \equiv 1$
5. $3 + 4b + 2c \equiv 5 \Rightarrow 4b + 2c \equiv 2$

Solution: From equation 2: $c \equiv -b$. Substituting into equation 3:

$$\begin{aligned} 2b + 4(-b) &\equiv 5 \\ -2b &\equiv 5 \\ 2b &\equiv 2 \\ b \equiv 1 &\Rightarrow c \equiv 6 \end{aligned}$$

Verification:

$$\begin{aligned} &\text{Equation 4: } 3(1) + 2(6) = 15 \equiv 1 \quad \text{(which is correct)} \\ &\text{Equation 5: } 4(1) + 2(6) = 16 \equiv 2 \quad \text{(which is also correct)} \end{aligned}$$

Reconstructed polynomial:

$$\boxed{f(x) = 3 + x + 6x^2}$$

Conclusion

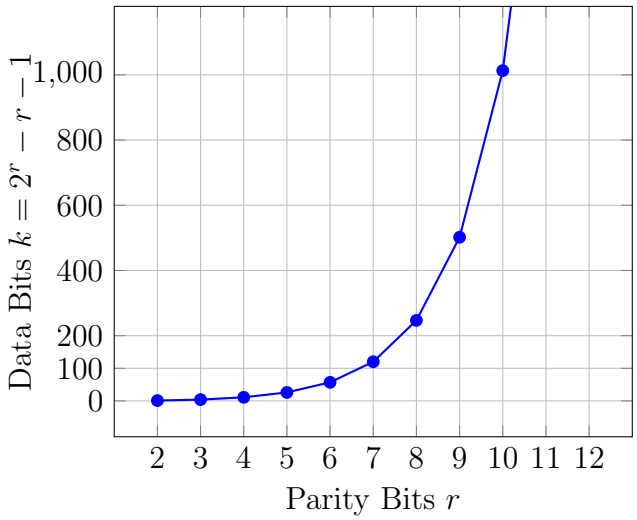
The corrupted point $(1, 2)$ creates an inconsistent system that cannot satisfy all equations simultaneously (it failed 4/5), while the correct value $(1, 3)$ yields a consistent solution across all of the systems.

10.1 Comparison between Reed-Solomon and Hamming Codes

As you know, we can evaluate the polynomial at **as many points as we want** meaning we can add as many bits for redundancy as we want. This

means that the redundancy in the Reed-Solomon codes is configurable to your liking (or demands). When looking at hamming codes, they are shockingly efficient in terms of how much data can be encoded with a small amount of parity bits. See the graph below:

Hamming Code Data Bits vs. Parity Bits



As you can see, this grows linearly meaning as we add more parity bits (see $x = 10, 11 \ldots$ on the graph) the amount we can encode explodes. For example, from 11 parity bits (r) to 12, we go from 2036 max data bits to 4083 max data bits, a massive increase. When we add another parity bit, the amount of data bits is doubled.

This seems awesome, however, Hamming codes still have the same weakness they always have; error correction. Hamming codes are unable to operate over symbols (like bytes) like Reed-Solomon codes meaning they are limited to correcting just one-bit errors. With 12 parity bits, $n = 4083 + 12$, we can only correct **one 1-bit error** anywhere in those 4095 bits. As you can imagine, over a noisy channel this will not suffice.

In environments with **very little noise** where single-bit errors dominate and space is at a premium (such as RAM where burst errors are rare) Hamming codes are great, however over noisy channels like wireless networks and deep space communication where burst errors and **very** common you would want something like Reed-Solomon codes.

Code	Redundancy	Error Types
Hamming (7,4)	75%	Single-bit
Reed-Solomon	Configurable	Burst errors

Table 1: Comparison of Error-Correcting Codes

11 Variations of Reed-Solomon

While we have been looking at the **basic** Reed-Solomon (RS) encoding method which involved expressing a message as the coefficients of a polynomial and then evaluating it at various points (x values) is good for building an understanding. However, multiple real-world applications use a slightly modified variation of this scheme to improve efficiency, resilience (to types of errors) and compatibility.

11.1 Generator Polynomial

As we have looked at Generator matrices in Hamming Codes we know they are a way to create a new codeword, like a constructor in a class in a programming language. When working with polynomials, we can multiply the message polynomial by a known generator polynomial instead of having a traditional "generator" we are used to from Hamming codes.

Rather then directly transmitting the evaluations of the message polynomials (the points) we first **multiply it by a generator polynomial** whom we will call $g(x)$. Similar to our generator matrix, this is agreed upon by both sender and receiver and results in a codeword polynomial $c(x) = m(x)\dot{g}(x)$ where:

- $m(x)$ is the original message polynomial (constructed from the message)
- $g(x)$ is a carefully choses polynomial (like how our G matrix is "carefully" chosen) that determines the error-correcting capabilities of the code.

The key take away here is kind of similar to our parity check matrix. In our H matrix, we were systematically checking if relationships hold, and here we are checking the relationship "is it divisible by $g(x)$". This means that if there were no errors, dividing the encoded polynomial ($c(x)$) by $g(x)$ will

yield the original $m(x)$.

Think of this like simple encryption, we are encrypting out message $m(x)$ with our encryption key $g(x)$, sending it over the network and the decrypting it by dividing by $g(x)$ to see if anyone has tampered with the data during transition (errors). This may seem useless in the beginning as we are just adding another layer of complexity for "no reason" however it actually enables us to do something very interesting which is finding the "closest" polynomial.

11.2 Why is this even important?

Ultimately, the use of $g(x)$ allows us to simplify error detection and correction. If the received polynomial is not divisible by $g(x)$, we know straight away that it must of been corrupted. If it is corrupted, we can then find the "closest" polynomial.

12 Encoding with a Generator Polynomial

For this example, let's use $\mathbb{F}_5$ and represent our message as a polynomial (which we have been doing in basic Reed-Solomon. Lets say the message polynomial (of degree 2) we wanted to send was:

$$m(x) = 4 + 2x + x^2$$

Now this is where it differs a little bit. To introduce redundancy for error detection and correction, the sender and receiver will agree on a generator polynomial $g(x)$ similar to how the generator and parity-check matrix is agreed on before. Let say that:

$$g(x) = 1 + 2x + x^2$$

And then we can simply encode the message by multiplying our message $(m(x))$ by the generator polynomial$(g(x))$:

$$c(x) = m(x) \cdot g(x)$$

12.1 Encoding the Message

Let's compute the encoded polynomial $c(x)$:

$$\begin{aligned} m(x) &= 4 + 2x + x^2 \\ g(x) &= 1 + 2x + x^2 \\ c(x) &= (4 + 2x + x^2)(1 + 2x + x^2) \end{aligned}$$

Multiply terms:

$$\begin{aligned} c(x) &= 4(1 + 2x + x^2) + 2x(1 + 2x + x^2) + x^2(1 + 2x + x^2) \\ &= (4 + 8x + 4x^2) + (2x + 4x^2 + 2x^3) + (x^2 + 2x^3 + x^4) \end{aligned}$$

Now, just combine the like terms, remember we are in $\mathbb{F}_5$ (so mod 5):

$$c(x) = 4 + (8 + 2)x + (4 + 4 + 1)x^2 + (2 + 2)x^3 + x^4$$

$$c(x) = 4 + 10x + 9x^2 + 4x^3 + x^4 \equiv 4 + 0x + 4x^2 + 4x^3 + x^4$$

This means that our encoded message, $c(x)$ is:

$$c(x) = 4 + 4x^2 + 4x^3 + x^4$$

12.2 Transition phase

Now we are ready to transmit our code word. Up until this point, we have just added redundancy to $g(x)$ (the message). The process is same for all codes, we now transmit the encoded message and detect and correct errors on the other side.

Case 1: No Corruption

This one is pretty simple, the receiver gets the exact polynomial $c(x)$.

That means that to decodes the original message, the receiver divides $c(x)$ by the agreed generator polynomial $g(x)$:

$$\frac{c(x)}{g(x)} = m(x)$$

If the division yields a polynomial with no remainder, we have recovered the original message.

Case 2: With Corruption

Suppose a transmission error occurs and for example the receiver gets a polynomial in the form (with $e(x)$ being the error):

$$r(x) = c(x) + e(x)$$

So that means our polynomial is:

$$e(x) = x^2 \quad \Rightarrow \quad r(x) = c(x) + x^2$$

So:

$$r(x) = 4 + 4x^2 + x^2 + 4x^3 + x^4 = 4 + 0x + (4+1)x^2 + 4x^3 + x^4$$

$$r(x) = 4 + 0x + 0x^2 + 4x^3 + x^4$$

Now the receiver tries to divide $r(x)$ by $g(x)$. This time, the result will not divide evenly:

$$\frac{r(x)}{g(x)} \text{ yields a non-zero remainder}$$

Since we get a non-zero remainder, we know that there is some error here, we have detected. Now, to detect this error its a little bit more difficult, we can use something called the Berklamp-Massey algorithm which calculates the "closest" correct polynomial.

12.3 Comparison of the variations

As you may of noticed, in the "original" polynomial method we do not send the **polynomial at all**, we send the points $(f(x))$ on the polynomial which can be used to reconstruct it. The reason for the generator polynomial being so efficient is that it guarantees certain mathematical properties, like all codewords being divisible by $g(x)$ meaning algorithms only have to do **one** check to see if there is an error compared to checking many relationships like in Hamming. Algorithms such as the Berlekamp-Massey are designed to use the structure of encoded messages with a generator polynomial allow us to effectively work with the structure (with computers). Both variants actually have similar constraints, for example the max message size being k (degree of message polynomial) and total size meaning one is not physically better than the other, its the fact that our second variation is more efficient than the first, they both still do the same thing.

13 Berlekamp-Massey Algorithm

As we previously touched on, the Berlekamp-Massey algorithms is a method used to detect and correct errors in Reed-Solomon and other linear black codes. The algorithms main goal is to find the **error location polynomial** based upon a sequence of syndromes which are just values that describe how the received messages is different from the codeword. Syndromes are like clues in a puzzle and we can take those clues and figure out things like where or how many errors occurred without knowing the original message.

13.1 Why do we use this?

The Berlekamp-Massey algorithms allow us to **correct** and detect errors in a efficient way without having to check every possible original message which would take far too long. This algorithm also helps transform a corrupted message into a form in which it can be decoded back int the original even though parts are still wrong.

13.2 How Does It Work? (Simplified Explanation)

In reality, Berlekamp-Massey is quite complex mathematically, so instead, I will take the core features of the algorithm and explain how they work for a high-level understanding of how it operates.

1. **Calculate syndromes:**
   When we receive our message, the decoder calculates a set of syndromes $S_1, S_2, \ldots, S_{2t}$ by evaluating the received polynomial at different x values. This is the exact same thing that we were doing with origional variation. These computed values will give us information about how "far off" the message is from being valid which we can narrow down later on.

2. **Build the error locator polynomial:**
   Our goal is to find some polynomial which we will denote as $L(x)$ (known as the error locator polynomial) whose roots point to the positions of the errors, like our H matrix from Hamming-Codes.

3. **Update the guess:**
   Starting with $L(x) = 1$, the algorithm updates the polynomial step-by-step, kind of like a guessing game. At each step, it checks if the

current $L(x)$ explains all the syndromes. If not, it will modifies $L(x)$ using a discrepancy value (a kind of error indicator) to get "closer" to the original each iteration.

4. **Find error positions:**
   Once $L(x)$ is completed, we find its roots using a technique like the Chien search (which is another algorithm). Each root tells us the position of one error.

5. **Find error values:**
   Now that we know the error positions, we can calculate the actual values to fix with some other algorithm like Forney's algorithm.

13.3 Example (Simplified Concept)

Let's take an example to see how this process works. Let's say that we are using a Reed-Solomon code over the field $\mathbb{F}_5$, and we receive some message that may or may not have one error. The decoder (us) will calculate two syndromes:

$$S_1 = 3, \quad S_2 = 1$$

Since these are not both zero it means that an error has occurred.

Now as discussed in the previous section, we want to find an error location polynomial to find errors in the form:

$$L(x) = 1 + \lambda_1 x$$

The Berlekamp-Massey algorithm attempts to find the smallest degree polynomial $L(x)$ that satisfies a certain recurrence relation involving these syndromes. Let's say that after one step, it may find:

$$L(x) = 1 + 3x$$

This tells us that there is one error, and its position can be determined by finding the inverse root of $L(x)$ in $\mathbb{F}_5$.
Now that we know the position of the error, we can calculate the correct value using some additional algorithms like Forney's algorithms to recover the original message.

14 Shannons Entropy

Shannon's entropy formula is key in information theory and ECCs as it measures the uncertainty or information of a random variable [13]. Shannon's Entropy also explores how much data can be "safely" sent over a noisy channel without risking an unmanageable amount of affected bits (or any data for that matter), such as the channels that NASA uses to communicate with satellites.

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

$H(X)$ is the entropy of some random variable X, $P(x_i)$ is the probability of x_i occurring and log_b is the base in which we are working, since we are working binary (binary entropy) we will be using log_2. In Shannon's formula, x_i represents the n distinct outcomes in the sample space X. For example, when considering a fair 6-sided die, $x_1, x_2, \ldots x_6$ represent the outcomes $1, 2, \ldots, 6$ after each roll.

Shannon's entropy is very important in the field of ECCs as they give a limit on how much you can encode. We simply can't have error-correcting algorithms that repair something beyond that limit. Imagine you received some encoded message in which 90 percent of bits have been affected, it's now simply impossible to distinguish either a correct, or incorrect bit. If your source produces information with entropy $H(X)$ and the channel capacity C, you cannot send data at a rate greater than C, no matter how good your error correction algorithms are.

14.1 Channel Capacity

Let's assume we have some source/message (s) which includes 3 symbols which have the following probabilities of occurring.

$$p(A) = 0.5\%$$
$$p(B) = 0.333\%$$
$$p(C) = 0.167\%$$

If we now compute the the entropy of s we are left with:

$$
\begin{aligned}
H(X) &= -\left[0.5\log_2 0.5 + 0.333\log_2 0.333 + 0.167\log_2 0.167\right] \\
&\approx -\left[0.5(-1) + 0.333(-1.585) + 0.167(-2.585)\right] \\
&\approx 1.46 \text{ bits}
\end{aligned}
$$

This means that on average, each symbol of s contains 1.46 bits of uncertainty. This means that it is the minimum amount of bits we would need per symbol to encode s with no data loss. Since s has four symbols, we would need$\lceil 5.84 \rceil$ (as we cant have 5.84 bits, which is 4×1.46 meaning we need 6 bits to represent our three symbols A, B and C.

Let's assume that the channel we are sending source s over has a capacity c of 1.25 bits per symbol. Since our message entropy $H(X) = 1.46$ and

$$H(X) > C$$

This means that no matter how we encode s or how effective our ECC algorithms are, there is no way we can send data through the channel without data loss or errors. This serves as the upper bound of the information we can send, we want $H(X) > C$ while $H(X)$ is as close to C as possible.

Since the channel is unable to remove enough uncertainty because it is unable to transmit enough uncertainty, in other words, transmit enough information or parity bits, then some of the original message is completely lost or malformed. An initial solution to this would be adding additional redundancy to your data, however in doing this, additional parity bits increase $H(X)$ which further increases its distance from C is the upper bound meaning you actually start to transmit more unsalvageable data. At this stage, it seems that we are stuck and there is simply nothing we can do because $H(X)$ is either too large or C is too small [13].

14.2 A solution to our problem

The only solution to our current issue is by making the channel wider, meaning we can send more information $H(X)$ such that $H(X)$ is less than C. Making the channel wider refers to increase its capacity which can be denoted as:

$$C = \max_{p(x)} I(X;Y)$$

We can increase the bandwidth over an along channel by using Shannons formula:

$$C = B \log_2 \left(1 + \frac{S}{N}\right) \quad \text{(bits/sec)}$$

where B is the bandwidth, S is the signal power, N is the noise power, and N over N is the SNR.

In modern wireless systems such as wifi or recent cellular networks we can transmit data over multiple channels, known as Multi-input multi-output, or MIMO.

$$C_{\text{total}} = \sum_{i=1}^{n} C_i$$

By transmitting multiple data streams in a parallel fashion, we are able to increase the total capacity by simply adding more, even though a singular channel may have a low capacity C, we are able to duplicate channels [13].

14.3 Connections to Other Mathematical Areas

After exploring Error Correcting codes, it's obvious that they have a rich interplay between other mathematical concepts to ensure reliable data transmission. Below are some of the key concepts which are covered in the report, however, there are some other links to mathematic concepts such as probability, coding theory, computational complexity and combinatorics.

14.3.1 Finite Fields

Through codes like Hamming codes, we operate over $\mathbb{F}_2$ where XOR and AND logic aligns with modular arithmetic. Regarding more involved codes such as Reed-Solomon codes where we operate over multi-bit symbols ($\mathbb{F}_q$ allow using to use polynomial algebra. Finite fields allow use to correct single bit errors, which also correcting burst errors.

14.3.2 Information Theory

Regarding Shannons theorem, we can establish limits on error correcting abilities (based on channel capacity) to determine noise resilience and how much redundancy we will need with each message. Shannon also provides

ways to improve channel capacities to maximise the efficiency in which we can send data.

14.3.3 Linear Algebra

using linear codes like Hamming and Reed-Solomon are defined as a subspace over $\mathbb{F}_q$, we use generator matrices to encode data into codewords through matrix multiplication which gives us a way to systematically add redundancy to messages and therefore have a way to identify and correct these errors. To detect and correct errors, we use a parity check matrix which is like a map allowing us to track where and if a bit is incorrect.

15 Practical Explanations

15.1 QR Codes

Compared to what was discussed previously, CDs use an extension of Reed-Solomon Codes called Cross-Interleaved Reed-Solomon Coding (CIRC). This specific method combines Reed-Solomon codes with something called interleaving to detect and correct errors caused by physical damage such as scratches on a CD [12]. We know that burst errors are much harder to detect and fix as there are multiple corrupted bits that damage the systems of equations or bit relations in the parity-check matrix. If too many bits are corrupted, we cannot use the relationship between the simultaneous equations (or XOR operations) to detect and locate the corrupted bit. Interleaving is the process of distributing these corrupted bits in a way so they can easily be identified, like converting a burst error into a single-bit error in a specific subsection of an encoded message. For example, if some CD c consisted of 5 bytes of data, the first 4 being unaffected and the last fourth being completely corrupted. If we distribute this last fourth into the third so we have only one error in each byte, we significantly increase the chance of detection and therefore correct the corrupted data in c.

15.1.1 The encoding process of CDS

Considering this, the process of encoding, reading, detecting, and correcting CD errors is pretty simple. The encoding process is split into 2 distinctions sections which we will call $C1$ and $C2$ [12].

C1 Encoder

Firstly, the raw audio/data is split into distinct frames of 24 bytes each. Using Reed-Solomon codes, 4 additional parity bits are added to each frame meaning that each frame will be 28 bytes in size considering our new redundancy bits. This means that we can correct up to 2 errors per frame during the decoding process alone.

Interleaving

As discussed previously, we may have some burst errors presented in the form of physical damage. This process spreads out the burst errors across different codewords (encoded frames) to convert them into many scattered single byte errors.

C2 Encoder

Now, we have sections of interleaved data. CDs then add another 4 parity bytes to create 32 byte frames. This means we can now correct up to two errors per frame during **decoding** (not encoding)

15.2 Deep-Space Communication

Regarding Shannon's entropy and capacity formula, the voyager had a very low $\frac{S}{N}$ or signal over noise pointer. Since data had to travel $\approx$ 20 billion km, and weak transmitters of 23W, this constrained channel capacity heavily, meaning that highly efficient error correction was demanded. The voyager used RS(225,223) codes [11] which is very powerful in terms of error correcting compared to the RS(7,4) we have been exploring, however, still follows the same fundamental concept. The RS(225,223) code uses **32 parity symbols** which equates to around 14 percent redundancy across 223 data symbols (polynomials). As this redundancy lowers the effective entropy of transmitted data while still being under the channel's limited capacity, it was a huge step in terms of data transmission.

The RS code rate $R = \frac{k}{n} = \frac{223}{225} = \approx 0.875$. The correction capability of RS(225,233) is around 16 symbols per block which is effective considering the channel capacity constraint. Since space channels such as the ones the Voyager communicated from suffer from large amounts of burst errors from things like cosmic radiation [11]. RS codes corrected **128-bit bursts** (or 16 symbols $\times$ 8 bits), matching Shannon's emphasis on tailoring codes to the specific channel and its constraints. Through Shannon's entropy and Reed-Solomon codes, Voyager missions were able to navigate harsh communication conditions balancing redundancy and reliability.

16 Conclusions

16.1 Summary

Having explored the most simple repetition codes to configurable Reed-Solomon code algorithms, ECCs, hold a powerful place in mathematics and digital communication through WIFI, CD's QR codes, and even space missions. As new innovations surrounding tech are starting to explode, ECCs remain critical as they allow us to communicate effectively, and also safely. In the future, work may focus on near-capacity codes and integration into emerging fields such as quantum computing.

References

[1] Hamming, R.W. (1950). *Error Detecting and Error Correcting Codes*.

[2] Reed, I.S.; Solomon, G. (1960). *Polynomial Codes over Certain Finite Fields*.

[3] Wikipedia contributors. (2024, April 4). *Repetition code*. Retrieved from `https://en.wikipedia.org/wiki/Repetition_code`

[4] Wright, G. (2022, July 11). *Hamming code*. Retrieved from `https://www.techtarget.com/whatis/definition/Hamming-code#:~:text=Hamming%20code%20is%20an%20error,correction%20code%20(ECC)%20RAM.`

[5] Unknown author. *Generator and parity check matrices — Coding Theory Class notes — Fiveable*. Retrieved from https://library.fiveable.me/coding-theory/unit-3/generator-parity-check-matrices/study-guide/ksLqWc4jgK20edck

[6] Wikipedia contributors. (2024, November 6). *Parity-check matrix*. Retrieved from https://en.wikipedia.org/wiki/Parity-check_matrix

[7] Markowsky, G. (2025, April 14). *Information theory — Definition, History, Examples, & Facts*. Retrieved from https://www.britannica.com/science/information-theory/Discrete-noisy-communication-and-the-problem-of-error

[8] Wolfram Research, Inc. *Finite Field – from Wolfram MathWorld*. Retrieved from https://mathworld.wolfram.com/FiniteField.html

[9] GeeksforGeeks. (2024, August 2). *Galois Fields and its properties*. Retrieved from https://www.geeksforgeeks.org/galois-fields-and-its-properties/

[10] I2i Communications Ltd. *Reed-Solomon codes*. Retrieved from https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html

[11] Destevez, D. (2021, December 11). *Voyager 1 and Reed-Solomon – Daniel Estévez*. Retrieved from https://destevez.net/2021/12/voyager-1-and-reed-solomon/

[12] Hc. (2023, June 9). *The Audio CD flaw — Headphone commute*. Retrieved from https://headphonecommute.com/2023/03/27/the-crucial-flaw-with-audio-cds/

[13] Elsevier. (2023). *Shannon Entropy*. ScienceDirect Topics. Retrieved from https://www.sciencedirect.com/topics/engineering/shannon-entropy